

Creating Scientific Graphics in C

SVG Programming in C

SVG Programming

An algorithm produces an image. The program describes the geometry, while SVGC generates human-readable SVG. All examples are written in C and require no graphics editor.

Preface

The SVGC library was developed as a practical tool for producing scientific and engineering illustrations in C. It enables programs that model the behavior of real-world systems to produce human-readable SVG documents as they run. It can also be used to write specialized drawing programs whose geometry requires substantial preliminary computation. In either case, the application computes the geometry and generates a reproducible, human-readable SVG document. This is the approach used to create illustrations for <https://dynamics.su/>.

The central idea of this book is simple: *an algorithm produces an image*. Numerical data, geometric rules, and coordinate transformations belong to the application algorithm, while SVGC provides the objects and operations needed to describe graphics and write an SVG document. Data therefore need not be forced into the coordinate system of a particular graphics editor.

The book progresses from simple concepts to more complex ones through practical examples. It begins with geometric primitives, followed by data-driven coordinate axes, polylines, polygons, arrows, and flowcharts. It then covers line styles, SVG path commands, Bézier curves, elliptical arcs, and compound geometric shapes. Subsequent topics include text, gradients, engineering diagrams, splines, and the investigation of the dynamic properties of numerical methods for ordinary differential equations. The final chapters explain how to prepare vector PDFs for printing and describe the public SVGC API.

The book is intended for researchers, engineers, and students in technical disciplines.

At first glance, producing illustrations with SVGC may seem to require specialized programming expertise. Modern artificial intelligence systems, however, can make the task substantially easier. The library source code and this book can be supplied to a neural network, along with a description of the desired illustration in ordinary language. In many cases, a useful result can be produced in minutes. The geometry of the drawing and the correctness of the generated program must, of course, still be verified.

The book includes numerous C examples and simple Makefiles. Each example also includes a `view.html` page for examining its output in a browser.

To prepare the printed figures, the C programs produce SVG files that CairoSVG converts directly to vector PDFs. These are inserted into the book pages with `\includegraphics`. This process preserves the geometry and, when the required fonts are available, the appearance of text labels and mathematical notation.

Contents

Contents	ii
1 Building and Installing SVGC	1
1.1 Purpose of the Library	1
1.2 Source Package Layout	1
1.3 Preparing and Configuring the Build	1
1.4 Building and Testing	1
1.5 Installation	2
1.6 Building Your Own Program	2
2 SVG Documents, Sheets, and Scenes	4
2.1 SVG as Program Output	4
2.2 Screen Coordinates	4
2.3 Sheet and Scene	5
2.4 The First C Program	5
2.5 Building and Viewing the Example	7
2.6 What to Remember When Reading SVG	7
3 Basic Geometric Primitives	8
3.1 From Individual Lines to a Composition	8
3.2 Rectangles and Rounded Corners	8
3.3 Circles and Ellipses	8
3.4 Lines, Fills, and Strokes	8
3.5 Example 2	9
3.6 Arcs, Chords, and Arrows: What Comes Next	10
4 Coordinate Axes Derived from Data	11
4.1 Why Axes Should Not Coincide with the Frame	11
4.2 Bounds of the Input Data	11
4.3 Affine Coordinate Transformation	11
4.4 An Example with Discrete Values	11
5 Polylines, Polygons, and Discrete Geometry	15
5.1 From Segments to Arrays of Vertices	15
5.2 Vertex Order and Closure	15
5.3 The Public SVGC Interface	15
5.4 Drawing Discrete Measurements	15
5.5 Example 4	16
5.6 The Boundary Between Visualization and Computation	18
6 Arrows and Arrowheads	19
6.1 Why an Arrow Is a Compound Object	19
6.2 The Unit Direction Vector	19
6.3 Start and End Point in Opposite Directions	19
6.4 Example 5	19
6.5 Scale, Styles, and Painting Order	24
7 Generating Flowcharts Programmatically	25
7.1 A Graph of Control Transfers	25
7.2 Example: A Computational Loop	25

7.3	Example 6	25
7.4	Labels Inside Blocks: Why Centering Is Manual	28
7.5	SVG Text and XML Safety	29
7.6	Automatic Layout Is a Separate Problem	29
8	Object Appearance: Strokes, Fills, and Dashes	30
8.1	Geometry and Style Are Separate	30
8.2	Stroke and Fill Colors	30
8.3	Line Caps and Segment Joins	30
8.4	Dashes and Visual Rhythm	30
8.5	Example 7	30
9	Compound SVG Outlines: Path Commands	33
9.1	A Path as a Sequence of Geometric Operations	33
9.2	Moves, Lines, and Closure	33
9.3	Absolute and Relative Coordinates	33
9.4	Outlines, Fills, and Traversal Order	33
9.5	Example 8	33
10	Quadratic and Cubic Bézier Curves	37
10.1	From Polylines to Smooth Curves	37
10.2	The Quadratic Curve	37
10.3	The Cubic Curve	37
10.4	The Geometry of Control Points	37
10.5	Smoothing Discrete Data Is Not Guesswork	38
10.6	Example 9	38
11	Elliptical Arcs, Chords, and Sectors	42
11.1	Elliptical Arcs as Path Elements	42
11.2	Arc Selection Flags	42
11.3	Arc, Chord, and Sector	42
11.4	Example 10	43
12	Labels, Fonts, and Text Coordinates	46
12.1	Text Is a Graphical Object	46
12.2	Font Family, Size, and Color	46
12.3	Why Character Counts Cannot Determine Text Width	46
12.4	Example 11	46
12.5	Escaping Text in XML	48
12.6	External Fonts and CSS Definitions Inside SVG	48
12.7	A Shared Font Directory for Examples	49
12.8	Subscripts and Superscripts with tspan	49
12.9	Font Sizes and Families When Bringing SVG into TeX	50
13	Linear and Radial Gradients	51
13.1	Definitions and References	51
13.2	Linear Color Interpolation	51
13.3	Radial Gradients	51
13.4	Example 12	52
13.5	Verifying the Print Conversion	53
14	A Structural Diagram of a Gyroscopic System	55
14.1	A Graphical Representation of a Mathematical Model	55
14.2	Differential Model of the Angular Stabilization System	55
14.3	Blocks, Connections, and Transfer Factors	55

14.4	Summing Junctions, Branches, and Painting Order	55
14.5	Mathematical Labels and Nested tspan Elements	56
14.6	Adding CSS with SVGC	56
14.7	Example 13	56
15 A Diagram of the Classical Runge–Kutta Method		67
15.1	From Equations to Computational Structure	67
15.2	Figure Geometry	67
15.3	Example 14	68
16 A Mechanical Oscillator: Geometry of Forces and Connections		72
16.1	An Author's Illustration as a Program	72
16.2	Coordinate System and Geometric Parameters	72
16.3	A Local Basis and Decomposition of the Vertical Force	72
16.4	Arrow Directions and Rotation Rules	73
16.5	Arcs and Partial Circles	74
16.6	Labels with Mathematical Subscripts	74
16.7	Example 15	74
16.8	Next: Coordinate Axes and Computational Results	81
17 Coordinate Axes, Scales, and Tick Labels		83
17.1	Two Independent Coordinate Systems	83
17.2	Data Ranges and Protective Margins	83
17.3	Choosing a Grid Step	83
17.4	Painting Order	84
17.5	Example 16	84
17.6	Limitations of Simple Automatic Labeling	89
18 Discrete Observations and Model Comparison		90
18.1	Observations Are Not a Continuous Function	90
18.2	Two Views of the Same Data	90
18.3	Measuring Deviations from the Model	90
18.4	Example 17	91
18.5	Trustworthy Visual Comparisons	94
19 Natural Cubic Interpolating Splines		95
19.1	Why Smoothness Conditions Matter	95
19.2	Choosing Boundary Conditions	95
19.3	The Tridiagonal System	95
19.4	Evaluating the Spline	96
19.5	Programmatic Verification	96
19.6	Example 18	97
19.7	Limits of Interpolation	102
20 Phase Trajectories and Time Series		103
20.1	One Model, Two Kinds of Scientific Plot	103
20.2	The Runge–Kutta Numerical Scheme	103
20.3	Independent Analytical Verification	103
20.4	From a Time Series to the Phase Plane	104
20.5	Example 19	104
20.6	What the Line Actually Represents	109
21 Contour Lines of Functions of Two Variables		110
21.1	From a Scalar Field to a Level Set	110
21.2	A Rectangular Computational Grid	110

21.3	The Marching Squares Algorithm	110
21.4	Ambiguous Cells	110
21.5	Contours as Computed Results	111
21.6	Example 20	111
21.7	Smoothing and Geometric Accuracy	115
22	Dynamic Properties of the Gauss–Legendre Method	116
22.1	From a Numerical Method to Its Stability Function	116
22.2	Geometry of Zeros and Poles	116
22.3	The Logarithmic Dynamic Characteristic	116
22.4	The One-Percent Region of Dynamic Consistency	117
22.5	Regions Localizing Zeros and Poles	117
22.6	Marching Squares for Stability-Function Contours	118
22.7	Figure 22.1 and the Complete C Example	118
23	Converting SVG to PDF and Including Figures in TeX	121
23.1	Converting SVG with CairoSVG	121
23.2	Including PDF in a TeX Document	121
24	Reference to the Public SVGC API	122
24.1	Scope of This Reference	122
24.2	Sheets, Scenes, and Resource Ownership	122
24.3	Geometric Primitives	122
24.4	Vertex Arrays and Closed Outlines	123
24.5	Paths and Geometric Commands	123
24.6	Strokes, Fills, and Text	124
24.7	Images and External References	125
24.8	Gradients and Color Stops	125
24.9	Sheet Definitions, CSS, and JavaScript	126
24.10	Identifiers, References, and Output	127
24.11	Global Styles and Their Limitations	127
Conclusion		128
A	SVG Path Commands and Their SVGC API Equivalents	129
A.1	Path Syntax Rules	129
A.2	Main Commands and Argument Counts	129
A.3	Bézier Curve Equations	129
A.4	Arguments of an Elliptical Arc	130
A.5	C Construction Functions	130
B	Coordinate Transformation and Scaling Formulas	132
B.1	The Screen Rectangle and Data Domain	132
B.2	The Inverse Transformation	132
B.3	Vectors, Lengths, and Unequal Scales	132
B.4	Selecting Bounds and Protective Margins	133
B.5	Tick Labels and the 1–2–5 Family	133
B.6	An Independent Verification Example	133
C	Glossary of Technical Terms	134
C.1	How to Use This Glossary	134
C.2	Terms That Must Not Be Confused	135
D	Index of C Programs, Chapters, and Figures	136
D.1	Numbering Conventions	136

D.2	Programs and Illustrations	136
D.3	Finding an Example by Task	136

1. Building and Installing SVGC

1.1 PURPOSE OF THE LIBRARY

SVGC (*SVG C Library*) provides C programs with facilities for describing graphical objects and writing **SVG** (*Scalable Vector Graphics*) documents. The output is an ordinary XML text file. It can be inspected without a specialized editor, compared using version-control tools, and opened in a browser. Because the program source defines the drawing's geometry directly, its output is particularly suitable for reproducible numerical experiments.

The library's principal responsibility is to record geometry and styling. Determining numerical data bounds, calculating coordinate transformations, performing interpolation, and finding function zeros remain tasks for the application itself.

1.2 SOURCE PACKAGE LAYOUT

The SVGC source tree contains the public header `src/svgc.h`, the library implementation `src/svgc.c`, and the doubly linked-list support files `src/svgc-dlist.h` and `src/svgc-dlist.c`. At the package root are `configure.ac`, `acsite.m4`, `svgc.m4`, `Makefile.am`, and the build-support files `svgc-config` and `svgc.pc`. The `examples/` directory contains standalone usage examples. GNU Autoconf, Automake, and Libtool are used for building the package.

Building a prepared source distribution requires a C compiler, `make`, a command shell, and standard build tools. Running Autoconf and Automake again is unnecessary: the required scripts are already included.

1.3 PREPARING AND CONFIGURING THE BUILD

An out-of-tree build is recommended. For example, after obtaining the library source archive, run:

```
tar xJvf svgc-1.0.10.tar.xz
mkdir build-svgc
cd build-svgc
../svgc-1.0.10/configure \
  --prefix=/usr \
  --libdir=/usr/lib64 \
  --enable-static=no
```

The `build-svgc/` directory is created alongside the extracted SVGC source directory. This keeps source files separate from object files and other build products. The `--prefix=/usr` option selects the installation prefix, while `--enable-static=no` disables the static library. The `/usr/lib64` directory is appropriate for systems that follow that convention; other systems may require `/usr/lib` or a different `--libdir` value.

1.4 BUILDING AND TESTING

After configuration succeeds, build the library with:

```
make
```

Optionally run the tests supplied with the source package:

```
make check
```

Both commands run inside `build-svgc/`. Developers can also request stricter compiler diagnostics by passing options such as `-O2 -Wall -Wextra -Werror -std=gnu23` through `CFLAGS` during configuration. This is not required for a first build.

1.5 INSTALLATION

Install the compiled library with:

```
make install
```

If the chosen system directories require administrative privileges, use `sudo make install` or the platform's customary privileged procedure. Package maintainers can use `DESTDIR` to install into a temporary staging tree without changing the running system. That is a packaging operation rather than a required part of normal installation.

1.6 BUILDING YOUR OWN PROGRAM

Once SVGC is installed, a C program can include `svgc.h` and link against the library. The following Makefile is sufficient for a simple `main.c` program (recipe lines begin with a tab character):

```
CC      = gcc
CFLAGS  = -Wall -g `svgc-config --cflags`
DEFS    = -D__DEBUG__

LDLFLAGS = `svgc-config --ldflags`
LIBS     = `svgc-config --libs` -lm

all: main

SOURCES = main.c
OBJECTS = main.o

.SUFFIXES:
.SUFFIXES: .o .c
%.o : %.c
    $(CC) -g $(CFLAGS) $(DEFS) -c $< -o $@

main: $(OBJECTS)
    $(CC) $(LDLFLAGS) -o main $(OBJECTS) $(LIBS)

clean:
    rm -f *.o main
```

The `make` command produces the main executable, and `make clean` removes build products. The `svgc-config --cflags` command provides compiler options; `--ldflags` provides the library search path for linking; and `--libs` supplies the library name. The extra `-lm` option links the mathematics library. Header and library paths therefore need not be hard-coded in the Makefile.

For developers using Autotools. With the installation prefix `/usr`, SVGC installs `svgc.m4` into `/usr/share/aclocal/`. This file defines the Autoconf macro `AC_CHECK_SVGC`, which can check the library's version, locate its headers, test linking, and optionally execute a test program. After configuring the compiler in your own `configure.ac`, use:

```

dn1 =====
dn1 Check for SVG C Library
dn1 =====
AC_CHECK_SVGC(
  [0],
  [yes],
  [yes],
  [yes],
  [
    AC_DEFINE(
      [HAVE_SVGC],
      [1],
      [Define to 1 if SVGC is available.]
    )
    SVGC_VERSION=`$SVGC_CONFIG --version 2>/dev/null`
    AC_SUBST([SVGC_VERSION])
  ],
  [
    AC_MSG_ERROR([SVGC is required.])
  ]
)

```

The arguments to `AC_CHECK_SVGC` are defined in `svgc.m4`: the minimum version, defaults for `--with-svgc` and `--with-svgc-test`, a switch governing build-flag extension, and the actions for success and failure. The final two arguments must not be placed in positions three and four, which are reserved for check configuration. Upon successful detection the macro also provides `SVGC_CFLAGS`, `SVGC_LDFLAGS`, and `SVGC_LIBS`, which can be used in `Makefile.am`. The `aclocal` utility processes `svgc.m4` when preparing `configure`; users building a released SVGC source package need not run these tools.

2. SVG Documents, Sheets, and Scenes

2.1 SVG AS PROGRAM OUTPUT

SVG is an XML document describing geometric objects, their coordinates, and their appearance. Even a large drawing remains readable text: individual lines, rectangles, circles, and labels can be located and examined independently of any viewer. This human-readable representation is particularly useful when debugging drawing algorithms.

Three distinct stages should be kept in mind:

1. the application computes object coordinates, dimensions, and relationships;
2. SVGC turns these values into SVG elements;
3. a viewer renders the SVG on screen or on a printed page.

2.2 SCREEN COORDINATES

In an SVG document without additional transformations, the origin is at the upper-left corner. The x coordinate increases to the right, and y increases downward. This is a convenient convention for screen images. These coordinates need not represent centimeters or mathematical variables: they belong to the chosen SVG scene and its representation.

In the first illustration, the sheet is 640 units wide and 400 units high. The program explicitly draws two segments at $(80, 80)$ to serve as visible axes. The true SVG origin, however, remains at $(0, 0)$.

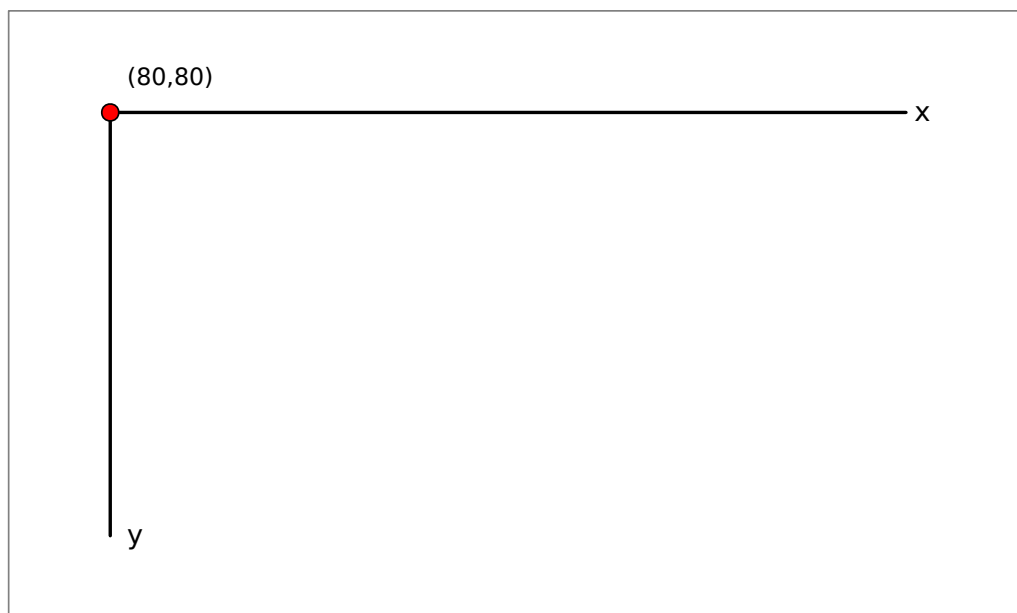


Figure 2.1. The first SVGC sheet: screen coordinates and a selected point within the scene.

2.3 SHEET AND SCENE

SVGC provides `__svg_new_sheet()` to create a sheet, `__svg_new_scene()` to create a scene, and `__svg_scene_add_node()` to append graphical nodes. The sheet establishes the canvas geometry, the scene determines drawing order, and a node represents an element such as a line or circle. A program creates nodes, sets their attributes, and eventually passes the scene to `__svg_paint_scene()`.

Insertion order matters: SVG elements are painted sequentially. The example therefore adds its white background rectangle first, followed by the frame, axes, and labels.

2.4 THE FIRST C PROGRAM

The complete first example is in `svgc-examples/001-first-sheet/`. It deliberately avoids any additional graphics framework so that every SVGC call is visible. The program checks creation of the scene and every node, and frees the complete scene after writing the SVG file.

```

/*****
MAIN.C

    First sheet, scene, and screen coordinates.

    The coordinate origin (0,0) is at the upper-left corner.

*****/

#include <stdio.h>

#include <svgc.h>

static int
add_node( struct __svg_scene *scene, struct __svg_node *node )
{
    if( !node ) return( 0 );

    if( __svg_scene_add_node( scene, node ) != 1 )
    {
        __svg_free_node( node );
        return( 0 );
    }

    return( 1 );
}

/* End of add_node() */

int main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    struct __svg_node *node;
    int result = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 640.0, 400.0,
                                                    20.0, 20.0,
                                                    20.0, 20.0 );

    if( !sheet )
    {
        fprintf( stderr, "Cannot create SVG sheet\n" );
        return( 1 );
    }

```

```

}

scene = __svg_new_scene( sheet );
if( !scene )
{
    __svg_free_node( (struct __svg_node *)sheet );
    fprintf( stderr, "Cannot create SVG scene\n" );
    return( 1 );
}

/* A white background makes the paper explicit in the SVG. */
node = __svg_new_rect( 0.0, 0.0, 640.0, 400.0, 0.0, 0.0 );
if( !node ) goto cleanup;
(void)__svg_set_stroke( node, "none" );
(void)__svg_set_fill( node, "white" );
if( !add_node( scene, node ) ) goto cleanup;

/* Draw the visible boundary of our working area. */
node = __svg_new_rect( 20.0, 20.0, 600.0, 360.0, 0.0, 0.0 );
if( !node ) goto cleanup;
(void)__svg_set_stroke( node, "gray" );
(void)__svg_set_fill( node, "none" );
if( !add_node( scene, node ) ) goto cleanup;

/* A horizontal axis: x grows from left to right. */
node = __svg_new_line( 80.0, 80.0, 550.0, 80.0 );
if( !node ) goto cleanup;
(void)__svg_set_stroke( node, "black" );
(void)__svg_set_stroke_width( node, 2.0 );
if( !add_node( scene, node ) ) goto cleanup;

/* A vertical axis: SVG y grows from top to bottom. */
node = __svg_new_line( 80.0, 80.0, 80.0, 330.0 );
if( !node ) goto cleanup;
(void)__svg_set_stroke( node, "black" );
(void)__svg_set_stroke_width( node, 2.0 );
if( !add_node( scene, node ) ) goto cleanup;

/* Mark our chosen coordinate reference point. */
node = __svg_new_circle( 80.0, 80.0, 5.0 );
if( !node ) goto cleanup;
(void)__svg_set_stroke( node, "black" );
(void)__svg_set_fill( node, "red" );
if( !add_node( scene, node ) ) goto cleanup;

node = __svg_new_text( 90.0, 64.0, "(80,80)", "DejaVu Sans", 14 );
if( !add_node( scene, node ) ) goto cleanup;

node = __svg_new_text( 555.0, 85.0, "x", "DejaVu Sans", 16 );
if( !add_node( scene, node ) ) goto cleanup;

node = __svg_new_text( 90.0, 335.0, "y", "DejaVu Sans", 16 );
if( !add_node( scene, node ) ) goto cleanup;

__svg_paint_scene( scene, 0, "first-sheet.svg" );
result = 0;

cleanup:
if( result ) fprintf( stderr, "Cannot complete SVG scene\n" );
__svg_free_scene( scene );

return( result );
} /* End of main() */

```

2.5 BUILDING AND VIEWING THE EXAMPLE

The programs discussed in this book are also distributed in a separate `svgc-examples.tar.xz` archive. Each example has its own directory with a C source file, a `Makefile`, a generated SVG document, and an HTML page for viewing it. After extraction, the first example can be found in `svgc-examples/001-first-sheet/`.

Build it using GNU Make:

```
tar xJvf svgc-examples.tar.xz
cd svgc-examples/001-first-sheet/
make
```

The `Makefile` describes compilation and linking against the installed SVGC library. A successful build runs the `first-sheet` program, producing the SVG document `first-sheet.svg`.

The file can be opened in a browser using `view.html`. Because SVG is plain XML text, its contents can also be inspected in an ordinary text editor.

Creating a graphical object and choosing its coordinates are separate operations. A graphics library should not decide where the mathematical origin of an application lies.

2.6 WHAT TO REMEMBER WHEN READING SVG

Do not make an algorithm depend on automatically generated element IDs. An XML node's identifier may change when the program is modified, while its geometric meaning is expressed by numerical coordinates and style attributes. Examine not only the rendered image but also attributes such as `stroke`, `fill`, `x1`, `y1`, `cx`, and `cy`.

3. Basic Geometric Primitives

3.1 FROM INDIVIDUAL LINES TO A COMPOSITION

A drawing can be assembled from a small set of familiar primitives: segments, rectangles, circles, and ellipses. This approach suits educational diagrams, block diagrams, plots, and illustrations of numerical algorithms. Every object is defined separately, and the scene controls the order in which objects are placed.

3.2 RECTANGLES AND ROUNDED CORNERS

SVGC creates rectangles with `__svg_new_rect(x, y, w, h, rx, ry)`. The first two arguments specify the upper-left corner; w and h give the width and height. Parameters r_x and r_y describe corner rounding. Setting both to zero produces an ordinary rectangle with square corners.

3.3 CIRCLES AND ELLIPSES

A circle is defined by its center (c_x, c_y) and radius r : `__svg_new_circle(cx, cy, r)`. An ellipse has a center and two semiaxes: `__svg_new_ellipse(cx, cy, rx, ry)`. Individual primitives need no private coordinate system; they share the SVG sheet.

3.4 LINES, FILLS, AND STROKES

`__svg_new_line()` creates a segment between two points. `__svg_set_stroke()` sets its stroke color, `__svg_set_fill()` sets its fill color, and `__svg_set_stroke_width()` sets the line width. The same primitive may be used as an independent shape, part of a conventional symbol, or an element of a more complex diagram.

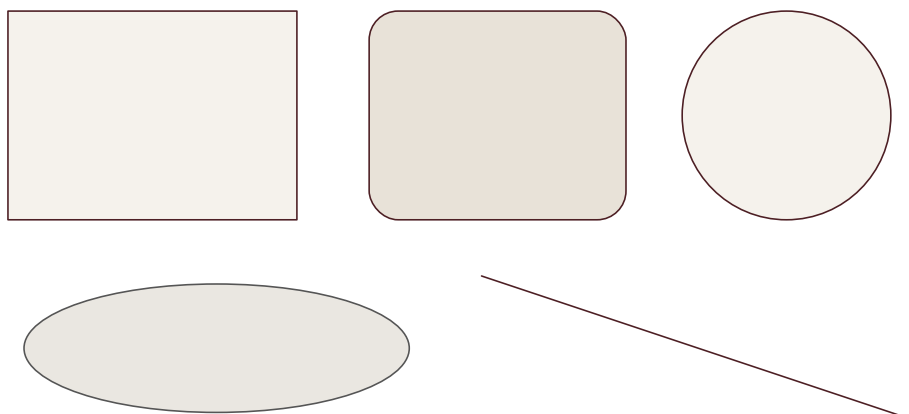


Figure 3.1. Several basic primitives with different styles.

3.5 EXAMPLE 2

Example 002-primitives is deliberately compact. It creates a background, two rectangles, a circle, an ellipse, and a segment, then writes `primitives.svg`.

```

/*****
  SVGC EXAMPLE 002: GEOMETRIC PRIMITIVES

  The scene contains rectangles, circles, ellipses and lines.
  The upper-left corner of the SVG sheet is (0, 0).
  *****/

#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill )
{
    if( !node ) return( 0 );

    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );

    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );

    __svg_free_node( node );
    return( 0 );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 660, 340, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );

    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    if( !append( scene, __svg_new_rect( 0, 0, 660, 340, 0, 0 ), "none", "white" ) ) goto done;
    if( !append( scene, __svg_new_rect( 25, 40, 180, 130, 0, 0 ), "#4D1F24", "#F5F2EC" ) ) goto done;
    if( !append( scene, __svg_new_rect( 250, 40, 160, 130, 18, 18 ), "#4D1F24", "#E8E2D8" ) ) goto done;
    if( !append( scene, __svg_new_circle( 510, 105, 65 ), "#4D1F24", "#F5F2EC" ) ) goto done;
    if( !append( scene, __svg_new_ellipse( 155, 250, 120, 40 ), "#555555", "#EAE7E1" ) ) goto done;
    if( !append( scene, __svg_new_line( 320, 205, 590, 295 ), "#4D1F24", NULL ) ) goto done;

    __svg_paint_scene( scene, 0, "primitives.svg" );
    status = 0;

done:
    if( status ) fprintf( stderr, "Could not construct SVG scene\n" );
    __svg_free_scene( scene );
    return( status );
}

```

```
cd svgc-examples/002-primitives
make
# open view.html in a browser
```

3.6 ARCS, CHORDS, AND ARROWS: WHAT COMES NEXT

Circular arcs, chords, and arrows require further geometry. SVGC provides paths (*path*), arcs, and curves for compound outlines. A chord is simply a segment connecting two points on an arc; an arrow can be assembled from a line and the closed outline of its head.

Rather than introducing a special library function for every such shape, we calculate the necessary parameters in application code. Later chapters construct different arrowheads, flowcharts, and control-system diagrams in detail.

4. Coordinate Axes Derived from Data

4.1 WHY AXES SHOULD NOT COINCIDE WITH THE FRAME

In computational problems, mathematical coordinates come from an input array. If the upper-left corner of the SVG sheet were simply treated as the mathematical origin, the positive y direction would point down, and displaying symmetric negative values would be awkward. The positions and scales of mathematical axes should instead be computed from the data range and the useful dimensions of the drawing area.

4.2 BOUNDS OF THE INPUT DATA

For a set of pairs (x_i, y_i) , define its minimum and maximum coordinates by

$$x_{\min} = \min_i x_i, \quad x_{\max} = \max_i x_i, \quad y_{\min} = \min_i y_i, \quad y_{\max} = \max_i y_i.$$

In this teaching example, the ranges are expanded to include the mathematical origin, and then small margins are added on both axes. This is one possible plotting policy, not a requirement imposed by SVGC. Logarithmic scales, fixed scales, or phase portraits may call for different rules.

4.3 AFFINE COORDINATE TRANSFORMATION

Suppose the usable drawing rectangle begins at (L, T) , with width W and height H . A mathematical point (x, y) is converted to SVG coordinates (X, Y) by

$$X = L + W \frac{x - x_{\min}}{x_{\max} - x_{\min}}, \quad Y = T + H \frac{y_{\max} - y}{y_{\max} - y_{\min}}.$$

Thus the upward mathematical y direction is mapped to SVG's downward screen direction. The mathematical axes are derived from the position of $(0, 0)$ and may cross the plotting area anywhere, depending on the actual data set.

A degenerate interval requires an explicit rule: division by zero is not permitted. The example's `find_bounds` function expands a zero-width interval by a nonzero amount. A complete plotting system should state this policy clearly.

4.4 AN EXAMPLE WITH DISCRETE VALUES

Consider seven samples of $y = x^2 - 2$, taken with unit spacing in x . The values are chosen solely to illustrate the geometry: SVGC does not know the function's formula. The application must find the bounds, select the scale, draw the axes, and connect the points with a polyline.

```

/*****
SVG EXAMPLE 003: DATA-DRIVEN CARTESIAN AXES

Coordinate conversion depends on the actual data extrema.
SVG is used only to emit graphical nodes.
*****/

```

```

#include <float.h>
#include <math.h>
#include <stdio.h>
#include <svgc.h>

struct point
{
    double x;
    double y;
};

struct plot
{
    double xmin, xmax, ymin, ymax;
    double left, top, width, height;
};

static double
plot_x( const struct plot *p, double x )
{
    return( p->left + ( x - p->xmin ) * p->width / ( p->xmax - p->xmin ) );
}

static double
plot_y( const struct plot *p, double y )
{
    return( p->top + ( p->ymax - y ) * p->height / ( p->ymax - p->ymin ) );
}

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );

    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );

    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );

    __svg_free_node( node );
    return( 0 );
}

static void
find_bounds( struct plot *p, const struct point *a, unsigned long count )
{
    unsigned long i;
    double dx, dy;

    p->xmin = p->xmax = a[0].x;
    p->ymin = p->ymax = a[0].y;

    for( i = 1; i < count; ++i )
    {
        if( a[i].x < p->xmin ) p->xmin = a[i].x;
        if( a[i].x > p->xmax ) p->xmax = a[i].x;
        if( a[i].y < p->ymin ) p->ymin = a[i].y;
    }
}

```

```

    if( a[i].y > p->ymax ) p->ymax = a[i].y;
}

/* Include the mathematical origin, not the SVG canvas origin. */
if( p->xmin > 0.0 ) p->xmin = 0.0;
if( p->xmax < 0.0 ) p->xmax = 0.0;
if( p->ymin > 0.0 ) p->ymin = 0.0;
if( p->ymax < 0.0 ) p->ymax = 0.0;

dx = p->xmax - p->xmin;
dy = p->ymax - p->ymin;
if( dx == 0.0 ) dx = 1.0;
if( dy == 0.0 ) dy = 1.0;

p->xmin -= 0.12 * dx;
p->xmax += 0.12 * dx;
p->ymin -= 0.12 * dy;
p->ymax += 0.12 * dy;
}

int
main( void )
{
    const struct point a[] = {
        { -3.0, 6.0 }, { -2.0, 2.0 }, { -1.0, -1.0 },
        { 0.0, -2.0 }, { 1.0, -1.0 }, { 2.0, 2.0 },
        { 3.0, 6.0 }
    };
    const unsigned long n = sizeof( a ) / sizeof( a[0] );
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    struct plot      p;
    unsigned long    i;
    int               status = 1;

    p.left = 70.0;
    p.top = 35.0;
    p.width = 560.0;
    p.height = 310.0;
    find_bounds( &p, a, n );

    sheet = (struct __svg_sheet *)__svg_new_sheet( 700, 400, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    if( !append( scene, __svg_new_rect( 0, 0, 700, 400, 0, 0 ), "none", "white", 0.0 ) ) goto done;
    if( !append( scene, __svg_new_rect( p.left, p.top, p.width, p.height, 0, 0 ), "#B3B3B3", "none", 0.7 ) )
        goto done;

    if( !append( scene, __svg_new_line( p.left, plot_y( &p, 0.0 ), p.left + p.width, plot_y( &p, 0.0 ) ), "
        #555555", NULL, 1.2 ) ) goto done;
    if( !append( scene, __svg_new_line( plot_x( &p, 0.0 ), p.top, plot_x( &p, 0.0 ), p.top + p.height ), "
        #555555", NULL, 1.2 ) ) goto done;

    for( i = 0; i < n; ++i )
    {
        double x = plot_x( &p, a[i].x );
        double y = plot_y( &p, a[i].y );

```

```

    if( i > 0 )
    {
        if( !append( scene, __svg_new_line( plot_x( &p, a[i-1].x ), plot_y( &p, a[i-1].y ), x, y ), "#4D1F24", NULL, 2.0 ) ) goto done;
    }

    if( !append( scene, __svg_new_circle( x, y, 4.5 ), "#4D1F24", "#4D1F24", 0.7 ) ) goto done;
}

__svg_paint_scene( scene, 0, "data-coordinates.svg" );
status = 0;

done:
    if( status ) fprintf( stderr, "Could not construct SVG graph\n" );
    __svg_free_scene( scene );
    return( status );
}

```

```

cd svgc-examples/003-data-coordinates
make
# open view.html in a browser

```

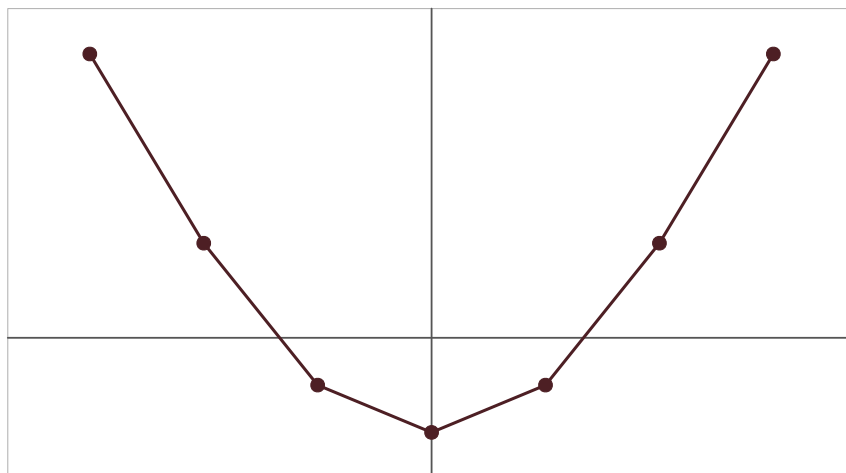


Figure 4.1. A discrete plot whose coordinate origin is calculated from the data array.

Figure 4.1 is intentionally simple. At this stage we are interested in constructing a polyline from a given sequence of points, so the focus is on the geometry of individual segments and their SVG representation. Axis styling, scaling, tick placement, and numerical labels are left for later.

Connecting adjacent sample points with straight segments does not produce a smooth curve. Many scientific graphics tasks require a continuous, smooth curve passing through specified nodes, constructed using interpolation methods.

Subsequent chapters explain how to construct and style coordinate axes and how to apply cubic interpolation splines. These techniques lead from simple geometric constructions to full scientific plots, where curve shapes follow mathematical methods and the appearance satisfies the needs of presenting research results.

5. Polylines, Polygons, and Discrete Geometry

5.1 FROM SEGMENTS TO ARRAYS OF VERTICES

A segment is defined by two points. When a drawing consists of a sequence of segments, it is more convenient to keep the vertices in an array than to construct every segment independently. This pattern appears in engineering wireframes, experimental curves, boundaries of computational domains, and many diagrams of numerical methods.

SVGC represents these two cases with different SVG elements: `polyline` is an open **polyline**, while `polygon` is a closed **polygon**. Both accept an ordered sequence of two-dimensional points.

Let the array contain vertices

$$P_0 = (x_0, y_0), P_1 = (x_1, y_1), \dots, P_{n-1} = (x_{n-1}, y_{n-1}).$$

A polyline contains segments $P_0P_1, \dots, P_{n-2}P_{n-1}$. A polygon also contains the closing edge $P_{n-1}P_0$. There is no need to repeat the first vertex in the source array.

5.2 VERTEX ORDER AND CLOSURE

Vertex order determines segment order. Exchanging two vertices changes the geometry. This matters for polygons as well: a careless reordering can make the outline self-intersect. SVG filling rules determine how such shapes are filled; using a polygon alone does not establish that the region is simple or geometrically valid.

An open polyline should normally be drawn without fill, using `fill:none`. A closed shape may have both a stroke (`stroke`) and a fill (`fill`). Styling does not change the coordinate array: the same vertices can be rendered with different line widths and colors.

5.3 THE PUBLIC SVGC INTERFACE

A point sequence is created with `__svg_new_points()`. Each vertex is appended using `__svg_add_point(points, x, y)`. The resulting sequence is passed to `__svg_new_polyline(points)` or `__svg_new_polygon(points)`.

Once a graphical node is created successfully, it owns the point sequence, which is released along with the corresponding scene. If node creation fails, the application must free the untransferred points itself. Our example does so in `append_points()`.

An important detail: `__svg_add_point()` returns a positive value on success, not necessarily the number one. Testing `< 1` is therefore correct, whereas requiring `== 1` would reject the second and later successfully appended vertices.

5.4 DRAWING DISCRETE MEASUREMENTS

The left side of Figure 5.1 shows eight observations. The connecting line is a polyline, while small circles emphasize the coordinates of the original samples. This distinction matters: missing intermediate experimental values must not be disguised by presenting a polyline as an analytic curve.

The right side shows a closed region. All vertices are explicitly specified, and the fill indicates the figure's

interior.

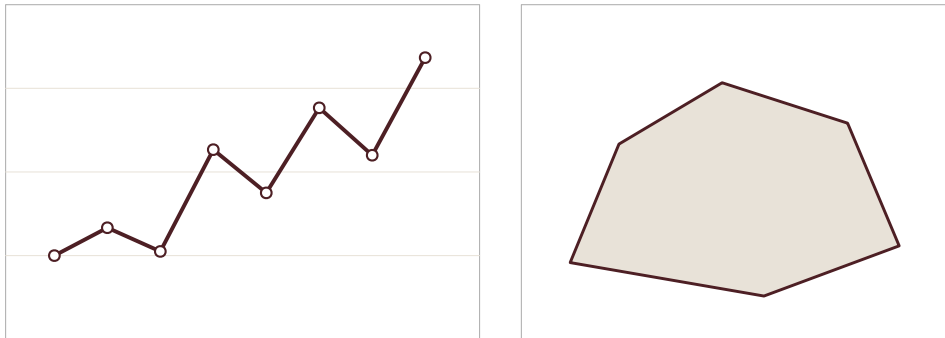


Figure 5.1. An open polyline through discrete data and a closed polygon.

5.5 EXAMPLE 4

Example 004-polylines needs no additional plotting libraries. Its source array serves both as the drawing's geometric model and as input to SVGC. The library performs no hidden interpolation.

```

/*****
SVGC EXAMPLE 004: POLYLINES AND POLYGONS

A polyline does not close its contour; a polygon does.
Point arrays belong to the application, not the library.
*****/

#include <stdio.h>
#include <svgc.h>

struct point
{
    double x;
    double y;
};

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );

    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill )   (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );

    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );

    __svg_free_node( node );
    return( 0 );
}

```

```

static int
append_points( struct __svg_scene *scene, const struct point *a,
               unsigned long n, int closed, const char *stroke,
               const char *fill, double width )
{
    struct __svg_points *points;
    struct __svg_node *node;
    unsigned long i;

    points = __svg_new_points();
    if( !points ) return( 0 );

    for( i = 0; i < n; ++i )
    {
        if( __svg_add_point( points, a[i].x, a[i].y ) < 1 )
        {
            __svg_free_points( points );
            return( 0 );
        }
    }

    if( closed ) node = __svg_new_polygon( points );
    else node = __svg_new_polyline( points );

    if( !node )
    {
        __svg_free_points( points );
        return( 0 );
    }

    return( append( scene, node, stroke, fill, width ) );
}

int
main( void )
{
    const struct point measurements[] = {
        { 75, 255 }, { 113, 235 }, { 151, 252 }, { 189, 179 },
        { 227, 210 }, { 265, 149 }, { 303, 183 }, { 341, 113 }
    };
    const struct point region[] = {
        { 445, 260 }, { 480, 175 }, { 554, 131 },
        { 644, 160 }, { 681, 248 }, { 584, 284 }
    };
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    unsigned long i;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 760, 360, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );

    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    if( !append( scene, __svg_new_rect( 0, 0, 760, 360, 0, 0 ), "none", "white", 0.0 ) ) goto done;
    if( !append( scene, __svg_new_rect( 40, 75, 340, 240, 0, 0 ), "#B3B3B3", "none", 0.8 ) ) goto done;
    if( !append( scene, __svg_new_rect( 410, 75, 310, 240, 0, 0 ), "#B3B3B3", "none", 0.8 ) ) goto done;

```

```

for( i = 1; i <= 3; ++i )
{
    double y = 75.0 + 60.0 * i;
    if( !append( scene, __svg_new_line( 40, y, 380, y ), "#E8E2D8", "none", 0.7 ) ) goto done;
}

if( !append_points( scene, measurements, sizeof( measurements ) / sizeof( measurements[0] ),
                    0, "#4D1F24", "none", 2.8 ) ) goto done;

for( i = 0; i < sizeof( measurements ) / sizeof( measurements[0] ); ++i )
{
    if( !append( scene, __svg_new_circle( measurements[i].x, measurements[i].y, 3.8 ),
                "#4D1F24", "white", 1.5 ) ) goto done;
}

if( !append_points( scene, region, sizeof( region ) / sizeof( region[0] ),
                    1, "#4D1F24", "#E8E2D8", 2.1 ) ) goto done;

__svg_paint_scene( scene, 0, "polylines.svg" );
status = 0;

done:
if( status ) fprintf( stderr, "Could not construct polyline/polygon drawing\n" );
__svg_free_scene( scene );
return( status );
}

```

```

cd svgc-examples/004-polylines
make
# open view.html in a browser

```

5.6 THE BOUNDARY BETWEEN VISUALIZATION AND COMPUTATION

Here the vertex positions are specified directly in screen coordinates. In a numerical experiment they will usually be mathematical points (x_i, y_i) . The transformation function introduced in Chapter 4 first computes their screen positions, and `__svg_add_point()` receives the resulting values.

An array of vertices is a geometric representation of the input data. SVGC records it in SVG; the application algorithm determines what the point order means, how the outline is closed, and which scale is used.

6. Arrows and Arrowheads

6.1 WHY AN ARROW IS A COMPOUND OBJECT

In a technical drawing, an arrow conveys direction: data flow, control flow, coordinate-axis orientation, or the action of a force. Its two endpoints are not enough to define it. The shape and dimensions of the arrowhead, its direction, whether a second head is present, and how the head joins the line must also be specified.

This chapter's example illustrates five constructions: a filled triangular head, an open head, a double-headed arrow, an arrow with a diamond at the start and a triangle at the end, and an axis arrow based on Figure 16.1. All are composed of ordinary SVG primitives available through the public SVGC interface.

6.2 THE UNIT DIRECTION VECTOR

Let a line start at $A = (x_1, y_1)$ and end at $B = (x_2, y_2)$. Its length and unit direction vector are

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}, \quad \mathbf{u} = \frac{1}{d}(x_2 - x_1, y_2 - y_1).$$

Rotating this direction by a right angle gives the normal

$$\mathbf{n} = (-u_y, u_x).$$

If the arrowhead tip is at B , its length is ℓ , and half the base width is w , the corners of a filled triangular head are

$$C_+ = B - \ell\mathbf{u} + w\mathbf{n}, \quad C_- = B - \ell\mathbf{u} - w\mathbf{n}.$$

A single three-vertex polygon is therefore sufficient. An open head can be drawn as the polyline $C_+ \rightarrow B \rightarrow C_-$. A diamond additionally uses the opposite corner $B - \ell\mathbf{u}$ and two lateral vertices halfway along the head length.

The axis arrowhead (see Figure 16.1) uses a different outline: it is a path, not a polygon. Two straight edges lead to the tip, while a short smooth curve closes the base. This shape works especially well at the positive ends of coordinate axes.

6.3 START AND END POINT IN OPPOSITE DIRECTIONS

The head at the end of a line uses $\mathbf{u}$; the head at its start uses $-\mathbf{u}$. Without the reversal, both heads of a double-headed arrow would point the same way. The example uses `hypot` from the standard C math library to compute nonzero lengths. Coincident endpoints give an undefined direction, so no meaningful arrow can be constructed.

The example specifies arrowhead dimensions in screen units, independently of line length. This is convenient for engineering diagrams: connections can change length while the direction symbols remain a consistent size.

6.4 EXAMPLE 5

In `005-arrows`, the `head_type` enumeration describes arrowhead semantics. `add_head` constructs the selected geometry, and `add_arrow()` combines the segment with up to two heads. A separate `add_axis_head` function creates the axis head with a path and a short smooth curve at its base.

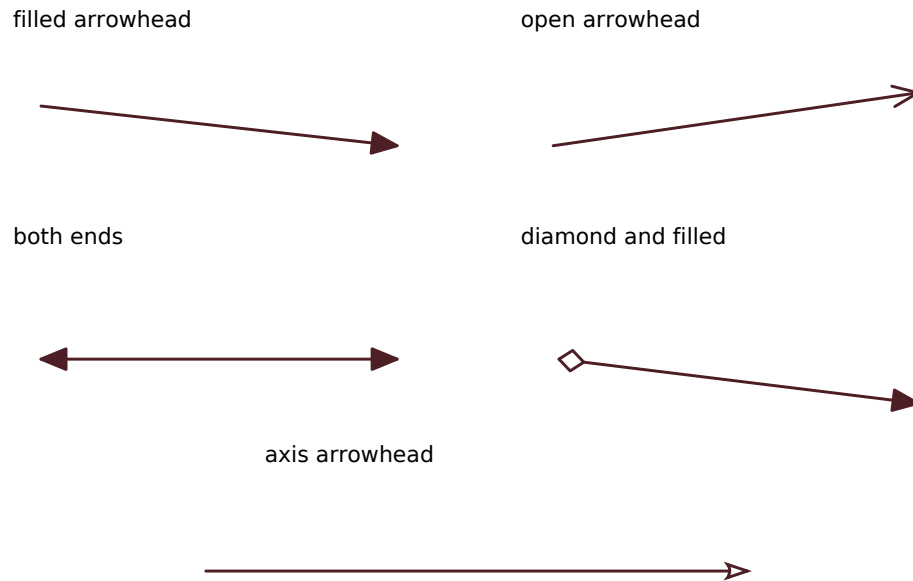


Figure 6.1. Five kinds of arrowhead for directed connections.

```

/*****
  SVGC EXAMPLE 005: ARROW ENDINGS

  Each arrow is a composition of a line and endpoint shapes.
  The vector geometry stays in the application.
  *****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

struct point
{
    double x;
    double y;
};

enum head_type
{
    HEAD_NONE,
    HEAD_TRIANGLE,
    HEAD_OPEN,
    HEAD_DIAMOND,
    HEAD_AXIS
};

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );

```

```

    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
append_path( struct __svg_scene *scene, struct __svg_cmds *cmds,
             const char *stroke, const char *fill, double width,
             const char *linecap, const char *linejoin )
{
    struct __svg_node *node;

    if( !cmds ) return( 0 );
    node = __svg_new_path( cmds );
    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( linecap ) (void)__svg_set_stroke_linecap( node, linecap );
    if( linejoin ) (void)__svg_set_stroke_linejoin( node, linejoin );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_points( struct __svg_scene *scene, const struct point *a,
            unsigned long n, int closed, const char *fill )
{
    struct __svg_points *points = __svg_new_points();
    struct __svg_node *node;
    unsigned long i;

    if( !points ) return( 0 );
    for( i = 0; i < n; ++i )
    {
        if( __svg_add_point( points, a[i].x, a[i].y ) < 1 )
        {
            __svg_free_points( points );
            return( 0 );
        }
    }
    if( closed ) node = __svg_new_polygon( points );
    else node = __svg_new_polyline( points );
    if( !node )
    {
        __svg_free_points( points );
        return( 0 );
    }
    return( append( scene, node, "#4D1F24", fill, 2.0 ) );
}

static int
add_axis_head( struct __svg_scene *scene, double x, double y,
              double ux, double uy )
{
    double px = -uy, py = ux;

```

```

struct __svg_cmds *cmds;
double          ax1, ay1, ax2, ay2, ax3, ay3, ax4, ay4;

ax1 = x - 4.5*px - 13.5*ux;
ay1 = y - 4.5*py - 13.5*uy;
ax2 = x;
ay2 = y;
ax3 = x + 4.5*px - 13.5*ux;
ay3 = y + 4.5*py - 13.5*uy;
ax4 = x - 10.0*ux;
ay4 = y - 10.0*uy;

cmds = __svg_new_cmds();
if( !cmds ) return( 0 );

if( __svg_add_moveto_cmd( cmds, ABSOLUTE, ax1, ay1 ) < 1 )
{
    __svg_free_cmds( cmds );
    return( 0 );
}
if( __svg_add_lineto_cmd( cmds, ABSOLUTE, ax2, ay2 ) < 1 ||
    __svg_add_lineto_cmd( cmds, ABSOLUTE, ax3, ay3 ) < 1 ||
    __svg_add_short_curveto_cmd( cmds, ABSOLUTE, ax4, ay4, ax1, ay1 ) < 1 ||
    __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
{
    __svg_free_cmds( cmds );
    return( 0 );
}

return( append_path( scene, cmds, "#4D1F24", "white", 2.0,
                    LINECAP_SQUARE, LINEJOIN_MITER ) );
}

static int
add_head( struct __svg_scene *scene, double x, double y,
          double ux, double uy, enum head_type type )
{
    double px = -uy, py = ux;
    double length = 17.0, half_width = 7.0;
    struct point a[4];

    if( type == HEAD_NONE ) return( 1 );
    if( type == HEAD_AXIS ) return( add_axis_head( scene, x, y, ux, uy ) );

    a[0] = (struct point){ x, y };
    a[1] = (struct point){ x - ux*length + px*half_width,
                          y - uy*length + py*half_width };
    a[2] = (struct point){ x - ux*length - px*half_width,
                          y - uy*length - py*half_width };

    if( type == HEAD_OPEN )
    {
        struct point v[3] = { a[1], a[0], a[2] };
        return( add_points( scene, v, 3, 0, "none" ) );
    }
    if( type == HEAD_TRIANGLE )
        return( add_points( scene, a, 3, 1, "#4D1F24" ) );

    a[1] = (struct point){ x - ux*length*0.5 + px*half_width,
                          y - uy*length*0.5 + py*half_width };
    a[2] = (struct point){ x - ux*length, y - uy*length };
    a[3] = (struct point){ x - ux*length*0.5 - px*half_width,
                          y - uy*length*0.5 - py*half_width };
    return( add_points( scene, a, 4, 1, "white" ) );
}

```

```

}

static int
add_arrow( struct __svg_scene *scene, double x1, double y1,
           double x2, double y2, enum head_type at_start,
           enum head_type at_end )
{
    double dx = x2 - x1, dy = y2 - y1;
    double length = hypot( dx, dy );
    double ux, uy;

    if( length <= 0.0 ) return( 0 );
    ux = dx / length;
    uy = dy / length;

    if( !append( scene, __svg_new_line( x1, y1, x2, y2 ), "#4D1F24", "none", 2.0 ) ) return( 0 );
    if( !add_head( scene, x1, y1, -ux, -uy, at_start ) ) return( 0 );
    if( !add_head( scene, x2, y2, ux, uy, at_end ) ) return( 0 );
    return( 1 );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 720, 500, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    if( !append( scene, __svg_new_rect( 0, 0, 720, 500, 0, 0 ), "none", "white", 0 ) ) goto done;
    if( !append( scene, __svg_new_text( 45, 44, "filled arrowhead", "DejaVu Sans", 15 ), "none", "none", 0 )
        ) goto done;
    if( !append( scene, __svg_new_text( 390, 44, "open arrowhead", "DejaVu Sans", 15 ), "none", "none", 0 )
        ) goto done;
    if( !append( scene, __svg_new_text( 45, 192, "both ends", "DejaVu Sans", 15 ), "none", "none", 0 )
        ) goto done;
    if( !append( scene, __svg_new_text( 390, 192, "diamond and filled", "DejaVu Sans", 15 ), "none", "none",
        0 ) ) goto done;
    if( !append( scene, __svg_new_text( 216, 340, "axis arrowhead", "DejaVu Sans", 15 ), "none", "none", 0 )
        ) goto done;

    if( !add_arrow( scene, 64, 98, 306, 125, HEAD_NONE, HEAD_TRIANGLE ) ) goto done;
    if( !add_arrow( scene, 412, 125, 660, 89, HEAD_NONE, HEAD_OPEN ) ) goto done;
    if( !add_arrow( scene, 64, 270, 306, 270, HEAD_TRIANGLE, HEAD_TRIANGLE ) ) goto done;
    if( !add_arrow( scene, 416, 270, 660, 300, HEAD_DIAMOND, HEAD_TRIANGLE ) ) goto done;
    if( !add_arrow( scene, 176, 414, 544, 414, HEAD_NONE, HEAD_AXIS ) ) goto done;

    __svg_paint_scene( scene, 0, "arrows.svg" );
    status = 0;

done:
    if( status ) fprintf( stderr, "Could not construct arrows drawing\n" );
    __svg_free_scene( scene );
    return( status );
}

```

```
cd svgc-examples/005-arrows
make
# open view.html in a browser
```

6.5 SCALE, STYLES, AND PAINTING ORDER

The example appends the line segment to the scene before its arrowhead. The filled triangle therefore covers the line beneath it rather than showing an unwanted intersection. The hollow diamond and axis head have white fills, concealing any line continuation behind them.

For compound diagrams, it helps to control connection widths, arrowhead geometry, and color independently. Whether information flows should be color-coded or the image should remain monochrome for printing is a decision for the author of the scientific illustration.

An arrow has a semantic direction and a geometric head. The application computes both explicitly; SVGC records the chosen composition of segments, polylines, polygons, and path objects.

7. Generating Flowcharts Programmatically

7.1 A GRAPH OF CONTROL TRANSFERS

A program flowchart is a graph: its vertices represent actions, tests, and entry or exit points, while directed edges represent possible transfers of control. The drawing is useful only when its geometry expresses the actual algorithm rather than an arbitrary arrangement of boxes on a page.

Four conventional elements provide a starting point: an ellipse for start or end, a rectangle for an operation, a diamond for a condition, and an arrow for a connection. They suffice to represent a simple iterative computational process.

7.2 EXAMPLE: A COMPUTATIONAL LOOP

Consider a program that performs computations while the step counter is less than a specified value n :

```
set initial data
while( step < n )
{
    calculate;
    step++;
}
```

This is intentionally pseudocode: our concern is the control transfers, not the choice of a modeling language. The condition diamond has two branches. When the condition is true, execution proceeds to the calculation block and returns to the test; otherwise, the loop ends. The operation rectangle names both actions: performing the computation and incrementing the counter.

For legibility, edges are drawn before vertices. Ellipses, rectangles, and the diamond are then added, followed by their labels. This avoids unwanted lines appearing through the fill of a block. The returning branch follows several orthogonal segments calculated in C.

7.3 EXAMPLE 6

`add_polygon()` creates the condition diamond and triangular arrowheads. `add_line()` and `add_arrow()` define connections, while `add_text()` places labels. These are application-level helpers, not additions to the SVG C library.

```
/******
SVG C EXAMPLE 006: PROGRAM FLOWCHART

Control-flow structure is explicitly defined by the C program.
All coordinates are screen coordinates, with Y downwards.
*****/

#include <stdio.h>
#include <svgc.h>
```

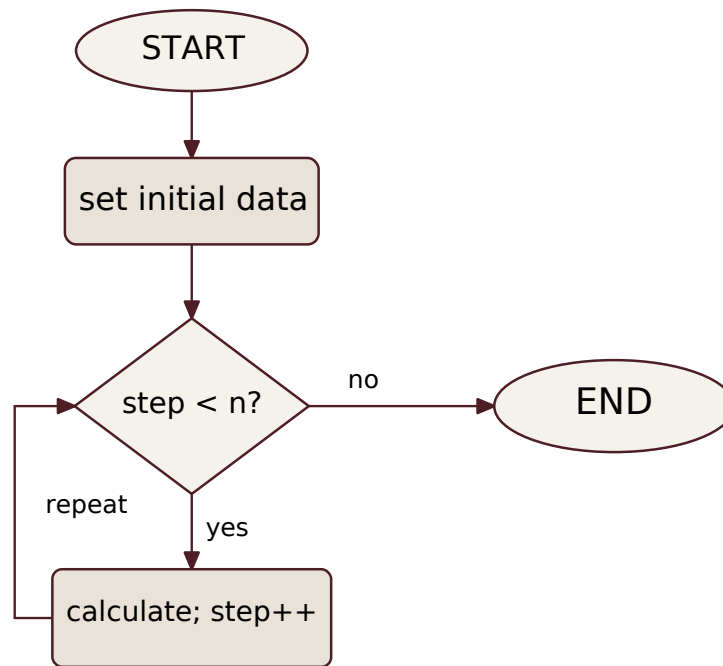


Figure 7.1. A flowchart of an iterative algorithm with a loop exit.

```

struct point
{
    double x;
    double y;
};

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_polygon( struct __svg_scene *scene, const struct point *a,
             unsigned long n, const char *fill )
{
    struct __svg_points *points = __svg_new_points();
    struct __svg_node *node;
    unsigned long i;

    if( !points ) return( 0 );
    for( i = 0; i < n; ++i )
    {
        if( __svg_add_point( points, a[i].x, a[i].y ) < 1 )

```

```

    {
        __svg_free_points( points );
        return( 0 );
    }
}

node = __svg_new_polygon( points );
if( !node )
{
    __svg_free_points( points );
    return( 0 );
}
return( append( scene, node, "#4D1F24", fill, 1.8 ) );
}

static int
add_line( struct __svg_scene *scene, double x1, double y1,
          double x2, double y2 )
{
    return( append( scene, __svg_new_line( x1, y1, x2, y2 ),
                  "#4D1F24", "none", 1.7 ) );
}

static int
add_arrow( struct __svg_scene *scene, double x1, double y1,
           double x2, double y2 )
{
    const double size = 12.0, half = 5.0;
    struct point p[3];
    double dx = x2 - x1, dy = y2 - y1;
    double norm;
    double ux, uy;

    /* Orthogonal links are enough for this introductory flowchart. */
    if( dx == 0.0 && dy == 0.0 ) return( 0 );
    norm = ( dx > 0.0 ? dx : -dx ) + ( dy > 0.0 ? dy : -dy );
    ux = dx / norm;
    uy = dy / norm;

    if( !add_line( scene, x1, y1, x2, y2 ) ) return( 0 );
    p[0] = (struct point){ x2, y2 };
    p[1] = (struct point){ x2 - ux*size - uy*half,
                          y2 - uy*size + ux*half };
    p[2] = (struct point){ x2 - ux*size + uy*half,
                          y2 - uy*size - ux*half };
    return( add_polygon( scene, p, 3, "#4D1F24" ) );
}

static int
add_text( struct __svg_scene *scene, double x, double y,
          const char *caption, int size )
{
    return( append( scene, __svg_new_text( x, y, caption, "DejaVu Sans", size ),
                  "none", "none", 0 ) );
}

int
main( void )
{
    const struct point decision[] = {
        { 180, 260 }, { 264, 323 }, { 180, 386 }, { 96, 323 }
    };

```

```

struct __svg_sheet *sheet;
struct __svg_scene *scene;
int status = 1;

sheet = (struct __svg_sheet *)__svg_new_sheet( 760, 590, 20, 20, 20, 20 );
if( !sheet ) return( 1 );
scene = __svg_new_scene( sheet );
if( !scene )
{
    __svg_free_node( (struct __svg_node *)sheet );
    return( 1 );
}

if( !append( scene, __svg_new_rect( 0, 0, 760, 590, 0, 0 ), "none", "white", 0 ) ) goto done;

/* Connections are inserted before blocks, so lines pass behind them. */
if( !add_arrow( scene, 180, 97, 180, 145 ) ) goto done;
if( !add_arrow( scene, 180, 207, 180, 260 ) ) goto done;
if( !add_arrow( scene, 180, 386, 180, 440 ) ) goto done;
if( !add_line( scene, 80, 475, 53, 475 ) ) goto done;
if( !add_line( scene, 53, 475, 53, 323 ) ) goto done;
if( !add_arrow( scene, 53, 323, 96, 323 ) ) goto done;
if( !add_arrow( scene, 264, 323, 397, 323 ) ) goto done;

if( !append( scene, __svg_new_ellipse( 180, 68, 83, 29 ), "#4D1F24", "#F5F2EC", 1.8 ) ) goto done;
if( !append( scene, __svg_new_rect( 89, 145, 182, 62, 8, 8 ), "#4D1F24", "#E8E2D8", 1.8 ) ) goto done;
if( !add_polygon( scene, decision, 4, "#F5F2EC" ) ) goto done;
if( !append( scene, __svg_new_rect( 80, 440, 200, 70, 7, 7 ), "#4D1F24", "#E8E2D8", 1.8 ) ) goto done;
if( !append( scene, __svg_new_ellipse( 483, 323, 86, 33 ), "#4D1F24", "#F5F2EC", 1.8 ) ) goto done;

if( !add_text( scene, 144, 74, "START", 23 ) ) goto done;
if( !add_text( scene, 99, 182, "set initial data", 23 ) ) goto done;
if( !add_text( scene, 130, 328, "step &lt; n?", 21 ) ) goto done;
if( !add_text( scene, 90, 477, "calculate; step++", 20 ) ) goto done;
if( !add_text( scene, 455, 329, "END", 26 ) ) goto done;
if( !add_text( scene, 190, 416, "yes", 18 ) ) goto done;
if( !add_text( scene, 292, 310, "no", 18 ) ) goto done;
if( !add_text( scene, 75, 400, "repeat", 18 ) ) goto done;

__svg_paint_scene( scene, 0, "flowchart.svg" );
status = 0;

done:
if( status ) fprintf( stderr, "Could not construct program flowchart\n" );
__svg_free_scene( scene );
return( status );
}

```

```

cd svgc-examples/006-flowchart
make
# open view.html in a browser

```

7.4 LABELS INSIDE BLOCKS: WHY CENTERING IS MANUAL

Unlike the dimensions of rectangles and the positions of their vertices, the rendered width of a label cannot be found simply by counting characters. Each character is drawn as a **glyph**; its width depends on the font family, style, size, character spacing, and treatment of spaces. For instance, *W* and *i* typically have quite different widths even though each is a single character in the source text.

SVG C records font family, size, and coordinates in a text element; it does not parse font files or compute the

rendered width of a whole string for a particular rendering engine. A label cannot therefore be reliably centered by multiplying its character count by a fixed character width, nor will SVGC center it automatically.

In `006-flowchart`, `add_text()` receives starting coordinates *chosen manually in advance*. Check their appearance in a browser after generating the SVG, especially when changing fonts or labels. The block's geometric center is known exactly, but the visual center of the text depends on its actual rendered shape.

An application could, in principle, use an external text-measurement engine or known metrics for a particular font. Such typesetting, however, is separate from the primitives used to create SVG objects. We will not add a font engine to SVGC merely to center labels.

7.5 SVG TEXT AND XML SAFETY

SVG is XML. Some characters valid in C string literals cannot be inserted directly into XML element content. In particular, the `<` character starts markup and must be represented as `<` in the output document. The SVGC implementation used here writes text directly, without universal content escaping. Our source therefore passes `step < n?`, which the browser renders as the familiar condition `step < n?`.

In practical generators, labels may originate in external data. The application must then escape XML special characters correctly, including `&` and `<`, as appropriate to the context. Arbitrary byte sequences cannot safely be inserted into XML without preparation.

7.6 AUTOMATIC LAYOUT IS A SEPARATE PROBLEM

This example does not attempt automatic graph layout. All dimensions and positions are predetermined. Larger automatically generated diagrams require algorithms for vertex placement, edge routing, crossing avoidance, and legibility checks. SVGC should not be made a graph-analysis engine: it merely writes the resulting lines and shapes.

The same approach can later be applied to control-system diagrams: a mathematical or functional model defines the blocks and links, an application algorithm computes their positions, and SVGC writes the SVG document.

A flowchart does not emerge spontaneously from a graphics editor. It is a geometric representation of a control-flow graph: the algorithm defines the graph, application C code lays it out, and SVGC produces the document.

8. Object Appearance: Strokes, Fills, and Dashes

8.1 GEOMETRY AND STYLE ARE SEPARATE

A drawn line is geometrically a segment defined by its endpoints. Its visible appearance is specified separately by its color, width, end-cap style, and dash pattern. Likewise, a rectangle's dimensions define its region, but its fill color does not alter its coordinates. This distinction is fundamental to programmatic scientific drawing.

Example 007-styles renders the same line using different cap and dash settings. The original C program still determines all geometry, and SVGC writes the settings to the corresponding SVG element's `style` attribute.

8.2 STROKE AND FILL COLORS

The `stroke` property sets the outline color, while `fill` sets the color inside a closed region. SVGC provides `__svg_set_stroke()` and `__svg_set_fill()`, as well as `..._rgb` variants that accept individual color components. The value `none` means that nothing is drawn; it does not mean white. A white fill, by contrast, is a real opaque region that can conceal lines underneath it.

Stroke thickness is specified by `stroke-width` using `__svg_set_stroke_width()`. A thick stroke extends visibly to both sides of the mathematical line. Even perfectly calculated geometry can therefore look different when its styling changes.

8.3 LINE CAPS AND SEGMENT JOINS

An open segment can have three kinds of cap: `butt` ends exactly at the endpoint; `round` produces a semicircular cap; and `square` extends beyond the endpoint by approximately half the stroke width. These distinctions are noticeable on fine axes and arrows.

Polyline joins use the `stroke-linejoin` property: `miter`, `round`, or `bevel`. A sharp corner with `miter` may create a long protrusion. Choose the join according to the figure's purpose rather than accepting an arbitrary default.

8.4 DASHES AND VISUAL RHYTHM

The `stroke-dasharray` property specifies alternating lengths of visible dashes and gaps. For example, `18, 12` means 18 units of stroke followed by 12 units of space, repeated. This matters for construction lines, grids, and auxiliary geometry: the dashed pattern should not distract from the computed result.

The sparse dashed pattern in Figure 8.1 uses `20, 20` with the `butt` cap style. Equal intervals form a recognizable dash-gap rhythm even when the printed illustration is reduced.

8.5 EXAMPLE 7

The program uses `__svg_set_stroke_linecap()`, `__svg_set_stroke_linejoin()`, and `__svg_set_stroke_dasharray()`. Only the styling of four segments changes: their placement and direction follow the same coordinate rules throughout.

butt



round



square



dasharray



fill

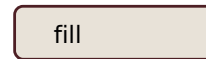


Figure 8.1. Line caps, a dashed stroke, and a filled object.

```

/*****
  SVG C BOOK EXAMPLE 007: STROKE AND FILL

  The application determines geometry; SVG C emits readable SVG.
  *****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_label( struct __svg_scene *scene, double x, double y,
          const char *caption )
{
    return( append( scene, __svg_new_text( x, y, caption, "DejaVu Sans", 17 ),
                  "none", "none", 0.0 ) );
}

static int
add_line_style( struct __svg_scene *scene, double y,
               const char *cap, const char *dash )
{

```

```

    struct __svg_node *line = __svg_new_line( 90, y, 650, y );

    if( !line ) return( 0 );
    (void)__svg_set_stroke( line, "#4D1F24" );
    (void)__svg_set_stroke_width( line, 12.0 );
    (void)__svg_set_stroke_linecap( line, cap );
    (void)__svg_set_stroke_dasharray( line, dash );
    return( append( scene, line, NULL, "none", 0.0 ) );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    struct __svg_node *box;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 760, 500, 18, 18, 18, 18 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 760, 500, 0, 0 ),
                "none", "white", 0.0 ) ) goto done;

    if( !add_label( scene, 60, 65, "butt" ) ) goto done;
    if( !add_label( scene, 60, 160, "round" ) ) goto done;
    if( !add_label( scene, 60, 255, "square" ) ) goto done;
    if( !add_label( scene, 60, 352, "dasharray" ) ) goto done;
    if( !add_line_style( scene, 110, LINECAP_BUTT, "none" ) ) goto done;
    if( !add_line_style( scene, 205, LINECAP_ROUND, "none" ) ) goto done;
    if( !add_line_style( scene, 300, LINECAP_SQUARE, "none" ) ) goto done;
    if( !add_line_style( scene, 396, LINECAP_BUTT, "20,20" ) ) goto done;

    box = __svg_new_rect( 502, 430, 145, 38, 6, 6 );
    if( !box ) goto done;
    (void)__svg_set_stroke_linejoin( box, LINEJOIN_ROUND );
    if( !append( scene, box, "#4D1F24", "#E8E2D8", 2.0 ) ) goto done;
    if( !add_label( scene, 530, 456, "fill" ) ) goto done;

    __svg_paint_scene( scene, 0, "styles.svg" );
    status = 0;
done:
    if( status ) fprintf( stderr, "Cannot construct style examples\n" );
    __svg_free_scene( scene );
    return( status );
}

```

```

cd svgc-examples/007-styles
make
# open view.html in a browser

```

The mathematical algorithm determines an object's geometry first. Its styling then helps the reader interpret that geometry without changing the computed data.

9. Compound SVG Outlines: Path Commands

9.1 A PATH AS A SEQUENCE OF GEOMETRIC OPERATIONS

In many scientific illustrations a single line or rectangle is no longer enough. A common boundary must be constructed from a sequence of operations: start an outline, move to a vertex, draw a segment, bend the line, and close the shape. SVG expresses such sequences with the path element and its `d` attribute (**path data** (*data describing a path*)).

A path need not be closed. One element may contain multiple **subpaths**, each starting with its own move command. Compound paths preserve the logical unity of an algorithmically generated object without manually joining many separate elements.

9.2 MOVES, LINES, AND CLOSURE

`M` starts a new subpath at a specified point; `L` draws a straight line to the next point. Commands `H` and `V` draw horizontal and vertical segments, respectively. `Z` connects the current point to the subpath's starting point and closes it. Thus `Z` has a specific geometric meaning: it does not merely advance to the next object.

SVGC provides `__svg_add_moveto_cmd()`, `__svg_add_lineto_cmd()`, `__svg_add_hlineto_cmd()`, `__svg_add_vlineto_cmd()`, and `__svg_add_closepath_cmd()`. A command list is created with `__svg_new_cmds()`, and the path node itself with `__svg_new_path()`.

9.3 ABSOLUTE AND RELATIVE COORDINATES

Uppercase `M`, `L`, `H`, and `V` specify absolute scene coordinates. Lowercase `m`, `l`, `h`, and `v` give offsets relative to the current point. The SVGC API selects these modes with `ABSOLUTE` and `RELATIVE`.

For example, after `M 350,170,h 155` moves to (505, 170), and the following `v 145` reaches (505, 315). The same outline can be built using absolute coordinates alone. Mixing both modes without carefully tracking the current point, however, risks accumulating errors through the sequence.

9.4 OUTLINES, FILLS, AND TRAVERSAL ORDER

Closing a path makes its boundary suitable for filling. The fill itself, however, is determined by styling rather than by `Z`. Under SVG rules, an open path may also appear filled. For a shape whose boundary is geometrically closed, explicit closure is clearer.

Boundary traversal order is especially important for compound paths with several components or holes. Such cases require understanding `fill-rule` and path orientation. Our introductory example avoids that complexity: each path contains one simple closed outline.

9.5 EXAMPLE 8

Both shapes are built with the same library functions. Only the `ABSOLUTE` and `RELATIVE` settings differ. The resulting SVG commands can be inspected in a text editor and compared directly with the C source.

M H V L Z (absolute)

M h v l Z (relative)

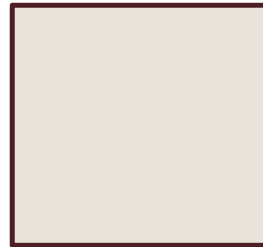
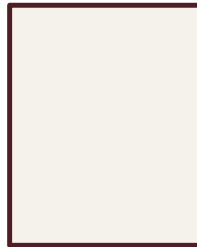


Figure 9.1. Two outlines constructed on the same principle with absolute and relative SVG path commands.

```

/*****
  SVG C BOOK EXAMPLE 008: PATH COMMANDS M L H V Z

  The application determines geometry; SVG C emits readable SVG.
  *****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_label( struct __svg_scene *scene, double x, double y,
           const char *caption )
{
    return( append( scene, __svg_new_text( x, y, caption, "DejaVu Sans", 17 ),
                   "none", "none", 0.0 ) );
}

static int
add_path( struct __svg_scene *scene, struct __svg_cmds *cmds,
          const char *stroke, const char *fill, double width )
{
    struct __svg_node *node;

```

```

    if( !cmds ) return( 0 );
    node = __svg_new_path( cmds );
    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( append( scene, node, stroke, fill, width ) );
}

static int
add_abs_path( struct __svg_scene *scene )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, 105, 170 ) < 1 ||
        __svg_add_hlineto_cmd( cmds, ABSOLUTE, 220 ) < 1 ||
        __svg_add_vlineto_cmd( cmds, ABSOLUTE, 315 ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE, 105, 315 ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "#F5F2EC", 3.0 ) );
}

static int
add_rel_path( struct __svg_scene *scene )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, 350, 170 ) < 1 ||
        __svg_add_hlineto_cmd( cmds, RELATIVE, 155 ) < 1 ||
        __svg_add_vlineto_cmd( cmds, RELATIVE, 145 ) < 1 ||
        __svg_add_lineto_cmd( cmds, RELATIVE, -155, 0 ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "#E8E2D8", 3.0 ) );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 640, 390, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 640, 390, 0, 0 ),
                "none", "white", 0.0 ) ) goto done;
}

```

```
if( !add_label( scene, 102, 110, "M H V L Z (absolute)" ) ) goto done;
if( !add_label( scene, 353, 110, "M h v l Z (relative)" ) ) goto done;
if( !add_abs_path( scene ) ) goto done;
if( !add_rel_path( scene ) ) goto done;

__svg_paint_scene( scene, 0, "path.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct path example\n" );
__svg_free_scene( scene );
return( status );
}
```

```
cd svgc-examples/008-path
make
# open view.html in a browser
```

A path records an algorithm for traversing a geometric boundary. The library should not guess the order of vertices or the mathematical meaning of the enclosed region.

10. Quadratic and Cubic Bézier Curves

10.1 FROM POLYLINES TO SMOOTH CURVES

A polyline joins adjacent points with straight segments. For plots, trajectories, and smooth technical outlines, this is sometimes insufficient: both tangent direction and curvature must be controlled. Bézier curves describe a smooth bend using endpoints and auxiliary **control points**.

Control points are not the same as measured data points. They usually do not lie on the curve, but they determine its shape and tangents. Computing suitable control points from numerical data is therefore an application-level task, not an obligation of SVGC.

10.2 THE QUADRATIC CURVE

A quadratic Bézier curve is defined by its starting point P_0 , control point P_1 , and endpoint P_2 . For $0 \leq t \leq 1$, its parametric equation is

$$B_2(t) = (1-t)^2 P_0 + 2(1-t)t P_1 + t^2 P_2.$$

Its initial tangent is directed along $P_1 - P_0$, and its final tangent along $P_2 - P_1$. SVG uses Q for this curve. The T command continues a quadratic curve by reflecting the previous control point across the shared endpoint, provided the preceding command was Q or T.

SVGC exposes these operations as `__svg_add_quadro_curveto_cmd()` and `__svg_add_short_quadro_curveto_cmd()`. They are useful for simple smooth contours and sequences of arc-like segments.

10.3 THE CUBIC CURVE

A cubic Bézier curve has two control points P_1 and P_2 , located between the start P_0 and end P_3 :

$$B_3(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3.$$

The endpoint derivatives are

$$B'_3(0) = 3(P_1 - P_0), \quad B'_3(1) = 3(P_3 - P_2).$$

Thus the control points determine both the amount of bending and the tangent directions.

SVG describes a cubic segment with C. The S continuation reconstructs its first control point by reflecting the second control point of the preceding cubic segment. The matching SVGC functions are `__svg_add_curveto_cmd()` and `__svg_add_short_curveto_cmd()`.

10.4 THE GEOMETRY OF CONTROL POINTS

The illustration shows a quadratic and a cubic curve. For the quadratic curve, thin dashed segments connect the endpoints to the control point. For the cubic curve, thin solid segments show the endpoint tangent directions, while a dashed segment connects the two control points. These auxiliary lines are separate SVG objects: they explain the geometry but are not part of the curves themselves.

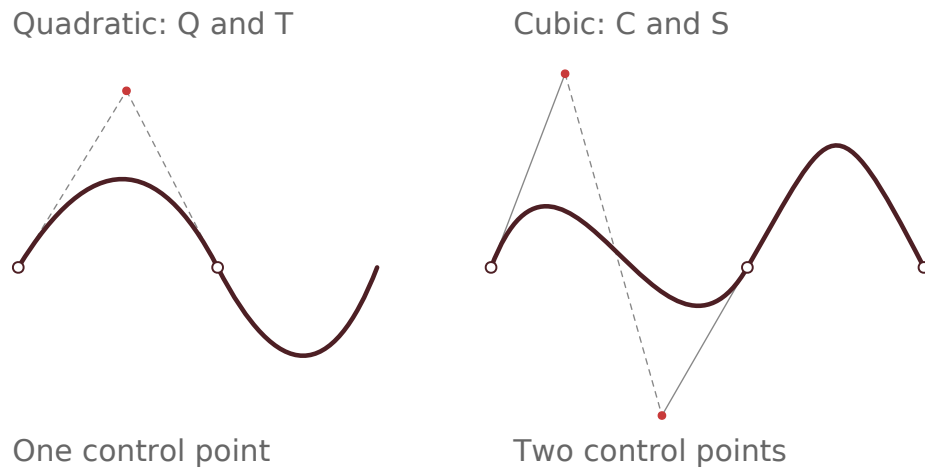


Figure 10.1. Two classes of Bézier curves: quadratic with one control point, and cubic with two.

10.5 SMOOTHING DISCRETE DATA IS NOT GUESSWORK

When points come from measurements, numerical integration, or a parametric calculation, the choice of control points needs a sound basis. Merely pursuing a visually pleasing line can create false extrema and oscillations not present in the original data.

A later chapter examines cubic interpolation splines in detail: first a system of equations determines compatible derivatives, then the derivatives are converted into the Bézier control points of individual segments, and only then does the library record the curve. This preserves the rule: compute first, draw afterward.

10.6 EXAMPLE 9

```

/*****
SVGC BOOK EXAMPLE 009: QUADRATIC AND CUBIC BEZIER CURVES

The application determines geometry; SVGC emits readable SVG.
*****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_label( struct __svg_scene *scene, double x, double y,

```

```

        const char *caption )
    {
        struct __svg_node *text;

        text = __svg_new_text( x, y, caption, "DejaVu Sans", 26 );
        if( !text ) return( 0 );
        (void)__svg_set_text_fill( text, "#666666" );
        return( append( scene, text, "none", "none", 0.0 ) );
    }

static int
add_path( struct __svg_scene *scene, struct __svg_cmds *cmds,
          const char *stroke, const char *fill, double width )
{
    struct __svg_node *node;

    if( !cmds ) return( 0 );
    node = __svg_new_path( cmds );
    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( append( scene, node, stroke, fill, width ) );
}

static int
add_quadratic( struct __svg_scene *scene )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, 75, 270 ) < 1 ||
        __svg_add_quadro_curveto_cmd( cmds, ABSOLUTE, 170, 115, 250, 270 ) < 1 ||
        __svg_add_short_quadro_curveto_cmd( cmds, ABSOLUTE, 390, 270 ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "none", 4.0 ) );
}

static int
add_cubic( struct __svg_scene *scene )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, 490, 270 ) < 1 ||
        __svg_add_curveto_cmd( cmds, ABSOLUTE,
                               555, 100, 640, 400, 715, 270 ) < 1 ||
        __svg_add_short_curveto_cmd( cmds, ABSOLUTE,
                                     790, 115, 870, 270 ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "none", 4.0 ) );
}

```

```

static int
add_hollow_point( struct __svg_scene *scene, double x, double y )
{
    struct __svg_node *circle = __svg_new_circle( x, y, 4.7 );

    if( !circle ) return( 0 );
    return( append( scene, circle, "#4D1F24", "white", 1.7 ) );
}

static int
add_helper_line( struct __svg_scene *scene,
                 double x1, double y1, double x2, double y2,
                 int dashed )
{
    struct __svg_node *line = __svg_new_line( x1, y1, x2, y2 );

    if( !line ) return( 0 );
    if( dashed ) (void)__svg_set_stroke_dasharray( line, "5,5" );
    return( append( scene, line, "#888888", "none", 1.2 ) );
}

static int
add_control_point( struct __svg_scene *scene, double x, double y )
{
    struct __svg_node *circle = __svg_new_circle( x, y, 3.2 );

    if( !circle ) return( 0 );
    return( append( scene, circle, "#C83A3A", "#C83A3A", 1.2 ) );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 930, 485, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 930, 485, 0, 0 ),
                "none", "white", 0.0 ) ) goto done;
    if( !add_label( scene, 70, 65, "Quadratic: Q and T" ) ) goto done;
    if( !add_label( scene, 510, 65, "Cubic: C and S" ) ) goto done;

    if( !add_helper_line( scene, 75, 270, 170, 115, 1 ) ) goto done;
    if( !add_helper_line( scene, 170, 115, 250, 270, 1 ) ) goto done;
    if( !add_helper_line( scene, 490, 270, 555, 100, 0 ) ) goto done;
    if( !add_helper_line( scene, 555, 100, 640, 400, 1 ) ) goto done;
    if( !add_helper_line( scene, 640, 400, 715, 270, 0 ) ) goto done;
    if( !add_quadratic( scene ) ) goto done;
    if( !add_cubic( scene ) ) goto done;

    if( !add_hollow_point( scene, 75, 270 ) ) goto done;
    if( !add_hollow_point( scene, 250, 270 ) ) goto done;

    if( !add_hollow_point( scene, 490, 270 ) ) goto done;
    if( !add_hollow_point( scene, 715, 270 ) ) goto done;

```

```
if( !add_hollow_point( scene, 870, 270 ) ) goto done;

if( !add_control_point( scene, 170, 115 ) ) goto done;
if( !add_control_point( scene, 555, 100 ) ) goto done;
if( !add_control_point( scene, 640, 400 ) ) goto done;
if( !add_label( scene, 70, 440, "One control point" ) ) goto done;
if( !add_label( scene, 510, 440, "Two control points" ) ) goto done;

__svg_paint_scene( scene, 0, "bezier.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct Bezier example\n" );
__svg_free_scene( scene );
return( status );
}
```

```
cd svgc-examples/009-bezier
make
# open view.html in a browser
```

Curve smoothness follows from the mathematical choice of control points. SVGC faithfully records those points in an SVG document.

11. Elliptical Arcs, Chords, and Sectors

11.1 ELLIPTICAL ARCS AS PATH ELEMENTS

A mathematical circle or ellipse defines a closed boundary. An engineering drawing often needs only part of it: a rotation angle, a sector, a region of action, or a connecting arc. SVG represents such portions with the `A` command inside a path.

Unlike a Bézier curve, an SVG elliptical arc is defined by the ellipse radii, rotation of its axes, and the selection of one of the possible arcs between the endpoints, rather than by control points. `__svg_add_arc_cmd()` accepts `rx`, `ry`, `x_axis_rotation`, `large_arc_flag`, `sweep_flag`, and the endpoint coordinates.

11.2 ARC SELECTION FLAGS

Given fixed endpoints and admissible radii, there are generally four possible arcs. The `large-arc-flag` chooses between the shorter and longer arc; `sweep-flag` selects traversal direction according to SVG screen coordinates. Each flag must be 0 or 1.

These values should not be equated with clockwise and counterclockwise motion without specifying the coordinate convention. SVG's y axis points downward, so visual orientation differs from a Cartesian mathematical plane. If the given radii are too small to connect the endpoints, SVG normalizes them when calculating the ellipse. This matters when preparing geometrically exact scientific drawings.

11.3 ARC, CHORD, AND SECTOR

Let an arc have endpoints A and B . A **chord** is the straight segment AB . The closed region bounded by the arc and chord can be built with `M A`, followed by `A . . . B`, followed by `Z`. The final command adds the straight closing edge to A .

A **sector** is bounded by an arc and two segments from its endpoints to the center O . A simple sector uses `M O, L A, A . . . B, Z`. An elliptical sector uses two semiaxes instead of one radius, with O denoting the ellipse center.

Arc: A (0 0)

Chord: A Z

Sector: L A Z



Figure 11.1. An open arc, the region between an arc and its chord, and an elliptical sector.

11.4 EXAMPLE 10

The program's three functions use the same path-construction mechanism but different command sequences: A alone; A followed by closure; and a move from the center to the arc, followed by A and closure. This composition remains application logic rather than becoming special SVGC primitives.

```

/*****
SVGC BOOK EXAMPLE 010: ELLIPTIC ARCS, CHORDS AND SECTORS

The application determines geometry; SVGC emits readable SVG.
*****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_label( struct __svg_scene *scene, double x, double y,
           const char *caption )
{
    return( append( scene, __svg_new_text( x, y, caption, "DejaVu Sans", 17 ),
                   "none", "none", 0.0 ) );
}

static int
add_path( struct __svg_scene *scene, struct __svg_cmds *cmds,
          const char *stroke, const char *fill, double width )
{
    struct __svg_node *node;

    if( !cmds ) return( 0 );
    node = __svg_new_path( cmds );
    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( append( scene, node, stroke, fill, width ) );
}

static int
add_arc_only( struct __svg_scene *scene, double cx, double cy,
              double r, int large_arc, int sweep )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, cx + r, cy ) < 1 ||
        __svg_add_arc_cmd( cmds, ABSOLUTE, r, r, 0,

```

```

        large_arc, sweep, cx, cy - r ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "none", 3.0 ) );
}

static int
add_chord( struct __svg_scene *scene, double cx, double cy,
           double r )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, cx + r, cy ) < 1 ||
        __svg_add_arc_cmd( cmds, ABSOLUTE, r, r, 0, 0, 0,
                           cx, cy - r ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "#E8E2D8", 2.5 ) );
}

static int
add_sector( struct __svg_scene *scene, double cx, double cy,
            double rx, double ry )
{
    struct __svg_cmds *cmds = __svg_new_cmds();

    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE, cx, cy ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE, cx + rx, cy ) < 1 ||
        __svg_add_arc_cmd( cmds, ABSOLUTE, rx, ry, 0, 0, 1,
                           cx, cy + ry ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    return( add_path( scene, cmds, "#4D1F24", "#F5F2EC", 2.5 ) );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 850, 410, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 850, 410, 0, 0 ),
                "none", "white", 0.0 ) ) goto done;
}

```

```
if( !add_label( scene, 65, 65, "Arc: A (0 0)" ) ) goto done;
if( !add_label( scene, 310, 65, "Chord: A Z" ) ) goto done;
if( !add_label( scene, 600, 65, "Sector: L A Z" ) ) goto done;
if( !add_arc_only( scene, 135, 250, 100, 0, 0 ) ) goto done;
if( !add_chord( scene, 400, 250, 100 ) ) goto done;
if( !add_sector( scene, 705, 205, 105, 90 ) ) goto done;

__svg_paint_scene( scene, 0, "arcs.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct arc examples\n" );
__svg_free_scene( scene );
return( status );
}
```

```
cd svgc-examples/010-arcs
make
# open view.html in a browser
```

For exact drawings, distinguish the mathematical reference geometry (arc parameters in the application) from its approximation in print. A printed illustration must not silently change the meaning of the original data.

Arcs, chords, and sectors are geometric constructions. The program chooses endpoints, radii, direction, and connection order; SVGC preserves the result as a readable SVG path.

12. Labels, Fonts, and Text Coordinates

12.1 TEXT IS A GRAPHICAL OBJECT

In a scientific illustration, a label is both content and part of the geometric composition. Its characters are stored in an SVG text node, but their appearance depends on the font family, size, and rendering engine. In C, a label is created with `__svg_new_text(x, y, text, font_family, font_size)`. The coordinates x and y specify the start of the text string on its **baseline**, not the upper-left corner of the rectangle bounding the glyphs.

To compare several typefaces, Figure 12.1 shows faint guides along their baselines. Descenders such as those in *g* and *y* make the difference easy to see: these shapes extend below the baseline even though the string's y coordinate remains constant.

12.2 FONT FAMILY, SIZE, AND COLOR

`__svg_set_text_font_family()`, `__svg_set_text_font_size()`, and `__svg_set_text_fill_rgb()` can change the attributes of an existing text node. Our example uses DejaVu Sans, DejaVu Serif, and DejaVu Sans Mono. Appearance on another computer depends on the installed fonts and substitution rules. Specifying a font family in SVG does not embed the font file in the image.

12.3 WHY CHARACTER COUNTS CANNOT DETERMINE TEXT WIDTH

A string's width depends on actual glyph metrics, spaces, kerning, and font settings. Even with the same number of characters, the strings *iiii* and *www* have noticeably different widths in a proportional typeface. Providing coordinates and a font name to SVGC does not mean the library can measure a text run as it will be rendered on the target platform.

The label in the lower rectangle of the illustration is therefore positioned manually after viewing the generated SVG. This layout decision belongs to the application, not to the library's mechanism for writing SVG nodes. A change of font, scale, or wording may require updated coordinates.

12.4 EXAMPLE 11

The C program creates a sheet, three guides, several text nodes, and an area containing a manually aligned label. No additional engine for complex font geometry is used.

```

/*****
SVGC BOOK EXAMPLE 011: TEXT AND FONT PRESENTATION

Text location is chosen by the application, without measuring glyphs.
*****/

#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,

```

Sans: scientific illustration

Serif: scientific illustration

Mono: 0123456789 ABC xyz

Manual alignment
W and i have different widths

Figure 12.1. Text baselines and visible differences between typefaces and labels.

```

const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_text( struct __svg_scene *scene, double x, double y,
          const char *caption, const char *family, int size,
          int red, int green, int blue )
{
    struct __svg_node *node;

    node = __svg_new_text( x, y, caption, family, size );
    if( !node ) return( 0 );
    (void)__svg_set_text_fill_rgb( node, red, green, blue );
    return( append( scene, node, NULL, NULL, 0.0 ) );
}

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 760, 460, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );

    scene = __svg_new_scene( sheet );
    if( !scene )
    {

```

```

__svg_free_node( (struct __svg_node *)sheet );
return( 1 );
}

if( !append( scene, __svg_new_rect( 0, 0, 760, 460, 0, 0 ),
    "none", "white", 0.0 ) ) goto done;

/* The horizontal guides pass through each text baseline. */
if( !append( scene, __svg_new_line( 40, 102, 720, 102 ),
    "#CACACA", "none", 1.0 ) ) goto done;
if( !append( scene, __svg_new_line( 40, 194, 720, 194 ),
    "#CACACA", "none", 1.0 ) ) goto done;
if( !append( scene, __svg_new_line( 40, 286, 720, 286 ),
    "#CACACA", "none", 1.0 ) ) goto done;

if( !add_text( scene, 55, 102, "Sans: scientific illustration",
    "DejaVu Sans", 23, 26, 41, 65 ) ) goto done;
if( !add_text( scene, 55, 194, "Serif: scientific illustration",
    "DejaVu Serif", 23, 26, 41, 65 ) ) goto done;
if( !add_text( scene, 55, 286, "Mono: 0123456789 ABC xyz",
    "DejaVu Sans Mono", 22, 26, 41, 65 ) ) goto done;

if( !append( scene, __svg_new_rect( 45, 334, 666, 86, 8, 8 ),
    "#4D1F24", "#F4F0EA", 2.0 ) ) goto done;
/* These two positions were adjusted after SVG visual inspection. */
if( !add_text( scene, 257, 370, "Manual alignment",
    "DejaVu Sans", 21, 77, 31, 36 ) ) goto done;
if( !add_text( scene, 224, 399, "W and i have different widths",
    "DejaVu Sans", 17, 26, 41, 65 ) ) goto done;

__svg_paint_scene( scene, 0, "text.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct text example\n" );
__svg_free_scene( scene );
return( status );
}

```

```

cd svgc-examples/011-text
make
# open view.html in a browser

```

12.5 ESCAPING TEXT IN XML

Characters such as < and & cannot be inserted unescaped into XML text: they require < and &. When strings come from files, user input, or a numerical experiment, the application must ensure correct escaping and valid character encoding. Creating text should not silently become font parsing or arbitrary string conversion.

12.6 EXTERNAL FONTS AND CSS DEFINITIONS INSIDE SVG

Naming a font family does not ensure that the reader's computer has that font installed. SVG permits CSS rules inside a `style` element in the `defs` section. For example, `@font-face` can describe a font family, style, and external woff2 files. Two operations must be distinguished: *embedding CSS text* in SVG and *embedding the binary font data itself*. With the first approach, the font remains an external resource referenced by `url( . . . )`.

SVGC provides `__svg_sheet_add_css_file( sheet, fname )`. The `fname` argument names an ordinary CSS

file containing style rules only, without enclosing XML, defs, style, or CDATA markup. The library supplies the required wrapping and stores the file contents in the selected sheet's definitions. Always check the result: 1 indicates success, 0 indicates a read or content error, and -1 indicates invalid arguments.

```
/* The working directory is src/svgc-examples/013-control-system/. */
if( __svg_sheet_add_css_file( sheet, "fonts.css" ) != 1 )
{
    fprintf( stderr, "Cannot attach fonts.css\\n" );
    return( 1 );
}
```

12.7 A SHARED FONT DIRECTORY FOR EXAMPLES

The separately distributed examples archive contains a common `svgc-examples/fonts/` directory. Programs that require special fonts can share it rather than duplicating the same files in every example. CSS rules remain alongside the program producing the corresponding SVG and can be included directly in the document. Relative `url( ... )` paths inside embedded CSS are resolved relative to the *generated SVG file*, not to the source CSS file.

12.8 SUBSCRIPTS AND SUPERSCRIPTS WITH TSPAN

Correct mathematical typography is essential in Figure 14.1: subscripts in M_z , I_γ , and k_θ must not be replaced with ordinary character sequences. An SVG text element can contain child `tspan` elements, which control properties of fragments, including `font-size`, `font-style`, a relative `dy` shift, or `baseline-shift`.

For example, the string passed to `__svg_new_text()` may contain the following XML fragment:

```
<tspan font-style="italic">M<tspan dy="4" font-size="12">z</tspan></tspan>
```

The SVG element is `tspan`, not the HTML element `span`. Unlike an external font file, this fragment is stored directly in XML. When the application passes markup to SVGC, it is responsible for XML correctness. The SVG viewer computes the actual text metrics from the chosen font; SVGC does not model them. When preparing a printed PDF, verify that subscripts retain their positions and sizes.

Greek Letters in SVG. Greek characters can be written directly as UTF-8 in mathematical labels. For example, the mechanical pendulum in Figure 16.1 draws φ as follows:

```
text = (struct __svg_text *)__svg_new_text( x0+10*OSC_SCALE, y0+r9+10*OSC_SCALE,
"<tspan font-family=\"Georgia\" font-style=\"oblique\">\varphi</tspan>",
"Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );
```

This preserves both the character and its styling in SVG. When preparing a PDF with CairoSVG, confirm that the chosen font contains the glyph and that the output is displayed correctly. The printed image is generated directly from SVG without rewriting Greek letters.

The program positions a label in its coordinate system, while the visible width and height of the glyphs depend on the font and rendering device. SVG preserves and renders text; it does not make text metrics independent of the font.

12.9 FONT SIZES AND FAMILIES WHEN BRINGING SVG INTO TeX

To preserve font families and sizes accurately, generate a vector PDF directly from SVG using CairoSVG:

```
cairosvg -o figures/011-text.pdf svgc-examples/011-text/text.svg
```

Insert the resulting file into TeX with the usual `\includegraphics` command, specifying the desired width:

```
\includegraphics[width=0.88\linewidth]{figures/011-text.pdf}
```

This workflow is particularly useful because converters that turn SVG into native TeX drawing commands do not always reproduce every text attribute. For example, `svg2tikz` may omit `font-size` and embedded CSS font definitions. Older Inkscape versions and certain workflows can also switch between Cartesian coordinates with an upward Y axis (origin at bottom left) and normal SVG coordinates (usually origin at top left). Such changes sometimes appear as transformation chains like `transform="scale(1,-1) translate(...)"`. Older tools such as `Inkscape2TikZ` may struggle with nested transformations or complex CSS styles.

CairoSVG, in contrast, processes the original SVG geometry, vector curves, `font-size` attributes, and coordinate system directly. If the necessary fonts are available, the PDF retains vector graphics and label appearance. `\includegraphics` scales the figure without rasterizing it. The finished PDF should still be checked for font embedding and correct appearance.

Embedded CSS rules or JavaScript, for example those added with `__svg_sheet_add_css_file()` or `__svg_sheet_add_script_file()`, are not always supported by CairoSVG. Processing `style` or `script` elements may cause conversion to fail. Such elements can be temporarily removed from a *copy* of the SVG before export, leaving the original unchanged. If the source itself was edited, restore the elements afterward. Always inspect the resulting figure: removing CSS may change its appearance.

13. Linear and Radial Gradients

13.1 DEFINITIONS AND REFERENCES

A uniform fill has one color. A **gradient** varies color continuously across an object's interior. In SVG, a gradient is defined separately inside defs; the object that uses it refers to that definition with `fill="url(##identifier)"`.

SVGC creates linear gradients with `__svg_new_l_gradient()` and radial ones with `__svg_new_r_gradient()`. Definitions are attached to the sheet with `__svg_sheet_add_definition()`. Color stops are constructed using `__svg_new_stop()` and added through `__svg_gradient_add_stop()`. The `offset` attribute specifies a stop's relative position.

13.2 LINEAR COLOR INTERPOLATION

For two stops at relative positions 0 and 1, each color channel can be interpolated linearly:

$$C(t) = (1 - t)C_0 + tC_1, \quad 0 \leq t \leq 1.$$

The channels can be ordinary RGB components. In our illustration, the first gradient runs from left to right, making the effect visible across a wide rectangle. The `objectBoundingBox` coordinate system relates the gradient to the filled object's geometry.

13.3 RADIAL GRADIENTS

Radial color variation depends on distance from the focal point and the boundary of the gradient region. In the simple symmetric case, the focus and center coincide; `cx`, `cy`, `r`, `fx`, and `fy` describe this geometry. On a rectangular object, a radial distribution in `objectBoundingBox` coordinates may appear elliptical because its sides have different lengths.

linearGradient



radialGradient



Figure 13.1. Linear and radial fills using references to SVG definitions.

13.4 EXAMPLE 12

The program creates two gradient definitions, each with two color stops. Once a definition is registered, `__svg_create_node_url()` returns a reference that can be passed to `__svg_set_fill()`. The application frees the returned string after setting the style. No extension of the public SVGC interface is needed.

```

/*****
SVGC BOOK EXAMPLE 012: LINEAR AND RADIAL GRADIENTS

Gradient definitions live in sheet defs, not in drawing coordinates.
*****/

#include <stdio.h>
#include <stdlib.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_stop( void *gradient, double offset, int r, int g, int b )
{
    struct __svg_stop *stop;

    stop = (struct __svg_stop *)__svg_new_stop( offset );
    if( !stop ) return( 0 );
    (void)__svg_set_stop_color_rgb( (struct __svg_node *)stop, r, g, b );
    (void)__svg_set_stop_opacity( (struct __svg_node *)stop, 1.0 );
    if( __svg_gradient_add_stop( gradient, stop ) == 1 ) return( 1 );
    __svg_free_node( (struct __svg_node *)stop );
    return( 0 );
}

static int
add_gradient_rect( struct __svg_scene *scene,
                  struct __svg_node *gradient,
                  double x, double y, double w, double h )
{
    struct __svg_node *rect;
    char *url;
    int ok;

    url = __svg_create_node_url( gradient );
    if( !url ) return( 0 );
    rect = __svg_new_rect( x, y, w, h, 12, 12 );
    ok = append( scene, rect, "#4D1F24", url, 3.0 );
    free( url );
    return( ok );
}

```

```

int
main( void )
{
    struct __svg_sheet      *sheet;
    struct __svg_scene      *scene;
    struct __svg_l_gradient *linear;
    struct __svg_r_gradient *radial;
    int                     status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 780, 385, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );

    linear = (struct __svg_l_gradient *)__svg_new_l_gradient(
        "objectBoundingBox", "pad", 0.0, 0.0, 1.0, 0.0 );
    if( !linear ) return( 1 );
    if( !add_stop( linear, 0.0, 225, 239, 251 ) ||
        !add_stop( linear, 1.0, 44, 89, 142 ) ) return( 1 );
    (void)__svg_sheet_add_definition( sheet, (struct __svg_node *)linear );

    radial = (struct __svg_r_gradient *)__svg_new_r_gradient(
        "objectBoundingBox", "pad", 0.5, 0.5, 0.5, 0.5, 0.5 );
    if( !radial ) return( 1 );
    if( !add_stop( radial, 0.0, 255, 241, 194 ) ||
        !add_stop( radial, 1.0, 168, 63, 73 ) ) return( 1 );
    (void)__svg_sheet_add_definition( sheet, (struct __svg_node *)radial );

    scene = __svg_new_scene( sheet );
    if( !scene ) return( 1 );
    if( !append( scene, __svg_new_rect( 0, 0, 780, 385, 0, 0 ),
        "none", "white", 0.0 ) ) goto done;
    if( !append( scene, __svg_new_text( 55, 62, "linearGradient", "DejaVu Sans", 21 ),
        "none", NULL, 0.0 ) ) goto done;
    if( !append( scene, __svg_new_text( 420, 62, "radialGradient", "DejaVu Sans", 21 ),
        "none", NULL, 0.0 ) ) goto done;

    if( !add_gradient_rect( scene, (struct __svg_node *)linear,
        55, 95, 305, 230 ) ) goto done;
    if( !add_gradient_rect( scene, (struct __svg_node *)radial,
        420, 95, 305, 230 ) ) goto done;

    __svg_paint_scene( scene, 0, "gradients.svg" );
    status = 0;
done:
    if( status ) fprintf( stderr, "Cannot construct gradient example\n" );
    __svg_free_scene( scene );
    return( status );
}

```

```

cd svgc-examples/012-gradients
make
# open view.html in a browser

```

13.5 VERIFYING THE PRINT CONVERSION

After converting SVG to PDF with CairoSVG, check the gradient direction, color-stop positions, and fill boundaries. Radial gradients deserve particular care: browser and printed-PDF output should retain the same meaning and color transitions. The SVG source is the reference, and the printed result is inspected visually.

A gradient is a graphical style definition, not a method of numerical computation. Many objects can reference the same definition without repeating its stops for every object.

14. A Structural Diagram of a Gyroscopic System

14.1 A GRAPHICAL REPRESENTATION OF A MATHEMATICAL MODEL

Control-system structural diagrams differ from program flowcharts. Their rectangles represent functional transformations of signals, summing junctions represent algebraic combinations, and arrows indicate the direction in which quantities are transmitted. Inputs and outputs may denote physical variables and their derivatives.

As a complete example, consider the structural diagram published as Figure 1.6 in the “Graphical Programming” section at <https://dynamics.su/theory/issues/>. The original C program uses SVGC to draw the structure of an active gyroscopic system.

14.2 DIFFERENTIAL MODEL OF THE ANGULAR STABILIZATION SYSTEM

Figure 14.1 depicts angular stabilization about the pitch axis using gyroscopic control actuators. The corresponding differential equations are published as system (1.9) at <https://dynamics.su/theory/issues/>:

$$\left. \begin{aligned} I_z \ddot{\theta} + 2H\dot{\beta} &= M_z, \\ I_\gamma \ddot{\beta} + \mu\dot{\beta} - H\dot{\theta} &= M_\beta. \end{aligned} \right\}$$

Here $M_\beta = k_\theta \theta + k_{\dot{\theta}} \dot{\theta}$ is the control law.

14.3 BLOCKS, CONNECTIONS, AND TRANSFER FACTORS

Tracing the diagram from left to right, the input moment M_z passes through a summing junction to the integrating elements. Blocks containing fractions represent factors involving the moment of inertia I_z , the Laplace variable s , and the integrator $1/s$. The lower branch contains the second coordinate β , the factor $1/(I_\gamma s)$, and connections with coefficients H and μ . Feedback paths couple the angular variables.

The diagram also includes the factor $-2H$ and the operator block for the control law. These denote mathematical operations, not decorative labels. The system’s dynamics are specified by its equations; the drawing program reproduces their structure.

14.4 SUMMING JUNCTIONS, BRANCHES, AND PAINTING ORDER

Summing junctions are circles with diagonal lines. Small filled circles have radii comparable to the thickness of the connecting lines. They mark branch points, removing ambiguity at T-junctions. Every directed connection has its own arrowhead.

Creation order matters: connections and arrows first, then filled functional rectangles, and finally labels. When preparing the printed diagram, examine stroke contrast, coefficient labels, and the joining of branches. The geometry of the original approved illustration is preserved.

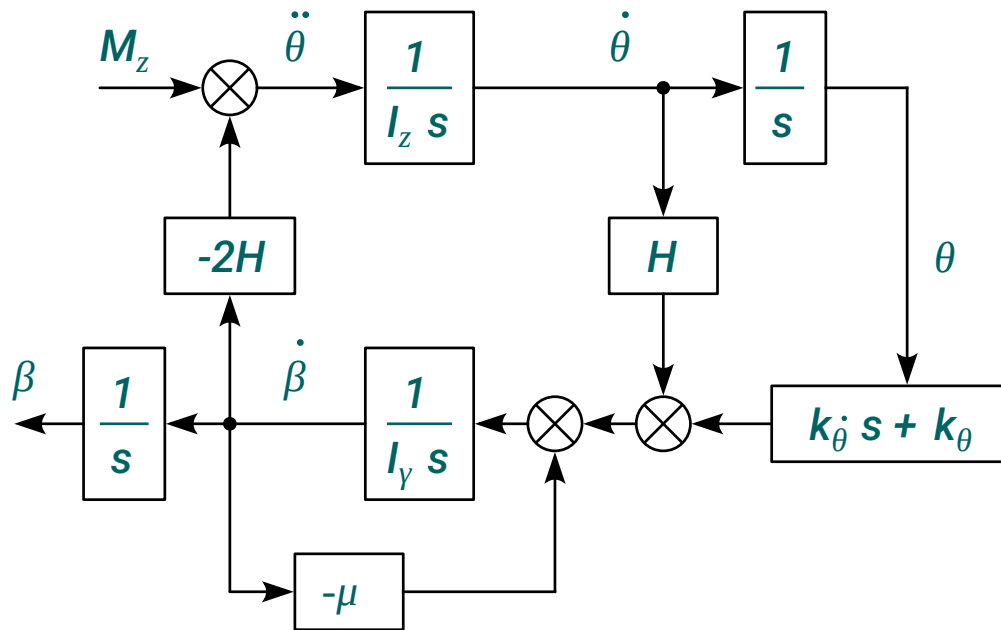


Figure 14.1. Structural diagram of an active gyroscopic system.

14.5 MATHEMATICAL LABELS AND NESTED TSPAN ELEMENTS

Signal and transfer-block labels need subscripts: M_z , I_z , I_γ , and k_θ . The C program implements them in SVG with nested `tspan` elements inside `text` nodes. Individual fragments can have their own font style, size, and baseline offset.

The exact width of these labels cannot be calculated from the length of their source strings alone. The metrics of each glyph, including spaces and reduced subscript characters, would be needed. All label coordinates are therefore determined in the application and verified visually.

14.6 ADDING CSS WITH SVGC

For Roboto and Georgia typography, the program reads a separate `fonts.css` file using `__svg_sheet_add_css_file()`. This is plain CSS text; it requires no enclosing style tags or CDATA sections. The library places the rules in the output SVG's `defs` section. Font paths in `url(...)` remain external and must be accessible relative to the generated `control-system.svg` file.

14.7 EXAMPLE 13

The listing below constructs the diagram and checks the CSS attachment. The arrangement of functional elements, arrow algorithms, original labels, and topology are unchanged. The example uses only the public SVG C interface and requires no extension of the library.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```

#include <time.h>
#include <unistd.h>

#include <math.h>

#include <svgc.h>

struct __svg_sheet    *sheet;
struct __svg_line     *line;
struct __svg_rect     *rect;
struct __svg_circle   *circle;
struct __svg_ellipse  *ellipse;
struct __svg_text     *text;

struct __svg_points *points;
struct __svg_polygon *polygon;
struct __svg_polyline *polyline;

struct __svg_cmd *cmd;
struct __svg_cmds *cmds;
struct __svg_path *path;

struct __svg_scene *scene;

void arrow( double sx0, double sy0, double f0 )
{
    double sx1, sy1, sx2, sy2, sx3, sy3, sx4, sy4;
    double px1, py1, px2, py2, px3, py3, px4, py4;

    sx1 = sx0 - 3.0;
    sy1 = sy0 + 3.0;
    sx2 = sx0;
    sy2 = sy0 - 6.0;
    sx3 = sx0 + 3.0;
    sy3 = sy0 + 3.0;
    sx4 = sx0;
    sy4 = sy0 + 1.0;

    px1 = sx0 + ( sx1 - sx0 )*cos( f0 ) - ( sy1 - sy0 )*sin( f0 );
    py1 = sy0 + ( sx1 - sx0 )*sin( f0 ) + ( sy1 - sy0 )*cos( f0 );
    px2 = sx0 + ( sx2 - sx0 )*cos( f0 ) - ( sy2 - sy0 )*sin( f0 );
    py2 = sy0 + ( sx2 - sx0 )*sin( f0 ) + ( sy2 - sy0 )*cos( f0 );
    px3 = sx0 + ( sx3 - sx0 )*cos( f0 ) - ( sy3 - sy0 )*sin( f0 );
    py3 = sy0 + ( sx3 - sx0 )*sin( f0 ) + ( sy3 - sy0 )*cos( f0 );
    px4 = sx0 + ( sx4 - sx0 )*cos( f0 ) - ( sy4 - sy0 )*sin( f0 );
    py4 = sy0 + ( sx4 - sx0 )*sin( f0 ) + ( sy4 - sy0 )*cos( f0 );

    cmds = __svg_new_cmds();
    __svg_add_moveto_cmd( cmds, ABSOLUTE, px1, py1 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, px2, py2 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, px3, py3 );
    __svg_add_short_curveto_cmd( cmds, ABSOLUTE, px4, py4, px1, py1 );
    __svg_add_closepath_cmd( cmds, ABSOLUTE );
    path = (struct __svg_path *)__svg_new_path( cmds );
    (void)__svg_set_stroke_rgb( (struct __svg_node *)path, 0, 0, 0 );
    (void)__svg_set_stroke_width( (struct __svg_node *)path, 1.0 );
    (void)__svg_set_stroke_linecap( (struct __svg_node *)path, "square" );
    (void)__svg_set_stroke_linejoin( (struct __svg_node *)path, "miter" );
    (void)__svg_set_fill_rgb( (struct __svg_node *)path, 0, 0, 0 );

    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

void small_arrow( double sx0, double sy0, double f0 )

```

```

{
    double sx1, sy1, sx2, sy2, sx3, sy3, sx4, sy4;
    double px1, py1, px2, py2, px3, py3, px4, py4;

    sx1 = sx0 - 2.0;
    sy1 = sy0 + 2.0;
    sx2 = sx0;
    sy2 = sy0 - 4.0;
    sx3 = sx0 + 2.0;
    sy3 = sy0 + 2.0;
    sx4 = sx0;
    sy4 = sy0 + 1.0;

    px1 = sx0 + ( sx1 - sx0 ) * cos( f0 ) - ( sy1 - sy0 ) * sin( f0 );
    py1 = sy0 + ( sx1 - sx0 ) * sin( f0 ) + ( sy1 - sy0 ) * cos( f0 );
    px2 = sx0 + ( sx2 - sx0 ) * cos( f0 ) - ( sy2 - sy0 ) * sin( f0 );
    py2 = sy0 + ( sx2 - sx0 ) * sin( f0 ) + ( sy2 - sy0 ) * cos( f0 );
    px3 = sx0 + ( sx3 - sx0 ) * cos( f0 ) - ( sy3 - sy0 ) * sin( f0 );
    py3 = sy0 + ( sx3 - sx0 ) * sin( f0 ) + ( sy3 - sy0 ) * cos( f0 );
    px4 = sx0 + ( sx4 - sx0 ) * cos( f0 ) - ( sy4 - sy0 ) * sin( f0 );
    py4 = sy0 + ( sx4 - sx0 ) * sin( f0 ) + ( sy4 - sy0 ) * cos( f0 );

    cmds = __svg_new_cmds();
    __svg_add_moveto_cmd( cmds, ABSOLUTE, px1, py1 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, px2, py2 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, px3, py3 );
    __svg_add_shortcurveto_cmd( cmds, ABSOLUTE, px4, py4, px1, py1 );
    __svg_add_closepath_cmd( cmds, ABSOLUTE );
    path = (struct __svg_path *)__svg_new_path( cmds );
    (void)__svg_set_stroke_rgb( (struct __svg_node *)path, 0, 0, 0 );
    (void)__svg_set_stroke_width( (struct __svg_node *)path, 0.5 );
    (void)__svg_set_stroke_linecap( (struct __svg_node *)path, "square" );
    (void)__svg_set_stroke_linejoin( (struct __svg_node *)path, "miter" );
    (void)__svg_set_fill_rgb( (struct __svg_node *)path, 255, 255, 255 );

    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

void line_arrow( double x0, double y0, double x1, double y1 )
{
    double f0;
    double l0, l2, l3, x2, y2, x3, y3;
    double x = x1 - x0, y = y1 - y0;

    f0 = atan2( y, x );
    l0 = sqrt( x*x + y*y );
    l2 = l0 - 4.0;
    l3 = l0 - 8.0;
    x2 = x0 + l2 * cos( f0 );
    y2 = y0 + l2 * sin( f0 );
    x3 = x0 + l3 * cos( f0 );
    y3 = y0 + l3 * sin( f0 );

    line = (struct __svg_line *)__svg_new_line( x0, y0, x2, y2 );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_ROUND );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    arrow( x3, y3, M_PI/2.0 + f0 );
}

void line_dash_arrow( double x0, double y0, double x1, double y1 )
{
    double f0;
    double l0, l2, l3, x2, y2, x3, y3;

```

```

double x = x1 - x0, y = y1 - y0;

f0 = atan2( y, x );
l0 = sqrt( x*x + y*y );
l2 = l0 - 4.0;
l3 = l0 - 8.0;
x2 = x0 + l2 * cos( f0 );
y2 = y0 + l2 * sin( f0 );
x3 = x0 + l3 * cos( f0 );
y3 = y0 + l3 * sin( f0 );

line = (struct __svg_line *)__svg_new_line( x0, y0, x2, y2 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_ROUND );
__svg_set_stroke_dasharray( (struct __svg_node *)line, "5,3" );

```

```

(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

arrow( x3, y3, M_PI/2.0 + f0 );
}

void script_box( double x, double y, double width, double height )
{
    double r0 = 6.0, r1 = r0 / 2.0;

    line = (struct __svg_line *)__svg_new_line( x+r0, y, x+width+r0, y );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x, y+r0, x, y+height );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x+width, y+r0, x+width, y+height+r0 );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x-r1, y+height, x+width-r0, y+height );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x-r1, y+height+2*r0, x+width-r0, y+height+2*r0 );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x+width+2*r1, y+2*r1, x+width, y+2*r1 );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x+width, y+2*r0, x+width+r0, y+2*r0 );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
    __svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_SQUARE );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    line = (struct __svg_line *)__svg_new_line( x+width-2*r1, y+height+r0, x+width-2*r0, y+height+r0 );
    __svg_set_stroke_width( (struct __svg_node *)line, 1.0 );

```

```

__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

{
    double x1 = x,    y1 = y+r0,
           x2 = x+r0, y2 = y;

    cmds = __svg_new_cmds();
    /* M x1,y1 */
    __svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

    __svg_add_arc_cmd( cmds,
                      ABSOLUTE,
                      r0, r0,    /* rx, ry */
                      0.0,      /* angle */
                      0,1,      /* lf, sf */
                      x2, y2 );

    path = (struct __svg_path *)__svg_new_path( cmds );
    __svg_set_stroke_linecap( (struct __svg_node *)path, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

{
    double x1 = x+width,    y1 = y+r0,
           x2 = x+width+r0, y2 = y+2*r0;

    cmds = __svg_new_cmds();
    /* M x1,y1 */
    __svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

    __svg_add_arc_cmd( cmds,
                      ABSOLUTE,
                      r0, r0,    /* rx, ry */
                      0.0,      /* angle */
                      1,1,      /* lf, sf */
                      x2, y2 );

    path = (struct __svg_path *)__svg_new_path( cmds );
    __svg_set_stroke_linecap( (struct __svg_node *)path, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

{
    double x1 = x+width,    y1 = y+height+r0,
           x2 = x+width-r0, y2 = y+height;

    cmds = __svg_new_cmds();
    /* M x1,y1 */
    __svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

    __svg_add_arc_cmd( cmds,
                      ABSOLUTE,
                      r0, r0,    /* rx, ry */
                      0.0,      /* angle */
                      1,1,      /* lf, sf */
                      x2, y2 );

    path = (struct __svg_path *)__svg_new_path( cmds );
    __svg_set_stroke_linecap( (struct __svg_node *)path, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

{
    double x1 = x-r1, y1 = y+height,
           x2 = x-r1, y2 = y+height+2*r0;

    cmds = __svg_new_cmds();

```

```

/* M x1,y1 */
__svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

__svg_add_arc_cmd( cmds,
    ABSOLUTE,
    r0, r0,      /* rx, ry */
    0.0,         /* angle */
    1,0,         /* lf, sf */
    x2, y2 );

path = (struct __svg_path *)__svg_new_path( cmds );
__svg_set_stroke_linecap( (struct __svg_node *)path, LINECAP_BUTT );
(void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

{
    double x1 = x+width+2*r1, y1 = y+2*r1,
           x2 = x+width+2*r1, y2 = y+4*r1;

    cmds = __svg_new_cmds();
    /* M x1,y1 */
    __svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

    __svg_add_arc_cmd( cmds,
        ABSOLUTE,
        r1, r1,      /* rx, ry */
        0.0,         /* angle */
        1,0,         /* lf, sf */
        x2, y2 );

    path = (struct __svg_path *)__svg_new_path( cmds );
    __svg_set_stroke_linecap( (struct __svg_node *)path, LINECAP_BUTT );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

{
    double x1 = x+width-2*r1, y1 = y+height,
           x2 = x+width-2*r1, y2 = y+height+2*r1;

```

```

cmds = __svg_new_cmds();
/* M x1,y1 */
__svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

__svg_add_arc_cmd( cmds,
    ABSOLUTE,
    r1, r1,      /* rx, ry */
    0.0,         /* angle */
    0,1,         /* lf, sf */
    x2, y2 );

path = (struct __svg_path *)__svg_new_path( cmds );
__svg_set_stroke_linecap( (struct __svg_node *)path, LINECAP_BUTT );
(void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

}

void circle_s( double x, double y, double r )
{
    double p, x1, y1, x2, y2, x3, y3, x4, y4;

    p = r*sqrt(2.0)/2;

    x1 = x - p;
    y1 = y - p;

```

```

x2 = x + p;
y2 = y - p;
x3 = x + p;
y3 = y + p;
x4 = x - p;
y4 = y + p;

circle = (struct __svg_circle *)__svg_new_circle( x, y, r );
__svg_set_stroke_width( (struct __svg_node *)circle, 1.0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

line = (struct __svg_line *)__svg_new_line( x1, y1, x3, y3 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x2, y2, x4, y4 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

}

int main( void )
{
    double x0 = 16.0,
           y0 = 16.0;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 400, 264, 16, 16, 16, 16 );
    if( !sheet )
    {
        printf( "Unable to create sheet\n" );
    }

    if( __svg_sheet_add_css_file( sheet, "fonts.css" ) != 1 )
    {
        fprintf( stderr, "Cannot attach fonts.css\n" );
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        printf( "Unable to create scene\n" );
    }

    __svg_set_scene_stroke( "black" );
    __svg_set_scene_stroke_width( 1.0 );
    __svg_set_scene_stroke_linecap( LINECAP_SQUARE );
    __svg_set_scene_text_fill_rgb( 0, 99, 99 );

    line_arrow( x0+32.0, y0+32.0, x0+70.0, y0+32.0 );
    circle_s( x0+80.0, y0+32.0, 10.0 );
    line_arrow( x0+90.0, y0+32.0, x0+130.0, y0+32.0 );

    text = (struct __svg_text *)__svg_new_text( x0+32.0, y0+22.0,
"<tspan font-weight=\\"500\\" font-style=\\"italic\\">M<tspan dy=\\"4\\" font-family=\\"Georgia\\" font-size
=\\"12\\">z</tspan></tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

```

```

/* theta */
text = (struct __svg_text *)__svg_new_text( x0+100.0, y0+22.0,
"<tspan font-style=\"italic\">&#x3b8;</tspan>",
        "Georgia", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

rect = (struct __svg_rect *)__svg_new_rect( x0+130.0, y0+4.0, 40, 56, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+144.0, y0+26.0,
"<tspan font-weight=\"500\" font-style=\"italic\">1</tspan>",
        "Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+138.0, y0+50.0,
"<tspan font-weight=\"500\" font-style=\"italic\">I<tspan dy=\"4\" font-family=\"Georgia\" font-size
    =\"12\">z</tspan></tspan>",
        "Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+153.0, y0+50.0,
"<tspan font-weight=\"500\" font-style=\"italic\">s</tspan>",
        "Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line = (struct __svg_line *)__svg_new_line( x0+138.0, y0+32.0, x0+162.0, y0+32.0 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
__svg_set_stroke_rgb( (struct __svg_node *)line, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

rect = (struct __svg_rect *)__svg_new_rect( x0+56.0, y0+80.0, 48, 28, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+68.0, y0+100.0,
"<tspan font-weight=\"500\" font-style=\"italic\">-2H</tspan>",
        "Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

rect = (struct __svg_rect *)__svg_new_rect( x0+26.0, y0+128.0, 30, 56, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+36.0, y0+150.0,
"<tspan font-weight=\"500\" font-style=\"italic\">1</tspan>",
        "Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+36.0, y0+174.0,
"<tspan font-weight=\"500\" font-style=\"italic\">s</tspan>",
        "Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line = (struct __svg_line *)__svg_new_line( x0+32.0, y0+156.0, x0+50.0, y0+156.0 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
__svg_set_stroke_rgb( (struct __svg_node *)line, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line_arrow( x0+130.0, y0+156.0, x0+56.0, y0+156.0 );
line_arrow( x0+26.0, y0+156.0, x0+0.0, y0+156.0 );

rect = (struct __svg_rect *)__svg_new_rect( x0+130.0, y0+128.0, 40, 56, 0, 0 );

```

```

(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+144.0, y0+150.0,
"<tspan font-weight=\"500\" font-style=\"italic\">1</tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+138.0, y0+174.0,
"<tspan font-weight=\"500\" font-style=\"italic\">I<tspan dy=\"4\" font-family=\"Georgia\" font-size
=\"12\">&#x3b3;</tspan></tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+153.0, y0+174.0,
"<tspan font-weight=\"500\" font-style=\"italic\">s</tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line = (struct __svg_line *)__svg_new_line( x0+138.0, y0+156.0, x0+162.0, y0+156.0 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
__svg_set_stroke_rgb( (struct __svg_node *)line, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

rect = (struct __svg_rect *)__svg_new_rect( x0+104.0, y0+204.0, 40, 28, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+114.0, y0+223.0,
"<tspan font-style=\"italic\">-&#x3bc;</tspan>",
"Georgia", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line_arrow( x0+80.0, y0+218.0, x0+80.0, y0+108.0 );
line_arrow( x0+80.0, y0+80.0, x0+80.0, y0+42.0 );

line_arrow( x0+80.0, y0+218.0, x0+104.0, y0+218.0 );

circle_s( x0+200.0, y0+156.0, 10.0 );
line_arrow( x0+190.0, y0+156.0, x0+170.0, y0+156.0 );
line_arrow( x0+200.0, y0+218.0, x0+200.0, y0+166.0 );

line = (struct __svg_line *)__svg_new_line( x0+144.0, y0+218.0, x0+200.0, y0+218.0 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_SQUARE );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

circle_s( x0+240.0, y0+156.0, 10.0 );
line_arrow( x0+230.0, y0+156.0, x0+210.0, y0+156.0 );

rect = (struct __svg_rect *)__svg_new_rect( x0+220.0, y0+80.0, 40, 28, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+234.0, y0+100.0,
"<tspan font-weight=\"500\" font-style=\"italic\">H</tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line_arrow( x0+240.0, y0+108.0, x0+240.0, y0+146.0 );
line_arrow( x0+240.0, y0+32.0, x0+240.0, y0+80.0 );

rect = (struct __svg_rect *)__svg_new_rect( x0+270.0, y0+4.0, 30, 56, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

```

```

text = (struct __svg_text *)__svg_new_text( x0+280.0, y0+26.0,
"<tspan font-weight=\"500\" font-style=\"italic\">1</tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+280.0, y0+50.0,
"<tspan font-weight=\"500\" font-style=\"italic\">s</tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line = (struct __svg_line *)__svg_new_line( x0+274.0, y0+32.0, x0+292.0, y0+32.0 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_BUTT );
__svg_set_stroke_rgb( (struct __svg_node *)line, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line_arrow( x0+170.0, y0+32.0, x0+270.0, y0+32.0 );

line_arrow( x0+280.0, y0+156.0, x0+250.0, y0+156.0 );
rect = (struct __svg_rect *)__svg_new_rect( x0+280.0, y0+142.0, 88, 28, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

text = (struct __svg_text *)__svg_new_text( x0+294.0, y0+162.0,
"<tspan font-weight=\"500\" font-style=\"italic\">k<tspan dy=\"4\" font-family=\"Georgia\" font-size
=\"12\">&#x3b8;</tspan></tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+313.0, y0+162.0,
"<tspan font-weight=\"500\" font-style=\"italic\">s +</tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+340.0, y0+162.0,
"<tspan font-weight=\"500\" font-style=\"italic\">k<tspan dy=\"4\" font-family=\"Georgia\" font-size
=\"12\">&#x3b8;</tspan></tspan>",
"Roboto", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

line_arrow( x0+330.0, y0+32.0, x0+330.0, y0+142.0 );
line = (struct __svg_line *)__svg_new_line( x0+300.0, y0+32.0, x0+330.0, y0+32.0 );
__svg_set_stroke_width( (struct __svg_node *)line, 1.0 );
__svg_set_stroke_linecap( (struct __svg_node *)line, LINECAP_SQUARE );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

text = (struct __svg_text *)__svg_new_text( x0+220.0, y0+22.0,
"<tspan font-style=\"italic\">&#x3b8;</tspan>",
"Georgia", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+340.0, y0+100.0,
"<tspan font-style=\"italic\">&#x3b8;</tspan>",
"Georgia", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+0.0, y0+144.0,
"<tspan font-style=\"italic\">&#x3b2;</tspan>",
"Georgia", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+100.0, y0+144.0,
"<tspan font-style=\"italic\">&#x3b2;</tspan>",
"Georgia", 16 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

```

```

circle = (struct __svg_circle *)__svg_new_circle( x0+104.0, y0+5.0, 0.75 );
__svg_set_stroke_rgb( (struct __svg_node *)circle, 0, 99, 99 );
__svg_set_fill_rgb( (struct __svg_node *)circle, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( x0+108.0, y0+5.0, 0.75 );
__svg_set_stroke_rgb( (struct __svg_node *)circle, 0, 99, 99 );
__svg_set_fill_rgb( (struct __svg_node *)circle, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( x0+226.0, y0+5.0, 0.75 );
__svg_set_stroke_rgb( (struct __svg_node *)circle, 0, 99, 99 );
__svg_set_fill_rgb( (struct __svg_node *)circle, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( x0+106.0, y0+126.0, 0.75 );
__svg_set_stroke_rgb( (struct __svg_node *)circle, 0, 99, 99 );
__svg_set_fill_rgb( (struct __svg_node *)circle, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( x0+306.8, y0+154.0, 0.5 );
__svg_set_stroke_rgb( (struct __svg_node *)circle, 0, 99, 99 );
__svg_set_fill_rgb( (struct __svg_node *)circle, 0, 99, 99 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

/*****
__svg_set_scene_stroke( "black" );
__svg_set_scene_stroke_width( 1.0 );
__svg_set_scene_stroke_dasharray( "none" );
__svg_set_scene_fill_rgb( 0, 0, 0 );

circle = (struct __svg_circle *)__svg_new_circle( x0+80.0, y0+156.0, 2 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( x0+240.0, y0+32.0, 2 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

__svg_paint_scene( scene, 1, "control-system.svg" );

__svg_free_scene( scene );

return( 0 );
}

```

```

cd svgc-examples/013-control-system
make
# open view.html in a browser

```

A structural diagram is a graphical form of a mathematical model. Its mathematical labels, branch junctions, and connection directions carry meaning that must be preserved during SVG generation, font selection, and preparation for print.

15. A Diagram of the Classical Runge–Kutta Method

15.1 FROM EQUATIONS TO COMPUTATIONAL STRUCTURE

The classical fourth-order Runge–Kutta method can be explained not only by equations but also by a structural diagram. At every integration step, four intermediate slopes k_1 , k_2 , k_3 , and k_4 are calculated and combined in a weighted sum. This diagram does not replace the formula; it illustrates the order of operations and dependencies among intermediate quantities.

For $y' = f(t, y)$, one step takes the form

$$\begin{aligned} k_1 &= f(t_n, y_n), & k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), & k_4 &= f(t_n + h, y_n + hk_3), \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

15.2 FIGURE GEOMETRY

The initial state t_n, y_n is placed on the left. To its right is a vertical column of four rectangles, one for each stage. Vertical arrows between neighboring blocks indicate prediction: each stage uses the preceding stage's result to construct a new intermediate point. On the far right, a combination block receives four horizontal arrows labeled k_1 , k_2 , k_3 , and k_4 .

The drawing makes it clear that RK4 is not simply a formula but an organized computational graph. This is the graph implemented by a numerical integration program.

Classical four-stage Runge-Kutta method

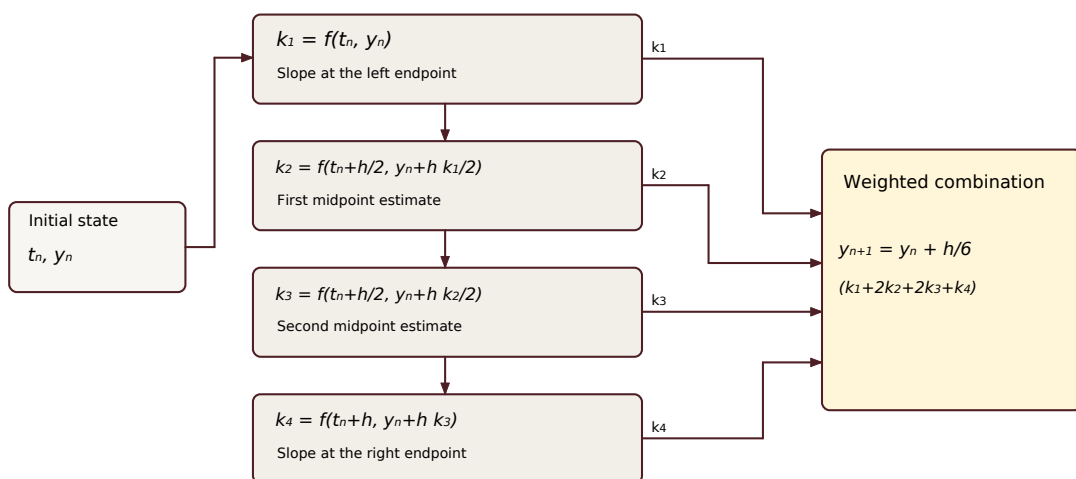


Figure 15.1. Structural diagram of one step of the classical RK4 method.

15.3 EXAMPLE 14

The example neither computes a numerical trajectory nor evaluates the actual k_i values. Its purpose is to demonstrate how an algorithmic diagram can be constructed in code: rectangular blocks, directed connections, labels, and a final combination block. It lies between an engineering diagram and an illustration of a numerical method.

As in earlier chapters, text within the rectangles is centered manually after visual inspection. Automatic string-width calculation would require font-metric analysis, which SVG C deliberately leaves to other tools.

```

/*****
SVG C BOOK EXAMPLE 014: RK4 STAGE SCHEME

A compact structural drawing for the classical Runge--Kutta method.
*****/

#include <stdio.h>
#include <svgc.h>

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
text_at( struct __svg_scene *scene, double x, double y,
         double size, const char *text )
{
    return( append( scene, __svg_new_text( x, y, text, "DejaVu Sans", size ),
                   "none", NULL, 0.0 ) );
}

static int
line( struct __svg_scene *scene, double x1, double y1,
      double x2, double y2 )
{
    struct __svg_node *node;
    node = __svg_new_line( x1, y1, x2, y2 );
    if( !node ) return( 0 );
    (void)__svg_set_stroke_linecap( node, LINECAP_ROUND );
    return( append( scene, node, "#4D1F24", "none", 2.2 ) );
}

static int
arrow_head( struct __svg_scene *scene, double x, double y,
            double ux, double uy )
{
    struct __svg_points *pts;
    struct __svg_node *node;
    double nx = -uy, ny = ux;

    pts = __svg_new_points();
    if( !pts ) return( 0 );
    if( __svg_add_point( pts, x, y ) < 1 ||
        __svg_add_point( pts, x - 12.0*ux + 4.5*nx, y - 12.0*uy + 4.5*ny ) < 1 ||

```

```

    __svg_add_point( pts, x - 12.0*ux - 4.5*nx, y - 12.0*uy - 4.5*ny ) < 1 )
    {
        __svg_free_points( pts );
        return( 0 );
    }
    node = __svg_new_polygon( pts );
    if( !node )
    {
        __svg_free_points( pts );
        return( 0 );
    }
    return( append( scene, node, "#4D1F24", "#4D1F24", 1.0 ) );
}

static int
h_arrow( struct __svg_scene *scene, double x1, double y, double x2 )
{
    if( !line( scene, x1, y, x2, y ) ) return( 0 );
    return( arrow_head( scene, x2, y, 1.0, 0.0 ) );
}

static int
v_arrow( struct __svg_scene *scene, double x, double y1, double y2 )
{
    if( !line( scene, x, y1, x, y2 ) ) return( 0 );
    return( arrow_head( scene, x, y2, 0.0, 1.0 ) );
}

static int
box( struct __svg_scene *scene, double x, double y, double w, double h,
      const char *fill )
{
    return( append( scene, __svg_new_rect( x, y, w, h, 8, 8 ),
                    "#4D1F24", fill, 2.2 ) );
}

/* A subordinate SVG tspan is the correct notation for indices.
 * No HTML span elements are used in this SVG document. */
#define SUB_1 "<tspan baseline-shift=\"sub\" font-size=\"12\">1</tspan>"
#define SUB_2 "<tspan baseline-shift=\"sub\" font-size=\"12\">2</tspan>"
#define SUB_3 "<tspan baseline-shift=\"sub\" font-size=\"12\">3</tspan>"
#define SUB_4 "<tspan baseline-shift=\"sub\" font-size=\"12\">4</tspan>"
#define SUB_N "<tspan baseline-shift=\"sub\" font-size=\"12\">n</tspan>"
#define SUB_N1 "<tspan baseline-shift=\"sub\" font-size=\"12\">n+1</tspan>"

int
main( void )
{
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 1220, 650, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    if( !append( scene, __svg_new_rect( 0, 0, 1220, 650, 0, 0 ),
                "none", "white", 0.0 ) ) goto done;

    if( !text_at( scene, 48, 40, 19, "Classical four-stage Runge-Kutta method" ) ) goto done;

```

```

if( !box( scene, 40, 270, 188, 96, "#F8F6F2" ) ) goto done;
if( !text_at( scene, 61, 295, 17, "Initial state" ) ) goto done;
if( !text_at( scene, 60, 333, 20, "<tspan font-style=\<italic\>t" SUB_N ", y" SUB_N "</tspan>" ) ) goto
done;

if( !box( scene, 300, 70, 415, 95, "#F2EEE8" ) ) goto done;
if( !text_at( scene, 325, 104, 21,
    "<tspan font-style=\<italic\>k" SUB_1 " = f(t" SUB_N ", y" SUB_N "</tspan>" ) ) goto done;
if( !text_at( scene, 326, 138, 15, "Slope at the left endpoint" ) ) goto done;

if( !box( scene, 300, 205, 415, 95, "#F2EEE8" ) ) goto done;
if( !text_at( scene, 324, 239, 18,
    "<tspan font-style=\<italic\>k" SUB_2 " = f(t" SUB_N "+h/2, y" SUB_N "+h k" SUB_1 "/2)</tspan>" )
    ) goto done;
if( !text_at( scene, 326, 273, 15, "First midpoint estimate" ) ) goto done;

if( !box( scene, 300, 340, 415, 95, "#F2EEE8" ) ) goto done;
if( !text_at( scene, 324, 374, 18,
    "<tspan font-style=\<italic\>k" SUB_3 " = f(t" SUB_N "+h/2, y" SUB_N "+h k" SUB_2 "/2)</tspan>" )
    ) goto done;
if( !text_at( scene, 326, 408, 15, "Second midpoint estimate" ) ) goto done;

if( !box( scene, 300, 475, 415, 95, "#F2EEE8" ) ) goto done;
if( !text_at( scene, 324, 509, 19,
    "<tspan font-style=\<italic\>k" SUB_4 " = f(t" SUB_N "+h, y" SUB_N "+h k" SUB_3 "</tspan>" ) )
    goto done;
if( !text_at( scene, 326, 543, 15, "Slope at the right endpoint" ) ) goto done;

if( !box( scene, 907, 214, 277, 278, "#FFF5D9" ) ) goto done;
if( !text_at( scene, 930, 255, 19, "Weighted combination" ) ) goto done;
if( !text_at( scene, 925, 326, 19,
    "<tspan font-style=\<italic\>y" SUB_N1 " = y" SUB_N " + h/6</tspan>" ) ) goto done;
if( !text_at( scene, 925, 367, 17,
    "<tspan font-style=\<italic\>(k" SUB_1 "+2k" SUB_2 "+2k" SUB_3 "+k" SUB_4 "</tspan>" ) ) goto
done;

/* One input route, three stage dependencies, four distinct outputs. */
if( !line( scene, 228, 318, 259, 318 ) ) goto done;
if( !line( scene, 259, 318, 259, 116 ) ) goto done;
if( !h_arrow( scene, 259, 116, 300 ) ) goto done;
if( !v_arrow( scene, 505, 165, 205 ) ) goto done;
if( !v_arrow( scene, 505, 300, 340 ) ) goto done;
if( !v_arrow( scene, 505, 435, 475 ) ) goto done;

if( !line( scene, 715, 117, 844, 117 ) ) goto done;
if( !line( scene, 844, 117, 844, 282 ) ) goto done;
if( !h_arrow( scene, 844, 282, 907 ) ) goto done;

if( !line( scene, 715, 252, 786, 252 ) ) goto done;
if( !line( scene, 786, 252, 786, 335 ) ) goto done;
if( !h_arrow( scene, 786, 335, 907 ) ) goto done;

if( !h_arrow( scene, 715, 387, 907 ) ) goto done;

if( !line( scene, 715, 522, 844, 522 ) ) goto done;
if( !line( scene, 844, 522, 844, 441 ) ) goto done;
if( !h_arrow( scene, 844, 441, 907 ) ) goto done;

if( !text_at( scene, 725, 110, 15, "k" SUB_1 ) ) goto done;
if( !text_at( scene, 725, 245, 15, "k" SUB_2 ) ) goto done;
if( !text_at( scene, 725, 380, 15, "k" SUB_3 ) ) goto done;
if( !text_at( scene, 725, 515, 15, "k" SUB_4 ) ) goto done;

__svg_paint_scene( scene, 0, "rk4-scheme.svg" );
status = 0;

```

```
done:
    if( status ) fprintf( stderr, "Cannot construct RK4 scheme\n" );
    __svg_free_scene( scene );
    return( status );
}
```

```
cd svgc-examples/014-rk4-scheme
make
# open view.html in a browser
```

After this structural illustration, we can move on to more substantial figures: solution plots, discrete data, splines, and diagrams for investigating zeros and poles.

When an algorithm has a clear internal structure, that structure is worth drawing as an object of study in its own right. SVGC makes such a diagram programmatic and reproducible.

16. A Mechanical Oscillator: Geometry of Forces and Connections

16.1 AN AUTHOR'S ILLUSTRATION AS A PROGRAM

In the previous chapter, we illustrated the sequence of computations in a numerical method. We now consider a fundamentally different case: a *mechanical diagram* in which the positions of bodies, directions of forces, and auxiliary constructions are calculated from parameters.

This chapter is based on Figure 2.1, published in the section “Constructing a Program Model” at <https://dynamics.su/theory/issues/#targets-setting>. The drawing contains a suspension point, a rod, a material point M , a vertical reference line, geometric arcs, and directed segments. The symbols $B, G, G_\tau, G_n, N, \tau, n$, and ν are separate text elements. Their subscripts are formed using nested SVG `tspan` elements.

The program *does not solve a differential equation* or plot a trajectory. It constructs a single parameterized geometric frame that can subsequently support the formulation and analysis of a mathematical model.

16.2 COORDINATE SYSTEM AND GEOMETRIC PARAMETERS

Let the fixed suspension point be $O = (x_0, y_0)$, the rod length be r_0 , and its direction make an angle α with the horizontal axis pointing right. In SVG, the y axis points *downward*. The rod's endpoint is therefore computed directly as

$$x_M = x_0 + r_0 \cos \alpha, \quad y_M = y_0 + r_0 \sin \alpha.$$

The original example uses

$$(x_0, y_0) = (100, 30), \quad r_0 = 250, \quad \alpha = 60^\circ.$$

Consequently,

$$x_M = 225, \quad y_M = 30 + 125\sqrt{3} \approx 246.506.$$

These coordinates can be checked against the attributes of the resulting `oscillator.svg`. The dashed vertical through the suspension ends at $(x_0, y_0 + r_0)$, while a separate arc indicates the angle between the rod and the vertical. It is labeled φ ; for the chosen configuration, $\varphi = 90^\circ - \alpha = 30^\circ$.

An angle measured counterclockwise from the positive mathematical x axis cannot be substituted blindly into these formulas: the screen y axis has the opposite direction. Sign conventions belong to the application algorithm, not to SVG configuration.

16.3 A LOCAL BASIS AND DECOMPOSITION OF THE VERTICAL FORCE

At M , define a radial unit vector from O to M and a tangent unit vector:

$$\mathbf{e}_n = (\cos \alpha, \sin \alpha), \quad \mathbf{e}_\tau = (-\sin \alpha, \cos \alpha).$$

Both vectors use screen coordinates; they are mutually orthogonal and have unit length.

Let the downward vertical force have magnitude $G \geq 0$. In screen coordinates, $\mathbf{G} = (0, G)$. Its projections onto the chosen basis are

$$G_n = \mathbf{G} \cdot \mathbf{e}_n = G \sin \alpha, \quad G_\tau = \mathbf{G} \cdot \mathbf{e}_\tau = G \cos \alpha.$$

This is the geometry illustrated by the original program: the segment labeled G_n follows the rod, and G_τ follows the tangent. At $\alpha = 60^\circ$, the ratios of the projected force magnitudes to G are $\sqrt{3}/2$ and $1/2$, respectively.

Distinguish *physical force magnitudes* from *screen arrow lengths*. The source uses an illustrative graphic scale $r_3 = 90$, from which $r_{11} = r_3 \cos \alpha$ and $r_{12} = r_3 \sin \alpha$ are calculated. These are construction lengths, not automatically evaluated physical parameters of the model. Similarly, B , N , and other vectors are represented by symbolic segments with predetermined sizes for a readable diagram.

16.4 ARROW DIRECTIONS AND ROTATION RULES

The original program has two arrowhead types, `arrow` and `small_arrow`. Each is described by four control points relative to the local tip. The points are rotated through φ according to

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \end{pmatrix} + \begin{pmatrix} \cos \varphi & -\sin \varphi \\ \sin \varphi & \cos \varphi \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}.$$

A path is then created: straight sides are joined by a short rounded base using `__svg_add_short_curveto_cmd()`. A white fill conceals the continuation of the line underneath. We encountered this method in the chapter on coordinate-axis arrowheads; here it is applied to several inclined directions.

Some geometry uses solid strokes of varying widths, while auxiliary projections and the large angular sector use thin dashed lines. Styling thus separates *vector quantities* from *geometric constructions* without changing their coordinates.

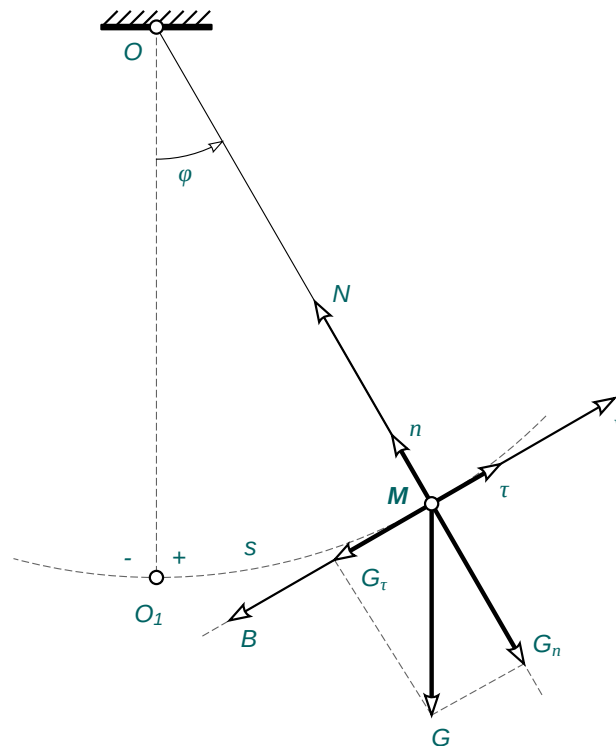


Figure 16.1. Mechanical oscillator diagram: suspension point, vector geometry, and projections of gravity.

16.5 ARCS AND PARTIAL CIRCLES

The diagram contains two arcs serving different purposes. A small arc near the suspension indicates the inclination angle; a larger dashed arc illustrates possible positions of the rod endpoint at fixed length r_0 . Both use the SVG A command rather than an approximation with many short segments.

The small arc has radius $r_9 = r_{10} = 60$; its endpoints are computed from $\sin \alpha$ and $\cos \alpha$. The large arc uses radius r_0 and two additional angular positions. SVG elliptical arcs were introduced in Chapter 11; here we see the same primitive applied to a mechanical illustration linked to a mathematical model.

16.6 LABELS WITH MATHEMATICAL SUBSCRIPTS

A force diagram needs more than strings such as Gtau, Gn, and O1. The correct symbols are G_τ , G_n , and O_1 . The C source uses tspan fragments with adjusted dy and font-size attributes. The structured SVG thus stores the base symbol and its subscript separately instead of rendering them to a bitmap.

CairoSVG preserves these fragments as parts of the original SVG text objects when generating the printed PDF. Check subscript sizes, Greek glyphs, and label positions in the result; the mathematical notation need not be re-created manually for print.

16.7 EXAMPLE 15

The example source is in `svgc-examples/015-oscillator`. It retains the original figure's geometry, parameters, and labels.

The result comprises 52 graphical objects: 22 segments, 11 paths, 3 circles, 15 text nodes, and one rectangle. This count excludes the SVG document and its definitions section.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <math.h>

#include <svgc.h>

#define OSC_SCALE 2.5

struct __svg_sheet *sheet;
struct __svg_line *line;
struct __svg_rect *rect;
struct __svg_circle *circle;
struct __svg_ellipse *ellipse;
struct __svg_text *text;

struct __svg_points *points;
struct __svg_polygon *polygon;
struct __svg_polyline *polyline;

struct __svg_cmd *cmd;
struct __svg_cmds *cmds;
struct __svg_path *path;
```

```

struct __svg_scene *scene;

void arrow( double sx0, double sy0, double f0 )
{
    double sx1, sy1, sx2, sy2, sx3, sy3, sx4, sy4;
    double px1, py1, px2, py2, px3, py3, px4, py4;

    sx1 = sx0 - 3.0 * OSC_SCALE;
    sy1 = sy0 + 3.0 * OSC_SCALE;
    sx2 = sx0;
    sy2 = sy0 - 6.0 * OSC_SCALE;
    sx3 = sx0 + 3.0 * OSC_SCALE;
    sy3 = sy0 + 3.0 * OSC_SCALE;
    sx4 = sx0;
    sy4 = sy0 + 1.0 * OSC_SCALE;

    px1 = sx0 + ( sx1 - sx0 )*cos( f0 ) - ( sy1 - sy0 )*sin( f0 );
    py1 = sy0 + ( sx1 - sx0 )*sin( f0 ) + ( sy1 - sy0 )*cos( f0 );
    px2 = sx0 + ( sx2 - sx0 )*cos( f0 ) - ( sy2 - sy0 )*sin( f0 );
    py2 = sy0 + ( sx2 - sx0 )*sin( f0 ) + ( sy2 - sy0 )*cos( f0 );
    px3 = sx0 + ( sx3 - sx0 )*cos( f0 ) - ( sy3 - sy0 )*sin( f0 );
    py3 = sy0 + ( sx3 - sx0 )*sin( f0 ) + ( sy3 - sy0 )*cos( f0 );
    px4 = sx0 + ( sx4 - sx0 )*cos( f0 ) - ( sy4 - sy0 )*sin( f0 );
    py4 = sy0 + ( sx4 - sx0 )*sin( f0 ) + ( sy4 - sy0 )*cos( f0 );

    cmds = __svg_new_cmds();
    __svg_add_moveto_cmd( cmds, ABSOLUTE, px1, py1 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, px2, py2 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, px3, py3 );
    __svg_add_short_curveto_cmd( cmds, ABSOLUTE, px4, py4, px1, py1 );
    __svg_add_closepath_cmd( cmds, ABSOLUTE );
    path = (struct __svg_path *)__svg_new_path( cmds );
    (void)__svg_set_stroke_rgb( (struct __svg_node *)path, 0, 0, 0 );
    (void)__svg_set_stroke_width( (struct __svg_node *)path, 1.0 * OSC_SCALE );
    (void)__svg_set_stroke_linecap( (struct __svg_node *)path, "square" );
    (void)__svg_set_stroke_linejoin( (struct __svg_node *)path, "miter" );
    (void)__svg_set_fill_rgb( (struct __svg_node *)path, 255, 255, 255 );

    (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

void small_arrow( double sx0, double sy0, double f0 )
{
    double sx1, sy1, sx2, sy2, sx3, sy3, sx4, sy4;
    double px1, py1, px2, py2, px3, py3, px4, py4;

    sx1 = sx0 - 2.0 * OSC_SCALE;
    sy1 = sy0 + 2.0 * OSC_SCALE;
    sx2 = sx0;
    sy2 = sy0 - 4.0 * OSC_SCALE;
    sx3 = sx0 + 2.0 * OSC_SCALE;
    sy3 = sy0 + 2.0 * OSC_SCALE;
    sx4 = sx0;
    sy4 = sy0 + 1.0 * OSC_SCALE;

    px1 = sx0 + ( sx1 - sx0 )*cos( f0 ) - ( sy1 - sy0 )*sin( f0 );
    py1 = sy0 + ( sx1 - sx0 )*sin( f0 ) + ( sy1 - sy0 )*cos( f0 );
    px2 = sx0 + ( sx2 - sx0 )*cos( f0 ) - ( sy2 - sy0 )*sin( f0 );
    py2 = sy0 + ( sx2 - sx0 )*sin( f0 ) + ( sy2 - sy0 )*cos( f0 );
    px3 = sx0 + ( sx3 - sx0 )*cos( f0 ) - ( sy3 - sy0 )*sin( f0 );
    py3 = sy0 + ( sx3 - sx0 )*sin( f0 ) + ( sy3 - sy0 )*cos( f0 );
    px4 = sx0 + ( sx4 - sx0 )*cos( f0 ) - ( sy4 - sy0 )*sin( f0 );
    py4 = sy0 + ( sx4 - sx0 )*sin( f0 ) + ( sy4 - sy0 )*cos( f0 );
}

```

```

cmds = __svg_new_cmds();
__svg_add_moveto_cmd( cmds, ABSOLUTE, px1, py1 );
__svg_add_lineto_cmd( cmds, ABSOLUTE, px2, py2 );
__svg_add_lineto_cmd( cmds, ABSOLUTE, px3, py3 );
__svg_add_short_curveto_cmd( cmds, ABSOLUTE, px4, py4, px1, py1 );
__svg_add_closepath_cmd( cmds, ABSOLUTE );
path = (struct __svg_path *)__svg_new_path( cmds );
(void)__svg_set_stroke_rgb( (struct __svg_node *)path, 0, 0, 0 );
(void)__svg_set_stroke_width( (struct __svg_node *)path, 0.5 * OSC_SCALE );
(void)__svg_set_stroke_linecap( (struct __svg_node *)path, "square" );
(void)__svg_set_stroke_linejoin( (struct __svg_node *)path, "miter" );
(void)__svg_set_fill_rgb( (struct __svg_node *)path, 255, 255, 255 );

(void)__svg_scene_add_node( scene, (struct __svg_node *)path );
}

int main( void )
{
    int i;
    double x0 = 100.0 * OSC_SCALE,
           y0 = 30.0 * OSC_SCALE,
           r0 = 250.0 * OSC_SCALE,
           a0 = 60.0 * M_PI / 180.0,
           xM, yM;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 350 * OSC_SCALE, 380 * OSC_SCALE,
                                                  10 * OSC_SCALE, 10 * OSC_SCALE,
                                                  10 * OSC_SCALE, 10 * OSC_SCALE );

    if( !sheet )
    {
        printf( "Unable to create sheet\n" );
    }

    if( __svg_sheet_add_css_file( sheet, "fonts.css" ) != 1 )
    {
        fprintf( stderr, "Cannot attach fonts.css\n" );
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }

    (void)__svg_set_stroke_width( (struct __svg_node *)sheet, 1024.1236 * OSC_SCALE );

    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        printf( "Unable to create scene\n" );
    }

    __svg_set_scene_stroke( "black" );
    __svg_set_scene_stroke_width( 1.0 * OSC_SCALE );
    __svg_set_scene_stroke_linecap( LINECAP_SQUARE );

    rect = (struct __svg_rect *)__svg_new_rect( x0-25.0*OSC_SCALE, y0-1.0*OSC_SCALE,
                                                50*OSC_SCALE, 2*OSC_SCALE, 0, 0 );

```

```

(void)__svg_set_fill_rgb( (struct __svg_node *)rect, 0, 0, 0 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)rect );

{
    double x1 = x0-24.0*OSC_SCALE, y1 = y0-1.0*OSC_SCALE,

```

```

        x2 = x0-18.0*OSC_SCALE, y2 = y0-7.0*OSC_SCALE;

        for( i = 0; i < 8; ++i )
        {
            line = (struct __svg_line *)__svg_new_line( x1, y1, x2, y2 );
            (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

            x1 += 6*OSC_SCALE; x2 += 6*OSC_SCALE;
        }
    }

    xM = x0 + r0 * cos( a0 );
    yM = y0 + r0 * sin( a0 );

    __svg_set_scene_stroke_width( 0.5 * OSC_SCALE );

    line = (struct __svg_line *)__svg_new_line( x0, y0, xM, yM );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    /* Set current values for newest objects: */
    __svg_set_scene_stroke_width( 0.5 * OSC_SCALE );
    __svg_set_scene_stroke_rgb( 99, 99, 99 );
    __svg_set_scene_stroke_dasharray( "7.5,5" );

    line = (struct __svg_line *)__svg_new_line( x0, y0, x0, y0+r0 );
    (void)__svg_scene_add_node( scene, (struct __svg_node *)line );

    {
        double x1, y1, x2, y2, a1 = 45.0, a2 = 75.0;

        a1 = a1 * M_PI / 180.0;
        a2 = a2 * M_PI / 180.0;
        x1 = x0 + r0 * cos(a1);
        y1 = y0 + r0 * sin(a1);
        x2 = x0 - r0 * cos(a2);
        y2 = y0 + r0 * sin(a2);

        cmds = __svg_new_cmds();
        /* M x1,y1 */
        __svg_add_moveto_cmd( cmds, ABSOLUTE, x1, y1 );

        __svg_add_arc_cmd( cmds,
                           ABSOLUTE,
                           r0, r0,      /* rx, ry */
                           0.0,        /* angle */
                           0,1,        /* lf, sf */
                           x2, y2 );

        path = (struct __svg_path *)__svg_new_path( cmds );
        (void)__svg_scene_add_node( scene, (struct __svg_node *)path );
    }

    {
        double x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6,
               x7, y7, x8, y8, x9, y9, x10, y10, x11, y11, x12, y12;

        double x3_a, y3_a, x11_a, y11_a, x12_a, y12_a;

        double r1 = 120.0*OSC_SCALE,
               r2 = 100.0*OSC_SCALE,
               r3 = 90.0*OSC_SCALE,
               r4 = 100.0*OSC_SCALE,
               r5 = 90.0*OSC_SCALE,
               r6 = 30.0*OSC_SCALE,

```

```

        r7 = 100.0*OSC_SCALE,
        r8 = 30.0*OSC_SCALE,
        r9 = 60.0*OSC_SCALE,
        r10 = 60.0*OSC_SCALE,
        r11, r12;

/* Set current values for newest objects: */
__svg_set_scene_stroke_rgb( 99, 99, 99 );
__svg_set_scene_stroke_width( 0.5 * OSC_SCALE );
__svg_set_scene_stroke_dasharray( "7.5,5" );
__svg_set_scene_fill( "none" );

x1 = xM - r1 * sin( a0 );
y1 = yM + r1 * cos( a0 );

x2 = xM - r2 * sin( a0 );
y2 = yM + r2 * cos( a0 );

x3 = xM;
y3 = yM + r3;

x3_a = xM;
y3_a = yM + r3 + 6.0*OSC_SCALE;

x4 = xM + r4 * cos( a0 );
y4 = yM + r4 * sin( a0 );

x5 = xM + r5 * sin( a0 );
y5 = yM - r5 * cos( a0 );

x6 = xM + r6 * sin( a0 );
y6 = yM - r6 * cos( a0 );

x7 = xM - r7 * cos( a0 );
y7 = yM - r7 * sin( a0 );

x8 = xM - r8 * cos( a0 );
y8 = yM - r8 * sin( a0 );

x9 = x0 + r9 * cos( a0 );
y9 = y0 + r9 * sin( a0 );

x10 = x0;
y10 = y0 + r10;

r11 = r3 * cos( a0 );
x11 = xM - r11 * sin( a0 );
y11 = yM + r11 * cos( a0 );
x11_a = xM - (r11+6.0*OSC_SCALE) * sin( a0 );
y11_a = yM + (r11+6.0*OSC_SCALE) * cos( a0 );

r12 = r3 * sin( a0 );
x12 = xM + r12 * cos( a0 );
y12 = yM + r12 * sin( a0 );
x12_a = xM + (r12+6.0*OSC_SCALE) * cos( a0 );
y12_a = yM + (r12+6.0*OSC_SCALE) * sin( a0 );

line = (struct __svg_line *)__svg_new_line( x1, y1, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x4, y4, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x3_a, y3_a, x11_a, y11_a );

```

```

(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x3_a, y3_a, x12_a, y12_a );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

__svg_set_scene_stroke( "black" );
__svg_set_scene_stroke_width( 0.5 * OSC_SCALE );
__svg_set_scene_stroke_dasharray( "none" );

cmds = __svg_new_cmds();
/* M x1,y1 */
__svg_add_moveto_cmd( cmds, ABSOLUTE, x9, y9 );

__svg_add_arc_cmd( cmds,
                  ABSOLUTE,

```

```

                  r10, r10,      /* rx, ry */
                  0.0,         /* angle */
                  0,1,         /* lf, sf */
                  x10, y10 );

path = (struct __svg_path *)__svg_new_path( cmds );
(void)__svg_scene_add_node( scene, (struct __svg_node *)path );

__svg_set_scene_stroke_width( 1.0 * OSC_SCALE );

line = (struct __svg_line *)__svg_new_line( x2, y2, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x5, y5, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x7, y7, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

__svg_set_scene_stroke_width( 2.0 * OSC_SCALE );

line = (struct __svg_line *)__svg_new_line( x3, y3, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x6, y6, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x8, y8, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x11, y11, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

line = (struct __svg_line *)__svg_new_line( x12, y12, xM, yM );
(void)__svg_scene_add_node( scene, (struct __svg_node *)line );

/* стрелки */
arrow( x2, y2, M_PI + a0 );
arrow( x3, y3, M_PI );
arrow( x11, y11, M_PI + a0 );
arrow( x12, y12, M_PI - (M_PI / 2 - a0) );
arrow( x5, y5, a0 );
arrow( x6, y6, a0 );
arrow( x7, y7, -(M_PI / 2 - a0) );
arrow( x8, y8, -(M_PI / 2 - a0) );

```

```

{
    double fx1, fy1;
    fx1 = x9 - 4.0 * OSC_SCALE * sin( a0 );
    fy1 = y9 + 4.0 * OSC_SCALE * cos( a0 );
    small_arrow( fx1, fy1, a0 );
}

__svg_set_scene_text_fill_rgb( 0, 99, 99 );
text = (struct __svg_text *)__svg_new_text( x2, y2+15*OSC_SCALE,
"<tspan font-style=\"oblique\">B</tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x11+7*OSC_SCALE, y11+15*OSC_SCALE,
"<tspan font-style=\"oblique\">G<tspan dy=\"4\" font-family=\"Georgia\" font-size=\"20\"></tspan></tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x12+7*OSC_SCALE, y12,
"<tspan font-style=\"oblique\">G<tspan dy=\"4\" font-family=\"Georgia\" font-size=\"20\">n</tspan></tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x3, y3+20*OSC_SCALE,
"<tspan font-style=\"oblique\">G</tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( xM-20*OSC_SCALE, yM,
"<tspan font-weight=\"bold\" font-style=\"oblique\">M</tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x6+5*OSC_SCALE, y6+12*OSC_SCALE,
"<tspan font-family=\"Georgia\" font-style=\"oblique\"></tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x8+5*OSC_SCALE, y8-5*OSC_SCALE,
"<tspan font-family=\"Georgia\" font-style=\"oblique\">n</tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x5+5*OSC_SCALE, y5+12*OSC_SCALE,
"<tspan font-family=\"Georgia\" font-style=\"oblique\"></tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x7+5*OSC_SCALE, y7-5*OSC_SCALE,
"<tspan font-style=\"oblique\">N</tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+10*OSC_SCALE, y0+r9+10*OSC_SCALE,
"<tspan font-family=\"Georgia\" font-style=\"oblique\"></tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0-10*OSC_SCALE, y0+r0+20*OSC_SCALE,
"<tspan font-style=\"oblique\">0<tspan dy=\"4\" font-size=\"20\">1</tspan></tspan>",
        "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

```

```

text = (struct __svg_text *)__svg_new_text( x0-15*OSC_SCALE, y0+15*OSC_SCALE,
"<tspan font-style=\"oblique\">0</tspan>",
      "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0-15*OSC_SCALE, y0+r0-5*OSC_SCALE,
"<tspan font-style=\"oblique\">-</tspan>",
      "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+7*OSC_SCALE, y0+r0-5*OSC_SCALE,
"<tspan font-style=\"oblique\">+</tspan>",
      "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );

text = (struct __svg_text *)__svg_new_text( x0+40*OSC_SCALE, y0+r0-10*OSC_SCALE,
"<tspan font-style=\"oblique\">s</tspan>",
      "Verdana", 28 );
(void)__svg_scene_add_node( scene, (struct __svg_node *)text );
}

/* Set current values for newest objects: */
__svg_set_scene_stroke( "black" );
__svg_set_scene_stroke_width( 1.0 * OSC_SCALE );
__svg_set_scene_stroke_dasharray( "none" );
__svg_set_scene_fill_rgb( 255, 255, 255 );

circle = (struct __svg_circle *)__svg_new_circle( x0, y0, 3*OSC_SCALE );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( xM, yM, 3*OSC_SCALE );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

circle = (struct __svg_circle *)__svg_new_circle( x0, y0+r0, 3*OSC_SCALE );
(void)__svg_scene_add_node( scene, (struct __svg_node *)circle );

__svg_set_scene_fill( "none" );

__svg_paint_scene( scene, 0, "oscillator.svg" );

```

```
стрелкиттвφ
```

```

__svg_free_scene( scene );

return( 0 );
}

```

```

cd svgc-examples/015-oscillator
make
# open view.html in a browser

```

16.8 NEXT: COORDINATE AXES AND COMPUTATIONAL RESULTS

This chapter has drawn one prescribed geometric frame. Plotting a time series additionally requires solving the equations of motion, determining the ranges of values, constructing axes and ticks, and only then transforming numerical points into screen coordinates. These topics are addressed in the following chapters on scientific plots, discrete observations, and interpolation.

A mathematical model first determines the quantities and their relationships. The application then turns them into geometric coordinates, styles, and labels. SVGC records the result without interfering with its physical meaning.

17. Coordinate Axes, Scales, and Tick Labels

17.1 TWO INDEPENDENT COORDINATE SYSTEMS

Chapter 4 introduced the conversion of numerical data to SVG coordinates. A finished scientific illustration needs more: a plotting area, numerical grid, tick labels, and an explicit scaling policy. Even if data are displayed correctly, poorly chosen bounds or grid spacing can conceal important features.

Distinguish the *mathematical domain* from the *display rectangle*. The domain is defined by four real numbers $x_{\min}$, $x_{\max}$, $y_{\min}$, and $y_{\max}$. The rectangle is defined by the position (L, T) of its upper-left corner and its SVG width W and height H . The affine mapping is

$$X(x) = L + W \frac{x - x_{\min}}{x_{\max} - x_{\min}}, \quad Y(y) = T + H \frac{y_{\max} - y}{y_{\max} - y_{\min}}.$$

The second expression reverses the mathematical vertical direction because SVG Y increases downward. The coordinate transformation belongs to the application; SVGC receives the calculated screen coordinates.

17.2 DATA RANGES AND PROTECTIVE MARGINS

Initially, find the minimum and maximum of all finite sample values x_i and y_i . Then expand the range by a small relative amount:

$$x'_{\min} = x_{\min} - \delta_x \Delta x, \quad x'_{\max} = x_{\max} + \delta_x \Delta x,$$

where $\Delta x = x_{\max} - x_{\min}$; similarly for y . This does not change the data; it reserves visual space between extreme samples and the plotting boundary. If $\Delta x = 0$ or $\Delta y = 0$, the degenerate interval must be replaced by a finite, positive scale to avoid division by zero.

Our example plots 41 horizontal coordinates from -2.4 to 6.4 and the corresponding values of

$$y(x) = 1.6 \sin(0.82x) + 0.24x.$$

The plotted domain is derived from the samples and its bounds rounded outward: to the nearest half-unit horizontally and to the nearest quarter-unit vertically. This gives

$$x \in [-2.5, 6.5], \quad y \in [-2.25, 2.25].$$

The axes therefore end midway between neighboring grid lines. Horizontal ticks have spacing 1, and vertical ticks spacing 0.5. The curve remains strictly inside the calculated bounds.

17.3 CHOOSING A GRID STEP

Dividing every interval mechanically into ten equal parts rarely produces readable labels. A range of length 13.7, for example, would have step 1.37, leading to awkward fractional labels. Instead, select a step from the family

$$d \in \{1, 2, 5, 10\} \cdot 10^k, \quad k \in \mathbb{Z}.$$

The program's `nice_step()` function receives the interval width and a target number of divisions. Let $d_0 = (x_{\max} - x_{\min})/N$ and $k = \lfloor \log_{10} d_0 \rfloor$. Then $d = d_1 10^k$, where d_1 is the smallest of 1, 2, 5, and 10 that is *not less than* the normalized step $d_0/10^k$. This yields a neat numerical scale without excessive labeling.

It is convenient to compute visible tick indices as integers:

$$i_{\min} = \left\lceil \frac{x_{\min}}{d} \right\rceil, \quad i_{\max} = \left\lfloor \frac{x_{\max}}{d} \right\rfloor, \quad x_i = i d.$$

This avoids accumulating rounding error by repeatedly adding a small floating-point increment. The same method works for both horizontal and vertical grids.

17.4 PAINTING ORDER

The illustration uses a thin light-gray grid; axes through the mathematical origin are darker, and the curve is shown in #4D1F24. Elements are painted in this order: background, grid, axes with ticks and arrowheads, then the data curve. No frame surrounds the grid, since the axes pass through the mathematical origin. The fine grid does not obscure the dependence being studied.

The example calculates grid lines in a loop rather than copying them from an existing SVG. Short tick marks lie directly on the axes, while their numbers are separate text elements. In their positive directions, the axes end in hollow curved arrowheads built like the last arrowhead in Figure 6.1. Actual string widths depend on font metrics and cannot be computed by SVGC, so label offsets are chosen manually.

These offsets do not change the mathematical content, but they substantially improve readability. A label that overlaps the curve, an axis, or a neighboring tick should be moved slightly; an unimportant tick can even be omitted when necessary to keep the figure clear.

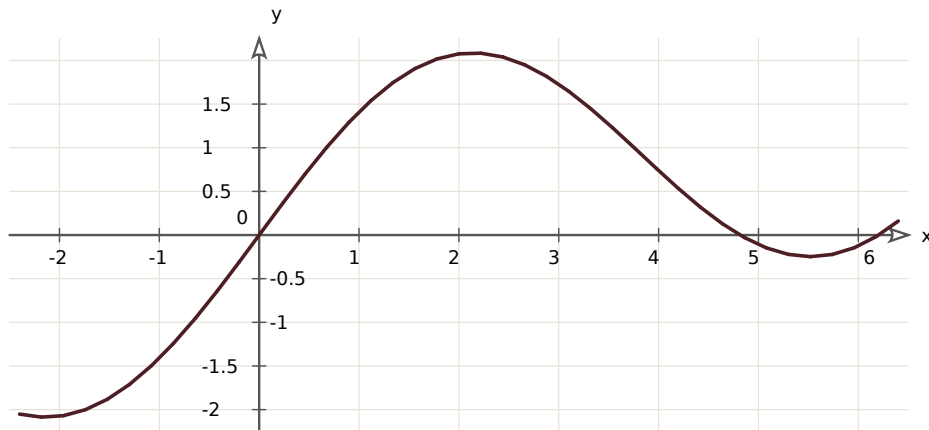


Figure 17.1. Coordinate axes with arrowheads, a grid, and tick labels; bounds derived from the data range.

17.5 EXAMPLE 16

The source `svgc-examples/016-axes-grid/main.c` contains `find_bounds()`, `nice_step()`, `screen_x()`, `screen_y()`, and `draw_grid()`. The same coordinate functions are used to draw the dependence itself. The code has no SVG parser and does not access SVGC internals.

```

/*****
SVG C BOOK EXAMPLE 016: AXES, TICKS AND GRID

Grid steps are chosen by the application, not by SVGC.

```

```

    Plot geometry is calculated from the data bounds.
    *****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

struct sample
{
    double x, y;
};

struct plot
{
    double xmin, xmax, ymin, ymax;
    double left, top, width, height;
};

static double
screen_x( const struct plot *p, double x )
{
    return( p->left + ( x - p->xmin ) * p->width / ( p->xmax - p->xmin ) );
}

static double
screen_y( const struct plot *p, double y )
{
    return( p->top + ( p->ymax - y ) * p->height / ( p->ymax - p->ymin ) );
}

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_line( struct __svg_scene *scene, double x1, double y1,
          double x2, double y2, const char *color, double width )
{
    return( append( scene, __svg_new_line( x1, y1, x2, y2 ),
                  color, "none", width ) );
}

static int
add_label( struct __svg_scene *scene, double x, double y,
           const char *label )
{
    return( append( scene, __svg_new_text( x, y, label, "DejaVu Sans", 14 ),
                  "none", NULL, 0 ) );
}

```

```

/* An approximate 1-2-5 decade ladder for readable major ticks. */
static double
nice_step( double span, int desired_intervals )
{
    double raw, base, fraction;

    raw = span/(double)desired_intervals;
    base = pow( 10.0, floor( log10( raw ) ) );
    fraction = raw/base;
    if( fraction <= 1.0 ) return( base );
    if( fraction <= 2.0 ) return( 2.0*base );
    if( fraction <= 5.0 ) return( 5.0*base );
    return( 10.0*base );
}

/* Snap data bounds outward to half x-cells and quarter y-cells.
   The chosen sample spans produce [-2.5,6.5] by [-2.25,2.25]. */
static void
find_bounds( struct plot *p, const struct sample *a, size_t n )
{
    size_t i;

    p->xmin = p->xmax = a[0].x;
    p->ymin = p->ymax = a[0].y;
    for( i = 1; i < n; ++i )
    {
        if( a[i].x < p->xmin ) p->xmin = a[i].x;
        if( a[i].x > p->xmax ) p->xmax = a[i].x;
        if( a[i].y < p->ymin ) p->ymin = a[i].y;
        if( a[i].y > p->ymax ) p->ymax = a[i].y;
    }
    p->xmin = floor( p->xmin*2.0 )/2.0;
    p->xmax = ceil( p->xmax*2.0 )/2.0;
    p->ymin = floor( p->ymin*4.0 )/4.0;
    p->ymax = ceil( p->ymax*4.0 )/4.0;
}

/* Curved, hollow axis arrowhead, constructed as in Figure 6.1. */
static int
add_axis_head( struct __svg_scene *scene, double x, double y,
               double ux, double uy )
{
    const double px = -uy, py = ux;
    struct __svg_cmds *cmds;
    struct __svg_node *node;

    cmds = __svg_new_cmds();
    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE,
                             x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE, x, y ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE,
                             x+4.5*px-13.5*ux, y+4.5*py-13.5*uy ) < 1 ||
        __svg_add_short_curveto_cmd( cmds, ABSOLUTE,
                                     x-10.0*ux, y-10.0*uy,
                                     x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
}

```

```

node = __svg_new_path( cmds );
if( !node )
{
    __svg_free_cmds( cmds );
    return( 0 );
}
(void)__svg_set_stroke_linecap( node, LINECAP_SQUARE );
(void)__svg_set_stroke_linejoin( node, LINEJOIN_MITER );
return( append( scene, node, "#555555", "white", 1.8 ) );
}

static int
draw_grid( struct __svg_scene *scene, const struct plot *p )
{
    char buf[64];
    double dx = nice_step( p->xmax-p->xmin, 9 );
    double dy = nice_step( p->ymax-p->ymin, 9 );
    double x, y;
    double xzero = screen_x( p, 0.0 ), yzero = screen_y( p, 0.0 );
    int i;

    /* Data-derived bounds give x=-2.5..6.5, y=-2.25..2.25.
       Every grid line is at an integer x or half-integer y. */
    for( i = (int)ceil( p->xmin/dx ); i <= (int)floor( p->xmax/dx ); ++i )
    {
        x = i*dx;
        if( !add_line( scene, screen_x( p, x ), p->top,
                      screen_x( p, x ), p->top+p->height,
                      "#E8E2D8", 0.9 ) ) return( 0 );
        if( !add_line( scene, screen_x( p, x ), yzero-5.0,
                      screen_x( p, x ), yzero+5.0,
                      "#555555", 1.1 ) ) return( 0 );
        if( x != 0.0 )
        {
            double lx = screen_x( p, x )-8;
            if( fabs( x-6.0 ) < 1e-9 ) lx = screen_x( p, x )+4;
            (void)snprintf( buf, sizeof( buf ), "%.5g", x );
            if( !add_label( scene, lx, yzero+22, buf ) ) return( 0 );
        }
    }
    for( i = (int)ceil( p->ymin/dy ); i <= (int)floor( p->ymax/dy ); ++i )
    {
        y = i*dy;
        if( !add_line( scene, p->left, screen_y( p, y ),
                      p->left+p->width, screen_y( p, y ),
                      "#E8E2D8", 0.9 ) ) return( 0 );
        if( fabs( y-2.0 ) > 1e-9 )
        {
            if( !add_line( scene, xzero-5.0, screen_y( p, y ),
                          xzero+5.0, screen_y( p, y ),
                          "#555555", 1.1 ) ) return( 0 );
        }
        if( y != 0.0 && fabs( y-2.0 ) > 1e-9 )
        {
            double lx = xzero-43;
            if( fabs( y+0.5 ) < 1e-9 || fabs( y+1.0 ) < 1e-9 ) lx = xzero+8;
            (void)snprintf( buf, sizeof( buf ), "%.5g", y );
            if( !add_label( scene, lx, screen_y( p, y )+5, buf ) ) return( 0 );
        }
    }
    if( !add_line( scene, p->left, yzero, p->left+p->width,
                  yzero, "#555555", 1.8 ) ) return( 0 );
    if( !add_line( scene, xzero, p->top+p->height, xzero,
                  p->top, "#555555", 1.8 ) ) return( 0 );
    if( !add_axis_head( scene, p->left+p->width, yzero, 1.0, 0.0 ) ||

```

```

    !add_axis_head( scene, xzero, p->top, 0.0, -1.0 ) ) return( 0 );
    if( !add_label( scene, xzero-17, yzero-8, "0" ) ) return( 0 );
    return( 1 );
}

int
main( void )
{
    struct sample      a[41];
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    struct plot        p;
    size_t              i;
    int                 status = 1;

    for( i = 0; i < 41; ++i )
    {
        a[i].x = -2.4 + 0.22*i;
        a[i].y = 1.6*sin( 0.82*a[i].x ) + 0.24*a[i].x;
    }
    p.left = 90;
    p.top = 65;
    p.width = 675;
    p.height = 295;
    find_bounds( &p, a, 41 );

    sheet = (struct __svg_sheet *)__svg_new_sheet( 850, 425, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 850, 425, 0, 0 ),
                "none", "white", 0 ) ) goto done;
    if( !draw_grid( scene, &p ) ) goto done;
    for( i = 1; i < 41; ++i )
    {
        if( !add_line( scene, screen_x( &p, a[i-1].x ), screen_y( &p, a[i-1].y ),
                      screen_x( &p, a[i].x ), screen_y( &p, a[i].y ),
                      "#4D1F24", 2.7 ) ) goto done;
    }
    if( !add_label( scene, p.left+p.width+10, screen_y( &p, 0.0 )+5, "x" ) ) goto done;
    if( !add_label( scene, screen_x( &p, 0.0 )+9, p.top-14, "y" ) ) goto done;

    __svg_paint_scene( scene, 0, "axes-grid.svg" );
    status = 0;
done:
    if( status ) fprintf( stderr, "Cannot construct axes/grid drawing\n" );
    __svg_free_scene( scene );
    return( status );
}

```

```

cd svgc-examples/016-axes-grid
make
# open view.html in a browser

```

17.6 LIMITATIONS OF SIMPLE AUTOMATIC LABELING

The 1–2–5 step family suffices for an introductory numerical plot. Logarithmic axes, secondary scales, geographical coordinates, and very long labels require different tick and layout rules. Also distinguish *including zero in the range* from actually drawing the axes $x = 0$ and $y = 0$: the mathematical origin need not lie within the domain being displayed.

The plotting algorithm determines axes, ticks, and grid from the numerical data. SVGC records the resulting geometry in SVG; it does not make hidden choices about scaling.

18. Discrete Observations and Model Comparison

18.1 OBSERVATIONS ARE NOT A CONTINUOUS FUNCTION

Numerical data are often a finite collection of separate measurements rather than a continuous curve:

$$\{(t_i, y_i)\}_{i=0}^{n-1}, \quad t_0 < t_1 < \dots < t_{n-1}.$$

Having a few samples tells us nothing about system behavior between them. Connecting measurements with an unexplained polyline may lead a reader to mistake interpolation for the actual evolution of the measured quantity.

A scientific plot should therefore distinguish *measurements*, *models*, and *interpolation policies*. Discrete samples can be shown as circles, columns, vertical stems, or specialized markers. A connecting line needs separate justification.

18.2 TWO VIEWS OF THE SAME DATA

The example displays ten measurements in two ways. On the left, each y_i is shown as a point with a vertical segment extending to the baseline $y = 0$. On the right, the same measurements appear as hollow circles over a piecewise-linear model $\hat{y}(t)$.

Model values are calculated independently of measurements:

$$\hat{y}_i = 1.6 + 0.7 t_i.$$

The program and illustration maintain separate measured and model columns. Collapsing both into a single list of screen coordinates would obscure the meaning of their differences.

Both panels use the same domains, $x \in [-0.4, 9.4]$ and $y \in [0, 10]$. Otherwise, comparing the scatter visually would be misleading. Arrowheads mark the positive ends of the axes. The outermost ticks 9 and 10 are not labeled: numbers positioned on arrowheads would impair readability.

18.3 MEASURING DEVIATIONS FROM THE MODEL

Define residuals by

$$r_i = y_i - \hat{y}_i.$$

A common scalar summary is the **root mean square error (RMSE)**:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} r_i^2}.$$

This quantity has the same units as the measured variable. Unlike a signed mean error, positive and negative residuals cannot cancel each other.

For the example data, RMSE is computed directly in C and printed below the right-hand plot. The program does not estimate instrument measurement uncertainty or a confidence interval; these would require an additional statistical model.

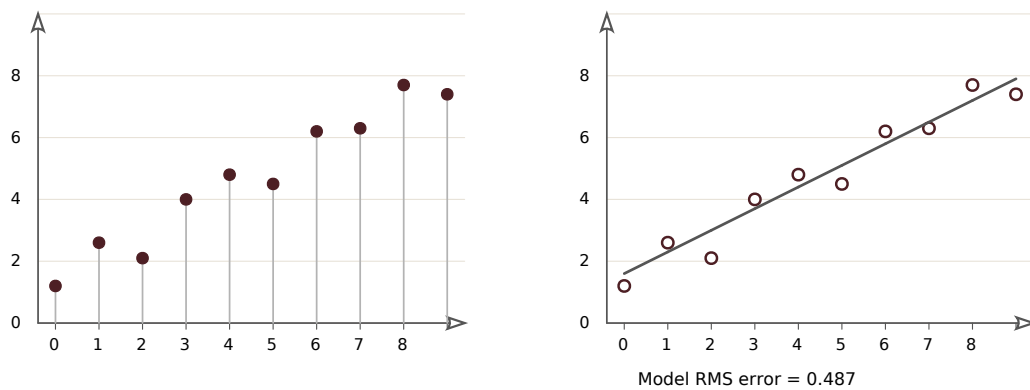


Figure 18.1. Separate discrete observations (left) compared with model values (right).

18.4 EXAMPLE 17

Each source-array element has three fields: time t , the observed value measured, and the predicted value `model`. A plot structure defines mathematical and screen domains. Both panels use the same coordinate-transform functions but contain different graphical objects.

Measured points are separate `circle` objects. The model line on the right connects only model values. This is a *model* curve, not an unjustified interpolation between experimental observations.

```

/*****
SVGC BOOK EXAMPLE 017: DISCRETE DATA AND MODEL COMPARISON

Left: measured samples drawn as independent impulses.
Right: the same samples compared with a model line.
*****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

struct observation
{
    double t, measured, model;
};

struct plot
{
    double left, top, width, height;
    double xmin, xmax, ymin, ymax;
};

static double
plot_x( const struct plot *p, double t )
{
    return( p->left + p->width*( t-p->xmin )/( p->xmax-p->xmin ) );
}

```

```

static double
plot_y( const struct plot *p, double y )
{
    return( p->top + p->height*( p->ymax-y )/( p->ymax-p->ymin ) );
}

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
line( struct __svg_scene *scene, double x1, double y1,
      double x2, double y2, const char *color, double width )
{
    return( append( scene, __svg_new_line( x1, y1, x2, y2 ),
                   color, "none", width ) );
}

static int
label( struct __svg_scene *scene, double x, double y,
       const char *text, double size )
{
    return( append( scene, __svg_new_text( x, y, text, "DejaVu Sans", size ),
                   "none", NULL, 0 ) );
}

/* Curved, hollow axis arrowhead, constructed as in Figure 6.1. */
static int
add_axis_head( struct __svg_scene *scene, double x, double y,
              double ux, double uy )
{
    const double px = -uy, py = ux;
    struct __svg_cmds *cmds;
    struct __svg_node *node;

    cmds = __svg_new_cmds();
    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE,
                             x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE, x, y ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE,
                             x+4.5*px-13.5*ux, y+4.5*py-13.5*uy ) < 1 ||
        __svg_add_short_curveto_cmd( cmds, ABSOLUTE,
                                     x-10.0*ux, y-10.0*uy,
                                     x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    node = __svg_new_path( cmds );
}

```

```

    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    (void)__svg_set_stroke_linecap( node, LINECAP_SQUARE );
    (void)__svg_set_stroke_linejoin( node, LINEJOIN_MITER );
    return( append( scene, node, "#555555", "white", 1.8 ) );
}

static int
axes( struct __svg_scene *scene, const struct plot *p )
{
    char buf[24];
    int i;

    for( i = 0; i <= 10; i += 2 )
    {
        if( !line( scene, p->left, plot_y( p, i ), p->left+p->width,
                  plot_y( p, i ), "#E8E2D8", 0.9 ) ) return( 0 );
        (void)snprintf( buf, sizeof( buf ), "%d", i );
        if( i != 10 && !label( scene, p->left-22, plot_y( p, i )+4, buf, 13 ) ) return( 0 );
    }
    for( i = 0; i <= 9; ++i )
    {
        double x = plot_x( p, i );
        if( !line( scene, x, p->top+p->height,
                  x, p->top+p->height+5, "#555555", 0.9 ) ) return( 0 );
        (void)snprintf( buf, sizeof( buf ), "%d", i );
        if( i != 9 && !label( scene, x-5, p->top+p->height+22, buf, 13 ) ) return( 0 );
    }
    if( !line( scene, p->left, p->top, p->left, p->top+p->height,
              "#555555", 1.4 ) ) return( 0 );
    if( !line( scene, p->left, p->top+p->height, p->left+p->width,
              p->top+p->height, "#555555", 1.4 ) ) return( 0 );
    if( !add_axis_head( scene, p->left, p->top, 0.0, -1.0 ) ||
        !add_axis_head( scene, p->left+p->width,
                        p->top+p->height, 1.0, 0.0 ) ) return( 0 );

    return( 1 );
}

int
main( void )
{
    const struct observation a[] = {
        { 0, 1.2, 1.6 }, { 1, 2.6, 2.3 }, { 2, 2.1, 3.0 },
        { 3, 4.0, 3.7 }, { 4, 4.8, 4.4 }, { 5, 4.5, 5.1 },
        { 6, 6.2, 5.8 }, { 7, 6.3, 6.5 }, { 8, 7.7, 7.2 },
        { 9, 7.4, 7.9 }
    };
    const size_t n = sizeof( a )/sizeof( a[0] );
    struct plot p1 = { 65, 85, 345, 250, -0.4, 9.4, 0.0, 10.0 };
    struct plot p2 = { 525, 85, 345, 250, -0.4, 9.4, 0.0, 10.0 };
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    size_t i;
    double sumsq = 0.0;
    char text[80];
    int status = 1;

    sheet = (struct __svg_sheet *)__svg_new_sheet( 925, 420, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )

```

```

{
    __svg_free_node( (struct __svg_node *)sheet );
    return( 1 );
}
if( !append( scene, __svg_new_rect( 0, 0, 925, 420, 0, 0 ),
            "none", "white", 0 ) ) goto done;
if( !axes( scene, &p1 ) || !axes( scene, &p2 ) ) goto done;

for( i = 0; i < n; ++i )
{
    double x1 = plot_x( &p1, a[i].t );
    double y1 = plot_y( &p1, a[i].measured );
    double x2 = plot_x( &p2, a[i].t );
    double y2 = plot_y( &p2, a[i].measured );
    double err = a[i].measured-a[i].model;

    if( !line( scene, x1, plot_y( &p1, 0 ), x1, y1,
            "#B3B3B3", 1.4 ) ) goto done;
    if( !append( scene, __svg_new_circle( x1, y1, 4.6 ),
            "#4D1F24", "#4D1F24", 1.0 ) ) goto done;
    if( !append( scene, __svg_new_circle( x2, y2, 4.6 ),
            "#4D1F24", "white", 2.0 ) ) goto done;
    if( i > 0 &&
        !line( scene, plot_x( &p2, a[i-1].t ), plot_y( &p2, a[i-1].model ),
            plot_x( &p2, a[i].t ), plot_y( &p2, a[i].model ),
            "#555555", 2.2 ) ) goto done;
    sumsq += err*err;
}
(void)sprintf( text, sizeof( text ), "Model RMS error = %.3f", sqrt( sumsq/n ) );
if( !label( scene, 550, 385, text, 14 ) ) goto done;

__svg_paint_scene( scene, 0, "discrete.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct discrete data diagram\n" );
__svg_free_scene( scene );
return( status );
}

```

```

cd svgc-examples/017-discrete
make
# open view.html in a browser

```

18.5 TRUSTWORTHY VISUAL COMPARISONS

A published experiment should identify the meaning and units of its axes, maintain identical bounds when comparing panels, and clearly distinguish calculations from observations. Units have been omitted in this educational illustration because its data are artificial; for real measurements this would be unacceptable.

A continuous model curve can be sampled more finely than the observations. A high output sampling frequency, however, does not establish model accuracy: it changes only how the function is drawn.

A discrete point represents an observation; a line connecting points represents a separate assumption about continuity. A scientific illustration should make that distinction unmistakable.

19. Natural Cubic Interpolating Splines

19.1 WHY SMOOTHNESS CONDITIONS MATTER

A polyline through tabulated nodes is continuous, but its derivative usually jumps at every node. To represent a sufficiently smooth dependence, we can construct a **cubic interpolating spline** (a *piecewise function made of cubic polynomials*).

Suppose the nodes are

$$x_0 < x_1 < \dots < x_{n-1}, \quad y_0, \dots, y_{n-1}.$$

Each interval $[x_i, x_{i+1}]$ has its own cubic polynomial $S_i(x)$. The interpolation conditions are

$$S_i(x_i) = y_i, \quad S_i(x_{i+1}) = y_{i+1},$$

with the following conditions at each interior node:

$$S'_{i-1}(x_i) = S'_i(x_i), \quad S''_{i-1}(x_i) = S''_i(x_i).$$

The individual polynomials therefore form one C^2 function.

19.2 CHOOSING BOUNDARY CONDITIONS

Interpolation and interior smoothness conditions alone do not determine all the coefficients. The example uses a *natural spline* (**natural spline** (a *spline with zero second derivatives at both endpoints*)):

$$S''(x_0) = S''(x_{n-1}) = 0.$$

This is convenient, but not the only choice. If the first derivatives at the boundaries are known from the physical problem, a **clamped spline** (a *spline with prescribed endpoint derivatives*) is appropriate. Boundary conditions are part of the mathematical model, not of the SVG format.

19.3 THE TRIDIAGONAL SYSTEM

Let $h_i = x_{i+1} - x_i > 0$ and $m_i = S''(x_i)$. For each interior index $i = 1, \dots, n-2$, the equations are

$$h_{i-1}m_{i-1} + 2(h_{i-1} + h_i)m_i + h_im_{i+1} = 6 \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right).$$

Only the main diagonal and its two neighbors contain nonzero matrix entries. A dense Gaussian elimination is unnecessary: the **Thomas algorithm** (*tridiagonal matrix algorithm*) solves the system in time linear in the number of nodes.

The program's upper array holds the coefficients for back substitution, and rhs holds the transformed right-hand side. After the backward pass, m contains the required second derivatives at the nodes. The conditions $m_0 = m_{n-1} = 0$ are built directly into the calculation.

19.4 EVALUATING THE SPLINE

On $[x_i, x_{i+1}]$, let $u = x_{i+1} - x$, $v = x - x_i$, and $h = h_i$. Then

$$S_i(x) = \frac{m_i u^3}{6h} + \frac{m_{i+1} v^3}{6h} + \left(y_i - \frac{m_i h^2}{6}\right) \frac{u}{h} + \left(y_{i+1} - \frac{m_{i+1} h^2}{6}\right) \frac{v}{h}.$$

This formula automatically reproduces the specified endpoint values. The calculated m_i also ensure continuity of the first and second derivatives.

Drawing samples may be much denser than the original nodes. Here every interval is subdivided into 18 equal pieces. This changes only the *rendering step*; it introduces no additional interpolation nodes.

19.5 PROGRAMMATIC VERIFICATION

Before generating an image, independently check basic invariants:

$$\max_i |S(x_i) - y_i| \leq \varepsilon, \quad m_0 = m_{n-1} = 0.$$

The example sets an absolute nodal-residual threshold of 10^{-12} in the chosen data units. Incorrectly ordered nodes or an excessive residual cause the program to exit with an error.

Passing this check does not establish that the spline equals an unknown original continuous function between nodes. It verifies the solution of the *specified interpolation problem*. Meaningful analysis also needs information about the data, noise, and acceptable error.

Alongside the interpolating curve, the figure shows $x = 0$ and $y = 0$ axes crossing at the origin. They are darker than the grid, carry ticks, and end in hollow arrowheads before reaching the grid boundary.

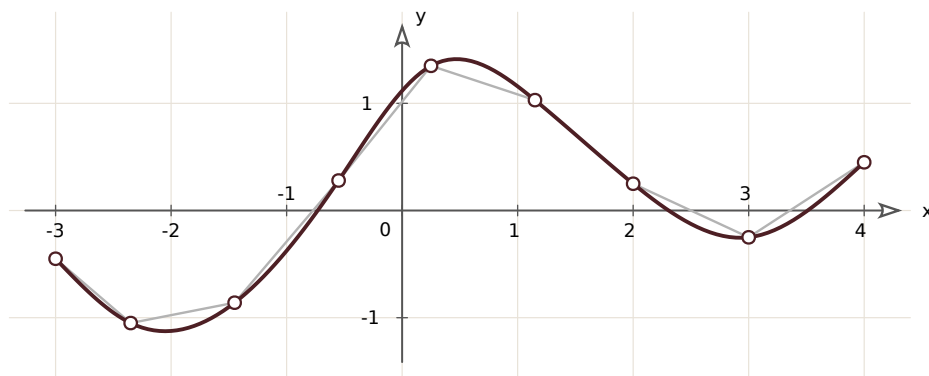


Figure 19.1. Natural cubic spline (dark curve), original nodes, and piecewise-linear interpolation (gray polyline).

19.6 EXAMPLE 18

Program 018-cubic-spline has three logically separate parts: `natural_spline()` calculates the nodal derivatives, `spline_at()` evaluates the spline, and `add_spline()` converts calculated points into SVG geometry.

The main curve is drawn as a polyline with many vertices. This is a finite vector representation of the computed function, but the mathematical interpolation is determined by the cubic polynomial coefficients. The original nodes appear as separate hollow circles. The gray polyline is for comparison and does not belong to the spline algorithm.

```

/*****
SVGC BOOK EXAMPLE 018: NATURAL CUBIC INTERPOLATING SPLINE

Solve the tridiagonal system for knot second derivatives.
Evaluate each cubic interval and draw the computed curve.
*****/

#include <float.h>
#include <math.h>
#include <stdio.h>
#include <svgc.h>

#define N 9
#define STEPS 18

struct knot
{
    double x, y;
};

struct plot
{
    double xmin, xmax, ymin, ymax;
    double left, top, width, height;
};

static double
plot_x( const struct plot *p, double x )
{
    return( p->left + p->width*( x-p->xmin )/( p->xmax-p->xmin ) );
}

static double
plot_y( const struct plot *p, double y )
{
    return( p->top + p->height*( p->ymax-y )/( p->ymax-p->ymin ) );
}

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

```

```

static int
add_label( struct __svg_scene *scene, double x, double y,
           const char *s, double size )
{
    return( append( scene, __svg_new_text( x, y, s, "DejaVu Sans", size ),
                   "none", NULL, 0.0 ) );
}

static int
add_line( struct __svg_scene *scene, double x1, double y1,
          double x2, double y2, const char *stroke, double width )
{
    return( append( scene, __svg_new_line( x1, y1, x2, y2 ),
                   stroke, "none", width ) );
}

/* Curved, hollow axis arrowhead, constructed as in Figure 6.1. */
static int
add_axis_head( struct __svg_scene *scene, double x, double y,
               double ux, double uy )
{
    const double px = -uy, py = ux;
    struct __svg_cmds *cmds;
    struct __svg_node *node;

    cmds = __svg_new_cmds();
    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE,
                             x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE, x, y ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE,
                             x+4.5*px-13.5*ux, y+4.5*py-13.5*uy ) < 1 ||
        __svg_add_short_curveto_cmd( cmds, ABSOLUTE,
                                     x-10.0*ux, y-10.0*uy,
                                     x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    node = __svg_new_path( cmds );
    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    (void)__svg_set_stroke_linecap( node, LINECAP_SQUARE );
    (void)__svg_set_stroke_linejoin( node, LINEJOIN_MITER );
    return( append( scene, node, "#555555", "white", 1.8 ) );
}

/* Natural boundary conditions:  $m[0] = m[N-1] = 0$ . */
static int
natural_spline( const struct knot a[N], double m[N] )
{
    double h[N-1], upper[N], rhs[N];
    double beta, cprev = 0.0, dprev = 0.0;
    int i;

    for( i = 0; i < N-1; ++i )

```

```

{
    h[i] = a[i+1].x-a[i].x;
    if( h[i] <= 0.0 ) return( 0 );
}
m[0] = 0.0;
m[N-1] = 0.0;
upper[0] = rhs[0] = 0.0;

/* Thomas forward elimination for interior second derivatives. */
for( i = 1; i < N-1; ++i )
{
    double b = 2.0*( h[i-1]+h[i] );
    double d = 6.0*( ( a[i+1].y-a[i].y )/h[i]
                    -( a[i].y-a[i-1].y )/h[i-1] );
    beta = b - h[i-1]*cprev;
    if( beta <= DBL_MIN ) return( 0 );
    upper[i] = h[i]/beta;
    rhs[i] = ( d-h[i-1]*dprev )/beta;
    cprev = upper[i];
    dprev = rhs[i];
}
for( i = N-2; i >= 1; --i )
    m[i] = rhs[i]-upper[i]*m[i+1];

return( 1 );
}

static double
spline_at( const struct knot a[N], const double m[N],
           int i, double x )
{
    double h = a[i+1].x-a[i].x;
    double u = a[i+1].x-x;
    double v = x-a[i].x;

    return( m[i]*u*u*u/(6.0*h) + m[i+1]*v*v*v/(6.0*h)
           +( a[i].y-m[i]*h*h/6.0 )*u/h
           +( a[i+1].y-m[i+1]*h*h/6.0 )*v/h );
}

static int
add_spline( struct __svg_scene *scene, const struct plot *p,
            const struct knot a[N], const double m[N] )
{
    struct __svg_points *points;
    struct __svg_node *node;
    int i, k;

    points = __svg_new_points();
    if( !points ) return( 0 );
    for( i = 0; i < N-1; ++i )
    {
        for( k = 0; k < STEPS; ++k )
        {
            double x = a[i].x+( a[i+1].x-a[i].x )*(double)k/STEPS;
            double y = spline_at( a, m, i, x );
            if( __svg_add_point( points, plot_x( p, x ), plot_y( p, y ) ) < 1 )
            {
                __svg_free_points( points );
                return( 0 );
            }
        }
    }
}

```

```

    if( __svg_add_point( points, plot_x( p, a[N-1].x ),
                        plot_y( p, a[N-1].y ) ) < 1 )
    {
        __svg_free_points( points );
        return( 0 );
    }
    node = __svg_new_polyline( points );
    if( !node )
    {
        __svg_free_points( points );
        return( 0 );
    }
    return( append( scene, node, "#4D1F24", "none", 2.9 ) );
}

int
main( void )
{
    const struct knot a[N] = {
        { -3.0, -0.45 }, { -2.35, -1.05 }, { -1.45, -0.86 },
        { -0.55, 0.28 }, { 0.25, 1.35 }, { 1.15, 1.03 },
        { 2.0, 0.25 }, { 3.0, -0.25 }, { 4.0, 0.45 }
    };
    struct plot p = { -3.4, 4.4, -1.55, 1.85, 82, 73, 680, 275 };
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    double m[N], max_error = 0.0;
    int i, status = 1;

    if( !natural_spline( a, m ) )
    {
        fprintf( stderr, "Spline construction failed: invalid knot sequence\n" );
        return( 1 );
    }
    for( i = 0; i < N-1; ++i )
    {
        double e1 = fabs( spline_at( a, m, i, a[i].x ) - a[i].y );
        double e2 = fabs( spline_at( a, m, i, a[i+1].x ) - a[i+1].y );
        if( e1 > max_error ) max_error = e1;
        if( e2 > max_error ) max_error = e2;
    }
    if( max_error > 1e-12 )
    {
        fprintf( stderr, "Spline interpolation residual too large: %.9g\n", max_error );
        return( 1 );
    }
    (void)printf( "Spline: %d knots; max nodal residual %.3g; "
                 "natural endpoints %.3g, %.3g\n", N, max_error, m[0], m[N-1] );

    sheet = (struct __svg_sheet *)__svg_new_sheet( 845, 425, 15, 15, 15, 15 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 845, 425, 0, 0 ),
                "none", "white", 0 ) ) goto done;
    for( i = -3; i <= 4; ++i )
    {
        char buf[16];
        double x = plot_x( &p, i );
        if( !add_line( scene, x, p.top, x, p.top+p.height,

```

```

        "#E8E2D8", 0.9 ) ) goto done;
(void)sprintf( buf, sizeof( buf ), "%d", i );
if( !add_line( scene, x, plot_y( &p, 0.0 )-4, x,
        plot_y( &p, 0.0 )+4, "#555555", 1.0 ) ) goto done;
if( i != 0 )
{
    double ly = plot_y( &p, 0.0 )+20;
    if( i == -1 || i == 3 ) ly = plot_y( &p, 0.0 )-8;
    if( !add_label( scene, x-7, ly, buf, 14 ) ) goto done;
}
}
for( i = -1; i <= 1; ++i )
{
    double y = plot_y( &p, i );
    if( !add_line( scene, p.left, y, p.left+p.width, y,
        "#E8E2D8", 0.9 ) ) goto done;
    if( i != 0 )
    {
        char buf[16];
        (void)sprintf( buf, sizeof( buf ), "%d", i );
        if( !add_line( scene, plot_x( &p, 0 )-4, y,
            plot_x( &p, 0 )+4, y, "#555555", 1.0 ) ||
            !add_label( scene, plot_x( &p, 0 )-31, y+5,
                buf, 14 ) ) goto done;
    }
}
/* Axes cross at (0,0), with free space between tips and grid edges. */
if( !add_line( scene, p.left+12, plot_y( &p, 0 ),
    p.left+p.width-9, plot_y( &p, 0 ), "#555555", 1.5 ) ) goto done;
if( !add_line( scene, plot_x( &p, 0 ), p.top+p.height-11,
    plot_x( &p, 0 ), p.top+11, "#555555", 1.5 ) ) goto done;
if( !add_axis_head( scene, p.left+p.width-9,
    plot_y( &p, 0 ), 1.0, 0.0 ) ||
    !add_axis_head( scene, plot_x( &p, 0 ), p.top+11,
        0.0, -1.0 ) ) goto done;
if( !add_label( scene, plot_x( &p, 0 )-17,
    plot_y( &p, 0 )+19, "0", 14 ) ) goto done;
if( !add_label( scene, p.left+p.width+9,
    plot_y( &p, 0 )+5, "x", 14 ) ) goto done;
if( !add_label( scene, plot_x( &p, 0 )+10,
    p.top+7, "y", 14 ) ) goto done;

/* Piecewise linear interpolation is deliberately a different object. */
for( i = 0; i < N-1; ++i )
{
    if( !add_line( scene, plot_x( &p, a[i].x ), plot_y( &p, a[i].y ),
        plot_x( &p, a[i+1].x ), plot_y( &p, a[i+1].y ),
            "#B3B3B3", 1.8 ) ) goto done;
}
if( !add_spline( scene, &p, a, m ) ) goto done;
for( i = 0; i < N; ++i )
{
    if( !append( scene, __svg_new_circle( plot_x( &p, a[i].x ),
        plot_y( &p, a[i].y ), 4.7 ),
        "#4D1F24", "white", 1.7 ) ) goto done;
}

__svg_paint_scene( scene, 0, "cubic-spline.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct spline illustration\n" );
__svg_free_scene( scene );
return( status );
}

```

```
cd svgc-examples/018-cubic-spline
make
# The program prints the nodal residual and creates an SVG file.
# open view.html in a browser
```

19.7 LIMITS OF INTERPOLATION

Even a cubic spline may exceed the values of neighboring nodes. If the data must remain monotone or nonnegative, ordinary natural interpolation offers no such guarantee. Specialized monotonicity- preserving or constrained interpolants are used in these cases.

A spline is calculated from the data and boundary conditions; SVG is only a means of displaying the resulting function. Verify the numerical interpolation and smoothness conditions before drawing the result.

20. Phase Trajectories and Time Series

20.1 ONE MODEL, TWO KINDS OF SCIENTIFIC PLOT

Computational experiments often require showing both how a quantity changes with time and how a system moves through state space. These plots answer different questions. A **time series** displays one or more state components at successive times. A **phase trajectory** is drawn from the components of the state vector themselves: time determines the order of the points but is not an axis of the phase plane.

Consider the linearly damped oscillator

$$\ddot{x} + 2\gamma\dot{x} + \omega^2x = 0, \quad \gamma > 0, \quad \omega > \gamma.$$

Introducing velocity $v = \dot{x}$ gives a first-order system:

$$\frac{d}{dt} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -\omega^2x - 2\gamma v \end{pmatrix}.$$

This form is convenient for explicit numerical integration. In the source program, x denotes the oscillator's displacement, and v its time derivative.

20.2 THE RUNGE-KUTTA NUMERICAL SCHEME

For the state $\mathbf{y} = (x, v)^T$ and vector right-hand side $\mathbf{f}(\mathbf{y})$, one classical RK4 step uses four slope estimates:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_n), \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_1), \\ \mathbf{k}_3 &= \mathbf{f}(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_2), \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{y}_n + h\mathbf{k}_3), \end{aligned}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4).$$

Each component is added independently. `advance()` constructs an intermediate state, `rhs()` evaluates the right-hand side, and `rk4_step()` performs one full step. Here $h = 0.04$, $N = 480$, and $(x_0, v_0) = (1, 0)$, giving a total simulated time of 19.2.

20.3 INDEPENDENT ANALYTICAL VERIFICATION

For the selected initial conditions and underdamping $\gamma < \omega$, the exact solution is

$$\begin{aligned} \beta &= \sqrt{\omega^2 - \gamma^2}, \\ x(t) &= e^{-\gamma t} \left(\cos \beta t + \frac{\gamma}{\beta} \sin \beta t \right), \\ v(t) &= -\frac{\omega^2}{\beta} e^{-\gamma t} \sin \beta t. \end{aligned}$$

At each step the program evaluates the analytical state and calculates the Euclidean error

$$\varepsilon = \max_{0 \leq n \leq N} \sqrt{(x_n - x(t_n))^2 + (v_n - v(t_n))^2}.$$

The observed verification value is approximately $6.14 \cdot 10^{-7}$, below the specified limit 10^{-5} . If the limit is exceeded, the program does not write SVG.

The physical invariant of nonincreasing energy is also checked:

$$E(t) = \frac{1}{2}(v(t)^2 + \omega^2 x(t)^2), \quad \frac{dE}{dt} = -2\gamma v^2 \leq 0.$$

Numerical energy values can differ because of rounding. The sequence check tolerates an absolute upward deviation of at most 10^{-12} . This is not a mathematical proof of RK4 stability for arbitrary step sizes, but it is a useful check of the present experiment.

20.4 FROM A TIME SERIES TO THE PHASE PLANE

The upper panel maps pairs (t_n, x_n) to screen coordinates: time is horizontal, displacement vertical. The lower panel uses the same arrays but plots (x_n, v_n) . The result is a gradually contracting spiral, characteristic of damped oscillations. In the phase plane, the axes $x = 0$ and $v = 0$ pass through the origin without reaching the outer grid boundary. Their positive ends have arrowheads like those in Figure 6.1. This distinguishes mathematical axes from a rectangular plotting frame.

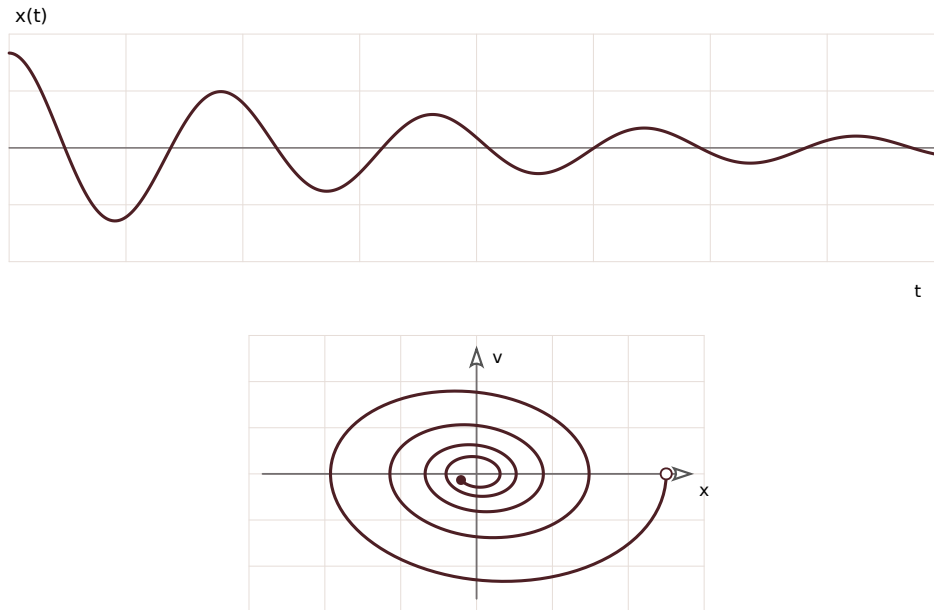


Figure 20.1. Damped oscillator: displacement time series and phase trajectory computed from the same RK4 state sequence.

20.5 EXAMPLE 19

The `019-phase-portrait` program first computes an array of $N + 1$ state values. Only after the numerical checks pass does it create the SVG scene. The same `add_curve()` function draws two different polylines: `phase=0` maps (t, x) , while `phase=1` maps (x, v) .

Both panels use the same affine screen-coordinate formula but different display rectangles and data bounds. The starting point of the phase trajectory is a hollow circle, and its final point is filled.

```

/*****
SVGC BOOK EXAMPLE 019: DAMPED OSCILLATOR, RK4 AND PHASE PORTRAIT

Integrate a two-component differential equation and display
time history and phase-plane trajectory using the same samples.
*****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

#define N_STEPS 480
#define STEP 0.04
#define OMEGA 1.45
#define GAMMA 0.12

struct state
{
    double x, v;
};

struct panel
{
    double left, top, width, height;
    double xmin, xmax, ymin, ymax;
};

static double
px( const struct panel *p, double x )
{
    return( p->left + p->width*(x-p->xmin)/(p->xmax-p->xmin) );
}

static double
py( const struct panel *p, double y )
{
    return( p->top + p->height*(p->ymax-y)/(p->ymax-p->ymin) );
}

static struct state
rhs( struct state a )
{
    return( (struct state){ a.v, -OMEGA*OMEGA*a.x-2.0*GAMMA*a.v } );
}

static struct state
advance( struct state a, struct state k, double h )
{
    return( (struct state){ a.x+h*k.x, a.v+h*k.v } );
}

static struct state
rk4_step( struct state a, double h )
{
    struct state k1 = rhs( a );
    struct state k2 = rhs( advance( a, k1, h/2.0 ) );
    struct state k3 = rhs( advance( a, k2, h/2.0 ) );

```

```

    struct state k4 = rhs( advance( a, k3, h ) );

    return( (struct state){
        a.x+h*(k1.x+2.0*k2.x+2.0*k3.x+k4.x)/6.0,
        a.v+h*(k1.v+2.0*k2.v+2.0*k3.v+k4.v)/6.0
    } );
}

static struct state
exact_solution( double t )
{
    double b = sqrt( OMEGA*OMEGA-GAMMA*GAMMA );
    double d = exp( -GAMMA*t );

    return( (struct state){
        d*(cos( b*t )+GAMMA*sin( b*t )/b),
        -d*OMEGA*OMEGA*sin( b*t )/b
    } );
}

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
add_line( struct __svg_scene *s, double x1, double y1,
          double x2, double y2, const char *color, double w )
{
    return( append( s, __svg_new_line( x1, y1, x2, y2 ),
                   color, "none", w ) );
}

static int
add_text( struct __svg_scene *s, double x, double y,
          double size, const char *text )
{
    return( append( s, __svg_new_text( x, y, text, "DejaVu Sans", size ),
                   "none", NULL, 0.0 ) );
}

static int
add_curve( struct __svg_scene *scene, const struct panel *p,
           const struct state a[N_STEPS+1], int phase )
{
    struct __svg_points *points = __svg_new_points();
    struct __svg_node *node;
    int i;

    if( !points ) return( 0 );
    for( i = 0; i <= N_STEPS; ++i )
    {

```

```

    double xx = phase ? a[i].x : i*STEP;
    double yy = phase ? a[i].v : a[i].x;
    if( __svg_add_point( points, px( p, xx ), py( p, yy ) ) < 1 )
    {
        __svg_free_points( points );
        return( 0 );
    }
}

node = __svg_new_polyline( points );
if( !node )
{
    __svg_free_points( points );
    return( 0 );
}
return( append( scene, node, "#4D1F24", "none", 2.5 ) );
}

/* Curved, hollow axis arrowhead, constructed as in Figure 6.1. */
static int
add_axis_head( struct __svg_scene *scene, double x, double y,
               double ux, double uy )
{
    const double px = -uy, py = ux;
    struct __svg_cmds *cmds;
    struct __svg_node *node;

    cmds = __svg_new_cmds();
    if( !cmds ) return( 0 );
    if( __svg_add_moveto_cmd( cmds, ABSOLUTE,
                             x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE, x, y ) < 1 ||
        __svg_add_lineto_cmd( cmds, ABSOLUTE,
                             x+4.5*px-13.5*ux, y+4.5*py-13.5*uy ) < 1 ||
        __svg_add_short_curveto_cmd( cmds, ABSOLUTE,
                                     x-10.0*ux, y-10.0*uy,
                                     x-4.5*px-13.5*ux, y-4.5*py-13.5*uy ) < 1 ||
        __svg_add_closepath_cmd( cmds, ABSOLUTE ) < 1 )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    node = __svg_new_path( cmds );
    if( !node )
    {
        __svg_free_cmds( cmds );
        return( 0 );
    }
    (void)__svg_set_stroke_linecap( node, LINECAP_SQUARE );
    (void)__svg_set_stroke_linejoin( node, LINEJOIN_MITER );
    return( append( scene, node, "#555555", "white", 1.8 ) );
}

static int
add_panel( struct __svg_scene *scene, const struct panel *p,
           int nx, int ny, int phase )
{
    int i;
    for( i = 0; i <= nx; ++i )
    {
        double x = p->left+p->width*(double)i/nx;
        if( !add_line( scene, x, p->top, x, p->top+p->height,
                       "#E6DDD8", 0.8 ) ) return( 0 );
    }
}

```

```

for( i = 0; i <= ny; ++i )
{
    double y = p->top+p->height*(double)i/ny;
    if( !add_line( scene, p->left, y, p->left+p->width, y,
                  "#E6DDD8", 0.8 ) ) return( 0 );
}
if( phase )
{
    /* The phase-plane origin is (0,0); axes stop before plot edges. */
    double ox = px( p, 0.0 ), oy = py( p, 0.0 );
    if( !add_line( scene, p->left+11, oy, p->left+p->width-11,
                  oy, "#746F70", 1.5 ) ||
        !add_line( scene, ox, p->top+p->height-11, ox,
                  p->top+11, "#746F70", 1.5 ) ) return( 0 );
    if( !add_axis_head( scene, p->left+p->width-11,
                      oy, 1.0, 0.0 ) ||
        !add_axis_head( scene, ox, p->top+11,
                      0.0, -1.0 ) ) return( 0 );
}
else if( p->ymin < 0.0 && p->ymax > 0.0 &&
        !add_line( scene, p->left, py( p, 0 ), p->left+p->width,
                  py( p, 0 ), "#746F70", 1.2 ) ) return( 0 );
return( 1 );
}

int
main( void )
{
    struct state a[N_STEPS+1] = { [0] = { 1.0, 0.0 } };
    const struct panel time_plot = { 80, 90, 760, 185, 0, N_STEPS*STEP, -1.2, 1.2 };
    const struct panel phase_plot = { 275, 335, 370, 225, -1.2, 1.2, -1.65, 1.65 };
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    double err = 0.0, last_energy;
    int i, status = 1;

    last_energy = 0.5*OMEGA*OMEGA;
    for( i = 1; i <= N_STEPS; ++i )
    {
        double e, energy;
        struct state ref;

        a[i] = rk4_step( a[i-1], STEP );
        ref = exact_solution( i*STEP );
        e = hypot( a[i].x-ref.x, a[i].v-ref.v );
        if( e > err ) err = e;
        energy = 0.5*(a[i].v*a[i].v+OMEGA*OMEGA*a[i].x*a[i].x);
        if( energy > last_energy+1e-12 )
        {
            fprintf( stderr, "Energy increased at step %d\n", i );
            return( 1 );
        }
        last_energy = energy;
    }
    if( err > 1e-5 )
    {
        fprintf( stderr, "RK4 analytic error too large: %.12g\n", err );
        return( 1 );
    }
    (void)printf( "RK4 %d steps: max state error %.4g; E(end)/E(0)=%.7g\n",
                 N_STEPS, err, last_energy/(0.5*OMEGA*OMEGA) );

    sheet = (struct __svg_sheet *)__svg_new_sheet( 920, 670, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );

```

```

scene = __svg_new_scene( sheet );
if( !scene )
{
    __svg_free_node( (struct __svg_node *)sheet );
    return( 1 );
}
if( !append( scene, __svg_new_rect( 0, 0, 920, 670, 0, 0 ),
    "none", "white", 0 ) ) goto done;
if( !add_text( scene, 85, 80, 15, "x(t)" ) ) goto done;
if( !add_panel( scene, &time_plot, 8, 4, 0 ) ) goto done;
if( !add_curve( scene, &time_plot, a, 0 ) ) goto done;
if( !add_text( scene, 816, 304, 14, "t" ) ) goto done;
if( !add_panel( scene, &phase_plot, 6, 6, 1 ) ) goto done;
if( !add_curve( scene, &phase_plot, a, 1 ) ) goto done;
if( !append( scene, __svg_new_circle( px( &phase_plot, a[0].x ),
    py( &phase_plot, a[0].v ), 4.5 ),
    "#4D1F24", "white", 1.5 ) ) goto done;
if( !append( scene, __svg_new_circle( px( &phase_plot, a[N_STEPS].x ),
    py( &phase_plot, a[N_STEPS].v ), 3.5 ),
    "#4D1F24", "#4D1F24", 1 ) ) goto done;
if( !add_text( scene, phase_plot.left+phase_plot.width-4,
    py( &phase_plot, 0.0 )+18, 14, "x" ) ) goto done;
if( !add_text( scene, px( &phase_plot, 0.0 )+13,
    phase_plot.top+22, 14, "v" ) ) goto done;

__svg_paint_scene( scene, 0, "phase-portrait.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct phase portrait\n" );
__svg_free_scene( scene );
return( status );
}

```

```

cd svgc-examples/019-phase-portrait
make
# Check the RK4 error and decreasing energy reported by the program.
# Open view.html in a browser.

```

20.6 WHAT THE LINE ACTUALLY REPRESENTS

An SVG line is a finite polyline with $N + 1$ vertices, not an analytical curve between integration nodes. Reducing the step size changes both the numerical trajectory and the detail of its SVG representation. Scientific analysis must distinguish these two effects: *integration error* and *geometric rendering error*.

A phase portrait and time series should be derived from the same verified trajectory. The numerical algorithm determines the system state; SVGC only converts the calculated data into vector lines.

21. Contour Lines of Functions of Two Variables

21.1 FROM A SCALAR FIELD TO A LEVEL SET

For a real function $F(x, y)$, a **contour line** (*isoline*) at level c is defined not by an inequality or a pixel grid, but by the equation

$$F(x, y) = c.$$

Elevation contours on a topographic map and equal-potential lines in mathematical physics are examples. On a computer, however, function values are usually known only at the nodes of a finite grid. Contours must be reconstructed by interpolating between neighboring nodes.

Our example uses the positive-definite quadratic form

$$F(x, y) = \frac{x^2}{4} + \frac{y^2}{2} + 0.2xy.$$

For $c > 0$, its level sets are nested rotated ellipses. This field is convenient for testing geometry: the levels 0.35, 0.80, 1.45, and 2.35 do not intersect and should appear in order from the center outward.

21.2 A RECTANGULAR COMPUTATIONAL GRID

Choose the rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ and integers N_x, N_y . Grid nodes are

$$\begin{aligned} x_i &= x_{\min} + i\Delta x, & \Delta x &= \frac{x_{\max} - x_{\min}}{N_x}, \\ y_j &= y_{\min} + j\Delta y, & \Delta y &= \frac{y_{\max} - y_{\min}}{N_y}. \end{aligned}$$

The array `f[j][i]` stores field values at exactly $(N_x + 1)(N_y + 1)$ nodes. The example uses $N_x = 36$ and $N_y = 32$. Between nodes, it does not evaluate the analytical function again: the contours are reconstructed from previously calculated nodal values.

21.3 THE MARCHING SQUARES ALGORITHM

Marching Squares (*a cell-by-cell traversal of a rectangular grid*) processes every cell independently. Its four corner values are classified by the test $F > c$, yielding a four-bit mask. An edge is crossed by the contour when its endpoint values are on opposite sides of the chosen level.

For edge endpoints A, B and values F_A, F_B , the crossing point is found by linear interpolation:

$$\alpha = \frac{c - F_A}{F_B - F_A}, \quad P = A + \alpha(B - A).$$

A crossing detected with strict comparisons has $0 \leq \alpha \leq 1$. The algorithm has no knowledge of the analytic curve within the cell; it connects crossings with straight segments to obtain a piecewise-linear approximation.

21.4 AMBIGUOUS CELLS

Special care is needed when diagonally opposite corners have the same classification and adjacent corners have different classifications. Four edge crossings occur, and there are two possible pairings. These are the cases with masks 5 and 10.

The demonstration uses a simple rule: the value of the bilinear interpolant at the cell center (the average of its four nodal values) determines the edge pairing. This is adequate for the smooth field chosen here, but it is *not a full asymptotic decider*. Fields containing saddles or requiring strict topological correctness call for the full criterion based on the saddle value of the bilinear interpolant.

21.5 CONTOURS AS COMPUTED RESULTS

Each contour segment is transformed from mathematical to screen coordinates and then written as an SVG line. Four shades from the book's palette distinguish the levels. Higher level values form the outer lines, and lower values form the inner ones.

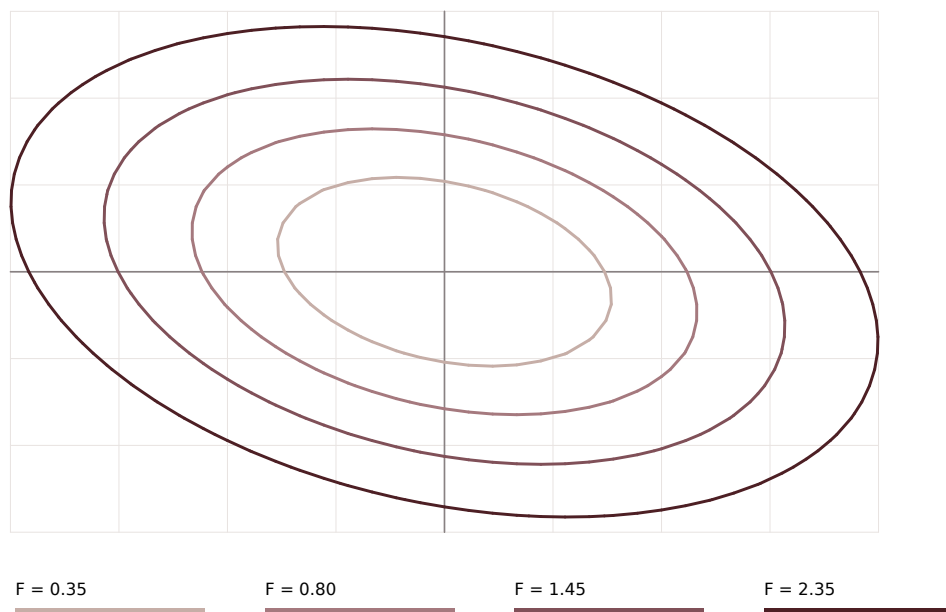


Figure 21.1. Four level sets of a quadratic form calculated by Marching Squares on a rectangular grid.

21.6 EXAMPLE 20

The `020-contours` program separates field evaluation (`field()`), edge interpolation (`crossing()`), cell traversal (`march_cell()`), and rendering of one segment (`segment()`). On success, it prints the number of segments found for each of the four levels.

```

/*****
SVGC BOOK EXAMPLE 020: ISOLINES BY MARCHING SQUARES

Sample a smooth quadratic function on a Cartesian grid.
Interpolate crossings on cell edges and connect level sets.
*****/

#include <math.h>
#include <stdio.h>
#include <svgc.h>

```

```

#define NX 36
#define NY 32
#define NLEVELS 4

struct point
{
    double x, y;
};

struct plot
{
    double xmin, xmax, ymin, ymax;
    double left, top, width, height;
};

static double
field( double x, double y )
{
    return( x*x/4.0+y*y/2.0+0.20*x*y );
}

static double
px( const struct plot *p, double x )
{
    return( p->left+p->width*(x-p->xmin)/(p->xmax-p->xmin) );
}

static double
py( const struct plot *p, double y )
{
    return( p->top+p->height*(p->ymax-y)/(p->ymax-p->ymin) );
}

static int
append( struct __svg_scene *scene, struct __svg_node *node,
        const char *stroke, const char *fill, double width )
{
    if( !node ) return( 0 );
    if( stroke ) (void)__svg_set_stroke( node, stroke );
    if( fill ) (void)__svg_set_fill( node, fill );
    if( width > 0.0 ) (void)__svg_set_stroke_width( node, width );
    if( __svg_scene_add_node( scene, node ) == 1 ) return( 1 );
    __svg_free_node( node );
    return( 0 );
}

static int
segment( struct __svg_scene *scene, const struct plot *p,
        struct point a, struct point b, const char *color, double width )
{
    return( append( scene,
        __svg_new_line( px( p, a.x ), py( p, a.y ),
            px( p, b.x ), py( p, b.y ) ),
        color, "none", width ) );
}

static int
label( struct __svg_scene *scene, double x, double y,
        const char *text, double size )

```

```

{
    return( append( scene, __svg_new_text( x, y, text, "DejaVu Sans", size ),
                  "none", NULL, 0.0 ) );
}

static struct point
crossing( struct point a, struct point b,
          double fa, double fb, double level )
{
    double alpha = (level-fa)/(fb-fa);
    return( (struct point){ a.x+alpha*(b.x-a.x),
                           a.y+alpha*(b.y-a.y) } );
}

static int
march_cell( struct __svg_scene *scene, const struct plot *plot,
             struct point corners[4], double values[4],
             double level, const char *color, int *count )
{
    const int from[4] = { 0, 1, 3, 0 };
    const int to[4]   = { 1, 2, 2, 3 };
    struct point edge[4];
    int active[4], order[4], n = 0, e;

    for( e = 0; e < 4; ++e )
    {
        int i = from[e], j = to[e];
        active[e] = (values[i] > level) != (values[j] > level);
        if( active[e] )
        {
            edge[e] = crossing( corners[i], corners[j], values[i], values[j], level );
            order[n++] = e;
        }
    }
    if( n == 0 ) return( 1 );
    if( n == 2 )
    {
        if( !segment( scene, plot, edge[order[0]], edge[order[1]],
                      color, 2.3 ) ) return( 0 );
        ++count;
        return( 1 );
    }
    if( n != 4 ) return( 0 );

    /* Asymptotic-style center decision for ambiguous 5/10 cases.
       A bilinear cell interpolant has its center at the mean value.
       Center classification determines which pair of branches joins.
       For more demanding saddle geometry one can use the full
       asymptotic decider based on the bilinear saddle value. */
    {
        int mask = (values[0] > level) |
                   ((values[1] > level) << 1) |
                   ((values[2] > level) << 2) |
                   ((values[3] > level) << 3);
        int high_center = (values[0]+values[1]+values[2]+values[3])/4.0 > level;
        int pair01 = (mask == 5 && high_center) || (mask == 10 && !high_center);
        int a = 0;
        int b = pair01 ? 1 : 3;
        int c = pair01 ? 2 : 1;
        int d = pair01 ? 3 : 2;

        if( !segment( scene, plot, edge[a], edge[b], color, 2.3 ) ||
            !segment( scene, plot, edge[c], edge[d], color, 2.3 ) ) return( 0 );
    }
}

```

```

    *count += 2;
}
return( 1 );
}

int
main( void )
{
    const struct plot p = { -3.2, 3.2, -2.4, 2.4, 76, 70, 675, 405 };
    const double levels[NLEVELS] = { 0.35, 0.8, 1.45, 2.35 };
    const char *colors[NLEVELS] = { "#C6AEA7", "#A5797E", "#805058", "#4D1F24" };
    double f[NY+1][NX+1];
    struct __svg_sheet *sheet;
    struct __svg_scene *scene;
    struct point corners[4];
    double vals[4];
    int i, j, k, count[NLEVELS] = { 0 }, status = 1;

    for( j = 0; j <= NY; ++j )
    {
        double y = p.ymin+(p.ymax-p.ymin)*(double)j/NY;
        for( i = 0; i <= NX; ++i )
        {
            double x = p.xmin+(p.xmax-p.xmin)*(double)i/NX;
            f[j][i] = field( x, y );
            if( !isfinite( f[j][i] ) ) return( 1 );
        }
    }

    sheet = (struct __svg_sheet *)__svg_new_sheet( 880, 575, 20, 20, 20, 20 );
    if( !sheet ) return( 1 );
    scene = __svg_new_scene( sheet );
    if( !scene )
    {
        __svg_free_node( (struct __svg_node *)sheet );
        return( 1 );
    }
    if( !append( scene, __svg_new_rect( 0, 0, 880, 575, 0, 0 ),
                "none", "white", 0 ) ) goto done;
    for( i = 0; i <= 8; ++i )
    {
        struct point a = { p.xmin+(p.xmax-p.xmin)*i/8.0, p.ymin };
        struct point b = { a.x, p.ymax };
        if( !segment( scene, &p, a, b, "#E8E3E1", 0.7 ) ) goto done;
    }
    for( j = 0; j <= 6; ++j )
    {
        struct point a = { p.xmin, p.ymin+(p.ymax-p.ymin)*j/6.0 };
        struct point b = { p.xmax, a.y };
        if( !segment( scene, &p, a, b, "#E8E3E1", 0.7 ) ) goto done;
    }
    if( !segment( scene, &p, (struct point){p.xmin,0},
                (struct point){p.xmax,0}, "#888182", 1.3 ) ||
        !segment( scene, &p, (struct point){0,p.ymin},
                (struct point){0,p.ymax}, "#888182", 1.3 ) ) goto done;

    for( k = 0; k < NLEVELS; ++k )
    {
        for( j = 0; j < NY; ++j )
        {
            double y0 = p.ymin+(p.ymax-p.ymin)*(double)j/NY;
            double y1 = p.ymin+(p.ymax-p.ymin)*(double)(j+1)/NY;
            for( i = 0; i < NX; ++i )
            {

```

```

    double x0 = p.xmin+(p.xmax-p.xmin)*(double)i/NX;
    double x1 = p.xmin+(p.xmax-p.xmin)*(double)(i+1)/NX;
    corners[0] = (struct point){ x0, y0 };
    corners[1] = (struct point){ x1, y0 };
    corners[2] = (struct point){ x1, y1 };
    corners[3] = (struct point){ x0, y1 };
    vals[0] = f[j][i];
    vals[1] = f[j][i+1];
    vals[2] = f[j+1][i+1];
    vals[3] = f[j+1][i];
    if( !march_cell( scene, &p, corners, vals,
                    levels[k], colors[k], &count[k] ) ) goto done;
    }
}
}
for( k = 0; k < NLEVELS; ++k )
{
    char title[80];
    (void)sprintf( title, sizeof(title), "F = %.2f", levels[k] );
    if( count[k] == 0 ) goto done;
    if( !label( scene, 80+k*194, 524, title, 13 ) ) goto done;
    if( !append( scene, __svg_new_rect( 80+k*194, 534, 147, 4, 0, 0 ),
            "none", colors[k], 0 ) ) goto done;
    (void)printf( "level=%.2f segments=%d\n", levels[k], count[k] );
}

__svg_paint_scene( scene, 0, "contours.svg" );
status = 0;
done:
if( status ) fprintf( stderr, "Cannot construct contour map\n" );
__svg_free_scene( scene );
return( status );
}

```

```

cd svgc-examples/020-contours
make
# The program prints the number of contour segments at each level.
# Open view.html in a browser.

```

21.7 SMOOTHING AND GEOMETRIC ACCURACY

The example retains straight contour segments. Refining the grid makes contours appear smoother, but increases the number of SVG objects. An alternative is to join segments into ordered polylines and only then apply smoothing or Bézier approximation. This must be done carefully: unconstrained smoothing may displace the contour away from the prescribed level $F = c$ or even change its topology. *Numerical fidelity* matters more than decorative smoothness.

A contour line must be calculated from the scalar field and selected level. SVG does not replace the algorithm that locates crossings; it only records the resulting geometry.

22. Dynamic Properties of the Gauss–Legendre Method

22.1 FROM A NUMERICAL METHOD TO ITS STABILITY FUNCTION

The structural and geometric facilities of SVG are particularly valuable when a figure presents the result of genuine research. This chapter examines the three-stage, sixth-order Gauss–Legendre **implicit Runge–Kutta method**. We construct its region of dynamic consistency and regions localizing the zeros and poles of its stability function. The formulation and example follow the material at <https://dynamics.su/methods/gauss-legendre-33/>.

For the test differential equation

$$y' = \lambda y, \quad z = h\lambda,$$

a numerical step can be written as

$$y_{n+1} = R(z)y_n.$$

Here h is the integration step, λ a complex test parameter, and $R(z)$ the method's **stability function** (*stability function*). For this three-stage method it is the rational function

$$R(z) = \frac{1 + \frac{1}{2}z + \frac{1}{10}z^2 + \frac{1}{120}z^3}{1 - \frac{1}{2}z + \frac{1}{10}z^2 - \frac{1}{120}z^3}.$$

This is the diagonal $[3/3]$ Padé approximant to e^z . Define

$$P(z) = 1 + \frac{1}{2}z + \frac{1}{10}z^2 + \frac{1}{120}z^3, \quad Q(z) = P(-z).$$

The source `desc/g1_33_6.c` stores the coefficients of P and Q in long double arrays and evaluates $R(z)$ with Horner's scheme over complex numbers.

22.2 GEOMETRY OF ZEROS AND POLES

The zeros of $R(z)$ are roots of $P(z)$, and its poles are roots of $Q(z)$, assuming no common factors cancel. The identity $Q(z) = P(-z)$ maps each zero ζ to a pole $-\zeta$; real coefficients also imply complex-conjugate root pairs. Consequently, the zeros of this function lie in the left half-plane and its poles in the right half-plane. Their placement follows the *specific* stability function; it is not an arbitrary choice of graphic symbols.

Numerically, the cubic roots are approximately

$$\begin{aligned} \text{zeros:} & \quad -4.64437, \quad -3.67781 \pm 3.50876i, \\ \text{poles:} & \quad 4.64437, \quad 3.67781 \pm 3.50876i. \end{aligned}$$

These values support independent checks of the plotted regions. The `rk-dynamics` program itself calculates *level contours*; it neither solves for these roots nor marks them with special symbols.

For the Gauss–Legendre method, $|R(z)| \leq 1$ throughout the left half-plane (A-stability), and $|R(z)| = 1$ on the imaginary axis. The region of absolute stability, however, must not be confused with the region in which a numerical solution is *dynamically consistent* with the exact solution.

22.3 THE LOGARITHMIC DYNAMIC CHARACTERISTIC

Following the definition used at <https://dynamics.su/>, consider the complex **dynamic characteristic** (*dynamic characteristic*)

$$D(z) = \ln R(z), \quad z = \mu + i\nu.$$

For the exact exponential solution, the corresponding quantity is $D_*(z) = z$. Thus

$$D_R(z) = \Re D(z), \quad D_I(z) = \Im D(z)$$

describe the amplitude and phase components of the deviation from the ideal characteristic

$$D_{R*}(z) = \mu, \quad D_{I*}(z) = \nu.$$

The C program uses `clogl()` to calculate the principal branch of the complex logarithm. Away from zeros and poles, its real part is unambiguous:

$$D_R(z) = \ln |R(z)|.$$

The imaginary part depends on the chosen argument branch; this matters when examining phase errors away from the origin.

22.4 THE ONE-PERCENT REGION OF DYNAMIC CONSISTENCY

At each point of the complex plane, define the relative errors of the real and imaginary characteristics (in percent):

$$\delta_R(z) = 100 \left| \frac{D_R(z) - \mu}{\mu} \right|, \\ \delta_I(z) = 100 \left| \frac{D_I(z) - \nu}{\nu} \right|.$$

Where $\mu = 0$ or $\nu = 0$, these expressions require special handling of limits: direct division by a zero coordinate is invalid. The example's `delta_contours()` does not evaluate the function along the four coordinate rays and uses neighboring samples when forming the closed contour.

The region in which both conditions

$$\delta_R(z) \leq 1\%, \quad \delta_I(z) \leq 1\%,$$

hold is called here the *one-percent region of dynamic consistency*. Its boundary is drawn in green. This is a much more demanding measure of how faithfully the dynamics are reproduced than $|R(z)| \leq 1$ alone.

To trace this boundary, `delta_contours()` sweeps directions in increments of 1° and increases the radius along each ray in steps of 0.025. The last point satisfying both one-percent limits is included in the resulting contour. This line is a *grid approximation*, not an analytically exact boundary; the radial step and behavior near the axes need separate checks.

22.5 REGIONS LOCALIZING ZEROS AND POLES

For colored level contours, introduce the shifted real characteristic

$$F_\sigma(z) = D_R(z) + \sigma = \ln |R(z)| + \sigma.$$

The equation $F_\sigma(z) = 0$ is equivalent to

$$|R(z)| = e^{-\sigma}.$$

For $\sigma > 0$, this is a small modulus level that isolates regions near zeros. For $\sigma < 0$, it is a large modulus level that isolates regions near poles. Figure 22.1 shows the small-level contours in red and the large-level contours in blue.

Why do bounded components of these levels contain singularities? In a domain without zeros or poles, $\ln |R(z)|$ is harmonic and cannot possess a strict interior maximum or minimum. A bounded small-level component whose interior has modulus below the boundary value must therefore contain a zero, provided the boundary

has no other singularities. The analogous argument for a large-level component yields a pole. This explains the localization principle; but a *numerically obtained* contour becomes a rigorous guarantee only after verifying closure, classification of its interior, and absence of missed singularities. The argument principle can also be used to verify root counts.

The program first uses `cauchy_radius()` to estimate a common root bound for numerator and denominator. The crude Cauchy bound in this example is 121, safely exceeding all root moduli. `dx_radius()` and `kx_radius()` then search for a much smaller working radius and radial shifts. A verified run yields

$$r_{\text{work}} = 5.5, \quad \sigma_{\text{zero}} = 4.3, \quad \sigma_{\text{pole}} = -4.3.$$

These are results of the numerical search, not substitutes for the rigorous Cauchy bound.

22.6 MARCHING SQUARES FOR STABILITY-FUNCTION CONTOURS

To obtain closed components of the levels $F_\sigma = 0$, the example uses **Marching Squares**, introduced in Section 21.3. On a uniform grid, `rdx_sign()` evaluates the sign of $\Re \ln R(z) + \sigma$; `find_contours()` then selects intersected cells and joins segments into contours.

The essential difference from Chapter 21 is the nature of the scalar field. Here it has logarithmic singularities at zeros and poles of a rational function. Grid-step control, verification of closed components, and careful interpretation of smoothing are therefore essential. A geometrically smoothed line is not a new solution of $F_\sigma = 0$.

22.7 FIGURE 22.1 AND THE COMPLETE C EXAMPLE

Program 021-rk-dynamics incorporates the supplied research example without changing its mathematical algorithms. The main source defines the rational stability function, logarithmic characteristic, radial procedures, and the dynamic-consistency region. The modules `marching-squares.c` and `svg-paint.c` extract contours and write SVG; their headers `marching-squares.h` and `svg-paint.h`, and the `desc/*.c` collection, are retained with the example. `desc/gl_33_6.c` is the active method description in this edition.

The complete `main.c`, together with all supporting modules, can be compiled using the example's single Makefile. This example is much larger than previous teaching programs: radial localization and contour traversal occupy a substantial part of the source. The following listing recalls the essential calculations:

```
/* Selected functions from svgc-examples/021-rk-dynamics/main.c. */

long double complex dx( long double complex z )
{
    return clogl(polynom(z));
}

long double rdx( long double complex z, long double shift )
{
    return creall(clogl(polynom(z))) + shift;
}

long double idx( long double complex z, long double shift )
{
    return cimagl(clogl(polynom(z))) + shift;
}

/*****
    Вычисление знака вещественной динамической характеристики в точке Z:
    */
```

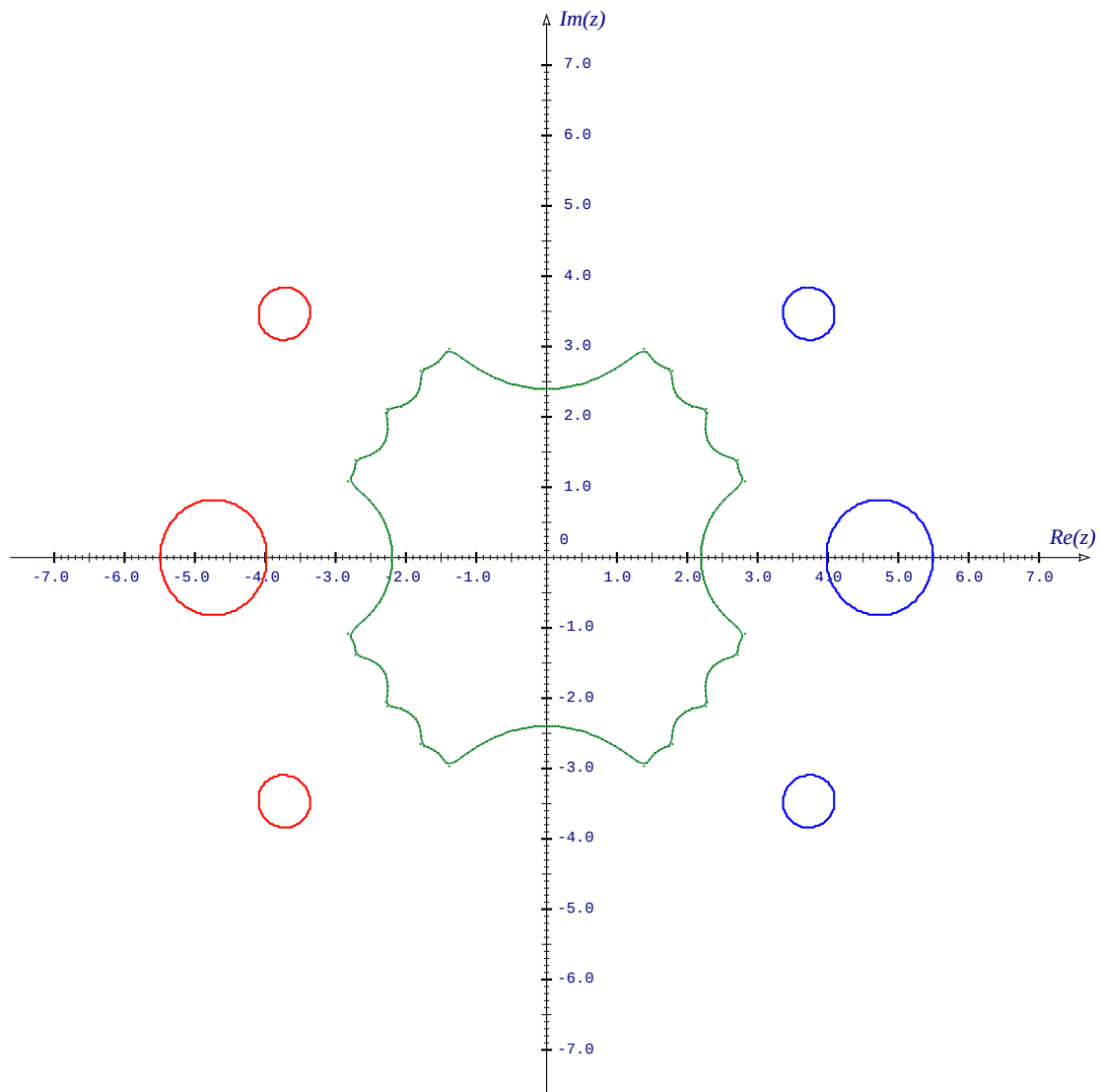


Figure 22.1. Three-stage Gauss–Legendre method: green boundary of the one-percent region of dynamic consistency, red regions localizing zeros, and blue regions localizing poles of the stability function.

```
int rdx_sign( long double complex z, long double shift )
{
    return SIGN(creal1(clog1(polynom(z))) + shift);
}
```

The other functions and their relationships are described above. To reproduce the complete algorithm, compile the source from 021-rk-dynamics, not just the shortened book excerpt.

```
cd svgc-examples/021-rk-dynamics
make
# Open dynamic-consistency-area.svg or view.html.
```

SVG output uses only the public SVGC API. CairoSVG converts the result to a vector PDF for print. Coordinates, paths, and contour colors are preserved; the numerical science is not repeated in TeX. Neither special fonts nor additional analysis tools are needed to view the figure or repeat the experiment.

A substantial scientific illustration begins with a mathematical problem and computed data. Here the regions are determined by the Gauss–Legendre stability function and contour algorithms. SVGC is responsible for rendering the results, not for the properties of the numerical method.

23. Converting SVG to PDF and Including Figures in TeX

23.1 CONVERTING SVG WITH CAIROSVG

An SVG document can be converted to a vector PDF with CairoSVG:

```
cairosvg -o figure.pdf figure.svg
```

Lines, curves, fills, and other graphic objects retain their vector representation. There is no need to rasterize the image or select a PNG resolution for the printed page.

To render text correctly, the required fonts must be available to CairoSVG. If the SVG references external resources or contains complex CSS definitions, the conversion depends on the availability of those resources and on the converter's support for them.

23.2 INCLUDING PDF IN A TEX DOCUMENT

Load `graphicx` in the document preamble:

```
\usepackage{graphicx}
```

Include the resulting PDF with `\includegraphics` :

```
\begin{figure}[htbp]
\centering
\includegraphics[width=0.85\linewidth]{figure.pdf}
\caption{Scientific illustration.}
\end{figure}
```

The width option sets the figure width relative to the text block. The height changes proportionally, and the geometry and line thickness scale together with the image. Its constituent graphics remain vector objects.

24. Reference to the Public SVGC API

24.1 SCOPE OF THIS REFERENCE

This chapter groups the principal SVGC operations as exposed by the public header `svgc.h`. The organization follows the needs of an author of scientific illustrations: scene creation, geometric primitives, paths, styles, text, definitions, and SVG output. The installed header contains the complete structure declarations and helper functions. Here we explain how the groups of interfaces are intended to be used together.

24.2 SHEETS, SCENES, AND RESOURCE OWNERSHIP

The basic sequence begins with a sheet containing dimensions and definitions, and a scene containing the graphical objects.

```
struct __svg_node *__svg_new_sheet(  
    double width, double height,  
    double left_margin, double right_margin,  
    double top_margin, double bottom_margin );  
  
struct __svg_scene *__svg_new_scene(  
    struct __svg_sheet *sheet );  
  
int __svg_scene_add_node(  
    struct __svg_scene *scene,  
    struct __svg_node *node );  
  
void __svg_free_scene( struct __svg_scene *scene );  
void __svg_free_node( void *data );
```

In the example programs, the sheet is created first and then passed to the scene constructor. Once drawing is complete, freeing the scene is normally sufficient: its nodes and definitions are released according to the current SVGC implementation. If an error occurs *before* the sheet is transferred to the scene, the application must free the sheet itself. If adding a node fails, ownership of the unadded node must be checked; ordinary primitives are freed explicitly in the examples.

Do not confuse `__svg_node`, a generic graphical-object description, with `__svg_scene`, an ordered list. Although the structures are declared publicly, it is safer to use the supplied constructors, styling operations, and destructors.

24.3 GEOMETRIC PRIMITIVES

There are separate constructors for basic objects:

```
struct __svg_node *__svg_new_line(  
    double x1, double y1, double x2, double y2 );  
  
struct __svg_node *__svg_new_rect(  

```

```
double x, double y, double width, double height,
double rx, double ry );

struct __svg_node *__svg_new_circle(
    double cx, double cy, double r );

struct __svg_node *__svg_new_ellipse(
    double cx, double cy, double rx, double ry );
```

All arguments use SVG coordinates. For a rectangle, zero values of *rx* and *ry* mean unrounded corners. Parameters *r*, *rx*, and *ry* are radii, not diameters. Numerical validation and checks of physical units are the calling program's responsibility. Mathematical model coordinates must be distinguished from coordinates in the finished SVG.

24.4 VERTEX ARRAYS AND CLOSED OUTLINES

For polygons and polylines, the library offers a point-list object:

```
struct __svg_points *__svg_new_points( void );
int __svg_add_point(
    struct __svg_points *ps, double x, double y );

struct __svg_node *__svg_new_polyline(
    struct __svg_points *ps );
struct __svg_node *__svg_new_polygon(
    struct __svg_points *ps );

void __svg_free_points( struct __svg_points *p );
```

A polyline describes an open series of segments; a polygon closes the outline and may be filled. First populate the point list using `__svg_add_point()`. After transferring the list to a primitive constructor, do not free it independently without verifying ownership: the newly created node uses those resources. Memory-allocation errors must be handled before transfer and checked after each operation.

24.5 PATHS AND GEOMETRIC COMMANDS

A general path stores an ordered sequence of SVG commands. The principal constructors and operations are:

```
struct __svg_cmds *__svg_new_cmds( void );
int __svg_add_moveto_cmd(
    struct __svg_cmds *cmds, int abs, double x, double y );
int __svg_add_lineto_cmd(
    struct __svg_cmds *cmds, int abs, double x, double y );
int __svg_add_hlineto_cmd(
    struct __svg_cmds *cmds, int abs, double x );
int __svg_add_vlineto_cmd(
    struct __svg_cmds *cmds, int abs, double y );
int __svg_add_closepath_cmd(
    struct __svg_cmds *cmds, int abs );
```

```

struct __svg_node *__svg_new_path(
    struct __svg_cmds *cmds );
void __svg_free_cmds( struct __svg_cmds *cmds );

```

Use ABSOLUTE for absolute commands and RELATIVE for relative ones. A relative position is measured from the path's current point, not from the sheet origin. The first command of a new connected outline is normally a `moveto`.

Curves and arcs are supported by the following operations:

```

int __svg_add_curveto_cmd(
    struct __svg_cmds *cmds, int abs,
    double x1, double y1, double x2, double y2,
    double x, double y );

int __svg_add_short_curveto_cmd(
    struct __svg_cmds *cmds, int abs,
    double x2, double y2, double x, double y );

int __svg_add_quadro_curveto_cmd(
    struct __svg_cmds *cmds, int abs,
    double x1, double y1, double x, double y );

int __svg_add_short_quadro_curveto_cmd(
    struct __svg_cmds *cmds, int abs,
    double x, double y );

int __svg_add_arc_cmd(
    struct __svg_cmds *cmds, int abs,
    double rx, double ry, double x_axis_rotation,
    int large_arc_flag, int sweep_flag,
    double x, double y );

```

A cubic Bézier segment uses two control points and an endpoint; a quadratic one uses one control point and an endpoint. The short forms apply SVG smooth-continuation rules by reflecting a previous control point. They are not ordinary abbreviations with arbitrarily omitted required coordinates.

An elliptical arc takes two radii, the rotation angle of its axes, the flags selecting the larger arc and traversal direction, and an endpoint. Neither `large_arc_flag` nor `sweep_flag` is itself an angle: they are Boolean choices among geometrically possible arcs.

24.6 STROKES, FILLS, AND TEXT

Object styling is applied after construction:

```

int __svg_set_stroke(
    struct __svg_node *node, const char *stroke );
int __svg_set_stroke_rgb(
    struct __svg_node *node, int r, int g, int b );
int __svg_set_stroke_width(
    struct __svg_node *node, double width );
int __svg_set_stroke_linecap(
    struct __svg_node *node, const char *linecap );
int __svg_set_stroke_linejoin(

```

```

    struct __svg_node *node, const char *linejoin );
int __svg_set_stroke_dasharray(
    struct __svg_node *node, const char *dasharray );

int __svg_set_fill(
    struct __svg_node *node, const char *fill );
int __svg_set_fill_rgb(
    struct __svg_node *node, int r, int g, int b );

```

stroke and fill values can be CSS colors or references to definitions. The value none disables the corresponding stroke or fill. Cap styles are LINECAP_BUTT, LINECAP_ROUND, and LINECAP_SQUARE; join styles are LINEJOIN_MITER, LINEJOIN_ROUND, and LINEJOIN_BEVEL. Check stroke widths and dash patterns again after changing the figure scale.

Text is handled by:

```

struct __svg_node *__svg_new_text(
    double x, double y, const char *text,
    const char *font_family, int font_size );

int __svg_set_text_text(
    struct __svg_node *node, const char *text );
int __svg_set_text_fill(
    struct __svg_node *node, const char *fill );
int __svg_set_text_fill_rgb(
    struct __svg_node *node, int r, int g, int b );
int __svg_set_text_font_family(
    struct __svg_node *node, const char *family );
int __svg_set_text_font_size(
    struct __svg_node *node, int size );

```

In SVG, y places the text baseline rather than automatically centering the label vertically. Mathematical subscripts in the author's examples use nested SVG `tspan` elements. Validate their output and content escaping as XML, especially for strings assembled from external input.

24.7 IMAGES AND EXTERNAL REFERENCES

An external image can be included with:

```

struct __svg_node *__svg_new_image(
    double x, double y, int width, int height,
    char *href, char *title );

```

The href argument refers to a resource; it is not an in-memory copy of the image file. Successful SVG construction does not guarantee that the resource will remain accessible when the drawing is moved to another directory. External images must therefore be included in reproducibility checks.

24.8 GRADIENTS AND COLOR STOPS

An SVG gradient is a definition referenced by an object's fill. The constructors are:

```

struct __svg_node *__svg_new_l_gradient(
    const char *units, const char *spread,
    double x1, double y1, double x2, double y2 );

struct __svg_node *__svg_new_r_gradient(
    const char *units, const char *spread,
    double cx, double cy, double r, double fx, double fy );

struct __svg_node *__svg_new_stop( double offset );
int __svg_gradient_add_stop(
    void *gradient, struct __svg_stop *stop );

```

A linear gradient follows the direction from its first point to its second. A radial gradient is specified by a center, radius, and focus. `units` selects the coordinate system, while `spread` controls behavior outside the main gradient segment or radial region. For stop colors and opacity, use:

```

int __svg_set_stop_color(
    struct __svg_node *node, const char *stop_color );
int __svg_set_stop_color_rgb(
    struct __svg_node *node, int r, int g, int b );
int __svg_set_stop_opacity(
    struct __svg_node *node, double stop_opacity );

```

Keep the gradient definition separate from its application to a rectangle or other primitive. Multiple objects can use one gradient.

24.9 SHEET DEFINITIONS, CSS, AND JAVASCRIPT

Definitions belong in the sheet's definition list, not in the scene's ordinary list of graphical objects:

```

int __svg_sheet_add_definition(
    void *sheet, struct __svg_node *definition );

int __svg_sheet_add_css_file(
    struct __svg_sheet *sheet, const char *fname );

int __svg_sheet_add_script_file(
    struct __svg_sheet *sheet, const char *fname );

```

The CSS and JavaScript file-loading functions accept only the *contents* of the corresponding language. Source files must not duplicate `defs`, `style`, `script`, or `CDATA` wrapping. SVG C supplies the XML markup and places the content among the definitions.

Both file functions use the same return convention: 1 means success; 0 means a read, encoding, or memory error; and -1 means invalid arguments. The file is read when the function is called, not deferred until SVG output. Callers must check the return value. These functions neither execute JavaScript nor download external CSS fonts.

24.10 IDENTIFIERS, REFERENCES, AND OUTPUT

Use the following operations to obtain node IDs and create SVG references:

```
const char *__svg_get_node_id(  
    struct __svg_node *node );  
char *__svg_create_node_url(  
    struct __svg_node *node );
```

The first returns a pointer to an identifier owned by the node. The second creates a separate string suitable for a style such as `url(##id)`. The examples free this string after passing it to `__svg_set_fill()`.

A scene is finally written using

```
void __svg_paint_scene(  
    struct __svg_scene *scene, int group, char *fname );
```

Notice the `void` return type: the function provides no separate success status to its caller. Automated build procedures should therefore verify that the generated SVG exists and is valid. The examples do this through a separate XML check after execution.

24.11 GLOBAL STYLES AND THEIR LIMITATIONS

The header also provides functions such as `__svg_set_scene_stroke_width()`, `__svg_set_scene_fill_rgb()`, `__svg_set_scene_text_font_family()`, and `__svg_set_scene_default_style()`. They establish common styles used by subsequently created objects. This is convenient for uniform drawings, but the result depends on the order of operations. When examples have different global settings, explicitly establish the initial state instead of relying on the previous figure's settings.

The public header also contains destructors for style objects and additional command constructors. They support applications that manage the lifetimes of intermediate objects themselves. The basic teaching sequence remains simple: create a sheet, create a scene, construct geometry, append objects, write SVG, and free the scene.

Conclusion

In this book we have moved from elementary geometric primitives to programmatic scientific illustrations whose geometry is determined by source data and computational results. Lines, polygons, curves, text, coordinate axes, and diagrams together form a graphical language capable of expressing an algorithm's structure or the results of a numerical experiment.

Every illustration discussed here is produced by a C program. The application algorithm performs calculations, selects coordinates, and places graphical elements, while SVGC records the resulting geometry in SVG. The source drawing remains available for inspection, regeneration, and direct modification through the program.

For publication, SVG can be converted to vector PDF and included in a TeX document. Computation, graphical representation, and the printed page thus become stages of one coherent workflow: from a numerical problem to a finished scientific illustration.

A. SVG Path Commands and Their SVGC API Equivalents

A.1 PATH SYNTAX RULES

The `d` attribute of a path element contains a sequence of commands and numeric arguments. Uppercase command letters select absolute coordinates; lowercase letters select relative coordinates. Relative offsets are measured from the current path point, not from the upper-left corner of the sheet. `M/m` starts a subpath, and `Z/z` closes the current subpath to its starting point.

The public C API uses `ABSOLUTE` and `RELATIVE` to select the mode. The application creates `__svg_cmds`, appends commands, and transfers the list to `__svg_new_path()`. Do not free a transferred list independently of its owning node.

A.2 MAIN COMMANDS AND ARGUMENT COUNTS

Command	Arguments	Meaning
<code>M/m</code>	2	Move the current point to (x, y) ; start a subpath.
<code>L/l</code>	2	Straight line from the current point to (x, y) .
<code>H/h</code>	1	Horizontal segment ending at the specified x .
<code>V/v</code>	1	Vertical segment ending at the specified y .
<code>C/c</code>	6	Cubic Bézier curve: two control points and endpoint.
<code>S/s</code>	4	Cubic continuation with reflected first control point when the preceding command permits it.
<code>Q/q</code>	4	Quadratic Bézier curve: control point and endpoint.
<code>T/t</code>	2	Quadratic continuation with reflected control point when the preceding command permits it.
<code>A/a</code>	7	Elliptical arc: radii, rotation, two flags, and endpoint.
<code>Z/z</code>	0	Close the current subpath.

After `M`, additional coordinate pairs without a new command letter are treated as `L`; after `m`, as `l`. Many commands allow repeated argument groups. Searching for command letters in the `d` attribute is therefore not a substitute for a full SVG path parser.

A.3 BÉZIER CURVE EQUATIONS

A cubic curve with points P_0, P_1, P_2, P_3 uses $0 \leq t \leq 1$:

$$B_3(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3.$$

A quadratic curve with P_0, P_1, P_2 uses

$$B_2(t) = (1-t)^2 P_0 + 2(1-t) t P_1 + t^2 P_2.$$

The equations apply independently to both coordinates. At the start and end of a cubic curve, the tangents are proportional to $P_1 - P_0$ and $P_3 - P_2$.

S reflects the previous second control point about the current point only after C or S; otherwise, its new first control point coincides with the current point. Similarly, T performs reflection after Q or T; otherwise, its control point equals the current point. These rules are essential geometry, not merely shorthand syntax.

A.4 ARGUMENTS OF AN ELLIPTICAL ARC

The A command takes arguments in this order:

```
A rx ry x-axis-rotation large-arc-flag sweep-flag x y
```

The radii r_x, r_y define the ellipse's dimensions, while x-axis-rotation rotates it in degrees. The large-arc-flag selects the larger or smaller arc, and sweep-flag selects the traversal direction in SVG coordinates. Both flags are 0 or 1; they are not angles in radians. A zero radius makes the arc degenerate into a segment, while insufficient nonzero radii are corrected under SVG rules. Coincident endpoints do not automatically describe a complete ellipse.

A.5 C CONSTRUCTION FUNCTIONS

The most frequently used SVG commands correspond to public API calls as follows:

```
M/m : __svg_add_moveto_cmd()
L/l : __svg_add_lineto_cmd()
H/h : __svg_add_hlineto_cmd()
V/v : __svg_add_vlineto_cmd()
C/c : __svg_add_curveto_cmd()
S/s : __svg_add_short_curveto_cmd()
Q/q : __svg_add_quadro_curveto_cmd()
T/t : __svg_add_short_quadro_curveto_cmd()
A/a : __svg_add_arc_cmd()
Z/z : __svg_add_closepath_cmd()
```

Each function appends one command to a list. Individual moveto, lineto, and curve commands do not create separate SVG elements. The shared path is constructed after the application has defined the required geometry.

```
struct __svg_cmds *cmds = __svg_new_cmds();
if( cmds )
{
    __svg_add_moveto_cmd( cmds, ABSOLUTE, 20, 20 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, 140, 20 );
    __svg_add_lineto_cmd( cmds, ABSOLUTE, 140, 90 );
    __svg_add_closepath_cmd( cmds, ABSOLUTE );
    /* Creating a path and adding it to the scene comes next. */
}
```

Complete command-error and ownership handling is demonstrated in 008-path, 009-bezier, and 010-arcs. The fragment above illustrates only the operation sequence; it is not a complete example.

The geometric meaning of each segment follows from its command and arguments. Two visually similar paths need not have equivalent source data or construction rules.

B. Coordinate Transformation and Scaling Formulas

B.1 THE SCREEN RECTANGLE AND DATA DOMAIN

Suppose the mathematical data lie in $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, and the SVG drawing area begins at (L, T) with width W and height H . Assume $x_{\max} > x_{\min}$, $y_{\max} > y_{\min}$, $W > 0$, and $H > 0$.

The mathematical vertical coordinate increases upward, while normal SVG screen coordinate Y increases downward. Thus the mapping $(x, y) \mapsto (X, Y)$ is

$$X = L + \frac{W(x - x_{\min})}{x_{\max} - x_{\min}}, \quad Y = T + \frac{H(y_{\max} - y)}{y_{\max} - y_{\min}}.$$

Define

$$s_x = \frac{W}{x_{\max} - x_{\min}}, \quad s_y = \frac{H}{y_{\max} - y_{\min}}.$$

Then the coordinates can be calculated as

$$X = L + s_x(x - x_{\min}), \quad Y = T + s_y(y_{\max} - y).$$

This form avoids repeated division by range widths when processing large arrays of points.

B.2 THE INVERSE TRANSFORMATION

Mathematical coordinates can be recovered from screen coordinates:

$$x = x_{\min} + \frac{X - L}{s_x}, \quad y = y_{\max} - \frac{Y - T}{s_y}.$$

This is useful when viewing SVG interactively: a pointer position can be interpreted as an approximate point in the mathematical model. Additional scaling by the viewer or CSS must be accounted for if the pointer coordinates refer to the application window rather than to SVG's internal coordinate system.

B.3 VECTORS, LENGTHS, AND UNEQUAL SCALES

For an increment $\Delta \mathbf{r} = (\Delta x, \Delta y)$, the screen vector is

$$\Delta X = s_x \Delta x, \quad \Delta Y = -s_y \Delta y.$$

In general $s_x \neq s_y$, so equal mathematical lengths can have different apparent sizes. Anisotropic scaling does not necessarily preserve a right angle between arbitrary vectors. An application must account for this when drawing circles, arrows of prescribed direction, or curve tangents and normals.

For equal scaling, choose $s_x = s_y = s$. If the entire mathematical domain must fit within the screen rectangle without clipping, use

$$s = \min\left(\frac{W}{x_{\max} - x_{\min}}, \frac{H}{y_{\max} - y_{\min}}\right).$$

The remaining space must be distributed explicitly, for example in equal margins on both sides of each direction. This changes the visual margins, not the original data coordinates.

B.4 SELECTING BOUNDS AND PROTECTIVE MARGINS

If bounds are derived from arrays of x_i and y_i , first compute

$$\Delta x = x_{\max} - x_{\min}, \quad \Delta y = y_{\max} - y_{\min}.$$

The range may then be expanded by a fraction δ :

$$x'_{\min} = x_{\min} - \delta \Delta x, \quad x'_{\max} = x_{\max} + \delta \Delta x.$$

Expand the vertical range in the same way. If a minimum equals its maximum, first prescribe a positive width manually; inserting $\Delta x = 0$ into a division is invalid. NaN and infinity must also be excluded from range boundaries.

B.5 TICK LABELS AND THE 1-2-5 FAMILY

Chapter 17 describes a practical numerical-grid spacing policy. For a positive range D and desired number of intervals $N > 0$, calculate

$$d_0 = \frac{D}{N}, \quad k = \lfloor \log_{10} d_0 \rfloor.$$

Round the normalized step $d_0/10^k$ upward to the nearest member of $\{1, 2, 5, 10\}$ and multiply it by 10^k .

Instead of accumulating error by repeatedly adding d , use integer indices:

$$i_{\min} = \left\lceil \frac{x_{\min}}{d} \right\rceil, \quad i_{\max} = \left\lfloor \frac{x_{\max}}{d} \right\rfloor, \quad x_i = id.$$

Compute the second coordinate similarly. Each label is then positioned by the same affine transformation as its tick.

B.6 AN INDEPENDENT VERIFICATION EXAMPLE

Let the mathematical domain be $x \in [-2, 8]$ and $y \in [-1, 3]$, with display rectangle $L = 40$, $T = 30$, $W = 500$, and $H = 200$. Then $s_x = s_y = 50$, and the origin maps to

$$X(0) = 40 + 50 \cdot 2 = 140, \quad Y(0) = 30 + 50 \cdot 3 = 180.$$

The point $(4, 2)$ maps to $(340, 80)$. Substituting these values into the inverse formulas recovers exactly $(4, 2)$ in ideal real arithmetic.

Keep model data separate from sheet coordinates. The application owns the coordinate transformation; SVGC receives the resulting coordinates of graphical objects.

C. Glossary of Technical Terms

C.1 HOW TO USE THIS GLOSSARY

The terms below are defined as they are used in this book. Brief English explanations distinguish the SVG markup language, the C library API, mathematical algorithms, and tools for preparing printed material. The glossary does not replace formal SVG specifications or extend the SVGC API.

English term	Meaning in this book
Affine transform	Coordinate transformation consisting of a linear part and a translation.
Algorithm	Defined sequence of computations and geometric construction steps.
Antialiasing	Smoothing of rasterized edges during display.
Arc	Portion of a circle or ellipse.
Axis	Coordinate axis of a mathematical plot.
Baseline	Reference line used to position text glyphs.
Bounding box	Smallest enclosing rectangle of a geometric object.
Browser	Viewer used to display and inspect SVG.
CDATA section	XML section whose contents are not parsed as ordinary markup.
Clip / clipping path	Restriction of visible graphics to a prescribed region.
Closed contour	Outline whose final point connects to its start.
Control point	Point influencing the shape of a Bézier curve.
Coordinate mapping	Transformation from data coordinates to display coordinates.
Cubic Bézier curve	Bézier curve defined by two control points.
Dash array	Pattern of alternating stroke and gap lengths.
Data range	Numerical interval spanned by the data.
Definition	SVG definition, commonly stored in defs.
Display scale	Scale used to render mathematical coordinates.
Endpoint	Terminal point of a segment or curve.
Fill	Color or pattern applied to an object's interior.
Font family	Related set of font faces and styles.
Font metrics	Measurements controlling the dimensions and spacing of text.
Glyph	Visible graphical shape of a character in a font.
Gradient	Continuous color variation across a filled region.
Interpolation	Reconstruction of a function from known nodes.
Kerning	Adjustment of spacing between particular character pairs.
Line cap	Segment-end style: butt, round, or square.
Line join	Segment-connection style: miter, round, or bevel.
Marching Squares	Contour-extraction algorithm operating on rectangular grid cells.
Natural cubic spline	Cubic spline with zero second derivatives at both endpoints.
Path	Compound SVG outline containing straight and curved segments.
Phase portrait	Representation of a trajectory in state space.
Polyline	Connected sequence of line segments.
Polygon	Closed outline defining a polygonal region.
Quadratic Bézier curve	Bézier curve defined by one control point.

Radial gradient	Color gradient varying about a center or focus.
Rasterization	Conversion of vector geometry into pixels.
Regression test	Check against a previously accepted result.
Reproducibility	Ability to recreate results from original data and tools.
Root / zero	Point at which a function vanishes.
Scene	Ordered collection of graphical objects.
Stroke	Visible outline, including its color and thickness.
Style sheet	Collection of CSS rules controlling appearance.
SVG	Scalable Vector Graphics, an XML-based vector graphics format.
Symbolic label	Mathematical annotation containing indices or special symbols.
CairoSVG	Converter from SVG to vector PDF.
Time series	Sequence of observed or calculated values indexed by time.
Tridiagonal solver	Algorithm for tridiagonal systems, such as the Thomas algorithm.
Viewport	Visible area of an SVG document.
Visual regression	Comparison of images before and after a change.
XML	Extensible Markup Language.

C.2 TERMS THAT MUST NOT BE CONFUSED

A **polyline** is a sequence of segments and need not be closed. A **polygon** defines a closed region that may be filled. A **path** is more general: its segments can be straight or curved.

Font metrics (*font metrics*) cannot be replaced with the number of characters in a string. Accurate text alignment requires accounting for glyphs, spaces, and kerning. SVGC deliberately leaves such width calculations to the application or external tools.

An **SVG source** and its **printed rendering** (*PDF rendering*) are related but distinct objects of verification. Establishing their equivalence requires checking the intended content, not merely XML validity or a successful XeLaTeX compilation.

D. Index of C Programs, Chapters, and Figures

D.1 NUMBERING CONVENTIONS

The three-digit prefixes in directory names such as `001-first-sheet` identify the sequence of C examples. Chapter numbering starts at one and does not coincide with the example number: Chapter 1 covers installation and has no separate figure of its own.

Each example generates an SVG and includes `view.html`. The PDF book contains a corresponding vector-PDF rendition. Figure numbers follow chapter numbers and the order of figures within each chapter. In this edition, each chapter containing a demonstration C program has one principal illustration.

D.2 PROGRAMS AND ILLUSTRATIONS

Chapter	Example directory	Subject and figure
2	<code>001-first-sheet</code>	Sheet and first scene, Fig. 2.1.
3	<code>002-primitives</code>	Basic geometric primitives, Fig. 3.1.
4	<code>003-data-coordinates</code>	Data-driven coordinates, Fig. 4.1.
5	<code>004-polylines</code>	Polylines and polygons, Fig. 5.1.
6	<code>005-arrows</code>	Arrowhead geometries, Fig. 6.1.
7	<code>006-flowchart</code>	Algorithmic flowchart, Fig. 7.1.
8	<code>007-styles</code>	Strokes, fills, and dashes, Fig. 8.1.
9	<code>008-path</code>	Compound path commands, Fig. 9.1.
10	<code>009-bezier</code>	Bézier curves, Fig. 10.1.
11	<code>010-arcs</code>	Arcs, chords, and sectors, Fig. 11.1.
12	<code>011-text</code>	Text and baselines, Fig. 12.1.
13	<code>012-gradients</code>	Gradients, Fig. 13.1.
14	<code>013-control-system</code>	Gyroscopic structural diagram, Fig. 14.1.
15	<code>014-rk4-scheme</code>	RK4 step structure, Fig. 15.1.
16	<code>015-oscillator</code>	Original mechanical oscillator, Fig. 16.1.
17	<code>016-axes-grid</code>	Axes, ticks, and grid, Fig. 17.1.
18	<code>017-discrete</code>	Observations and model, Fig. 18.1.
19	<code>018-cubic-spline</code>	Cubic interpolation, Fig. 19.1.
20	<code>019-phase-portrait</code>	Phase portrait and time series, Fig. 20.1.
21	<code>020-contours</code>	Level contours, Fig. 21.1.
22	<code>021-rk-dynamics</code>	Gauss–Legendre dynamic properties, Fig. 22.1.

D.3 FINDING AN EXAMPLE BY TASK

For basic geometry, start with `002-primitives`, `004-polylines`, and `008-path`. For mathematical notation, examine `011-text` and the author's diagrams `013-control-system` and `015-oscillator`.

The numerical programs can be studied in increasing order of complexity: `016-axes-grid`, `017-discrete`, `018-cubic-spline`, `019-phase-portrait`, `020-contours`, and `021-rk-dynamics`. The final example analyzes Gauss–Legendre stability with Marching Squares contours of a logarithmic characteristic.