

Создание научной графики на языке C

Программирование SVG на языке C

SVG Programming

*Алгоритм порождает образ. Программа описывает геометрию, а библиотека SVGС формирует
человекочитаемый SVG. Учебные примеры написаны на языке C и не требуют графического
редактора.*

Предисловие

Библиотека SVGC появилась как рабочий инструмент для построения научных и инженерных изображений на языке C. Она позволяет создавать человекочитаемые SVG-документы непосредственно во время работы программ, моделирующих поведение реальных объектов. С её помощью можно также разрабатывать специальные программы для создания изображений, геометрия которых требует предварительных сложных вычислений. В обоих случаях программа вычисляет геометрию, а затем формирует воспроизводимый и человекочитаемый SVG-документ. Именно таким способом подготавливались рисунки для сайта <https://dynamics.su/>.

Основная идея книги проста: *алгоритм порождает образ*. Численные данные, геометрические правила и преобразования координат содержатся в прикладном алгоритме, а SVGC остаётся библиотекой описания графических объектов и записи документа SVG. Поэтому не требуется подгонять данные под координатную систему какого-либо графического редактора.

Книга построена по принципу от простого к сложному. Изучение библиотеки ведётся на практических примерах: сначала рассматриваются геометрические примитивы, затем координатные оси, определяемые данными, ломанные, многоугольники, стрелки и блок-схемы. Далее изучаются стили линий, команды SVG path, кривые Безье, эллиптические дуги и составные геометрические формы. Рассматриваются вывод текста, градиенты, инженерные схемы, сплайны, а также исследование динамических характеристик численных методов интегрирования обыкновенных дифференциальных уравнений. Заключительная часть посвящена подготовке векторных PDF для печати и публичному API SVGC.

Книга может быть полезна научным работникам, инженерам и студентам технических специальностей вузов.

На первый взгляд может показаться, что для создания изображений с помощью библиотеки SVGC необходимы специальные знания в области программирования. Однако современные системы искусственного интеллекта заметно упрощают эту задачу. Исходный код библиотеки вместе с текстом книги можно предоставить нейронной сети, а задачу построения рисунка сформулировать обычными словами. Во многих случаях это позволяет получить необходимую иллюстрацию за считанные минуты. При этом геометрия рисунка и корректность созданной программы, разумеется, требуют проверки.

Книга сопровождается многочисленными примерами на языке C и простыми файлами Makefile. Для просмотра изображений в браузере каждый пример содержит страницу `view.html`.

Для подготовки иллюстраций книги C-программы создают SVG-файлы, а CairoSVG преобразует их непосредственно в векторные PDF, включаемые в страницы посредством `\includegraphics`. Это позволяет сохранить геометрию рисунков и, при наличии шрифтов, оформление текстовых подписей и математических обозначений.

Оглавление

Оглавление	ii
1 Сборка и установка SVGC	1
1.1 Назначение библиотеки	1
1.2 Структура исходного пакета	1
1.3 Подготовка и конфигурирование	1
1.4 Сборка и проверка	2
1.5 Установка	2
1.6 Сборка собственной программы	2
2 SVG-документ, лист и сцена	4
2.1 SVG как результат выполнения программы	4
2.2 Экранные координаты	4
2.3 Лист и сцена	5
2.4 Первая программа на C	5
2.5 Сборка примера и просмотр результата	7
2.6 О чём помнить при чтении SVG	7
3 Простые геометрические примитивы	8
3.1 От линии к геометрической композиции	8
3.2 Прямоугольник и скруглённые углы	8
3.3 Окружность и эллипс	8
3.4 Линии, заливки и обводки	8
3.5 Пример 2	9
3.6 Дуги, хорды и стрелки: направление развития	10
4 Координатные оси, определяемые данными	11
4.1 Почему оси не должны совпадать с рамкой	11
4.2 Границы входного набора	11
4.3 Аффинное преобразование координат	11
4.4 Пример с дискретными значениями	11
5 Ломаные, многоугольники и дискретная геометрия	16
5.1 От отдельных отрезков к массиву вершин	16
5.2 Что означают порядок и замыкание	16
5.3 Публичный интерфейс SVGC	16
5.4 Изображение дискретных измерений	17
5.5 Пример 4	17
5.6 Граница между визуализацией и вычислением	19
6 Стрелки и их окончания	20
6.1 Почему стрелка — составной объект	20
6.2 Единичный направляющий вектор	20
6.3 Начало и конец имеют противоположные направления	20
6.4 Пример 5	21
6.5 Масштаб, стили и порядок наложения	25
7 Программное построение блок-схем	26
7.1 Граф управляющих переходов	26
7.2 Пример: вычислительный цикл	26

7.3	Пример 6	26
7.4	Подписи в блоках: почему центрирование делается вручную	29
7.5	Текст в SVG и безопасность XML	30
7.6	Автоматическая компоновка — отдельная задача	30
8	Оформление объектов: контур, заливка и штрих	31
8.1	Геометрия и стиль — разные составляющие рисунка	31
8.2	Цвет обводки и цвет заполнения	31
8.3	Окончания штриха и соединения сегментов	31
8.4	Пунктир и ритм изображения	31
8.5	Пример 7	32
9	Составной контур SVG: команды path	35
9.1	Путь как последовательность геометрических действий	35
9.2	Перемещение, линия и замыкание	35
9.3	Абсолютные и относительные координаты	35
9.4	Контур, заливка и порядок обхода	35
9.5	Пример 8	36
10	Квадратичные и кубические кривые Безье	39
10.1	Переход от ломаной к плавной кривой	39
10.2	Квадратичная кривая	39
10.3	Кубическая кривая	39
10.4	Геометрия управляющих точек	39
10.5	Сглаживание дискретных данных не является рисованием наугад	40
10.6	Пример 9	40
11	Эллиптические дуги, хорды и секторы	44
11.1	Эллиптическая дуга как самостоятельный элемент пути	44
11.2	Флаги выбора дуги	44
11.3	Дуга, хорда и сектор	44
11.4	Пример 10	44
12	Надписи, шрифты и текстовые координаты	48
12.1	Текст является графическим объектом	48
12.2	Гарнитура, размер и цвет	48
12.3	Почему ширину строки нельзя узнать из числа символов	48
12.4	Пример 11	48
12.5	Экранирование текста в XML	50
12.6	Внешние шрифты и определения CSS внутри SVG	50
12.7	Один общий каталог шрифтов для примеров	51
12.8	Нижние и верхние индексы посредством <code>tspan</code>	51
12.9	Размеры и гарнитуры шрифтов при преобразовании SVG в TeX	52
13	Линейные и радиальные градиенты	53
13.1	Определения и ссылки на них	53
13.2	Линейное изменение цвета	53
13.3	Радиальный градиент	53
13.4	Пример 12	54
13.5	Проверка печатного преобразования	55
14	Структурная схема гироскопической системы	57
14.1	Графическое описание математической модели	57
14.2	Дифференциальная модель системы угловой стабилизации	57
14.3	Блоки, связи и передаточные множители	57

14.4	Сумматоры, ответвления и порядок наложения	57
14.5	Математические подписи и вложенные tspan	58
14.6	Подключение CSS средствами SVGC	58
14.7	Пример 13	58
15	Схема классического метода Рунге–Кутты	69
15.1	От формулы к структуре вычисления	69
15.2	Геометрия рисунка	69
15.3	Пример 14	70
16	Механический осциллятор: геометрия сил и связей	74
16.1	Авторский рисунок как программа	74
16.2	Система координат и геометрическая параметризация	74
16.3	Локальный базис и разложение вертикальной силы	74
16.4	Направления стрелок и правила поворота	75
16.5	Дуги и частичные окружности	75
16.6	Подписи с математическими индексами	75
16.7	Пример 15	76
16.8	Что дальше: координатные оси и результаты расчёта	84
17	Координатные оси, масштаб и подписи делений	85
17.1	Две независимые системы координат	85
17.2	Диапазон данных и защитные поля	85
17.3	Откуда берётся шаг сетки	85
17.4	Порядок наложения объектов	86
17.5	Пример 16	87
17.6	Ограничения простой автоматической разметки	91
18	Дискретные наблюдения и сравнение с моделью	92
18.1	Почему ряд наблюдений нельзя путать с непрерывной функцией	92
18.2	Две формы изображения одних и тех же данных	92
18.3	Величина отклонений от модели	92
18.4	Пример 17	93
18.5	Достоверность визуального сравнения	96
19	Натуральный кубический интерполяционный сплайн	97
19.1	Зачем нужны условия гладкости	97
19.2	Выбор граничных условий	97
19.3	Трёхдиагональная система	97
19.4	Формула вычисления сплайна	98
19.5	Программная проверка	98
19.6	Пример 18	99
19.7	Ограничения интерполяции	104
20	Фазовые траектории и временные ряды	105
20.1	Одна модель — два вида научного графика	105
20.2	Численная схема Рунге–Кутты	105
20.3	Независимый аналитический контроль	105
20.4	От временного ряда к фазовой плоскости	106
20.5	Пример 19	107
20.6	Что именно изображает линия	111
21	Контурные линии функций двух переменных	112
21.1	От скалярного поля к множеству уровня	112
21.2	Прямоугольная вычислительная сетка	112

21.3	Алгоритм Marching Squares	112
21.4	Неоднозначные ячейки	113
21.5	Изолинии как результат вычисления	113
21.6	Пример 20	113
21.7	Сглаживание и геометрическая точность	117
22	Динамические характеристики метода Гаусса–Лежандра	119
22.1	От численного метода к функции устойчивости	119
22.2	Геометрия нулей и полюсов	119
22.3	Логарифмическая динамическая характеристика	120
22.4	Однопроцентная область динамической согласованности	120
22.5	Области локализации нулей и полюсов	121
22.6	Marching Squares для контуров функций устойчивости	121
22.7	Рисунок 22.1 и полный C-пример	121
23	Преобразование SVG в PDF и включение рисунка в TeX	124
23.1	Преобразование SVG средствами CairoSVG	124
23.2	Включение PDF в документ TeX	124
24	Справочник публичных интерфейсов SVG C	125
24.1	Область применения справочника	125
24.2	Лист, сцена и владение ресурсами	125
24.3	Геометрические примитивы	125
24.4	Массивы вершин и замкнутые контуры	126
24.5	Путь и команды геометрии	126
24.6	Контур, заливка и текст	127
24.7	Изображения и внешние ссылки	128
24.8	Градиенты и цветовые остановки	129
24.9	Определения листа, CSS и JavaScript	129
24.10	Идентификаторы, ссылки и вывод	130
24.11	Глобальные стили и границы их использования	130
	Заключение	131
A	Команды SVG path и соответствие API SVG C	132
A.1	Правила записи пути	132
A.2	Основные команды и число параметров	132
A.3	Формулы кривых Безье	132
A.4	Аргументы эллиптической дуги	133
A.5	Функции построения в C	133
B	Формулы координатных преобразований и масштаба	135
B.1	Экранный прямоугольник и область данных	135
B.2	Обратное преобразование	135
B.3	Векторы, длины и неодинаковые масштабы	135
B.4	Подбор диапазона и защитного поля	136
B.5	Подписи делений и семейство 1–2–5	136
B.6	Пример для независимой проверки	136
C	Русско-английский словарь терминов	137
C.1	Как пользоваться словарём	137
C.2	Термины, которые нельзя смешивать	138
D	Указатель C-программ, глав и рисунков	139
D.1	Как устроена нумерация	139

D.2	Соответствие исходных программ и иллюстраций	139
D.3	Как найти программу по задаче	139

1. Сборка и установка SVGC

1.1. НАЗНАЧЕНИЕ БИБЛИОТЕКИ

SVG C (SVG C Library) предоставляет программе на языке C средства описания графических объектов и последовательной записи документа **SVG** (*Scalable Vector Graphics, масштабируемая векторная графика*). Результат работы приложения — обычный текстовый XML-файл. Его можно прочесть без специализированного редактора, сравнить средствами контроля версий и открыть в браузере. Форма изображения задаётся непосредственно исходным кодом, что особенно важно для воспроизводимости численного опыта.

Главное назначение библиотеки — запись геометрии и стилей. Вычисление границ численных данных, построение координатных преобразований, интерполяция или поиск нулей функции — задачи отдельной C-программы.

1.2. СТРУКТУРА ИСХОДНОГО ПАКЕТА

Дерево исходного кода пакета SVG C содержит публичный заголовочный файл `src/svgc.h`, реализацию библиотеки `src/svgc.c` и вспомогательные файлы двусвязного списка `src/svgc-dlist.h` и `src/svgc-dlist.c`. В корне пакета находятся `configure.ac`, `acsite.m4`, `svgc.m4`, `Makefile.am`, а также файлы для сборки `svgc-config` и `svgc.pc`. Каталог `examples/` содержит самостоятельные примеры применения библиотеки. Сборочная система пакета использует GNU Autoconf, Automake и Libtool.

Для сборки готового исходного пакета достаточно компилятора C, `make`, командной оболочки и обычных инструментов сборки. Повторно запускать Autoconf и Automake не требуется: необходимые сценарии уже включены в исходный архив.

1.3. ПОДГОТОВКА И КОНФИГУРИРОВАНИЕ

Исходный код рекомендуется собирать в отдельном каталоге. Например, после получения исходного архива библиотеки выполняются команды:

```
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
```

Здесь каталог `build-svgc/` располагается рядом с каталогом исходного пакета SVG C. При таком способе сборки исходные файлы не смешиваются с объектными файлами и прочими результатами компиляции. Параметр `--prefix=/usr` задаёт префикс установки, а `--enable-static=no` отключает сборку статической библиотеки. Каталог `/usr/lib64` выбран для систем, использующих это соглашение; в других системах может потребоваться `/usr/lib` или другое значение `--libdir`.

1.4. СБОРКА И ПРОВЕРКА

После успешного конфигурирования библиотека собирается командой:

```
make
```

При желании запускаются предусмотренные исходным пакетом проверки:

```
make check
```

Эти команды выполняются внутри каталога `build-svgc/`. Для дополнительной проверки качества исходного кода разработчик может задать компилятору более строгие параметры, например `-O2 -Wall -Wextra -Werror -std=gnu23`, передав их в `CFLAGS` при конфигурировании. Начинающему пользователю это необязательно: для обычной сборки достаточно предыдущих команд.

1.5. УСТАНОВКА

Для установки собранной библиотеки выполняется:

```
make install
```

Если запись в выбранные системные каталоги требует прав администратора, используется `sudo make install` или предусмотренный системой способ выполнения команды от имени администратора. Для подготовки собственного системного пакета установку можно направить во временное дерево с помощью `DESTDIR`, не изменяя работающую систему; это отдельная задача для разработчиков пакетов, а не обязательный этап обычной установки.

1.6. СБОРКА СОБСТВЕННОЙ ПРОГРАММЫ

После установки SVGC программа на языке C может подключать заголовочный файл `svgc.h` и компоноваться с библиотекой. Для простой программы `main.c` достаточно следующего Makefile (команды в правилах начинаются символом табуляции):

```
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) $(LDFLAGS) -o main $(OBJECTS) $(LIBS)

clean:
    rm -f *.o main

```

Команда `make` создаёт исполняемый файл `main`, а `make clean` удаляет результаты сборки. Скрипт `svgc-config --cflags` возвращает параметры компиляции, `--ldflags` — путь поиска библиотеки при компоновке, `--libs` — её имя для компоновщика. Дополнительный ключ `-lm` подключает математическую библиотеку. Таким образом, пути к заголовкам и библиотеке не нужно жёстко записывать в `Makefile`.

Для разработчиков исходных пакетов с Autotools. Пакет SVG C Library устанавливает `svgc.m4` в каталог `/usr/share/aclocal/` при выбранном префиксе `/usr`. Этот файл определяет макрос `AC_CHECK_SVGC`, позволяющий проверить версию библиотеки, наличие заголовков, возможность компоновки и, при включённой проверке, выполнение тестовой программы. В собственном `configure.ac` после настройки компилятора можно записать:

```

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

```

Порядок аргументов `AC_CHECK_SVGC` определён в самом `svgc.m4`: минимальная версия, значения по умолчанию для `--with-svgc` и `--with-svgc-test`, признак расширения флагов сборки, действия при успешной и неуспешной проверках. Последние два аргумента нельзя помещать на третью и четвёртую позиции: там находятся параметры настройки проверки. При успешном обнаружении библиотеки макрос также предоставляет переменные `SVG_C_FLAGS`, `SVG_C_LDFLAGS` и `SVG_C_LIBS`, которые можно подставлять в `Makefile.am`. Файл `svgc.m4` обрабатывается утилитой `aclocal` при подготовке сценария `configure`; пользователям готового исходного пакета запускать эти инструменты для сборки SVG C Library не требуется.

2. SVG-документ, лист и сцена

2.1. SVG КАК РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

SVG — это XML-документ, описывающий геометрические объекты, их координаты и оформление. Даже большой рисунок остаётся текстом: отдельные линии, прямоугольники, окружности и подписи могут быть прочитаны и найдены независимо от способа просмотра. Человекочитаемая форма SVG особенно удобна при отладке алгоритма построения изображения.

Здесь необходимо разделять три стадии:

1. прикладная программа вычисляет точки, размеры и отношения объектов;
2. SVGC формирует из этих данных элементы SVG;
3. средство просмотра отображает SVG на экране или печатной странице.

2.2. ЭКРАННЫЕ КООРДИНАТЫ

Для документа SVG без дополнительных преобразований начало координат расположено в левом верхнем углу. Координата x возрастает вправо, координата y — вниз. Именно такой порядок удобен при построении изображений на экране. Координаты не обязаны соответствовать ни сантиметрам, ни математическим переменным: это координаты выбранной сцены и её изображения.

В первом рисунке ширина листа равна 640 условным единицам, а высота — 400. Программа самостоятельно рисует в точке $(80, 80)$ два отрезка, служащие видимыми осями. При этом настоящее начало координат SVG по-прежнему находится в точке $(0, 0)$.

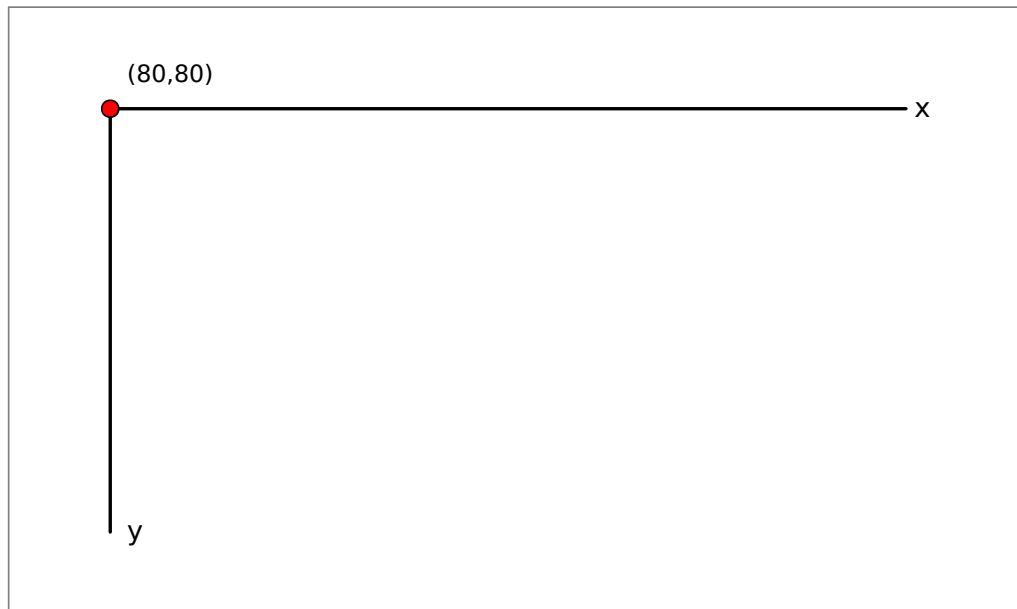


Рис. 2.1. Первый лист SVGС. Экранная система координат и выбранная точка внутри сцены.

2.3. ЛИСТ И СЦЕНА

SVGC предоставляет функцию `__svg_new_sheet()` для создания листа, `__svg_new_scene()` для сцены и `__svg_scene_add_node()` для добавления графических узлов. Лист определяет геометрию полотна, сцена — порядок объектов, а узел представляет элемент, например линию или окружность. Программа может создать несколько узлов, настроить их свойства, а затем передать сцену функции `__svg_paint_scene()`.

Важен и порядок добавления: элементы SVG наносятся последовательно. Поэтому белый прямоугольник фона в примере помещается в сцену первым, а рамка, оси и подписи располагаются поверх него.

2.4. ПЕРВАЯ ПРОГРАММА НА C

Полный текст первого примера находится в каталоге `svgc-examples/001-first-sheet/`. Он специально написан без вспомогательного графического фреймворка, чтобы читатель видел каждое обращение к SVGC. Проверяется создание сцены и каждого узла; после записи SVG вся сцена освобождается.

```

/*****
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. СБОРКА ПРИМЕРА И ПРОСМОТР РЕЗУЛЬТАТА

Примеры программ, рассматриваемых в книге, поставляются в отдельном архиве `svgc-examples.tar.xz`. Каждый пример находится в собственном каталоге и содержит исходный текст программы на языке C, файл `Makefile`, готовый SVG-документ и HTML-страницу для его просмотра. После распаковки архива первый пример находится в каталоге `svgc-examples/001-first-sheet/`.

Сборка примера осуществляется с помощью утилиты GNU Make:

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

`Makefile` описывает сценарий сборки программы и её компоновки с установленной библиотекой SVGC. После успешной сборки запускается исполняемая программа `first-sheet`, которая создаёт SVG-документ `first-sheet.svg`.

Полученный файл удобно открыть в браузере с помощью HTML-документа `view.html`. Кроме того, SVG является текстовым XML-документом, поэтому его содержимое можно просмотреть и изучить в обычном текстовом редакторе.

Создание объекта на рисунке и выбор координат — разные операции. Графическая библиотека не должна самостоятельно решать, где находится математический нуль исходной задачи.

2.6. О ЧЁМ ПОМНИТЬ ПРИ ЧТЕНИИ SVG

Не следует привязывать алгоритм к автоматически созданным идентификаторам элементов. Название узла в XML может меняться после изменения программы, но геометрический смысл определяется численными координатами и стилевыми атрибутами. Полезно сравнивать не только изображение, но и записанные свойства — например `stroke`, `fill`, `x1`, `y1`, `cx` и `cy`.

3. Простые геометрические примитивы

3.1. От линии к ГЕОМЕТРИЧЕСКОЙ КОМПОЗИЦИИ

Один рисунок можно построить из небольшого числа понятных примитивов: отрезков, прямоугольников, окружностей и эллипсов. Такой способ удобен для учебных чертежей, структурных схем, графиков и иллюстраций к численным алгоритмам. Каждый объект задаётся отдельно, а сцена задаёт порядок их размещения.

3.2. Прямоугольник и скруглённые углы

Для прямоугольника в SVGC используется `__svg_new_rect(x, y, w, h, rx, ry)`. Первые две координаты указывают верхний левый угол; размеры w и h задают ширину и высоту. Параметры r_x и r_y служат для описания скругления углов. При нулевых значениях получается обычный прямоугольник с прямыми углами.

3.3. Окружность и эллипс

Окружность задаётся центром (c_x, c_y) и радиусом r : `__svg_new_circle(cx, cy, r)`. Эллипс — центром и двумя полуосями: `__svg_new_ellipse(cx, cy, gx, ry)`. Отдельные примитивы не требуют задания собственной системы координат; они располагаются на общем листе SVG.

3.4. Линии, заливки и обводки

С помощью `__svg_new_line()` создаётся отрезок между двумя точками. Цвет контура задаёт `__svg_set_stroke()`, цвет заливки — `__svg_set_fill()`, толщину линии — `__svg_set_stroke_width()`. Один и тот же примитив может использоваться как самостоятельная фигура, как часть условного обозначения или как фрагмент более сложной схемы.

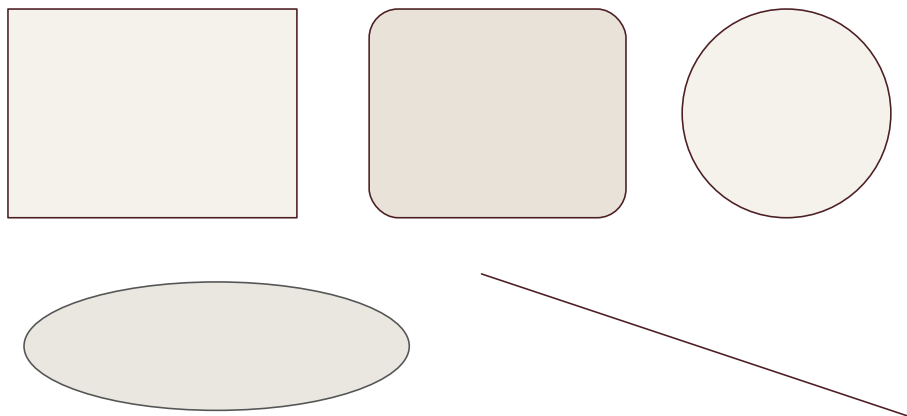


Рис. 3.1. Несколько базовых примитивов и вариации их оформления.

3.5. ПРИМЕР 2

Пример 002-primitives намеренно оставлен небольшим. Он формирует фон, два прямоугольника, окружность, эллипс и один отрезок, после чего записывает primitives.svg.

```

/*****
  SVG 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. Дуги, хорды и стрелки: НАПРАВЛЕНИЕ РАЗВИТИЯ

Дуга окружности, хорда и стрелка требуют дополнительного геометрического построения. В SVG C для составных контуров предусмотрены пути (*path*), дуги и кривые. Обычная хорда есть отрезок, соединяющий две точки дуги; стрелка может состояться из линии и замкнутого контура наконечника.

Мы пока не вводим новую функцию уровня библиотеки для каждой такой фигуры: её геометрические параметры определяются прикладным кодом. Подробная реализация разных окончаний стрелок, блок-схем и схем управления будет дана в последующих главах.

4. Координатные оси, определяемые данными

4.1. Почему оси не должны совпадать с рамкой

В вычислительных задачах математические координаты содержатся во входном массиве. Если просто принять левый верхний угол SVG за математический нуль, положительная часть оси y окажется направленной вниз, а симметричные отрицательные значения будут неудобны для изображения. Поэтому положение математических осей и масштабы должны вычисляться на основе диапазона данных и размеров полезной части листа.

4.2. Границы входного набора

Для набора пар (x_i, y_i) определим минимальные и максимальные координаты:

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

В учебном примере диапазон дополнительно расширяется так, чтобы включить математический нуль. Затем к диапазонам добавляется небольшой отступ по каждой оси. Это конкретное решение задачи построения графика, а не обязательное поведение SVGC. Для других задач (логарифмические шкалы, фиксированный масштаб, фазовые портреты) правило может отличаться.

4.3. Аффинное преобразование координат

Пусть полезный прямоугольник изображения начинается в точке (L, T) , имеет ширину W и высоту H . Тогда координаты точки (x, y) преобразуются в координаты SVG (X, Y) по формулам:

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

Таким образом математическое направление y вверх отображается в экранное направление SVG вниз. Математические оси выводятся из координат точки $(0, 0)$; они могут пересекать область построения где угодно, в зависимости от фактического массива.

При нулевой ширине диапазона надо заранее определить специальное правило: деление на нуль недопустимо. Учебная функция `find_bounds` расширяет вырожденный диапазон на ненулевую величину. В полноценной системе отображения такие решения должны быть сформулированы явно.

4.4. ПРИМЕР С ДИСКРЕТНЫМИ ЗНАЧЕНИЯМИ

Рассмотрим семь точек, полученных выборкой функции $y = x^2 - 2$ с шагом по x , равным единице. Эти значения выбраны только для объяснения геометрии; библиотека SVGC ничего не знает о формуле функции. Задача приложения — найти границы, определить масштаб, вывести оси и соединить точки ломаной.

```

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

Coordinate conversion depends on the actual data extrema.
SVG C 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

```

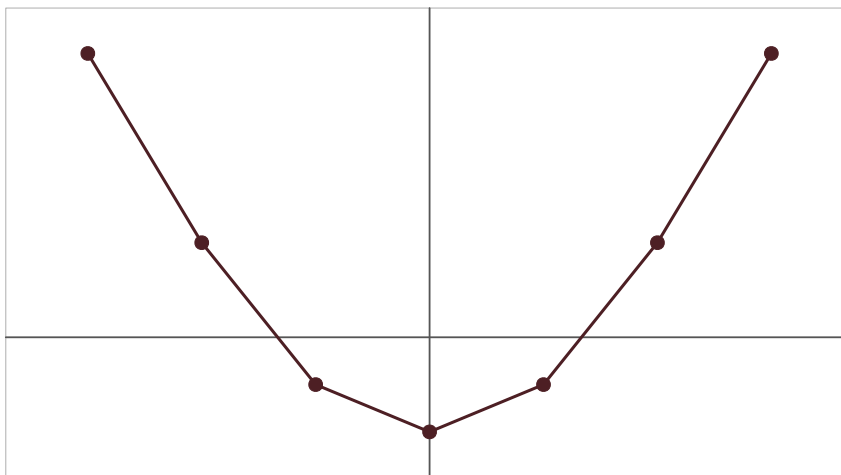


Рис. 4.1. Дискретный график: положение начала координат вычислено из массива точек.

Рисунок 4.1 намеренно выполнен в упрощённом виде. На данном этапе рассматривается построение ломаной линии по заданной последовательности точек, поэтому основное внимание уделяется геометрии отдельных отрезков и способу их представления в SVG. Вопросы оформления координатных осей, выбора масштаба, размещения делений и числовых подписей здесь пока не затрагиваются.

Следует также учитывать, что соединение соседних точек прямолинейными отрезками не обеспечивает гладкости полученной кривой. Во многих задачах научной графики требуется построить непрерывную гладкую линию, проходящую через заданные узлы, с использованием методов интерполяции.

В последующих главах будут рассмотрены построение и оформление координатных осей, а также применение кубических интерполяционных сплайнов. Эти средства позволят перейти от простейших геомет-

рических построений к созданию полноценных научных графиков, в которых форма кривых определяется математическими методами, а оформление соответствует требованиям к представлению результатов исследований.

5. Ломанные, многоугольники и дискретная геометрия

5.1. ОТ ОТДЕЛЬНЫХ ОТРЕЗКОВ К МАССИВУ ВЕРШИН

Отрезок определяется двумя точками. Когда изображение состоит из последовательности отрезков, удобнее хранить вершины в массиве, а не создавать каждый отрезок независимо. Так устроены каркасы технических чертежей, экспериментальные кривые, границы расчётных областей и многие схемы численных методов.

В SVGC этим двум задачам соответствуют разные элементы SVG: `polyline` обозначает **ломаную** (*polyline*) с открытым контуром, а `polygon` — **многоугольник** (*polygon*) с замкнутым контуром. Оба примитива получают упорядоченную последовательность двумерных точек.

Пусть массив содержит вершины

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

Ломаная состоит из отрезков $P_0P_1, \dots, P_{n-2}P_{n-1}$. Многоугольник, кроме этих отрезков, содержит замыкающую сторону $P_{n-1}P_0$. В исходном массиве для этого не обязательно повторять первую вершину.

5.2. ЧТО ОЗНАЧАЮТ ПОРЯДОК И ЗАМЫКАНИЕ

Порядок вершин определяет порядок следования отрезков. Если поменять местами две вершины, получится другая геометрия. Это относится и к многоугольникам: необдуманная перестановка может привести к самопересечению контура. Для таких фигур результат заполнения определяется правилами SVG; сам по себе тип `polygon` не проверяет, является ли область простой и геометрически корректной.

Незамкнутую ломаную следует, как правило, изображать без заливки, используя `fill:none`. Замкнутая фигура допускает и обводку (`stroke`), и заполнение (`fill`). При этом стиль не меняет массив координат: одни и те же вершины можно нарисовать различными толщинами и цветами.

5.3. ПУБЛИЧНЫЙ ИНТЕРФЕЙС SVGC

Создание последовательности точек начинается с `__svg_new_points()`. Каждая вершина добавляется через `__svg_add_point(points, x, y)`. Затем готовый набор передаётся одному из конструкторов: `__svg_new_polyline(points)` либо `__svg_new_polygon(points)`.

После успешного создания графического узла набор вершин принадлежит этому узлу и освобождается при освобождении соответствующей сцены. Если создание примитива не удалось, прикладная программа должна освободить ещё не переданный набор точек самостоятельно. В нашей программе это делается в функции `append_points()`.

Следует обратить внимание на то, что функция `__svg_add_point()` при успехе возвращает положительное значение, не обязательно равное единице. Поэтому проверка `< 1` правильна, а требование `== 1` отвергало бы вторую и последующие успешно добавленные вершины.

5.4. ИЗОБРАЖЕНИЕ ДИСКРЕТНЫХ ИЗМЕРЕНИЙ

В левой половине рисунка 5.1 помещена выборка восьми измерений. Соединяющая их линия — ломаная, а небольшие окружности подчёркивают координаты исходных узлов. Это различие принципиально: отсутствие промежуточных экспериментальных значений нельзя скрывать, выдавая ломаную за аналитическую кривую.

В правой половине изображена замкнутая область. Все её вершины заданы явно, а заполнение показывает, где находится внутренность фигуры.

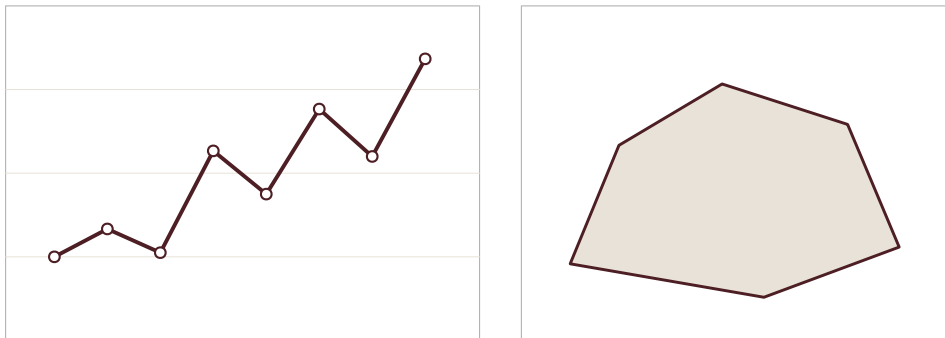


Рис. 5.1. Открытая ломаная дискретных данных и замкнутый многоугольник.

5.5. ПРИМЕР 4

Пример 004-polylines не требует дополнительных библиотек построения графиков. Его исходный массив является одновременно и геометрической моделью рисунка, и входом для SVG. Ни одной скрытой интерполяции библиотека не выполняет.

```

/*****
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. ГРАНИЦА МЕЖДУ ВИЗУАЛИЗАЦИЕЙ И ВЫЧИСЛЕНИЕМ

Положение этих вершин заранее задано в экранных координатах. Для численного эксперимента на их месте обычно будут точки (x_i, y_i) в математической системе координат. Тогда функция преобразования, рассмотренная в главе 4, сначала вычисляет экранные координаты, а `__svg_add_point()` получает уже готовый результат.

Массив вершин есть геометрическое представление исходных данных. SVGC сохраняет его в SVG-документе; смысл порядка точек, правило замыкания и выбор шкалы определяет прикладной алгоритм.

6. Стрелки и их окончания

6.1. ПОЧЕМУ СТРЕЛКА — СОСТАВНОЙ ОБЪЕКТ

В техническом рисунке стрелка несёт семантику направления: поток данных, порядок переходов, ориентацию координатной оси или действие силы. Для её построения недостаточно знать две конечные точки отрезка. Нужно также определить форму и размеры наконечника, направление, наличие второго окончания и способ соединения наконечника с линией.

Пример этой главы показывает пять конструкций: заполненный треугольный наконечник, открытый наконечник, двустороннюю стрелку, стрелку с ромбом в начале и треугольником в конце, а также осевую стрелку по образцу рисунка 16.1. Все они состоят из обычных SVG-примитивов, доступных через публичный интерфейс SVGC.

6.2. ЕДИНИЧНЫЙ НАПРАВЛЯЮЩИЙ ВЕКТОР

Пусть линия начинается в $A = (x_1, y_1)$ и заканчивается в $B = (x_2, y_2)$. Её длина и единичный направляющий вектор:

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

Повернув направление на прямой угол, получим нормаль

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

Если вершина наконечника находится в точке B , его длина равна ℓ , а половина ширины основания равна w , то углы основания заполненного треугольника вычисляются по формулам

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

Таким образом достаточно одного `polygon` с тремя вершинами. Открытый наконечник можно выполнить ломаной $C_+ \rightarrow B \rightarrow C_-$. Для ромба дополнительно используется противоположная вершина $B - \ell \mathbf{u}$ и две боковые вершины на расстоянии $\ell/2$ от острия.

Осевой наконечник (см. рис. 16.1) строится иначе: его контур является не многоугольником, а `path`. Сначала задаются две прямые стороны к острию, а затем основание замыкается короткой гладкой кривой. Такая форма особенно хорошо смотрится на концах координатных осей.

6.3. НАЧАЛО И КОНЕЦ ИМЕЮТ ПРОТИВОПОЛОЖНЫЕ НАПРАВЛЕНИЯ

Наконечник в конце линии строится с направлением $\mathbf{u}$, а в начале — с направлением $-\mathbf{u}$. Без этой операции двусторонняя стрелка имела бы два наконечника, направленных в одну сторону. Для ненулевой длины в программе применяется `hypot` из стандартной математической библиотеки C. Если концы совпадают, направляющий вектор неопределён: такую стрелку построить нельзя.

Размеры наконечника в настоящем примере заданы в экранных единицах. Они не зависят от длины самой линии. Это удобно для инженерных схем: длина связи меняется, но условное обозначение направления остаётся одного размера.

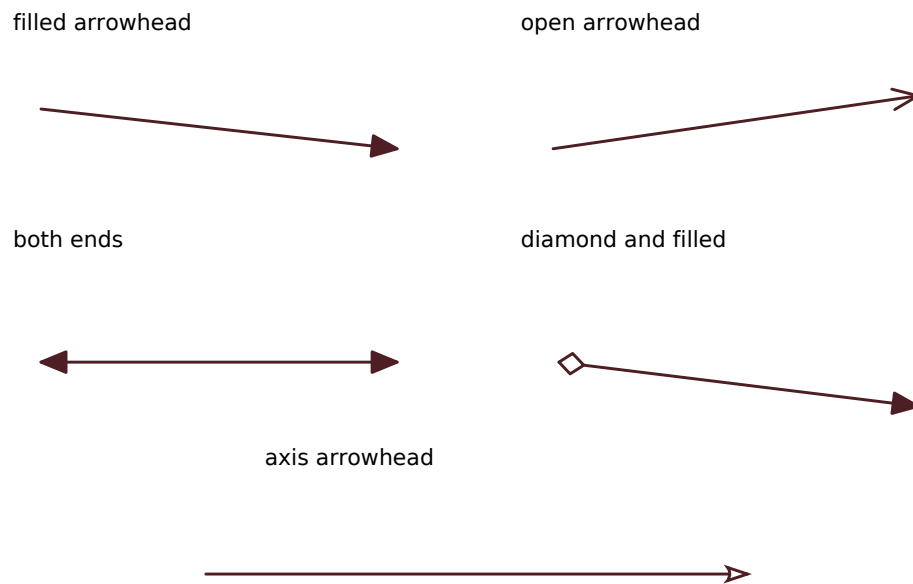


Рис. 6.1. Пять вариантов окончания направленных связей.

6.4. ПРИМЕР 5

В программе 005-arrows перечисление `head_type` задаёт семантику окончания. Функция `add_head` строит геометрию выбранного варианта, а `add_arrow()` объединяет отрезок с двумя возможными окончаниями. Для осевой стрелки введена отдельная функция `add_axis_head`, которая использует `path` и короткую гладкую кривую в основании.

```

/*****
  SVG C 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. МАСШТАБ, СТИЛИ И ПОРЯДОК НАЛОЖЕНИЯ

В коде отрезок добавляется в сцену раньше наконечника. Поэтому при закрашивании треугольником линия оказывается под его заливкой, а не пересекает видимую фигуру. Незакрашенный ромб и осевой наконечник имеют белую заливку, которая скрывает продолжение линии под ними.

В составных схемах полезно отдельно определять толщину основных связей, геометрию наконечников и цвет оформления. Применять ли цвет к информационным потокам или оставить монохромное изображение для печати — решение автора научной иллюстрации.

У стрелки есть смысловое направление и геометрическое окончание. Приложение вычисляет их явно, а SVGC записывает выбранную композицию отрезков, ломаных, многоугольников и путей path.

7. Программное построение блок-схем

7.1. ГРАФ УПРАВЛЯЮЩИХ ПЕРЕХОДОВ

Блок-схема программы есть граф: вершины обозначают действия, проверки и точки входа или выхода, а направленные рёбра — допустимые переходы управления. Рисунок полезен только тогда, когда его геометрия выражает настоящий алгоритм, а не произвольный порядок прямоугольников на странице.

В первом приближении можно выделить четыре условных элемента: эллипс начала или конца, прямоугольник действия, ромб условия и стрелку между ними. Из этих элементов строится схема итерационного вычислительного процесса.

7.2. ПРИМЕР: ВЫЧИСЛИТЕЛЬНЫЙ ЦИКЛ

Рассмотрим программу, выполняющую вычисления до тех пор, пока счётчик `step` меньше заданного значения `n`:

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

Приведённая запись намеренно является псевдокодом: здесь нас интересуют переходы, а не выбор конкретного языка моделирования. На схеме будут две ветви ромба: при истинном условии управление переходит к вычислительному блоку и возвращается к проверке; при ложном — завершается. В вычислительном прямоугольнике подчёркнуты оба действия: вычисление и увеличение счётчика.

Для удобства чтения рёбра рисуются до вершин. Затем добавляются эллипсы, прямоугольники и ромб, а подписи размещаются поверх фигур. Так мы не получаем нежелательных линий, проходящих через заливку блока. Маршрут возвращающейся ветви состоит из нескольких прямоугольных сегментов, вычисленных в C-коде.

7.3. ПРИМЕР 6

Функция `add_polygon()` служит для ромба условия и треугольных наконечников. Функции `add_line()` и `add_arrow()` определяют связи между блоками, а `add_text()` добавляет надписи. Это прикладные вспомогательные функции; они не добавлены в SVGC.

```
/******
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>
```

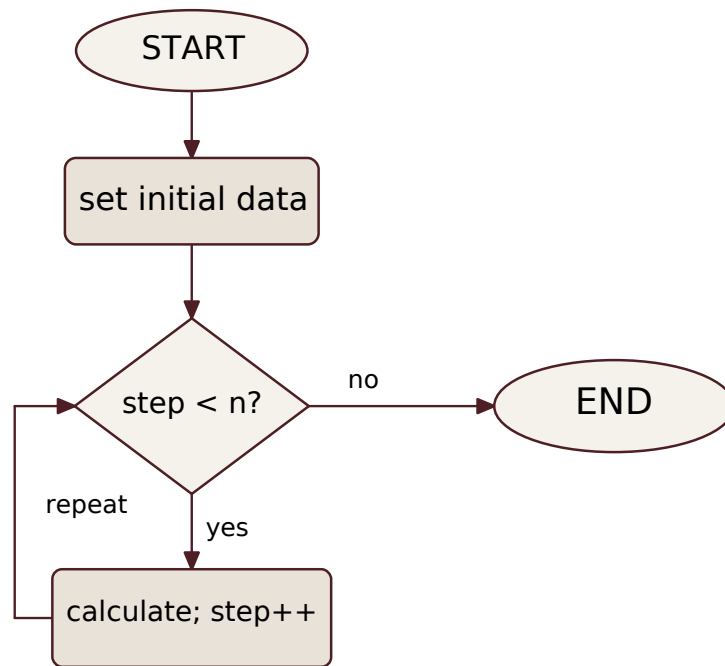


Рис. 7.1. Блок-схема итерационного алгоритма с выходом из цикла.

```

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. Подписи в блоках: ПОЧЕМУ ЦЕНТРИРОВАНИЕ ДЕЛАЕТСЯ ВРУЧНУЮ

В отличие от размеров прямоугольников и координат их вершин, фактическая ширина надписи не определяется только количеством букв. Каждый символ рисуется своим **глифом** (*glyph*): его горизонтальный размер зависит от выбранной гарнитуры, начертания, размера шрифта, межсимвольных расстояний и особенностей пробелов. В частности, буквы W и i обычно занимают разную ширину, даже если их размеры

в исходном тексте одинаковы.

Библиотека SVGC записывает название шрифта, размер и координаты элемента `text`, но не выполняет разбор файлов шрифтов и не вычисляет графическую ширину всей строки для конкретного механизма отображения. Поэтому нельзя надёжно вычислить центр надписи по формуле «количество символов умножить на постоянную ширину символа» или ожидать, что SVGC автоматически поместит её посередине блока.

В примере `006-flowchart` функции `add_text()` передаются *заранее подобранные вручную* координаты начала надписи. По этой причине её положение нужно проверить в браузере на готовом SVG, особенно после изменения шрифта или текста. Геометрический центр блока нам известен точно, но положение визуального центра строки определяется её фактическим рисунком.

Теоретически приложение может применить внешний механизм измерения текста или заранее определённые метрики конкретного шрифта, однако это уже отдельная задача компоновки, не относящаяся к функциям создания SVG-примитивов. Добавлять внутрь SVGC собственный шрифтовой движок ради центрирования подписей мы не будем.

7.5. Текст в SVG и безопасность XML

SVG — XML-документ. Некоторые символы, допустимые в строковой константе C, нельзя напрямую вставлять в содержимое XML-элемента. В частности, знак `<` обозначает начало разметки, поэтому в выходном документе он должен представляться как `<`. В рассматриваемой реализации SVGC строка текста записывается непосредственно, без универсального экранирования содержимого. Поэтому в исходнике мы передаём `step < n?`; браузер показывает привычное условие `step < n?`.

В реальном программном генераторе надписи могут поступать из внешних данных. В этом случае прикладная программа должна корректно экранировать специальные XML-символы, включая `&`, `<` и другие символы в зависимости от контекста. Нельзя считать, что любая последовательность байтов может быть помещена в XML без подготовки.

7.6. Автоматическая компоновка — отдельная задача

Наш пример не претендует на автоматическую расстановку блоков. Все размеры и позиции определены заранее. Для автоматического построения больших схем понадобились бы алгоритмы размещения вершин, маршрутизации рёбер, исключения пересечений и проверки читаемости. SVGC не должен принимать на себя задачи такого анализатора графов: он лишь записывает готовые линии и фигуры.

Именно по такому принципу можно будет позднее строить структурные схемы систем управления: математическая или функциональная модель определяет блоки и связи, прикладной алгоритм вычисляет их размещение, а библиотека формирует SVG.

Блок-схема не появляется из графического редактора сама собой. Она является геометрическим отображением графа управляющих переходов; алгоритм задаёт граф, прикладной C-код — композицию, SVGC — документ.

8. Оформление объектов: контур, заливка и штрих

8.1. ГЕОМЕТРИЯ И СТИЛЬ — РАЗНЫЕ СОСТАВЛЯЮЩИЕ РИСУНКА

Нарисованная линия представляет собой геометрический отрезок с координатами концов. Её видимый облик определяется отдельно: цветом, толщиной, способом завершения штриха и типом пунктирной последовательности. Так же и у прямоугольника: размеры задают область, но цвет заполнения не обязан влиять на её координаты. Это различие принципиально при создании научных рисунков программным способом.

В примере 007-styles одна и та же линия выводится с разными параметрами окончания и штриховки. Вся геометрия по-прежнему определяется исходной C-программой, а SVGC сохраняет параметры в свойстве `style` соответствующего SVG-элемента.

8.2. ЦВЕТ ОБВОДКИ И ЦВЕТ ЗАПОЛНЕНИЯ

Свойство `stroke` отвечает за цвет контура, `fill` — за заполнение замкнутой области. В интерфейсе SVGC применяются функции `__svg_set_stroke()`, `__svg_set_fill()`, а также варианты `..._rgb`, принимающие компоненты цвета. Значение `none` означает отсутствие соответствующей отрисовки, а не белый цвет. Белая заливка, напротив, является настоящей непрозрачной областью и может закрыть расположенные под ней линии.

Толщина `stroke-width` задаётся функцией `__svg_set_stroke_width()`. При значительной толщине линия визуально расширяется по обе стороны своей геометрической оси. Следовательно, даже идеально рассчитанная геометрия может выглядеть по-разному при смене оформления.

8.3. ОКОНЧАНИЯ ШТРИХА И СОЕДИНЕНИЯ СЕГМЕНТОВ

Для открытого отрезка предусмотрены три вида окончания: `butt` — ровное завершение точно у последней точки; `round` — скругление; `square` — прямоугольное окончание, выступающее за точку примерно на половину толщины линии. При построении мелких осей и стрелок это различие особенно заметно.

Соединения ломаной задаются свойством `stroke-linejoin`: `miter`, `round` или `bevel`. Острый угол при `miter` может давать длинный выступ. Выбор соединения должен исходить из характера рисунка, а не из случайного значения по умолчанию.

8.4. ПУНКТИР И РИТМ ИЗОБРАЖЕНИЯ

Последовательность длин видимых и пропущенных интервалов задаётся свойством `stroke-dasharray`. Например, `18,12` означает 18 единиц штриха, затем 12 единиц пробела и повторение этой пары. Это важно для обозначения вспомогательных линий, сетки и геометрических построений: пунктир не должен отвлекать от основного результата вычисления.

Для разреженного штрихового ряда рисунка 8.1 используется последовательность `20,20` с ровным окончанием каждого штриха `butt`. Равные интервалы дают легко различимый рисунок «штрих — пробел» даже после уменьшения иллюстрации при печати.

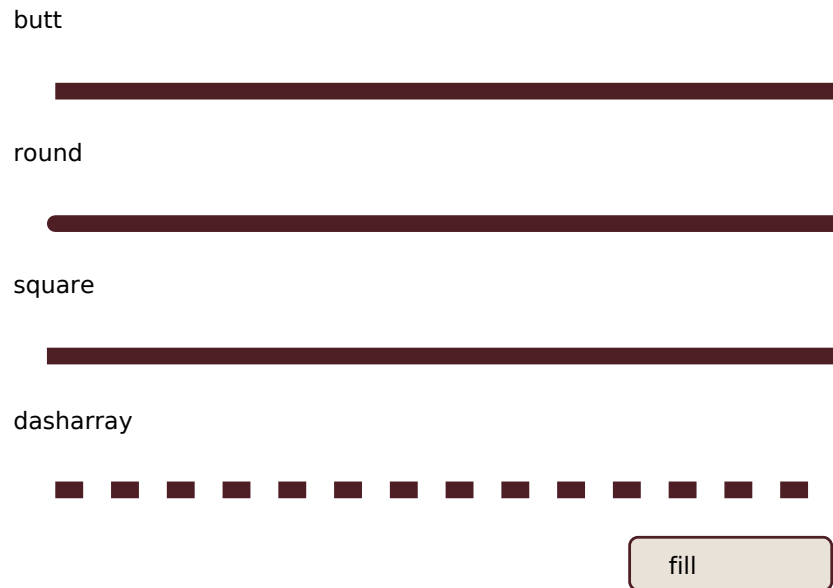


Рис. 8.1. Окончания линий, пунктирный контур и заполненный объект.

8.5. ПРИМЕР 7

В программе применяются `__svg_set_stroke_linecap()`, `__svg_set_stroke_linejoin()` и `__svg_set_stroke_dasharray()`. Для сравнения изменяется лишь оформление четырёх отрезков, а их направления и положение задаются теми же координатными правилами.

```

/*****
  SVGC BOOK EXAMPLE 007: STROKE AND FILL

  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_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

```

Сначала математический алгоритм определяет геометрию объекта. Затем оформление помогает читателю правильно интерпретировать эту геометрию, не изменяя вычисленных данных.

9. Составной контур SVG: команды path

9.1. Путь как последовательность геометрических действий

Для многих научных иллюстраций отдельного отрезка или прямоугольника уже недостаточно. Требуется одна последовательность команд, составляющих общую границу: начать контур, перейти к новой вершине, провести отрезок, выполнить изгиб и замкнуть фигуру. В SVG эта последовательность описывается элементом `path` и атрибутом `d` (**path data** (данные контура)).

Такой контур не обязан быть замкнутым. Внутри одного элемента могут находиться несколько **подпутей** (*subpaths*), каждый из которых начинается отдельной командой перемещения. Составной путь позволяет сохранить логическую целостность объекта, порождаемого алгоритмом, без ручного склеивания множества элементов.

9.2. ПЕРЕМЕЩЕНИЕ, ЛИНИЯ И ЗАМЫКАНИЕ

Команда `M` начинает новый подпуть в указанной точке, `L` проводит прямую к следующей точке. Команды `H` и `V` проводят соответственно горизонтальный и вертикальный отрезки. Команда `Z` соединяет текущую точку с началом подпути и замыкает его. Поэтому `Z` имеет самостоятельный геометрический смысл: это не простой переход к следующему объекту.

В SVGC этим конструкциям соответствуют функции `__svg_add_moveto_cmd()`, `__svg_add_lineto_cmd()`, `__svg_add_hlineto_cmd()`, `__svg_add_vlineto_cmd()` и `__svg_add_closepath_cmd()`. Список команд создаётся посредством `__svg_new_cmds()`, а сам узел контура — `__svg_new_path()`.

9.3. АБСОЛЮТНЫЕ И ОТНОСИТЕЛЬНЫЕ КООРДИНАТЫ

Заглавные буквы `M`, `L`, `H`, `V` означают абсолютные координаты сцены. Строчные `m`, `l`, `h`, `v` задают смещения относительно текущей точки. В публичном интерфейсе SVGC выбор производится аргументами `ABSOLUTE` и `RELATIVE`.

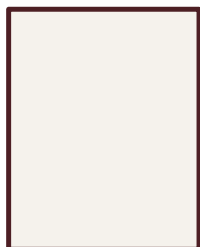
Например, после `M 350, 170` команда `h 155` приведёт к точке (505, 170), а последующая `v 145` — к точке (505, 315). Тот же контур можно получить только абсолютными координатами. Но смешивать эти способы без строгого учёта текущей точки опасно: ошибка будет накапливаться вдоль последовательности.

9.4. КОНТУР, ЗАЛИВКА И ПОРЯДОК ОБХОДА

Замыкание делает построенную границу пригодной для заливки. При этом сама заливка определяется стилем, а не командой `Z`. Даже незамкнутый путь может визуально иметь заполненную область по правилам SVG, поэтому для геометрически определённой замкнутой фигуры лучше явно указывать замыкание.

Порядок обхода границы особенно важен в сложных контурах с несколькими компонентами и отверстиями. В таком случае понадобится понимание правила заполнения *fill-rule* и ориентации обхода. В нашем вводном примере эта сложность отсутствует: каждый путь содержит один простой замкнутый контур.

M H V L Z (absolute)



M h v l z (relative)



Рис. 9.1. Два одинаковых по принципу построения контура: абсолютные и относительные команды SVG path.

9.5. ПРИМЕР 8

Обе фигуры строятся одним и тем же набором функций библиотеки. Различаются лишь значения ABSOLUTE и RELATIVE; результирующие SVG-команды можно непосредственно прочитать в текстовом редакторе и сопоставить с C-исходником.

```

/*****
  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
```

Путь является записью алгоритма обхода геометрической границы. Библиотека не должна угадывать ни последовательность вершин, ни математический смысл замкнутой области.

10. Квадратичные и кубические кривые Безье

10.1. ПЕРЕХОД ОТ ЛОМАННОЙ К ПЛАВНОЙ КРИВОЙ

Ломаная соединяет соседние точки отрезками. Для графиков, траекторий и гладких технических контуров это не всегда достаточно: необходимо контролировать направление касательной и форму изгиба. Кривые Безье позволяют описать такой изгиб через конечные точки и вспомогательные **управляющие точки** (*control points*).

Они не должны путаться с исходными точками наблюдений. Управляющие точки обычно не принадлежат самой кривой, но определяют её касательные и форму. Поэтому вычисление управляющих точек из численных данных является задачей прикладного алгоритма, а не обязанностью SVGC.

10.2. КВАДРАТИЧНАЯ КРИВАЯ

Квадратичная кривая задана тремя точками: началом P_0 , управляющей точкой P_1 и концом P_2 . Её параметрическое уравнение при $0 \leq t \leq 1$ имеет вид

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

Касательная в начале направлена вдоль $P_1 - P_0$, а в конце — вдоль $P_2 - P_1$. SVG-команда Q задаёт такую кривую, а команда T продолжает её, используя отражение предыдущей управляющей точки относительно общего конца (если предыдущей командой была Q или T).

В SVGC эти действия доступны через `__svg_add_quadro_curveto_cmd()` и `__svg_add_short_quadro_curveto_cmd()`. Таким способом удобно строить простые плавные очертания и последовательности дугообразных сегментов.

10.3. КУБИЧЕСКАЯ КРИВАЯ

Кубическая кривая использует две управляющие точки P_1 и P_2 между началом P_0 и концом P_3 :

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

Производные на концах равны

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

Следовательно, положение управляющих точек задаёт не только степень изгиба, но и направления касательных.

В SVG кубический сегмент записывается командой C. Продолжение S восстанавливает первую управляющую точку из отражения второй точки предшествующего кубического сегмента. Соответствующие функции SVGC называются `__svg_add_curveto_cmd()` и `__svg_add_short_curveto_cmd()`.

10.4. ГЕОМЕТРИЯ УПРАВЛЯЮЩИХ ТОЧЕК

На рисунке показаны квадратичная и кубическая кривые. Для квадратичной кривой тонкие пунктирные отрезки соединяют исходные и управляющую точки. Для кубической кривой тонкие сплошные отрезки

показывают направления касательных в концах, а пунктирный отрезок соединяет две управляющие точки. Эти вспомогательные элементы создаются отдельными SVG-примитивами: они необходимы для объяснения геометрического смысла кривых, но не являются частью описания самой кривой.

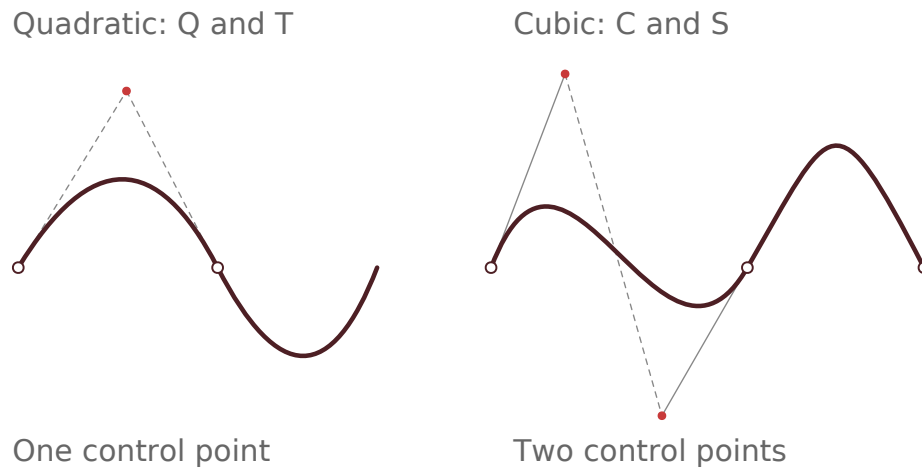


Рис. 10.1. Два класса кривых Безье: квадратичная с одной управляющей точкой и кубическая с двумя.

10.5. Сглаживание ДИСКРЕТНЫХ ДАННЫХ НЕ ЯВЛЯЕТСЯ РИСОВАНИЕМ НАУГАД

Когда точки возникают из наблюдений, метода интегрирования или результата параметрического расчёта, выбор управляющих точек должен быть обоснован. Простое желание получить красивую линию может привести к появлению ложных экстремумов и колебаний, которых нет в исходных данных.

Позднее мы отдельно разберём кубические интерполяционные сплайны: сначала система уравнений определяет согласованные производные, затем производные переводятся в управляющие точки сегментов Безье, и только после этого библиотека записывает графическую кривую. Так сохраняется принцип «сначала вычисление, затем изображение».

10.6. ПРИМЕР 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
```

Гладкость кривой определяется математическим выбором управляющих точек. Библиотека SVG C лишь точно записывает эти точки в SVG.

11. Эллиптические дуги, хорды и секторы

11.1. ЭЛЛИПТИЧЕСКАЯ ДУГА КАК САМОСТОЯТЕЛЬНЫЙ ЭЛЕМЕНТ ПУТИ

Математическая окружность или эллипс описывают замкнутую границу. В инженерной графике часто требуется только её часть: например, угол поворота, сектор плоскости, область действия или дуга сопряжения. SVG позволяет задавать такие участки отдельной командой `A` в составе общего `path`.

В отличие от кривой Безье, дуга определяется не управляющими точками, а радиусами эллипса, поворотом его осей и выбором одного из допустимых участков между двумя конечными точками. Аргументы функции `__svg_add_arc_cmd()` включают `rx`, `ry`, `x_axis_rotation`, флаги `large_arc_flag` и `sweep_flag`, а также координаты конечной точки.

11.2. ФЛАГИ ВЫБОРА ДУГИ

Для фиксированных начальной и конечной точек и допустимых радиусов в общем положении существует четыре варианта дуги. Флаг `large-arc-flag` выбирает малую или большую дугу, а `sweep-flag` — направление обхода согласно экранной системе координат SVG. Значения обоих флагов должны быть 0 или 1.

При этом «0» и «1» нельзя заменять понятиями «по часовой» и «против часовой» без указания системы координат. В SVG ось y направлена вниз, поэтому визуальная ориентация отличается от принятой в математической декартовой плоскости. Если заданные радиусы слишком малы для соединения выбранных концов, SVG нормализует их при вычислении геометрии эллипса. Это следует учитывать при подготовке точных научных чертежей.

11.3. ДУГА, ХОРДА И СЕКТОР

Пусть концы дуги находятся в точках A и B . **Хорда** (*chord*) — прямой отрезок AB . Следовательно, замкнутая фигура, ограниченная дугой и хордой, может быть построена последовательностью `M A`, затем `A ... B`, затем `Z`. Последняя команда замыкает путь прямым отрезком до A .

Сектор (*sector*) — область, ограниченная дугой и двумя отрезками, соединяющими её концы с центром O . В простом случае его путь имеет вид `M O, L A, A ... B, Z`. Для эллиптического сектора вместо одного радиуса используются две полуоси, а точка O обозначает центр эллипса.

11.4. ПРИМЕР 10

В программе три функции используют один и тот же механизм создания пути, но с разными последовательностями команд: `A`; `A` с замыканием; переход от центра к началу дуги, `A` и замыкание. Эта композиция намеренно остаётся на стороне приложения, а не добавляется в виде специальных примитивов SVGC.

```

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

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

```

Arc: A (0 0)

Chord: A Z

Sector: L A Z

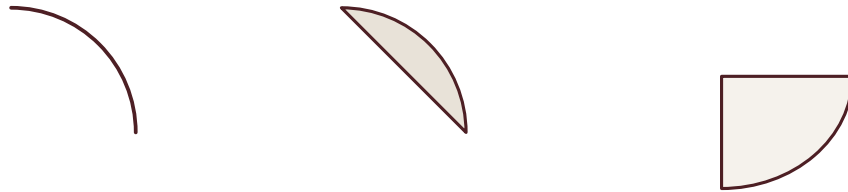


Рис. 11.1. Открытая дуга, область между дугой и хордой, эллиптический сектор.

```

#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
```

Для точных чертежей полезно отличать математический эталон — параметры дуги в прикладной программе — от её полиграфического приближения. Печатный рисунок не должен незаметно менять геометрический смысл исходных данных.

Дуга, хорда и сектор являются геометрическими построениями. Программа выбирает концы, радиусы, направление и порядок соединения, а SVGC сохраняет готовую конструкцию как человекочитаемый путь SVG.

12. Надписи, шрифты и текстовые координаты

12.1. ТЕКСТ ЯВЛЯЕТСЯ ГРАФИЧЕСКИМ ОБЪЕКТОМ

В научной иллюстрации подпись одновременно является частью содержания и частью геометрической композиции. Её строка хранится в текстовом узле SVG, но видимый результат зависит от конкретной гарнитуры, размера и шрифтового движка. В программе на C надпись создаётся посредством `__svg_new_text(x, y, text, font_family, font_size)`. Аргументы x и y задают положение текстовой базовой линии (**baseline** (базовая линия)) и начала текстовой строки, а не верхний левый угол прямоугольника, окружающего глифы.

Для сравнения нескольких гарнитур в примере 12.1 нарисованы светлые направляющие, проходящие через базовые линии. Увидеть разницу проще всего у символов с нижними выносными элементами, например g и y : они выходят ниже базовой линии, хотя значение y для всей строки постоянно.

12.2. ГАРНИТУРА, РАЗМЕР И ЦВЕТ

Функции `__svg_set_text_font_family()`, `__svg_set_text_font_size()` и `__svg_set_text_fill_rgb()` позволяют менять параметры уже созданного текстового узла. В нашем примере используются DeJaVu Sans, DeJaVu Serif и DeJaVu Sans Mono. Внешний вид на другой системе зависит от наличия этих шрифтов и от правил подстановки: указание гарнитуры в SVG само по себе не встраивает её файл в изображение.

12.3. ПОЧЕМУ ШИРИНУ СТРОКИ НЕЛЬЗЯ УЗНАТЬ ИЗ ЧИСЛА СИМВОЛОВ

Ширина надписи определяется суммой реальных горизонтальных метрик глифов, пробелов, парных поправок и настроек шрифта. Даже при одинаковом числе символов строки `iiii` и `WWWW` в пропорциональной гарнитуре заметно различаются по ширине. Описание координаты и имени шрифта в SVG не означает, что библиотека располагает механизмом измерения текстового рисунка на целевой системе.

Поэтому надпись в нижнем прямоугольнике на рисунке позиционируется вручную после просмотра исходного SVG. Нельзя смешивать это прикладное решение задачи компоновки с записью SVG-узлов в библиотеке. При изменении шрифта, масштаба или текста координаты могут потребовать коррекции.

12.4. ПРИМЕР 11

C-программа создаёт лист, три направляющие, несколько текстовых объектов и область для подписи с ручным выравниванием. Отдельный обработчик сложной шрифтовой геометрии не применяется.

```

/*****
SVG C BOOK EXAMPLE 011: TEXT AND FONT PRESENTATION

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

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

```

Sans: scientific illustration

Serif: scientific illustration

Mono: 0123456789 ABC xyz

Manual alignment
W and i have different widths

Рис. 12.1. Базовые линии и визуальные различия гарнитур и надписей.

```
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_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. ЭКРАНИРОВАНИЕ ТЕКСТА В XML

Символы < и & нельзя без обработки передавать в XML-текст: для них нужны сущности < и &. Если исходные строки поступают из файлов, пользовательского ввода либо численного эксперимента, прикладная программа должна гарантировать корректное экранирование и допустимую кодировку. Создание текста не должно незаметно превращаться в парсинг шрифтов или в преобразование произвольных строк.

12.6. ВНЕШНИЕ ШРИФТЫ И ОПРЕДЕЛЕНИЯ CSS ВНУТРИ SVG

Выбор имени гарнитуры не гарантирует, что на компьютере читателя она установлена. SVG позволяет поместить правила CSS в элемент `style` внутри секции `defs`. Например, правило `@font-face` описывает семейство шрифта, его начертание и внешние файлы `woff2`. Следует различать две операции: *встраивание*

текста CSS в SVG и встраивание самих двоичных данных шрифта. В первом случае файл шрифта остаётся внешним ресурсом, на который ссылается `url(...)`.

Библиотека предоставляет функцию `__svg_sheet_add_css_file(sheet, fname)`. Аргумент `fname` указывает на обычный файл CSS, содержащий только правила стилей, без окружающих тегов XML, `defs`, `style` или разделителей CDATA. Библиотека самостоятельно добавляет оформление и сохраняет текст файла в определениях конкретного листа. Проверять возвращаемое значение необходимо: 1 означает успех, 0 — ошибку чтения или содержимого, -1 — неверные аргументы.

```
/* 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. Один ОБЩИЙ КАТАЛОГ ШРИФТОВ ДЛЯ ПРИМЕРОВ

Отдельный архив примеров, сопровождающий книгу, содержит общий каталог `svgc-examples/fonts/`. Все программы из этого архива, которым необходимы специальные шрифты, могут использовать этот каталог, не дублируя одни и те же файлы шрифтов в каждом примере. CSS-правила при этом хранятся рядом с программой, создающей конкретный SVG, и могут включаться непосредственно в документ. Поэтому относительные пути `url(...)` во встроенном CSS разрешаются относительно *готового SVG-файла*, а не относительно исходного файла CSS.

12.8. НИЖНИЕ И ВЕРХНИЕ ИНДЕКСЫ ПОСРЕДСТВОМ TSPAN

Например, на рисунке 14.1 важна корректная математическая типографика: индексы в M_z , I_γ , k_θ нельзя заменять последовательностью обычных символов. В SVG текстовая фигура `text` может содержать дочерние элементы `tspan`. Они задают индивидуальные свойства фрагментов текста, включая `font-size`, `font-style`, относительное смещение `dy` либо атрибут `baseline-shift`.

Например, строка, переданная в `__svg_new_text()`, может содержать следующий XML-фрагмент:

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

Именно `tspan`, а не HTML-элемент `span`, является частью языка SVG. В отличие от внешних шрифтовых файлов, такой текстовый фрагмент хранится непосредственно в XML. Передача в SVGC готовой разметки предполагает, что прикладная программа отвечает за её XML-корректность. SVG-просмотрщик вычисляет фактические метрики текста по выбранному шрифту; SVGC эти метрики не моделирует. При подготовке печатного PDF следует проверить, что индексы остаются на своих местах и сохраняют нужный размер.

Греческие символы в SVG. В математических подписях можно использовать символы греческого алфавита, записанные непосредственно в UTF-8. Например, на рисунке механического маятника (рис. 16.1) буква φ выводится следующим образом:

```
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 );
```

Такая запись сохраняет символ и его оформление в самом SVG. При подготовке PDF через CairoSVG нужно убедиться, что выбранная гарнитура содержит нужные глифы, а итоговый файл правильно отображается в программе просмотра. Печатное изображение создаётся непосредственно из SVG, без переписывания греческих букв вручную.

Программа располагает надпись в выбранной системе координат, а фактическая ширина и высота её глифов зависят от шрифта и устройства отображения. SVG сохраняет и воспроизводит текст, но не делает его метрически независимым от шрифта.

12.9. РАЗМЕРЫ И ГАРНИТУРЫ ШРИФТОВ ПРИ ПРЕОБРАЗОВАНИИ SVG В TEX

Если требуется точно передать гарнитуры и размеры шрифтов, рекомендуется создавать векторный PDF непосредственно из SVG средствами CairoSVG:

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

Полученный файл включается в документ TeX обычной командой `\includegraphics`, задающей необходимую ширину рисунка:

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

Подобный путь особенно удобен потому, что некоторые преобразователи SVG в графические команды TeX не всегда воспроизводят все свойства текста. Например, утилита `svg2tikz` может пропускать атрибут `font-size` и встроенные CSS-определения шрифтов. Кроме того, в старых версиях и отдельных сценариях работы Inkscape встречается переход между декартовым представлением с осью Y , направленной вверх (с началом отсчёта внизу слева), и привычными координатами SVG, где начало отсчёта обычно расположено вверху слева. Такой переход иногда записывается цепочками преобразований вида `transform="scale(1,-1) translate(...)"`. Старые скрипты, например `Inkscape2TikZ`, могут испытывать затруднения при обработке подобных вложенных преобразований и сложных CSS-стилей.

CairoSVG, напротив, непосредственно обрабатывает SVG-геометрию, векторные кривые, атрибуты `font-size` и систему координат исходного изображения. При наличии необходимых гарнитур полученный PDF сохраняет векторную графику и оформление подписей. Команда `\includegraphics` позволяет масштабировать такой рисунок без растривания; окончательное качество и корректность встраивания шрифтов полезно проверить в готовом PDF.

Встроенные в SVG CSS-правила или сценарии JavaScript, например добавленные функциями `__svg_sheet_add_css_file()` и `__svg_sheet_add_script_file()`, не всегда поддерживаются CairoSVG: при обработке элементов `style` или `script` конвертер может завершиться ошибкой. Перед экспортом такие элементы можно временно удалить из копии SVG, сохранив оригинал. Если удаление выполнялось непосредственно в исходном файле, после экспорта элементы необходимо вернуть на место. Полученный рисунок следует проверить: удаление CSS может изменить оформление.

13. Линейные и радиальные градиенты

13.1. ОПРЕДЕЛЕНИЯ И ССЫЛКИ НА НИХ

Однородная заливка описывается единственным цветом. **Градиент** (*gradient*) позволяет изменять цвет непрерывно внутри площади объекта. В SVG он оформлен как отдельное определение в секции defs; сам прямоугольник или другой объект ссылается на него значением `fill="url(##identifier)"`.

В SVGC линейный градиент создаётся функцией `__svg_new_l_gradient()`, радиальный — `__svg_new_r_gradient()`. Они помещаются в лист посредством `__svg_sheet_add_definition()`; цветовые остановки создаются `__svg_new_stop()` и добавляются с помощью `__svg_gradient_add_stop()`. Значение `offset` определяет относительное положение заданного цвета.

13.2. ЛИНЕЙНОЕ ИЗМЕНЕНИЕ ЦВЕТА

Для двух остановок с относительными координатами 0 и 1 линейная интерполяция каждого канала цвета имеет вид

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

Каналы могут быть заданы обычными компонентами RGB. В нашем рисунке первое определение направлено слева направо, поэтому разница хорошо заметна на широком прямоугольнике. Параметр `objectBoundingBox` связывает координаты градиента с геометрией закрашиваемого объекта.

13.3. РАДИАЛЬНЫЙ ГРАДИЕНТ

Радиальное изменение цвета зависит от расстояния до фокуса и границы градиентной области. В простом симметричном случае фокус совпадает с центром; параметры `sx`, `sy`, `r`, `fx`, `fy` задают геометрию этого изменения. Для прямоугольного объекта радиальное распределение в координатах `objectBoundingBox` визуально может выглядеть эллиптическим из-за неодинаковых размеров сторон.

linearGradient



radialGradient



Рис. 13.1. Линейная и радиальная заливки как ссылки на определения SVG.

13.4. ПРИМЕР 12

Программа создаёт два определения и по две цветовые остановки на каждое. После регистрации определения `__svg_create_node_url()` получает ссылку, которую можно передать `__svg_set_fill()`. Полученная строка освобождается приложением после установки стиля; менять публичный интерфейс SVG C для применения градиента не нужно.

```

/*****
  SVG C 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. ПРОВЕРКА ПЕЧАТНОГО ПРЕОБРАЗОВАНИЯ

После преобразования SVG в PDF средствами CairoSVG необходимо проверить направление градиента, расположение цветовых остановок и границы заливки. Особого внимания требуют радиальные градиенты: отображение в браузере и в печатном PDF должно совпадать по смыслу и сохранять цветовые переходы. Исходный SVG остаётся эталоном; печатный результат проверяется визуально.

Градиент является определением графического стиля, а не методом вычисления численных данных. Описание рисунка может многократно ссылаться на одно определение без дублирования его остановок у каждого объекта.

14. Структурная схема гироскопической системы

14.1. ГРАФИЧЕСКОЕ ОПИСАНИЕ МАТЕМАТИЧЕСКОЙ МОДЕЛИ

Структурные схемы систем управления отличаются от блок-схем программ. Здесь прямоугольники представляют функциональные преобразования сигналов, сумматоры — алгебраические комбинации, а стрелки задают направление передачи величин. Входы и выходы могут представлять физические величины и их производные.

В качестве законченного примера рассмотрим структурную схему, опубликованную на рисунке 1.6 в разделе «Графическое программирование» на сайте <https://dynamics.su/theory/issues/>. Исходная C-программа использует SVG C для построения структуры активной гироскопической системы.

14.2. ДИФФЕРЕНЦИАЛЬНАЯ МОДЕЛЬ СИСТЕМЫ УГЛОВОЙ СТАБИЛИЗАЦИИ

На рис. 14.1 изображена система угловой стабилизации объекта с гироскопическими управляющими органами по каналу тангажа. Ей соответствует система дифференциальных уравнений, опубликованная на странице <https://dynamics.su/theory/issues/> под номером (1.9):

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

где $M_\beta = k_\theta \theta + k_{\dot{\theta}} \dot{\theta}$ — закон управления.

14.3. БЛОКИ, СВЯЗИ И ПЕРЕДАТОЧНЫЕ МНОЖИТЕЛИ

Слева направо прослеживается прохождение входного момента M_z через сумматор к интегрирующим звеньям. Прямоугольники с дробями содержат множители, включающие момент инерции I_z , переменную Лапласа s и интегратор $1/s$. Нижняя ветвь содержит вторую координату β , множитель $1/(I_\gamma s)$ и связи с коэффициентами H и μ . Обратные связи формируют взаимозависимость угловых переменных.

На рисунке используются также множители $-2H$ и операторный блок закона управления. Их следует трактовать как математические операции, а не как декоративные обозначения. Динамическое содержание системы задаётся её уравнениями, а программа рисования воспроизводит их структуру.

14.4. СУММАТОРЫ, ОТВЕТВЛЕНИЯ И ПОРЯДОК НАЛОЖЕНИЯ

Сумматоры представлены окружностями с диагональными линиями. Малые заливные кружки имеют радиус, сопоставимый с толщиной соединительных линий: они отмечают узлы ответвления и устраняют двусмысленность Т-образных пересечений. Направление каждой связи показывается отдельным наконечником.

Последовательность создания графических объектов имеет значение: линии и стрелки, затем функциональные прямоугольники с заливкой, наконец текстовые обозначения. При подготовке печатной схемы проверяют контраст контуров, подписи коэффициентов и соединения ветвей. Геометрия утверждённого рисунка сохраняется.

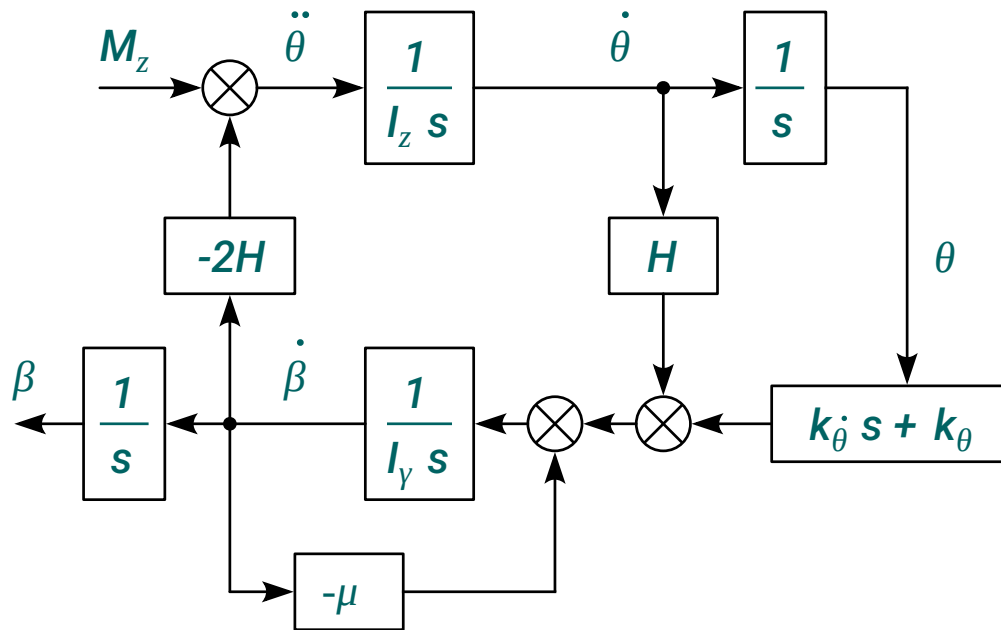


Рис. 14.1. Структурная схема активной гироскопической системы.

14.5. МАТЕМАТИЧЕСКИЕ ПОДПИСИ И ВЛОЖЕННЫЕ TSPAN

Заголовки сигналов и передаточных звеньев требуют индексов: M_z , I_z , I_y , k_{θ} . В C-программе это решено средствами SVG, а именно вложенными элементами `tspan` внутри текстового узла `text`. У отдельных фрагментов изменяются начертание, размер и смещение по базовой линии.

Нельзя вычислить точную ширину такого обозначения только по длине его исходной строки: нужно знать метрики каждого глифа, включая пробелы и уменьшенные символы индексов. Поэтому координаты всех надписей определяются в самой прикладной программе и контролируются визуально.

14.6. ПОДКЛЮЧЕНИЕ CSS СРЕДСТВАМИ SVGC

Для оформления шрифтами Roboto и Georgia программа читает отдельный `fonts.css` файл посредством функции `__svg_sheet_add_css_file()`. Это обычный текст CSS; никаких тегов `style` или `CDATA` в нём писать не нужно. Библиотека включает CSS в раздел `defs` готового SVG. Пути к шрифтам, используемые в `url(...)`, остаются внешними и должны быть доступны относительно расположения выходного файла `control-system.svg`.

14.7. ПРИМЕР 13

Ниже приведён исходный текст построения схемы с проверкой подключения CSS. Положения функциональных звеньев, алгоритмы стрелок, оригинальные текстовые фрагменты и топология оставлены неизменными. Пример не требует расширения SVGC и использует только публичный интерфейс библиотеки.

```
#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_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 );
    (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

```

Структурная схема является графической формой математической модели. Её математические надписи, узлы ответвлений и направления связей имеют смысл, который должен сохраняться при создании SVG, выборе шрифтов и подготовке печатного варианта.

15. Схема классического метода Рунге–Кутты

15.1. ОТ ФОРМУЛЫ К СТРУКТУРЕ ВЫЧИСЛЕНИЯ

Классический метод Рунге–Кутты четвёртого порядка удобно описывать не только формулами, но и структурной схемой. На каждом шаге интегрирования вычисляются четыре промежуточных наклона k_1, k_2, k_3, k_4 , после чего они объединяются в взвешенную сумму. Такой рисунок полезен не как замена формуле, а как наглядное представление порядка вычислений и зависимостей между промежуточными величинами.

Для дифференциального уравнения $y' = f(t, y)$ схема одного шага имеет вид

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

15.2. ГЕОМЕТРИЯ РИСУНКА

Начальное состояние t_n, y_n расположено слева. Справа от него строится вертикальная колонка из четырёх прямоугольников — по одному на каждую стадию. Между соседними блоками проходят вертикальные стрелки **predict** (*предсказать*): каждая стадия использует результат предыдущей стадии для построения новой промежуточной точки. Справа помещён блок итогового объединения, в который входят четыре горизонтальные стрелки, помеченные k_1, k_2, k_3, k_4 .

Такой рисунок хорошо показывает, что метод RK4 — это не одна формула, а организованный граф вычислений. Именно этот граф потом реализуется в программе численного интегрирования.

Classical four-stage Runge-Kutta method

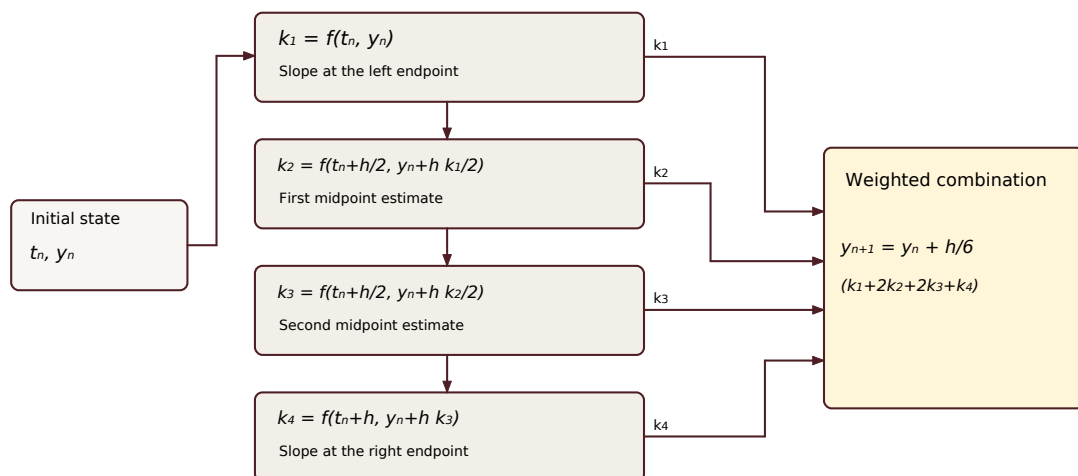


Рис. 15.1. Структурная схема вычисления одного шага классического метода RK4.

15.3. ПРИМЕР 14

Пример не строит численную траекторию и не вычисляет реальные значения k_i . Его цель — показать, как программно оформить схему вычислительного алгоритма: прямоугольные блоки, направленные связи, подписи и итоговый комбинирующий блок. В этом смысле он находится на границе между инженерной схемой и иллюстрацией к численному методу.

Как и в предыдущих главах, текст в прямоугольниках центрируется вручную по визуальной проверке. Автоматическое вычисление ширины строки потребовало бы анализа метрик шрифта, чем SVG C специально не занимается.

```

/*****
  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
```

После такой структурной иллюстрации можно переходить к более содержательным рисункам: графикам решения, дискретным данным, сплайнам и схемам исследования нулей и полюсов.

Если алгоритм имеет ясную внутреннюю структуру, её полезно рисовать как отдельный объект исследования. SVGC позволяет получить такую схему программно и воспроизводимо.

16. Механический осциллятор: геометрия сил и связей

16.1. АВТОРСКИЙ РИСУНОК КАК ПРОГРАММА

В предыдущей главе мы изображали последовательность вычислений численного метода. Теперь рассмотрим принципиально иной случай: *механическую схему*, в которой положение тела, направление сил и вспомогательные построения рассчитываются из параметров.

Основой настоящей главы служит рисунок 2.1, опубликованный на странице <https://dynamics.su/theory/issues/#targets-setting> в разделе «Построение программной модели». На рисунке показаны подвес, стержень, материальная точка M , вертикаль, геометрические дуги и система направленных отрезков. Буквы $B, G, G_\tau, G_n, N, \tau, n$ и ν нанесены как отдельные текстовые элементы; индексы формируются вложенными SVG-элементами `tspan`.

Программа *не решает дифференциальное уравнение* и не строит траекторию движения. Она конструирует один параметризованный геометрический кадр, который затем можно использовать при постановке и исследовании математической модели.

16.2. СИСТЕМА КООРДИНАТ И ГЕОМЕТРИЧЕСКАЯ ПАРАМЕТРИЗАЦИЯ

Пусть неподвижная точка подвеса обозначена $O = (x_0, y_0)$, длина стержня равна r_0 , а его направление задаётся углом α относительно горизонтальной оси, направленной вправо. В SVG ось y направлена вниз. Поэтому координаты концевой точки стержня вычисляются непосредственно:

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

В исходном примере заданы

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

Следовательно,

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

Эти координаты можно непосредственно сверить с атрибутами выходного `oscillator.svg`. Вертикальная штриховая линия, проходящая через подвес, заканчивается в точке $(x_0, y_0 + r_0)$, а угол между стержнем и вертикалью отображается отдельной дугой. На рисунке он обозначен φ ; для выбранного положения $\varphi = 90^\circ - \alpha = 30^\circ$.

Нельзя безоговорочно подставлять в эти формулы угол, измеренный в математической системе против часовой стрелки от положительной оси x : экранная ось y имеет противоположное направление. Выбор знаков является частью прикладного алгоритма, а не настройкой библиотеки SVG C.

16.3. ЛОКАЛЬНЫЙ БАЗИС И РАЗЛОЖЕНИЕ ВЕРТИКАЛЬНОЙ СИЛЫ

Для точки M определим единичный радиальный вектор, направленный от O к M , и касательный вектор:

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

Оба вектора записаны в экранной системе координат; они взаимно перпендикулярны и имеют единичную длину.

Пусть вертикальная сила, направленная вниз, имеет модуль $G \geq 0$. В экраных координатах её вектор равен $\mathbf{G} = (0, G)$. Проекция на выбранный базис:

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

Именно такую геометрию демонстрирует исходная программа: отрезок, обозначенный G_n , направлен вдоль стержня, а G_τ — по касательной. При $\alpha = 60^\circ$ отношения проекций к модулю силы равны $\sqrt{3}/2$ и $1/2$ соответственно.

Важно различать *величины физических сил* и *экраные длины стрелок*. В коде используется условная графическая шкала $r_3 = 90$: из неё вычисляются $r_{11} = r_3 \cos \alpha$ и $r_{12} = r_3 \sin \alpha$. Это длины построения, а не автоматически определённые значения физических параметров модели. Аналогичным образом векторы B , N и другие элементы представлены условными отрезками с заранее заданными размерами для читаемой схемы.

16.4. НАПРАВЛЕНИЯ СТРЕЛОК И ПРАВИЛА ПОВОРОТА

В исходной программе два вида наконечников: `arrow` и `small_arrow`. Каждый описывается четырьмя контрольными точками относительно локального острия. Точки поворачиваются на угол φ по формулам

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

Затем создаётся контур `path`: прямые стороны соединяются с коротким закруглением основания посредством `__svg_add_short_curveto_cmd()`. Белая заливка скрывает продолжение линии под стрелкой. Этот приём мы уже встречали в главе о наконечниках координатных осей, но здесь он применяется сразу к нескольким наклонным направлениям.

Часть геометрии задаётся сплошными линиями разной толщины, а вспомогательные проекции и большой угловой сектор — тонкими штриховыми линиями. Таким образом стиль позволяет отделить *векторные величины* от *геометрических построений*, не меняя их координат.

16.5. ДУГИ И ЧАСТИЧНЫЕ ОКРУЖНОСТИ

На рисунке имеются две дуги с разными назначениями. Малая дуга около точки подвеса обозначает угол наклона, а большая пунктирная дуга помогает проследить возможное положение конца стержня при неизменной длине r_0 . Обе строятся командой SVG `A`, а не аппроксимацией с помощью большого числа коротких отрезков.

Для малой дуги в программе берётся радиус $r_9 = r_{10} = 60$; начало и конец вычисляются через $\sin \alpha$ и $\cos \alpha$. Большая дуга использует радиус r_0 и две дополнительные угловые позиции. Эллиптические дуги SVG уже разобраны в главе 11; здесь мы видим, как тот же примитив применяется в механической иллюстрации, связанной с математической моделью.

16.6. ПОДПИСИ С МАТЕМАТИЧЕСКИМИ ИНДЕКСАМИ

В графике сил недостаточно вывести буквенные строки `Gtau`, `Gn` и `O1`. Нужны соответствующие математические обозначения G_τ , G_n и O_1 . В исходном C-коде для этого применяются текстовые фрагменты `tspan` с изменёнными `dy` и `font-size`. В результате структурированный SVG хранит отдельно основной символ и знак нижнего индекса, а не рисует их как растр.

При подготовке печатного PDF средствами CairoSVG эти фрагменты сохраняются как части исходных текстовых объектов SVG. Поэтому в итоговом изображении важно проверить размеры индексов, греческие

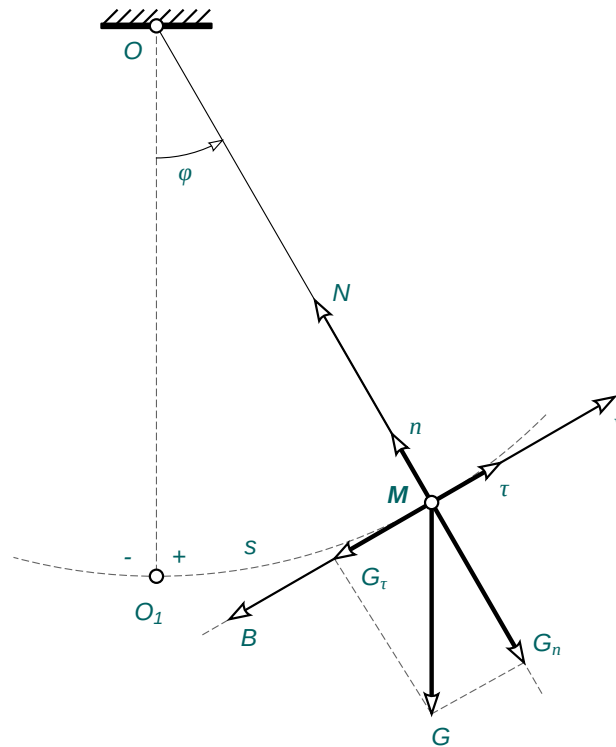


Рис. 16.1. Схема механического осциллятора: точка подвеса, векторная геометрия и проекции силы тяжести.

буквы и расположение подписей; вручную переписывать математические обозначения для печати не требуется.

16.7. ПРИМЕР 15

Исходный код примера находится в каталоге `svgc-examples/015-oscillator`. В нём сохранены геометрия, параметры и подписи оригинального рисунка.

В результате получаются 52 графических объекта: 22 отрезка, 11 контуров, 3 окружности, 15 текстовых узлов и один прямоугольник. В это число не входят сам SVG-документ и его секция определений.

```
#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" );

```

```
стрелкиттvφ
```

```

__svg_free_scene( scene );

return( 0 );
}

```

```

cd svgc-examples/015-oscillator
make

```

```
# open view.html in a browser
```

16.8. ЧТО ДАЛЬШЕ: КООРДИНАТНЫЕ ОСИ И РЕЗУЛЬТАТЫ РАСЧЁТА

В этой главе изображён один заданный геометрический кадр. Для построения временного ряда необходимо отдельно получить решение уравнений движения, определить диапазоны значений, провести оси с делениями и только затем преобразовать численные точки в экранные координаты. Эти задачи станут предметом следующих глав о научных графиках, дискретных наблюдениях и интерполяции.

Сначала математическая модель определяет величины и их связи; затем прикладная программа преобразует их в геометрические координаты, стили и подписи. Библиотека SVGC записывает полученную структуру без вмешательства в физический смысл.

17. Координатные оси, масштаб и подписи делений

17.1. ДВЕ НЕЗАВИСИМЫЕ СИСТЕМЫ КООРДИНАТ

В главе 4 мы научились преобразовывать численные данные в координаты SVG. Но законченная научная иллюстрация требует большего: рамки построения, числовой сетки, подписей делений и самостоятельной политики масштабирования. Даже если данные корректно отображаются на листе, неудачный выбор границ или шага сетки способен скрыть особенности исследуемой зависимости.

Важно различать *математическую область* и *прямоугольник вывода*. Первая задаётся четырьмя действительными числами $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$. Второй задаётся положением левого верхнего угла (L, T) и размерами W, H в координатах SVG. Связь между ними определяется аффинными формулами

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

Вторая формула обращает математическую ориентацию вертикальной оси, поскольку в обычном SVG координата Y возрастает вниз. Алгоритм преобразования принадлежит приложению; SVGC получает уже рассчитанные экранные координаты.

17.2. ДИАПАЗОН ДАННЫХ И ЗАЩИТНЫЕ ПОЛЯ

Начальный диапазон определяется минимумом и максимумом всех конечных значений x_i и y_i . Затем удобно расширить его на небольшую относительную величину:

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

где $\Delta x = x_{\max} - x_{\min}$; аналогично для y . Это не произвольное изменение данных, а выделение свободного пространства между крайними узлами и рамкой изображения. Если $\Delta x = 0$ или $\Delta y = 0$, соответствующий интервал нужно заменить конечным положительным масштабом. В противном случае преобразование координат содержит деление на ноль.

В примере по горизонтали откладываются 41 значение от -2.4 до 6.4 , а по вертикали — значения функции

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

Область изображения определяется из вычисленных значений, а затем её границы округляются наружу: по горизонтали до ближайшей половины единицы, по вертикали — до ближайшей четверти. Для данной выборки получаем

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

Поэтому оси завершаются посередине промежутка между соседними линиями сетки: горизонтальные деления имеют шаг 1, а вертикальные — 0.5. Сам график располагается строго внутри полученных границ.

17.3. ОТКУДА БЕРЁТСЯ ШАГ СЕТКИ

Механическое деление диапазона на десять частей редко даёт читаемые подписи. Например, для интервала длиной 13.7 получится шаг 1.37, а множество подписей окажется заполнено неудобными дробями. Вместо этого выбирают шаг из семейства

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

В программе функция `nice_step()` получает ширину интервала и желаемое приблизительное число промежутков. Пусть $d_0 = (x_{\max} - x_{\min})/N$ и $k = \lfloor \log_{10} d_0 \rfloor$. Тогда $d = d_1 10^k$, где d_1 выбирается из набора 1, 2, 5, 10 как ближайшее *не меньшее* нормализованного шага $d_0/10^k$. Это даёт аккуратную числовую шкалу и ограничивает плотность подписей.

Индексы видимых делений удобно вычислять в целых числах:

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

Такой способ исключает накопление ошибок при многократном прибавлении малого числа с плавающей точкой. Он же позволяет строить обе сетки — горизонтальную и вертикальную — одним и тем же приёмом.

17.4. Порядок наложения объектов

На рисунке сетка имеет малую толщину и светло-серый цвет; оси, проходящие через математический нуль, имеют более тёмный контур; график выделен цветом `##4D1F24`. Порядок рисования следующий: фон, сетка, оси с делениями и наконечниками, затем линия данных. Рамка вокруг сетки не используется: оси проходят через математический нуль. Благодаря этому тонкая сетка не закрывает исследуемую зависимость.

В учебном примере сетка вычисляется циклом, а не копируется из заранее подготовленного SVG. Малые риски делений располагаются непосредственно на осях, числовые подписи — отдельными элементами `text`. В положительных направлениях оси завершаются полыми криволинейными стрелками, построенными по той же схеме, что и последний наконечник на рис. 6.1. Настоящая ширина текста зависит от метрик шрифта и не вычисляется средствами SVG; смещения подписей здесь намеренно подобраны вручную.

Такие смещения не изменяют математическое содержание рисунка, но заметно улучшают его визуальное восприятие. Если подпись перекрывает сам график, ось или соседнее деление, её полезно немного сдвинуть, а второстепенное деление при необходимости даже убрать, чтобы сохранить читаемость рисунка в целом.

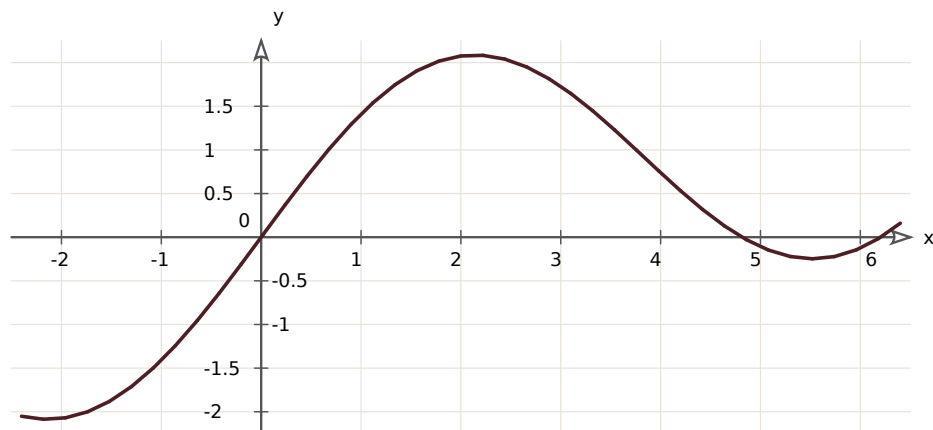


Рис. 17.1. Координатные оси со стрелками, сетка и подписи делений; границы выбраны из диапазона данных.

17.5. ПРИМЕР 16

В исходном файле `svgc-examples/016-axes-grid/main.c` представлены функции `find_bounds()`, `nice_step()`, `screen_x()`, `screen_y()` и `draw_grid()`. Те же координатные функции используются и при рисовании самой исследуемой зависимости. Код не содержит отдельного парсера SVG и не обращается к внутренним структурам библиотеки.

```

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

  Grid steps are chosen by the application, not by SVG C.
  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. ОГРАНИЧЕНИЯ ПРОСТОЙ АВТОМАТИЧЕСКОЙ РАЗМЕТКИ

Семейство шагов 1–2–5 вполне достаточно для вводного численного графика. Для логарифмических осей, двойных шкал, географических координат и очень длинных подписей нужны другие правила делений и верстки. Следует также различать *включение нуля в область* и построение осей $x = 0, y = 0$: математический нуль не обязан находиться внутри рассматриваемого диапазона.

Оси, деления и сетку задаёт алгоритм компоновки на основе численных данных. SVGC записывает полученную геометрию в SVG и не выполняет скрытого выбора масштаба.

18. Дискретные наблюдения и сравнение с моделью

18.1. Почему ряд наблюдений нельзя путать с непрерывной функцией

Численные данные часто представляют собой не непрерывную кривую, а конечный набор отдельных измерений:

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

Само наличие нескольких точек ничего не говорит о поведении системы между ними. Если без комментариев соединить измерения ломаной, читатель может ошибочно принять интерполяцию за действительную эволюцию наблюдаемой величины.

Поэтому в научном графике следует отделять измерения от модели и от способа интерполяции. Дискретный набор можно изображать кружками, столбцами, вертикальными импульсами или специальными маркерами. Связующая линия требует дополнительного объяснения.

18.2. Две формы изображения одних и тех же данных

Пример этой главы показывает два изображения одного набора из десяти измерений. Слева каждое значение y_i выделено отдельной точкой и вертикальным отрезком, проведённым до базовой горизонтали $y = 0$. Справа те же измерения показаны незакрашенными кружками поверх кусочно-линейной модельной зависимости $\hat{y}(t)$.

Модельные значения вычисляются независимо от измерений:

$$\hat{y}_i = 1.6 + 0.7 t_i.$$

В программе и рисунке имеются два отдельных столбца данных: `measured` и `model`. Их нельзя подменять общим списком экранных координат: смысл расхождений станет неразличимым.

На двух панелях используются одинаковые диапазоны $x \in [-0.4, 9.4]$, $y \in [0, 10]$. В противном случае визуальное сопоставление разброса точек оказалось бы некорректным. Стрелки на правых и верхних концах осей обозначают положительные направления. Крайние отметки 9 и 10 не подписываются: цифры на самих кончиках мешали бы чтению графика.

18.3. Величина отклонений от модели

Для оценки расхождения вводятся остатки

$$r_i = y_i - \hat{y}_i.$$

Одна из распространённых скалярных характеристик — среднеквадратичная ошибка (**root mean square error** (корень из средней квадратичной ошибки)):

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} r_i^2}.$$

Это число выражено в единицах измеряемой величины. В отличие от усреднённой знаковой ошибки, взаимно противоположные остатки не компенсируют друг друга.

Для выборки примера RMSE вычисляется непосредственно в C и помещается под правым графиком. Программа не определяет погрешность измерительного прибора и не вычисляет доверительный интервал: для таких выводов нужно дополнительное описание статистической модели.

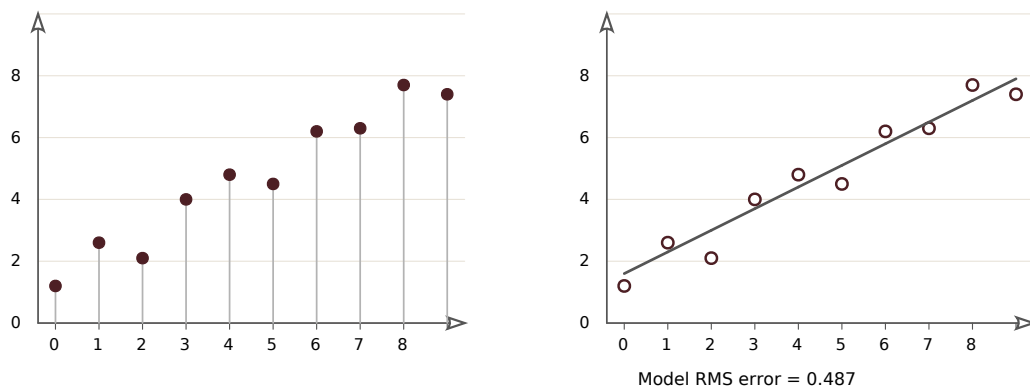


Рис. 18.1. Дискретные наблюдения отдельно (слева) и их сопоставление с модельным рядом (справа).

18.4. ПРИМЕР 17

Каждый элемент исходного массива имеет три компонента: время t , измеренное значение `measured` и модельное значение `model`. Структура `plot` задаёт математическую и экранную области. Обе панели используют одни и те же преобразования координат, но отличаются составом нарисованных объектов.

Измерения формируются отдельными объектами `circle`. Правая модельная линия состоит из отрезков, соединяющих только модельные значения. Подчеркнём: это линия *модели*, а не необоснованная интерполяция между экспериментальными точками.

```

/*****
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)snprintf( 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. ДОСТОВЕРНОСТЬ ВИЗУАЛЬНОГО СРАВНЕНИЯ

При публикации эксперимента следует сохранять смысл осей, показывать размерности, одинаковые пределы при сравнении нескольких панелей и явно различать расчёт и наблюдение. В настоящем учебном рисунке размерности опущены, поскольку данные условны; для реальных измерений это было бы недопустимым упрощением.

Непрерывный модельный график может быть построен с более мелким шагом, чем шаг наблюдений. Однако частота вывода точек сама по себе не доказывает точности модели: она влияет только на способ отображения функции.

Дискретная точка обозначает наблюдение; линия между точками означает отдельное предположение о непрерывной зависимости. Научный рисунок должен делать это различие очевидным.

19. Натуральный кубический интерполяционный сплайн

19.1. ЗАЧЕМ НУЖНЫ УСЛОВИЯ ГЛАДКОСТИ

Ломаная, соединяющая табличные узлы, непрерывна, но её производная обычно терпит скачок в каждом узле. Для представления достаточно гладкой зависимости полезно строить **кубический интерполяционный сплайн** (*cubic interpolating spline* (совместную функцию из кубических многочленов)).

Пусть заданы узлы

$$x_0 < x_1 < \dots < x_{n-1}, \quad y_0, \dots, y_{n-1}.$$

На каждом интервале $[x_i, x_{i+1}]$ сплайн задаётся собственным кубическим многочленом $S_i(x)$. При этом требуются условия

$$S_i(x_i) = y_i, \quad S_i(x_{i+1}) = y_{i+1},$$

а в каждом внутреннем узле

$$S'_{i-1}(x_i) = S'_i(x_i), \quad S''_{i-1}(x_i) = S''_i(x_i).$$

Таким образом кусочные многочлены образуют общую функцию класса C^2 .

19.2. ВЫБОР ГРАНИЧНЫХ УСЛОВИЙ

Одних условий интерполяции и гладкости недостаточно, чтобы определить все коэффициенты. В примере выбран **натуральный сплайн** (*natural spline* (сплайн с нулевыми вторыми производными на концах)):

$$S''(x_0) = S''(x_{n-1}) = 0.$$

Это удобный, но не единственно возможный выбор. Если из физической постановки известны первые производные на концах, следует использовать сплайн с заданными касательными (**clamped spline** (сплайн с краевыми производными)). Граничные условия — это часть математической модели, а не свойство графического формата SVG.

19.3. ТРЁХДИАГОНАЛЬНАЯ СИСТЕМА

Обозначим $h_i = x_{i+1} - x_i > 0$, $m_i = S''(x_i)$. Тогда для каждого внутреннего индекса $i = 1, \dots, n-2$ справедлива система

$$h_{i-1}m_{i-1} + 2(h_{i-1} + h_i)m_i + h_im_{i+1} = 6 \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right).$$

Матрица имеет ненулевые элементы только на главной диагонали и на двух соседних диагоналях. Поэтому обычный плотный метод Гаусса не нужен: применяется **метод прогонки** (*Thomas algorithm*), имеющий линейную стоимость по числу узлов.

В программе массив `upper` хранит коэффициенты обратного хода, `rhs` — преобразованную правую часть. Массив `m` после обратного хода содержит искомые вторые производные в узлах. Условия $m_0 = m_{n-1} = 0$ включены непосредственно в расчёт.

19.4. ФОРМУЛА ВЫЧИСЛЕНИЯ СПЛАЙНА

На интервале $[x_i, x_{i+1}]$ положим $u = x_{i+1} - x$, $v = x - x_i$, $h = h_i$. Тогда

$$S_i(x) = \frac{m_i u^3}{6h} + \frac{m_{i+1} v^3}{6h} + \left(y_i - \frac{m_i h^2}{6}\right) \frac{u}{h} + \left(y_{i+1} - \frac{m_{i+1} h^2}{6}\right) \frac{v}{h}.$$

Формула автоматически воспроизводит заданные значения на концах интервала. Из найденных m_i также следует непрерывность первой и второй производных.

Точки рисования можно выбирать заметно чаще исходных узлов. В данном примере каждый интервал делится на 18 равных отрезков. Это только шаг *вывода кривой*; никаких дополнительных узлов интерполяции при этом не вводится.

19.5. ПРОГРАММНАЯ ПРОВЕРКА

Прежде чем создавать изображение, полезно независимо проверить базовые инварианты:

$$\max_i |S(x_i) - y_i| \leq \varepsilon, \quad m_0 = m_{n-1} = 0.$$

В примере порог для невязки интерполяции составляет 10^{-12} в абсолютных единицах выбранных данных. Если узлы некорректно упорядочены или невязка превышает порог, программа завершает работу с ошибкой.

Проверка не означает, что построенный сплайн совпадает с неизвестной исходной непрерывной функцией между узлами. Она подтверждает корректное решение *заданной задачи интерполяции*. Для осмысленного анализа нужны сведения о природе данных, шуме и допустимой погрешности.

Наряду с интерполяционной кривой на рисунке показаны оси $x = 0$ и $y = 0$, пересекающиеся в начале координат. Они отделены от линий сетки тёмным контуром, снабжены делениями и заканчиваются полыми стрелками, не достигая границы сетки.

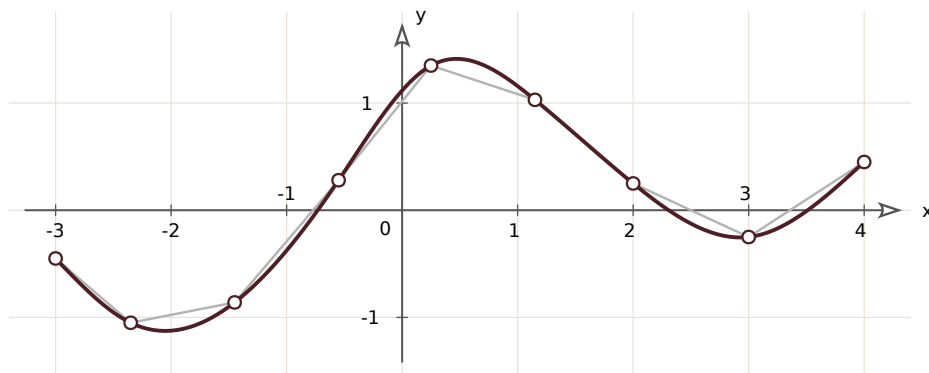


Рис. 19.1. Натуральный кубический сплайн (тёмная кривая), исходные узлы и кусочно-линейная интерполяция (серая ломаная).

19.6. ПРИМЕР 18

Программа 018-cubic-spline содержит три логически отдельные части: вычисление узловых производных `natural_spline()`, оценивание значения сплайна `spline_at()` и преобразование рассчитанных точек в SVG посредством `add_spline()`.

Для основного графика используется многоугольная линия `polyline` с большим числом вершин; это конечное векторное представление вычисленной функции, но сама математическая интерполяция определяется коэффициентами кубических многочленов. Исходные узлы выведены отдельно незакрашенными кружками. Серая ломаная служит для сравнения, а не является частью алгоритма сплайна.

```

/*****
  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)snprintf( 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)snprintf( 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
# программа печатает невязку в узлах и создаёт SVG
# open view.html in a browser
```

19.7. ОГРАНИЧЕНИЯ ИНТЕРПОЛЯЦИИ

Даже кубический сплайн может выходить за пределы значений соседних узлов. Если данные должны оставаться монотонными или неотрицательными, обычная натуральная интерполяция не гарантирует выполнение этих требований. Для таких случаев применяют специальные монотонные или ограниченные интерполянты.

Сплайн вычисляется из данных и граничных условий; SVG служит лишь способом изобразить полученную функцию. Корректный рисунок следует строить после проверки численных условий интерполяции и гладкости.

20. Фазовые траектории и временные ряды

20.1. Одна модель — два вида научного графика

Во многих вычислительных экспериментах важно одновременно видеть, как величина меняется во времени, и как система движется в пространстве состояний. Эти два рисунка отвечают на разные вопросы. **Временной ряд** (*time series*) показывает значения одного или нескольких компонентов состояния при последовательных моментах времени. **Фазовая траектория** (*phase trajectory*) строится из самих компонентов вектора состояния: время определяет порядок точек, но не является координатой фазовой плоскости.

Рассмотрим линейный затухающий осциллятор

$$\ddot{x} + 2\gamma\dot{x} + \omega^2 x = 0, \quad \gamma > 0, \quad \omega > \gamma.$$

После введения скорости $v = \dot{x}$ уравнение второго порядка превращается в систему

$$\frac{d}{dt} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -\omega^2 x - 2\gamma v \end{pmatrix}.$$

Это удобная постановка для явного интегрирования. Переменная x в исходной программе означает координату осциллятора, v — её производную по времени.

20.2. Численная схема Рунге–Кутты

Для состояния $\mathbf{y} = (x, v)^T$ и векторного правого члена $\mathbf{f}(\mathbf{y})$ один шаг классического метода RK4 задаётся четырьмя оценками наклона:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_n), \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_1), \\ \mathbf{k}_3 &= \mathbf{f}(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_2), \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{y}_n + h\mathbf{k}_3), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \end{aligned}$$

Каждая компонента вычисляется независимо при складывании векторов; функция `advance()` создаёт промежуточное состояние, `rhs()` вычисляет правую часть, `rk4_step()` выполняет один полный шаг. Для этого примера $h = 0.04$, $N = 480$, начальное состояние $(x_0, v_0) = (1, 0)$. Общее время моделирования составляет 19.2.

20.3. НЕЗАВИСИМЫЙ АНАЛИТИЧЕСКИЙ КОНТРОЛЬ

Для выбранных начальных условий и недокритического затухания $\gamma < \omega$ точное решение имеет вид

$$\begin{aligned} \beta &= \sqrt{\omega^2 - \gamma^2}, \\ x(t) &= e^{-\gamma t} \left(\cos \beta t + \frac{\gamma}{\beta} \sin \beta t \right), \\ v(t) &= -\frac{\omega^2}{\beta} e^{-\gamma t} \sin \beta t. \end{aligned}$$

В программе для каждого шага вычисляется аналитическое состояние, а затем ошибка в евклидовой норме

$$\varepsilon = \max_{0 \leq n \leq N} \sqrt{(x_n - x(t_n))^2 + (v_n - v(t_n))^2}.$$

Контрольная величина при сборке составляет примерно $6.14 \cdot 10^{-7}$, что меньше заданного предела 10^{-5} . Если предел превышен, программа не записывает SVG.

Проверяется и физический инвариант монотонности энергии

$$E(t) = \frac{1}{2}(v(t)^2 + \omega^2 x(t)^2), \quad \frac{dE}{dt} = -2\gamma v^2 \leq 0.$$

В программе численная энергия может различаться на величину погрешности округления; при проверке последовательности допускается абсолютное отклонение не более 10^{-12} в сторону роста. Это не математическое доказательство устойчивости метода RK4 для произвольного шага, но полезная проверка текущего эксперимента.

20.4. ОТ ВРЕМЕННОГО РЯДА К ФАЗОВОЙ ПЛОСКОСТИ

В верхней части рисунка каждая пара (t_n, x_n) преобразуется в экранные координаты. По горизонтали расположено время, по вертикали — перемещение. В нижней части используются те же численные массивы, но отображается (x_n, v_n) . В результате видна постепенно сжимающаяся спираль, характерная для затухающего колебательного процесса. На фазовой плоскости оси $x = 0$ и $v = 0$ проходят через начало координат, но не достигают границы сетки; их положительные концы отмечены стрелками, выполненными как на рис. 6.1. Это отличает математические оси от прямоугольной рамки диаграммы.

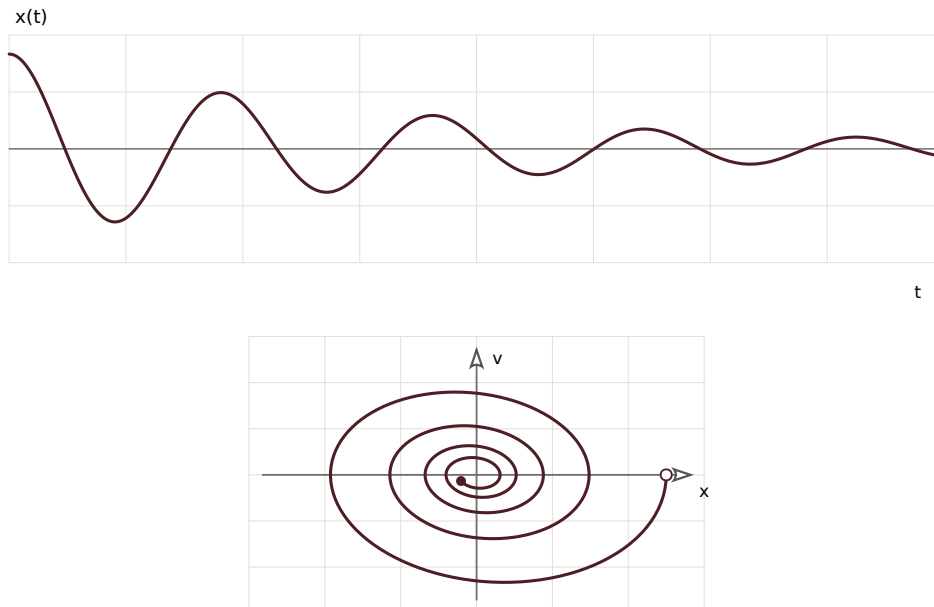


Рис. 20.1. Затухающий осциллятор: временной ряд перемещения и фазовая траектория, вычисленные из одной последовательности состояний RK4.

20.5. ПРИМЕР 19

В программе 019-phase-portrait сначала строится массив $state$ из $N + 1$ элементов. Только после успешной численной проверки создаётся SVG-сцена. Одна функция `add_curve()` строит две разные ломаные: с аргументом `phase=0` по координатам (t, x) , с `phase=1` — по (x, v) .

Для обеих панелей используется одна и та же аффинная формула экранного преобразования, но различные границы прямоугольника и интервалы допустимых данных. Начальная точка фазовой траектории показана полым кружком, конечная — закрашенным.

```

/*****
  SVG C 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
# Проверить сообщение об ошибке RK4 и уменьшении энергии.
# Открыть view.html в браузере.

```

20.6. ЧТО ИМЕННО ИЗОБРАЖАЕТ ЛИНИЯ

Линия в SVG представляет собой конечную ломаную из $N + 1$ точек. Она не является аналитической кривой между узлами вычисления. При уменьшении шага интегрирования одновременно меняются численная траектория и подробность её SVG-представления. Для научного анализа эти два эффекта следует различать: *погрешность интегрирования* и *погрешность геометрического отображения*.

Фазовый портрет и временной ряд должны строиться из одной проверенной траектории. Численный алгоритм определяет состояние системы, а SVGC лишь переводит полученные данные в векторные линии.

21. Контурные линии функций двух переменных

21.1. ОТ СКАЛЯРНОГО ПОЛЯ К МНОЖЕСТВУ УРОВНЯ

Для вещественной функции двух переменных $F(x, y)$ **контурная линия** (*contour line, isoline*) уровня c задаётся не неравенством и не сеткой пикселей, а уравнением

$$F(x, y) = c.$$

Например, множество высоты c на карте рельефа или уровня потенциала в задаче математической физики можно изображать как систему таких линий. В компьютере значения функции обычно известны лишь в узлах конечной сетки. Поэтому контур строится по правилам интерполяции данных между её соседними узлами.

В учебном примере используется положительно определённая квадратичная форма

$$F(x, y) = \frac{x^2}{4} + \frac{y^2}{2} + 0.2xy.$$

Для уровней $c > 0$ её контуры представляют собой вложенные повёрнутые эллипсы. Такая функция удобна для проверки геометрии: уровни 0.35, 0.80, 1.45 и 2.35 не пересекаются и должны следовать друг за другом от центра к периферии.

21.2. ПРЯМОУГОЛЬНАЯ ВЫЧИСЛИТЕЛЬНАЯ СЕТКА

Выберем прямоугольник $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ и целые числа N_x, N_y . Узлы сетки задаются формулами

$$\begin{aligned} x_i &= x_{\min} + i\Delta x, & \Delta x &= \frac{x_{\max} - x_{\min}}{N_x}, \\ y_j &= y_{\min} + j\Delta y, & \Delta y &= \frac{y_{\max} - y_{\min}}{N_y}. \end{aligned}$$

Массив $f[j][i]$ содержит значения функции ровно в $(N_x+1)(N_y+1)$ узлах. В демонстрации используется $N_x = 36, N_y = 32$. Между узлами мы не обращаемся к самой аналитической функции, а вычисляем контур через интерполяцию уже полученных значений.

21.3. АЛГОРИТМ MARCHING SQUARES

Marching Squares (*последовательный обход ячеек квадратной сетки*) рассматривает каждую ячейку отдельно. Четыре её угла классифицируются по условию $F > c$. Получается двоичная маска из четырёх бит. Если значения по концам одного ребра лежат по разные стороны уровня, через ребро проходит контур.

Для координат ребра A, B и значений F_A, F_B точка пересечения линейно интерполируется:

$$\alpha = \frac{c - F_A}{F_B - F_A}, \quad P = A + \alpha(B - A).$$

Здесь $0 \leq \alpha \leq 1$ при пересечении по строгому условию. У алгоритма нет знания о форме исходной аналитической кривой внутри ячейки. Он соединяет точки пересечения отрезками, создавая кусочно-линейное приближение.

21.4. НЕОДНОЗНАЧНЫЕ ЯЧЕЙКИ

Особого внимания требуют комбинации, где диагональные углы имеют одинаковую классификацию, а соседние — противоположную. В таких ячейках возникает четыре пересечения и есть два способа соединить их попарно. Это случаи масок 5 и 10.

В демонстрационном алгоритме используется простой критерий: значение билинейной интерполяции в центре (среднее четырёх узловых значений) определяет соединяемые пары рёбер. Такого решения достаточно для выбранного гладкого поля, но это не *полный asymptotic decider*. Для полей с седловыми точками и строгими топологическими требованиями следует использовать полноценный асимптотический критерий, основанный на седловом значении билинейного интерполянта.

21.5. ИЗОЛИНИИ КАК РЕЗУЛЬТАТ ВЫЧИСЛЕНИЯ

Каждый найденный сегмент превращается в SVG-линии после преобразования математических координат. Для разных уровней используются четыре оттенка нашей книжной палитры. Крупные уровни представлены внешними линиями, меньшие — внутренними.

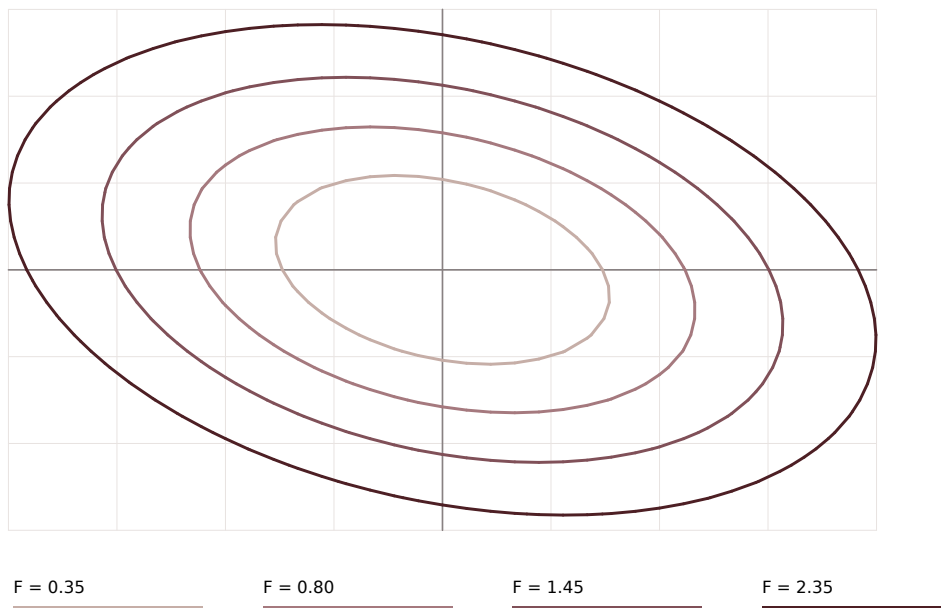


Рис. 21.1. Четыре линии уровня квадратичной формы, вычисленные алгоритмом Marching Squares на прямоугольной сетке.

21.6. ПРИМЕР 20

Программа `020-contours` разделяет вычисление поля `field()`, интерполяцию точки на ребре `crossing()`, обход ячейки `march_cell()` и вывод отдельного сегмента `segment()`. При успешной работе она печатает число полученных отрезков для каждого из четырёх уровней.

```

/*****
  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
# Программа печатает число контурных сегментов каждого уровня.
# Открыть view.html в браузере.

```

21.7. Сглаживание и геометрическая точность

В данном примере сегменты остаются прямолинейными. При уменьшении шага сетки контур визуаль-но становится плавнее, но число создаваемых SVG-объектов растёт. Другой подход: соединять сегменты в упорядоченные полилинии и лишь после этого применять алгоритм сглаживания или аппроксимацию кривыми Безье. Делать это следует осторожно: свободное сглаживание может сдвигать контур относительно заданного уровня $F = c$ и даже менять его топологию. Таким образом *численная достоверность* важнее декоративной гладкости.

Контурная линия должна вычисляться из скалярного поля и выбранного уровня. SVG не должен подменять алгоритм нахождения пересечений, а является только способом записать рассчитанную геометрию.

22. Динамические характеристики метода Гаусса–Лежандра

22.1. От численного метода к функции устойчивости

Структурные и геометрические возможности SVG особенно полезны, когда графическое изображение представляет результат реального исследования. В этой главе мы рассмотрим трёхстадийный **неявный метод Рунге–Кутты** (*implicit Runge–Kutta method*) Гаусса–Лежандра шестого порядка и построим для него область динамической согласованности, а также области локализации нулей и полюсов функции устойчивости. Постановка и вычислительный пример соответствуют материалам <https://dynamics.su/methods/gauss-legendre-33/>.

Для тестового дифференциального уравнения

$$y' = \lambda y, \quad z = h\lambda,$$

шаг численного метода можно выразить как

$$y_{n+1} = R(z)y_n.$$

Здесь h — шаг интегрирования, λ — комплексный параметр тестовой задачи, а $R(z)$ — **функция устойчивости** (*stability function*) метода. Для рассматриваемого трёхстадийного метода она является рациональной функцией:

$$R(z) = \frac{1 + \frac{1}{2}z + \frac{1}{10}z^2 + \frac{1}{120}z^3}{1 - \frac{1}{2}z + \frac{1}{10}z^2 - \frac{1}{120}z^3}.$$

Это диагональная аппроксимация Паде $[3/3]$ функции e^z . Положим

$$P(z) = 1 + \frac{1}{2}z + \frac{1}{10}z^2 + \frac{1}{120}z^3, \quad Q(z) = P(-z).$$

В программе `desc/gl_33_6.c` коэффициенты P и Q заданы массивами `long double`, а значение $R(z)$ вычисляется по схеме Горнера над комплексными числами.

22.2. ГЕОМЕТРИЯ НУЛЕЙ И ПОЛЮСОВ

Нули $R(z)$ — корни числителя $P(z)$, а полюса — корни знаменателя $Q(z)$, если общие множители не сокращаются. Соотношение $Q(z) = P(-z)$ означает, что каждому нулю ζ соответствует полюс $-\zeta$; вещественность коэффициентов обеспечивает также сопряжённые пары корней. Поэтому нули данной функции расположены в левой полуплоскости, а полюса — в правой. Это не произвольное размещение графических значков: оно следует из конкретной функции устойчивости.

Численное вычисление корней третьей степени даёт приближённо

$$\begin{aligned} \text{нули:} & \quad -4.64437, \quad -3.67781 \pm 3.50876i, \\ \text{полюса:} & \quad 4.64437, \quad 3.67781 \pm 3.50876i. \end{aligned}$$

Эти числа приводятся для независимой проверки расположения областей. Сама программа из `rk-dynamics` строит *контуры уровней*, а не вычисляет данные корни и не наносит их условными символами.

Для метода Гаусса–Лежандра $|R(z)| \leq 1$ в левой полуплоскости (А-устойчивость), а на мнимой оси $|R(z)| = 1$. Однако область абсолютной устойчивости не следует смешивать с областью, в которой численное решение *динамически согласовано* с точным.

22.3. ЛОГАРИФИЧЕСКАЯ ДИНАМИЧЕСКАЯ ХАРАКТЕРИСТИКА

Следуя принятому на <https://dynamics.su/> определению, рассмотрим комплексную **динамическую характеристику** (*dynamic characteristic*)

$$D(z) = \ln R(z), \quad z = \mu + i\nu.$$

Для точного экспоненциального решения аналогичная величина равна $D_*(z) = z$. Поэтому

$$D_R(z) = \Re D(z), \quad D_I(z) = \Im D(z)$$

описывают соответственно амплитудную и фазовую составляющие отклонения от идеальной характеристики

$$D_{R*}(z) = \mu, \quad D_{I*}(z) = \nu.$$

В C-программе используется стандартная функция `slogl()`, вычисляющая главное значение комплексного логарифма. Вещественная часть однозначна вдали от нулей и полюсов:

$$D_R(z) = \ln |R(z)|.$$

Мнимая часть зависит от выбора ветви аргумента; это следует учитывать при анализе частотных погрешностей вдали от начала координат.

22.4. ОДНОПРОЦЕНТНАЯ ОБЛАСТЬ ДИНАМИЧЕСКОЙ СОГЛАСОВАННОСТИ

По каждой точке комплексной плоскости определим относительные погрешности вещественной и мнимой характеристик (в процентах):

$$\delta_R(z) = 100 \left| \frac{D_R(z) - \mu}{\mu} \right|, \\ \delta_I(z) = 100 \left| \frac{D_I(z) - \nu}{\nu} \right|.$$

На лучах, где $\mu = 0$ либо $\nu = 0$, эта запись требует специальной обработки предела: непосредственное деление на нулевую координату недопустимо. Исходный пример `delta_contours()` не выполняет вычисление на четырёх координатных лучах и использует соседние отсчёты при формировании замкнутого контура.

Область, внутри которой одновременно выполняются условия

$$\delta_R(z) \leq 1\%, \quad \delta_I(z) \leq 1\%,$$

называется здесь *однопроцентной областью динамической согласованности*. На рисунке её граница показана зелёной линией. Это существенно более строгая характеристика воспроизведения динамики, чем одно условие $|R(z)| \leq 1$.

Для построения границы `delta_contours()` перебирает направления с угловым шагом 1° и вдоль каждого луча увеличивает расстояние от начала координат с шагом 0.025. Последняя точка, в которой обе погрешности не превосходят 1%, попадает в итоговый контур. Полученная линия является *сеточной аппроксимацией*, а не аналитически точной границей; размер шага и поведение около осей необходимо контролировать отдельно.

22.5. ОБЛАСТИ ЛОКАЛИЗАЦИИ НУЛЕЙ И ПОЛЮСОВ

Для построения цветных контуров введём сдвинутый уровень вещественной характеристики

$$F_{\sigma}(z) = D_R(z) + \sigma = \ln |R(z)| + \sigma.$$

Уравнение $F_{\sigma}(z) = 0$ эквивалентно

$$|R(z)| = e^{-\sigma}.$$

При положительном сдвиге $\sigma > 0$ это малый уровень модуля, позволяющий выделять области около нулей. При отрицательном сдвиге $\sigma < 0$ уровень велик и выделяет области около полюсов. На рис. 22.1 контуры малого уровня показаны красным, большого — синим.

Почему замкнутые компоненты этих уровней содержат особенности? В области, свободной от нулей и полюсов, функция $\ln |R(z)|$ гармоническая и не может иметь внутреннего строгого минимума или максимума. Поэтому ограниченная компонента малого уровня, внутри которой модуль меньше граничного значения, должна содержать нуль (при отсутствии других особенностей на границе); для большого уровня аналогичное рассуждение даёт полюс. Это объясняет идею локализации, но численно найденный контур становится математической гарантией только после проверки замкнутости, правильной классификации внутренней области и отсутствия пропущенных особенностей. Проверку числа корней можно дополнить принципом аргумента.

Исходная программа сначала оценивает общий радиус корней числителя и знаменателя `cauchy_radius()`. Грубая оценка Коши в этом примере равна 121, что заведомо больше модулей всех корней. Процедуры `dx_radius()` и `kx_radius()` затем ищут значительно меньший рабочий радиус и сдвиги по лучам. На проверенном запуске получено:

$$r_{\text{work}} = 5.5, \quad \sigma_{\text{zero}} = 4.3, \quad \sigma_{\text{pole}} = -4.3.$$

Эти значения являются результатом описанной численной процедуры, а не заменой строгой оценки Коши.

22.6. MARCHING SQUARES для КОНТУРОВ ФУНКЦИЙ УСТОЙЧИВОСТИ

Для определения замкнутых компонент уровней $F_{\sigma} = 0$ пример применяет алгоритм **Marching Squares** (марширующие квадраты), введённый в § 21.3. На узлах равномерной сетки функция `rdx_sign()` возвращает знак $\Re \ln R(z) + \sigma$, после чего `find_contours()` выбирает пересечённые ячейки и объединяет отрезки в контуры.

Ключевое отличие от главы 21 заключается в природе скалярного поля: теперь оно имеет логарифмические особенности в нулях и полюсах рациональной функции. Поэтому необходимы контроль сеточного шага, проверка замкнутости выделенных компонент и осторожная интерпретация сглаживания. Геометрическая сглаженная кривая не является новым решением уравнения $F_{\sigma} = 0$.

22.7. РИСУНОК 22.1 И ПОЛНЫЙ C-ПРИМЕР

Программа `021-rk-dynamics` переносит предоставленный исследовательский пример без изменения его математических алгоритмов. Главный исходник определяет рациональную функцию устойчивости, логарифмическую характеристику, радиальные процедуры и построение области динамической согласованности. Модули `marching-squares.c` и `svg-paint.c` отвечают за выделение контуров и запись SVG; заголовки `marching-squares.h` и `svg-paint.h` и набор `desc/*.c` сохранены вместе с примером. В данном выпуске активным остаётся `desc/gl_33_6.c`.

Полный текст `main.c`, а также все необходимые модули доступны в каталоге примера и компилируются одним `Makefile`. Он существенно объёмнее предыдущих учебных программ: значительную часть ис-

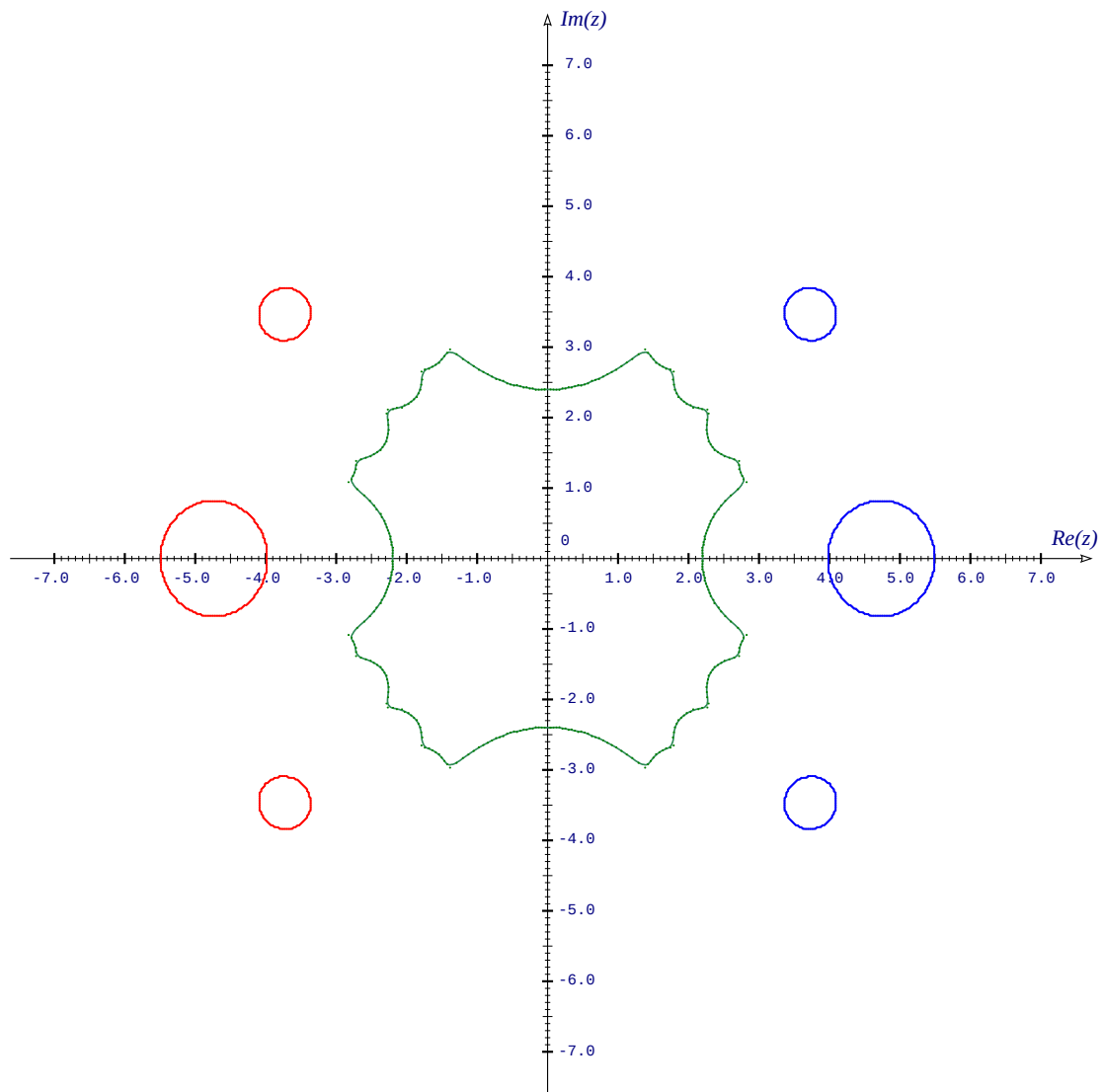


Рис. 22.1. Трёхстадийный метод Гаусса–Лежандра: зелёная граница 1%-й области динамической согласованности, красные области локализации нулей и синие области локализации полюсов функции устойчивости.

ходника составляют процедуры радиальной локализации и обхода контуров. Здесь напомним ключевые вычисления:

```

/* 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:
*/
int rdx_sign( long double complex z, long double shift )
{
    return SIGN(creal(clogl(polynom(z))) + shift);
}
```

Остальные функции и их взаимодействие описаны выше; для воспроизведения полного алгоритма следует собирать исходники из каталога 021-rk-dynamics, не сокращённый фрагмент книги.

```
cd svgc-examples/021-rk-dynamics
make
# Открыть dynamic-consistency-area.svg или view.html
```

SVG строится только посредством публичного API SVGC. Печатное векторное PDF-представление получается из этого SVG средствами CairoSVG. При преобразовании сохраняются координаты, пути и цвета контуров; научные вычисления в TeX не дублируются. Для просмотра изображения и повторного эксперимента не требуются специальные шрифты.

Содержательная научная иллюстрация начинается с математической постановки и вычисления данных. В рассматриваемом примере положение областей определяется рациональной функцией устойчивости метода Гаусса–Лежандра и алгоритмами построения контуров. SVGC отвечает за графическое представление результата, а не за свойства численного метода.

23. Преобразование SVG в PDF и включение рисунка в TeX

23.1. ПРЕОБРАЗОВАНИЕ SVG СРЕДСТВАМИ CAIROSVG

SVG-документ можно преобразовать в векторный PDF с помощью CairoSVG:

```
cairosvg -o figure.pdf figure.svg
```

При таком преобразовании линии, кривые, заливки и другие графические объекты сохраняются в векторном представлении. Растеризация и выбор разрешения PNG для печатной страницы не требуются.

Для корректного оформления текстовых элементов необходимые шрифты должны быть доступны при выполнении CairoSVG. Если в SVG используются внешние ресурсы или сложные определения CSS, результат преобразования зависит от их доступности и поддержки со стороны конвертера.

23.2. ВКЛЮЧЕНИЕ PDF В ДОКУМЕНТ TEX

В преамбуле документа подключается пакет `graphicx`:

```
\usepackage{graphicx}
```

Полученный PDF включается командой `\includegraphics` :

```
\begin{figure}[htbp]
\centering
\includegraphics[width=0.85\linewidth]{figure.pdf}
\caption{Научная иллюстрация.}
\end{figure}
```

Параметр `width` задаёт ширину рисунка относительно текстовой полосы. Высота меняется пропорционально; геометрия и толщина линий масштабируются вместе со всем изображением. Исходные графические элементы при этом остаются векторными.

24. Справочник публичных интерфейсов SVGC

24.1. ОБЛАСТЬ ПРИМЕНЕНИЯ СПРАВОЧНИКА

В этой главе собраны основные операции библиотеки SVGC непосредственно по её публичному заголовочному файлу `svgc.h`. Разбиение выполнено по задачам, с которыми сталкивается автор научного рисунка: создание сцены, геометрические примитивы, пути, стили, текст, определения и запись SVG. Полные объявления структур и все вспомогательные функции доступны в установленном заголовке; здесь объясняются роли групп интерфейсов и типичные сочетания вызовов.

24.2. ЛИСТ, СЦЕНА И ВЛАДЕНИЕ РЕСУРСАМИ

Базовая последовательность состоит из листа, содержащего размеры и определения, и сцены, содержащей объекты изображения.

```
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 );
```

В существующих программах лист создаётся первым, затем передаётся в конструктор сцены. После завершения работы обычно достаточно освободить сцену: её список узлов и принадлежащие ему определения освобождаются в соответствии с действующей реализацией SVGC. При ошибке до передачи листа сцене приложение должно самостоятельно очистить уже созданный объект. При ошибке добавления узла следует проверить, кому принадлежит не добавленный объект; для обычных примитивов в примерах он освобождается явно.

Нельзя путать `__svg_node` — универсальное описание графического объекта — с `__svg_scene`, представляющей упорядоченный список. Структуры опубликованы в заголовке, однако безопаснее работать с ними через предусмотренные конструкторы, функции оформления и удаления.

24.3. ГЕОМЕТРИЧЕСКИЕ ПРИМИТИВЫ

Для примитивных объектов существуют отдельные конструкторы:

```
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 );

```

Аргументы задаются в координатах SVG. Если r_x и r_y прямоугольника равны нулю, его углы не скругляются. Параметры r , r_x , r_y задают радиусы, а не диаметры. Численные ограничения геометрии и проверка физических единиц относятся к вызывающей программе. Необходимо различать математические координаты модели и координаты готового SVG.

24.4. МАССИВЫ ВЕРШИН И ЗАМКНУТЫЕ КОНТУРЫ

Для многоугольников и ломаных библиотека предоставляет объект списка точек:

```

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 );

```

`polyline` описывает открытую последовательность отрезков; `polygon` замыкает её и может иметь заливку. Список точек предварительно заполняется через `__svg_add_point()`. После передачи списка конструктору его нельзя без проверки владения уничтожить как независимый объект: эти ресурсы используются созданным узлом. Обработку неудачного выделения памяти нужно проводить до передачи ресурсов и после каждого вызова.

24.5. Путь и КОМАНДЫ ГЕОМЕТРИИ

Универсальный путь хранит упорядоченную последовательность SVG-команд. Конструкторы и управляющие функции представлены следующей группой:

```

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 );

```

Для абсолютных команд применяется макрос ABSOLUTE, для относительных — RELATIVE. Относительная координата интерпретируется относительно текущей точки пути, а не относительно начала листа. Первая команда нового связного контура обычно задаёт moveto.

Для кривых и дуг доступны следующие операции:

```

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 );

```

Кубическая кривая Безье задаётся двумя управляющими точками и конечной точкой; квадратичная — одной управляющей и конечной точкой. Короткие формы используют условия гладкости и отражение предыдущей управляющей точки по правилам SVG. Поэтому их нельзя использовать как обычные сокращения с пропущенными обязательными координатами.

Эллиптическая дуга использует два радиуса, угол поворота эллипса, признаки выбора большой дуги и направления, а также конечную точку. Ошибочно считать, что large_arc_flag и sweep_flag обозначают сами углы: это логические флаги выбора одного из возможных геометрических решений.

24.6. КОНТУР, ЗАЛИВКА И ТЕКСТ

Объект получает оформление после создания:

```

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 );

```

Значения `stroke` и `fill` могут задаваться в виде CSS-цветов или ссылок на определения. Значение `none` отключает соответствующее изображение контура или заливки. Стиль окончаний линии выбирается из `LINECAP_BUTT`, `LINECAP_ROUND`, `LINECAP_SQUARE`; стиль соединений — из `LINEJOIN_MITER`, `LINEJOIN_ROUND`, `LINEJOIN_BEVEL`. Толщина линии и элементы пунктира должны проверяться после изменения масштаба рисунка.

Для текста предназначены функции:

```

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 );

```

В SVG координата `y` описывает положение базовой линии текста, а не автоматически вертикальный центр надписи. Для математических индексов в авторских примерах применяются вложенные SVG-элементы `tspan`. Их вывод и экранирование содержимого нужно проверять как XML, особенно если строка формируется из внешних данных.

24.7. ИЗОБРАЖЕНИЯ И ВНЕШНИЕ ССЫЛКИ

Для включения внешнего изображения предусмотрена функция:

```

struct __svg_node *__svg_new_image(
    double x, double y, int width, int height,
    char *href, char *title );

```

В ней `href` обозначает ссылку на ресурс, а не сам файл изображения в памяти. Успех создания SVG не

гарантирует, что этот ресурс будет доступен при переносе рисунка в другую директорию. Поэтому внешние изображения должны входить в контроль воспроизводимости.

24.8. ГРАДИЕНТЫ И ЦВЕТОВЫЕ ОСТАНОВКИ

В SVG градиент — определение, на которое ссылается заливка примитива. Соответствующие конструкторы:

```
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 );
```

Линейный градиент определяется направлением от первой точки ко второй. Радиальный — центром, радиусом и фокусом. Параметр `units` выбирает систему координат, а `spread` — поведение вне основного отрезка или радиальной области градиента. Для цветов и непрозрачности остановки используются:

```
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 );
```

Нужно различать определение градиента и конкретное применение к прямоугольнику или другому примитиву. Один градиент может использоваться несколькими объектами.

24.9. ОПРЕДЕЛЕНИЯ ЛИСТА, CSS И JAVASCRIPT

Определения добавляются не в обычный список графических объектов сцены, а в соответствующий список листа:

```
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 );
```

Функции чтения файлов CSS и JavaScript принимают *только содержимое* соответствующего языка: исход-

ный файл не должен повторять `defs`, `style`, `script` и `CDATA`. Библиотека сама добавляет XML-обрамление и помещает содержимое в определения.

Для обеих файловых функций договорённость о результатах одинакова: 1 означает успех, 0 — ошибку чтения, кодирования или памяти, -1 — недопустимые аргументы. Текст файла читается при вызове, а не откладывается до записи SVG. Пользовательская программа обязана проверять результат. Эти функции не исполняют JavaScript и не загружают внешние шрифты из CSS.

24.10. Идентификаторы, ссылки и вывод

Для доступа к идентификатору узла и построения SVG-ссылки предусмотрены:

```
const char *__svg_get_node_id(  
    struct __svg_node *node );  
char *__svg_create_node_url(  
    struct __svg_node *node );
```

Первая функция возвращает указатель на идентификатор, принадлежащий узлу. Вторая создаёт отдельную строку для использования в стиле вида `url(##id)`; в примерах книги эта строка освобождается приложением после передачи её в `__svg_set_fill()`.

Окончательная запись сцены выполняется через

```
void __svg_paint_scene(  
    struct __svg_scene *scene, int group, char *fname );
```

Обрати внимание на тип результата `void`: функция не предоставляет вызывающему коду отдельного статуса успеха записи. Поэтому внешняя проверка существования и корректности полученного SVG особенно важна для автоматических сценариев сборки. В примерах книги это выполняется отдельной проверкой XML после запуска программы.

24.11. Глобальные стили и границы их использования

В заголовке доступны функции вида `__svg_set_scene_stroke_width()`, `__svg_set_scene_fill_rgb()`, `__svg_set_scene_text_font_family()` и `__svg_set_scene_default_style()`. Они устанавливают общие значения оформления, которые используются при создании последующих объектов. Это удобно для однородного чертежа, однако автору нужно понимать зависимость результата от порядка операций. Если два примера используют разные общие настройки, следует явно установить исходное состояние, а не полагаться на результаты предыдущего рисунка.

Публичный заголовок также содержит функции освобождения объектов стиля и вспомогательные конструкторы команд. Они необходимы для случаев, когда приложение вручную управляет жизненным циклом промежуточных объектов. Однако основная учебная последовательность остаётся простой: создать лист, создать сцену, построить геометрию, добавить объекты, записать SVG, освободить сцену.

Заключение

В этой книге мы прошли путь от простейших геометрических объектов до программного построения научных рисунков, для которых геометрию определяют исходные данные и результаты вычислений. Линии, многоугольники, кривые, текст, координатные оси и диаграммы образуют общий графический язык, на котором можно выразить структуру алгоритма или результаты численного эксперимента.

Все рассмотренные изображения создаются программами на языке C. Прикладной алгоритм отвечает за вычисления, выбор координат и расположение элементов, а библиотека SVGC записывает полученную геометрию в SVG. Благодаря этому исходный рисунок остаётся доступным для просмотра, повторного построения и изменения непосредственно из программы.

Для публикации SVG можно преобразовать в векторный PDF и поместить в документ TeX. Так вычисление, графическое представление и печатная страница оказываются частями одного последовательного процесса: от численной задачи до готовой научной иллюстрации.

A. Команды SVG path и соответствие API SVGC

A.1. ПРАВИЛА ЗАПИСИ ПУТИ

Атрибут `d` элемента `path` представляет последовательность команд и числовых аргументов. Большая буква обозначает абсолютные координаты, малая — относительные. Относительные приращения отсчитываются от текущей точки пути, а не от верхнего левого угла листа. После команды `M/m` начинается новый подпуть; при `Z/z` текущий подпуть замыкается к его началу.

В публичном C API для переключения режима используются `ABSOLUTE` и `RELATIVE`. Приложение создаёт `__svg_cmds`, добавляет команды и передаёт список в `__svg_new_path()`. Нельзя освобождать переданный список независимо от созданного объекта.

A.2. ОСНОВНЫЕ КОМАНДЫ И ЧИСЛО ПАРАМЕТРОВ

Команда	Аргументы	Значение
<code>M/m</code>	2	Перенос текущей точки к (x, y) ; начало подпути.
<code>L/l</code>	2	Прямой отрезок от текущей точки к (x, y) .
<code>H/h</code>	1	Горизонтальный отрезок до заданной координаты x .
<code>V/v</code>	1	Вертикальный отрезок до заданной координаты y .
<code>C/c</code>	6	Кубическая кривая Безье: две управляющие и конечная точки.
<code>S/s</code>	4	Продолжение кубической кривой с отражённой первой управляющей точкой при допустимом предшественнике.
<code>Q/q</code>	4	Квадратичная кривая Безье: управляющая и конечная точки.
<code>T/t</code>	2	Продолжение квадратичной кривой с отражённой управляющей точкой при допустимом предшественнике.
<code>A/a</code>	7	Эллиптическая дуга: радиусы, поворот, два флага и конечная точка.
<code>Z/z</code>	0	Замыкание текущего подпути.

После `M` дополнительные пары чисел без новой буквы трактуются как `L`; после `m` — как `l`. Многие команды допускают повторяющиеся группы параметров. Поэтому простой поиск по буквам в атрибуте `d` не заменяет полноценный синтаксический разбор SVG.

A.3. ФОРМУЛЫ КРИВЫХ БЕЗЬЕ

Для кубической кривой с контрольными точками P_0, P_1, P_2, P_3 используется параметр $0 \leq t \leq 1$:

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

Для квадратичной кривой с P_0, P_1, P_2 :

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

Эти формулы применяются к обеим координатам. В начале и конце кубической кривой касательные пропорциональны $P_1 - P_0$ и $P_3 - P_2$.

Команда S отражает предыдущую вторую управляющую точку относительно текущей точки только после C или S; иначе новая первая управляющая точка совпадает с текущей. Аналогично T использует отражение после Q или T; в остальных случаях её управляющая точка совпадает с текущей. Это существенная часть геометрии, а не только сокращение числа записываемых аргументов.

A.4. АРГУМЕНТЫ ЭЛЛИПТИЧЕСКОЙ ДУГИ

Команда A получает аргументы в следующем порядке:

```
A rx ry x-axis-rotation large-arc-flag sweep-flag x y
```

Радиусы r_x, r_y задают размеры эллипса, а *x-axis-rotation* — его поворот в градусах. *large-arc-flag* выбирает большую или малую дугу; *sweep-flag* — направление обхода в системе координат SVG. Оба флага принимают значения 0 и 1, но не являются углами в радианах. При нулевом радиусе соответствующая дуга вырождается в отрезок; в случае недостаточных ненулевых радиусов SVG предусматривает их коррекцию. Совпадение начальной и конечной точек не задаёт полного эллипса автоматически.

A.5. ФУНКЦИИ ПОСТРОЕНИЯ В C

Соответствие наиболее употребительных команд публичному API:

```
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()
```

Каждая функция добавляет команду в список. Сами *moveto*, *lineto* и кривые не создают самостоятельных SVG-элементов: общий path создаётся после того, как приложение определило всю нужную геометрию.

```
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 );
    /* Создание path и добавление его к сцене --- следующий шаг. */
}
```

Полная обработка ошибок добавления команд и владения памятью показана в программах 008-path, 009-bezier и 010-arcs. Приведённый фрагмент иллюстрирует лишь порядок операций, не заменяет закон-

ченный пример.

Геометрический смысл каждого сегмента определяется командой и её параметрами. Одно только визуальное сходство двух путей не доказывает эквивалентности исходных данных и правил построения.

В. Формулы координатных преобразований и масштаба

В.1. ЭКРАННЫЙ ПРЯМОУГОЛЬНИК И ОБЛАСТЬ ДАННЫХ

Пусть данные заданы в прямоугольнике $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, а область рисования SVG имеет левый верхний угол (L, T) , ширину W и высоту H . Предполагается $x_{\max} > x_{\min}$, $y_{\max} > y_{\min}$, $W > 0$ и $H > 0$.

Математическая вертикальная координата возрастает вверх; в обычном SVG экранная координата Y возрастает вниз. Поэтому отображение $(x, y) \mapsto (X, Y)$ имеет вид

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

Обозначим

$$s_x = \frac{W}{x_{\max} - x_{\min}}, \quad s_y = \frac{H}{y_{\max} - y_{\min}}.$$

Тогда удобно вычислять координаты в форме

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

Эта запись исключает повторное деление на длину диапазона при обработке большого массива точек.

В.2. ОБРАТНОЕ ПРЕОБРАЗОВАНИЕ

По экранным координатам можно получить математические:

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

Обратное преобразование полезно при интерактивном просмотре SVG: положение указателя в изображении можно интерпретировать как приблизительную точку математической модели. При этом необходимо учитывать возможный дополнительный масштаб просмотрщика или CSS, если координаты указателя получены в координатах окна, а не внутренней системы SVG.

В.3. ВЕКТОРЫ, ДЛИНЫ И НЕОДИНАКОВЫЕ МАСШТАБЫ

Для приращения $\Delta \mathbf{r} = (\Delta x, \Delta y)$ экранный вектор равен

$$\Delta X = s_x \Delta x, \quad \Delta Y = -s_y \Delta y.$$

В общем случае $s_x \neq s_y$, поэтому равные математические длины могут иметь разные экранные размеры. Прямой угол между произвольными векторами при анизотропном масштабе тоже не обязательно сохраняется. Программа должна учитывать это, если строит окружности, стрелки фиксированного направления, касательные или нормали к кривой.

Для одинакового масштаба выбирают $s_x = s_y = s$. Если всю заданную математическую область нужно поместить в экранный прямоугольник без обрезания, можно взять

$$s = \min\left(\frac{W}{x_{\max} - x_{\min}}, \frac{H}{y_{\max} - y_{\min}}\right).$$

Оставшееся место следует распределить явно — например, поровну по обе стороны каждого направления. При этом меняются поля изображения, а не координаты исходных данных.

В.4. ПОДБОР ДИАПАЗОНА И ЗАЩИТНОГО ПОЛЯ

Если крайние значения найдены по массиву x_i, y_i , сначала вычисляют

$$\Delta x = x_{\max} - x_{\min}, \quad \Delta y = y_{\max} - y_{\min}.$$

Затем границы можно расширить на долю δ :

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

Аналогично расширяется вертикальный интервал. При вырожденном диапазоне, когда минимум равен максимуму, сначала необходимо задать положительную ширину вручную; подстановка $\Delta x = 0$ непосредственно в формулы недопустима. Значения NaN и бесконечности также нельзя использовать как границы диапазона.

В.5. ПОДПИСИ ДЕЛЕНИЙ И СЕМЕЙСТВО 1–2–5

В главе 17 описана практическая политика выбора шага числовой сетки. Для длины диапазона $D > 0$ и желаемого числа промежутков $N > 0$ сначала получают

$$d_0 = \frac{D}{N}, \quad k = \lfloor \log_{10} d_0 \rfloor.$$

Нормализованный шаг $d_0/10^k$ округляют вверх до ближайшего элемента из множества $\{1, 2, 5, 10\}$, затем умножают на 10^k .

Вместо накопления погрешности при повторном прибавлении d рекомендуется вычислять целочисленные индексы:

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

Вторая координата рассчитывается тем же способом. Положение каждой подписи после этого вычисляется тем же аффинным преобразованием, что и положение самого деления.

В.6. ПРИМЕР ДЛЯ НЕЗАВИСИМОЙ ПРОВЕРКИ

Пусть математическая область есть $x \in [-2, 8]$, $y \in [-1, 3]$, а экраный прямоугольник задан значениями $L = 40$, $T = 30$, $W = 500$, $H = 200$. Тогда $s_x = s_y = 50$. Начало координат отображается в

$$X(0) = 40 + 50 \cdot 2 = 140, \quad Y(0) = 30 + 50 \cdot 3 = 180.$$

Точка $(4, 2)$ переходит в $(340, 80)$. Подстановка этих значений в обратные формулы возвращает ровно $(4, 2)$ в идеальной вещественной арифметике.

Данные модели и координаты листа следует хранить отдельно. Координатное преобразование принадлежит прикладной программе, а SVGC получает готовые координаты графических объектов.

С. Русско-английский словарь терминов

С.1. КАК ПОЛЬЗОВАТЬСЯ СЛОВАРЁМ

Термины приведены в том смысле, в каком они употребляются в этой книге. Английские выражения сопровождаются русскими эквивалентами и краткими пояснениями. Различаются язык разметки SVG, библиотечный интерфейс C, математический алгоритм и средства печатной подготовки. Словарь не заменяет формальные определения SVG и не расширяет API SVGC.

English term	Русское название и пояснение
Affine transform	Аффинное преобразование; линейная часть и сдвиг координат.
Algorithm	Алгоритм; последовательность вычислений и построения геометрии.
Antialiasing	Сглаживание растровых границ при отображении.
Arc	Дуга; часть окружности или эллипса.
Axis	Координатная ось.
Baseline	Базовая линия текста, относительно которой располагаются глифы.
Bounding box	Ограничивающий прямоугольник геометрического объекта.
Browser	Браузер; средство отображения и проверки SVG.
CData section	Секция CDATA в XML; текст без обычной обработки разметки.
Clip / clipping path	Отсечение; ограничение видимой части рисунка заданной областью.
Closed contour	Замкнутый контур.
Control point	Управляющая точка кривой Безье.
Coordinate mapping	Преобразование координат данных в координаты вывода.
Cubic Bezier curve	Кубическая кривая Безье с двумя управляющими точками.
Dash array	Шаблон чередования штрихов и промежутков.
Data range	Числовой диапазон данных.
Definition	Определение в SVG, часто расположенное в элементе defs.
Display scale	Масштаб отображения.
Endpoint	Конечная точка отрезка или кривой.
Fill	Заливка внутренней области графического объекта.
Font family	Гарнитура шрифта, семейство начертаний.
Font metrics	Метрики шрифта; величины, определяющие размеры текста.
Glyph	Глиф; графическая форма символа шрифта.
Gradient	Градиент; непрерывное изменение цвета в области заливки.
Interpolation	Интерполяция; восстановление функции по узлам.
Kerning	Кернинг; корректировка межбуквенных расстояний.
Line cap	Вид окончания линии: butt, round, square.
Line join	Вид соединения сегментов: miter, round, bevel.
Marching Squares	Метод построения контуров на прямоугольной сетке.

Natural cubic spline	Натуральный кубический сплайн с нулевыми вторыми производными на краях.
Path	Составной SVG-контур из отрезков и кривых.
Phase portrait	Фазовый портрет; траектория в пространстве состояний.
Polyline	Ломаная линия.
Polygon	Многоугольник.
Quadratic Bezier curve	Квадратичная кривая Безье с одной управляющей точкой.
Radial gradient	Радиальный градиент.
Rasterization	Растеризация векторной геометрии.
Regression test	Регрессионная проверка; сравнение с ранее принятой версией.
Reproducibility	Воспроизводимость; возможность повторить результат из исходных данных и инструментов.
Root / zero	Корень, или нуль, функции.
Scene	Сцена; упорядоченный набор графических объектов.
Stroke	Контур или обводка, включая цвет и толщину линии.
Style sheet	Таблица стилей; правила CSS, управляющие оформлением.
SVG	Масштабируемая векторная графика (Scalable Vector Graphics).
Symbolic label	Математическая подпись с индексами и обозначениями.
CairoSVG	Средство преобразования SVG в векторный PDF.
Time series	Временной ряд наблюдений или расчётных значений.
Tridiagonal solver	Метод решения трёхдиагональной системы, в примерах — метод прогонки.
Viewport	Видимая область SVG-документа.
Visual regression	Визуальная регрессия; сравнение изображений, полученных до и после изменения.
XML	Расширяемый язык разметки (Extensible Markup Language).

С.2. ТЕРМИНЫ, КОТОРЫЕ НЕЛЬЗЯ СМЕШИВАТЬ

Ломаная (*polyline*) — последовательность отрезков, не обязательно замкнутая. **Многоугольник** (*polygon*) — замкнутая геометрическая область, способная иметь заливку. **Путь** (*path*) является более общей конструкцией; его сегментами могут быть и прямые, и кривые.

Метрика шрифта (*font metric*) не равна числу символов строки. Выравнивание текста по фактической ширине требует учитывать глифы, пробелы и кернинг. SVG C намеренно не вычисляет эту ширину вместо прикладной программы.

Векторный эталон (*SVG source*) и **печатное представление** (*PDF rendering*) — два связанных, но разных объекта проверки. Их совпадение по смыслу требует анализа содержания, а не только валидности XML или отсутствия ошибок XeLaTeX.

D. Указатель C-программ, глав и рисунков

D.1. КАК УСТРОЕНА НУМЕРАЦИЯ

В названиях каталогов 001-first-sheet и далее трёхзначный номер задаёт последовательность учебных C-примеров. Номер главы начинается с единицы и не совпадает с номером C-примера: первая глава посвящена установке и не содержит отдельного рисунка.

Каждый пример создаёт свой SVG и сопровождается файлом view.html. В PDF приводится соответствующее печатное векторное PDF-представление. Номер рисунка определяется номером главы и локальной последовательностью иллюстраций. В настоящем выпуске у каждой главы с демонстрационной C-программой ровно одна основная иллюстрация.

D.2. СООТВЕТСТВИЕ ИСХОДНЫХ ПРОГРАММ И ИЛЛЮСТРАЦИЙ

Глава	Каталог примера	Назначение, рисунок
2	001-first-sheet	Лист и первая сцена, рис. 2.1.
3	002-primitives	Базовые геометрические примитивы, рис. 3.1.
4	003-data-coordinates	Координаты по данным, рис. 4.1.
5	004-polylines	Ломаные и многоугольники, рис. 5.1.
6	005-arrows	Формы стрелок, рис. 6.1.
7	006-flowchart	Алгоритмическая блок-схема, рис. 7.1.
8	007-styles	Контур, заливка и пунктир, рис. 8.1.
9	008-path	Команды составного пути, рис. 9.1.
10	009-bezier	Кривые Безье, рис. 10.1.
11	010-arcs	Дуги, хорды и секторы, рис. 11.1.
12	011-text	Текст и базовая линия, рис. 12.1.
13	012-gradients	Градиенты, рис. 13.1.
14	013-control-system	Гироскопическая схема, рис. 14.1.
15	014-rk4-scheme	Структура шага RK4, рис. 15.1.
16	015-oscillator	Авторский механический осциллятор, рис. 16.1.
17	016-axes-grid	Оси, деления и сетка, рис. 17.1.
18	017-discrete	Наблюдения и модель, рис. 18.1.
19	018-cubic-spline	Кубическая интерполяция, рис. 19.1.
20	019-phase-portrait	Фазовый портрет и временной ряд, рис. 20.1.
21	020-contours	Линии уровня, рис. 21.1.
22	021-rk-dynamics	Динамические характеристики метода Гаусса–Лежандра, рис. 22.1.

D.3. КАК НАЙТИ ПРОГРАММУ ПО ЗАДАЧЕ

Если нужна простая геометрия, следует начать с 002-primitives, 004-polylines и 008-path. Для математических обозначений полезны 011-text, а также авторские схемы 013-control-system и 015-oscillator.

Численные программы удобно изучать по нарастающей сложности: 016-axes-grid, 017-discrete, 018-cubic-spline, 019-phase-portrait, 020-contours, 021-rk-dynamics. Последний пример развивает исследование функции устойчивости метода Гаусса–Лежандра и использует Marching Squares для построения уровней логарифмической характеристики.