Программирование в Scilab

Микаэль Боден (Michaël Baudin)

Сентябрь 2011 года

Аннотация

В этом документе мы представляем программирование в Scilab[1]. В первой части мы представляем управление памятью в Scilab. Во второй части мы представляем различные типы данных и анализируем методы программирования, связанные с теми структурами данных. В третьей части мы представляем характеристики для разработки гибких и устойчивых функций. В последней части мы представляем методы, которые позволяют получить наилучшие характеристики, основанные на вызовах высокооптимизированных числовых библиотек. Предоставляется много примеров, которые позволят увидеть эффективность методов, которые мы представляем.

[1]Прим. перев. — произносится «сайлаб»

Содержание

1 Введение 6

2 Управление переменными и памятью 6
- 2.1 Стек . . . 6
- 2.2 Подробнее об управлении памятью . . . 9
  - 2.2.1 Ограничения памяти со стороны Scilab . . . 9
  - 2.2.2 Ограничения памяти в 32^x- и 64^x-битных системах . . . 11
  - 2.2.3 Алгоритм в `stacksize` . . . 12
- 2.3 Список переменных и функция `who` . . . 12
- 2.4 Переносимость переменных и функций . . . 13
- 2.5 Уничтожение переменных : `clear` . . . 16
- 2.6 Функции `type` и `typeof` . . . 17
- 2.7 Заметки и ссылки . . . 18
- 2.8 Упражнения . . . 18

3 Специальные типы данных 19
- 3.1 Строки . . . 19
- 3.2 Многочлены . . . 22
- 3.3 Гиперматрицы . . . 26
- 3.4 Типы и размерности выделенной гиперматрицы . . . 28
- 3.5 Тип данных `list` (список) . . . 31
- 3.6 Тип данных `tlist` . . . 33
- 3.7 Имитация объектно-ориентированного программирования с помощью типизированных списков . . . 37
  - 3.7.1 Ограничения позиционных аргументов . . . 37
  - 3.7.2 Класс «person» в Scilab . . . 38
  - 3.7.3 Расширение класса . . . 41
- 3.8 Перегрузка типизированных списков . . . 42
- 3.9 Тип данных `mlist` . . . 44
- 3.10 Тип данных `struct` . . . 45
- 3.11 Массив структур `struct` . . . 46
- 3.12 Тип данных `cell` . . . 48
- 3.13 Сравнение типов данных . . . 50
- 3.14 Заметки и ссылки . . . 51
- 3.15 Упражнения . . . 52

4 Управление функциями 53
- 4.1 Продвинутое управление функциями . . . 54
  - 4.1.1 Как сделать запрос о функции . . . 54
  - 4.1.2 Функции не зарезервированы . . . 56
  - 4.1.3 Функции — это переменные . . . 57
  - 4.1.4 Функции обратного вызова . . . 58
- 4.2 Разработка гибких функций . . . 59
  - 4.2.1 Обзор функции `argn` . . . 60
  - 4.2.2 Практические вопросы . . . 61
  - 4.2.3 Использование переменных аргументов на практике . . . 64

4.2.4 Значения по умолчанию для необязательных аргументов 67
4.2.5 Функции с переменным типом входных аргументов 69
4.3 Устойчивые функции 71
4.3.1 Функции `warning` и `error` 71
4.3.2 Общая схема для проверок входных аргументов 73
4.3.3 Пример устойчивой функции 74
4.4 Использование `parameters` 75
4.4.1 Обзор модуля 75
4.4.2 Практический случай 78
4.4.3 Проблемы с модулем `parameters` 81
4.5 Область видимости переменных в стеке вызовов 83
4.5.1 Обзор области видимости переменных 83
4.5.2 Плохая функция: неоднозначный случай 85
4.5.3 Плохая функция: случай тихого обрушения 86
4.6 Проблемы с функциями обратного вызова 89
4.6.1 Бесконечная рекурсия 89
4.6.2 Неправильный индекс 91
4.6.3 Решения 92
4.6.4 Функции обратного вызова с дополнительными аргументами 93
4.7 Мета-программирование: `execstr` и `deff` 95
4.7.1 Основное использование `execstr` 95
4.7.2 Основное использование `deff` 97
4.7.3 Практический пример оптимизации 98
4.8 Заметки и ссылки 101

5 Производительность 102
5.1 Измерение производительности 102
5.1.1 Основные применения 103
5.1.2 Пользовательское время и время ЦП 104
5.1.3 Профилирование функции 105
5.1.4 Функция `benchfun` 112
5.2 Основы векторизации 113
5.2.1 Интерпретатор 114
5.2.2 Циклы в сравнении с векторизацией 115
5.2.3 Пример анализа производительности 116
5.3 Уловки оптимизации 118
5.3.1 Опасность динамических матриц 118
5.3.2 Линейная индексация матрицы 120
5.3.3 Обращение через матрицу логических значений 122
5.3.4 Повтор строк или столбцов вектора 123
5.3.5 Комбинирование векторизованных функций 123
5.3.6 Постолбцовое обращение быстрее 125
5.4 Оптимизированные библиотеки линейной алгебры 127
5.4.1 BLAS, LAPACK, ATLAS и MKL 128
5.4.2 Методы низкоуровневой оптимизации 129
5.4.3 Установка оптимизированных библиотек линейной алгебры для Scilab под Windows 131

5.4.4 Установка оптимизированных библиотек линейной алгебры для Scilab под Linux . . . 133
5.4.5 Пример улучшения производительности . . . 134
5.5 Измерение числа операций с плавающей запятой за секунду (флопс) . . . 135
5.5.1 Произведение матрицы на матрицу . . . 135
5.5.2 Обратный слэш . . . 138
5.5.3 Многоядерные вычисления . . . 140
5.6 Замечания и ссылки . . . 140
5.7 Упражнения . . . 142

6 Благодарности 143

7 Ответы к упражнениям 144
7.1 Ответы к разделу 2 . . . 144
7.2 Ответы к разделу 3 . . . 145
7.3 Ответы к разделу 5 . . . 146

Предметный указатель 150

Список литературы 150

Copyright © 2008-2010 - Michael Baudin
Copyright © 2010-2011 - DIGITEO - Michael Baudin
Copyright © 2011 - Stanislav Kroter (translation into Russian)
Этот файл должен использоваться на условиях Creative Commons Attribution-ShareAlike 3.0 Unported License:

http://creativecommons.org/licenses/by-sa/3.0

1 Введение

Этот документ является проектом с открытым исходным кодом. Исходный код на LaTeXдоступен на Scilab Forge:

http://forge.scilab.org/index.php/p/docprogscilab

Исходные коды на LaTeXпредоставляются на условиях лицензии Creative Commons Attribution-ShareAlike 3.0 Unported License:

http://creativecommons.org/licenses/by-sa/3.0

Файлы-сценарии Scilab предоставляются на Forge, внутри проекта в подкаталоге `scripts`. Файлы-сценарии распространяются на условиях лицензии CeCiLL:

http://www.cecill.info/licences/Licence_CeCILL_V2-en.txt

2 Управление переменными и памятью

В этом разделе мы опишем несколько характеристик Scilab, которые позволяют управлять переменными и памятью. На самом деле, мы иногда обращаемся к большим вычислениям, так что, для того, чтобы взять от Scilab всё, мы должны увеличить память, доступную для переменных.

Мы начнём с представления стека, который Scilab использует для управления своей памятью. Мы представим как использовать функцию `stacksize` для того, чтобы настроить размер стека. Затем мы проанализируем максимально доступную память для Scilab, в зависимости от ограничений операционной системы. Мы кратко представим функцию `who` в качестве инструмента осведомления о переменных, определённых в данный момент. Затем мы выделим переносимость переменных и функций, так что мы сможем разрабатывать сценарии, которые будут работать одинаково хорошо в различных операционных системах. Мы представим функцию `clear`, которая позволяет уничтожать переменные когда кончается память. Наконец, мы представим две функции часто используемые когда мы хотим динамически менять поведение алгоритма в зависимости от типа переменной, то есть мы представим функции `type` и `typeof`.

Информация, представленная в этом разделе будет интересна тем пользователям, которые хотят более глубоко понять внутреннее устройство Scilab. Между прочим, точное управление памятью является ключевой характеристикой, которая позволит выполнить вычисления наиболее требовательные к памяти. Команды, которые позволяют управлять переменными и памятью представлены на рисунке 1.

2.1 Стек

В Scilab версии 5 (и прежних версиях), память управляется с помощью *стека*. При запуске Scilab распределяет фиксированное количество памяти для хранения переменных данной сессии. Некоторые переменные уже предопределены при запуске, что потребляет небольшое количество памяти, но бо́льшая часть памяти свободна и предоставлена пользователю. Когда бы пользователь ни определил переменную, внутри стека используется соответствующая память и в соответствующее количество памяти удаляется из оставшейся

clear	уничтожить переменные
clearglobal	уничтожить глобальные переменные
global	определить глобальные переменные
isglobal	проверить является ли переменная глобальной
stacksize	установить размер стека Scilab
gstacksize	установить/получить размер стека Scilab
who	листинг переменных
who_user	листинг пользовательских переменных
whos	листинг переменных в формате long

Рис. 1: Функции для управления переменными.

свободной части стека. Эта ситуация представлена на рисунке 2. Когда не остаётся свободной памяти в стеке, пользователь больше не может создать новую переменную. В этом случае пользователь должен либо разрушить существующую переменную, либо увеличить размер стека.

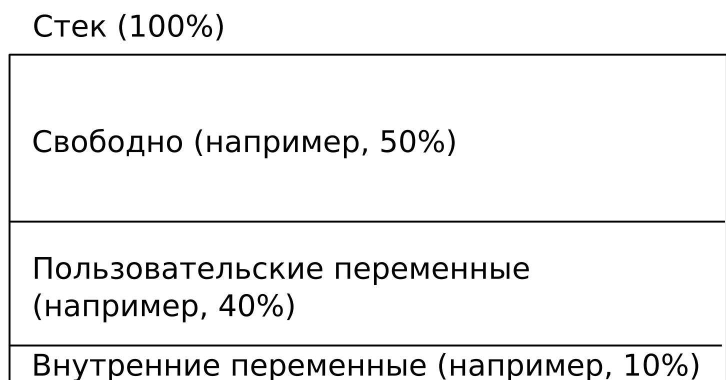

Рис. 2: Стек Scilab.

Всегда есть некоторое сомнение о бите и байтах, их обозначениях и их единицах измерения. Бит (двоичная цифра) — это переменная, которая может быть равна только 0 или 1. Байт (обозначается латинской B или Б) состоит из восьми битов. Есть два разных типа обозначений единиц измерения для множества байтов. В десятичной системе единиц измерения один килобайт состоит из 1000 байт, так что используется обозначение (латинское KB или КБ) (10^3 байт), MB (10^6 байт), GB (10^9 байт) и более (такие как TB для терабайт и PB для петабайт). В двоичной системе единиц измерения один килобайт состоит из 1024 байт, так что обозначения включают в себя строчную букву «i» в своих единицах измерения: KiB, MiB, и так далее. . . . В данном документе мы используем только десятичную систему единиц измерения.

Функция `stacksize` позволяет узнать текущее состояние стека. В сессии, выполняемой после запуска Scilab'а, мы вызываем `stacksize` для того, чтобы получить текущие свойства стека.

```
-->stacksize()
 ans  =
    5000000.    33360.
```

Первое число, *5 000 000*, это общее число 64ˣ-битных слов, которые могут быть сохранены в стеке. Второе число, *33 360*, это число 64ˣ-битных слов, которые уже используются. Это значит, что только *5 000 000 – 33 360 = 4 966 640* 64ˣ-битных слов доступны для пользователя.

Число *5 000 000* равно числу 64ˣ-битных чисел с плавающей запятой двойной точности (т. е. «double»), которые могут быть сохранены, *если стек содержит только этот тип данных.* Общий размер стека *5 000 000* соответствует 40 МБ, поскольку $5\,000\,000 \times 8 = 40\,000\,000$. Эта память может быть полностью заполнена плотной квадратной матрицей размерами 2236 на 2236 с числами двойной точности (double), поскольку $\sqrt{5000000} \approx 2236$.

Действительно, стек используется для хранения как действительных значений, целочисленных, строковых, так и более сложных структур данных. Когда хранится 8ⁱ-битное целочисленное значение, это соответствует 1/8 памяти, требуемой для хранения 64ˣ-битного слова (поскольку $8 \times 8 = 64$). В данном случае требуется только 1/8 часть памяти, требуемой для хранения 64ˣ-битного слова, чтобы сохранить целочисленное значение. В общем, второе целочисленное значение сохраняется во 2/8 части того же слова, так что память не теряется.

Установка по умолчанию возможно достаточна для большинства случаев, но может быть ограничение для некоторых приложений

В сессии Scilab'a мы видим, что создание случайной плотной матрицы 2300×2300 генерирует ошибку в то время, как создание матрицы 2200×2200 возможно.

```
\begin{scriptsize}
 \begin{verbatim}
  -->A=rand(2300,2300)
                   !--error 17
  rand: Превышен допустимый размер стека (используйте функцию
  stacksize для его увеличения).
-->clear A
-->A=rand(2200,2200);
```

В том случае, где нам нужно сохранить большие наборы данных, нам нужно увеличить размер стека. Команда `stacksize("max")` позволяет настроить размер стека таким, чтобы распределить максимально возможное количество памяти системы. Следующий файл-сценарий предоставляет пример этой функции, выполненной на ноутбуке с операционной системой GNU/Linux с объёмом памяти 1 ГБ. Функция `format` используется для того, чтобы отображались все знаки.

```
-->format(25)
-->stacksize("max")
-->stacksize()
 ans  =
    28176384.    35077.
```

Мы можем видеть на этот раз, что общая доступная память в стеке соответствует *28 176 384* единицам 64ˣ-битных слов, что соответствует 225 МБ (поскольку $28\,176\,384 \times 8 = 225\,411\,072$). Максимальная плотная матрица, которая может быть сохранена, сейчас размером 5308 на 5308, поскольку $\sqrt{28176384} \approx 5308$.

Теперь увеличим размер стека до максимума и создадим плотную квадратную матрицу чисел типа double размером 3000 на 3000.

```
-->stacksize("max")
-->A=rand(3000,3000);
```

Предположим, что у нас 32ˣ-битная машина с Windows XP и 4 ГБ памяти. На этой машине у нас установлен Scilab 5.2.2. Определим плотную квадратную матрицу чисел типа double размером 12 000 на 12 000. Это соответствует приблизительно 1,2 ГБ памяти.

```
-->stacksize("max")
-->format(25)
-->stacksize()
 ans  =
    152611536.    36820.
-->sqrt(152611536)
 ans  =
    12353.604170443539260305
-->A=zeros(12000,12000);
```

Мы задали размер плотной матрицы типа double для того, чтобы получить грубое представление о том, чему эти числа соответствуют на практике. Конечно, пользователь может по-прежнему управлять гораздо бо́льшими матрицами, например, если они являются разреженными матрицами. Но в любом случае общая используемая память может превысить размер стека.

Scilab версии 5 (и ранние) может адресовать $2^{31} \approx 2{,}1 \times 10^9$ байт, т. е. 2,1 ГБ памяти. Это соответствует $2^{31}/8 = 268\,435\,456$ чисел типа double, которыми может быть заполнена плотная квадратная матрица типа double размером 16 384 на 16 384. Это ограничение вызвана внутренним устройством Scilab в любой операционной системе. Более того, ограничения, налагаемые операционной системой могут больше ограничить эту память. Эти темы детально рассматриваются в следующем разделе, который представляет внутренние ограничения Scilab и ограничения, вызванные различными операционными системами, на которых может быть запущен Scilab.

2.2 Подробнее об управлении памятью

В этом разделе мы расскажем более детально о доступной памяти в Scilab с точки зрения пользователя. Мы отделим ограничения памяти, вызванные устройством Scilab от ограничений, вызванных устройством операционной системы.

В первом разделе мы анализируем внутреннее устройство Scilab и способ управления памятью с помощью 32ˣ-битных целых чисел со знаком. Затем мы покажем ограничение распределения памяти на различных 32ˣ- и 64ˣ-битных операционных системах. В последнем разделе мы представляем алгоритм, используемый функцией `stacksize`.

Эти разделы скорее технические и могут быть пропущены большинством пользователей. Но пользователи, более опытные в теме памяти, или те, кто хочет узнать как точное устройство Scilab версии 5, могут узнать о точных причинах этих ограничений.

2.2.1 Ограничения памяти со стороны Scilab

Scilab версии 5 (и ранние) обращается к своей внутренней памяти (т. е. стеку) 32ˣ-битными целыми числами со знаком в любой операционной системе на которой он запущен. Это объясняет то, что максимальное количество памяти, которое Scilab может использовать, равно 2,1 ГБ. В этом разделе мы расскажем как это реализовано в большинстве частей Scilab'а, которые управляют памятью.

В Scilab, *шлюз* — это функция C или Fortran, которая даёт пользователю особую функцию. А конкретнее, она соединяет интерпретатор с отдельным набором библиотечных функк-

ций чтением входных аргументов, заданных пользователем, и записью выходных аргументов, затребованных пользователем.

Рассмотрим следующий пример Scilab.

```
x = 3
y = sin(x)
```

Здесь переменные `x` и `y` являются матрицами типа double. В шлюзе функции `sin` мы проверяем число входных аргументов и их тип. Как только переменная `x` проходит проверку, мы создаём выходную переменную `y` в стеке. Наконец, мы вызываем функцию `sin`, предоставленную математической библиотекой и кладём результат в `y`.

Шлюз точно получает доступ к адресам в стеке, который содержит данные переменных `x` и `y`. Для этой цели заголовок шлюза Fortran может содержать объявление, такое как:

```
      integer il
```

где `il` — переменная, которая хранит положение в стеке, который хранит переменную, которая используется. Внутри стека каждый адрес соответствует одному байту и точно управляется с помощью исходного кода каждого шлюза. Целые числа представляют различные места в стеке, т. е. различные адреса байтов. Рисунок 3 демонстрирует способ, которым ядро Scilab'а версии 5 управляет байтами в стеке. Если целочисленная переменная `il` связана с отдельным адресом в стеке, выражение `il+1` идентифицирует следующий адрес стека.

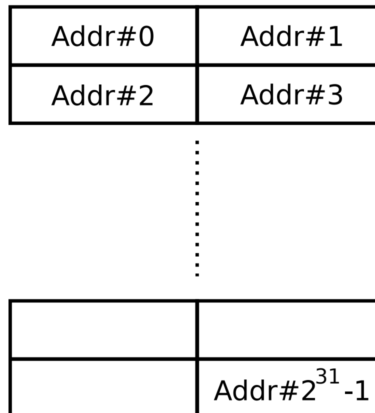

Рис. 3: Детали управления стеком. – Адреса управляются точно ядром Scilab с помощью 32^x-битных чисел со знаком.

Каждая переменная хранится в виде пары связанных друг с другом заголовка и данных. Это изображено на рисунке 4, который раскрывает детали заголовка переменной. Следующий исходный код Fortran является типичным кодом первой строки в шлюзе наследия в Scilab. Переменная `il` содержит адрес начала текущей переменной, скажем `x` для примера. В действительном исходном коде, мы уже проверили, что текущая переменная является матрицей чисел типа double, то есть, мы уже проверили тип переменной. Затем переменная `m` устанавливается равной числу строк матрицы, а переменная `n` — числу столбцов. Наконец, переменная `it` обнуляется, если матрица вещественная, и устанавливается равной 1, если матрица комплексная (т. е. содержит как вещественную, так и мнимую часть).

```
m=istk(il+1)
n=istk(il+2)
it=istk(il+3)
```

Как мы можем видеть, мы просто используем такое выражение как `il+1` или `il+2` для того, чтобы переместиться от одного адреса к другому, то есть, от одного бита к следующему. Из-за целочисленной арифметике, используемой в шлюзах на целых числах, таких как `il`, мы должны обратить внимание на диапазон значений, которые мы можем достичь этим отдельным типом данных.

Рис. 4: Детали управления стеком. – Каждая переменная связана с заголовком, который позволяет доступ к данным.

Тип данных `integer` в Fortran является 32ˣ-битным целым числом со знаком. 32ˣ-битные целые числа со знаком лежат в диапазоне от $-2^{31} = -2\,147\,483\,648$ до $2^{31} - 1 = 2\,147\,483\,647$. В ядре Scilab мы не используем отрицательные целые числа, для идентификации адресов внутри стека. Это напрямую подразумевает, что Scilab не может адресовать более 2 147 483 647 байт, т. е. 2,1 ГБ.

Из-за множества шлюзов в Scilab, нельзя непосредственно передать управление памятью от стека более динамичному распределению памяти. Это является проектом Scilab версии 6, которая сейчас разрабатывается и появится в следующих годах.

Ограничения, связанные с различными операционными системами анализируются в следующих разделах.

2.2.2 Ограничения памяти в 32^{x}- и 64^{x}-битных системах

В этом разделе мы анализируем ограничения памяти Scilab версии 5 в зависимости от версии операционной системы, на которой запущен Scilab.

В 32^{x}-битных операционных системах память адресуется оперированием 32^{x}-битными целыми числами без знака. Следовательно в 32^{x}-битной системе мы можем адресовать 4,2 ГБ, то есть $2^{32} = 4\,294\,967\,296$. В зависимости от конкретной операционной системы (Windows, Linux) и в зависимости от конкретной версии этой операционной системы (Windows XP, Windows Vista, версия ядра Linux и т. д.), этот предел может (или не может) быть достигнут.

В 64^{x}-битных системах память адресуется операционной системой с помощью 64^{x}-битными целыми числами без знака. Следовательно, очевидно, что максимально доступная память в Scilab равна $2^{64} \approx 1{,}8 \times 10^{10}$ ГБ. Это больше, чем любая доступная физическая память на рынке (на момент написания этой статьи).

Но будь это 32^{x}- или 64^{x}-битная операционная система, будь это Windows или Linux, стек Scilab по-прежнему управляется внутри через 32^{x}-битные целые числа со знаком. Следовательно, можно использовать не более 2,1 ГБ памяти для переменных Scilab, которые хранятся внутри стека.

На практике мы можем иметь дело с некоторыми отдельными работами по линейной алгебре или графике в 64^x-битных системах, которые не работают в 32^x-битных системах. Это может быть вызвано временным использованием памяти операционной системы в противовес стеку Scilab'а. Например, разработчик может использовать функции `malloc/free` вместо использования части стека. Это также может случиться, если память выделена не библиотекой Scilab, а на более низком уровне подбиблиотекой, используемой Scilab'ом. В некоторых случаях этого достаточно, чтобы заставить файл-сценарий работать в 64^x-битной системе, в то время как этот же самый файл-сценарий не будет работать в 32^x-битной системе.

2.2.3 Алгоритм в `stacksize`

В этом разделе мы представляем алгоритм, используемый функцией `stacksize` для распределения памяти.

Когда вызывается команда `stacksize("max")`, мы сначала вычисляем размер текущего стека. Затем, мы вычисляем размер наибольшей области свободной памяти. Если текущий размер стека равен наибольшей области свободной памяти, мы немедленно возвращаемся. Если нет, то мы устанавливаем размер стека в минимум, который распределяет минимальное количество памяти, которое позволяет сохранить используемую в данный момент память. Используемые переменные затем копируются в новое пространство памяти, а старый стек перераспределяется. Затем мы устанавливаем размер стека в максимум.

Мы уже видели, что Scilab может адресовать $2^{31} \approx 2{,}1 \times 10^9$ байт, т. е. 2,1 ГБ памяти. На практике, это может быть трудно или даже невозможно распределить такое большое количество памяти. Это частично вызвано ограничениями различных операционных систем в которых запущен Scilab, что и анализируется в следующем разделе.

2.3 Список переменных и функция `who`

Следующая команда показывает поведение функции `who`, которая показывает текущий список переменных, а также состояние стека.

```
-->who

Ваши переменные:

                    WSCI                    home             scinoteslib     modules_managerlib
            atomsguilib                atomslib                matiolib         parameterslib
simulated_annealinglib  genetic_algorithmslib             umfpacklib                   fft
         scicos_autolib         scicos_utilslib                 xcoslib        spreadsheetlib
          demo_toolslib     development_toolslib               soundlib            texmacslib
             tclscilib                m2scilib         maple2scilablib compatibility_functilib
          statisticslib        windows_toolslib                timelib             stringlib
   special_functionslib               sparselib     signal_processinglib                    %z
                     %s          polynomialslib          overloadinglib       optimsimplexlib
           optimbaselib           neldermeadlib        optimizationlib      interpolationlib
     linear_algebralib                  jvmlib        output_streamlib                 iolib
            integerlib         dynamic_linklib              uitreelib                guilib
   data_structureslib               cacsdlib      graphic_exportlib           datatipslib
           graphicslib              fileiolib           functionslib elementary_functionslib
differential_equationlib           helptoolslib                corelib                   PWD
                    %tk                    %pvm                  MSDOS                    %F
                     %T                    %nan                   %inf                   SCI
                SCIHOME                  TMPDIR                   %gui                 %fftw
                      $                      %t                     %f                  %eps
                    %io                      %i                     %e                   %pi

 используются       7936 элементов из      5000000.
```

```
 и          80 переменных из       9231.

Ваши глобальные переменные:

          %modalWarning                 demolist              %driverName          %exportFileName

 используется        601 елементов из     999.
 и           4 переменных из       767.
```

Все переменные, чьи имена оканчиваются на «lib» (как `optimizationlib`, например) связаны с библиотеками внутренних функций Scilab'a. Некоторые переменные, начинающиеся со знака «%», т. е. `%i`, `%e` и `%pi`, связаны с предопределёнными переменными Scilab, поскольку они являются математическими константами. Другие переменные, чьи имена начинаются со знака «%» являются связанными с точностью чисел с плавающей запятой и стандартом IEEE. Эти переменные — `%eps`, `%nan` и `%inf`. Переменные в именами в верхнем регистре `SCI`, `SCIHOME`, `MSDOS` и `TMPDIR` позволяют создать переносимые файлы-сценарии, т. е. файлы-сценарии, которые могут выполняться независимо от директории установки или операционной системы. Эти переменные описаны в следующем разделе.

2.4 Переносимость переменных и функций

Есть несколько предопределённых переменных и функций, которые позволяют создавать переносимые файлы-сценарии, то есть, файлы-сценарии, которые одинаково хорошо работают в Windows, Linux или Mac. Эти переменные представлены на рисунках 5 и 6. Эти переменные и функции используются главным образом для создания внешних модулей, но могут быть иметь практическую ценность в широком наборе ситуаций.

`SCI`	директория, куда установлен Scilab
`SCIHOME`	директория, в которой находятся пользовательские
`MSDOS`	*истина*, если текущая операционная система — Windows
`TMPDIR`	временная директория для текущей сессии Scilab'a

Рис. 5: Переменные, используемые для переносимости.

`[OS,Version]=getos()`	имя текущей операционной системы
`f = fullfile(p1,p2,...)`	формирование пути файла из частей
`s = filesep()`	разделитель файлов, зависящий от операционной системы

Рис. 6: Функции, используемые для переносимости.

В следующем примере мы проверяем значения некоторых предопределённых переменных на Linux-машине.

```
-->SCI
 SCI  =
 /home/myname/Programs/Scilab-5.1/scilab-5.1/share/scilab
-->SCIHOME
 SCIHOME  =
 /home/myname/.Scilab/scilab-5.1
```

```
-->MSDOS
 MSDOS  =
  F
-->TMPDIR
 TMPDIR  =
 /tmp/SD_8658_
```

Переменная TMPDIR, которая содержкт имя временной директории, связана с текущей сессией Scilab: каждая сессия Scilab имеет уникальную временную директорию.

Временная директория Scilab'a создаётся при запуске Scilab'a (и не уничтожается до самого выхода из Scilab). На практике, мы можем использовать переменную TMPDIR в тестовых файлах-сценариях, где мы должны создавать временные файлы. В этом случае файловая система не засоряется временными файлами и гораздо меньше шансов переписать важные файлы.

Мы часто используем переменную SCIHOME для размещения файла .startup на текущей машине.

Для того, чтобы проиллюстрировать использование переменной SCIHOME, мы рассмотрим задачу поиска файла .startup Scilab'a.

Например, мы иногда вручную загружаем некоторые специфичные внешние модули при запуске Scilab'a. Чтобы это сделать, мы вводим команды exec в файл .startup, который автоматически загружается при запуске Scilab'a. Для того, чтобы открыть файл .startup мы можем использовать следующую команду.

```
editor(fullfile(SCIHOME,".scilab"))
```

На самом деле, функция fullfile создаёт абсолютное имя директории из директории SCIHOME до файла .startup. Более того, функция fullfile автоматически использует разделитель директорий, соответствующий операционной системе / для Linux и \ для Windows. Следующий пример показывает результат работы функции fullfile в операционной системе Windows.

```
-->fullfile(SCIHOME,".scilab")
 ans  =
 C:\DOCUME~1\Root\APPLIC~1\Scilab\scilab-5.3.0-beta-2\.scilab
```

Функция filesep возвращает строку, представляющую разделитель директорий в текущей операционной системе. В следующем примере мы вызовем функцию filesep в Windows.

```
-->filesep()
 ans  =
 \
```

В Linux-системах, функция filesep вернёт «/».

В Scilab'е есть две возможности, которые позволяют получить имя операционной системы: функция getos и переменная MSDOS. Обе возможности позволяют создавать файлы-сценарии, которые управляют особыми настройками, зависящими от операционной системы. Функция getos позволяет управлять единообразно во всех операционных системах, что ведёт к улучшенной переносимости. Вот почему первым мы выбрали представление этой функции

Функция getos возвращает строку, содержащую имя текущей операционной системы и, по выбору, её версию. В следующем примере мы вызываем функцию getos в Windows XP.

```
-->[OS,Version]=getos()
 Version  =
 XP
 OS  =
 Windows
```

Функция getos может быть обычно использована в командах select, как показано ниже.

```
OS=getos()
select OS
case "Windows" then
  disp("Scilab on Windows")
case "Linux" then
  disp("Scilab on Linux")
case "Darwin" then
  disp("Scilab on MacOs")
else
  error("Scilab on Unknown platform")
end
```

Обычное использование переменной MSDOS представлено на следующем примере.

```
if ( MSDOS ) then
  // команды Windows
else
  // команды Linux
end
```

На практике рассмотрим ситуацию, где мы хотим вызвать внешнюю программу с максимально возможной переносимостью. Предположим, что под Windows эта программа файлом-сценарием «.bat», под Linux эта программа вызывается скриптом «.sh». В данном случае мы могли бы написать файл-сценарий, используя переменную MSDOS и функцию unix, которая выполняет внешнюю программу. Несмотря на своё имя, функция unix работает одинаково под Linux и Windows.

```
if ( MSDOS ) then
  unix("myprogram.bat")
else
  unix("myprogram.sh")
end
```

Предыдущий пример показывает, что можно написать переносимую Scilab-программу, которая работает одинаково в разных операционных системах. Может возникнуть более сложная ситуация, поскольку часто случается, что путь, ведущий к программе, также зависит от операционной системы. Другая ситуация состоит в том, что когда мы хотим скомпилировать исходный код при помощи Scilab, используя, например, функции ilib_for_link или tbx_build_src. В этом случае мы хотели бы передать компилятору некоторые особенные опции, которые зависят от операционной системы. Во всех этих ситуациях переменная MSDOS позволяет сделать единый исходный код, который остаётся переносимым в различных системах.

Переменная SCI позволяет вычислить пути относительно текущей установки Scilab. Следующий пример демонстрирует образец использования этой переменной. Сначала мы получаем путь макроса, указанного Scilab'ом. Затем объединяем переменную SCI с относительным путём до файла и передаём это в функцию ls. Наконец, мы конкатенируем переменную SCI со строкой, содержащей относительный путь до файла-сценария и передаём это

в функцию editor. В обоих случаях команды не зависят от абсолютного пути до файла, что делает их более переносимыми.

```
-->get_function_path("numdiff")
 ans  =
 /home/myname/Programs/Scilab-5.1/scilab-5.1/...
   share/scilab/modules/optimization/macros/numdiff.sci
-->ls (fullfile(SCI,"/modules/optimization/macros/numdiff.sci"))
 ans  =
 /home/myname/Programs/Scilab-5.1/scilab-5.1/...
   share/scilab/modules/optimization/macros/numdiff.sci
-->editor(fullfile(SCI,"/modules/optimization/macros/numdiff.sci"))
```

В общих ситуациях часто кажется, что проще писать файл-сценарий только для конкретной операционной системы, поскольку она единственная, которую мы используем в данный момент. В этом случае мы склонны использовать инструменты, которые недоступны для других операционных систем. Например, мы можем полагаться на особое расположение файла, который может быть найден только в Linux. Или мы можем использовать какую-нибудь консоль, зависящую от операционной системы, или инструкции управления файлами. В конечном счёте, полезный файл-сценарий имеет высокую вероятность быть использованным в операционной системе, которая не была первичной целью исходного автора. В данной ситуации тратится много времени, чтобы обновить файл-сценарий так, чтобы он заработал на новой платформе.

Поскольку Scilab сам по себе является переносимым языком, то в большинстве случаев было бы неплохо подумать о том, как этот файл-сценарий будет переноситься прямо с начала разработки. Если есть действительно трудный или неизбежный пункт, то, конечно, разумно снизить переносимость и просто решить задачу так, чтобы сократить разработку. На практике такое случается не так часто, как кажется, так что обычное правило состоит в том, чтобы писать код как можно более переносимым.

2.5 Уничтожение переменных : clear

Переменная может быть создана по желанию, когда она потребуется. Она также может быть уничтожена напрямую с помощью функции clear, когда в ней больше не нужды. Это может быть полезным, когда большая матрица хранится в данный момент в памяти и если память становится проблемой.

В следующем примере мы определим большую матрицу случайных чисел A. Когда мы попытаемся определить вторую матрицу B, то видим, что нет больше памяти. Следовательно, мы используем функцию clear для уничтожения матрицы A. Тогда мы можем создать матрицу B.

```
-->A = rand(2000,2000);
-->B = rand(2000,2000);
                      !--error 17
rand: Превышен допустимый размер стека
(используйте функцию stacksize для его увеличения).
-->clear A
-->B = rand(2000,2000);
```

Функцию clear следует использовать только по необходимости, то есть, когда вычисление не может быть выполнено из-за памяти. Это предупреждение верно для разработчиков, которые используют компилируемые языки, где памятью управляют напрямую. В языке Scilab'а память управляется Scilab'ом и, вообще, нет причин управлять ею самостоятельно.

Родственной темой является управление переменными в функциях. Когда тело функции выполнилось интерпретатором, все переменные, которые были использованы в теле, и которые не являются выходными аргументами, автоматически удаляются. Следовательно нет нужды напрямую вызывать функцию `clear` в этом случае.

2.6 Функции `type` и `typeof`

Scilab может создавать различные типы переменных, такие как матрицы, полиномы, булевы переменные, целочисленные и другие типы структур данных. Функции `type` и `typeof` позволяют узнать конкретный тип данной переменной. Функция `type` возвращает целое число с плавающей запятой, а функция `typeof` возвращает строку. Эти функции представлены на рисунке 7.

`type`	Возвращает строку
`typeof`	Возвращает целое число с плавающей запятой

Рис. 7: Функции для вычисления типа переменной.

Рисунок 8 представляет различные выходные значения функций `type` и `typeof`.

type	**typeof**	**описание**
1	`constant`	матрица вещественных или комплексных констант
2	`polynomial`	матрица многочленов
4	`boolean`	булева матрица
5	`sparse`	разреженная матрица
6	`boolean sparse`	разреженная булева матрица
7	`Matlab sparse`	разреженная матрица Matlab'a
8	`int8`, `int16`, `int32`, `uint8`, `uint16` или `uint32`	матрица целочисленных значений, хранимых в 1, 2 или 4 байтах
9	`handle`	матрица указателей графиков
10	`string`	матрица символьных строк
11	`function`	некомпилированная функция (код Scilab)
13	`function`	скомпилированная функция (код Scilab)
14	`library`	библиотека функций
15	`list`	список
16	`rational`, `state-space` или тип	типизированный список (tlist)
17	`hypermat`, `st`, `ce` или тип	типизированный список ориентированный на матрицу (mlist)
128	`pointer`	разреженная матрица LU-разложения
129	`size implicit`	размер неявного многочлена, используемый для индексации
130	`fptr`	код, свойственный Scilab'y (C или Fortran'y)

Рис. 8: Возвращаемые значения функций `type` и `typeof`.

В следующем примере мы создаём матрицу 2×2 чисел типа double и используем `type` и `typeof` для получения типа этой матрицы.

```
-->A=eye(2,2)
 A  =
    1.    0.
    0.    1.
-->type(A)
 ans  =
    1.
-->typeof(A)
 ans  =
 constant
```

Эти две функции полезны во время обработки входных аргументов функции. Эта тема будет ещё раз рассмотрена в данном документе позднее, в разделе 4.2.5, когда мы рассмотрим функции с переменным типом входных аргументов.

Если тип переменной `tlist` или `mlist`, то значение, возвращаемое функцией `typeof`, находится в первой строке первого пункта списка. Эта тема ещё раз затрагивается в разделе 3.6, где мы представляем типизированные списки.

Типы данных `cell` и `struct` являются особыми формами `mlist`'ов, поэтому они связаны с `type`, равным 17 или `typeof`, равным «ce» и «st».

2.7 Заметки и ссылки

Похоже, что в будущих версиях Scilab'а, память Scilab'а не будет управляться стеком. На самом деле, эта характерная черта является наследием предшественника Scilab'а, т. е. Matlab'а. Мы подчёркиваем, что Matlab не использует стек уже давно, приблизительно с 80ˣ годов, со времени, когда исходный код Matlab'а был переработан [41]. С другой стороны, Scilab сохраняет этот довольно старый способ управления памятью.

Некоторые дополнительные подробности об управлении памятью в Matlab даны в [34]. В технических заметках [35] авторы представляют методы, избегающие ошибки памяти в Matlab. В технических заметках [36] авторы представляют максимальный размер матрицы, доступный в Matlab на различных платформах. В технических заметках [37] авторы представляют выгоду использования 64ˣ-битного Matlab'а перед 32ˣ-битным.

2.8 Упражнения

Упражнение 2.1 (***Максимальный размер стека***) Проверьте максимальный размер стека вашей машины.

Упражнение 2.2 (*who_user*) Запустите Scilab, выполните следующий код и посмотрите результат на вашей машине.

```
who_user()
A=ones(100,100);
who_user()
```

Упражнение 2.3 (*whos*) Запустите Scilab, выполните следующий код и посмотрите результат на вашей машине.

```
whos()
```

3 Специальные типы данных

В этом разделе мы анализируем типы данных Scilab'а, которые наиболее часто используются на практике. Мы рассмотрим строки, целочисленные значения, многочлены, гиперматрицы, `list`'ы и `tlist`'ы. Мы представим некоторые из наиболее полезных свойств системы перегрузки, которые позволяют определять новые поведения для типизированных списков. В последнем разделе мы кратко рассмотрим `cell`, `struct` и `mlist` и сравним их с другими структурами данных.

3.1 Строки

Хотя Scilab не разрабатывался изначально в качестве инструмента управления строками, он предоставляет солидный и мощный набор функций для управления этим типом данных. Список команд, которые связаны со строками Scilab'а представлены на рисунке 9.

string	преобразование в строку
sci2exp	преобразовать выражение в строку
ascii	преобразовать строку в коды ascii (и обратно)
blanks	создать строку из пробелов
convstr	преобразовать регистры символов
emptystr	строка нулевой длины
grep	найти соответствия строки в векторе строк
justify	выравнять символы в столбцах матрицы
length	длина объекта
part	выделение части строки
regexp	найти подстроки, которые соответствуют строке регулярного выражения
strcat	конкатенировать символьные строки
strchr	найти первую встречу символа в строке
strcmp	сравнить символьные строки
strcmpi	сравнить символьные строки (независимо от регистра)
strcspn	получить интервал до указанного символа в строке
strindex	найти позицию символьной строки в другой строке
stripblanks	обрезает пробелы (и табуляторы) в начале и в конце строк
strncpy	копирует символы из строк
strrchr	найти последнюю встречу символа в строке
strrev	возвращает строку задом наперёд
strsplit	разбивает строку на векторы строк
strspn	получить интервал символов в строке
strstr	определить позицию подстроки
strsubst	заменить символьную строку другой символьной строкой
strtod	преобразование строки в число типа double
strtok	разделение строки на лексемы
tokenpos	возвращает позиции лексем в символьной строке
tokens	возвращает лексемы символьной строки
str2code	возвращает целочисленные коды Scilab, связанные с символьной строкой
code2str	возвращает символьную строку, связанную с целочисленными кодами Scilab

Рис. 9: Символьные функции Scilab.

Для того, чтобы создать матрицу строк, мы можем использовать символ « " » и обычный синтаксис для матриц. В следующем примере мы создаём матрицу строк размером 2×3.

```
-->x = ["11111" "22" "333"; "4444" "5" "666"]
 x  =
!11111  22  333  !
!                !
!4444   5   666  !
```

Для того, чтобы вычислить размер матрицы, мы можем использовать функцию `size` как для обычных матриц.

```
-->size(x)
 ans  =
    2.    3.
```

Функция `length`, с другой стороны, возвращает число символов в каждом элементе матрицы

```
-->length(x)
 ans  =
    5.    2.    3.
    4.    1.    3.
```

Возможно, что самая распространённая функция, которую мы используем, это функция `string`. Эта функция позволяет преобразовать свой входной аргумент в строку. В следующем примере мы определим вектор-строку и используем функцию `string` для преобразования его в строку. Затем мы используем функцию `typeof` и проверим, что переменная `str` действительно является строкой. Наконец, мы используем функцию `size` и проверим, что переменная `str` является матрицей строк размером 1×5

```
-->x = [1 2 3 4 5];
-->str = string(x)
 str  =
!1  2  3  4  5  !
-->typeof(str)
 ans  =
 string
-->size(str)
 ans  =
    1.    5.
```

Функция `string` может принимать любой тип входного аргумента.

Позже мы увидим, что `tlist` может быть использован для определения нового типа данных. В этом случае мы можем определить функцию, которая делает так, что функция `string` может работать с этим новым типом данных способом, определённым пользователем. Эта тема рассматривается в разделе 3.8, где мы представим метод, который позволяет определить поведение функции `string`, когда её входной аргумент является типизированным списком, когда её входной аргумент является типизированным списком.

Функция `strcat` конкатенирует свой первый входной аргумент с разделителем, определённым во втором входном аргументе. В следующем примере мы используем функцию `strcat` для получения строки, представляющую сумму целых от 1 до 5.

```
-->strcat(["1" "2" "3" "4" "5"],"+")
 ans  =
 1+2+3+4+5
```

Мы можем сочетать функции `string` и `strcat` для получения строк, которые могут быть легко скопированы и вставлены в файлы-сценарии или доклады. В следующем примере мы определяем вектор-строку `x`, который состоит из целых чисел с плавающей запятой. Затем мы используем функцию `strcat` с пробелом в качестве разделителя для получения чистой строки целых чисел.

```
-->x = [1 2 3 4 5]
 x  =
    1.    2.    3.    4.    5.
-->strcat(string(x)," ")
 ans  =
 1 2 3 4 5
```

Предыдущая строка может быть напрямую скопирована и вставлена в исходный код или доклад. Давайте рассмотрим задачу разработки функции, которая печатает данные в консоли. Предыдущая комбинация функций является эффективным способом формирования компактных сообщений. В следующем примере мы используем функцию `mprintf` для отображения содержимого переменной `x`. Мы используем формат `%s`, который соответствует строкам. Для того, чтобы получить строку, мы объединяем функции `strcat` и `string`.

```
-->mprintf("x=[%s]\n",strcat(string(x)," "))
x=[1 2 3 4 5]
```

Функция `sci2exp` преобразует выражение в строку. Это может быть использовано в тех же целях, что и предыдущий метод, основанный на `strcat`, но форматирование менее гибко. В следующем примере мы используем функцию `sci2exp` для преобразования матрицы-строки целых чисел в матрицу строк размером 1×1.

```
-->x = [1 2 3 4 5];
-->str = sci2exp(x)
 str  =
 [1,2,3,4,5]
-->size(str)
 ans  =
    1.    1.
```

Операторы сравнения, такие как «<» или «>», например, не определены, когда используются строки, т. е., команда `"a« "b"` приводит к ошибке. Вместо этого, мы можем использовать функцию `strcmp` для этих целей. Она возвращает 1, если первый аргумент лексикографически меньше второго, она возвращает 0, если две строки равны или возвращает −1, если второй аргумент лексикографически меньше первого. Поведение функции `strcmp` представлено в следующем примере.

```
-->strcmp("a","b")
 ans  =
  - 1.
-->strcmp("a","a")
 ans  =
    0.
-->strcmp("b","a")
 ans  =
    1.
```

Функции, представленные в таблице 10 позволяют различить различные классы строк, таких как ASCII-символы, цифры, буквы и числа.

Например, функция `isdigit` возвращает матрицу логических значений, где каждый элемент `i` является *истиной*, если символ в позиции `i` строки является цифрой. В следующем

примере мы используем функцию `isdigit`, чтобы проверить, что `"0"` является цифрой, а `"d"` — нет.

```
-->isdigit("0")
 ans  =
  T
-->isdigit("12")
 ans  =
  T T
-->isdigit("d3s4")
 ans  =
  F T F T
```

Функция `regexp` является мощным движком регулярных выражений. Этот компонент был включён в пятой версии Scilab. Он основан на библиотеке PCRE [22], которая предназначена для PERL-совместимости. Шаблон должен быть задан как строка, которая окружена слешами в виде `"/x/"`, где `x` — регулярное выражение.

Далее мы представляем пример использования функции `regexp`. Символ `i` в конце регулярного выражения обозначает, что мы не хотим учитывать регистр символов. Первая буква `a` заставляет выражение соответствовать только строкам, которые начинаются с этой буквы.

```
-->regexp("AXYZC","/a.*?c/i")
 ans  =
    1.
```

Регулярные выражения являются чрезвычайно мощным инструментом для манипуляций с текстом и данными. Очевидно, что этот документ не может даже оцарапать поверхность данной темы. Для дополнительной информации о функции `regexp` пользователь может обратиться к страницам помощи, предоставленных Scilab'ом. Для более глубокого введения читатель может поинтересоваться в [20] Фридла. Как выразился Дж. Фридл, «регулярные выражения позволяют вам кодировать сложную и изощрённую обработку текста, которую вы не когда не могли и представить себе как закодировать».

Упражнение 3.1 представляет практический пример использования функции `regexp`.

3.2 Многочлены

Scilab позволяет управлять одномерными многочленами.Реализация основана на векторе, содержащем коэффициенты многочлена. На уровне пользователя мы можем управлять матрицей многочленов. Основные операции, такие как сложение, вычитание, умножение и деление, применимы для многочленов. Мы можем, конечно, вычислить значение многочлена $p(x)$ для отдельного входного значения x. Более того, Scilab может выполнять операции

isalphanum	проверить, являются ли символы алфавитно-цифровыми
isascii	проверить, являются ли символы 7-битным US-ASCII-кодом
isdigit	проверить, являются ли символы цифрами от 0 до 9
isletter	проверить, являются ли символы буквами алфавита
isnum	проверить, являются ли символы числами

Рис. 10: Функции, связанные с особым классом строк.

более высокого уровня, такие, как вычисление корней, разложение на множители и вычисление наибольшего общего делителя или наименьшего общего кратного двух многочленов. Когда мы делим один многочлен на другой многочлен, мы получаем новую структуру данных, представляющую рациональную функцию.

В этом разделе мы сделаем краткий обзор данной темы. Многочлены и рациональные типы данных представлены на рис. 11. Некоторые из наиболее используемых функций, относящихся к многочленам, представлены на рис. 12. Полный список функций, относящихся к многочленам, представлен на рис. 13.

`polynomial`	многочлен, определённый своими коэффициентами
`rational`	отношение двух многочленов

Рис. 11: Типы данных, относящихся к многочленам.

`poly`	определяет многочлен
`horner`	вычисляет значение многочлена
`coeff`	коэффициенты многочлена
`degree`	степень многочлена
`roots`	корни многочлена
`factors`	вещественное разложение полиномов
`gcd`	наибольший общий делитель
`lcm`	наименьшее общее кратное

Рис. 12: Некоторые основные функции, относящиеся к многочленам.

bezout	clean	cmndred	coeff	coffg	colcompr	degree	denom
derivat	determ	detr	diophant	factors	gcd	hermit	horner
hrmt	htrianr	invr	lcm	lcmdiag	ldiv	numer	pdiv
pol2des	pol2str	polfact	residu	roots	rowcompr	sfact	simp
simp_mode	sylm	systmat					

Рис. 13: Функции, относящиеся к многочленам.

Функция `poly` позволяет определить многочлены. Есть два способа определить их: с помощью их коэффициентов или с помощью их корней. В следующем примере мы создаём многочлен $p(x) = (x-1)(x-2)$ с помощью функции `poly`. Корни этого многочлена, очевидно, $x = 1$ и $x = 2$ — вот почему первым входным аргументом функции `poly` является матрица `[1 2]`. Второй входной аргумент — это символьная строка, используемая для отображения многочлена.

```
-->p=poly([1 2],"x")
 p          =
                   2
     2 - 3x + x
```

В следующем примере мы вызываем функцию `typeof` и проверяем, что переменная `p` является многочленом (polynomial).

```
-->typeof(p)
 ans  =
 polynomial
```

Мы можем также определить многочлен на основе его коэффициентов в порядке возрастания. В следующем примере мы определяем многочлен $q(x) = 1 + 2x$. Мы передаём функции `poly` третий аргумент `"coeff"`, так что она знает, что матрица `[1 2]` представляет коэффициенты многочлена.

```
-->q=poly([1 2],"x","coeff")
 q         =
    1 + 2x
```

Теперь, когда многочлены `p` и `q` определены, мы можем выполнять с ними алгебраические операции. В следующем примере мы суммируем, вычитаем, умножаем и делим многочлены `p` и `q`.

```
-->p+q
 ans  =
                 2
    3 - x + x
-->p-q
 ans  =
                 2
    1 - 5x + x
-->p*q
 ans  =
                 2    3
    2 + x - 5x + 2x
-->r = q/p
 r  =
      1 + 2x
    ----------
                 2
    2 - 3x + x
```

Когда мы делим многочлен `p` на многочлен `q`, то получаем рациональную функцию `r`. Это показано в следующем примере.

```
-->typeof(r)
 ans  =
 rational
```

Для того, чтобы вычислить значение многочлена $p(x)$ для конкретного значения x, мы можем использовать функцию `horner`. В следующем примере мы определяем многочлен $p(x) = (x - 1)(x - 2)$ и вычисляем его значение для точек $x = 0$, $x = 1$, $x = 3$ и $x = 3$, представленных матрицей `[0 1 2 3]`.

```
-->p=poly([1 2],"x")
 p  =
                 2
    2 - 3x + x
-->horner(p,[0 1 2 3])
 ans  =
    2.    0.    0.    2.
```

Название функции `horner` происходит от математика Хорнера (Horner), который разработал алгоритм, используемый в Scilab для вычисления значений многочлена. Этот алгоритм поз-

воляет уменьшить число умножений и сложений, требуемых для этих вычислений. (смотри [26], раздел 4.6.4, "Evaluation of Polynomials").

Если в качестве первого аргумента функции poly квадратная матрица, то она возвращает характеристический многочлен, связанный с матрицей. То есть, если **A** — вещественная квадратная матрица размером $n \times n$, то функция poly может дать многочлен $\det(\mathbf{A} - x\mathbf{I})$, где **I** — единичная матрица. Это представлено на следующем примере.

```
-->A = [1 2;3 4]
 A  =
    1.    2.
    3.    4.
-->p = poly(A,"x")
 p  =
                2
  - 2 - 5x + x
```

Мы легко можем проверить, что предыдущий результат верен с помощью его математического определения. Во-первых, мы можем вычислить корни многочлена p с помощью функции roots, как в предыдущем примере. Во-вторых мы можем вычислить собственные значения матрицы A с помощью функции spec.

```
-->roots(p)
 ans  =
  - 0.3722813
    5.3722813
-->spec(A)
 ans  =
  - 0.3722813
    5.3722813
```

Есть другой способ получить тот же самый многочлен p. В следующем примере мы определяем многочлен px, который представляем одночлен x. Затем мы используем функцию det для вычисления детерминанта матрицы $\mathbf{B} = \mathbf{A} - x\mathbf{I}$. Мы используем функцию eye для получения единичной матрицы размером 2×2.

```
->px = poly([0 1],"x","coeff")
 px  =
    x
-->B = A-px*eye()
 B  =
    1 - x     2
    3         4 - x
-->det(B)
 ans  =
                2
  - 2 - 5x + x
```

Если мы сравним этот результат с предыдущим, то увидим, что они одинаковы. Мы видим, что функция det определена для многочлена A-px*eye(). Это пример того факта, что множество функций определены для входных аргументов многочлена.

В предыдущем примере матрица B была матрицей многочленов размером 2×2. Таким же образом мы можем определить матрицу рациональной функции, как это показано на следующем примере.

```
-->x=poly(0,"x");
-->A=[1 x;x 1+x^2];
-->B=[1/x 1/(1+x);1/(1+x) 1/x^2]
```

```
 B          =
!   1            1     !
! ------       ------ !
!   x          1 + x   !
!                      !
!     1        1       !
!    ---      ---      !
!               2      !
!   1 + x      x       !
```

Это очень полезная возможность Scilab'а для теории систем. Связь между типами данных многочленов, рациональных функций с одной стороны и теорией управления с другой кратко проанализирована в разделе 3.14.

3.3 Гиперматрицы

Матрица — это структура данных, к которым можно получить доступ с помощью двух целочисленных индексов (например, i и j). Гиперматрицы являются более обобщённым типом матриц, к которым можно обратиться с помощью более двух индексов. Эта характеристика знакома разработчикам на фортране, где многомерные массивы являются одной из основных структур данных. Несколько функций, имеющих отношение к гиперматрицам, представлены на рисунке 14.

`hypermat`	Создаёт гиперматрицу
`zeros`	Создаёт гиперматрицу нулей
`ones`	Создаёт гиперматрицу единиц
`matrix`	Создаёт матрицу с новыми размерностями
`squeeze`	Удаление измерений с единичным размером

Рис. 14: Функции, имеющие отношение к гиперматрицам.

В большинстве ситуаций мы можем управлять гиперматрицами как обычными матрицами. В следующем примере мы создаём гиперматрицу `A` размерами $4 \times 3 \times 2$ значений типа double с помощью функции `ones`. Затем мы используем функцию `size` для вычисления размеров этой гиперматрицы.

```
-->A=ones(4,3,2)
 A  =
(:,:,1)

    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
(:,:,2)

    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
-->size(A)
 ans  =
    4.    3.    2.
```

В следующем примере мы создаём гиперматрицу размерами $4\times2\times3$. Первый аргумент функции `hypermat` является матрицей, содержащей число измерений гиперматрицы.

```
-->A=hypermat([4,3,2])
 A  =
(:,:,1)
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
(:,:,2)
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
```

Чтобы ввести или выделить значение из гиперматрицы, мы можем использовать тот же самый синтаксис, что и для матриц. В следующем примере мы установим и получим значение элемента (3,1,2).

```
-->A(3,1,2)=7
 A  =
(:,:,1)
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
(:,:,2)
    0.    0.    0.
    0.    0.    0.
    7.    0.    0.
    0.    0.    0.
-->A(3,1,2)
 ans  =
    7.
```

Оператор двоеточие «`:`» может использоваться для гиперматрицы, как это показано в следующем примере.

```
-->A(2,:,2)
 ans  =
    0.    0.    0.
```

Большинство операций, которые могут быть сделаны с матрицами, могут быть так же сделаны с гиперматрицами. В следующем примере мы определим гиперматрицу `B` и сложим её с `A`.

```
-->B=2 * ones(4,3,2);
-->A + B
 ans  =
(:,:,1)
    3.    3.    3.
    3.    3.    3.
    3.    3.    3.
    3.    3.    3.
(:,:,2)
    3.    3.    3.
    3.    3.    3.
    3.    3.    3.
```

```
    3.    3.    3.
```

Функция hypermat может быть использована когда мы хотим создать гиперматрицу из вектора. В следующем примере мы определяем гиперматрицу размерами $2 \times 3 \times 2$, где значения взяты из набора $\{1, 2, \ldots, 12\}$.

```
-->hypermat([2 3 2],1:12)
 ans  =
(:,:,1)
!   1.    3.    5. !
!   2.    4.    6. !
(:,:,2)
!   7.    9.     11. !
!   8.    10.    12. !
```

Заметим особый порядок значений в полученной гиперматрице. Этот порядок соответствует правилу, по которому самые левые индексы перебираются в первую очередь. Это просто следствие того факта, что для двумерной матрицы значения хранятся столбец за столбцом.

Гиперматрица может также содержать строки, как показано в следующем примере.

```
-->A=hypermat([3,1,2],["a","b","c","d","e","f"])
 A  =
(:,:,1)
!a  !
!   !
!b  !
!   !
!c  !
(:,:,2)
!d  !
!   !
!e  !
!   !
!f  !
```

Важным свойством гиперматриц является то, что все элементы должны быть одного типа. Например, если мы попытаемся ввести число типа double в гиперматрицу строк, созданную прежде, то мы получим сообщение об ошибке:

```
-->A(1,1,1)=0
 !--error 43
Не реализовано в scilab...

at line       8 of function %s_i_c called by :
at line     103 of function generic_i_hm called by :
at line      21 of function %s_i_hm called by :
A(1,1,1)=0
```

3.4 Типы и размерности выделенной гиперматрицы

В этом разделе мы представляем выделение гиперматрицы у которых одно измерение единичного размера. Мы представляем эффект, который это производит это может иметь на характеристики и описываем функцию squeeze.

Мы можем выделить срезы гиперматрицы, производя различные размерности выходных переменных. Например, давайте рассмотрим трёхмерную гиперматрицу размерами $2 \times 4 \times 3$. Иногда полезно использовать оператор двоеточие : для выделения целых строк

или столбцов матрицы или гиперматрицы. Например, команда `A(1,:,:)` выделяет значения, связанные с первым индексом, равным 1, и формирует гиперматрицу размерами $1 \times 4 \times 3$. Похожим образом команда `A(:,1,:)` выделяет значения, связанные со вторым индексом, равным 1, и формирует гиперматрицу размерами $2 \times 1 \times 3$. С другой стороны, команда `A(:,:,1)` выделяет значения, связанные с третьим индексом, равным 1, но формирует матрицу размерами 4×3. Следовательно, и тип и форма выделенной гиперматрицы могут измениться, в зависимости от индексов, по которым мы выделяем.

Этот факт исследован на следующем примере.

```
-->A=hypermat([2,4,3],[1:24])
 A  =
(:,:,1)
    1.    3.    5.    7.
    2.    4.    6.    8.
(:,:,2)
    9.     11.    13.    15.
    10.    12.    14.    16.
(:,:,3)
    17.    19.    21.    23.
    18.    20.    22.    24.
```

Сначала мы экспериментируем с командой выделения `B=A(1,:,:)`.

```
-->B = A(1,:,:)
 B  =
(:,:,1)
    1.    3.    5.    7.
(:,:,2)
    9.    11.    13.    15.
(:,:,3)
    17.    19.    21.    23.
-->typeof(B)
 ans  =
 hypermat
-->size(B)
 ans  =
    1.    4.    3.
```

Мы могли бы поэкспериментировать с командой `B=A(:,1,:)` и получить похожее поведение. Вместо этого, мы поэкспериментируем с командой `B=A(:,:,1)`. Мы видим, что в этом случае выделенная переменная является матрицей вместо гиперматрицы.

```
-->B = A(:,:,1)
 B  =
    1.    3.    5.    7.
    2.    4.    6.    8.
-->typeof(B)
 ans  =
 constant
-->size(B)
 ans  =
    2.    4.
```

Общее правило гласит, что, если последнее измерение выделенной гиперматрицы является единичным, то оно удаляется. Это правило применимо к результатам выделения, получающем в конечном счёте либо гиперматрицу, либо обычную двумерную матрицу. Точнее, когда мы выделяем гиперматрицу размерами $n_1 \times \ldots \times n_j \times n_{j+1} \times \ldots \times n_k$, где $n_j \neq 1$ и

$n_{j+1} = n_{j+2} = \ldots = n_k = 1$, мы получили гиперматрицу размерами $n_1 \times n_j$. Более того, если выделяемая гиперматрица является обычной, т. е. двумерной матрицей, то есть $j \leqslant 2$, то выделенная переменная является матрицей, а не гиперматрицей. Эти два правила доказывают, что Scilab совместим с Matlab в выделении гиперматриц.

Такое поведение может иметь существенное влияние на характеристики выделения гиперматриц. В следующем примере мы создаём гиперматрицу $2 \times 4 \times 3$ чисел типа double и измеряем характеристики выделения по второму или третьему индексам. Мы объединяем выделение и команду `matrix` для того, чтобы получить обычную двумерную матрицу `B` в обоих случаях. Поскольку это выделение чрезвычайно быстрое, мы сделаем это 100 000 раз в цикле и измерим пользовательское время с помощью функций `tic` и `toc`. Мы можем видеть, что выделение второго измерения гораздо медленнее, чем выделение третьего измерения.

```
-->A=hypermat([2,4,3],1:24);
-->B=matrix(A(:,2,:),2,3)
 B  =
    3.    11.    19.
    4.    12.    20.
-->tic();for i=1:100000;B=matrix(A(:,2,:),2,3);end;toc()
 ans  =
    7.832
-->B=matrix(A(:,:,2),2,4)
 B  =
    9.     11.    13.    15.
    10.    12.    14.    16.
-->tic();for i=1:100000;B=matrix(A(:,:,2),2,4);end;toc()
 ans  =
    0.88
```

Причина такого различия характеристик в том, что `A(:,2,:)` — это гиперматрица размером $2 \times 1 \times 3$, а `A(:,:,2)` — это обычная матрица размером 2×4.

В самом деле, команда `matrix(A(:,2,:),2,3)` заставляет функцию `matrix` преобразовать гиперматрицу `A(:,2,:)` размерами $2{\times}1{\times}3$ в матрицу $2{\times}3$, что требует дополнительных шагов от интерпретатора. С другой стороны, команда `matrix(A(:,:,2),2,4)` не требует какой-либо обработки от функции `matrix` function, что в этом случае не требует операций.

Мы можем захотеть просто игнорировать промежуточные подматрицы с размером 1, которые созданы системой выделения подматриц. В этом случае мы можем использовать функцию `squeeze`, которая удаляет измерения с единичным размером.

```
-->A=hypermat([3,1,2],1:6)
 A  =
(:,:,1)
    1.
    2.
    3.
(:,:,2)
    4.
    5.
    6.
-->size(A)
 ans  =
    3.    1.    2.
-->B=squeeze(A)
 B  =
```

```
    1.     4.
    2.     5.
    3.     6.
-->size(B)
 ans   =
    3.     2.
```

3.5 Тип данных list (список)

В этом разделе мы опишем тип данных `list` (список), который используется для управления коллекцией объектов различных типов. Мы очень часто используем списки, когда хотим собрать в один объект набор переменных, которые нельзя сохранить в один, более основной, тип данных. Список может содержать любые уже обсуждённые типы данных (включая функции), а также другие списки. Это позволяет создавать вложенные списки, которые могут быть использованы для создания дерева структур данных. Списки чрезвычайно полезны для определения объектов структурированных данных. Некоторые функции, относящиеся к спискам, представлены на рисунке 15.

`list`	создать список
`null`	удалить элемент списка
`lstcat`	конкатенировать списки
`size`	для списка число элементов (для списка то же, что и `length`)

Рис. 15: Функции, относящиеся к спискам.

На самом деле в Scilab'е есть различные типы списков: обычные списки, типизированные списки и `mlist`. Этот раздел касается обычных списков. Типизированные списки будут рассмотрены в следующем разделе. Тип данных `mlist` будет рассматриваться в разделе 3.9.

В следующем примере мы определяем целые числа с плавающей запятой, строку и матрицу. Затем мы используем функцию `list`, чтобы создать список `mylist`, содержащий эти три элемента.

```
-->myflint = 12;
-->mystr = "foo";
-->mymatrix = [1 2 3 4];
-->mylist = list ( myflint , mystr , mymatrix )
 mylist   =
       mylist(1)
    12.
       mylist(2)
 foo
       mylist(3)
    1.     2.     3.     4.
```

Однажды создав, мы можем получить доступ к `i`-тому элементу списка `mylist` с помощью команды `mylist(i)`, как в следующем примере.

```
-->mylist(1)
 ans   =
    12.
->mylist(2)
 ans   =
```

```
 foo
-->mylist(3)
 ans  =
    1.    2.    3.    4.
```

Число элементов списка может быть вычислено с помощью `size`.

```
-->size(mylist)
 ans  =
    3.
```

В случае, когда мы хотим получить несколько элементов одной командой, мы можем использовать оператор двоеточие «:». В этом случае мы должны установить столько выходных аргументов, сколько элементов требуется получить. В следующем примере мы получаем два элемента с индексами 2 и 3 и устанавливаем переменные `s` и `m`.

```
-->[s,m] = mylist(2:3)
 m  =
    1.    2.    3.    4.
 s  =
 foo
```

Элемент №3 в нашем списке является матрицей. Предположим, что мы хотим получить четвёртое значение в этой матрице. Мы можем иметь доступ к нему непосредственно с помощью следующего синтаксиса.

```
-->mylist(3)(4)
 ans  =
    4.
```

Это гораздо быстрее, чем хранение третьего элемента списка во временной переменной и выделение его четвёртого элемента. В случае, если мы хотим установить значение этого элемента, мы можем использовать тот же синтаксис и обычный оператор `=`, как показано ниже.

```
-->mylist(3)(4) = 12
 mylist  =
       mylist(1)
    12.
       mylist(2)
 foo
       mylist(3)
    1.    2.    3.    12.
```

Очевидно, мы могли бы сделать ту же операцию несколькими промежуточными операциями: выделение третьего элемента списка, обновление матрицы и снова хранение матрицы в списке. Но использование команды `mylist(3)(4) = 12` действительно проще и быстрее.

Для того, чтобы просматривать элементы списка, мы можем использовать прямой метод или метод, основанный, в частности, на команде `for` . Сначала используется прямой метод для того, чтобы подсчитать количество элементов в списке с помощью функции `size`. Затем мы получаем доступ к элементам одному за другим как показано на следующем примере.

```
for i = 1:size(mylist)
  e = mylist(i);
  mprintf("Element #%d: type=%s.\n",i,typeof(e))
end
```

Предыдущие примеры производят следующий вывод.

```
Element #1: type=constant.
Element #2: type=string.
Element #3: type=constant.
```

Это более простой способ, который использует напрямую список как аргумент команды `for`.

```
-->for e = mylist
-->  mprintf("Type=%s.\n",typeof(e))
-->end
Type=constant.
Type=string.
Type=constant.
```

Мы можем заполнить список динамически, добавляя новые элементы к концу. В следующем примере мы определяем пустой список с помощью команды `list()`. Затем, мы используем оператор `$+1` для введения новых элементов в конец списка. Это даёт точно такой же список, что и раньше.

```
mylist = list();
mylist($+1) = 12;
mylist($+1) = "foo";
mylist($+1) = [1 2 3 4];
```

3.6 Тип данных tlist

Типизированные списки позволяют определять новые структуры данных, которые могут быть настроены в зависимости от конкретных задач, которые необходимо решить. Эти новые структуры данных ведут себя подобно базовым типам данных Scilab'а. В частности, любая обычная функция, такая как `size`, `disp` или `string` может быть перегружена так, что у неё будет особое поведение, если её входным аргументом является новый `tlist`. Это позволяет расширить возможности, предоставляемые Scilab'ом и вводить новые объекты. На самом деле, типизированный список также используется внутри числовыми функциями Scilab'а из-за его гибкости. Бóльшую часть времени мы даже не знаем, что мы используем список, что показывает насколько этот тип данных удобен.

В этом разделе мы создадим и используем непосредственно `tlist` для получения близкой с ней структуры данных. В следующем разделе мы представим общую схему, которая позволит имитировать объектно-ориентированное программирование с типизированными списками.

Рисунок 16 представляет все функции, относящиеся к типизированным спискам.

`tlist`	создать типизированный список
`typeof`	получить тип данного типизированного списка
`fieldnames`	вернуть все поля типизированного списка
`definedfields`	вернуть все поля, которые определены
`setfield`	установить поле типизированного списка
`getfield`	получить поле типизированного списка

Рис. 16: Функции, относящиеся к `tlist`.

Для того, чтобы создать типизированный список, мы используем функцию `tlist`, чей первый аргумент является матрицей строковых элементов. Первая строка — это *тип* спис-

ка. Остальные строковые элементы определяют *поля* типизированного списка. В следующем разделе мы определим типизированный список `person`, который позволяет сохранить информацию о человеке. Поля `person` являются именем, фамилией и годом рождения.

```
-->p = tlist(["person","firstname","name","birthyear"])
 p  =
       p(1)
!person  firstname  name  birthyear  !
```

В этом месте создаётся `person p`, но его поля не определены. Для того, чтобы установить поля `p`, мы используем точку «`.`», которая ставится после имени поля. В следующем примере мы определим три поля, связанных с гипотетическим Полом Смитом (Paul Smith), который родился в 1997.

```
p.firstname = "Paul";
p.name = "Smith";
p.birthyear = 1997;
```

Все поля теперь определены и мы можем использовать переменное имя `p` для того, чтобы увидеть содержимое типизированного списка как в следующем примере.

```
-->p
 p  =
       p(1)
!person  firstname  name  birthyear  !
       p(2)
 Paul
       p(3)
 Smith
       p(4)
    1997.
```

Для того, чтобы получить поле типизированного списка, мы можем использовать синтаксис `p(i)`, как в следующем примере.

```
-->p(2)
 ans  =
 Paul
```

Но может быть более удобно получать значение поля `firstname` с помощью синтаксиса `p.firstname` как показано в следующем примере.

```
-->fn = p.firstname
 fn  =
 Paul
```

Мы также можем использовать функцию `getfield`, который принимает в качестве входных аргументов индексы полей и типизированный список.

```
->fn = getfield(2,p)
 fn  =
 Paul
```

Тот же синтаксис может быть использован для установки значения поля. В следующем примере мы обновляем значение имени с «Paul» на «John».

```
-->p.firstname = "John"
 p  =
       p(1)
!person  firstname  name  birthyear  !
```

```
       p(2)
 John
       p(3)
 Smith
       p(4)
    1997.
```

Мы также можем использовать функцию `setfield` для смены поля имени с «John» на «Ringo».

```
-->setfield(2,"Ringo",p)
-->p
 p  =
       p(1)
!person  firstname  name  birthyear  !
       p(2)
 Ringo
       p(3)
 Smith
       p(4)
    1997.
```

Может так случиться, что мы знаем значения полей во время создания типизированного списка. Мы можем добавить эти значения ко входным аргументам функции `tlist` в строгом порядке. В следующем примере мы определим `person p` и установим значения полей во время создания типизированного списка.

```
-->p = tlist( ..
-->  ["person","firstname","name","birthyear"], ..
-->  "Paul", ..
-->  "Smith", ..
-->  1997)
 p  =
       p(1)
!person  firstname  name  birthyear  !
       p(2)
 Paul
       p(3)
 Smith
       p(4)
    1997.
```

Интересной чертой типизированного списка является то, что функция `typeof` возвращает текущий тип списка. В следующем примере мы проверяем что функция `type` возвращает 16, что соответствует списку. Но функция `typeof` возвращает строку «person».

```
-->type(p)
 ans  =
    16.
-->typeof(p)
 ans  =
 person
```

Это позволяет динамически менять поведение функций для типизированных списков с типом «person». Эта возможность связана с перегрузкой функций, темой, которая будет рассмотрена в разделе 3.8.

Теперь мы рассмотрим функции, которые позволяют динамически получать информацию о типовых списках. Для того, чтобы получить список полей «person», мы можем

использовать синтаксис p(1) и получить матрицу строк размером 1×4, как в следующем примере.

```
-->p(1)
 ans  =
!person  firstname  name  birthyear  !
```

Тот факт, что первый строковый элемент представляет тип, может быть полезен или мешать, в зависимости от ситуации. Если мы только хотим получить поля типизированного списка (и не распечатывать), то мы можем использовать функцию fieldnames.

```
-->fieldnames(p)
 ans  =
!firstname  !
!           !
!name       !
!           !
!birthyear  !
```

Когда мы создаём типизированный список, мы можем определить его поля без установки его значений. Действующее значение поля может быть на самом деле позже установлено динамически в файле-сценарии. В этом случае было бы полезно знать, определено ли поле уже ли нет.

Следующий пример показывает как функция definedfields возвращает матрицу целых чисел с плавающей запятой, представляющую поля, которые уже определены. Мы начинаем с определения p человека без установки какого-либо значения любого из полей. Вот почему единственное определённое поле равно числу 1. Затем мы устанавливаем поле «firstname», которое соответствует индексу 2.

```
-->p = tlist(["person","firstname","name","birthyear"])
 p  =
       p(1)
!person  firstname  name  birthyear  !
-->definedfields(p)
 ans  =
    1.
-->p.firstname = "Paul";
-->definedfields(p)
 ans  =
    1.    2.
```

Как мы можем видеть, индекс 2 был добавлен к матрице с помощью определённых полей, возвращённых функцией definedfields.

Функции, которые мы рассмотрели, позволяют программировать типизированные списки очень динамичным образом. Теперь мы увидим как использовать функции definedfields для динамического вычисления определено ли поле, идентифицированное по его строке, или нет. Это позволит получить немного больше практики с типизированными списками. Вспомним, что мы можем создать типизированный список без действительного определения значений его полей. Эти поля могут быть определены позже, так что в конкретное время мы не знаем все ли поля определены или нет. Следовательно, может потребоваться функция isfielddef, которая бы вела себя как в следующем примере.

```
-->p = tlist(["person","firstname","name","birthyear"]);
-->isfielddef ( p , "name" )
 ans  =
  F
```

```
-->p.name = "Smith";
-->isfielddef ( p , "name" )
 ans  =
  T
```

Целью упражнения 3.2 является динамическое определение: существует ли в типизированном списке поле, связанное с заданным полем, определённым по его строке.

3.7 Имитация объектно-ориентированного программирования с помощью типизированных списков

В этом разделе мы рассмотрим как типизированные списки могут использоваться для имитации объектно-ориентированного программирования (ООП). Это обсуждение частично было представлено в вики Scilab [7].

Мы представляем простой метод имитирования ООП с текущими функциями Scilab'a. Предлагаемый метод является классическим, когда мы хотим имитировать ООП на не-ООП языке, например, на Си или фортране. В первой части мы анализировали ограничения функций, которые используют позиционные аргументы. Далее мы представляем метод имитации ООП в Scilab с помощью типизированных списков.

В заметках, связанных с этим разделом, мы представляем похожие методы на других языках. Мы подчёркиваем тот факт, что объектно-ориентированное программирование использовалось и используется десятками не-ООП языков, таких как Си или фортран, например, методами, которые очень похожи на тот, что мы собираемся представить.

3.7.1 Ограничения позиционных аргументов

Перед тем как перейти к деталям, мы сначала представим причины того, почему имитация ООП в Scilab'e удобна и, иногда, необходима. В самом деле, метод, который мы защищаем, может позволить упростить многие функции, которые основаны на необязательных, позиционных аргументах. Факт, что выполнение, основанное на позиции входных и выходных аргументов функции, является жёстким ограничением в некоторых случаях, как мы вскоре увидим.

Например, примитив `optim` является встроенной функцией, которая выполняет неограниченную и частично ограниченную числовую оптимизацию. Эта функция имеет 20 аргументов, некоторые из которых необязательные. Далее приводим заголовок функции, где квадратные скобки `[...]` обозначают необязательные параметры.

```
[f [,xopt [,gradopt [,work]]]]= ..
  optim(costf [,<contr>],x0 [,algo] [,df0 [,mem]] [,work] ..
  [,<stop>] [,<params>] [,imp=iflag])
```

Эта усложнённая последовательность вызова делает практическое использование `optim` сложным (но работоспособным), особенно если мы хотим настроить его аргументы. Например, переменная `<params>` является списком четырёх необязательных аргументов:

```
'ti', valti ,'td', valtd
```

Многие параметры алгоритма могут быть сконфигурированы, но, довольно удивительно, что многие не могут быть сконфигурированы пользователем функции `optim`. Например, в случае квази-Ньютоновского алгоритма без ограничений, процедуры фортрана позволяют сконфигурировать длину, представляющую оценку расстояния до оптимума. Этот параметр не может быть сконфигурирован на уровне Scilab и по умолчанию используется значение

0,1. Причина, которая кроется за этим выбором, очевидна: есть уже слишком много параметров для функции `optim` и добавление других необязательных параметров привело бы к невозможности пользования функцией.

Более того, пользователи и разработчики хотят добавлять новые возможности в примитив `optim`, но это может привести к нескольким трудностям.

- Расширение текущего шлюза `optim` очень сложно из-за сложного управления 20-ю внутренними необязательными аргументами. Более того, поддержка и разработка интерфейса в течение жизни проекта Scilab трудны, поскольку порядок аргументов имеет значение. Например, мы можем осознавать, что один аргумент может быть ненужным (из-за того, например, что эта же самая информация может быть получена через другую переменную). В этом случае мы не можем удалить аргумент из последовательности вызова, поскольку это сломает обратную совместимость всех файлов-сценариев, которые используют эту функцию.

- Трудно расширить список выходных аргументов. Например, нас может заинтересовать целое число, представляющее статус оптимизации (например, достигнута ли сходимость, достигнуто ли максимальное число итераций, максимальное число вызовов функций и т. д.). Мы могли бы быть также заинтересованы в числе итераций, числе вызовов функций, конечном значении аппроксимации матрицы Гессе, и в другой информации. Ограниченное число выходных аргументов на самом деле ограничивает количество информации, которую пользователь может вытянуть из имитации. Более того, если мы захотим получить выходной аргумент №6, например, то мы должны вызвать функцию со всеми аргументами от №1 до №6. Это может породить много ненужных данных.

- Мы могли бы захотеть установить необязательный входной аргумент №7, а не необязательный аргумент №6, который отсутствует в данном интерфейсе. Это из-за того, что система обработки аргументов основана на порядке аргументов.

В общем и целом, факт, что входные и выходные аргументы используются явно и на основе их последовательности, что очень неудобно когда число аргументов велико.

Если окружение объектно-ориентированного программирования было бы доступно для Scilab'а, то управление аргументами было бы решено с меньшими трудностями.

3.7.2 Класс «person» в Scilab

В этом разделе мы даём конкретный пример метода, основанного на разработке класса «person».

Метод, который мы представляем в этом документе, для имитации объектно-ориентированного программирования является классическим для других языков. На самом деле о общий для расширения необъектных языков в ООП-системе. Возможный метод:

- мы создаём тип абстрактных данных (ТАД) со структурами основных данных языка,

- мы имитируем методы с помощью функций, у которых первый элемент, по имени `this`, представляет текущий объект.

Мы будем использовать этот метод и имитировать в качестве примера отдельный класс «person».

Класс «person» сделан из следующих функций.

- Функция person_new, «конструктор», создаёт новый объект «person».
- Функция person_free, «деструктор», разрушает существующий объект «person».
- Функция person_configure и функция person_cget, позволяют сконфигурировать и опросить поля существующего объекта «person».
- Функция person_display, «метод», который отображает текущий объект в консоли.

В этих функциях текущий объект будет храниться в переменной this. Чтобы реализовать наш класс, мы используем типизированный список.

Следующая функция person_new возвращает this нового объекта «person». Этот новый «person» определён по своим фамилии, имени, номеру телефона и адресу электронной почты. Мы выбираем для использования пустые значения строк для всех полей.

```
function this = person_new ()
  this = tlist(["TPERSON","name","firstname","phone","email"])
  this.name=""
  this.firstname=""
  this.phone=""
  this.email=""
endfunction
```

Следующая функция person_free разрушает существующий объект «person».

```
function this = person_free (this)
  // Нет никаких действий.
endfunction
```

Поскольку сейчас нет никаких действий, то тело функции person_free пустое. По-прежнему, в силу разумных причин и из-за того, что реальное тело функции может развиться позже во время разработки компонента, мы создаём эту функцию в любом случае.

Мы подчёркиваем, что переменная this является как входным, так и выходным аргументом функции person_free. В самом деле, текущий объект «person», в принципе, является модифицируемым через работу функции person_free.

Функция person_configure позволяет сконфигурировать поле текущего объекта «person». Каждое поле идентифицируется строкой, «ключом», который соответствует значению. Следовательно, это просто отображение «ключ-значение». Функция устанавливает значение value, соответствующее заданному ключу key, и возвращает обновлённый объект this. Для класса «person» ключами являются "-name", "-firstname", "-phone" и "-email".

```
function this = person_configure (this,key,value)
  select key
  case "-name" then
    this.name = value
  case "-firstname" then
    this.firstname = value
  case "-phone" then
    this.phone = value
  case "-email" then
    this.email = value
  else
    errmsg = sprintf("Неизвестный ключ %s",key)
    error(errmsg)
  end
endfunction
```

Мы выбрали префиксацию каждого ключа знаком минус «-». В последовательностях вызова это позволит легко отличить ключ от значения.

Мы подчёркиваем, что переменная `this` является как входным, так и выходным аргументом функции `person_configure`. Она похожа на функцию `person_free`, которую мы создали прежде.

Аналогично, функция `person_cget` позволяет получить заданное поле текущего объекта «person». `person_cget` возвращает значение `value`, соответствующее заданному ключу `key` текущего объекта.

```
function value = person_cget (this,key)
  select key
  case "-name" then
    value = this.name
  case "-firstname" then
    value = this.firstname
  case "-phone" then
    value = this.phone
  case "-email" then
    value = this.email
  else
    errmsg = sprintf("Неизвестный ключ %s",key)
    error(errmsg)
  end
endfunction
```

Точнее, функция `person_cget` позволяет получить значение конфигурируемого ключа. «c» в «cget» относится к первой букве «configure» (*настроить*). Если, вместо этого, мы захотим создать функцию, которая возвращает значение ненастраиваемого ключа, мы можем назвать её `person_get`.

Теперь, когда наш класс установлен, мы можем создать новый «метод». Следующая функция `person_display` распечатать текущий объект `this` в консоли.

```
function person_display (this)
  mprintf("Person\n")
  mprintf("Name: %s\n", this.name)
  mprintf("First name: %s\n", this.firstname)
  mprintf("Phone: %s\n", this.phone)
  mprintf("E-mail: %s\n", this.email)
endfunction
```

Теперь мы представим простое использование класса «person», который мы только что разработали. В следующем примере мы создаём новый объект «person» с помощью вызова функции `person_new` . Затем мы вызовем функцию `person_configure` несколько раз для того, чтобы сконфигурировать различные поля объекта «person».

```
p1 = person_new();
p1 = person_configure(p1,"-name","Backus");
p1 = person_configure(p1,"-firstname","John");
p1 = person_configure(p1,"-phone","01.23.45.67.89");
p1 = person_configure(p1,"-email","john.backus@company.com");
```

В следующем примере мы вызовем функцию `person_display` и распечатаем текущий «person».

```
-->person_display(p1)
Person
Name: Backus
```

```
First name: John
Phone: 01.23.45.67.89
E-mail: john.backus@company.com
```

Мы можем также запросить имя текущего «person» вызовом функции `person_get`.

```
-->name = person_cget(p1,"-name")
 name  =
 Backus
```

Наконец, мы уничтожим текущий «person».

```
p1 = person_free(p1);
```

3.7.3 Расширение класса

В этом разделе мы обсудим способы расширения класса, основанные на имитации метода, который мы представили. Наша цель — возможность управлять более сложными компонентами с бо́льшим количеством полей, бо́льшим количеством методов или бо́льшим количеством классов.

Сперва мы подчеркнём, что управление опциями класса безопасно. На самом деле система, которую мы только что разработали, просто сопоставляет значение ключу. Следовательно, список ключей, определяется один раз для всех, пользователь не может конфигурировать или получать значение ключа, который не существует. Если мы попытаемся, то функции `person_configure` или `person_cget` сформируют ошибку.

Новый ключ в класс добавляется напрямую. Сначала мы должны обновить функцию `person_new`, добавив новое поле к типизированному списку. Мы можем решить, какое значение по умолчанию использовать для нового поля. Заметим, что существующие файлы-сценарии, использующие класс «person», будут работать. Если требуется, то файл-сценарий может быть обновлён для того, чтобы конфигурировать новый ключ значением не по умолчанию.

Мы можем решить различать публичные поля, которые могут быть сконфигурированы пользователем или классом, и частные поля, которые не могут. Для того, чтобы добавить новое частное поле в класс «person», скажем «bankaccount» (*банковский счёт*), мы изменяем функцию `person_new` и добавляем соответствующую строку в типизированный список. Следовательно, нам не нужно делать его доступным ни в функции `person_configure`, ни в функции `person_cget`, пользователь класса не сможет получить доступ к нему.

Мы можем быть немного гибче, позволяя пользователю класса получать значения полей без возможности изменения значения. В этом случае нам следует создать отдельную функцию `person_get` (заметим отсутствие буквы «c»). Это позволяет отделить конфигурируемые опции от неконфигурируемых опций.

Мы можем создать даже более сложные структуры данных с помощью вложения классов. Это легко, поскольку типизированные списки могут содержать типизированный список, который может содержать типизированный список, и т. д. до любого требуемого уровня вложения. Следовательно, мы можем имитировать ограниченное наследование, идея, которая является одним из главных вопросов в ООП.

К примеру, допустим, что мы хотим создать класс «company». Эта «company» определяется по её имени, адресу, назначению и списком всех людей («person»), работающих в ней. Следующая функция `company_new` предлагает возможную реализацию.

```
function this = company_new ()
  this = tlist(["TCOMPANY","name","address","purpose","employees"])
```

```
  this.name=""
  this.address=""
  this.purpose=""
  this.employees = list()
endfunction
```

Следующая функция company_addperson позволяет добавлять нового человека в список работников.

```
function this = company_addperson ( this , person )
  this.employees($+1) = person
endfunction
```

На самом деле, методы, которые широко используются в ООП, могут применяться используя эту схему имитации. Это позволяет создавать отдельные компоненты, предоставляющие прозрачный публичный интерфейс и избегающие необходимости в сложных функциях, использующих большое количество позиционных аргументов.

3.8 Перегрузка типизированных списков

В этом разделе мы увидим как перегрузить функцию string так, что бы мы могли преобразовать типизированный список в строку. Мы также рассмотрим как перегрузить функцию disp, т. е. систему печати, которая позволяет настраивать печать типизированных списков.

Следующая функция %TPERSON_string возвращает матрицу строковых значений, содержащую описание текущего объекта «person». Она сначала создаёт пустую матрицу. Затем она добавляет строки одну за другой с помощью выходного значения функции sprintf, которая позволяет форматировать строки.

```
function str = %TPERSON_string (this)
  str = []
  k = 1
  str(k) = sprintf("Person:")
  k = k + 1
  str(k) = sprintf("======================")
  k = k + 1
  str(k) = sprintf("Name: %s", this.name)
  k = k + 1
  str(k) = sprintf("First name: %s", this.firstname)
  k = k + 1
  str(k) = sprintf("Phone: %s", this.phone)
  k = k + 1
  str(k) = sprintf("E-mail: %s", this.email)
endfunction
```

Функция %TPERSON_string позволяет перегрузить функцию string для любого объекта с типом TPERSON.

Следующая функция %TPERSON_p распечатывает текущий объект «person». Она сначала вызывает функцию string для того, чтобы вычислить матрицу строк, описывающих объект «person». Затем она делает циклы по рядам матрицы и распечатывает их один за другим в консоли.

```
function %TPERSON_p ( this )
  str = string(this)
  nbrows = size(str,"r")
```

```
  for i = 1 : nbrows
    mprintf("%s\n",str(i))
  end
endfunction
```

Функция `%TPERSON_p` позволяет перегрузить распечатку объектов с типом `TPERSON` (буква «p» обозначает распечатку («print»)).

В следующем примере мы вызываем функцию `string` с объектом «person» `p1`.

```
-->p1 = person_new();
-->p1 = person_configure(p1,"-name","Backus");
-->p1 = person_configure(p1,"-firstname","John");
-->p1 = person_configure(p1,"-phone","01.23.45.67.89");
-->p1 = person_configure(p1,"-email","john.backus@company.com");
-->string(p1)
 ans  =
!Person:                          !
!                                 !
!======================           !
!                                 !
!Name: Backus                     !
!                                 !
!First name: John                 !
!                                 !
!Phone: 01.23.45.67.89            !
!                                 !
!E-mail: john.backus@company.com  !
```

В предыдущем примере функция `string` автоматически вызвала функцию `%TPERSON_string`, которую мы определили ранее.

В следующем примере мы просто напечатаем имя переменной `p1` (и немедленно нажмём клавишу enter в консоли), как и с любой другой переменной. Это распечатает содержимое нашей переменной `p1`.

```
-->p1
 p1  =
Person:
======================
Name: Backus
First name: John
Phone: 01.23.45.67.89
E-mail: john.backus@company.com
```

В предыдущем примере система печати автоматически вызвала функцию `%TPERSON_p`, которую мы уже определили ранее. Заметим, что будет сделан тот же вывод как если бы мы использовали команду `disp(p1)`.

Наконец, мы уничтожим текущий объект «person».

```
p1 = person_free(p1);
```

Есть много других функций, которые могли бы быть определены для типизированных списков. Например, мы можем перегрузить оператор `+` так, что мы можем суммировать два объекта «person». Это описано более подробно на страницах справки, посвящённых перегрузке:

```
help overloading
```

3.9 Тип данных mlist

В этом разделе мы представляем тип данных mlist, который является матрично-ориентированным списком. В первой части этого раздела мы представляем главную мотивацию для типа данных mlist сравнивая с tlist. Затем мы представляем пример mlist, где мы определяем функцию выделения.

Главная разница между tlist и mlist состоит в отношении к выделению и вставке функций. В самом деле, для tlist M выделение, основанное на индексах, т. е. команда x=M(2), например, определена по умолчанию. Она может быть перегружена пользователем, но это не обязательно, как мы увидим далее.

В следующем примере мы определяем tlist с типом «V». Этот типизированный список имеет два поля, «name» (*имя*) и «age» (*возраст*).

```
M=tlist(["V","name","age"],["a","b";"c" "d"],[1 2; 3 4]);
```

Как ожидалось от tlist, команда M(2) позволяет выделить второе поле, т. е. «name» (*имя*) переменной.

```
-->M(2)
 ans  =
!a  b  !
!      !
!c  d  !
```

То же самое для вставки (т. е. M(2)=x) в list (или tlist).

С другой стороны, для mlist выделение и вставка функций *должна* быть определена. Если нет, то формируется ошибка.

В следующем примере мы определяем mlist с типом «V». Как и прежде, этот список матриц имеет два поля, «name» и «age».

```
M=mlist(["V","name","age"],["a","b";"c" "d"],[1 2; 3 4]);
```

В следующем примере мы видим, что не можем напрямую выделять второе поле этого списка

```
-->M(2)
     !--error 144
Операция для заданных операндов не определена.

отметьте или определите функцию %l_e как перегружаемую.
```

Сообщение об ошибке говорит нам, что мы не можем выделить второй элемент переменной M и что должна быть определена некая функция, которую мы увидим позднее в этом разделе.

Давайте рассмотрим предыдущий пример и предположим, что M — это tlist. Особая проблема с M заключается в том, что второй элемент M(2) может не соответствовать тому, что нам надо. В самом деле, переменная M(2) является матрицей строковых переменных ["a","b";"c" "d"]. Но возможны такие ситуации, где нам бы хотелось выразить, что «второй элемент» представляет особый элемент, который не соответствует особому значению, которое связано с tlist.

К примеру, мы можем захотеть конкатенировать имя со значением для формирования строки «Name: b, age: 2», которая бы была вторым элементом нашей матрицы M. Следовательно, кажется, что нам нужно переопределить выделение (или вставку) значений для tlist. Но это невозможно, поскольку система перегрузки позволяет только определять функции, которые *не существуют*: в этом случае функция выделения уже существует, так что мы не можем определять её. Это случай, когда полезен mlist: мы можем определить

функции выделения и вставки для `mlist` и настроить их поведение. Фактически, мы даже принуждаем делать так: поскольку, как мы видели до этого, это обязательно.

Для того, чтобы определить функцию выделения для `mlist` с «V», мы должны определить функцию `%V_e`, где буква «e» обозначает «extraction» (*выделение*). Это сделано в следующем примере. Заголовок функции выделения должен быть `[x1,...,xm]=%<type_of_a>_e_(i1,...,in,a)`, где `x1,...,xm` — выделенные значения, `i1,...,in` — индексы значений, которые нужно выделить, и `a` — текущая переменная. Следовательно, команда `M=varargin($)` позволяет установить в `M` текущую переменную из которой выделяются значения. Затем индексы могут быть получены с помощью команды `varargin(1:$-1)`, которая создаёт список переменных, содержащий действующие индексы. Наконец, мы создаём матрицу строковых переменных конкатенацией имени со строкой, представляющей возраст.

```
function r=%V_e(varargin)
  M = varargin($)
  r= "Name: " + M.name(varargin(1:$-1)) + ..
       ", Age: " + string(M.age(varargin(1:$-1)))
endfunction
```

В следующем примере мы выделяем второй элемент `M`.

```
-->M(2)
 ans  =
 Name: c, Age: 3
```

Мы можем также использовать синтаксис, который очень похож на матрицы, как в следующем примере.

```
-->M(2:3)
 ans  =
!Name: c, Age: 3  !
!                 !
!Name: b, Age: 2  !
-->M(2,:)
 ans  =
!Name: c, Age: 3  Name: d, Age: 4  !
-->M(:,1)
 ans  =
!Name: a, Age: 1  !
!                 !
!Name: c, Age: 3  !
```

3.10 Тип данных `struct`

В этом разделе мы кратко представляем `struct`, которая является структурой данных, с беспорядочными составляющими разнородных типов. Мы сравним её характеристики с `list`.

Внутри `struct` все составляющие могут быть выделены по их именам. В следующем примере мы определяем переменную `d` как `struct` с тремя полями «day» (*день*), «month» (*месяц*) и «year» (*год*).

```
-->d=struct("day",25,"month" ,"DEC","year",2006)
 d  =
   day: 25
   month: "DEC"
   year: 2006
```

Заметьте, что поля структуры `struct` не обязаны быть одного типа: первое поле — это строка, а второе поле число типа double. Для того, чтобы выделить переменную из `struct`, мы просто используем имя переменной после точки «.» и имя поля.

```
-->d.month
 ans  =
 DEC
```

Для того, чтобы вставить значение в `struct`, мы просто используем оператор присвоения «=».

```
-->d.month=AUG
 d  =
   day: 25
   month: "AUG"
   year: 2006
```

Структура `struct` может содержать другую структуру `struct`, что может привести к структурам вложенных данных. Например, в следующем примере мы создаём уик-энд из двух последовательных дней.

```
-->d1=struct("day",01,"month" ,"JAN","year",2011);
-->d2=struct("day",02,"month" ,"JAN","year",2011);
-->weekend = struct("Sat",d1,"Sun",d2)
 weekend  =
   Sat: [1x1 struct]
   Sun: [1x1 struct]
-->weekend.Sat
 ans  =
   day: 01
   month: "JAN"
   year: 2011
```

С точки зрения гибкости `tlist` более мощный нежели `struct`. Это из-за того, что мы можем перегружать функции так, что их поведение может быть настроено для `tlist`. Действительно, настройка поведения структуры `struct` невозможна. Вот почему большинство пользователей предпочитают тип данных `tlist`.

Фактически, главное преимущество `struct` — это совместимость с Matlab и Octave.

3.11 Массив структур `struct`

Массив структур `struct` — это массив, где каждый индекс связан со структурой `struct`. Есть несколько способов создать массив структур. В первой части этого раздела мы конкатенируем несколько структур для создания массива. Во второй части мы инициализируем целый массив структур за один вызов, а затем заполняем поля одно за другим.

Если несколько структур имеют одинаковые поля, и если все эти поля имеют одинаковый размер, то они могут быть конкатенированы в один массив структур. В следующем примере мы создаём четыре отдельных структуры.

```
s1=struct("firstname","John","birthyear",1940);
s2=struct("firstname","Paul","birthyear",1942);
s3=struct("firstname","George","birthyear",1943);
s4=struct("firstname","Ringo","birthyear",1940);
```

Заметим, что поля не обязаны быть одного типа: первое поле — строка, а второе поле — число типа double. Затем мы конкатенируем их в один массив структур, используя синтаксис `[]`, который схож с матрицами.

```
-->s=[s1,s2,s3,s4]
 s  =
1x4 struct array with fields:
   firstname
   name
```

Мы можем выделить третью структуру из массива используя синтаксис `s(3)`, который схож с матрицами.

```
-->s(3)
 ans  =
   firstname: "George"
   name: "Harrisson"
```

Мы можем получить доступ к полю «firstname» всех структур, как показано в следующем примере. Это даёт список строк.

```
-->s.firstname
 ans  =
       ans(1)
 John
       ans(2)
 Paul
       ans(3)
 George
       ans(4)
 Ringo
-->typeof(s.firstname)
 ans  =
 list
```

По причине производительности мы не советуем позволять структура расти динамически. Это из-за того, что это заставляет интерпретатор динамически распределять больше и больше памяти и может привести к медленным файлам-сценариям. Вместо этого, где только возможно, нам следует предопределять массивы структур, а затем заполнять существующие ячейки.

В следующем примере мы определяем массив размером 4×1 структур `struct`, где каждая структура содержит пустое `firstname` и пустое `name`.

```
t(1:4)=struct("firstname",[],"birthyear",[])
```

Затем мы заполняем каждую структуру как в следующем примере.

```
t(1).firstname="John";
t(1).birthyear=1940;
t(2).firstname="Paul";
t(2).birthyear=1942;
t(3).firstname="George";
t(3).birthyear=1943;
t(4).firstname="Ringo";
t(4).birthyear=1940;
```

Мы можем проверить, что это даёт в точности тот же массив структур, что и прежде.

```
-->t
 t  =
4x1 struct array with fields:
   firstname
   birthyear
-->t(1)
```

```
ans  =
   firstname: "John"
   birthyear: 1940
```

В предыдущим примерах мы видели только массивы с одним индексом. Должно быть ясно, что массив структур является в действительности 2^x-индексным массивом, т. е. матрицей. Кстати, мы собираемся рассмотреть другой метод инициализации массива структур.

В следующем примере мы инициализируем массив структур размером 2×2.

```
-->u(2,2).firstname=[]
 u  =
2x2 struct array with fields:
   firstname
-->u(2,2).birthyear=[]
 u  =
2x2 struct array with fields:
   firstname
   birthyear
```

Затем мы можем заполнить ячейки с помощью синтаксиса, который похож на матрицы.

```
u(1,1).firstname="John";
u(1,1).birthyear=1940;
u(2,1).firstname="Paul";
u(2,1).birthyear=1942;
u(1,2).firstname="George";
u(1,2).birthyear=1943;
u(2,2).firstname="Ringo";
u(2,2).birthyear=1940;
```

Сделав однажды, мы можем выделять структуры, связанные с индексами (2,1), например.

```
-->u(2,1)
 ans  =
   firstname: "Paul"
   birthyear: 1942
```

3.12 Тип данных cell

В этом разделе мы кратко представляем `cell`, который является разнородным массивом переменных. Затем мы сравниваем его характеристики с характеристиками гиперматрицы `hypermatrix` и списка `list`. Рисунок 17 представляет несколько функций, связанных с массивами `cell`.

`cell`	Создать cell-массив с пустыми ячейками.
`cell2mat`	Преобразовать cell-массив в матрицу.
`iscell`	Проверить, является ли переменная cell-массивом.
`makecell`	Создать cell-массив и инициализировать его ячейки.

Рис. 17: Функции, связанные с массивами `cell`.

Функция `cell` позволяет создавать массив. Этот `cell`-массив может содержать другие типы переменных, включая double, integer, string и т. д. В следующем примере мы создаём cell-массив размером 2×3.

```
-->c = cell(2,3)
 c  =
!{}  {}  {}  !
!            !
!{}  {}  {}  !
```

Размер cell-массива может быть вычислен с помощью функции `size`.

```
-->size(c)
 ans  =
    2.    3.
```

Для того, чтобы ввести значение в cell-массив, мы не можем использовать тот же синтаксис, что и для матриц.

```
-->c(2,1)=12
 !--error 10000
Invalid assignement: for insertion in cell, use e.g. x(i,j).entries=y
at line       3 of function generic_i_ce called by :
at line       3 of function %s_i_ce called by :
c(2,1)=12
```

Вместо этого мы можем использовать поле `entries` элемента (2,1), как в следующем примере.

```
-->c(2,1).entries=12
 c  =
!{}  {}  {}  !
!            !
!12  {}  {}  !
```

В следующем примере мы вводим строку в элемент (1,3).

```
-->c(1,3).entries="5"
 c  =
!{}  {}  "5"  !
!             !
!12  {}  {}   !
```

Для того, чтобы удалить элемент (1,3), мы можем использовать матрицу `[]`.

```
-->c(1,3).entries=[]
 c  =
!{}  {}  {}  !
!            !
!12  {}  {}  !
```

Мы можем выделять часть cell-массива, используя синтаксис, схожий с матрицами. Заметим, что результатом является cell-массив.

```
-->x=c(1:2,1)
 x  =
!{}  !
!    !
!12  !
-->typeof(x)
 ans  =
 ce
```

С другой стороны, когда мы выделяем одну конкретную ячейку с помощью поля `entries`, мы получаем тот же тип данных, что и в данная конкретная ячейка. В следующем примере мы выделяем ячейку (2,1) в переменную `x` и проверяем, что эта переменная имеет тип double.

```
-->x = c(2,1).entries
 x  =
    12.
-->typeof(x)
 ans  =
 constant
```

Если мы выделим несколько ячеек за раз с помощью поля `entries`, то мы получим список. В следующем примере мы выделяем все ячейки в первом столбце.

```
-->x = c(1:2,1).entries
 x  =
         x(1)
       []
         x(2)
    12.
-->typeof(x)
 ans  =
 list
```

Мы можем создать многоиндексные массивы с помощью `cell`-массивов. Например, команда `c=cell(2,3,4)` создаёт массив размером $2 \times 3 \times 4$. Эту особенность можно будет сравнить с гиперматрицами с тем дополнительным преимуществом, что ячейки `cell` могут иметь различные типы, а все ячейки гиперматрицы должны быть одного типа.

Поведение `cell` можно также сравнить со списком `list`. Но к ячейкам `cell` можно получить доступ с помощью синтаксиса, похожего на матрицы, что может быть удобно в некоторых ситуациях.

Одна из особенностей `cell` заключается в том, что он частично совместим с Matlab и Octave. В Scilab'е `cell` ведёт себя не полностью идентично, та что эта совместимость только частичная. Например ячейка (2,1) `cell`-массива может быть выделена с помощью синтаксиса `c{2,1}` (заметьте «{}»): этот синтаксис недоступен в Scilab.

Синтаксис для выделения значений из массива `cell` может быть особенно полезным в некоторых ситуациях. Эта особенность разделяется со списками `tlist` и `mlist`, где выделение может быть перегружено.

3.13 Сравнение типов данных

В этом разделе мы сравним различные типы данных, которые мы представили. На рисунке 18 представлен обзор этих типов данных.

Рисунок 19 представляет различные типы данных, которые мы уже встретили, и предлагает другие типы данных для замены.

Замена одного типа данных другим может быть интересна, например в ситуации, где производительность играет значение. В этой ситуации эксперименты с различными реализациями и измерения производительности каждой из них может помочь улучшить производительность данного алгоритма.

Некоторые типы данных разнородны, т. е. эти типы данных могут содержать переменные, которые могут быть разных типов. Это ситуации структуры `struct`, матрично-ориентированного списка `mlist` и массива `cell`. Другие типы данных могут содержать только один тип переменных: это ситуация матрицы и гиперматрицы. Если все переменные, чей тип данных должен управляться, имеют один и тот же тип, то без сомнения следует выбирать матрицу или гиперматрицу.

Тип данных	Назначение	Преимущества	Недостатки
matrix	2^x-индексная матрица	эффективная, простая	однородная
hypermatrix	многоиндексная матрица		однородная
`list`	Упорядоченный набор элементов		не перегружается
`tlist`	типизированный список	можно перегружать	
`mlist`	матрично-ориентированный список	выделение/вставка, может быть определена пользователем.	
`struct`	Неупорядоченный набор элементов	совместим с Matlab	не перегружается
`cell`		частично совместим	не перегружается

Рис. 18: Структуры данных Scilab.

Типы данных	Реализация	Разнородность	Вложенность
matrix	встроено	Нет	Нет
hypermatrix	`mlist`	Нет	Нет
`list`	встроено	Да	Да
`tlist`	встроено	Да	Да
`mlist`	встроено	Да	Да
`struct`	`mlist`	Да	Да
`cell`	`mlist`	Да	Да

Рис. 19: Сравнение типов данных. Столбец «Реализация» указывает на реализацию этих типов данных в Scilab версии 5.

Вложение может быть полезным, когда мы хотим создать дерево структур данных. В этом случае ни матрица, ни гиперматрица не могут использоваться. Это действительно один из главных практических применений списка `list`. Столбец «Вложенность» на рисунке 19 представляет эту функциональность для всех структур данных.

Одна из причин, которой можно объяснить разницу производительностей заключается в реализации различных типов данных. Это представлено в столбце «Реализация» на рисунке 19 где мы представляем реализацию этих типов данных в Scilab версии 5.

Например, мы представляли гиперматрицы в разделе 3.3. В разделе 3.4 мы уже анализировали влияние производительности, которую может иметь выделение из гиперматриц. В Scilab версии 5 гиперматрицы реализованы через `mlist`, поддерживаемые как скомпилированным исходным кодом, так и макросами Scilab'а. Например, выделение гиперматриц чисел типа double основано на скомпилированном исходном коде, а выделение гиперматриц строковых переменных основано на макросе: это может также привести к различиям производительностей.

3.14 Заметки и ссылки

В первой части данного раздела мы представляем некоторые заметки и ссылки по теме имитации ООП в Scilab'е. Во второй части мы описываем связь между типом данных polynomial и функциями управления в Scilab'е.

Структура абстрактных данных для имитации ООП зависит от языка.

- В Си структура абстрактных данных выбора является структурой. Этот метод расширения Си частот используется для обеспечения общей схемы ООП [47, 45] когда использование C++ нежелательно или невозможно. Этот метод уже используется внутри исходного кода Scilab, в графическом модуле, для конфигурации свойств графики. Он известен как «handles» (*дескрипторы*): смотри заголовочный файл ObjectStructure.h [43].

- В Fortran 77 достаточна команда «common» (но она редко используется для имитации ООП).

- В Fortran 90 был разработан «производный тип» для этих целей (но редко используется для имитации истинного ООП). Стандарт Fortran 2003 — это Fortran с реальным ООП, основанным на производных типах.

- В Tcl, структура данных «array» является подходящей ТАД (он используется массивом STOOOOP [19] например). Но это можно также сделать с командой «variable», объединённой с пространствами имён (это сделано, к примеру, в пакете SNIT [17])

«Новый» конструктор имитируется возвратом образца ТАД. «Новый» метод может потребовать выделения памяти. «Свободный» деструктор берёт «this» в качестве первого аргумента и освобождает память которая была выделена ранее. Этот подход возможен в Фортране, Си и других компилируемых языках, если первый аргумент «this» может быть модифицирован методом. В Си это сделано переводом указателя на ТАД. В Fortran 90 это сделано добавлением «intent(inout)» к декларации (или ничего вообще).

Теперь обсудим связь между типом данных polynomial и управлением. Когда прежний открытый проект Matlab рассматривался исследователями в IRIA (French Institute for Research in Computer Science and Control) в начале 80^{х} годов, то они хотели создать программное обеспечение компьютеризированного проектирования систем управления — Computer Aided Control System Design (C.A.C.S.D.). В то время главными разработчиками были Франсуа Делебек (François Delebecque) и Серж Стир (Serge Steer). В контексте теории управления мы анализируем поведение динамических систем. В случае систем управления с одним входом и одним выходом (SISO), мы можем использовать преобразование Лапласа над переменными, что приводит к передаточной функции. Следовательно, тип данных rational был введён для поддержки анализа передаточных функций полученных для линейных систем. Есть много других функций, относящихся к CACSD в Scilab и представлять их не входит в задачи данного документа.

3.15 Упражнения

Упражнение 3.1 (***Поиск файлов***) Интерпретируемые языки часто используются для задач автоматизации, таких как поиск и переименование файлов на жёстком диске. В этой ситуации может быть использована функция `regexp`, чтобы обнаружить файлы с совпадением с заданным шаблоном. Например, мы можем использовать её для поиска файлов-сценариев Scilab в папке. Это легко, поскольку файлы-сценарии Scilab имеют расширения «.sci» (для файлов, определяющих функцию) и «.sce» (для файлов, выполняющих инструкции Scilab).

Разработайте функцию `searchSciFilesInDir`, которая ищет эти файлы в заданной директории. Мы предлагаем дать следующий заголовок.

```
function filematrix = searchSciFilesInDir ( directory , funname )
```

Если такой файл найдётся, мы добавим его в матрицу строковых переменных `filematrix`, которая возвращается в качестве выходного аргумента. Более того, мы вызовем функцию `funname`. Это позволит нам обработать файл если потребуется. Следовательно, мы можем использовать эту функциональность для отображения имени файла, перемещения файла в другую папку, удаления его и т. д.

Разработайте функцию `mydisplay` со следующим заголовком:

```
function mydisplay ( filename )
```

и используйте эту функцию совместно с `searchSciFilesInDir` для того, чтобы распечатать файлы в директории.

Упражнение 3.2 (***Запрос типизированных списков***) Функции, которые мы рассмотрели, позволяют программировать типизированные списки очень динамичным образом. Мы посмотрим сейчас как использовать функции `definedfields` для динамического вычисления: определено ли поле, идентифицированное по его строке, или нет. Это позволит получить немного больше практики с типизированными списками. Вспомните, что мы можем создавать типизированные списки без действительного определения значений полей. Эти поля могут быть определены позже, так что, в отдельный момент времени, мы не знаем были ли все поля определены или нет. Следовательно, нам может потребоваться функция `isfielddef`, которая бы вела себя как в следующем примере.

```
-->p = tlist(["person","firstname","name","birthyear"]);
-->isfielddef ( p , "name" )
 ans  =
  F
-->p.name = "Smith";
-->isfielddef ( p , "name" )
 ans  =
  T
```

Напишите реализацию функции `isfielddef`. Чтобы это сделать, вы можете объединить функции `find` и `definedfields`.

4 Управление функциями

В этом разделе мы рассмотрим управление функциями и представим возможности разработки гибких функций. Мы представляем методы получения информации о функциях и отделения макросов от примитивов. Мы обращаем внимание, что функции не зарезервированы в Scilab'е и предупреждаем об ошибках функции `funcprot`. Мы представляем использование вызовов, которые позволяет пользователю функции настроить часть алгоритма. Мы анализируем методы разработки функций с переменным числом входных и выходных переменных на основе команд `argn`, `varargin` и `varargout`. Мы покажем общие способы обеспечения значений по умолчанию для входных аргументов. Мы представляем как использовать пустую матрицу `[]` для преодоления проблемы, вызванной позиционными входными аргументами. Мы также рассматриваем определение функций, где выходные аргументы могут иметь различные типы. Затем мы представляем возможности, которые позволяют разрабатывать надёжные функции. Мы представляем практические примеры для функций `error`, `warning` и `gettext`. Мы представляем модуль `parameters`, который позволяет решить задачу проектирования функции с большим числом параметров. Детально анализируется обзор переменных через стек вызовов. Ошибки, вызванные плохим использованием этой возможности анализируются на основе типичных случаев использования. Мы представляем ошибки с вызовами и анализируем методы разрешения их. Мы представляем метод, основанный на списках, для обеспечения дополнительных аргументов для вызовов. Наконец, мы представляем инструменты мета-программирования, основанных на функциях `execstr` и `deff`.

4.1 Продвинутое управление функциями

В этом разделе мы представляем продвинутое управление функциями. В первой части мы представляем различия между макросами и примитивами. Затем мы обращаем внимание на то, что функции не зарезервированы, что подразумевает возможность переопределить функцию, которая уже существует. Мы покажем, что это вызвано тем фактом, что функции являются переменными. Мы представляем функцию `funcprot` и покажем как использовать вызовы функций.

4.1.1 Как сделать запрос о функции

В этом разделе мы представляем различия между макросами и примитивами. Мы анализируем различные значения, возвращённые функциями `type` и `typeof` для входных аргументов функций. Мы вводим функцию `deff`, которая позволяет динамически определять новую функцию на основе строк. Наконец, мы представляем функцию `get_function_path`, которая возвращает путь к файлу, определяющему макрос.

Есть два главных типа функций:

- макросы, которые написаны на языке Scilab,
- примитивы, которые написаны на компилируемом языке, таком, как, например, C, C++ или Fortran.

В большинстве случаев нет прямого способа найти различия между этими двумя типами функций, и эта хорошая черта: когда мы используем функцию, то не заботимся о языке, на котором эта функция написана, поскольку только результат имеет значение. Вот почему мы обычно используем имя *function* без дополнительных подробностей. Однако, иногда это важно, например, когда мы хотим проанализировать или отладить конкретную функцию: в этом случае необходимо знать откуда функция приходит.

Есть несколько возможностей, которые позволяют запрашивать информацию о конкретной функции. В этом разделе мы сосредотачиваемся на функциях `type`, `typeof`, `deff` и `get_function_path`.

Тип макроса равен или 11 или 13, а тип примитива равен 130. Эти типы сведены на рисунке 20.

type	typeof	Описание
11	"function"	Некомпилированный макрос
13	"function"	Компилированный макрос
130	"fptr"	Примитив

Рис. 20: Различные типы функций.

В следующем примере мы определяем функцию командой `function`. Затем мы вычисляем её тип с помощью функций `type` и `typeof`

```
-->function y = myfunction ( x )
-->  y = 2 * x
-->endfunction
-->type(myfunction)
 ans  =
```

```
    13.
-->typeof(myfunction)
 ans  =
 function
```

Функция `eye` является примитивом, который возвращает единичную матрицу. В следующем примере мы упражняемся с функциями `type` и `typeof` с входным аргументом `eye`.

```
-->type(eye)
 ans  =
    130.
-->typeof(eye)
 ans  =
 fptr
```

Функция `deff` позволяет динамически определять функцию, основанную на строках, представляющих её определение. Это позволяет динамически определять новую функцию, например для инкапсуляции одной функции в другую. Функция `deff` принимает необязательный входной аргумент, который позволяет изменить то, как функция обрабатывается интерпретатором. Такая функция может быть скомпилирована, скомпилирована и профилирована или некомпилирована. Процесс компиляции позволяет формировать более быстрый байт-код, который может быть использован интерпретатором непосредственно без дополнительный операций. Но она так же делает отладку невозможной и поэтому эта возможность может быть отключена. Мы рассмотрим `deff` более тщательно в разделе 4.7 этого документа

По умолчанию функция `deff` создаёт компилированный макрос. В следующем примере мы определим некомпилированный макрос с помощью функции `deff`. Первый аргумент функции `deff` является заголовком функции. В нашем случае функция `myplus` принимает два входных аргумента `y` и `z` и возвращает выходной аргумент `x`. Второй аргумент функции `deff` — это строка, определяющая тип функции, которую надо создать. Мы можем выбирать между `"c"` для компилированного макроса, `"p"` для компилированного и профилированного макроса и `"n"` для некомпилированного макроса.

```
-->deff("x=myplus(y,z)","x=y+z","n")
-->type(myplus)
 ans  =
    11.
-->typeof(myplus)
 ans  =
 function
```

Предыдущий пример позволяет проверить что тип некомпилированной функции равен 11.

Когда функция предоставлена в *библиотеке*, то функция `get_function_path` позволяет получить путь к функции. Например, модуль *оптимизации*, предоставляемый Scilab'ом содержит как примитивы (например функцию `optim`) и макросы (например функцию `derivative`). Макросы оптимизации собраны в библиотеку *optimizationlib*, которая анализируется в следующем примере.

```
-->optimizationlib
 optimizationlib  =

Расположение файлов функций: SCI\modules\optimization\macros\.

 aplat     bvodeS    datafit   derivative fit_dat   karmarkar
 leastsq   list2vec  lmisolver lmitool    NDcost    numdiff
 pack      pencost   qpsolve   recons    unpack     vec2list
```

Предыдущий пример не говорит нам является ли любая из функций, предоставленных `optimizationlib`, макросом или примитивом. Для этой цели мы можем вызвать функцию `typeof`, как в следующем примере, который показывает, что функция `derivative` является макросом.

```
-->typeof(derivative)
 ans  =
 function
```

В следующем примере мы вычисляем путь, ведущий к функции `derivative` используя функции `get_function_path` и `editor` для редактирования этого макроса.

```
-->get_function_path("derivative")
 ans  =
 D:/Programs/SCILAB~1.1-B\modules\optimization\macros\derivative.sci
-->editor(get_function_path("derivative"))
```

Заметим, что функция `get_function_path` не работает, когда входной аргумент является функцией, предоставляемой на компилируемом языке, например, функция `optim`.

```
-->typeof(optim)
 ans  =
 fptr
-->get_function_path("optim")
WARNING:  "optim" is not a library function
 ans  =
     []
```

4.1.2 Функции не зарезервированы

Можно переопределять функции, которые уже существуют. Часто, это является ошибкой, которая заставляет Scilab формировать сообщение об ошибке. В следующем примере мы определим функцию `rand` как обычную функцию и проверим, что мы можем вызвать её как любую другую, определённую пользователем, функцию.

```
-->function y = rand(t)
-->  y = t + 1
-->endfunction
Предупреждение : переопределение функции: rand.
  Используйте funcprot(0) чтобы не выводить это сообщение
-->y = rand(1)
 y  =
    2.
```

Предупреждение о том, что мы переопределили функцию `rand` должно заставить нас почувствовать себя некомфортно с нашим файлом-сценарием. Оно говорит нам, что функция `rand` уже существует в Scilab, что может быть легко проверено командой `help rand`. Действительно, встроенная функция `rand` позволяет генерировать случайные числа, и мы, конечно же, не хотим переопределять эту функцию.

В данном случае ошибка очевидна, но практические ситуации могут быть гораздо более сложными. Например, мы можем использовать сложное разветвлённое дерево функций, где функции очень низкого уровня вызывают это предупреждение. Изучая функцию, имеющую ошибку, и запуская её интерактивно в большинстве случаев можно найти ошибку. Более того, поскольку в Scilab'е есть много существующих функций, то вероятно создание новой программы изначально породит конфликт имён. В любом случае, мы должны исправить

эту ошибку без переопределения какой-нибудь существующей функции. Мы рассмотрим этот вопрос более детально в следующем разделе, где мы представляем пример, в котором временное отключение защиты функции действительно необходимо. То есть мы представляем функцию `funcprot`, которая упомянута в сообщении о предупреждении в предыдущем примере.

4.1.3 Функции — это переменные

В этом разделе мы покажем, что функции является переменными и представляем функцию `funcprot`.

Мощная возможность языка в том, что функции являются переменными. Это подразумевает то, что мы можем хранить функции в переменных и использовать переменные как функции. В компилируемых языках эта возможность часто называется как «указатели на функцию».

В следующем примере мы определяем функцию `f`. Затем, мы устанавливаем содержимое переменной `fp` равным функции `f`. Наконец, мы можем использовать функцию `fp` как обычную функцию.

```
-->function y = f ( t )
-->  y = t + 1
-->endfunction
-->fp = f
 fp  =
[y]=fp(t)
-->fp ( 1 )
 ans  =
    2.
```

Эта возможность позволяет нам использовать широко распространённый инструмент программирования, известный как «функции обратного вызова» («callback»).

Функция обратного вызова — это указатель функции, который может быть сконфигурирован пользователем компонента. Как только компонент сконфигурирован, он может производить обратный вызов функции, определённой пользователем так, что алгоритм может иметь, по крайней мере частично, настраиваемое поведение.

Поскольку функции являются переменными, мы можем установить значение переменной несколько раз. Фактически, для того, чтобы защитить пользователей от нежелательного переопределения функций, может появляться сообщение-предупреждение, как показано в следующем примере.

Мы начнём с определения двух функций `f1` и `f2`.

```
function y = f1 ( x )
  y = x^2
endfunction
function y = f2 ( x )
  y = x^4
endfunction
```

Затем мы можем установить переменную `f` дважды, как в следующем примере.

```
-->f = f1
 f  =
[y]=f(x)
-->f = f2
Предупреждение : переопределение функции: f .
```

```
 Используйте funcprot(0) чтобы не выводить это сообщение
 f  =
[y]=f(x)
```

Мы уже видели это предупреждение в разделе 4.1.2, в случае, где мы пытались переопределить встроенную функцию. Но в этой ситуации нет причин защищать себя от установки переменной `f` в новое значение. К сожалению нет у Scilab'а нет возможности различить эти две ситуации. К счастью есть простой способ отключить на время предупреждение. Функция `funcprot`, представленная на рисунке 21, позволяет конфигурировать режим защиты функций.

prot=funcprot()	Получить текущий режим защиты функций
funcprot(prot)	Установить режим защиты функций
prot==0	нет сообщений, когда функция переопределена
prot==1	предупреждение, когда функция переопределена (по умолчанию)
prot==2	ошибка, когда функция переопределена

Рис. 21: Функция `funcprot`.

Вообще, не рекомендуется конфигурировать режим защиты функций *долговременно*. Например, не следует писать команду `funcprot(0)` в файле запуска `.scilab`. Это из-за того, что мы не будет получать предупреждения, и, следовательно, можем использовать файлы-сценарии, содержащие ошибки, не зная об этом.

Но, для того, чтобы отключить ненужные сообщения предыдущего примера, мы можем *временно* отключить защиту. В следующем примере мы дублирование режима защиты функции, устанавливаем его временно равным нулю и затем восстанавливаем режим равным дублированному значению.

```
-->oldfuncprot = funcprot()
 oldfuncprot  =
    1.
-->funcprot(0)
-->f = f2
 f  =
[y]=f(x)
-->funcprot(oldfuncprot)
```

4.1.4 Функции обратного вызова

В этом разделе мы представляем метод управления функциями обратного вызова, то есть, мы рассматриваем случай, где входной аргумент функции сам является функцией. Например, мы рассматриваем вычисление численных производных конечными разностями.

В следующем примере мы используем встроенную (built-in) функцию `derivative` для вычисления производной функции `myf`. Мы сначала определяем функцию `myf`, которая возводит в квадрат свой входной аргумент `x`. Затем мы передаём функцию `myf` функции `derivative` в качестве обычного аргумента для вычисления численной производной в точке `x=2`.

```
-->function y = myf ( x )
-->  y = x^2
-->endfunction
```

```
-->y = derivative ( myf , 2 )
 y  =
    4.
```

Чтобы понять поведение функций обратного вызова, мы должны реализовать нашу упрощённую функцию численной производной, как это сделано в следующем примере. Наша реализация численной производной основана на разложении функции в ряд Тейлора первого порядка в соседстве с точкой `x`. Она принимает в качестве входных аргументов функцию `f`, точку `x`, где численная производная должна быть вычислена и шаг `h`, который должен использоваться.

```
function y = myderivative ( f , x , h )
  y = (f(x+h) - f(x))/h
endfunction
```

Подчеркнём, что это пример, который не надо использовать на практике, поскольку встроенная функция `derivative` гораздо более мощная, чем наша, упрощённая функция `myderivative`.

Заметим, что в теле функции `myderivative` входной аргумент `f` используется как обычная функция.

В следующем примере мы используем переменную `myf` в качестве входного аргумента функции `myderivative`.

```
-->y = myderivative ( myf , 2 , 1.e-8 )
 y  =
    4.
```

Мы сделали предположение, что функция `f` имеет заголовок `y=f(x)`. Нас может удивить что будет, если это не так.

В следующем примере мы определяем функцию `myf2`, которая берёт в качестве входных аргументов как `x` так и `a`.

```
function y = myf2 ( x , a )
  y = a * x^2
endfunction
```

Следующий пример показывает что происходит, если мы попытаемся вычислить численную производную функции `myf2`.

```
-->y = myderivative ( myf2 , 2 , sqrt(%eps) )
 !--error 4
Неизвестная переменная: a
at line       2 of function myf2 called by :
at line       2 of function myderivative called by :
y = myderivative ( myf2 , 2 , sqrt(%eps) )
```

Мы по-прежнему хотим вычислить численную производную нашей функции, даже если она имеет два аргумента. Мы увидим в разделе 4.5.1 как обзор переменных может быть использован, чтобы позволить функции `myf2` узнать о величине аргумента `a`. Метод программирования, который мы рассмотрим не считается «чисто программной» практикой. Вот почему мы представим в разделе 4.6.4 метод управления функциями обратного вызова с дополнительными аргументами.

4.2 Разработка гибких функций

В этом разделе мы представляем разработку функций с переменным числом входных и выходных переменных. Этот раздел обозревает функцию `argn` и переменные `varargin`

и varargout. Как мы увидим, указание переменного числа входных аргументов позволяет упростить использование функции, давая значения по умолчанию для аргументов, которые не устанавливаются пользователем.

Функции, относящиеся к этому вопросу представлены на рисунке 22.

argn	Возвращает текущее число входных и выходных аргументов.
varargin	Список, хранящий входные аргументы.
varargout	Список, хранящий выходные аргументы.

Рис. 22: Функции, имеющие отношение к переменному числу входных и выходных переменных.

В первой части мы анализируем простой пример, который позволяет нам видеть в действии функцию argn. Во второй части мы рассматриваем реализацию функции, которая вычисляет численные производные данной функции. Затем мы описываем как решить проблему, порождённую упорядоченными входными переменными, с помощью особого использования синтаксиса пустой матрицы. В заключительной части мы представляем функцию, чьё поведение зависит от типа её входных аргументов.

4.2.1 Обзор функции argn

В этом разделе мы делаем обзор функции argn и переменных varargin и varargout. Мы также представляем простую функцию, которая позволяет понять как выполняются эти функции.

Мы начинаем анализировать основы управления функций с переменным числом входных и выходных аргументов. Типичный заголовок таких функций следующий.

```
function varargout = myargndemo ( varargin )
  [lhs,rhs]=argn()
  ...
```

Аргументы varargout и varargin являются списками list, которые представляют входные и выходные аргументы. В телу определения функции, мы вызываем функцию argn, которая формирует две следующих выходных переменных.

- lhs, число выходных переменных,
- rhs, число входных переменных.

Переменные varargin и varargout определяются во время вызова функции. Число элементов в этих двух списках следующее.

- На входе число элементов в varargin равно rhs.
- На входе число элементов в varargout равно нулю.
- На выходе число элементов в varargout должно быть lhs.

Действительное число элементов в varargin зависит от аргументов, предоставленных пользователем. В противоположность этому список varargout всегда пуст на входе, а задача определения его содержимого оставлена на функцию.

Давайте теперь рассмотрим следующую функцию myargndemo, которая сделает эту тему более практичной. Функция myargndemo отображает число выходных аргументов lhs, число входных аргументов rhs и число элементов в списке varargin. Затем мы установим количество выходных аргументов в varargout равным 1.

```
function varargout = myargndemo ( varargin )
  [lhs,rhs]=argn()
  mprintf("lhs=%d, rhs=%d, length(varargin)=%d\n",..
    lhs,rhs,length(varargin))
  for i = 1 : lhs
    varargout(i) = 1
  end
endfunction
```

В нашей функции мы просто копируем входные аргументы в выходные аргументы.

Следующий пример показывает как функция работает, когда её вызывают.

```
-->myargndemo();
lhs=1, rhs=0, length(varargin)=0
-->myargndemo(1);
lhs=1, rhs=1, length(varargin)=1
-->myargndemo(1,2);
lhs=1, rhs=2, length(varargin)=2
-->myargndemo(1,2,3);
lhs=1, rhs=3, length(varargin)=3
-->y1 = myargndemo(1);
lhs=1, rhs=1, length(varargin)=1
-->[y1,y2] = myargndemo(1);
lhs=2, rhs=1, length(varargin)=1
-->[y1,y2,y3] = myargndemo(1);
lhs=3, rhs=1, length(varargin)=1
```

В предыдущем примере показано, что число элементов в списке varargin равно rhs.

Заменим, что минимальное число выходных аргументов равно 1, так что мы всегда имеем lhs $\geqslant$ 1. Это свойство интерпретатора Scilab, который заставляет функцию всегда возвращать по крайней мере один выходной аргумент.

Подчеркнём, что функция может быть определена как с varargin, так и с varargout, только с varargin или только с varargout. Это решение остаётся за разработчиком функции.

Функция myargndemo может быть вызвана с любым числом аргументов. На практике не все вызываемые последовательности будут разрешены, так что мы будем должны вводить команды ошибки, которые будут ограничивать использование функции. Эта тема будет рассмотрена в следующем разделе.

4.2.2 Практические вопросы

Теперь вернёмся к более практичному примеру, который включает в себя вычисление численных производных. Этот пример довольно сложен, но представляет ситуацию, где число входных и выходных аргументов переменное.

Функция derivative в Scilab позволяет вычислить первую (якобиан) и вторую (гессиан) производные функции со многими параметрами. Для того, чтобы сделать обсуждение этой темы более жизненной, мы рассмотрим проблему внедрения упрощённой версии этой функции. Реализация функции derivative основана на конечных разностях и использует формулы конечных разностей различных порядков (подробнее на эту тему см. в [46]).

Будем полагать, что функция гладкая и что относительная ошибка в вычислении функции равна машинной точности $\varepsilon \approx 10^{-16}$. Общая ошибка, которая связана с формулой конечных разностей, это сумма ошибок усечения (поскольку используется ограниченной число членов в разложении в ряд Тейлора) и ошибки округления (из-за ограниченной точности вычислений с плавающей запятой в вычислении функции). Следовательно, оптимальный шаг, который минимизирует общую ошибку может быть однозначно вычислен в зависимости от машинной точности.

Следующая функция `myderivative1` позволяет вычислять первую и вторую производные функции одной переменной. Её входными аргументами являются функция для дифференцирования `f` и вычисления производной в точке `x`, шаг `h` и порядок формулы `order`. Функция обеспечивает прямую формулу первого порядка и центрированную формулу второго порядка. Выходными аргументами являются значение первой производной `fp` и второй производной `fpp`. В комментариях функции мы написали оптимальный шаг в качестве памятки.

```
function [fp,fpp] = myderivative1 ( f , x , order , h )
  if ( order == 1 ) then
    fp = (f(x+h) - f(x))/h                    // h=%eps^(1/2)
    fpp = (f(x+2*h) - 2*f(x+h) + f(x) )/h^2 // h=%eps^(1/3)
  else
    fp = (f(x+h) - f(x-h))/(2*h)            // h=%eps^(1/3)
    fpp = (f(x+h) - 2*f(x) + f(x-h) )/h^2 // h=%eps^(1/4)
  end
endfunction
```

Теперь проанализируем поведение функции `myderivative1` на практике. Мы раскроем, что эта функция имеет несколько изъянов и отсутствие гибкости и производительности.

Рассмотрим вычисление численной производной функции косинуса в точке $x = 0$. Определим функцию `myfun`, которая вычисляет косинус её входного аргумента `x`.

```
function y = myfun ( x )
  y = cos(x)
endfunction
```

В следующем примере мы используем функцию `myderivative1` для того, чтобы вычислить первую производную косинуса.

```
-->format("e",25)
-->x0 = %pi/6;
-->fp = myderivative1 ( myfun , x0 , 1 , %eps^(1/2) )
 fp  =
  - 5.000000074505805969D-01
-->fp = myderivative1 ( myfun , x0 , 2 , %eps^(1/3) )
 fp  =
  - 5.000000000026094682D-01
```

Точное значение первой производной равно $-\sin(\pi/6) = -1/2$. Мы видим, что центрированная формула, связанная с порядком `order=2`, как и ожидалось, более точная, чем формула порядка `order=1`.

В следующем примере мы вызываем функцию `myderivative1` и вычисляем первую и вторую производные косинуса.

```
-->format("e",25)
-->x0 = %pi/6;
-->[fp,fpp] = myderivative1 ( myfun , x0 , 1 , %eps^(1/2) )
 fpp  =
```

```
  - 1.0000000000000000D+00
 fp  =
  - 5.0000000745058060D-01
-->[fp,fpp] = myderivative1 ( myfun , x0 , 2 , %eps^(1/3) )
 fpp  =
  - 8.6602990169071092D-01
 fp  =
  - 5.0000000000260947D-01
```

Точное значение второй производной `fpp` равно $-\cos(\pi/6) = -\sqrt{3}/2 \approx -8{,}6602 \cdot 10^{-1}$. И снова мы видим, что формула второго порядка более точная, чем формула первого порядка.

Мы проверили, что наша реализация верна. Теперь проанализируем разработку функции и введём необходимость в переменном числе аргументов.

Функция `myderivative1` имеет три недостатка.

- Она может делать ненужные вычисления, что является проблемой производительности.
- Она может давать неточные результаты, что является проблемой точности.
- Она не позволяет использовать значения по умолчанию для `order` и `h`, что является проблемой гибкости.

Проблема производительности вызвана тем фактом, что два выходных аргумента `fp` и `fpp` вычисляются даже если пользователь функции на самом деле не требует второй, необязательный, выходной аргумент `fpp`. Заметим, что вычисление `fpp` требует несколько дополнительных вычислений функции в сравнении с вычислениями `fp`. Это подразумевает, что если требуется только выходной аргумент `fp`, то выходной аргумент по-прежнему будет вычисляться, что бесполезно. Точнее, если используется вызывающая последовательность:

```
fp = myderivative1 ( f , x , order , h )
```

то, даже если `fpp` не является выходным аргументом, внутри переменная `fpp` по-прежнему будет вычисляться.

Проблема точности вызывается тем фактом, что оптимальный шаг может быть использован либо для вычисления первой производной, либо для вычисления второй производной, но не обеих. Действительно, оптимальный шаг разный для первой и для второй производной. Точнее, если используется вызывающая последовательность:

```
[fp,fpp] = myderivative1 ( myfun , x0 , 1 , %eps^(1/2) )
```

то шаг `%eps^(1/2)` является оптимальным для первой производной `fp`. Но, если используется вызывающая последовательность:

```
[fp,fpp] = myderivative1 ( myfun , x0 , 1 , %eps^(1/3) )
```

тогда шаг `%eps^(1/3)` является оптимальным для второй производной `fpp`. Во всех случаях мы должны выбирать и не имеем хорошей точности для первой и второй производных.

Проблема гибкости вызвана тем фактом, что пользователь должен определять как `order`, так и `h` входными аргументами. Проблема в том, что пользователь может не знать какое значение использовать для этих параметров. Это особенно очевидно для параметра `h`, который связан с проблемами плавающей запятой, которые могут быть совершенно неизвестны пользователю. Следовательно, было бы удобно, если мы бы мы могли использовать

значения этих входных переменных по умолчанию. Например, мы можем захотеть использовать значение по умолчанию `order=2`, поскольку оно даёт более точные значения производной с тем же количеством выходных значений функции. Задавая порядок, оптимальный шаг `h` может быть вычислен с помощью одной из оптимальных формул.

Теперь, когда мы знаем какие проблемы встречаются с нашей `myderivative1`, мы проанализируем реализацию, основанную на переменном количестве входных и выходных переменных.

4.2.3 Использование переменных аргументов на практике

Целью этого раздела заключается в том, чтобы предоставить реализацию функции, которая позволяет использовать следующие вызывающие последовательности.

```
fp = myderivative2 ( myfun , x0 )
fp = myderivative2 ( myfun , x0 , order )
fp = myderivative2 ( myfun , x0 , order , h )
[fp,fpp] = myderivative2 ( ... )
```

Главным преимуществом этой реализации являются

- возможно задать значения по умолчанию для `order` и `h`,
- вычислять `fpp` только если она задана пользователем,
- использовать оптимальный шаг `h` как для первой, так и для второй производной, если это не указано пользователем.

Следующая функция `myderivative2` содержит алгоритм конечной разности с оптимальным порядком `myderivative2` и шагом `h`.

```
1  \lstset{language=scilabscript}
2  \lstset{numbers=left}
3  \begin{lstlisting}
4  function varargout = myderivative2 ( varargin )
5    [lhs,rhs]=argn()
6    if ( rhs < 2 | rhs > 4 ) then
7      error ( msprintf(..
8        "%s: Ожидалось от %d до %d входных аргументов, но указано  %d",..
9        "myderivative2",2,4,rhs))
10   end
11   if ( lhs > 2 ) then
12     error ( msprintf(..
13       "%s: Ожидалось от %d до %d выходных аргументов, но указано  %d",..
14       "myderivative2",0,2,lhs))
15   end
16   f = varargin(1)
17   x = varargin(2)
18   if ( rhs >= 3 ) then
19     order = varargin(3)
20   else
21     order = 2
22   end
23   if ( rhs >= 4 ) then
24     h = varargin(4)
25     hflag = %t
26   else
```

```
27      hflag = %f
28    end
29    if ( order == 1 ) then
30      if ( ~hflag ) then
31        h = %eps^(1/2)
32      end
33      fp = (f(x+h) - f(x))/h
34    else
35      if ( ~hflag ) then
36        h = %eps^(1/3)
37      end
38      fp = (f(x+h) - f(x-h))/(2*h)
39    end
40    varargout(1) = fp
41    if ( lhs >= 2 )  then
42      if ( order == 1 ) then
43        if ( ~hflag ) then
44          h = %eps^(1/3)
45        end
46        fpp = (f(x+2*h) - 2*f(x+h) + f(x) )/(h^2)
47      else
48        if ( ~hflag ) then
49          h = %eps^(1/4)
50        end
51        fpp = (f(x+h) - 2*f(x) + f(x-h) )/(h^2)
52      end
53      varargout(2) = fpp
54    end
55  endfunction
```

Поскольку тело этой функции довольно сложное, то мы сейчас проанализируем его основные части.

Строчка №1 определяет функцию, как берущую переменную `varargin` в качестве входного аргумента, и переменную `varargout` в качестве выходного аргумента.

Строчка №2 использует функцию `argn`, которая возвращает настоящее число входных и выходных аргументов. Например, когда используется вызывающая последовательность `fp = myderivative2 ( myfun , x0 )`, то мы имеем `rhs=2` и `lhs=1`. Следующий пример представляет различные возможные вызывающие последовательности.

```
myderivative2 ( myfun , x0 )                    // lhs=1, rhs=2
fp = myderivative2 ( myfun , x0 )               // lhs=1, rhs=2
fp = myderivative2 ( myfun , x0 , order )       // lhs=1, rhs=3
fp = myderivative2 ( myfun , x0 , order , h ) // lhs=1, rhs=4
[fp,fpp] = myderivative1 ( myfun , x0 )         // lhs=2, rhs=2
```

Строчки с №2 по №12 добавлены для проверки того, что число входных и выходных аргументов правильное. Следующий пример покажет ошибку, которая формируется в случае, когда пользователь ошибочно вызовет функцию с 1 входным аргументом

```
-->fp = myderivative2 ( myfun )
 !--error 10000
myderivative2: Ожидалось от 2 до 4 входных аргументов, но указано 1
at line       6 of function myderivative2 called by :
fp = myderivative2 ( myfun )
```

Строчки №13 и №14 показывают как напрямую установить значения входных аргументов `f` и `x`, которые всегда должны быть в вызывающей последовательности. Строчки с №15

по №19 позволяют установить значения параметра `order`. Когда число входных аргументов `rhs` более 3, то мы делаем вывод, что значение переменной `order` задано пользователем, и мы используем это значение напрямую. В противном случае, мы устанавливаем значение этого параметра по умолчанию, то есть мы устанавливаем `order=2`.

Тот же самый процесс проведён в строчках с №20 по №25 для того, чтобы установить значение `h`. Более того, мы устанавливаем значение логической переменной `hflag`. Эта переменная устанавливается равной `%t`, если пользователь указал `h` и в `%f`, если нет.

Строчки с №26 по №36 позволяют вычислить первую производную, а вторые производные вычисляются в строчках с №38 по №51. В каждом случае, если пользователь не укажет `h`, то есть, если `hflag` установлен в «*ложь*», то используется оптимальный шаг.

Число выходных переменных должно быть больше или равно одному. Следовательно, не нужно проверять значение `lhs` перед установкой `varargout(1)` в строчке №37.

Один важный момент заключается в том, что вторая производная `fpp` вычисляется только если это потребуется пользователю. Это гарантируется в строчке №38, которая проверяет количество выходных переменных: вторая производная вычисляется только если второй выходной аргумент `fpp` был действительно записан в вызывающей последовательности.

Следующий пример показывает простое использование функции `myderivative2`, где мы использовали значение по умолчанию как для `order`, так и `h`.

```
-->format("e",25)
-->x0 = %pi/6;
-->fp = myderivative2 ( myfun , x0 )
 fp  =
  - 5.000000000026094682D-01
```

Заметим, что в этом случае вторая производная не вычислялась, что может сохранить значительное количество времени.

В следующем примере мы установили значение `order` и использовали значение по умолчанию `h`.

```
-->fp = myderivative2 ( myfun , x0 , 1 )
 fp  =
  - 5.000000074505805969D-01
```

Мы можем также вызвать эту функцию с двумя выходными аргументами, как в следующем примере.

```
-->[fp,fpp] = myderivative2 ( myfun , x0 , order )
 fpp  =
  - 8.660254031419754028D-01
 fp  =
  - 5.000000000026094682D-01
```

Заметим, что в этом случае оптимальный шаг `h` использовался как для первой, так и для второй производной.

В этом разделе мы видели как управлять числом переменных входных и выходных аргументов.

Но, в некоторых ситуациях, этого не достаточно и по-прежнему имеются ограничения. Например, метод, который мы представили ограничен порядком аргументов, то есть, их положением в вызывающей последовательности. В самом деле, функция `myderivative2` не может быть вызвана использованием значений по умолчанию для аргументов `order` и установкой настраиваемых значений для аргумента `h`. Это ограничение вызвано порядком

аргументов: входной аргумент h идёт после аргумента order. На практике удобно, если можно использовать значение по умолчанию для аргумента №i, и по-прежнему устанавливать значение аргумента №$i + 1$ (или любого другого аргумента в правой части вызывающей последовательности). Следующий раздел позволяет решить этот вопрос.

4.2.4 Значения по умолчанию для необязательных аргументов

В этом разделе мы опишем как управлять необязательными аргументами со значениями по умолчанию. Мы покажем как решить проблему, порождённую упорядоченными входными переменными с помощью специального использования синтаксиса пустых матриц [].

Действительно, мы видели в прошлом параграфе, что основное управление необязательными аргументами запрещает нам устанавливать входной аргумент №$i + 1$ и использовать значение по умолчанию для входного аргумента №i. В этом разделе мы покажем метод, где пустая матрица используется для представления параметра по умолчанию.

Давайте рассмотрим функцию myfun, которая принимает x, p и q в качестве входных переменных и возвращает выходной аргумент y.

```
function y = myfun ( x , p , q )
  y = q*x^p
endfunction
```

Если мы установим все x, p и q в качестве входных аргументов, то функция работает превосходно, как в следующем примере.

```
-->myfun(3,2,1)
 ans  =
    9.
```

Сейчас наша функция myfun не очень гибкая, так что, если мы передадим только один или два аргумента, то функция вызовет ошибку.

```
-->myfun(3)
 !--error 4
Неизвестная переменная: q

at line       2 of function myfun called by :
myfun(3)

->myfun(3,2)
 !--error 4
Неизвестная переменная: q

at line       2 of function myfun called by :
myfun(3,2)
```

Было бы удобно, если бы, например, параметр p имел по умолчанию значение 2, а параметр p имел по умолчанию значение 1. В этом случае, если ни p ни q не указаны, то команда foo(3) должна вернуть 9.

Мы можем использовать переменную varargin для того, чтобы иметь переменное число входных переменных. Но, используя напрямую, это не позволит установить третий аргумент и использовать значение по умолчанию для второго аргумента. Эта проблема вызвана упорядочиванием входных переменных. Для того, чтобы решить эту проблему, мы будем по-особенному использовать пустую матрицу [].

Преимущество этого метода заключается в возможности использовать значение по умолчанию аргумента p, при этом устанавливать значение аргумента q. Для того, чтобы информировать функцию, что особый аргумент должен быть установлен равным значению по умолчанию, мы устанавливаем его равным пустой матрицу. Эта ситуация представлена в следующем примере, где мы устанавливаем аргумент p равным пустой матрице.

```
-->myfun2(2,[],3) // то же, что и myfun2(2,2,3)
 ans  =
    12.
```

Для того, чтобы определить функцию, мы используем следующий метод. Если входной аргумент не указан, или если он указан равным пустой матрице, то мы используем значение по умолчанию. Если аргумент указан и отличается от пустой матрицы, то мы используем непосредственно его. Этот метод представлен в функции myfun2.

```
1  function y = myfun2 ( varargin )
2    [lhs,rhs]=argn()
3    if ( rhs<1 | rhs>3 ) then
4  msg=gettext("%s: Wrong number of input arguments: %d to %d expected.\n")
5      error(msprintf(msg,"myfun2",1,3))
6    end
7    x = varargin(1)
8    pdefault = 2
9    if ( rhs >= 2 ) then
10     if ( varargin(2) <> [] ) then
11       p = varargin(2)
12     else
13       p = pdefault
14     end
15   else
16     p = pdefault
17   end
18   qdefault = 1
19   if ( rhs >= 3 ) then
20     if ( varargin(3) <> [] ) then
21       q = varargin(3)
22     else
23       q = qdefault
24     end
25   else
26     q = qdefault
27   end
28   y = q * x^p
29 endfunction
```

С точки зрения алгоритма мы могли бы выбрать другое опорное значение, отличное от пустой матрицы. Например, мы могли бы рассматривать пустую строку или любое другое особое значение. Причина, почему пустая матрица предпочитается на практике состоит в том, что сравнение с пустой матрице быстрее, чем сравнение с любым другим особым значением. Действительное содержимое матрицы не имеет значения, поскольку сравнивается только размер матрицы. Следовательно, накладные расходы на производительность из-за управления значениями по умолчанию малы как только можно.

Конечно, мы по-прежнему можем использовать необязательные аргументы как обычно, как это показано ниже.

```
-->myfun2(2)
```

```
 ans  =
    4.
-->myfun2(2,3)
 ans  =
    8.
-->myfun2(2,3,2)
 ans  =
    16.
```

На практике этот метод и гибок и надёжен, с основным управлением входных переменных, которые по-прежнему главным образом зависят от их порядка.

Этот метод по-прежнему ограничен, если число входных переменных велико. В этом случае существуют другие решения, которые позволяют использовать упорядоченные аргументы.

В разделе 4.4 мы анализируем модуль `parameters`, который позволяет конфигурировать неупорядоченный набор входных аргументов отдельно из действительного использования параметров. Другое решение заключается в том, чтобы имитировать объектно-ориентированное программирование, как показано в разделе 3.7.

4.2.5 Функции с переменным типом входных аргументов

В этом разделе мы представляем функцию, чьё поведение зависит от типа её входных аргументов.

Следующая функция `myprint` указывает особое отображение для матрицы значений типа double и другое отображение для матрицы логических значений. Тело функции основано на команде `if`, которая переключает различные части исходного кода в зависимости от типа переменной `X`.

```
function myprint ( X )
  if ( type(X) == 1 ) then
    disp("Матрица значений типа double")
  elseif ( type(X) == 4 ) then
    disp("Матрица логических значений")
  else
    error ( "Неожиданный тип." )
  end
endfunction
```

В следующем примере мы вызываем функцию `myprint` с двумя разными типами матриц.

```
-->myprint ( [1 2] )
 Матрица значений типа double
-->myprint ( [%T %T] )
 Матрица логических значений
```

Мы можем сделать предыдущую функцию более полезной, определив отдельно правила форматирования для каждого типа данных. В следующей функции мы группируем три значения, отделённых большим пропуском, который визуально создаёт группы переменных.

```
function myprint ( X )
  if ( type(X) == 1 ) then
    // Вещественная или комплексная матрица
    for i = 1 : size(X,"r")
      for j = 1 : size(X,"c")
        mprintf("%-5d ",X(i,j))
```

```
      if ( j == 3 ) then
        mprintf("    ")
      end
    end
    mprintf("\n")
  end
elseif ( type(X) == 4 ) then
  // Матрица логических значений
  for i = 1 : size(X,"r")
    for j = 1 : size(X,"c")
      mprintf("%s ",string(X(i,j)))
      if ( j == 3 ) then
        mprintf("  ")
      end
    end
    mprintf("\n")
  end
else
  error ( "Неожиданный тип входного аргумента" )
end
endfunction
```

Следующий пример показывает использование предыдущей функции. Сначала отобразим матрицу значений типа double.

```
-->X = [
-->1 2 3 4 5 6
-->7 8 9 10 11 12
-->13 14 15 16 17 18
-->];
-->myprint ( X )
1      2      3           4      5      6
7      8      9           10     11     12
13     14     15          16     17     18
```

Затем отобразим матрицу логических значений.

```
-->X = [
-->%T %T %F %F %F %F
-->%T %F %F %T %T %T
-->%F %T %F %F %T %F
-->];
-->myprint ( X )
T T F   F F F
T F F   T T T
F T F   F T F
```

Многие встроенные функции разработаны на этом принципе. Например, функция `roots` возвращает корни многочлена. Её входные аргументы могут быть либо многочленом либо матрицей значений типа double, представляющих его коэффициенты. На практике, возможность управлять разными типами переменных в одной функции даёт дополнительный уровень гибкости, который гораздо сложнее получить в компилируемых языках, таких как Си или Фортран.

4.3 Устойчивые функции

В этом разделе мы представляем некоторые правила, которые можно применять ко всем функциям которые разрабатываются так, чтобы быть устойчивыми при неправильном использовании. В следующем разделе мы представим функции `warning` и `error`, которые являются основой разработки устойчивых функций. Затем мы представим общую схему работы для проверок, используемых внутри устойчивых функций. Наконец мы представим функцию, которая вычисляем матрицу Паскаля и покажем как применять эти правила на практике.

4.3.1 Функции warning и error

Часто случается, что входные аргументы функции могут иметь только ограниченное число возможных значений. Например, мы могли бы потребовать, чтобы данное входное число было положительным, или что данное число с плавающей запятой могло иметь только три возможных значения. В этом случае мы можем использовать функции `error` или `warning`, которые представлены на рисунке 23. Функция `gettext` относится к локализации и описана позднее в этом разделе.

`error`	Посылает сообщение об ошибке и останавливает вычисление.
`warning`	Посылает сообщение-предупреждение.
`gettext`	Получает текст, переведённый на местный язык.

Рис. 23: Функции, относящиеся к сообщениям об ошибке.

Теперь дадим пример, который показывает как эти функции могут использоваться для защиты пользователя функции от неправильного использования. В следующем примере мы определяем функцию `mynorm`, которая является упрощённой версией встроенной функции `norm`. Наша функция `mynorm` позволяет вычислить 1-, 2- или ∞-норму вектора. В противном случае наша функция не определена и поэтому формируется ошибка.

```
function y = mynorm ( A , n )
  if ( n == 1 ) then
    y = sum(abs(A))
  elseif ( n == 2 ) then
    y=sum(A.^2)^(1/2);
  elseif ( n == "inf" ) then
    y = max(abs(A))
  else
    msg = msprintf("%s: Invalid value %d for n.","mynorm",n)
    error ( msg )
  end
endfunction
```

В следующем примере мы проверяем выходные значения функции `mynorm`, когда входной аргумент `n` равен 1, 2, «inf» и неожиданному значению 12.

```
-->mynorm([1 2 3 4],1)
 ans  =
    10.
-->mynorm([1 2 3 4],2)
 ans  =
    5.4772256
```

```
-->mynorm([1 2 3 4],"inf")
 ans  =
    4.
-->mynorm([1 2 3 4],12)
 !--error 10000
mynorm: Invalid value 12 for n.
at line      10 of function mynorm called by :
mynorm([1 2 3 4],12)
```

Входной аргумент функции `error` является строкой. Мы могли бы использовать более простые сообщения, такие как `"Неверное значение n."`, например. Наше сообщение немного более сложное по нескольким причинам. Цель — дать пользователю полезную обратную связь, когда формируется ошибка в как можно более глубоком звене цепи вызовов. Для того, чтобы так сделать, мы даём как можно больше информации о происхождении ошибки. Во-первых, это удобно для пользователя, что его информируют о том, какое точно значение было отклонено алгоритмом. Следовательно, мы включаем действующее значение входного аргумента `n` в сообщение. Более того, мы делаем так, чтобы первой строкой сообщения об ошибке было название функции. Это позволяет немедленно получить имя функции, сформировавшей ошибку. В этом случае используется функция `msprintf` для форматирования строки и формирования переменной `msg`, которая передаётся функции `error`.

Мы можем сделать так, что сообщение об ошибке можно было перевести на другие языки, если необходимо. В самом деле, Scilab *локализован* так, что большинство сообщений появятся на языке пользователя. Следовательно, мы получим сообщения на английском в США и Великобритании, на французском во Франции и т. д. Для того, чтобы это сделать, мы можем использовать функцию `gettext`, которая возвращает переведённую строку, на основе базы данных локализации. Это приведено на следующем примере.

```
localstr = gettext ( "%s: Invalid value %d for n." )
msg = msprintf ( localstr , "mynorm" , n )
error ( msg )
```

Заметьте, что строка, которая передаётся в функцию `gettext`, является не выходным, а входным аргументом функции `msprintf`. Это из-за того, что система локализации предоставляет карту из строки `"%s: Invalid value %d for n."` в локализованные строки, такие как, например, французское сообщение `"%s: Valeur %d invalide pour n."`. Эти локализованные сообщения хранятся в файлах данных «.pot». Больше информации о локализации найдёте в [28].

Есть случаи, когда мы хотим сформировать сообщение, но не хотим прерывать вычисление. Например, мы хотим информировать нашего пользователя, что функция `mynorm` несколько хуже по сравнению со встроенной функцией `norm`. В этом случае мы не хотим прерывать алгоритм, поскольку вычисление правильное. Мы можем использовать функцию `warning`, как в следующем примере.

```
1  function y = mynorm2 ( A , n )
2    msg = msprintf("%s:  Please use norm instead.","mynorm2")
3    warning(msg)
4    if ( n == 1 ) then
5      y = sum(abs(A))
6    elseif ( n == 2 ) then
7      y=sum(A.^2)^(1/2)
8    elseif ( n == "inf" ) then
9      y = max(abs(A))
10   else
```

```
11      msg = msprintf("%s: Invalid value %d for n.","mynorm2",n)
12      error ( msg )
13    end
14  endfunction
```

Следующий пример показывает результат работы функции.

```
-->mynorm2([1 2 3 4],2)
ВНИМАНИЕ: mynorm2: Please use norm instead.
 ans  =

    5.4772256
```

4.3.2 Общая схема для проверок входных аргументов

Устойчивая функция должна защищать пользователя от неправильного использования функции. Например, если функция принимает матрицу значений типа double в качестве входного аргумента, мы можем получить сообщения об ошибке, которые далеко не ясные. Мы можем даже вообще не получить никаких сообщений от ошибке, например, если функция рушится тихо. В этом разделе мы представляем общую схему для проверок в устойчивых функциях.

Следующая функция `pascalup_notrobust` — это функция, которая возвращает верхнюю треугольную матрицу Паскаля `P` в зависимости от размера матрицы `n`.

```
function P = pascalup_notrobust ( n )
  P = eye(n,n)
  P(1,:) = ones(1,n)
  for i = 2:(n-1)
    P(2:i,i+1) = P(1:(i-1),i)+P(2:i,i)
  end
endfunction
```

Далее мы вычислим верхнюю треугольную матрицу Паскаля размером 5×5.

```
-->pascalup_notrobust ( 5 )
 ans  =
    1.    1.    1.    1.    1.
    0.    1.    2.    3.    4.
    0.    0.    1.    3.    6.
    0.    0.    0.    1.    4.
    0.    0.    0.    0.    1.
```

Функция `pascalup_notrobust` не делает никаких проверок входного аргумента `n`. В случае, когда входной аргумент отрицательный или не является числом с плавающей запятой, функция не формирует никакого сообщения от ошибке.

```
-->pascalup_notrobust(-1)
 ans  =
      []
-->pascalup_notrobust(1.5)
 ans  =
    1.
```

Это поведение не должно считаться «нормальным», поскольку она рушится тихо. Следовательно, пользователь может указать неверные входные аргументы функции и даже не заметить что произошло что-то неправильное.

Вот почему мы часто проверяем входные аргументы функций так, чтобы формировались как можно более понятные для пользователя сообщения об ошибке. Вообще, мы должны рассматривать следующие проверки:

- число входных/выходных аргументов
- тип входных аргументов,
- размер входных аргументов,
- содержание входных аргументов,

которые могут быть сокращены в виде «число/тип/размер/содержание». Эти правила следует включать как часть нашего стандартного способа написания публичных функций.

4.3.3 Пример устойчивой функции

В этом разделе мы представляем пример устойчивой функции, которая вычисляет матрицу Паскаля. Мы представляем примеры, где проверки, которые мы используем, полезны для обнаружения неверного использования функции.

Функция `pascalup` является улучшенной версией со всеми необходимыми проверками.

```
function P = pascalup ( n )
  //
  // Проверка числа аргументов
  [lhs , rhs] = argn()
  if ( rhs <> 1 ) then
lstr=gettext("%s: Wrong number of input arguments: %d to %d expected , but %d provided.")
    error ( msprintf(lstr,"pascalup",1,1,rhs))
  end
  //
  // Проверка типа аргументов
  if ( typeof(n) <> "constant" ) then
lstr=gettext("%s: Wrong type for input argument #%d: %s expected , but %s provided.")
    error ( msprintf(lstr,"pascalup",1,"constant",typeof(n)))
  end
  //
  // Проверка размера аргументов
  if ( size(n,"*") <> 1 ) then
lstr=gettext("%s: Wrong size for input argument #%d: %d entries expected , but %d provided.")
    error ( msprintf(lstr,"pascalup",1,1,size(n,"*")))
  end
  //
  // Проверка содержания аргументов
  if ( imag(n)<>0 ) then
lstr = gettext("%s: Wrong content for input argument #%d: complex numbers are forbidden.")
    error ( msprintf(lstr,"pascalup",1))
  end
  if ( n < 0 ) then
lstr=gettext("%s: Wrong content for input argument #%d: positive entries only are expected.")
    error ( msprintf(lstr,"pascalup",1))
  end
  if ( floor(n)<>n ) then
lstr=gettext("%s: Wrong content of input argument #%d: argument is expected to be a flint.")
    error ( msprintf(lstr,"specfun_pascal",1))
  end
  //
  P = eye(n,n)
  P(1,:) = ones(1,n)
  for i = 2:(n-1)
    P(2:i,i+1) = P(1:(i-1),i)+P(2:i,i)
  end
endfunction
```

В следующем примере мы запускаем функцию `pascalup` и формируем различные сообщения об ошибках.

```
-->pascalup ( )
 !--error 10000
pascalup: Wrong number of input arguments: 1 to 1 expected, but 0 provided.
at line       8 of function pascalup called by :
pascalup ( )
-->pascalup ( -1 )
 !--error 10000
pascalup: Wrong content for input argument #1: positive entries only are expected.
at line      33 of function pascalup called by :
pascalup ( -1 )
-->pascalup ( 1.5 )
 !--error 10000
specfun_pascal: Wrong content of input argument #1: argument is expected to be a flint.
at line      37 of function pascalup called by :
pascalup ( 1.5 )
```

Правила, которые мы представили, используются в большинстве макросов Scilab. Многочисленные эксперименты доказали, что эти методы предоставляют улучшенную устойчивость, так что пользователи с меньшей вероятностью будут использовать функции с неверными входными аргументами.

4.4 Использование `parameters`

Целью модуля `parameters` является возможность разработки функций, которые имеют, возможно, большое количество необязательных параметров. Использование этого модуля позволяет избежать проектирование функции с большим числом входных аргументов, что может привести к путанице. Этот модуль был введён в Scilab версии 5.0, после работы Янна Коллетта (Yann Collette) по объединению алгоритмов оптимизации, таких как алгоритмы генетики и имитация отжига. Функции модуля `parameters` представлены на рисунке 24.

`add_param`	Добавить параметр в список параметров
`get_param`	Получить значение параметра из списка параметров
`init_param`	Инициализировать структуру, которая будет управлять списком параметров
`is_param`	Проверяет, есть ли в наличии параметр, представленный в списке параметров
`list_param`	Перечислить все имена параметров в списке параметров
`remove_param`	Удалить параметр и связанное с ним значение из списка параметров
`set_param`	Установить значение параметра в списке параметров

Рис. 24: Функции из модуля `parameters`.

В первой части мы делаем обзор модуля и описываем его прямое использование. Во второй части мы представляем практический пример, основанный на алгоритме сортировки. Последняя часть концентрируется на безопасном использовании модуля, защищающем пользователя от общих ошибок.

4.4.1 Обзор модуля

В этом разделе мы представляем модуль `parameters` и даём пример его использования в контексте алгоритма слияния-сортировки. Точнее, мы представляем функции `init_param`,

add_param и get_param.

Модуль основан на установке связи ключей и значений.

- Каждый ключ соответствует полю списка параметров и сохраняется как строковая переменная. Список доступных ключей определён разработчиком функции. Нет ограничений на количество ключей.

- Тип каждого значения зависит от особых требований алгоритма. На самом деле любой тип значения может быть сохранён в структуре данных, включая (но не ограничиваясь) матрицу чисел типа double, строковые переменные, многочлены и т. д.

Следовательно, эта структура данных чрезвычайно гибка, как мы и увидим далее.

Для того, чтобы понять модуль parameters, мы можем выделить ситуации между двумя точками зрения:

- точка зрения пользователя,

- точка зрения разработчика.

Следовательно, в дальнейшем обсуждении мы будем представлять первую, которая имеет место, когда мы хотим использовать модуль parameters для того, чтобы вызвать функцию. Затем мы обсудим что́ разработчик должен сделать для того, чтобы управлять необязательными параметрами.

Для того, чтобы сделать обсуждение как можно более применимым, мы возьмём пример алгоритма сортировки и изучим его в этом разделе. Мы рассматриваем функцию mergesort, которая определяет алгоритм сортировки, основанный на комбинации рекурсивной сортировки и слияния. Функция mergesort связана со следующими вызывающими последовательностями

```
y = mergesort ( x )
y = mergesort ( x , params )
```

где x — матрица значений типа double, которые необходимо рассортировать, params — список параметров, и y — матрица рассортированных значений типа double. Действительная реализация функции mergesort будет определена позднее в этом разделе. Переменная params — это список параметров, который определяет два параметра:

- direction, логическое значение, *истина* для возрастающего порядка и *ложь* — для убывающего порядка (по умолчанию direction = %t),

- compfun, функция, которая определяет функцию сравнения (по умолчанию compfun = compfun_default).

Функция compfun_default — функция сравнения по умолчанию, будет представлена позднее в этом разделе.

Сначала мы рассмотрим точку зрения пользователя и представим пример использования функций из модуля parameters. Мы хотим отсортировать матрицу чисел в порядке возрастания. В следующем примере, мы вызываем функцию init_param, которая создаёт пустой список параметров params. Затем, мы добавим ключ "direction" к списку параметров. Наконец, мы вызовем функцию mergesort с переменной params в качестве второго входного аргумента.

```
params = init_param();
params = add_param(params,"direction",%f);
x = [4 5 1 6 2 3]';
y = mergesort ( x , params );
```

Теперь рассмотрим точку зрения разработчика и проанализируем команды, которые могут быть использованы в теле функции mergesort. В следующем примере мы вызываем функцию get_param для того, чтобы получить значения, соответствующие ключу "direction". Мы передаём значение %t функции get_param, которое является значением, используемым по умолчанию в случае, когда этот ключ не определён.

```
-->direction = get_param(params,"direction",%t)
 direction  =
  F
```

Поскольку ключ "direction" определён и является *ложью*, то мы просто получаем назад наше значение. Мы можем использовать расширенную вызывающую последовательность функции get_param с выходным аргументом err. Переменная err равна *истине*, если во время обработки аргумента произошла ошибка.

```
-->[direction,err] = get_param(params,"direction",%t)
 err  =
  F
 direction  =
  F
```

В нашем случае никакой ошибки не произошло, поэтому переменная err имеет значение *ложь*.

Мы можем анализировать другие комбинации событий. Например, давайте рассмотрим ситуацию, когда пользователь определяет только пустой список параметров, как показано ниже.

```
params = init_param();
```

В этом случае, в теле функции mergesort прямой вызов функции get_param формирует предупреждение.

```
-->direction = get_param(params,"direction",%t)
ВНИМАНИЕ: get_param: параметр direction не определен
 direction  =
  T
```

Предупреждение указывает, что ключ "direction" не определён, так что возвращается значение по умолчанию %t. Если мы используем дополнительный выходной аргумент err, то предупреждений больше нет, но переменная err становится *истиной*.

```
-->[direction,err] = get_param(params,"direction",%t)
 err  =
  T
 direction  =
  T
```

Как и в последнем примере, давайте рассмотрим случай, когда пользователь определяет переменную params как пустую матрицу.

```
params = [];
```

Как мы увидим позднее в этом разделе, этот случай может произойти когда мы хотим использовать значения по умолчанию. Следующий пример показывает, что ошибка формируется при вызове функции get_param.

```
-->direction = get_param(params,"direction",%t)
 !--error 10000
get_param: Неверный тип входного параметра №1: ожидался plist.
at line      40 of function get_param called by :
direction = get_param(params,"direction",%t)
```

Действительно, ожидается, что переменная `params` является `plist`. Это из-за того, что функция `init_param` создаёт переменные с типом `plist`, который стоит для «списка параметров». Фактически, действительная реализация функции `init_param` создаёт типизированный список с типом `plist`. Для того, чтобы предыдущая ошибка не формировалась, мы можем использовать выходной аргумент `err` как в следующем примере.

```
-->[direction,err] = get_param(params,"direction",%t)
 err  =
  T
 direction  =
  T
```

Мы видим, что переменная `err` — *истина*, указывающая, что была обнаружена ошибка во время обработки переменной `params`. Как и прежде, в этом случае функция `get_param` возвращает значение по умолчанию.

4.4.2 Практический случай

В этом разделе мы рассматриваем действительное применение функции `mergesort`, представленной в предыдущем разделе. Алгоритм сортировки, который мы собираемся представлять, основан на алгоритме, который Бруно Пинсьон (Bruno Pinçon) разработал в Scilab [44].

Модификации, включённые в реализацию, представленную в этом разделе, включают в себя управление необязательными аргументами `"direction"` и `"compfun"`. Заинтересованные читатели могут найти много подробностей об алгоритме сортировки в [27].

Следующая функция `mergesort` даёт алгоритм сортировки, основанный на методе слияния-сортировки. Её первый аргумент, `x` — это матрица значений типа double, которые нужно сортировать. Второй аргумент, `params`, необязательный и представляет список параметров. Мы делаем первый вызов функции `get_param` для того, чтобы получить значение параметра `direction` с `%t` в качестве значения по умолчанию. Затем мы получаем значение параметра `compfun` с `compfun_default` в качестве значения по умолчанию. Функция `compfun_default` определяется позже в этом разделе. Наконец, мы вызываем функцию `mergesortre`, которая реализует на практике рекурсивный алгоритм слияния-сортировки.

```
function y = mergesort ( varargin )
  [lhs,rhs]=argn()
  if ( and( rhs<>[1 2] ) ) then
    errmsg = sprintf(..
      "%s: Unexpected number of arguments : " + ..
      "%d provided while %d to %d are expected.",..
      "mergesort",rhs,1,2);
    error(errmsg)
  end
  x = varargin(1)
  if ( rhs<2 ) then
    params = []
  else
    params = varargin(2)
```

```
  end
  [direction,err] = get_param(params,"direction",%t)
  [compfun,err] = get_param(params,"compfun",compfun_default)
  y = mergesortre ( x , direction , compfun )
endfunction
```

Следующая функция mergesortre реализует рекурсивный алгоритм слияния-сортировки. Алгоритм делить матрицу x на две части x(1:m) и x(m+1:n), где m — целое число в середине между 1 и n. Как только отсортировали с помощью рекурсивного вызова, мы объединяем две упорядоченные части обратно в x.

```
function x = mergesortre ( x , direction , compfun )
  n = length(x)
  indices = 1 : n
  if ( n > 1 ) then
    m = floor(n/2)
    p = n-m
    x1 = mergesortre ( x(1:m) , direction , compfun )
    x2 = mergesortre ( x(m+1:n) , direction , compfun )
    x = merge ( x1 , x2 , direction , compfun )
  end
endfunction
```

Заметим, что функция mergesorte не вызывает саму себя. Вместо этого, функция mergesortre, которая на самом деле выполняет алгоритм рекурсивно, имеет фиксированное количество входных аргументов. Это позволяет уменьшить накладные расходы, вызванные обработкой необязательных переменных.

Следующая функция merge объединяет свои два отсортированных входных аргумента x1 и x2 в свой отсортированный выходной аргумент x. Операция сравнения выполняется функцией compfun. В случае, когда аргумент direction — *ложь*, мы меняем знак переменной order, так, чтобы порядок стал «убывающим» вместо «возрастающего» порядка по умолчанию.

```
function x = merge ( x1 , x2 , direction , compfun )
    n1 = length(x1)
    n2 = length(x2)
    n = n1 + n2
    x = []
    i = 1
    i1 = 1
    i2 = 1
    for i = 1:n
      order = compfun ( x1(i1) , x2(i2) )
      if ( ~direction ) then
        order = -order
      end
      if ( order<=0 ) then
        x(i) = x1(i1)
        i1 = i1+1
        if (i1 > m) then
          x(i+1:n) = x2(i2:p)
          break
        end
      else
        x(i) = x2(i2)
        i2 = i2+1
```

```
        if (i2 > p) then
          x(i+1:n) = x1(i1:m)
          break
        end
      end
    end
endfunction
```

Следующая функция `compfun_default` является функцией сравнения по умолчанию. Она возвращает `order=-1` если `x<y`, `order=0` если `x==y` и `order=1` если `x>y`.

```
function order = compfun_default ( x , y )
  if ( x < y ) then
    order = -1
  elseif ( x==y ) then
    order = 0
  else
    order = 1
  end
endfunction
```

Теперь проанализируем поведение функции `mergesort` вызывая её с особыми входными параметрами. В следующем примере мы сортируем матрицу scivarx, содержащую целый числа с плавающей запятой в порядке возрастания.

```
-->mergesort ( x )'
 ans  =
    1.    2.    3.    4.    5.    6.
```

В следующем примере мы устанавливаем параметр `direction` в `%f`, так что порядок сортированной матрицы убывающий.

```
-->params = init_param();
-->params = add_param(params,"direction",%f);
-->mergesort ( x , params )'
 ans  =
    6.    5.    4.    3.    2.    1.
```

Теперь мы хотим настроить функцию сравнения так, чтобы мы могли менять порядок сортированной матрицы. В следующем примере мы определяем функцию сравнения `mycompfun`, которая позволяет разделять чётные и нечётные целые числа с плавающей запятой.

```
function order = mycompfun ( x , y )
  if ( modulo(x,2) == 0 & modulo(y,2) == 1 ) then
    // чётное < нечётного
    order = -1
  elseif ( modulo(x,2) == 1 & modulo(y,2) == 0 ) then
    // чётное > нечётного
    order = 1
  else
    // 1<3 or 2<4
    if ( x < y ) then
      order = -1
    elseif ( x==y ) then
      order = 0
    else
      order = 1
    end
```

```
  end
endfunction
```

Мы можем настроить переменную `params` так, чтобы функция сортировки использовала нашу настраиваемую функцию сравнения `mycompfun` вместо функции сравнения по умолчанию `compfun_default`.

```
-->params = init_param();
-->params = add_param(params,"compfun",mycompfun);
-->mergesort ( [4 5 1 6 2 3]' , params )'
 ans  =
    2.    4.    6.    1.    3.    5.
```

Мы не настроили параметр `direction`, так что был использован порядок по умолчанию «возрастающий». Как мы можем видеть в выходном аргументе, чётные числа, как ожидалось, идут перед нечётными.

4.4.3 Проблемы с модулем parameters

У модуля `parameters` есть проблема, которая может заставить пользователя рассматривать значения по умолчанию вместо ожидаемых. Модуль `parameters` не защищает пользователя от неожиданных полей, что может произойти, если мы сделаем ошибку в имени ключа при настройке параметра. В этом разделе мы покажем как проблема может появиться с точки зрения пользователя. С точки зрения разработчика мы представляем функцию `check_param`, которая позволяет защитить пользователя от использования ключа, который не существует.

В следующем примере мы вызываем функцию `mergesort` с неверным ключом `"compffun"` вместо `"compfun"`.

```
-->params = init_param();
-->params = add_param(params,"compffun",mycompfun);
-->mergesort ( [4 5 1 6 2 3]' , params )'
 ans  =
    1.    2.    3.    4.    5.    6.
```

Мы видим, что наш неправильный ключ `"compffun"` был проигнорирован, так что использовалась функция сравнения по умолчанию: матрица была просто отсортирована в возрастающем порядке. Причина в том, что модуль `parameters` не проверяет, что ключ `"compffun"` не существует.

Поэтому мы предлагаем использовать следующую функцию `check_param`, которая принимает в качестве входных аргументов список параметров `params` и матрицу строчных элементов `allkeys`. Переменная `allkeys` хранит список всех ключей, которые доступны в списке параметров. Алгоритм проверяет, что каждый ключе в `params` существует в `allkeys`. Выходные аргументы — логическая `noerr` и строковая `msg`. Если нет ошибки, то переменная `noerr` — «*истина*» и строковая `msg` — пустая матрица. Если один ключ `params` не найден, мы устанавливаем `noerr` в значение «*ложъ*» и формируем сообщение об ошибке. Это сообщение об ошибке используется для действительного формирования ошибки только если выходной аргумент `msg` не использовался в вызывающей последовательности.

```
function [noerr,msg] = check_param(params,allkeys)
  if ( typeof(params) <> "plist" ) then
    error(sprintf(..
gettext("%s: Wrong type for input argument #%d: %s expected.\n"), ..
      "check_param", 1, "plist"))
```

```
  end
  if ( typeof(allkeys) <> "string" ) then
    error(sprintf(..
gettext("%s: Wrong type for input argument #%d: %s expected.\n"), ..
      "check_param", 2, "string"))
  end
  currkeys = getfield(1,params)
  nkeys = size(currkeys,"*")
  noerr = %t
  msg = []
  // Ключ №1 - это "plist".
  for i = 2 : nkeys
    k = find(allkeys==currkeys(i))
    if ( k == [] ) then
      noerr = %f
      msg = msprintf(..
gettext("%s: Unexpected key ""%s"" in parameter list."),..
      "check_param",currkeys(i))
      if ( lhs < 2 ) then
        error ( msg )
      end
      break
    end
  end
endfunction
```

Следующая версия функции mergesort использует функцию check_param для того, чтобы проверить, что список доступных ключей является матрицей ["directioncompfun"].

```
function y = mergesort ( varargin )
  [lhs,rhs]=argn()
  if ( and( rhs<>[1 2] ) ) then
    errmsg = sprintf(..
      "%s: Unexpected number of arguments : "+..
      "%d provided while %d to %d are expected.",..
      "mergesort",rhs,1,2);
    error(errmsg)
  end
  x = varargin(1)
  if ( rhs<2 ) then
    params = []
  else
    params = varargin(2)
  end
  check_param(params,["direction" "compfun"])
  [direction,err] = get_param(params,"direction",%t)
  [compfun,err] = get_param(params,"compfun",compfun_default)
  y = mergesortre ( x , direction , compfun )
endfunction
```

В следующем примере мы используем, как раньше, неверный ключ "compffun" вместо "compfun".

```
-->params = init_param();
-->params = add_param(params,"compffun",mycompfun);
-->mergesort ( [4 5 1 6 2 3]' , params )
 !--error 10000
check_param: Unexpected key "compffun" in parameter list.
```

```
at line      75 of function check_param called by :
at line      16 of function mergesort called by :
mergesort ( [4 5 1 6 2 3]' , params )
```

С нашей исправленной функцией формируется ошибка, которая позволяет посмотреть нашу ошибку.

4.5 Область видимости переменных в стеке вызовов

В этом разделе мы анализируем область видимости переменных и как это может взаимодействовать с поведением функций. Поскольку этот момент может привести к неожиданным ошибкам в программе, то мы предупреждаем об его использовании в функциях, которые могут быть разработаны лучшим образом. Затем мы представляем два различных случая, где область видимости переменных использовалась без должного внимания и как это может привести к труднообнаружимым ошибкам в программе.

4.5.1 Обзор области видимости переменных

В этом разделе мы представляем область видимости переменных и как это может влиять на поведение функций.

Предположим, что переменная, например `a`, определена в файле-сценарии и допустим, что та же переменная используется в функции, вызываемой прямо или косвенно. Что произойдёт в этом случае, зависит от чтения первой команды или написания переменной `a` в теле функции.

- Если первой прочитана переменная `a`, то значение этой переменной используется на более высоком уровне.
- Если первой написана переменная `a`, то местная переменная меняется (но переменная на более высоком уровне не меняется).

Представим эту общую схему работы на нескольких примерах.

В следующем примере мы представляем случай, когда переменная первой читается. Следующая функция `f` вызывает функцию `g` и вычисляет выражение, зависящее от входного аргумента `x` и переменной `a`.

```
function y = f ( x )
  y = g ( x )
endfunction
function y = g ( x )
  y = a(1) + a(2)*x + a(3)*x^2
endfunction
```

Заметим, что переменная `a` *не является* входным аргументом функции `g`. Заметим также, что в этом конкретном случае переменная `a` только читается, но не пишется.

В следующем примере мы определим переменные `a` и `x`. Затем мы вызовем функцию `f` с входным аргументом `x`. Когда мы определим значение `a`, то мы будет на уровне вызова №0 в стеке вызовов, в то время как в теле функции `g` мы находимся на уровне вызова №2 в стеке вызовов.

```
-->a = [1 2 3];
-->x = 2;
-->y = f ( x )
```

```
 y  =
    17.
-->a
 a  =
    1.    2.    3.
```

Мы видим, что функция `f` была вычислена правильно, используя в теле функции `g`, на уровне вызова №2, значение `a`, определённое на уровне вызова №0. Действительно, когда интерпретатор заходит в тело функции `g`, переменная `a` используется, но не определена. Это из-за того, что она не является ни входным аргументом, ни локально определённой в теле функции `g`. Следовательно, интерпретатор ищет `a` на более высоких уровнях в стеке вызовов. В теле функции `f`, на уровне №1 нет переменной `a`. Следовательно, интерпретатор ищет на более высоком уровне. На уровне №0 интерпретатор находит переменную `a`, которая содержит `[1 2 3]`. Эта переменная используется для вычисления выражения `y = a(1) + a(2)*x+a(3)*x^2`, что, наконец, позволяет вернуть переменную `y` с правильным содержанием. Этот процесс представлен на рисунке 25.

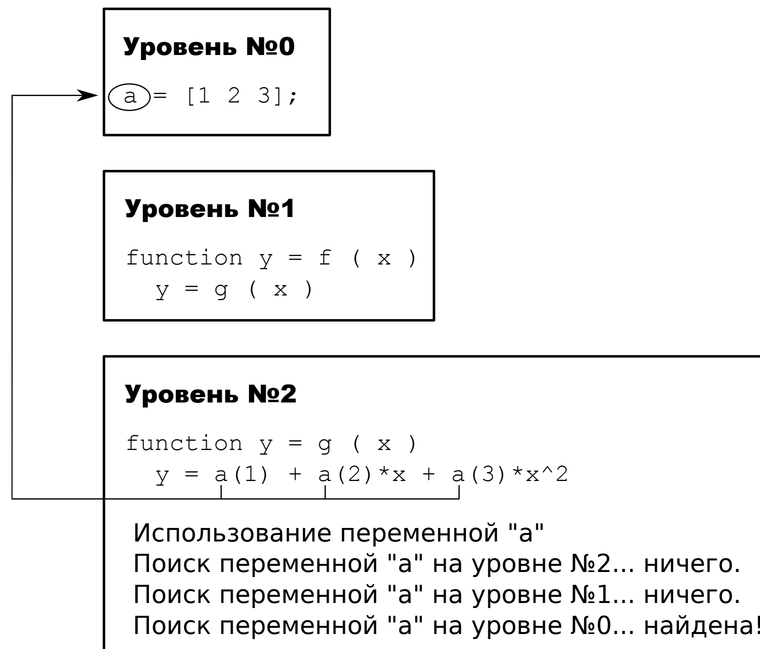

Рис. 25: Область видимости переменных. – Когда переменная используется, но не определена на нижнем уровне, то интерпретатор ищет ту же переменную на более высоком уровне стека вызовов. Будет использоваться первая переменная, которая будет найдена.

На практике, нам следует изменить построение функции `g` для того, чтобы сделать использование переменной `a` более ясным. Следующий пример определяет модифицированную версию функции `g`, где входной аргумент `a` делает явным тот факт, что мы используем переменную `a`.

```
function y = gfixed ( x , a )
  y = a(1) + a(2)*x + a(3)*x^2
endfunction
```

Функция `f`, соответственно, должна быть обновлена для того, чтобы указать функции `gfixed` аргумент, который нужен. Это поясняет следующий пример.

```
function y = ffixed ( x , a )
  y = gfixed ( x , a )
endfunction
```

Клиентский исходный код также следует обновить, как в следующем примере.

```
-->a = [1 2 3];
-->x = 2;
-->y = ffixed ( x , a )
 y  =
    17.
```

Функции `ffixed` и `gfixed`, с точки зрения разработки программного обеспечения, гораздо яснее, чем их предыдущие версии.

Теперь представим случай, где переменная `a` написана первой. Следующая функция `f2` вызывает функцию `g2` и вычисляет выражение, зависящее от входного аргумента `x` и переменной `a`.

```
function y = f2 ( x )
  y = g2 ( x )
endfunction
function y = g2 ( x )
  a = [4 5 6]
  y = a(1) + a(2)*x + a(3)*x^2
endfunction
```

Заметим, что переменная `a` написана прежде её использования.

Следующий пример использует те же данные, что и прежде.

```
-->a = [1 2 3];
-->x = 2;
-->y = f2 ( x )
 y  =
    38.
-->a
 a  =
    1.    2.    3.
```

Заметим, что значение, которое возвращено функцией `f2`, равно 38, а не предыдущему 17. Это подтверждает, что использовалось значение локальной переменной `a = [4 5 6]`, определённой на уровне №2, а не переменная `a = [1 2 3]`, определённая на уровне №0. Более того, заметим что переменная `a`, определённая на уровне №0 не изменилась после вызова `f2`: её значение по-прежнему `a = [1 2 3]`.

Стек вызовов может быть очень глубоким, но всегда будет то же поведение: какой бы ни был уровень в стеке вызовов, если переменная первой прочитана и была определена на более высоком уровне, то переменная будет использоваться напрямую. Это свойство кажется некоторым пользователям удобным. Мы подчёркиваем, что это может привести к программным ошибкам, которые невидимы и их трудно обнаружить. Действительно, разработчик функции `g` может непреднамеренно ошибочно использовать переменную `a` вместо другой переменной. Это приводит к проблемам, которые представлены в следующем разделе.

4.5.2 Плохая функция: неоднозначный случай

В следующих разделах мы представляем случаи, где неверные использования области видимости переменных ведут к программным ошибкам, которые трудно анализировать.

В первом случае мы представляем две вложенные функции, которые производят неправильный результат, потому что функция на уровне №2 в стеке вызовов содержит ошибку и использует переменную, определённую на уровне №1. Во втором случае функция, использующая функцию обратного вызова, тихо рушится, поскольку она использует то же имя переменной, что и пользователь.

Неоднозначный случай представлен в следующем примере ошибки. Мы сначала определяем функцию f3, которая принимает x и a в качестве входных аргументов, но, неожиданно, не использует значение a. Вместо этого, выражение содержит переменную b.

```
function y = f3 ( x , a )
  b = [4 5 6]
  y = g3 ( x , a )
endfunction
function y = g3 ( x , a )
  y = b(1) + b(2)*x + b(3)*x^2
endfunction
```

В следующем примере мы определим значение a и x и вызовем функцию f3.

```
-->a = [1 2 3];
-->x = 2;
-->y = f3 ( x , a )
 y  =
    38.
-->expected = a(1) + a(2)*x + a(3)*x^2
 expected  =
    17.
```

Значение, которое возвращает функция f3, соответствует параметрам, связанным с переменным b=[4 5 6]. Вместо этого переменная b, используемая в функции g3 была определена в функции f3, на более высоком уровне стека вызовов.

Очевидно, в этом примере функция g3 написана плохо и может содержать программную ошибку. Действительно, мы видим, что входной аргумент a никогда не используется, а b используется, но не является входным аргументов. Вся проблема в том, что, в зависимости от контекста, это может привести к ошибке, а может и не привести: мы на самом деле не знаем.

4.5.3 Плохая функция: случай тихого обрушения

В этом разделе мы представляем типовое обрушение, вызванное неправильным использование области видимости переменных.

Следующая функция myalgorithm, предоставленная разработчиком, использует формулу производной первого прядка, основанную на прямой конечной разницы. Функция берёт текущую точку scivarx, функцию f и шаг h и возвращает приблизительную производную y.

```
// От разработчика
function y = myalgorithm ( x , f , h )
  y = (f(x+h) - f(x))/h
endfunction
```

Следующая функция myfunction, написанная пользователем, вычисляет многочлен второй степени по прямой формуле. Заметим, что вычисление выходного аргумента y использует точку x, которая является входным аргументом, и параметр a, который не является входным аргументом.

```
// От пользователя
function y = myfunction ( x )
  y = a(1) + a(2)*x + a(3)*x^2
endfunction
```

В следующем примере мы устанавливаем `a`, `x`, `h` и вызываем `myalgorithm` для того, чтобы вычислить численную производную. Мы сравниваем с точной производной и получаем хорошее соответствие.

```
-->a = [1 2 3];
-->x = 2;
-->h = sqrt(%eps);
-->y = myalgorithm ( x , myfunction , h )
 y  =
    14.
-->expected = a(2) + 2 * a(3)*x
 expected  =
    14.
```

Область видимости переменной `a` позволяет выполнять функцию `myfunction`. Действительно, переменная `a` определена пользователем на уровне №0 в стеке вызовов. Когда достигается тело функции `myfunction`, то интерпретатор находится на уровне №2, где переменная `a` используется, но не определена. Следовательно интерпретатор ищет переменную `a` на более высоком уровне стека вызовов. Нет такой переменной на уровне №1, поэтому используется переменная `a`, определённая на уровне №0, производящая, наконец, ожидаемый результат.

Фактически, этот способ использования области видимости переменных является опасной практикой программирования. На самом деле, пользователь делает предположение о внутреннем устройстве функции `myalgorithm`, и эти предположения могут быть ложными, ведущими к неверным результатам.

Мы слегка изменим функцию, предоставленную разработчиком, и переименуем шаг в `a`, вместо предыдущего `h`.

```
// От разработчика
function y = myalgorithm2 ( x , f , a )
  y = (f(x+a) - f(x))/a
endfunction
```

В следующем примере мы выполняем те же команды, что и прежде.

```
-->a = [1 2 3];
-->x = 2;
-->h = sqrt(%eps);
-->y = myalgorithm2 ( x , myfunction , h )
 !--error 21
Неправильный индекс.
at line       2 of function myfunction called by :
at line       2 of function myalgorithm2 called by :
y = myalgorithm2 ( x , myfunction , h )
```

Ошибка «Неправильный индекс» является следствием того факта, что переменная `a` была переписана в теле функции `myalgorithm2`, на уровне №1 в стеке вызовов. Как и ранее, когда достигается тело функции `myfunction` на уровне №2 стека вызовов, то интерпретатор ищет переменную `a`, которая содержит шаг формулы конечной разности. Эта переменная `a` является матрицей значений типа double только с одной ячейкой. Когда интерпретатор пытается выполнить инструкцию `a(1) + a(2)*x + a(3)*x^2`, это не получается, поскольку

ни a(2) ни a(3) не существуют, то есть, целые числа 2 и 3 являются неверными значениями индексов переменной a.

Предыдущая ошибка была, фактически милой. Действительно, она предупреждает нас о том, что что-то неправильное произошло, так что мы можем изменить наш файл-сценарий. В следующем случае набор команд пройдёт, но с неверным результатом, и это гораздо более опасная ситуация.

Мы снова изменим тело алгоритма, предоставленного разработчиком. В этот раз мы устанавливаем переменную a равной произвольной матрице значений типа double, как в следующей функции myalgorithm3.

```
// От разработчика
function y = myalgorithm3 ( x , f , h )
  a = [4 5 6]
  y = (f(x+h) - f(x))/h
endfunction
```

Тело функции несколько странное, поскольку мы не используем переменную a. но функция остаётся правильной. На самом деле это представляет более сложные случаи, где реализация функции myalgorithm3 использует полный, по возможности широкий, набор переменных. В этом случае вероятность использования некоторых имён переменных в качестве пользователя гораздо выше.

В следующем примере мы вызываем функцию myalgorithm3 и сравниваем с ожидаемым результатом.

```
-->a = [1 2 3];
-->x = 2;
-->h = sqrt(%eps);
-->y = myalgorithm3 ( x , myfunction , h )
 y  =
    29.
-->expected = a(2) + 2 * a(3)*x
 expected  =
    14.
```

Мы видим, что этот набор команд не формирует ошибки. Мы также видим, что ожидаемый результат абсолютно неверен. Как и прежде, переменная a была переопределена на уровне №1, в теле функции myalgorithm3. Следовательно, когда выполняется выражение a(1) + a(2)*x + a(3)*x^2, то используется значение [4 5 6] вместо матрицы, которую указал пользователь на более высоком уровне.

В нашем конкретном случае легко отладить проблему, поскольку мы имеем точную формулу для производной. На практике мы, вероятно, не будем иметь точную формулу (вот почему мы используем численные производные...), так что будет гораздо труднее обнаружить и, в случае обнаружения, исправить проблему.

Использование переменной, определённой на более высоком уровне стека вызовов, может считаться плохой практикой программирования. В случае, когда разработчик функции этого не желал, это может вести к программным ошибкам, которые трудно обнаружить и может оказаться незамеченным долгое время, пока ошибки не исправят.

Используя этот способ, область видимости переменной рассматривается как плохая практика программирования. По-прежнему, может так случиться, что функция требует более одного входного аргумента, которые надо вычислить. Эта конкретная ошибка приводится в разделе 4.6.4, где мы представляем метод для указания дополнительных входных аргументов функции обратного вызова.

4.6 Проблемы с функциями обратного вызова

В этом разделе, мы анализируем характерные проблемы функций обратного вызова. Сначала мы рассмотрим взаимодействия между именами функций пользователя и разработчика. Мы представляем два различных типа проблем на специальных примерах. Затем мы представляем методы, которые позволяют частично решить эту проблему. Заключении раздела, мы представляем метод, который позволяет управлять функциями обратного вызова с дополнительными аргументами. Метод, который мы предлагаем, основан на списках, которые дают хорошую гибкость, поскольку список может содержать любой другой тип данных.

Мы подчёркиваем, что две первые проблемы, которые мы собираемся представлять в разделах 4.6.1 и 4.6.2 не являются программными ошибками интерпретатора. Как мы увидим, эти проблемы порождены областью видимости переменных, которая была представлена в предыдущем разделе. Вероятно, что эти проблемы останутся в будущих версиях интерпретатора. Следовательно, стоит проанализировать их детальнее, так чтобы мы могли понять и эффективно решать эти особые проблемы.

4.6.1 Бесконечная рекурсия

В этом разделе мы представляем проблемы, которые порождают бесконечную рекурсию. Эта проблема случается, когда есть взаимодействие между именем функции, выбранным разработчиком и пользователем алгоритма. Поэтому наш анализ должен разделять точки зрения разработчика и пользователя.

Следующая функция `myalgorithm`, предоставленная разработчиком, берёт переменные `x` и `f` в качестве входных переменных и возвращает `y`. Предполагается, что переменная `f` является функцией и выражение `y=f(x)` вычисляется напрямую.

```
// На уровне разработчика
function y = myalgorithm ( x , f )
  y = f(x)
endfunction
```

Следующая функция `myfunction`, предоставленная пользователем, берёт в качестве входного аргумента `x` и возвращает `y`. Для того, чтобы вычислить `y`, пользователь применяет вторую функцию `f`, которую он написал для этого.

```
// На уровне пользователя
function y = myfunction ( x )
  y = f ( x )
endfunction
function y = f ( x )
  y = x(1)^2 + x(2)^2
endfunction
```

В следующем примере пользователь устанавливает переменную `x` и вызывает функцию `myalgorithm` со входными аргументами `x` и `myfunction`.

```
-->x = [1 2];
-->y = myalgorithm ( x , myfunction )
 !--error 26
Слишком сложная рекурсия! (заполнена таблица рекурсии)
at line       2 of function myfunction called by :
at line       2 of function myfunction called by :
at line       2 of function myfunction called by :
[...]
```

Эта рекурсия, которая в теории бесконечна, фактически конечна, поскольку Scilab разрешает только ограниченное количество вызовов `f`.

Причина этой ошибки заключается в конфликте имён функций разработчика и пользователя, которые в обоих случаях используют имя переменной `f`. С точки зрения пользователя, функция `myfunction` просто вызывает `f`. Но интерпретатор не видит подобного файла-сценария. С точки зрения интерпретатора, символы `myfunction` и `f` являются переменными, которые хранятся во внутренних структурах данных интерпретатора. Переменная `myfunction` имеет особый тип данных: она является функцией. То же самое справедливо для пользовательской `f`. Обе этих переменных определены на уровне №0 в стеке вызовов.

Следовательно, перед вызовом `myalgorithm`, на уровне №0, функция `f` определяется пользователем, с телом `"y = x(1)^2 + x(2)^2"`. Когда вызывается `myalgorithm`, которую мы вводим на уровень №1 и переменная `f` рассматривается в качестве входного аргумента: её содержимое `f=myfunction`. Следовательно, функция `f`, которая ранее определена пользователем, была переписана и потеряла своё первоначальное значение. Интерпретатор выполняет выражение `y = f ( x )`, которое вызывает обратно `myfunction`. В теле функции `myfunction`, на уровне №2, интерпретатор находит выражение `y = f ( x )`. Затем интерпретатор ищет переменную `f` на текущем уровне в стеке вызовов. Там нет такой переменной. Следовательно интерпретатор ищет переменную `f` на более высоком уровне в стеке вызовов. На уровне №1 переменная `f` определена с содержанием `f=myfunction`. Это значение, следовательно, используется на уровне №2. Интерпретатор выполняет выражение `y = myfunction ( x )` и выходит на уровень №3. Затем интерпретатор ищет переменную `f` на текущем уровне. На этом уровне нет такой переменной. Как и прежде, интерпретатор ищет переменную `f` на более высоком уровне стека вызовов. И снова интерпретатор вызывает функцию `f`, которая является фактически самой функцией `myfunction`. Этот процесс повторяется снова и снова до тех пор, пока интерпретатор не достигнет максимального разрешённого размера стека вызовов и сформируется ошибка. Следующий пример показывает последовательность событий.

```
-->y = myalgorithm ( x , myfunction )
-1->f = myfunction
-1->y = f(x)
-1->y = myfunction(x)
-2->y = f ( x )
-2->y = myfunction ( x )
etc...
```

Подчеркнём, что проблема является следствием области видимости переменных в языке Scilab'а. Это напрямую означает следующие возможности языка.

- Функция хранится как любая другая переменная (но с особым типом данных).
- Если переменная неизвестна на уровне №k, то её ищут на более высоком уровне в стеке вызовов до тех пор, пока её или не найдут или не будет сформирована ошибка (а именно «Неизвестная переменная»).

Заметим, что это не значит, что пользователь знает, что разработчик использовал имя переменной `f`. Обратное так же верно для разработчика, который не может предугадать, что пользователь будет использовать переменную `f`. Заметим, что нет простого способа для пользователя изменить эту ситуацию. Функция `myalgorithm` может быть предоставлена внешним модулем. В этом случае смена имени переменной, используемой в функции разработчика может быть невозможным для пользователя. Как мы увидим позднее в этом

разделе, пользователь по-прежнему может найти решение на основе небольшой реорганизации исходного кода.

4.6.2 Неправильный индекс

В этом разделе мы представляем ситуацию, где интерпретатор формирует ошибку «Неправильный индекс» из-за конфликта между именами функций пользователя и разработчика.

В следующей функции `myalgorithm`, предоставленной разработчиком, мы устанавливаем локальную переменную `f` в 1. Затем мы вызываем обратно функцию `userf` с входным аргументом `x` и возвращаем выходной аргумент `y`.

```
// На уровне разработчика
function y = myalgorithm ( x , userf )
  f = 1
  y = userf(x)
endfunction
```

Заметим, что переменная `f` установлена, но не используется. Функция по-прежнему годная, и это не меняет наш анализ.

Следующая функция `myfunction`, предоставленная пользователем, вызывает функцию `f` с входным аргументом `x` и возвращает выходной аргумент `y`.

```
// На уровне пользователя
function y = myfunction ( x )
  y = f ( x )
endfunction
function y = f ( x )
  y = x(1)^2 + x(2)^2
endfunction
```

В следующем примере мы устанавливаем переменную `x` и вызываем функцию `myalgorithm`. Это формирует ошибку «Неправильный индекс» в теле функции `myfunction`.

```
-->x = [1 2];
-->y = myalgorithm ( x , myfunction )
 !--error 21
Неправильный индекс.

at line       2 of function myfunction called by :
at line       3 of function myalgorithm called by :
y = myalgorithm ( x , myfunction )
```

Объяснение следующее. На уровне №0 в стеке вызова, функция `f` определена так, как ожидает пользователь. В теле функции `myalgorithm` мы находимся на уровне №1 в стеке вызовов. Переменная `f` обновлена командой `f=1`: теперь `f` является матрицей значений типа double. Затем интерпретатор выполняет выражение `y = userf(x)`. Поскольку `userf` — это `myfunction`, то интерпретатор вызывает функцию `myfunction` и входит на уровень №2 в стеке вызовов. На этом уровне интерпретатор выполняет выражение `y = f ( x )`. Затем интерпретатор ищет переменную `f` на уровне №2: нет такой переменной на этом уровне. Затем интерпретатор ищет переменную `f` на более высоком уровне. На уровне №1 интерпретатор находит переменную `f`, которая является матрицей значений типа double. В этом контексте выражение `y = f ( x )` не имеет смысла: переменная `x` интерпретируется как индекс матрицы значений типа double `f`. Поскольку `x=[1 2]`, то интерпретатор ищет ячейки

с индексами 1 и 2 в `f`. Но `f` содержит только матрицу значений типа double размером 1×1. Следовательно есть только одна ячейка в `f`, поэтому формируется ошибка «Неправильный индекс».

Как и в предыдущем разделе, пользователь не может предугадать имена переменных, выбранных разработчиком, а обратное верно для разработчика.

В следующем разделе мы предлагаем методы решения этих проблем функций обратного вызова.

4.6.3 Решения

В этом разделе мы представляем решения для проблем функций обратного вызова, которые мы представили ранее. Решение проблем может быть сделано двумя разными способами.

- Изменить функцию разработчика так, чтобы было меньше шансов получить конфликт с именами пользовательских переменных.

- Изменить функцию пользователя так, чтобы он мог использовать функцию разработчика.

Очевидно, что мы можем также использовать и тот и другой. В общем, это задача разработчика предоставлять функции, которые достаточно хорошо разработаны, так что пользователь не должен озадачиваться странными программными ошибками. Поэтому мы представляем это обновление первым. Обновление пользовательской функции представлено во второй часть этого раздела.

Следующая модифицированная функция представляет метод, который разработчик может использовать для написания функции `myalgorithm`. Мы можем видеть, что второй аргумент `__myalgorithm_f__` имеет длинное и сложное имя. Это уменьшает вероятность получения конфликтов между именами переменных пользователя и разработчика.

```
function y = myalgorithm ( x , __myalgorithm_f__ )
  y = __myalgorithm_f__(x)
endfunction
```

Это обходной манёвр: по-прежнему есть вероятность, что пользователь получит ошибку. Действительно, если пользователь выберет имя переменной `__myalgorithm_f__`, то останется конфликт между именами переменных пользователя и разработчика. Но это весьма маловероятно.

Следующая модифицированная функция представляет метод, который пользователь может использовать для написания функции `myfunction`. Вместо определения функции `f` вне тела функции `myfunction`, мы определяем её внутри. В новой версии переменная `f` на этот раз определена локально, внутри `myfunction`. Это ограничивает видимость локальной функции `f`, к которой может получить доступ `myfunction` (и более низкие уровни), и она больше не появляется в глобальной области видимости. Следовательно, когда мы вызываем `f`, то интерпретатор не ищет на более высоких уровнях стека вызовов: переменная `f` определена на текущем уровне.

```
function y = myfunction ( x )
  function y = f ( x )
    y = x(1)^2 + x(2)^2
  endfunction
  y = f ( x )
endfunction
```

В следующем примере мы вызываем функцию myalgorithm с обновлённой функцией myfunction.

```
-->x = [1 2];
-->y = myalgorithm ( x , myfunction )
 y  =
    5.
```

Как мы можем видеть, теперь проблема решена и мы получаем ожидаемый результат.

4.6.4 Функции обратного вызова с дополнительными аргументами

В этом разделе мы представляем метод, который позволяет решить проблемы, связанные с функциями обратного вызова с дополнительными аргументами. Метод, который мы предлагаем основан на списках, которые предоставляют хорошую гибкость, поскольку они могут содержать любой другой тип данных.

Когда мы рассматриваем алгоритм, который берёт функцию обратного вызова в качестве входного аргумента, то заголовок функции, которая вызывается обратно, вообще определённо фиксирован, по выбору разработчика алгоритма. Но пользователь алгоритма может захотеть указать функцию, которая не точно совпадает с требуемым заголовком. Например, алгоритм может ожидать функцию с заголовком y=f(x), а пользователь имеет функцию, которая требует дополнительный параметр a. Для того, чтобы избежать эту путаницу, созданную использованием области видимости переменных, пользователь может выбрать обновление заголовка функции так, чтобы переменная a была теперь входным аргументом. Это ведёт к заголовку y=g(x,a). В этом случае мы предполагаем, что дополнительный аргумент a является константой, что значит, что алгоритм не изменяет её содержимое. Простое указание функции g не может работать, поскольку заголовок не совпадает. Для того, чтобы проиллюстрировать эту ситуацию, мы анализируем случай, когда мы хотим вычислить числовую производную.

Следующая функция myderivative1 использует прямую формулу конечной разности для вычисления числовой производной.

```
function fp = myderivative1 ( __myderivative_f__ , x , h )
  fp = (__myderivative_f__(x+h) - __myderivative_f__(x))/h
endfunction
```

Мы рассмотрим функцию myfun, которая вычисляет косинус многочлена второй степени.

```
function y = myfun ( x )
  y = cos(1+2*x+3*x^2)
endfunction
```

В следующем примере мы сравниваем числовую производную с точной производной.

```
-->format("e",25)
-->x = %pi/6;
-->h = %eps^(1/2);
-->fp = myderivative1 ( myfun , x , h )
 fp  =
  - 1.380975857377052307D+00
-->expected = -sin(1+2*x+3*x^2) * (2+6*x)
 expected  =
  - 1.380976033975957140D+00
```

Поскольку два значения хорошо совпадают, то мы теперь уверены в нашей реализации.

Но было бы яснее, если параметры многочлена были сохранены в матрице значений типа double. Это приводит к следующей функции `myfun2`, которая берёт точку `x` и параметр `a` в качестве входных аргументов.

```
function y = myfun2 ( x , a )
  y = cos(a(1)+a(2)*x+a(3)*x^2)
endfunction
```

Для того, чтобы управлять данной ситуацией, мы модифицируем реализацию алгоритма конечной разности и создадим функцию `myderivative2`, которая будет подробно рассмотрена позже в этом разделе. В следующем примере мы вызываем `myderivative2` и указываем список `list(myfun2,a)` в качестве первого аргумента. Функция `myderivative2` предполагает, что первый элемент списка — это имя функции, и что остальные элементы списка являются дополнительными аргументами функции, которая будет выполнена. Здесь единственный дополнительный элемент — это `a`.

```
-->x = %pi/6;
-->a = [1 2 3];
-->h = %eps^(1/2);
-->fp = myderivative2 ( list(myfun2,a) , x , h )
 fp  =
  - 1.380975857377052307D+00
-->expected = -sin(a(1)+a(2)*x+a(3)*x^2) * (a(2)+2*a(3)*x)
 expected  =
  - 1.380976033975957140D+00
```

Следующая функция `myderivative2` даёт гибкую реализацию числовой производной, которая позволяет различать дополнительные аргументы в заголовке функции.

```
1  function fp = myderivative2 ( __myderivative_f__ , x , h )
2    tyfun = typeof(__myderivative_f__)
3    if ( and(tyfun<>["list" "function" "fptr"]) ) then
4      error ( msprintf("%s: Unknown function type: %s",..
5        "myderivative2",tyfun))
6    end
7    if ( tyfun == "list" ) then
8      nitems = length(__myderivative_f__)
9      if ( nitems<2 ) then
10       error ( msprintf("%s: Too few elements in list: %d",..
11         "myderivative2",nitems))
12     end
13     __myderivative_f__fun__ = __myderivative_f__(1)
14     tyfun = typeof(__myderivative_f__fun__)
15     if ( and(tyfun<>["function" "fptr"]) ) then
16       error ( msprintf("%s: Unknown function type: %s",..
17         "myderivative2",tyfun))
18     end
19     fxph=__myderivative_f__fun__(x+h,__myderivative_f__(2:$))
20     fx = __myderivative_f__fun__ ( x , __myderivative_f__(2:$))
21   else
22     fxph = __myderivative_f__ ( x + h )
23     fx = __myderivative_f__ ( x )
24   end
25   fp = (fxph - fx)/h
26 endfunction
```

В строке №2 мы выясняем тип входного аргумента `__myderivative_f__`. Если этот аргумент не является ни списком ни указателем функции (т. е. примитивом), то мы формируем

ошибку. Тогда код рассматривает два случая. Если первый аргумент является списком, то мы проверяем в строках с №8 по №12 число пунктов списка. Затем, в строке №13, мы сохраним первый элемент в отдельной переменной, которая предполагается является функцией. В строках с №14 по №18 мы проверяем тип этой переменной и формируем ошибку, если переменная не является функцией. Затем в строках №19 и №20 мы вычисляем два значения функции `fxph` и `fx`. Заметьте, что мы используем выражение `__myderivative_f__(2:$)` для того, чтобы предоставить дополнительные аргументы вызывающей функции. Из-за особого способа, которым элементы выделяются из списка, это создаёт ряд входных аргументов, которые требуются заголовком функции, которую вызвали.

На практике метод, который мы уже представили чрезвычайно гибок. Функция может предоставить более одного дополнительного аргумента. Фактически, число дополнительных аргументов может быть уменьшено использованием сбора списка всех необходимых параметров. Действительно, списки могут быть вложены, что позволяет собрать требуемые параметры в единственную дополнительную переменную.

4.7 Мета-программирование: `execstr` и `deff`

В этом разделе мы представляем функции, которые позволяют выполнить инструкции, которые определены как строки. Поскольку эти строки могут формироваться динамически, мы можем сделать программы, которые имеют уровень гибкости, который не может быть достигнут другими средствами. В первой части мы представляем функцию `execstr` и даём пример использования этой чрезвычайно мощной функции. Во второй части мы представляем функцию `deff`, которая позволяет динамически создавать функции, основанные на двух строках, содержащих заголовок и тело функции. В заключительной части мы представляем практическое использование функции `execstr`, где мы объединяем два модуля, которые не могут модифицироваться пользователем и которые не совпадают точно.

Рисунок 26 представляет функции, которые позволяют динамически выполнять инструкции, основанные на строках.

`deff`	оперативное определение функции
`execstr`	выполнение инструкций в строке
`evstr`	выполнение строки

Рис. 26: Функции мета-программирования.

4.7.1 Основное использование `execstr`

В этом разделе мы представляем функцию `execstr`.

Функция `execstr` берёт в качестве своего первого входного аргумента строку, которая содержит правильную инструкцию Scilab'а. Затем функция выполняет инструкцию как если бы это было частью оригинального файла-сценария на том же самом уровне стека вызовов.

В следующем примере мы динамически формируем строку, содержащую инструкцию, которая отображает строку `"foo"`. Сначала мы определяем переменную `s`, которая содержит строку `"foo"`. Затем мы устанавливаем переменную `instr`, конкатенируя переменную `s` со строками `disp("` и `")`. Это выполняется оператором `+`, который позволяет конкатенировать свои операнды. Двойные кавычки `"` дублируются, что делает так, что выражение

интерпретируется как символ " внутри целевой строки. Действительно, если эти кавычки не появятся дважды, то это будет интерпретироваться как конец строки: дублирование кавычек позволяет «избежать» знака кавычек. Наконец, мы запустим функцию `execstr`, которая распечатывает `"foo"`.

```
-->s = "foo"
 s  =
 foo
-->instr = "disp(""" + s + """)"
 instr  =
 disp("foo")
-->execstr(instr)
 foo
```

Аргумент `execstr` может содержать любую правильную инструкцию Scilab'а. Например, мы можем динамически установить новую переменную с динамически созданным содержанием. В следующем примере мы создаём строку `instr`, содержащую инструкцию `z=1+2`. Затем мы выполняем это выражение и проверяем, что переменная `z` действительно установлена равной 3.

```
-->x = 1
 x  =
    1.
-->y = 2
 y  =
    2.
-->instr = "z="+string(x)+"+"+string(y)
 instr  =
 z=1+2
-->execstr(instr)
-->z
 z  =
    3.
```

В качестве другого примера использования `execstr` мы можем читать коллекцию файлов данных. Предположим, что файл `datafile1.txt` содержит строки

```
1 2 3
4 5 6
```

и файл `datafile2.txt` содержит строки

```
7 8 9
10 11 12
```

Следующий пример позволяет читать эти два файла последовательно. Функция `read` берёт строку, представляющую файл, содержащий матрицу элементов типа double и количество строк и столбцов в матрице. Набор команд устанавливает переменную `data`, которая содержит список матрицы, содержащей данные, прочитанные в двух файлах.

```
data = list();
for k = 1 : 2
  instr = "t = read(""datafile" + string(k) + ".txt"",2,3)"
  execstr(instr)
  data($+1)=t
end
```

Следующий пример показывает динамику предыдущего примера.

```
instr  =
t = read("datafile1.txt",2,3)
data  =
      data(1)
   1.    2.    3.
   4.    5.    6.
instr  =
t = read("datafile2.txt",2,3)
data  =
      data(1)

   1.    2.    3.
   4.    5.    6.
      data(2)
   7.     8.     9.
   10.    11.    12.
```

4.7.2 Основное использование deff

Функция `deff` позволяет динамически определять функцию, из двух строк, содержащих заголовок и тело функции.

В следующем примере мы определим функцию `myfun`, которая принимает матрицу значений типа double `x` в качестве входного аргумента и возвращает выходной аргумент `y`. Сначала мы определяем заголовок функции в виде строки `"y=myfun(x)"` и устанавливаем переменную `header`. Затем мы определяем тело функции и, наконец, мы вызываем функцию `deff`, которая создаёт требуемую функцию.

```
-->header = "y=myfun(x)"
 header  =
 y=myfun(x)
-->body = [
-->"y(1) = 2*x(1)+x(2)-x(3)^2"
-->"y(2) = 2*x(2) + x(3)"
-->]
 body  =
!y(1) = 2*x(1)+x(2)-x(3)^2  !
!                           !
!y(2) = 2*x(2) + x(3)       !
-->deff(header,body)
```

Следующий пример показывает, что новая функция `myfun` может быть использована как любая другая функция.

```
-->x = [1 2 3]
 x  =
   1.    2.    3.
-->y = myfun(x)
 y  =
  - 5.
    7.
```

Единственная разница заключается в типе функции, созданной с помощью `deff`, как представлено в разделе 4.1.1. В следующем примере мы показываем, что функция, созданная с помощью `deff`, имеет тип 13, который соответствует компилированному макросу.

```
-->type(myfun)
```

```
 ans  =
    13.
-->typeof(myfun)
 ans  =
 function
```

Если мы добавим аргумент "n" к вызову функции deff, то это скажет интерпретатору создать вместо этого некомпилированный макрос, который производит функцию с типом 11.

```
-->deff(header,body,"n")
Warning : redefining function: myfun.
Use funcprot(0) to avoid this message
-->type(myfun)
 ans  =
    11.
-->typeof(myfun)
 ans  =
 function
```

Мы подчёркиваем, что для того, чтобы создать новую функцию, могла бы использоваться функция execstr вместо deff. Например, следующий набор команд показывает способ определения новой функции с помощью execstr отдельным определением заголовка, тела и окончания функции. Эти строки затем конкатенируются в единую инструкцию, которая, наконец, исполняется для того чтобы определить новую функцию.

```
header = "function y = myfun(x)"
body = [
"y(1) = 2*x(1)+x(2)-x(3)^2"
"y(2) = 2*x(2) + x(3)"
]
footer = "endfunction"
instr = [
header
body
footer
]
execstr(instr)
```

4.7.3 Практический пример оптимизации

В этом разделе мы представляем практический случай использования методов, которые мы представили в предыдущих разделах.

Этом примере мы используем функцию optim, которая является программой для поиска решения числовой оптимизации. Мы представляем ситуацию, когда нам нужна функция для определения функции адаптера (т. е. «клей»), которая соединяет optim с модулем, предоставляющим набор задач оптимизации. В первом методе, мы определяем функцию адаптера и передаём функцию обратного вызова с дополнительными аргументами в функцию optim. Во втором методе, мы используем функцию execstr для создания функции, чьё тело формируется динамически.

Пример, который мы выбрали, может показаться сложным. Хотя на практике он представляет практическую задачу разработки программного обеспечения, где могут использоваться только нетривиальные решения.

Предположим, что нас интересует расчёт характеристик функции нелинейной оптимизации `optim`, предоставляемой Scilab. В этом случае, мы можем использовать модуль Atoms `uncprb`, который предоставляет коллекцию из 23 тестовых задач неограниченной оптимизации, которая известна как коллекция Морэ, Гарбоу и Хильстрома (More, Garbow, Hillstrom). Для того, чтобы установить этот модуль, мы используем следующую инструкцию:

```
atomsInstall("uncprb")
```

и перезапускаем Scilab.

Мы подчёркиваем, что мы рассматриваем здесь точку зрения пользователя, который не может модифицировать ни один из этих пакетов, то есть, не может модифицировать ни функцию `optim`, ни модуль `uncprb`. Как мы вскоре увидим, нет точного совпадения между заголовком целевой функции, которая требуется для функции `optim` и заголовок тестовых функций. которые предоставлены модулем `uncprb`. Точнее, число и тип входных и выходных аргументов, которые требуются для функции `optim`, не совпадают с числом и типом входных и выходных аргументов, которые предоставляются модулем `uncprb`. Это не из-за плохого устройства обоих инструментов: фактически невозможно предоставить «универсальный» заголовок, удовлетворяющий всевозможным нуждам. Следовательно, мы должны создать промежуточную функцию, которая делает «клей» между двумя этими компонентами.

Модуль `uncprb` предоставляет значение функции, градиент и вектор функции и якобиан для всех тестовых задач, и предоставляет матрицу Гессе для 18 задач. Точнее, этот модуль предоставляет следующие функции, где `nprob` — номер задачи от 1 до 23.

```
[n,m,x0]=uncprb_getinitf(nprob)
f=uncprb_getobjfcn(n,m,x,nprob)
g=uncprb_getgrdfcn(n,m,x,nprob)
```

Функция `uncprb_getinitf` возвращает размер `n` задачи, число `m` функции и начальное предположение `x0`. Действительно, существующая целевая функция является суммой квадратов `m` функций. Функция `uncprb_getobjfcn` возвращает значение целевой функции, при этом функция `uncprb_getgrdfcn` возвращает градиент.

Функция `optim` является программой поиска решения нелинейной неограниченной оптимизации, предоставляющей несколько алгоритмов для этого класса задач. Она может управлять неограниченными или частично ограниченными задачами. Самая простая инструкция вызова функции `optim`:

```
[fopt,xopt]=optim(costf,x0)
```

где `costf` — функция (т. е. целевая) цены, которую нужно минимизировать, `x0` — начальная точка (т. е. начальное предположение), `fopt` — значение минимума функции, а `xopt` — точка, которая достигает это значение. Функция цены `costf` должна иметь заголовок

```
[f,g,ind]=costf(x,ind)
```

где `x` — текущая точка, `ind` — целое число с плавающей запятой, представляющее что должно быть вычислено, `f` — значение функции, `g` — градиент. На выходе `ind` — это флаг, посылаемый функцией программе поиска решения оптимизации, например чтобы прервать алгоритм.

Мы должны сначала получить параметры задачи, так, чтобы мы могли установить клей-функцию. В следующем примере мы получаем параметры первой задачи оптимизации.

```
nprob = 1;
[n,m,x0] = uncprb_getinitf(nprob);
```

Вопрос в том как склеить эти два компонента так, чтобы optim могла использовать uncprb? Очевидно, что мы не можем передать ни uncprb_getobjfcn ни uncprb_getgrdfcn в качестве входных аргументов в optim, поскольку целевая функция должна вычислить и значение функции и градиент. Решение этой задачи заключается в определении промежуточной функции, которая имеет заголовок, требуемый функцией optim, и который вызывает функции uncprb_getobjfcn и uncprb_getgrdfcn. Единственной проблемой является управление nprob, n и m, которое зависит от задачи, которую нужно решить. Чтобы это сделать, возможны два разных способа.

- Мы можем определить функцию с дополнительными аргументами nprob, n и m и использовать специальные возможности optim для управления ими.
- Мы можем определить функцию, которая динамически определяется с помощью так, что в конце переменные nprob, n и m являются константами.

Далее мы изучим оба метода.

Первый метод основан на управлении дополнительными аргументами функции обратного вызова, которые были представлены в разделе 4.6.4. Определим целевую функцию costfun, которая имеет ожидаемую последовательность вызова, с дополнительными аргументами nprob, n и m.

```
function [f,g,ind]=costfun(x,ind,nprob,n,m)
    [n,m,x0] = uncprb_getinitf(nprob)
    // Делаем вектор-столбец
    x = x(:)
    f = uncprb_getobjfcn(n,m,x,nprob)
    g = uncprb_getgrdfcn(n,m,x,nprob)
endfunction
```

Затем, мы можем использовать специальные возможности функции optim, которые управляют случаем, в котором для целевой функции нужны дополнительные входные аргументы. Аргумент costf может также быть списком (myfun,a1,a2,...). В этом случае myfun, первый элемент списка, должна быть функцией и должна иметь заголовок:

```
    [f,g,ind]=myfun(x,ind,a1,a2,...)
```

Мы используем эту возможность в следующем файле-сценарии:

```
nprob = 1;
[n,m,x0] = uncprb_getinitf(nprob);
myobjfun = list(costfun,nprob,n,m);
[fopt,xopt]=optim(myobjfun,x0)
```

Предыдущий файл-сценарий производит следующий вывод:

```
-->[fopt,xopt]=optim(myobjfun,x0)
 xopt  =
    1.
    1.
 fopt  =
    0.
```

Мы знаем, что глобальный минимум этой задачи равен $\mathbf{x}^\star = (1,1)$, так что предыдущий результат верен.

Второй метод основан на динамическом создании целевой функции с помощью функции execstr.

В следующем примере мы определяем функцию objfun, которая будет передана в качестве входного аргумента функции optim. Тело функции objfun динамически определяется использованием переменных nprob, m и n, которые были определены ранее. Заголовок, тело и окончание функции затем конкатенируются в матрицу строковых элементов instr.

```
header = "function [fout,gout,ind]=objfun(x,ind)";
body = [
"fout=uncprb_getobjfcn("+string(n)+","+string(m)+",x,"+string(nprob)+")"
"gout=uncprb_getgrdfcn("+string(n)+","+string(m)+",x,"+string(nprob)+")"
];
footer = "endfunction"
instr = [
header
body
footer
]
```

Предыдущий исходный код довольно абстрактный. Фактически генерируемая функция простая, как та, что показана в следующем примере.

```
-->instr
 instr  =
!function [fout,gout,ind]=objfun(x,ind)  !
!fout=uncprb_getobjfcn(2,2,x,1)          !
!gout=uncprb_getgrdfcn(2,2,x,1)          !
!endfunction                             !
```

Как мы можем видеть, входной аргумент x функции objfun единственный оставшийся: остальные параметры заменены их действительными значениями для этой конкретной задачи. Следовательно, функция objfun явно определена и не использует своё окружение для получения, например, значения nprob. Мы подчёркиваем этот особый момент, который показывает, что использование области видимости переменных через стек вызовов, как представлено в разделе 4.5.1, может быть исключено использованием функции execstr.

Для того, чтобы определить целевую функцию, мы, наконец, используем execstr.

```
execstr(instr)
```

В следующем примере мы вызываем функцию optim и вычисляем решение первой тестовой задачи.

```
-->[fopt,xopt]=optim(objfun,x0)
 xopt  =
    1.
    1.
 fopt  =
    0.
```

4.8 Заметки и ссылки

Янн Коллетт (Yann Collette) разработал модуль parameters как инструмент для предоставления тех же возможностей, что и у функций optimset/optimget в Matlab. Он заметил, что функции optimset/optimget нельзя было настроить снаружи: нам приходится модифицировать функции для того, чтобы добавить новую опцию. Вот почему модуль parameters был создан таким образом, который позволяет позволить пользователю управлять столькими опциями, сколько потребуется.

Область видимости переменных представлена в разделе 4.5.1. Эта тема также представлена в [50].

Проблемы с функциями обратного вызова были представлены в разделе 4.6. Эта тема была проанализирована в сообщениях о программных ошибках №7102, №7103 и №7104 [12, 11, 9].

В [49] Энрико Сегре (Enrico Segre) описал несколько возможностей, связанных с функциями.

В разделе 4.3 мы представили шаблон для разработки устойчивых функций. Используя этот шаблон можно сделать код совместимым с соглашением по коду, представленным Пьером Марешалем (Pierre Maréchal) в [33].

В разделе 4.3 мы представляем правила написания устойчивых функций. Написание такой устойчивой функции требует большого числа проверок и может привести к дублированию кода. Модуль «apifun» [8] является экспериментальной попыткой предоставить API для проверки входных аргументов с бо́льшей простотой.

5 Производительность

В этом разделе мы представляем ряд уловок программирования в Scilab, которые позволяют делать *быстрые* файлы-сценарии. Эти методы известны как *векторизация* и являются ядром наиболее эффективных функций Scilab.

Мы покажем, как использовать функции `tic` и `toc` для измерения характеристик алгоритма. Во втором разделе мы анализируем исходный алгоритм и проверяем, что векторизация может кардинально улучшить характеристики, обычно на множитель от 10 до 100, но иногда и более. Затем, мы анализируем методы компилирования и представляем связь между проектированием интерпретатора и характеристиками. Мы представляем возможности профилирования Scilab и даём пример векторизации алгоритма метода исключения Гаусса. Мы представляем принципы векторизации и сравниваем характеристики циклов с характеристиками векторизованных инструкций. В следующем разделе мы представляем различные методы оптимизации, основанные на практических примерах. Мы представляем библиотеки линейной алгебры, используемые в Scilab. Мы представляем числовые библиотеки BLAS, LAPACK, ATLAS и Intel MKL, которые предоставляют оптимизированные функции линейной алгебры и можем увидеть существенную разницу относительно характеристик матричных операций. В последнем разделе мы представляем различные измерения производительности Scilab'а, основанные на подсчёте операций с плавающей запятой.

На рисунке 27 представлены некоторые функции Scilab, которые используются в контексте анализа характеристик файлов-сценариев Scilab.

`tic`, `toc`	измерение пользовательского времени
`timer`	измерение системного времени

Рис. 27: Функции измерения производительности.

5.1 Измерение производительности

В этом разделе мы описываем функции, которые позволяют измерить время, требуемое на вычисление. Действительно, перед оптимизацией любой программы нам следует точно

измерить её текущую производительность.

В первом разделе мы представляем функции `tic`, `toc` и `timer`. Затем мы сравниваем эти функции и подчёркиваем из отличие на многоядерных машинах. Мы представляем функциональности профилирования в Scilab'е, которые позволяю проанализировать части алгоритма, которые наиболее затратные. Наконец, мы представляем функцию `benchfun`, которая позволяет провести простой и надёжный анализ производительности.

5.1.1 Основные применения

Для того, чтобы измерить время, требуемое на вычисление, мы можем использовать функции `tic` и `toc`, которые измеряют *пользовательское* время в секундах. Пользовательское время — время настенных часов, то есть, время, которое требуется компьютеру от начала задачи до её конца. Это включает и само вычисление, конечно, но также все остальные операции системы, такие как обновление экрана, обновление файловой системы, позволение другим процессам делать часть их работы и т. д. В следующем примере мы используем функции `tic` and `toc` для измерения времени вычисления собственных значений случайной матрицы на основе функции `spec`.

```
-->tic(); lambda = spec(rand(200,200)); t = toc()
 t  =
    0.1
```

Вызов функции `tic` запускает счётчик, а вызов функции `toc` — останавливает его, возвращая число прошедших секунд. В этом случае мы печатаем инструкции в одной той же строке, используя разделитель `;`: действительно, если бы мы напечатали инструкции интерактивно на нескольких строчках, то измеренное время было бы главным образом временем, требуемым на набор инструкций с клавиатуры.

Предыдущее измерение времени не очень надёжно в том смысле, что если бы мы выполнили одни и те же инструкции несколько раз, то не получили бы одного и того же времени. Следующий пример демонстрирует это поведение.

```
-->tic(); lambda = spec(rand(200,200)); t = toc()
 t  =
    0.18
-->tic(); lambda = spec(rand(200,200)); t = toc()
 t  =
    0.19
```

Возможное решение этой проблемы заключается в том, чтобы выполнить одни и то же вычисление несколько раз, скажем, 10 раз, например, и затем распечатать минимальное, максимальное и среднее время.

```
for i = 1 : 10
  tic();
  lambda = spec(rand(200,200));
  t(i) = toc();
end
[min(t) mean(t) max(t)]
```

Предыдущий пример даёт следующий вывод.

```
-->[min(t) mean(t) max(t)]
 ans  =
    0.141    0.2214    0.33
```

Если другая программ на компьютере использует ЦП в то же время, когда Scilab выполняет свои вычисления, то пользовательское время может возрасти. Следовательно функции `tic` и `toc` не используются в ситуациях, когда имеет значение только время ЦП. В этом случае мы можем вместо этого использовать функцию `timer`, которая измеряет время системы. Оно соответствует времени, требуемому ЦП для выполнения специфических вычислений. требуемых Scilab'ом, а не другими процессами.

Следующий пример представляет функцию `timer`.

```
for i = 1 : 10
  timer();
  lambda = spec(rand(200,200));
  t(i) = timer();
end
[min(t) mean(t) max(t)]
```

The previous script produces the following output.

```
-->[min(t) mean(t) max(t)]
 ans  =
    0.100144    0.1161670    0.1602304
```

5.1.2 Пользовательское время и время ЦП

В этом разделе мы анализируем разницу между пользовательским временем и временем ЦП на одно- и многоядерных системах.

Давайте рассмотрим следующий файл-сценарий, который позволяет выполнить произведение двух квадратных матриц. Мы измеряем пользовательское время функциями `tic` и `toc`, а время ЦП измеряем функцией `timer`.

```
stacksize("max")
n = 1000;
A = rand(n,n);
B = rand(n,n);
timer();
tic();
C = A * B;
tUser = toc();
tCpu = timer();
disp([tUser tCpu tCpu/tUser])
```

Следующий пример представляет результат, когда мы запустили этот файл-сценарий на Linux-машине с одним ядром на 2 ГГц.

```
-->disp([tUser tCpu tCpu/tUser])
    1.469    1.348084    0.9176882
```

Как мы можем видеть два времени довольно близки и показывают, что 91 % времени настенных часов потрачено центральным процессором на конкретное вычисление.

Теперь рассмотрим следующий пример, выполненный на Windows-машине с 4-мя ядрами, где Scilab использует многопотоковую Intel MKL.

```
-->disp([tUser tCpu tCpu/tUser])
    0.245    1.0296066    4.2024759
```

Мы видим что есть существенная разница между временем ЦП и временем настенных часов. Отношение близко к 4, что равно числу ядер. Действительно, на многоядерной машине, если Scilab использует многопотоковую библиотеку, функция `timer` сообщает сумму времён,

потраченных на каждое ядро. Вот почему во многих ситуациях нам следует использовать функции `tic` и `toc` для измерения производительности алгоритма.

5.1.3 Профилирование функции

В этом разделе мы представляем возможности профилирования в Scilab'е. Профиль функции раскрывает части функции, которые наиболее затратны. Это позволяет нам сосредоточиться на частях исходного кода, которые наиболее всего требуется улучшить.

Сначала мы рассмотрим исходную, медленную реализацию алгоритма метода исключения Гаусса и проанализируем её профиль. Затем мы используем векторизацию для того, чтобы ускорить вычисления и проанализируем этот обновлённый профиль.

Функции в таблице 28 позволяют управлять профилированием функций.

`add_profiling`	Добавляет команды профилирования в функцию
`plotprofile`	Выделяет и отображает профили исполнения
`profile`	Выделение профилей исполнения
`remove_profiling`	Удаляет инструкции профилирования
`reset_profiling`	Сбрасывает счётчики профилирования
`showprofile`	Выделяет и отображает профили исполнения

Рис. 28: Команды Scilab'а для профилирования функций.

В этом разделе мы рассматриваем решение систем линейных уравнений $\mathbf{Ax} = \mathbf{b}$, где $\mathbf{A}$ — вещественная матрица размером $n \times n$, а $\mathbf{b}$ — вектор-столбец размером $n \times 1$. Одним из наиболее стабильных алгоритмов для этой задачи является метод исключения переменных Гаусса с перестановкой строк. Метод исключения переменных Гаусса с перестановкой строк используется в Scilab'е с помощью оператора обратный слеш \. Для того, чтобы проиллюстрировать профилирование функции, мы не будем использовать оператор обратного слеша. Вместо этого мы разработаем наш собственный алгоритм метода исключения Гаусса (который, очевидно, на практике не является хорошей идеей). Давайте вспомним, что первый шаг алгоритма требует разложить $\mathbf{A}$ в виде $\mathbf{PA} = \mathbf{LU}$, где $\mathbf{P}$ — матрица перестановок, $\mathbf{L}$ — нижняя треугольная матрица, а $\mathbf{U}$ — верхняя треугольная матрица. Сделав однажды, алгоритмы продолжают выполнение прямого и, затем, обратного исключения.

Алгоритм и короток и сложен, и фактически довольно интересный. Мы не ставим целью получение этого алгоритма через презентацию в этом документе (см. Голуб и Ван Лоан (Golub и Van Loan) [21] для более полного анализа). Этот алгоритм является хорошим кандидатом для профилирования, поскольку он более сложен, чем средняя функция. Следовательно, от него не получить напрямую хорошую производительность. Этот тот тип ситуации, где профилирование наиболее полезно. Действительно, интуиции иногда не достаточно, чтобы точно обнаружить «виновные строки», то есть строки, которые являются узким местом для производительности. В общем и целом, измерение производительности часто лучше, чем гадание.

Следующая функция `gausspivotalnaive` — это простая реализация метода исключения Гаусса с перестановкой строк. Она принимает матрицу `A` и `b` в качестве аргумента с правой стороны, и возвращает решение `x`. Она объединяет разложение $\mathbf{PA} = \mathbf{LU}$ с прямым и обратным ходом в одной единственной функции. Множители матрицы $\mathbf{L}$ сохранены в нижней части матрицы $\mathbf{A}$, а множители матрицы $\mathbf{U}$ сохранены в верхней части матрицы $\mathbf{A}$.

```
1 function x = gausspivotalnaive ( A , b )
2   n = size(A,"r")
3   // Инициализируем перестановку
4   P=zeros(1,n-1)
5   // Выполняем преобразования Гаусса
6   for k=1:n-1
7     // Поиск ведущего элемента
8     mu = k
9     abspivot = abs(A(mu,k))
10    for i=k:n
11      if (abs(A(i,k))>abspivot) then
12        mu = i
13        abspivot = abs(A(i,k))
14      end
15    end
16    // Перестановка строк k и mu из столбцов от k до n
17    for j=k:n
18      tmp = A(k,j)
19      A(k,j) = A(mu,j)
20      A(mu,j) = tmp
21    end
22    P(k) = mu
23    // Выполнение преобразования для строк от k+1 до n
24    for i=k+1:n
25      A(i,k) = A(i,k)/ A(k,k)
26    end
27    for i = k+1:n
28      for j = k+1:n
29        A(i,j) = A(i,j) - A(i,k) * A(k,j)
30      end
31    end
32  end
33  // Выполнение прямого хода
34  for k=1:n-1
35    // Перестановка b(k) и b(P(k))
36    tmp = b(k)
37    mu = P(k)
38    b(k) = b(mu)
39    b(mu) = tmp
40    // Подстановка
41    for i=k+1:n
42      b(i) = b(i) - b(k) * A(i,k)
43    end
44  end
45  // Выполнение обратного хода
46  for k=n:-1:1
47    t = 0.
48    for j=k+1:n
49      t = t + A(k,j) * b(j)
50    end
51    b(k) = (b(k) - t) / A(k,k)
52  end
53  x = b
54 endfunction
```

Перед тем, как действительно измерять производительность, мы должны проверить корректность. поскольку нет смысла делать быстрее функцию, в которой имеются про-

граммные ошибки. Это кажется очевидным, но на практике частот не берётся во внимание. В следующем примере мы формируем матрицу `A` размером 10×10 со значениями элементов случайно распределёнными на интервале $[0, 1)$. Мы выбрали этот тестовый случай за его очень малое число обусловленности. Мы выбираем ожидаемое решение `e` с единичными элементами и вычисляем правую сторону `b` из уравнения `b=A*e`. Наконец, мы используем нашу функцию `gausspivotalnaive` для вычисления решения `x`.

```
n = 10;
A = grand(n,n,"def");
e = ones(n,1);
b = A * e;
x = gausspivotalnaive ( A , b );
```

Затем мы проверяем, что число обусловленности матрицы не слишком большое. В следующем примере мы вычисляем число обусловленности матрицы `A` по 2-норме, и проверяем, что оно приблизительно равно 10^1, что довольно мало́ (только порядок величины играет значение). Мы также проверим, что относительная ошибка для `x` имеет ту же амплитуду, что и `%eps`$\approx 10^{-16}$, что превосходно.

```
-->log10(cond(A))
 ans  =
    1.3898417
-->norm(e-x)/norm(e)
 ans  =
    6.455D-16
```

Требовалась бы более полная проверка, но мы сейчас допускаем, что мы можем верить нашей функции.

В следующем примере мы вызовем функцию `add_profiling` для того, чтобы профилировать функцию `gausspivotalnaive`.

```
-->add_profiling("gausspivotalnaive")
Внимание : переопределение функции : gausspivotalnaive        .
Выполните funcprot(0) для отключения этого сообщения
```

Целью функции `add_profiling` является переопределение функции, которую профилируют, так чтобы Scilab мог сохранить число выполнений каждой строки. Как указано в предупреждении, она в действительности изменяет профилируемую функцию, делая её медленнее. Это не играет значения, поскольку мы не принимаем слишком серьёзно времена выполнения на графике профиля: мы, вместо этого, будем фокусироваться только на число раз, которое конкретная строчка исходного кода была выполнена.

Для того, чтобы измерить производительность нашей функции, мы должны выполнить её, как в следующем примере:

```
x = gausspivotalnaive ( A , b );
```

Во время прогона, система профилирования сохранила данные профилирования, так что мы можем теперь анализировать. В следующем примере мы вызываем функцию `plotprofile` для того, чтобы построить профиль функции.

```
plotprofile(gausspivotalnaive)
```

Предыдущий пример формирует фигуру 29. График сделан из трёх гистограмм. Каждая функция имеет 54 строчки, которые представлены на оси X каждой гистограммы. Ось Y зависит от гистограммы. На первой гистограмме ось Y представляет число вызовов (colls) каждой соответствующей строки в профилируемой функции. Это измеряется просто обновлением счётчика каждый раз, когда выполняется строчка. На второй гистограмме ось

Y представляет сложность каждой соответствующей строчки в профилируемой функции. Это измеряет усилия интерпретатора для выполнения соответствующей строчки. Ось Y на третьей гистограмме представляет время ЦП (CPU time), в секундах, требуемое на выполнение соответствующей строчки. На практике, измерение времени ЦП ненадёжно, когда оно недостаточно велико.

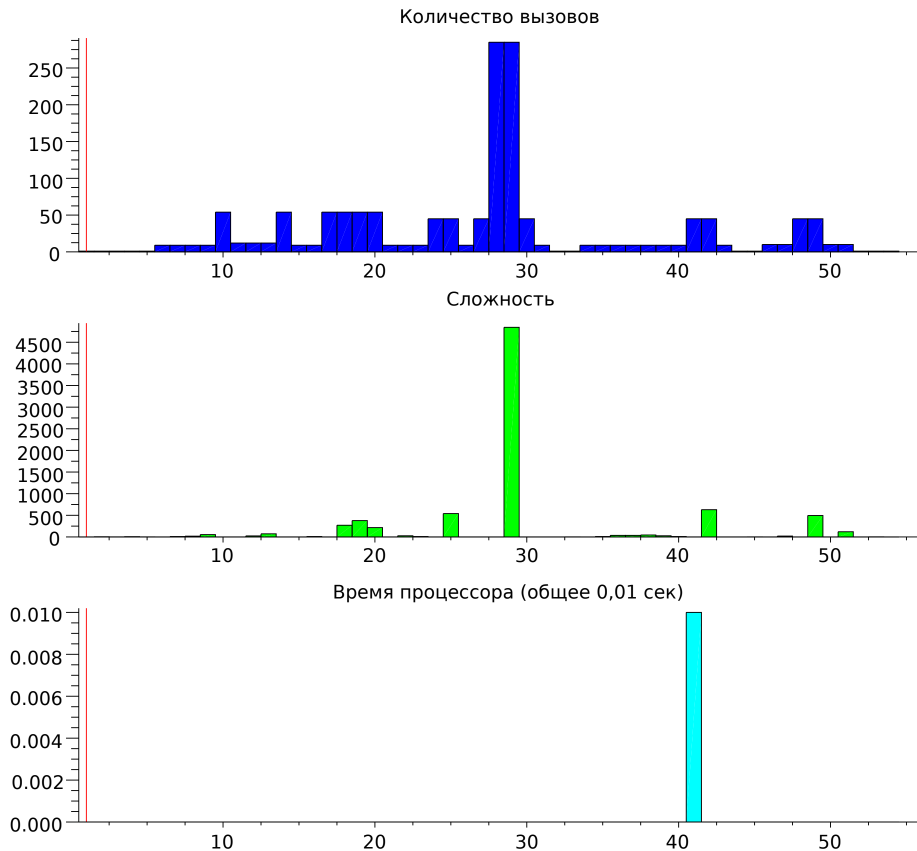

Рис. 29: Профиль функции `gausspivotalnaive`.

Функция `plotprofile` также открывает редактор и правит профилируемую функцию. Мышкой мы можем щёлкнуть левой кнопкой на цветные вертикальные полосы гистограммы. В редакторе это немедленно переместит курсор на соответствующую строчку. Мы сначала щёлкаем левой кнопкой мыши на самую большую полосу, которая соответствует строчке, которая выполнялась более 250 раз. Редактор затем перемещает курсор на строчки №23 и №24 в нашей функции. Это позволяет немедленно обратиться к следующим строчкам исходного кода.

```
for i = k+1:n
  for j = k+1:n
    A(i,j) = A(i,j) - A(i,k) * A(k,j)
```

В оставшейся части этого раздела мы собираемся постоянно использовать профилировщик. Анализируя различные части функции, которые наиболее затратные, мы можем измерить часть, которая имеет наибольшее влияние на производительность. Это позволяет применить принципы векторизации только на строчки исходного кода, которые имеют значение, и позволит нам сохранить большое количество времени. Как только мы это сде-

лаем, мы щёлкаем на меню «Exit» («Выход») в графическом окне, что закрывает график профиля.

Теперь проанализируем предыдущие вложенные циклы по переменным `i` и `j`. Мы можем сделать циклы по `j` быстрее, используя векторизованные инструкции. Используя оператор двоеточие `:`, мы заменяем переменную `j` на инструкцию `k+1:n`, как в следующем примере.

```
for i = k+1:n
  A(i,k+1:n) = A(i,k+1:n) - A(i,k) * A(k,k+1:n)
```

Мы можем распространить принцип далее, и векторизовать также цикл по переменной `i`, опять же используя оператор двоеточие. Это приводит к следующему исходному коду.

```
A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k) * A(k,k+1:n)
```

Эффект предыдущей инструкции заключается в обновлении подматрицы `A(k+1:n,k+1:n)` только одной инструкцией. Ключевым моментом является то, что предыдущая инструкция обновляет множество элементов матрицы только за один вызов Scilab'а. В данной ситуации, число обновлённых элементов равно $(n-k)^2$, что может быть много, если n велико. В общем, мы заменили примерно n^2 вызовов обработки интерпретатором матрицы 1×1 одним вызовом обработки матрицы размером $n \times n$. Заметим, что умножение `A(k+1:n,k)*A(k,k+1:n)` — это умножение вектора-столбца на вектор-строку, что даёт матрицу размером $(n-k) \times (n-k)$ с помощью только одной инструкции.

Аналогично, мы можем векторизовать другие инструкции в алгоритме.

Цикл в следующем исходном коде является очевидным кандидатом на векторизацию.

```
for i=k+1:n
  A(i,k) = A(i,k)/ A(k,k)
```

Предыдущий алгоритм математически эквивалентен, но гораздо медленнее, чем следующая векторизованная инструкция.

```
A(k+1:n,k) = A(k+1:n,k)/ A(k,k)
```

Поиск ведущих элементов в следующей части алгоритма кажется менее очевидным кандидатом на векторизацию.

```
mu = k
abspivot = abs(A(mu,k))
for i=k:n
  if (abs(A(i,k))>abspivot) then
    mu = i
    abspivot = abs(A(i,k))
  end
end
```

По существу, этот алгоритм является поиском элемента наибольшей величины в одном подстолбце матрицы `A`. Следовательно, мы можем использовать функцию `max`, которая возвращает как максимальный элемент её аргумента так и индекс, который получил это максимальное значение. Остаётся комбинировать несколько векторизованных функций для достижения нашей цели, что нетривиально.

Анализируя, как обновляется индекс `i` во время цикла, мы видим, что рассматриваемая подматрица — `A(k:n,k)`. Функция `abs` работает поэлементно, то есть, выполняет своё вычисление элемент за элементом. Следовательно, инструкция `abs(A(k:n,k))` вычисляет абсолютные величины, в которых мы заинтересованы. Мы можем, наконец, вызвать функцию `max` для выполнения поиска, как в следующем примере.

```
[abspivot,murel]=max(abs(A(k:n,k)))
mu = murel + k - 1
```

Заметьте, что функция `max` выполняет только поиск в части столбца матрицы `A`. Следовательно переменная `murel` хранит строку, достигнувшую максимума, относительно блока `A(k:n,k)`. Вот почему мы используем инструкцию `mu = murel + k - 1`, которая позволяет передать глобальный индекс строки в `mu`.

Следующий цикл позволяет переставлять строки `k` и `mu` матрицы `A`.

```
for j=k:n
  tmp = A(k,j)
  A(k,j) = A(mu,j)
  A(mu,j) = tmp
end
```

Сейчас для нас очевидно, что можно заменить предыдущий набор команд на следующий, векторизованный.

```
tmp = A(k,k:n)
A(k,k:n) = A(mu,k:n)
A(mu,k:n) = tmp
```

Это уже огромное улучшение предыдущего кода. Заметьте, что он создаст промежуточный вектор `tmp`, что не обязательно, если мы используем следующую инструкцию.

```
A([k mu],k:n) = A([mu k],k:n)
```

Действительно, матрица `[mu k]` является корректной матрицей индексов и может непосредственно использоваться для определения части матрицы `A`.

В общем следующая функция `gausspivotal` собирает все предыдущие улучшения и использует только векторизованные инструкции.

```
function x = gausspivotal ( A , b )
  n = size(A,"r")
  // Инициализация перестановок
  P=zeros(1,n-1)
  // Выполнение преобразований Гаусса
  for k=1:n-1
    // Поиск ведущего элемента
    [abspivot,murel]=max(abs(A(k:n,k)))
    // Сдвиг mu, поскольку  max возвращает относительно (k:n,k)
    mu = murel + k - 1
    // Перестановка строк k и mu из столбцов от k до n
    A([k mu],k:n) = A([mu k],k:n)
    P(k) = mu
    // Выполнение преобразования для строк от k+1 до n
    A(k+1:n,k) = A(k+1:n,k)/ A(k,k)
    A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k) * A(k,k+1:n)
  end
  // Выполнение прямого прохода
  for k=1:n-1
    // Перестановка b(k) и b(P(k))
    mu = P(k)
    b([k mu]) = b([mu k])
    // Замена
    b(k+1:n) = b(k+1:n) - b(k) * A(k+1:n,k)
  end
  // Выполнение обратного прохода
```

```
    for k=n:-1:1
      b(k) = (b(k) - A(k,k+1:n) * b(k+1:n)) / A(k,k)
    end
    x = b
  endfunction
```

Ещё раз, мы можем проверить что наша функция работает, как показано в следующем примере.

```
-->x = gausspivotal ( A , b );
-->norm(e-x)/norm(e)
 ans  =
    6.339D-16
```

Мы можем обновить профиль, запустив функцию `plotprofile` с улучшенной функцией `gausspivotal`. Она формирует фигуру 30.

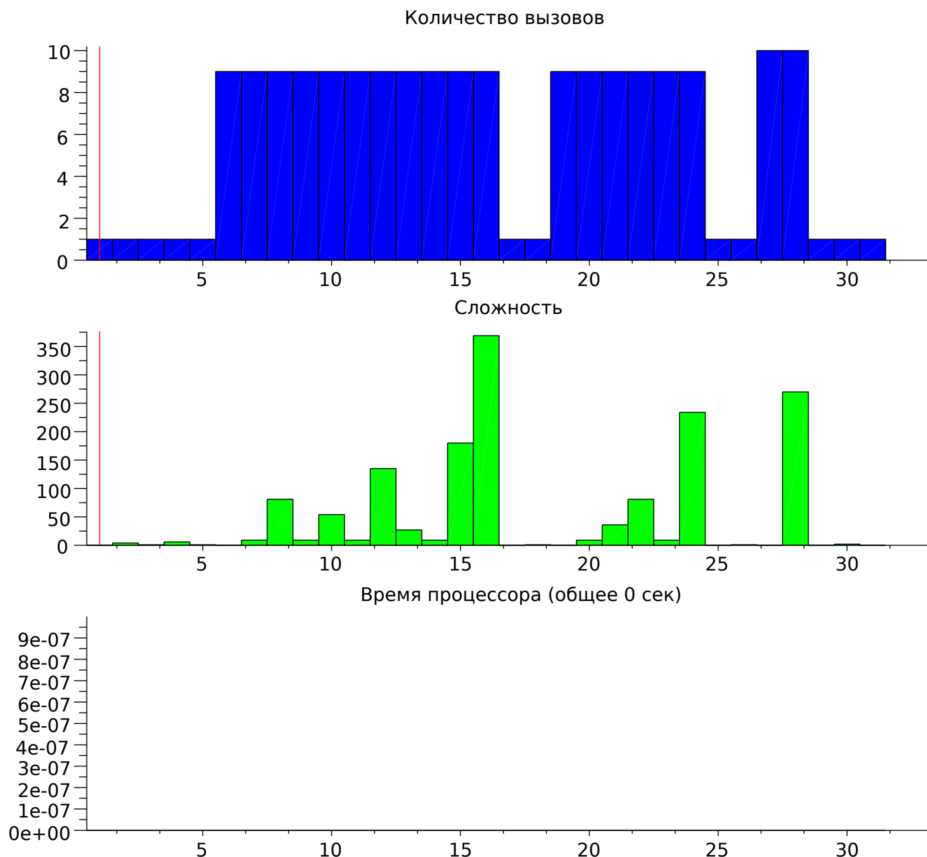

Рис. 30: Профиль функции `gausspivotal`.

Мы видим, что профиль числа вызовов гладкий, что означает, что нет строк, которые вызываются чаще других. Поскольку это великолепный результат, то мы можем остановить здесь наш процесс векторизации.

В упражнении 5.1, мы сравниваем действительные производительности этих двух функций. Мы видим, что для матрицы размером 100×100 (что скромно), среднее улучшение производительности может быть более 50. В общем, использование векторизации позволяет улучшить производительность на порядки.

5.1.4 Функция benchfun

В этом разделе мы представляем функцию, которая может использоваться для анализа производительности алгоритмов. Мы представляем изменчивость времени ЦП, требуемого для вычисления верхней треугольной матрицы Паскаля и представляем более надёжный способ измерения производительности алгоритма.

Попытаемся измерить производительность следующей функции `pascalup_col`, которая вычисляет верхнюю матрицу Паскаля.

```
function c = pascalup_col (n)
   c = eye(n,n)
   c(1,:) = ones(1,n)
   for i = 2:(n-1)
      c(2:i,i+1) = c(1:(i-1),i)+c(2:i,i)
   end
endfunction
```

Алгоритм, представленный здесь, основан на постолбцовом доступе к матрице, который особенно быстр, поскольку это именно тот способ, которым Scilab хранит элементы матрицы значений типа double. Эта тема представлена более глубоко в разделе 5.3.6.

В следующем примере мы измеряем время, требуемое для вычисления верхней треугольной матрицы Паскаля размером 1000×1000.

```
-->tic(); pascalup_col(1000); toc()
 ans  =
    0.172011
-->tic(); pascalup_col(1000); toc()
 ans  =
    0.15601
```

Как мы можем видеть, есть некоторая случайность в измерении затраченного времени. Эта случайность может быть объяснена несколькими фактами, но какие бы причины ни были, нам нужен более надёжный способ измерения времени, требуемого алгоритму. Общий метод заключается в запуске алгоритма несколько раз и усреднении измерений. Этот метод используется в следующем примере. где мы запускаем функцию `pascalup_col` 10 раз. Затем мы распечатываем минимальное, среднее и максимальное времена, требуемые во время эксперимента.

```
-->for k = 1 : 10
-->  tic();
-->  pascalup_col(1000);
-->  t(k) = toc();
-->end
-->disp([min(t) mean(t) max(t)])
    0.108006    0.1168072    0.16801
```

Предыдущий метод может быть автоматизирован, что ведёт к функции `benchfun`, которая представлена ниже. Эта функция запускает алгоритм для которого мы хотим получить надёжные измерения производительности. Входными аргументами являются имя функции `name`, которую тестируют, функция `fun`, которую запускают, список `iargs`, содержащий входные аргументы `fun`, число выходных аргументов `nlhs` и число итераций для выполнения `kmax`. Тело `benchfun` основано на выполнении функции `fun`, которая выполняется `kmax` раз. На каждой итерации измеряется производительность `fun` на основе функций `tic()` и `toc()`, которые измеряют затраченное функцией время (настенные часы). Функция `benchfun` возвращает матрицу `t` данных типа double размером `kmax1` $\times 1$, которая содержит

различные замеры времени, записанные во время исполнения fun. Она также возвращает строковую переменную msg, которая содержит дружественное к пользователю сообщение, описывающее производительность тестируемого алгоритма.

```
function [t,msg] = benchfun ( name , fun , iargs , nlhs , kmax )
  // Вычисляем строку инструкции, которую надо запустить
  ni = length ( iargs )
  instr = ""
  // Кладём аргументы левой стороны
  instr = instr + "["
  for i = 1 : nlhs
    if ( i > 1 ) then
      instr = instr + ","
    end
    instr = instr + "x" + string(i)
  end
  instr = instr + "]"
  // Кладём аргументы правой стороны
  instr = instr + "=fun("
  for i = 1 : ni
    if ( i > 1 ) then
      instr = instr + ","
    end
    instr = instr + "iargs("+string(i)+")"
  end
  instr = instr + ")"
  // Цикл по тестам
  for k = 1 : kmax
    tic()
    ierr = execstr ( instr , "errcatch" )
    t(k) = toc()
  end
  msgfmt = "%s: %d iterations, mean=%f, min=%f, max=%f\n"
  msg=msprintf(msgfmt,name,kmax,mean(t),min(t),max(t))
  mprintf("%s \n",msg)
endfunction
```

В следующем примере мы прогоняем функцию pascalup_col 10 раз для того, чтобы получить верхнюю треугольную матрицу Паскаля размером 2000×2000.

```
-->benchfun ( "pascalup_col" , pascalup_col , list(2000) , 1 , 10 );
pascalup_col: 10 iterations, mean=0.43162, min=0.41059, max=0.47067
```

Хотя функцию benchfun можно было бы сделать более устойчивой и более гибкой, этого достаточно для многих случаев. Мы будем часто использовать её в этом разделе.

5.2 Основы векторизации

В этом разделе мы представляем основы векторизации. Этот метод программирования является лучшим способом достичь хорошей производительности в Scilab. Первый раздел представляет основы интерпретатора, а также связь между примитивом и шлюзом. Мы сравниваем производительность циклов for с векторизованными инструкциями. В конце мы представляем пример анализа производительности, где мы преобразуем простой, медленный алгоритм в быстрый, векторизованный алгоритм.

5.2.1 Интерпретатор

В этом разделе мы кратко представляем методы компилирования так, что мы сможем иметь чёткие представления о значении интерпретатора на вопросы производительности. Мы также представляем связь между интерпретатором и шлюзами, которые являются функциями, которые связывают Scilab с нижележащими библиотеками.

Для того, чтобы понять что именно случается, когда выполняется такая инструкция как `y=f(x)`, мы должны взглянуть на структуру компилятора. Действительно, это позволит нам увидеть связь между инструкциями на уровне интерпретатора и выполнением сносно скомпилированного исходного кода на уровне библиотеки.

Компилятор [5] — это программа, которая читает программу на одном языке и переводит её в эквивалентную программу. Если целевая программа является исполнимой, написанной на машинном языке, она может быть вызвана пользователем для обработки входных значений и получения выходных значений. Обычные компиляторы этого типа являются компиляторами Fortran и C/C++ (среди других).

В отличие от предыдущей схемы, Scilab не является компилируемым языком: от *интерпретируемый* язык. Вместо производства программы на машинном языке, Scilab использует как программу в исходном коде, так и входные данные, и непосредственно производит выходные данные. Рисунок 31 представляет интерпретатор. Распространённые интерпретаторы этого типа — Scilab, Matlab и Python (среди прочих).

Рис. 31: Scilab — это интерпретатор.

Программа на машинном языке, сформированная компилятором вообще быстрее, чем программа интерпретатора в преобразовании входных данных в выходные. Интерпретатор, однако, может обычно предоставить лучшую диагностику, чем компилятор, поскольку он исполняет исходный код программы инструкцию за инструкцией. Более того, Scilab использует высокооптимизированные числовые библиотеки, которые в некоторых ситуациях могут быть быстрее, чем простая реализация на основе компилируемого языка. Эта тема будет рассмотрена в разделе 5.4.1, где мы представляем числовые библиотеки линейной алгебры, используемые в Scilab.

Компилятор обычно структурирован как последовательность операций. Первая операция включает в себя чтение символов исходного кода и выполняет лексический анализ. Этот шаг состоит в группировании символов в смысловые последовательности, именуемые лексемами. Лексемы затем передаются на анализатор синтаксиса (или парсер), который использует лексемы и создаёт древовидное представление исходного кода, который ассоциируется с грамматической структурой исходного кода. В семантическом анализе компилятор использует дерево синтаксиса и проверяет код на соответствие определению языка. Этот шаг включает в себя проверку типов операндов каждого оператора.

Затем могут быть включены несколько промежуточных операций, ведущих к финальному вызову библиотечных процедур. В Scilab этот финальный шаг реализован в *шлюзах*. Действительно, каждая функция Scilab'а или оператор неодинаково связан с вычислительными процедурами, внедрёнными в компилированный язык, обычно на C или Fortran. Шлюз читает входные аргументы, проверяет их типы и соответствие, выполняет необходимые вычисления и затем выталкивает обратно интерпретатору выходные значения. Такая структура представлена на рисунке 32.

Рис. 32: Scilab связывает интерпретатор с коллекцией библиотек через шлюзы.

5.2.2 Циклы в сравнении с векторизацией

В этом разделе мы представляем простой пример векторизации и анализируем почему циклы `for` являются хорошими кандидатами на векторизацию.

Рассмотрим следующую матрицу значений типа double, состоящую из $n = 5$ элементов.

```
x=[1 2 3 4 5];
```

Предположим, что мы хотим вычислить сумму 5-ти чисел, хранимых в переменной `x`. Мы сравним два файла-сценария, которые дают один и тот же верный результат, но имеют очень разную производительность.

В первом файле-сценарии мы вызываем функцию `sum` и устанавливаем переменную `y`.

```
y = sum(x);
```

Это требует только один вызов интерпретатора и включает в себя только один вызов к одному отдельному шлюзу. Внутри шлюза численная библиотека выполняет циклы по $n = 5$ в компилированном и оптимизированном исходном коде.

Во втором файле-сценарии мы используем циклы `for`.

```
y = 0;
n = size(x,"*");
for i = 1 : n
  y = y + x(i);
end
```

Предыдущий файл-сценарий привлекает интерпретатор по крайней мере $2 + n$ раз, где n — размер матрицы `x`. При $n = 5$ это 7 вызовов интерпретатора, но линейно возрастает с ростом размера `x`. Внутри шлюза численная библиотека выполняет одно единственное суммирование каждый раз, когда она вызывается. Когда число циклов большое, то дополнительные затраты на использование инструкции `for` вместо векторизованной инструкции являются слишком большими.

Поэтому *векторизованные* инструкции гораздо быстрее циклов. В общем, нам следует делать минимальное количество вызовов интерпретатора и позволить ему работать с достаточно большими наборами данных.

Теперь мы подчеркнём заключение предыдущего обсуждения.

- Если Scilab имеет встроенную функцию, то нам следует рассмотреть её использование до того, как создавать свою собственную. С точки зрения производительности в большинстве случаев они хорошо спроектированы, то есть, они как только возможно используют векторизованные инструкции. Более того, в большинстве случаев они принимают во внимание проблемы чисел с плавающей запятой, так что мы можем получить и быстрое вычисление и хорошую точность.

- Если мы должны создать свой собственный алгоритм, нам следует использовать векторизованные инструкции где только возможно. Действительно, ускорение векторизованных инструкции обычно 10–100, но может быть иногда 1000. В оставшейся части этого документа мы представим несколько примеров улучшения производительности.

Более того, поскольку Scilab может вызывать оптимизированные библиотеки, которые эффективно используют процессор, когда он должен выполнять вычисление с плавающей запятой. Файлы-сценарии Scilab могут быть такими же эффективными как компилированный исходный код, но они могут быть даже быстрее, чем нехитрый компилированный исходный код. Например, библиотеки линейной алгебры, используемые в Scilab'е, берут в расчёт размер кэша процессора и, более того, может выполнять многопотоковые операции. Это центральный момент в Scilab'е, который является преимущественно матричным языком. В разделе 5.4 представлены оптимизированные библиотеки линейной алгебры, используемые в Scilab'е. Не все программы могут получить выгоду от операций линейной алгебры, вот почему мы фокусируемся на методах векторизации, которые можно применять в большинстве ситуаций. Действительно, наш опыт говорит, что векторизация требует некоторой практики. Следующий раздел представляет практический пример анализа производительности на основе векторизации.

5.2.3 Пример анализа производительности

В этом разделе мы сравниваем производительность простого и векторизованного алгоритма. Мы проверяем, что ускорение в данном конкретном случае может быть равно 500.

Предположим, что мы имеем вектор-столбец, и что мы хотим вычислить среднее значение этого вектора. У нас две возможности выполнить эту задачу:

- использовать функцию `mean`, имеющуюся в Scilab,

- создать собственный алгоритм.

Функция `mean` может работать с входными матрицами различных размеров (векторы-строки, векторы-столбцы или общие матрицы) и с различными нормами (1-норма, 2-норма, ∞-норма и норма Фробениуса). Вот почему, в общем, нам не следует переопределять нашу собственную функцию, если в Scilab'е она уже реализована. Более того, производительность функции `mean` превосходна, как мы увидим. Но в целях иллюстрации вопросов производительности мы разработаем наш собственный алгоритм.

Следующая функция `mynaivemean` вычисляет среднее значение входного вектора `v`. Она выполняет цикл по данным с прямым накоплением суммы.

```
function y = mynaivemean ( v )
  y = 0
  n = length(v)
  for i = 1:n
    y = y + v(i)
```

```
  end
  y = y / n
endfunction
```

Перед оптимизацией нашего алгоритма, мы проверяем, что он выполняет свою задачу правильно: действительно, было бы потерей времени делать быстрее алгоритм, содержащий ошибку... Алгоритм, кажется, правильный, по крайней мере, с точки зрения следующего примера.

```
-->y = mynaivemean(1:9)
 y  =
    5.
```

Для того, чтобы проверить наш алгоритм, мы сформируем большие входные векторы `v`. Для этого, мы используем функцию `rand`, поскольку мы знаем, что производительность нашего алгоритма не не зависит от действительных значений входного вектора. Мы подчёркиваем эту деталь, поскольку практические алгоритмы могут показывать более или менее быструю производительность в зависимости от своих входных параметров. Сейчас не тот случай, потому что наш анализ упрощённый.

Мы формируем векторы размером $n = 10^j$ с $j = 1, 2, \ldots, 6$. Чтобы сформировать целые числа n, мы используем функцию `logspace`, которая возвращает значения на основе функции логарифма по основанию 10. Следующий файл-сценарий выполняет циклы от $n = 10^1$ до $n = 10^6$ и измеряет время, требуемое нашим простым алгоритмом.

```
n_data = logspace(1,6,6);
for i = 1:6
  n = n_data(i);
  v = rand(n,1);
  tic();
  ymean = mynaivemean ( v );
  t = toc();
  mprintf ("i=%d, n=%d, mynaivemean=%f, t=%f \n", i , n , ymean , t );
end
```

Далее представлена распечатка производимая файлом-сценарием.

```
i=1, n=10, mynaivemean=0.503694, t=0.000000
i=2, n=100, mynaivemean=0.476163, t=0.000000
i=3, n=1000, mynaivemean=0.505503, t=0.010014
i=4, n=10000, mynaivemean=0.500807, t=0.090130
i=5, n=100000, mynaivemean=0.499590, t=0.570821
i=6, n=1000000, mynaivemean=0.500216, t=5.257560
```

Мы видим, что время растёт очень быстро и является довольно большим для $n = 10^6$.

На самом деле, можно разработать гораздо более быстрый алгоритм. В предыдущем файле-сценарии мы заменим функцию `mynaivemean` на встроенную функцию `mean`. Это даёт следующую распечатку.

```
i=1, n=10, mean=0.493584, t=0.000000
i=2, n=100, mean=0.517783, t=0.000000
i=3, n=1000, mean=0.484507, t=0.000000
i=4, n=10000, mean=0.501106, t=0.000000
i=5, n=100000, mean=0.502038, t=0.000000
i=6, n=1000000, mean=0.499698, t=0.010014
```

Как мы видим, функция `mean` гораздо быстрее нашей функции `mynaivemean`. В данном случае мы видим, что для `n=1000000` отношение по скорости приблизительно $5{,}25/0{,}01 = 525$.

Для того, чтобы улучшить наш простой алгоритм, мы можем использовать функцию `sum`, которая возвращает сумму элементов её входного аргумента. Следующая функция `myfastmean` выполняет вычисление на основе функции `sum`.

```
function y = myfastmean ( v )
  y = sum(v);
  n = length(v);
  y = y / n;
endfunction
```

Следующая распечатка представляет производительность нашей улучшенной функции.

```
i=1, n=10, mean=0.419344, t=0.000000
i=2, n=100, mean=0.508345, t=0.000000
i=3, n=1000, mean=0.501551, t=0.000000
i=4, n=10000, mean=0.501601, t=0.000000
i=5, n=100000, mean=0.499188, t=0.000000
i=6, n=1000000, mean=0.499992, t=0.010014
```

Производительность нашей быстрой функции теперь приблизительно та же, что и производительность встроенной функции `mean`. В действительности, функция `mean` является макросом, который выполняет преимущественно тот же самый алгоритм, что и наша функция, то есть он использует функцию `sum`.

Почему же функция `sum` гораздо быстрее алгоритма на основе цикла `for`? Причина в том, что функция `sum` является *примитивом*, основанным на компилированном исходном коде. Это легко проверить, получив тип этой функции, как в следующем примере.

```
-->typeof(sum)
 ans  =
 fptr
```

Следовательно, цикл, выполняемый Scilab'ом внутри функции `sum` компилированным исходным кодом. Это действительно быстрее, чем цикл `for`, выполненный интерпретатором, и объясняет разницу производительностей между нашей и двумя предыдущими реализациями.

5.3 Уловки оптимизации

В этом разделе мы представляем общие уловки для получения более быстрых алгоритмов и обратим внимание на использование векторизованных инструкций. Мы представляем опасное использование матриц, чей размер растёт динамически во время работы алгоритма. Мы представляем функции общего назначения, такие, как векторизованные функции сортировки и поиска, которые могут быть объединены для создания эффективных алгоритмов. Мы также обсуждаем ориентацию матриц и даём пример улучшения производительности на основе этой уловки.

5.3.1 Опасность динамических матриц

В этом разделе мы анализируем вопросы производительности, которые вызваны использованием матриц, чей размер растёт динамически. Мы показываем числовой эксперимент, где предопределённая матрица известного размера гораздо быстрее матрицы, которой позволено расти динамически.

Давайте рассмотрим ситуацию, когда мы должны вычислить матрицу, но не знаем заранее конечный размер этой матрицы. Это случается когда, например, матрица прочитана из файла данных или интерактивно вводится пользователем программы. В этом случае мы можем использовать то, что размер матрицы может расти динамически. Но это требует затрат, которые могут быть чувствительными, когда число динамических обновлений велико. Эти затраты связаны с внутренним поведением обновления размера матрицы. Действительно, увеличение размера матрицы подразумевает, что интерпретатору приходится обновлять свою внутреннюю структуру данных. Следовательно, всё содержимое матрицы должно быть скопировано, что может потребовать значительное количество времени. Более того, пока циклы обрабатываются, размер матрицы на самом деле растёт, что в свою очередь подразумевает и более медленное копирование матрицы.

Для того, чтобы проанализировать вопросы производительности, мы сравним три функции с разными реализациями, но с одинаковым выходом. Целью является получение матрицы `A=[1,2,3,...,n]`, где `n` — большое целое число. Чисто в целях эффективности мы могли бы использовать оператор двоеточия «`:`» и инструкцию `A=(1:n)`. Для того, чтобы проиллюстрировать конкретный вопрос, который является темой этого раздела, мы не будем использовать оператор двоеточия, а, наоборот, мы используем следующие реализации.

Функция `growingmat1` принимает `n` в качестве входного аргумента и выполняет динамическое вычисление матрицы `A`. Сначала алгоритм инициализирует матрицу `A` в виде пустой матрицы. Затем он использует синтаксис `$+1` для динамического увеличения размеров матрицы и сохранения нового элемента `i`.

```
function growingmat1 ( n )
  A = []
  for i = 1 : n
    A($+1) = i
  end
endfunction
```

Мы не возвращаем матрицу `A` в качестве выходного аргумента, поскольку это необязательно для нашей демонстрации.

Следующая функция `growingmat2` также выполняет динамическое вычисление матрицы `A`, но использует другой синтаксис. В этот раз матрица динамически обновляется с помощью синтаксиса `[A,i]`, который создаёт новую матрицу-строку с элементом `i`, добавленным к концу.

```
function growingmat2 ( n )
  A = []
  for i = 1 : n
    A = [A,i]
  end
endfunction
```

Это создаёт матрицу-строку. Вариант этого алгоритма создавал бы матрицу-столбец синтаксисом `[A;i]`. Этот вариант не изменит фундаментально поведение алгоритма.

Следующая функция `growingmat3` полностью отличается от предыдущих. Вместо инициализации матрицы пустой матрицей, она предварительно создаёт матрицу-столбец нулей. Затем элементы заполняются один за другим.

```
function growingmat3 ( n )
  A = zeros(n,1)
  for i = 1 : n
    A(i) = i
```

```
  end
endfunction
```

В следующем файле-сценарии мы сравниваем производительности трёх предыдущих функций с n=20000 и повторяем замер времени 10 раз для получения более надёжной оценки производительности. Мы используем функцию benchfun, которая была представлена в разделе 5.1.4.

```
stacksize("max")
benchfun ( "growingmat1" , growingmat1 , list(20000) , 0 , 10 );
benchfun ( "growingmat2" , growingmat2 , list(20000) , 0 , 10 );
benchfun ( "growingmat3" , growingmat3 , list(20000) , 0 , 10 );
```

В следующем примере мы получаем замер времени трёх алгоритмов.

```
-->exec bench_dynamicmatrix.sce;
growingmat1: 10 iterations, mean=1.568255
growingmat2: 10 iterations, mean=0.901296
growingmat3: 10 iterations, mean=0.023033
```

Мы видим, что предварительное создание матрицы A с её известным размером n гораздо быстрее. Отношение от самого медленного до самого быстрого более 50.

5.3.2 Линейная индексация матрицы

В этом разделе мы покажем как можно получить доступ к обычной матрице с помощью одного-единственного *линейного* индекса k вместо пары подындексов (i,j). Эта возможность работает как для матриц, так и для гиперматриц, хотя мы в этом разделе рассматриваем только обычных матрицы.

11 *(1)*	14 *(4)*	17 *(7)*	20 *(10)*
12 *(2)*	15 *(5)*	18 *(8)*	21 *(11)*
13 *(3)*	16 *(6)*	19 *(9)*	22 *(12)*

Рис. 33: Линейное индексирование матрицы размером 3×4. Например, значение «21» расположено в ячейке с подындексами $(i, j) = (2, 4)$ и имеет линейный индекс $k = 11$. Линейные индексы ведут себя так, как если бы элементы были сохранены столбец за столбцом.

Если матрица A имеет m строк, то линейный индекс k, соответствующий подындексам (i, j), равен $k = i + (j - 1)m$. В этом случае инструкции A(i,j) и A(k) эквивалентны. Мы можем понимать эту возможность как следствие того, что матрицы хранятся столбец за столбцом (смотри подробности по этой теме в разделе 5.3.6). В следующем примере мы создаём матрицу значений типа double размером 3×4. Затем мы обнуляем элемент с линейным индексом k=11. Этот элемент соответствует элементу с подындексами $(i, j) = (2, 4)$.

```
-->A = matrix(10+(1:12),3,4)
 A  =
    11.    14.    17.    20.
    12.    15.    18.    21.
    13.    16.    19.    22.
-->A(11)=0
 A  =
```

```
    11.     14.     17.     20.
    12.     15.     18.     0.
    13.     16.     19.     22.
```

Предыдущий пример представлен на рисунке 33.

Функции sub2ind и ind2sub делают преобразование из подындексов в линейные индексы и обратно. Эти функции представлены на рисунке 34. Подчеркнём, что эти функции работают как с матрицами, так и гиперматрицами, но мы остановимся в оставшейся части этого раздела на обычных матрицах.

ind2sub	преобразует линейные индексы k в подындексы i1,i2,...
sub2ind	преобразует подындексы i1,i2,... линейные индексы k

Рис. 34: Функции Scilab'а для преобразования из подындексов в линейные индексы.

Например, мы можем преобразовать линейный индекс в пару подындексов с помощью инструкции [i,j]=ind2sub(dims,k), где dims является матрицей 1×2, содержащей размеры матрицы. Далее представлен пример функции ind2sub. Рассмотрим матрицу 3×4 (как на рисунке 33) и проверим, что линейный индекс k=11 соответствует подындексам i=2 и j=4.

```
-->[i,j]=ind2sub([3,4],11)
 j  =
    4.
 i  =
    2.
```

Теперь дадим простой пример как можно использовать функцию sub2ind на практике, когда используется векторизованное вычисление. Допустим, что у нас есть матрица 3×4 и мы хотим обнулить первую диагональ над главной диагональю. В нашем конкретном случае эта диагональ соответствует значениям 14, 18 и 22, которые расположены в строках 1, 2 и 3 и столбцах 2, 3 и 4. Конечно, мы могли бы использовать циклы for, но тогда это не будет векторизованным вычислением. Мы так же не можем напрямую использовать подындексы этих строк и столбцов, поскольку это обнулило бы всю соответствующую подматрицу 3×3, как показано в следующем примере.

```
-->A = matrix(10+(1:12),3,4)
 A  =
    11.     14.     17.     20.
    12.     15.     18.     21.
    13.     16.     19.     22.
-->A([1 2 3],[2 3 4]) = 0
 A  =
    11.     0.     0.     0.
    12.     0.     0.     0.
    13.     0.     0.     0.
```

Следовательно, мы используем функцию sub2ind для преобразования этих подындексов в вектор линейных индексов k и затем обнулим эти элементы.

```
-->A = matrix(10+(1:12),3,4)
 A  =
    11.     14.     17.     20.
    12.     15.     18.     21.
    13.     16.     19.     22.
-->k = sub2ind([3,4],[1 2 3],[2 3 4])
```

```
 k  =
    4.    8.    12.
-->A(k) = 0
 A  =
    11.    0.     17.    20.
    12.    15.    0.     21.
    13.    16.    19.    0.
```

Предыдущий пример может использоваться в качестве основы для алгоритмов, которые обращаются к диагоналям матриц, таким, как, например, в функции `spdiags`.

В следующем разделе 5.3.5 мы пересмотрим пример линейного индексирования. Там мы будем использовать функцию `find` для обращения к отдельным элементам матрицы.

5.3.3 Обращение через матрицу логических значений

В этом разделе мы покажем как можно обратиться к обычной матрице с помощью индексов, определённых матрицей логических значений.

Операторы, которые создают матрицы логических значений, такие, как `==` или `>`, могут использоваться для обращения к матрицам с помощью векторизованных инструкций. Предположим, что `A` является матрицей $m \times n$ значений типа double, и `B` является матрицей $m \times n$ логических значений. Следовательно, инструкция `A(B)` обращается ко всем элементам `A`, для которых `B` равно *истине*. Это показано в следующем примере, где мы вычисляем все элементы `A`, которые равны 1, и обнуляем их.

```
-->A = [1 2 1 2; 2 1 2 1; 1 3 1 3]
 A  =
    1.    2.    1.    2.
    2.    1.    2.    1.
    1.    3.    1.    3.
-->B = (A==1)
 B  =
  T F T F
  F T F T
  T F T F
-->A(B) = 0
 A  =
    0.    2.    0.    2.
    2.    0.    2.    0.
    0.    3.    0.    3.
```

Предыдущая последовательность операций может быть упрощена до одной-единственной инструкции `A(A==1)=0`. Это показано в следующем примере.

```
-->A = [1 2 1 2; 2 1 2 1; 1 3 1 3]
 A  =
    1.    2.    1.    2.
    2.    1.    2.    1.
    1.    3.    1.    3.
-->A(A==1)=0
 A  =
    0.    2.    0.    2.
    2.    0.    2.    0.
    0.    3.    0.    3.
```

В ситуации, когда поиск отдельных элементов `A` является слишком сложным, чтобы управляться таким методом, мы можем использовать функцию `find`, как показано в разделе 5.3.5.

5.3.4 Повтор строк или столбцов вектора

В этом разделе мы покажем как создать копии строк или столбцов вектора с помощью векторизованных инструкций.

Интересным свойством матриц является то, что выделение матрицы может быть использовано для повтора элементов одной матрицы. Действительно, если `A` является матрицей, то мы можем многократно выделять первую строку, что создаёт копии этой строки.

```
-->A = 1:3
 A  =
    1.    2.    3.
-->B = A([1 1 1],:)
 B  =
    1.    2.    3.
    1.    2.    3.
    1.    2.    3.
```

Интересным здесь является то, что создание `B` не требует циклов: оно векторизовано.

Эта уловка используется в функции `repmat`, которая может быть интегрирована в Scilab 5.4.

Эта тема изучена в упражнении 5.2.

5.3.5 Комбинирование векторизованных функций

Для того, чтобы получить хорошую производительность в Scilab'е, мы можем использовать векторизованные инструкции. В этом разделе мы представляем как комбинировать векторизованные функции.

Векторизованные файлы-сценарии лучше используют оптимизированные библиотеки Scilab'а. Главное правило — избегать циклы `for`.

Вторым правилом является использовать матричные операции как можно чаще, поскольку они используют высокооптимизированные библиотеки линейной алгебры, которые предоставлены Scilab'ом. Однако, есть ситуации, где выполнение векторизованных вычислений не очевидно. В этих случаях мы можем *комбинировать* векторизованные функции: целью этого раздела в том, чтобы представить относительно продвинутые векторизованные функции и подсказать как их комбинировать.

Рисунок 35 представляет наиболее общие векторизованные функции, которые мы можем использовать для получения хорошей производительности.

`+,-,*,/, \`	алгебраические операторы
`.*.`	произведение Кронекера
`min,max`	минимум, максимум
`sum`	сумма элементов матрицы
`cumsum`	накапливаемая сумма элементов матрицы
`prod`	произведение элементов матрицы
`cumprod`	накапливаемое произведение элементов матрицы
`find`	поиск *истинных* индексов в булевой матрице
`gsort`	сортирует элементы матрицы
`matrix`	генерирует матрицу из элементов матрицы

Рис. 35: Функции Scilab наиболее часто используемые при векторизации.

Алгебраические операторы +, -, *, /, \ представлены здесь из-за того, что они предоставляют превосходную производительность в Scilab'е. Для того, чтобы использовать эти операторы, мы должны сформировать матрицы к которым мы можем применить линейную алгебру. Мы можем использовать zeros, ones и другие матрично-ориентированные функции, но полезный оператор для формирования матриц — это произведение Кронекера. В следующем примере мы перемножаем две матрицы размерами 3×2 произведением Кронекера, которое создаёт матрицу размером 4×9.

```
-->[1 2 3;4 5 6] .*. [7 8 9;1 2 3]
 ans  =
    7.     8.     9.     14.    16.    18.    21.    24.    27.
    1.     2.     3.     2.     4.     6.     3.     6.     9.
    28.    32.    36.    35.    40.    45.    42.    48.    54.
    4.     8.     12.    5.     10.    15.    6.     12.    18.
```

Произведение Кронекера может быть, например, использовано в комбинаторных алгоритмах, где мы хотим получить комбинации величин из данного набора. Этот вопрос исследован в упражнении 5.2.

Рассмотрим теперь функции find, gsort, min и max, которые имеют специальные входные и выходные аргументы, которые делают их очень полезными, когда мы делаем поиск в быстрых программах.

Использование функции find является общим методом оптимизации, который может заменить алгоритмы поиска. Если B является матрицей $m \times n$ логических значений, то инструкция k=find(B) возвращает линейные индексы k, для которых B равна *истине*.

Если поиск завершился неудачно, то есть, если ни один из элементов не соответствует условию, то функция find возвращает пустую матрицу. Можно подумать, что необходимо явно управлять этим конкретным случаем. Однако инструкция A([])=0 ничего не делает, поскольку пустая матрица [] соответствует пустому набору индексов. Следовательно на выходе функции find в большинстве случаев нет нужды обрабатывать случай неудачи отдельной инструкцией if.

В следующем примере мы обнуляем все элементы матрицы A,которые равны 1.

```
-->A = [1 2 1 2; 2 1 2 1; 1 3 1 3]
 A  =
    1.    2.    1.    2.
    2.    1.    2.    1.
    1.    3.    1.    3.
-->B = (A==1)
 B  =
  T F T F
  F T F T
  T F T F
-->k = find(B)
 k  =
    1.    3.    5.    7.    9.    11.
-->A(k) = 0
 A  =
    0.    2.    0.    2.
    2.    0.    2.    0.
    0.    3.    0.    3.
```

Этот пример можно укоротить объединив функции в одну-единственную инструкцию, как показано ниже.

```
-->A = [1 2 1 2; 2 1 2 1; 1 3 1 3]
```

```
 A  =
    1.    2.    1.    2.
    2.    1.    2.    1.
    1.    3.    1.    3.
-->A(find(A==1)) = 0
 A  =
    0.    2.    0.    2.
    2.    0.    2.    0.
    0.    3.    0.    3.
```

Поэкспериментируем что будет, если значение, которое мы искали, не найдено.

```
-->k = find(A==999)
 k  =
     []
-->A(k)=0
 A  =
    0.    2.    0.    2.
    2.    0.    2.    0.
    0.    3.    0.    3.
```

Конечно, как показано в разделе 5.3.3, мы могли бы использовать более короткую инструкцию `A(A==1)=0`.

Однако, есть случаи, когда линейные индексы `k`должны быть обработаны перед тем, как использовать их в установке значений матрицы. Более того, есть у функции опции `find`, которые можно использовать для создания более сложных векторизованных операций, как мы увидим далее.

Если используются не все элементы, удовлетворяющие условию, а лишь их часть, то мы можем использовать второй входной аргумент функции `find`. Например, инструкция `k=find(A(:,2)>0,1)` возвращает первый индекс второго столбца `A`, который положительный.

Функции `min` и `max` могут быть использованы с двумя выходными аргументами. Действительно, инструкция `[m,k]=min(A)` возвращает минимальное значение `m` и индекс `k` элемента в `A`, который имеет минимальное значение. Другими словами, индекс `k` такой, что `A(k)==m`. Например, инструкция `[m,k]=min(abs(A))` сочетает функцию `min` и поэлементную функцию `abs`. Это позволяет получить индекс `k` того элемента матрицы `A`, который имеет минимальное абсолютное значение.

Функция `gsort` имеет также второй аргумент, который может быть полезен для получения хорошей производительности. Действительно, инструкция `[B,k]=gsort(A)` возвращает сортированную матрицу `B` и линейные индексы `k` элементов в исходной матрице. Другими словами, матрица `B` такова, что `B(:)==A(k)`. На практике мы можем применять ту же перестановку во второй матрице `C` с помощью инструкции `C = C(k)`.

5.3.6 Постолбцовое обращение быстрее

Мы можем получить бóльшую производительность с помощью постолбцового обращения к матрице, нежели с помощью построчному. В этом разделе мы представляем пример улучшения производительности с помощью изменения ориентации алгоритма и представим объяснение этому поведению.

Вопрос, который мы собираемся рассмотреть, обсуждался в сообщении об ошибке №7670 [13]. Проблема заключается в вычислении матрицы Паскаля, которая может быть получена из формулы бинома. Ряд отдельных наборов с j элементами, которые могут быть

выбраны из набора A с n элементами, является биномиальным коэффициентом и обозначается $\binom{n}{j}$. Он определяется как

$$\binom{n}{j} = \frac{n.(n-1)\ldots(n-j+1)}{1.2\ldots j}. \tag{1}$$

Для целых чисел $n > 0$ и $0 < j < n$, биномиальные коэффициенты удовлетворяют

$$\binom{n}{j} = \binom{n-1}{j} + \binom{n-1}{j-1}. \tag{2}$$

Следующие два файла-сценария позволяют вычислить матрицу Паскаля. В первой реализации мы вычисляем нижнюю треугольную матрицу Паскаля и выполняем построчный алгоритм.

```
// Pascal lower triangular: Row by row version
function c = pascallow_row ( n )
   c = eye(n,n)
   c(:,1) = ones(n,1)
   for i = 2:(n-1)
      c(i+1,2:i) = c(i,1:(i-1))+c(i,2:i)
   end
endfunction
```

Предыдущая реализация была предложена мне Каликстом Денизетом (Calixte Denizet) в обсуждении, связанном с сообщением о программной ошибке, но похожая идея была представлена Самуэлем Гужоном (Samuel Gougeon) в том же потоке. Во второй реализации мы вычисляем верхнюю треугольную матрицу Паскаля и выполняем постолбцовый алгоритм.

```
// Pascal upper triangular: Column by column version
function c = pascalup_col (n)
   c = eye(n,n)
   c(1,:) = ones(1,n)
   for i = 2:(n-1)
      c(2:i,i+1) = c(1:(i-1),i)+c(2:i,i)
   end
endfunction
```

Следующий пример показывает, что обе реализации математически корректны.

```
-->pascalup_col (5)
 ans  =
    1.    1.    1.    1.    1.
    0.    1.    2.    3.    4.
    0.    0.    1.    3.    6.
    0.    0.    0.    1.    4.
    0.    0.    0.    0.    1.
-->pascallow_row ( 5 )
 ans  =
    1.    0.    0.    0.    0.
    1.    1.    0.    0.    0.
    1.    2.    1.    0.    0.
    1.    3.    3.    1.    0.
    1.    4.    6.    4.    1.
```

Но между двумя этими алгоритмами есть существенная разница в производительности. Рисунок 36 представляет производительность этих двух алгоритмов для различных размеров

матриц. Рисунок представляет отношение между постолбцовой и построчной реализациями. Как видим, постоблцовая реализация в целом на 10 % быстрее построчной реализации. За

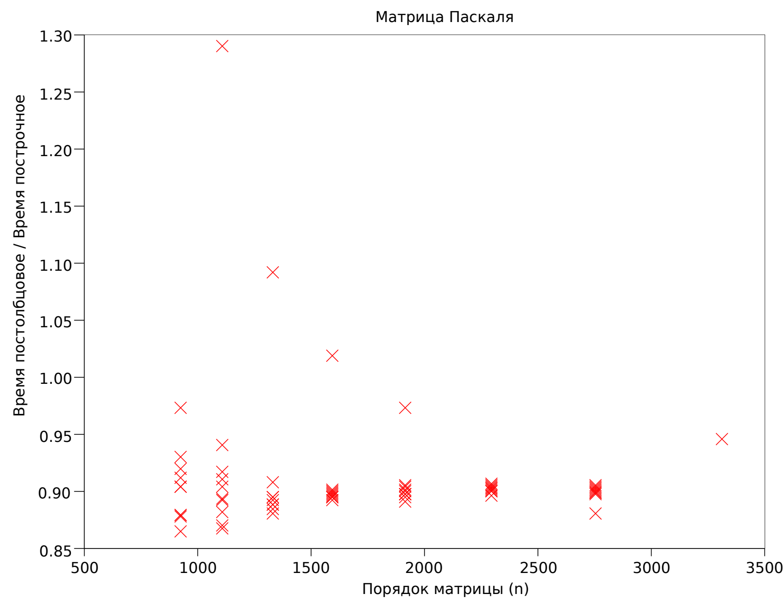

Рис. 36: Сравнение постолбцового и построчного алгоритмов для матрицы Паскаля.

исключением редких экспериментов, этот факт верен для матриц всех размеров.

Причиной этого является внутренняя структура данных для матрицы чисел типа double, которая ориентирована на Фортран. Поэтому элементы матрицы сохраняются столбец за столбцом как показано на рисунке 37. Следовательно, если мы упорядочим операторы так, что они выполняли постолбцовый алгоритм, то мы будем иметь быстрое обращение к элементам, из-за расположения данных. Это позволяет использовать наилучшим образом процедуру DCOPY (из библиотеки BLAS), которая используется для создания промежуточных векторов, используемых в алгоритме. Действительно, выражение `c(2:i,i)` создаёт промежуточный вектор-столбец, а выражение `c(i,2:i)` создаёт промежуточный вектор-строку. Это требует создания временных векторов, которые должны заполняться данными, скопированными из матрицы `c`. Когда вектор-столбец `c(2:i,i)` создан, то все элементы находятся друг за другом. Вектор-строка `c(i,2:i)`, напротив, требует пропускать все элементы, которые находятся между двумя последовательными элементами в исходной матрице `c`. Поэтому выделение вектора-строки из матрицы требует большего времени, чем выделение вектора-столбца последовательных элементов: это требует меньше перемещений памяти и лучше использует различные уровни кэша. Следовательно, если данный алгоритм может быть изменён так, чтобы он мог обращаться или обновлять матрицу постолбцово, то следует предпочесть эту ориентацию.

5.4 Оптимизированные библиотеки линейной алгебры

В этом разделе мы представляем библиотеки линейной алгебры, которые используются в Scilab'е. В следующем разделе мы представляем BLAS и LAPACK, которые являются

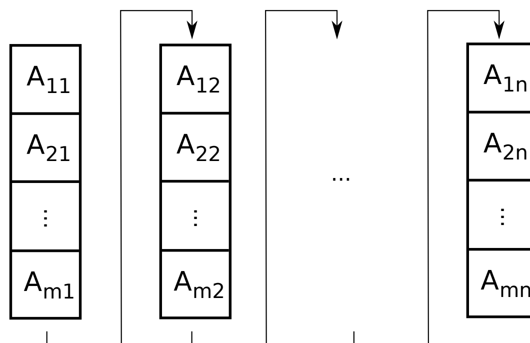

Рис. 37: Хранение матрицы чисел типа double размером $m \times n$ в Scilab. Два последовательных элемента в одном и том же столбце матрицы хранятся последовательно в памяти. Два последовательных элемента в одной строке матрицы разделены в памяти m адресами.

строительными блоками линейной алгебры в Scilab'е. Мы также представляем численные библиотеки ATLAS и Intel MKL. Мы представляем метод установки этих оптимизированных библиотек линейной алгебры в операционных системах Windows и Linux. Мы представляем влияние изменения библиотеки на производительность произведения общих неразреженных матриц.

5.4.1 BLAS, LAPACK, ATLAS и MKL

В этом разделе мы вкратце представляем библиотеки BLAS, LAPACK, ATLAS и MKL для того, чтобы иметь общее представление об этих библиотеках. Эти библиотеки используются Scilab'ом для обеспечения максимальной производительности в каждой системе.

Библиотека BLAS (Basic Linear Algebra Subprograms — основные подпрограммы линейной алгебры) [2] является коллекцией процедур Фортрана, предоставляющих низкоуровневые операции, такие, как добавление векторов, поэлементное и матричное умножения. Библиотека BLAS делится на три уровня.

- BLAS уровня 1 выполняет скалярные, векторные и векторно-векторные операции.
- BLAS уровня 2 выполняет матрично-векторные операции.
- BLAS уровня 3 выполняет матрично-матричные операции.

Библиотека LAPACK (Linear Algebra Package — пакет линейной алгебры) [3] является коллекцией процедур Фортрана для решения задач линейной алгебры высокого уровня. Это включает в себя решение систем линейных уравнений, решение линейных систем по методу наименьшего квадрата, задачи нахождения собственных значений и задачи нахождения сингулярного значения.

Библиотека ATLAS (Automatically Tuned Linear Algebra Software — автоматически настраиваемое программное обеспечение линейной алгебры) [1] предоставляет оптимизированные процедуры линейной алгебры BLAS и небольшой набор процедур, оптимизированных библиотекой LAPACK. Библиотека концентрируется на применении эмпирических методов для обеспечения характеристики переносимости. Библиотека ATLAS предоставляет во многих случаях чувствительное улучшение производительности по сравнению с системами «основных» BLAS/LAPACK.

Библиотека математического ядра фирмы Intel (Intel Math Kernel Library) [24] является библиотекой высокооптимизированных, широко разветвлённых математических процедур для научно-технических и финансовых приложений, которые требуют максимальной производительности. В данный момент под Windows Scilab использует только процедуры линейной алгебры от MKL взамен BLAS.

В Windows следует выбирать Intel MKL для установки (это сделано по умолчанию для Scilab версии 5). В Linux-системах следует использовать оптимизированную версию библиотеки ATLAS, если производительность играет роль. Эти вопросы рассмотрены в разделах 5.4.3 для Windows и 5.4.4 для Linux.

Библиотеки ATLAS и Intel MKL используют разные методы низкоуровневой оптимизации для получения производительности при вычислениях с плавающей запятой. Этот вопрос ещё раз рассмотрен в следующем разделе.

5.4.2 Методы низкоуровневой оптимизации

В этом разделе кратко представлены методы низкоуровневой оптимизации, используемые в численных библиотеках линейной алгебры, таких, как ATLAS и Intel MKL. Представлены различные уровни памяти в компьютере. Рассмотрены алгоритмы, используемые в ATLAS для улучшения использования различных уровней кэша. Кратко представлены наборы инструкций процессоров и используемые оптимизированными библиотеками линейной алгебры.

Главной причиной улучшения производительности векторизованных функций является то, что Scilab способен выполнять вычисления над *коллекцией* данных вместо обработки каждого вычисления отдельно. Действительно, производительность вычисления значительно зависит от того, насколько ЦП может обращаться к данным, которые доступны в различных типах памяти. От наиболее быстрой до наиболее медленной памяти, используемой в современных компьютерах, — это различные уровни кэша (встроенного в сам процессор) и оперативная память (RAM) (здесь мы не рассматриваем жёсткий диск в качестве запоминающего устройства, хотя он может быть включён в управление виртуальной памятью). Рисунок 38 представляет различные уровни памяти, обычно используемой в компьютере.

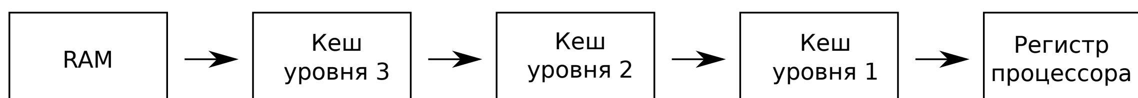

Рис. 38: Уровни памяти в компьютерной системе.

Теперь рассмотрим следующий пример, где мы выполняем перемножение матрицы на матрицу.

```
A = rand(10,10);
B = rand(10,10);
C = A * B;
```

В соответствующем шлюзе Scilab использует процедуру BLAS *DGEMM* для того, чтобы выполнить требуемое вычисление. Если пользователь выбрал библиотеки ATLAS или MKL во время инсталляции, то вычисления производятся оптимизированным способом.

Поскольку исходный код Intel MKL не является открытым, то мы не знаем какие точно методы использованы в этой библиотеке. В отличие от него, ATLAS является библиотекой с открытым исходным кодом, и мы можем подробно анализировать его.

Подход ATLAS [52] состоит в изоляции машинно-зависимых параметров процедур, которые имеют дело с выполнением оптимизированного матричного умножения на кристалле с кэшем (т. е. кэшем уровня 1). Перемножение на кристалле автоматически создаётся генератором кода, который использует измерения времени для определения факторов правильной блокировки и развёртывания циклов для выполнения оптимизированного умножения на кристалле.

Матричное умножение раскладывается на две части:

1. высокоуровневое, общего размера умножение вне кристалла, которое платформо-независимо,

2. низкоуровневое, фиксированного размера умножение на кристалле, которое машинно-зависимое.

Алгоритм высокого уровня основан на умножении блочных матриц. Предположим, что `A` — это матрица размером $m \times n$, а `B` — это матрица размером $k \times n$. Предположим, что целое число `NB`, количество блоков, делится на m, n и k. Алгоритма умножения блочных матриц мог бы быть написан на языке Scilab следующим образом.

```
function C = offchip-blockmatmul(A, B, NB)
    [m, k] = size(A)
    [k, n] = size(B)
    C = zeros(m, n)
    for ii = 1:NB:m
        I = ii:ii+NB-1
        for jj = 1:NB:n
            J = jj:jj+NB-1
            for kk = 1:NB:k
                K = kk:kk+NB-1
                C(I, J) = C(I, J) + A(I,K)*B(K, J) // on-chip
            end
        end
    end
endfunction
```

В реализации ATLAS инструкция `C(I, J) = C(I, J) + A(I,K)*B(K, J)` выполнена с помощью функции умножения на кристалле.

Главным параметром алгоритма вне кристалла является `NB`, но и другие факторы принимаются во внимание. Действительно, ATLAS может использовать другие реализации этого алгоритма, основанного, например, на другом упорядочивании трёх вложенных циклов. Во всех этих реализациях цикл по k является всегда самым внутренним циклом. А внешний цикл может быть по `m` (по строкам в `A`) и по `n` (по столбцам в `B`). Для того, чтобы найти оптимальное значение параметров (включая `NB`, порядок циклов и другие параметры), ATLAS выполняет автоматизированный эвристический поиск через замеры времени. Для каждого набора исследуемых параметров ATLAS использует генератор кода и измеряет производительность с этими установками. В конечном счёте сохраняется наилучшая реализация и генерируется соответствующий исходный код, затем компилируется.

Умножение на кристалле чувствительно к нескольким факторам, включающим повторное использование кэша, переполнение кэша инструкций, порядок инструкций с плавающей запятой, максимальное количество циклов, проявление возможного распараллеливания и ошибки кэша. Более того, умножение на кристалле может использовать специальные инструкции, доступные процессору, такие, как инструкции множественных данных одной инструкции (SIMD). Эти инструкции позволяют выполнять ту же работу над множественными

данными одновременно и встроить распараллеливание данных. Есть несколько наборов инструкций SIMD, включающих

- MMX, разработанный фирмой Intel в 1996,
- 3DNow!, разработанный фирмой AMD в 1998,
- Расширение потоковой SIMD (SSE), разработанное фирмой Intel в 1999,
- SSE2, разработанный фирмой Intel в 2001,
- SSE3, разработанный фирмой Intel в 2004,
- Продвинутые векторные расширения, будущее расширение, предложенное фирмой Intel в 2008.

Например, инструкция MULPD может быть использована для выполнения SIMD-умножения двух пакетных значений с плавающей запятой двойной точности. Похожей инструкцией является ADDPD, которая выполняет суммирование SIMD двух пакетных значений с плавающей запятой двойной точности.

На практике формирование бинарного файла для библиотеки ATLAS может потребовать несколько часов времени ЦП перед тем, как оптимизированные настройки автоматически идентифицируются системой ATLAS.

Эти низкоуровневые методы не является главной заботой пользователей Scilab. Одного это позволяет понимать почему специфичные библиотеки, поставляемые вместе с Scilab'ом, являются разными для Pentium III, Pentium 4 или Dual или Quad Core. Более важно то, что это позволяет знать уровень оптимизации, который мы можем получить от вычислений с плавающей запятой и особенно линейной алгебры, которая является главной целью языка Scilab.

5.4.3 Установка оптимизированных библиотек линейной алгебры для Scilab под Windows

В этом разделе мы представляем метод установки оптимизированных библиотек линейной алгебры для Scilab на Windows.

На Windows мы обычно устанавливаем Scilab после его скачивания с http://www.scilab.org (но некоторые пользователи имеют предустановленный Scilab на своих машинах). Когда мы скачиваем установщик Scilab для Windows и запускаем его, то мы можем выбирать из трёх возможностей:

- Полную установку (по умолчанию), где устанавливаются все модули,
- Установку, где некоторые модули отключены,
- Выборочную установку, где пользователь может выбирать включение или отключение модулей.

По умолчанию Scilab устанавливается с Intel MKL, но этот выбор может быть настроен пользователем в диалоге установки, который представлен на рисунке 39.

В разделе «CPU Optimization for Scilab» диалога установки мы можем выбирать между тремя следующими пунктами.

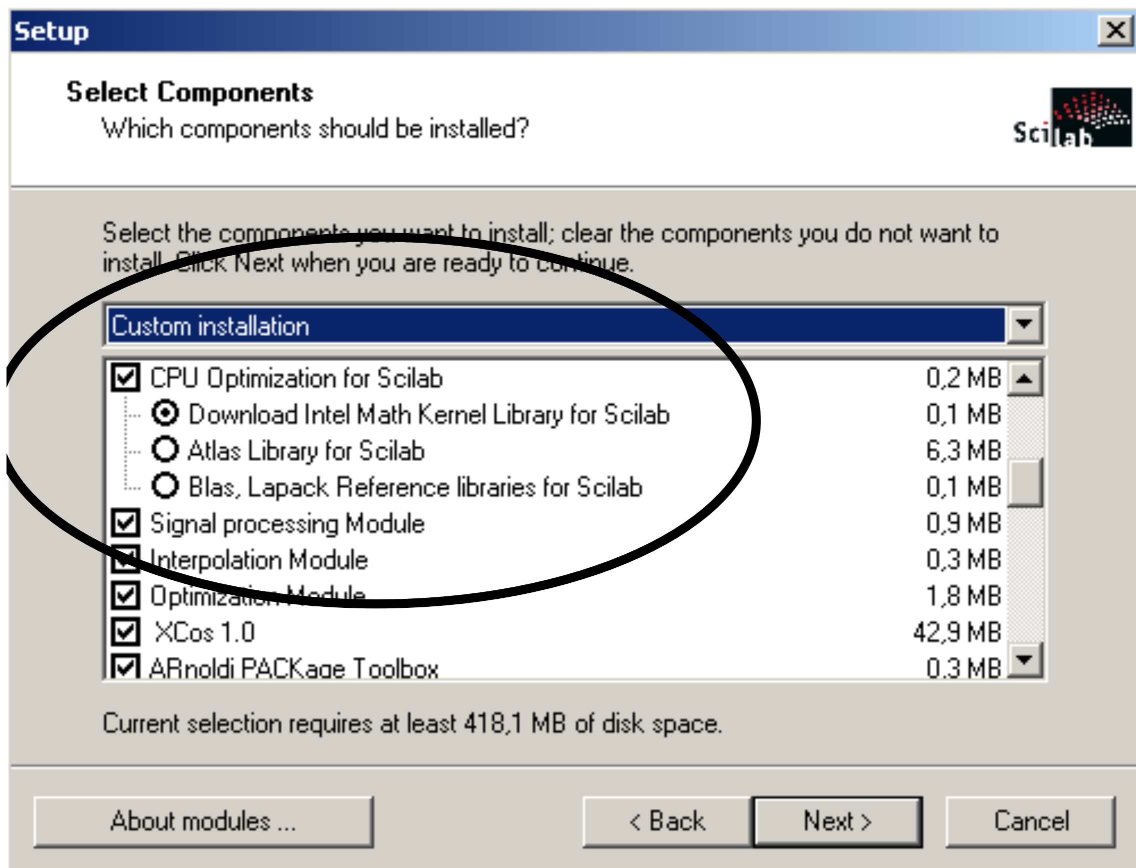

Рис. 39: Выбор библиотеки линейной алгебры под Windows (Scilab v5.2.2). Мы можем выбирать между BLAS, ATLAS и Intel MKL.

- Download Intel Math Kernel Library for Scilab. Эта установка по умолчанию Эта библиотека содержит оптимизированные библиотеки BLAS и набор LAPACK, предоставленный фирмой Intel. Эта библиотека является не полной Intel MKL, а только набором функций, которые используются в Scilab для BLAS и LAPACK.

- Atlas library for Windows. Эта библиотека скомпилирована консорциумом Scilab Consortium для обеспечения оптимальной производительности в широком диапазоне современных компьютеров.

- BLAS, LAPACK Reference library for Windows. Эта библиотека является опорной реализацией, предоставленной `http://www.netlib.org/blas` и `http://www.netlib.org/lapack`.

Эти опции имеют прямое влияние на две динамических библиотеки, которые хранятся в директории `bin` Scilab'а:

- `bin \blasplus.dll` : динамическая библиотека для BLAS,

- `bin \lapack.dll` : динамическая библиотека для LAPACK.

На практике можно иметь одну и ту же версию Scilab'а, установленную с разными библиотеками линейной алгебры. Действительно, несложно настроить директорию установки

и затем выбрать конкретную библиотеку, которую желают использовать. Следовательно, можно установить Scilab в три разные директории, в каждой свою конкретную библиотеку, так, что мы можем сравнить их поведение и производительность. Этот метод используется в разделе 5.4.5, где представлена разница производительности между этими тремя библиотеками в Windows на классическом тесте производительности.

5.4.4 Установка оптимизированных библиотек линейной алгебры для Scilab под Linux

В этом разделе представлен метод установки оптимизированных библиотек линейной алгебры для Scilab под операционной системой Gnu/Linux.

Под Linux сложнее описать процесс установки, поскольку есть много разных дистрибутивов Linux. Тем не менее есть два основных способа получения Scilab под Linux:

- скачать Scilab с http://www.scilab.org,
- использовать систему пакетов дистрибутива Linux, например, Debian, Ubuntu, и т. д. . .

Какой бы источник ни был, Scilab идёт с Reference BLAS и LAPACK, собранными из http://www.netlib.org/blas и http://www.netlib.org/lapack.

Чтобы установить оптимизированную библиотеку ATLAS, мы должны собрать нашу собственную версию ATLAS. Поскольку это несколько сложно и занимает время ЦП (обычно один или два часа сборки), то это рассмотрено в конце этого раздела.

Более простое решение в том, чтобы использовать бинарные файлы из дистрибутива Debian. Бинарные файлы ATLAS в Ubuntu приходят из дистрибутива Debian, так что можно так же получить бинарные файлы ATLAS на Ubuntu. Под Ubuntu мы можем использовать, например, менеджер пакетов Synaptic для установки и удаления бинарных пакетов. В этом случае мы можем использовать бинарный файл `libatlas3gf-base` «ATLAS generic shared», который предоставляется Debian'ом.

Использование бинарного файла оптимизированной библиотеки ATLAS под Linux требует несколько простых изменений вручную, которые мы собираемся описать. В директории `scilab/lib/thirdparty` мы найдём следующие файлы.

```
$ ls scilab-5.3.0/lib/thirdparty
-rw-r--r-- 1 mb mb  6.3M 2010-09-23 14:45 liblapack.so.3gf.0
-rw-r--r-- 1 mb mb  539K 2010-09-23 14:45 libblas.so.3gf.0
```

Когда мы удалим файл `libblas.so.3gf.0` из директории установки Scilab'а, то Scilab будет использовать библиотеку BLAS из системы Linux. Следовательно, чтобы убедиться, что Scilab использует библиотеку ATLAS, которую мы уже установили в систему, мы просто удаляем файл `libblas.so.3gf.0` как в следующем примере.

```
$ rm scilab-5.3.0/lib/thirdparty/libblas.so.3gf.0
```

Для того, чтобы получить хорошую производительность, нам не следует использовать бинарный файл, предоставленный Debian'ом. Вместо этого, следует собрать свою собственную библиотеку ATLAS на конкретную машину, которую мы используем. Это из-за обнаруженных во время сборки конкретных настроек, которые позволяют ATLAS'у улучшить свою производительность. Следовательно, двоичный файл ATLAS, который поставляется в Debian разумно оптимизирован для машины, которую использовал для сборки заведующий пакетом. Он довольно хорошо работает и его легко использовать, но на другой машине не даёт максимальной производительности, которую можно получить.

К счастью, исходные коды доступны на http://math-atlas.sourceforge.net, так что кто угодно в Linux'е может настроить собственную библиотеку ATLAS, чтобы получить хорошую производительность линейной алгебры. Более того, в Debian, процесс сделан гораздо легче, поскольку этот конкретный дистрибутив предоставляет все необходимые инструменты. Более подробно эта тема рассматривается в разделе 5.6.

5.4.5 Пример улучшения производительности

В этом разделе представлен простой эксперимент в Windows, который демонстрирует выгоду использования оптимизированной библиотеки линейной алгебры в Scilab'е. В качестве программы проверки производительности системы, мы рассматриваем перемножение двух квадратных, плотных, действительных матриц значений типа double.

Определим в следующем файле-сценарии функцию `mymatmul`, которая принимает размер матриц `n` в качестве входного аргумента и выполняет перемножение двух матриц, заполненных единичными элементами.

```
function mymatmul ( n )
  A = ones(n,n)
  B = ones(n,n)
  C = A * B
endfunction
```

Оператор умножения, в данном случае, связан с функцией библиотеки BLAS, которая называется DGEMM. Эта функция часто используется для измерения производительности оптимизированных библиотек линейной алгебры.

Следующий набор команд использует функцию `benchfun`, которую мы уже представляли в разделе 5.1.4. Мы устанавливаем размер стека в максимум, затем вычисляем умножение плотных матриц увеличивающегося размера.

```
stacksize("max")
benchfun ( "matmul" , mymatmul , list(100) , 0 , 10 );
benchfun ( "matmul" , mymatmul , list(200) , 0 , 10 );
benchfun ( "matmul" , mymatmul , list(400) , 0 , 10 );
benchfun ( "matmul" , mymatmul , list(1000) , 0 , 10 );
```

Использовался Scilab версии 5.2.2 под Windows XP 32 бита. ЦП — AMD Athlon 3200+ на 2 ГГц, и система работает с 1 ГБ памяти. В следующем примере мы выполняем файл-сценарий программы проверки производительности с библиотекой Atlas.

```
-->exec bench_matmul.sce;
matmul: 10 iterations, mean=0.00300, min=0.00000, max=0.01001
matmul: 10 iterations, mean=0.03304, min=0.02002, max=0.08011
matmul: 10 iterations, mean=0.27940, min=0.26037, max=0.37053
matmul: 10 iterations, mean=4.11892, min=4.08587, max=4.19603
```

Рисунок 40 представляет результаты, которые мы получили от трёх оптимизированных библиотек. Наилучшая производительность получена от Intel MKL, за которой близко следует библиотека Atlas. Производительность библиотеки Reference BLAS-LAPACK очевидно ниже.

Используя оптимизированные библиотеки линейной алгебры, пользователи Scilab'а могут получить быстрые вычисления линейной алгебры. На Windows Scilab поставляется с Intel MKL, которая даёт в большинстве случаев великолепную производительность. Заметим, что эта библиотека коммерческая и предоставляется бесплатно пользователям Scilab, поскольку

Библиотека	N	Среднее (с)
REF-LAPACK	100	0,003004
REF-LAPACK	200	0,033048
REF-LAPACK	400	0,279402
REF-LAPACK	1000	4,118923
ATLAS	100	0,001001
ATLAS	200	0,011016
ATLAS	400	0,065094
ATLAS	1000	0,910309
Intel MKL	100	0,001001
Intel MKL	200	0,010014
Intel MKL	400	0,063091
Intel MKL	1000	0,834200

Рис. 40: Чувствительность производительности перемножения матриц в зависимости от библиотеки линейной алгебры. Проверкой производительности является умножение матрицы на матрицу. Тест выполнялся в Scilab версии 5.2.2 под Windows XP 32 бита. ЦП — AMD Athlon 3200+ на 2 ГГц, и система работает с 1 ГБ памяти.

консорциум Scilab Consortium имеет лицензию на Intel MKL. Под Linux Scilab поставляется с библиотекой reference BLAS-LAPACK. Оптимизированные библиотеки Atlas доступны для из любого дистрибутива GNU/Linux. Более того, на GNU/Linux мы можем собрать свою собственную библиотеку Atlas и создать настроенную библиотеку, которая специально настроена на наш компьютер.

5.5 Измерение числа операций с плавающей запятой за секунду (флопс)

Общая практика в информатике заключается в вычислении числа операций с плавающей запятой, которое может быть выполнено в течение одной секунды. Это даёт аббревиатуру *flops*, что означает FLoating Point Operations by Second (количество операций с плавающей запятой за секунду). В этом разделе мы представляем два стандартных теста производительность в Scilab, включающих в себя вычисление произведения двух плотных матриц и вычисления решения линейных уравнений с помощь LU-разложения. В обоих случаях мы сравниваем производительности библиотек линейной алгебры Reference BLAS, ATLAS и Intel MKL, предоставляемых в Scilab под Windows.

5.5.1 Произведение матрицы на матрицу

В этом разделе мы рассматриваем производительность произведения матрицы на матрицу на машине с Windows, с несколькими библиотеками линейной алгебры.

Давайте рассмотрим произведение `A` и `B`, двух плотных матриц размером $n \times n$, используя оператор `*`. Число операций с плавающей запятой вычисляется напрямую по формуле:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, \qquad (3)$$

для $i, j = 1, 2, \ldots, n$. Чтобы вычислить один элемент C_{ij} матрицы $\mathbf{C}$, приходится выполнять n умножений и n суммирований. В результате, количество операций с плавающей запятой $2n$. Поскольку количество элементов в матрице $\mathbf{C}$ n^2, то общее число операций с плавающей запятой равно $2n^3$.

В этом случае Scilab использует процедуру DGEMM библиотеки BLAS, которая является частью BLAS уровня 3. Основы этого набора BLAS описаны в [15], где авторы подчёркивают, что общий руководящий принцип заключается в том, что эффективные реализации, вероятно, достигаются уменьшением отношения обмена памяти к арифметическим операциям. Это позволяет полностью использовать векторные операции (если возможно) и позволяет встраивать распараллеливание (если возможно). В случае DGEMM число обращений к памяти равно $3n^2$, а число операций с плавающей запятой равно $2n^3$. Следовательно, отношение равно $2n/3$, что является одним из наивысших в библиотеки BLAS. Например, отношение скалярного произведения (уровень 1) равно 1, а отношение для матрично-векторного произведения (уровень 2) равно 2. Следовательно, мы ожидаем получить хорошую производительность для процедуры DGEMM.

Более того, в [52] авторы подчёркивают, что многие процедуры BLAS уровня 3 могут быть эффективно реализованы, давая эффективное произведение матрицы на матрицу. Поэтому хорошая производительность DGEMM является ключевым моментом в общей производительности BLAS. В общем, матрицы $\mathbf{A}$, $\mathbf{B}$ и $\mathbf{C}$ слишком велики, чтобы уместиться в кэше процессора. Поэтому авторы ATLAS'а [52] предлагают использовать алгоритмы с разделением на блоки. Действительно, можно расположить операторы так, чтобы бóльшая часть операций была выполнена с данными в кэше, с помощью алгоритмов с разделением на блоки.

Для того, чтобы измерить производительность произведения матрицы на матрицу в Scilab, мы используем следующий файл-сценарий. Мы используем генератор случайных чисел `rand` для получения матриц, чьи элементы вычисляются из функции нормального распределения. Мы знаем, что генератор случайных чисел в основе `rand` является плохого статистического качества, и что функция `grand` может быть использована в качестве замены. Но это не имеет значения на производительность произведения матрицы на матрицу, так что мы оставим его для простоты. Множитель `1.e6` в вычислении переменной `mflops` преобразует флопсы в мегафлопсы.

```
stacksize("max");
rand( "normal" );
n = 1000;
A = rand(n,n);
B = rand(n,n);
tic();
C = A * B;
t = toc();
mflops = 2*n^3/t/1.e6;
disp([n t mflops])
```

Программа проверки производительности использовалась в Scilab версии 5.2.2 под Windows XP 32 бита. ЦП — AMD Athlon 3200+ на 2 ГГц, и система работает с 1 ГБ памяти. Мы сравниваем здесь производительности трёх библиотек линейной алгебры, предоставленных в Scilab, то есть Reference BLAS, ATLAS и Intel MKL. Для того, чтобы гарантировать выполнимость замеров времени, нам пришлось выбирать отдельно размеры матриц для различных библиотек. Результаты представлены на рисунке 41.

В этом эксперименте самыми быстрыми библиотеками являются Intel MKL и ATLAS

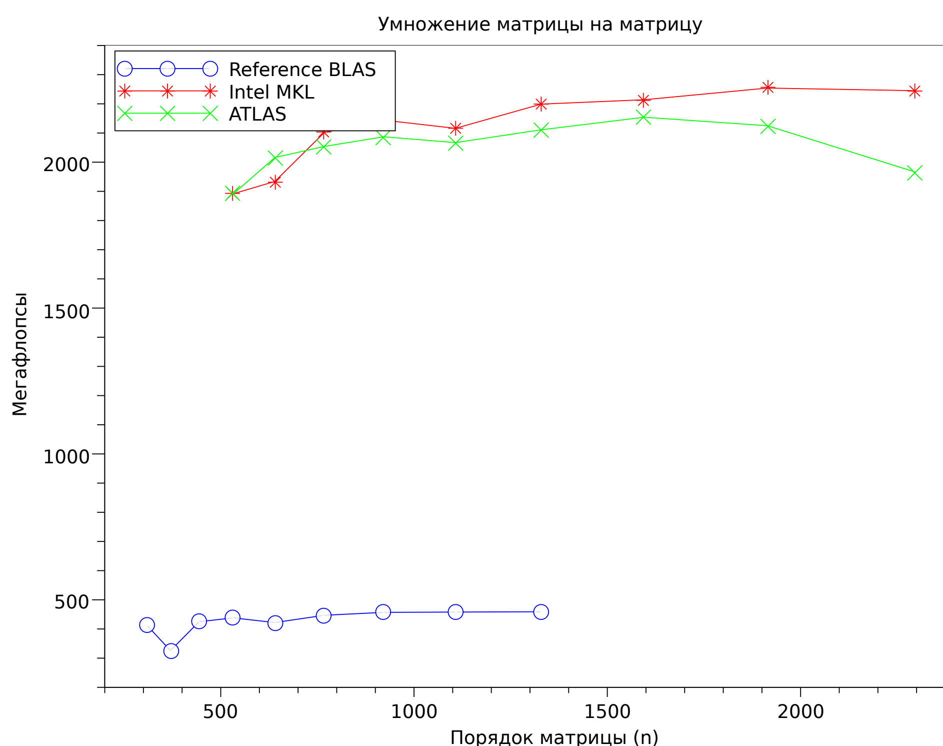

Рис. 41: Производительности умножения действительных, квадратных, плотных матриц в Scilab с различными библиотеками под Windows. Выше — лучше.

(свыше 2000 мегафлопсов), что намного быстрее библиотеки Reference BLAS (около 500 мегафлопсов). Размеры матрицы `n`, которые мы использовали в эксперименте, зависят от действительной производительности библиотек по причине, которую мы сейчас анализируем. Вы выбрали размер `n` для которого библиотека показывает время от 0,1 секунды до 8 секунд. Главным преимуществом создания как можно более быстрых программ определения производительности является сохранение хорошей воспроизводимости. Действительно, если `n` слишком мало, то может случиться, что измеренное время `t` равно нулю. В противоположность этому, библиотека Reference BLAS настолько медленна, что измерение времени растёт так быстро (согласно формуле $2n^3$), что запуск алгоритма для матриц большого размера приводит к очень большим временам, делая процесс замера времени затруднительным на практике.

Теперь сравним действительную производительность библиотек с пиковой производительностью, которую мы можем ожидать для данного конкретного процессора. Процессор — AMD Athlon 3200+, кодовое имя Venice[2], для сокета 939. Частота процессора 2 ГГц. Если процессор способен выполнить одну операцию с плавающей запятой за цикл, то частота 2 ГГц подразумевает, что пиковая производительность должна быть более 2 гигафлопсов, то есть 2000 мегафлопсов. Предыдущий численный эксперимент показывает, что ATLAS и Intel MKL способны достичь эту производительность и даже чуть больше. Это доказывает,

[2]Венеция (прим. перев.)

что процессор был способен выполнить от 1 до 2 операций с плавающей запятой на цикл процессора. Этот процессор — одноядерный, который поддерживает MMX, 3DNow!, SSE, SSE2 и SSE3. Тот факт, что процессор поддерживает эти наборы инструкций может объяснить почему действительная производительность была слегка выше 2000 мегафлопсов.

Давайте рассмотрим чувствительность производительности к размеру кэша. Может случиться, что библиотека линейной алгебры покажет повышенную производительность для матриц, которые могут быть полностью сохранены в кэше. Цель библиотек ATLAS и Intel MKL в том, чтобы этого не случилось. Следовательно, производительность не должна зависеть от от размера кэша. Давайте проверим, что это действительно происходит в этом конкретном эксперименте. Кэш данных L1 равен 64 КБ, а кэш L2 равен 512 КБ. Поскольку число операций с плавающей запятой двойной точности требует 8 байт (т. е. 64 бита), то число чисел двойной точности, которое может быть сохранено в кэше L1 равно $64000/8 = 8000$ чисел двойной точности. Это соответствует плотной, квадратной матрице чисел двойной точности размером 89×89. Кэш L2, с другой стороны, может хранить плотную, квадратную матрицу чисел двойной точности размером 256×256. Поскольку размеры наших матриц варьируются от 500×500 до 2000×2000, то мы видим, что общее число чисел двойной точности не может быть (по большей мере) целиком сохранено в кэше. Следовательно, мы приходим к заключению, что алгоритмы, используемые в Intel MKL и ATLAS не слишком чувствительны к размеру кэша.

5.5.2 Обратный слэш

В этом разделе мы измеряем производительность оператора обратный слеш в Scilab'е под Linux.

Этот тест производительности часто называется «тест производительности LINPACK», поскольку этот тест был создан при разработке проекта LINPACK. Теста производительности LINPACK используется и сейчас, например, в качестве меры производительности для высокопроизводительных компьютеров, таких, как машины из топа 500 (Top 500) [4], например. Эта библиотека вытеснена BLAS'ом и LAPACK'ом, которые используются в Scilab'е.

Давайте рассмотрим квадратную, плотную, действительную матрицу `A` размером $n \times n$ и действительный вектор `b` размером $n \times 1$. Оператор « \» позволяет вычислить решение `x` линейного уравнения `A*x=b`. Внутри, если матрица хорошо обусловлена, Scilab производит LU-разложение матрицы `A`, используя перемещение строк. Затем, мы выполняем прямую и обратную подстановки, используя LU-разложение. Эти операции основаны на библиотеке LAPACK, которая, внутри, использует библиотеку BLAS. Точные имена процедур LAPACK, используемых оператором обратного слэша, представлены в разделе 5.6.

Решение систем линейных уравнений является типичным случаем использования библиотек линейной алгебры. Поэтому этот конкретный тест производительности очень часто используется для измерения производительности вычислений с плавающей запятой на конкретной машине [39, 16, 42]. Число операций с плавающей запятой для всего процесса равно $\frac{2}{3}n^3 + 2n^2$. Поскольку это число растёт по закону n^3, а число элементов матрицы лишь n^2, то мы можем ожидать хорошей производительности в этом тесте.

Следующий файл-сценарий измерить производительность оператора обратного слэша в Scilab'е. Мы настраиваем генератор случайных чисел в функции `rand` таким образом, что он формирует матрицы с элементами, вычисленными по функции нормального распределения. Наконец, мы вычисляем число операций с плавающей запятой и делим на `1.e6` для

того, чтобы получить мегафлопсы.

```
n = 1000;
rand( "normal" );
A = rand(n,n);
b = rand(n,1);
tic();
x = A \b;
t = toc();
mflops = (2/3*n^3 + 2*n^2)/t/1.e6;
disp([n t mflops])
```

Мы выполняем этот тест в scilab-5.3.0-beta-4 на Linux Ubuntu 32 бита. Процессор Intel Pentium M с частотой 2 ГГц, оперативная память 1 ГБ. Размер кэша 2 МБ. Процессор поддерживает наборы инструкций MMX, SSE и SSE2. Рисунок 42 представляет производительности Reference BLAS и ATLAS. Версия ATLAS'а, которую мы используем, предоставлена Ubuntu. Из-за практических ограничений эти эксперименты остановлены, когда время, требуемое на выполнение одного оператора обратного слэша, стало больше 8 секунд.

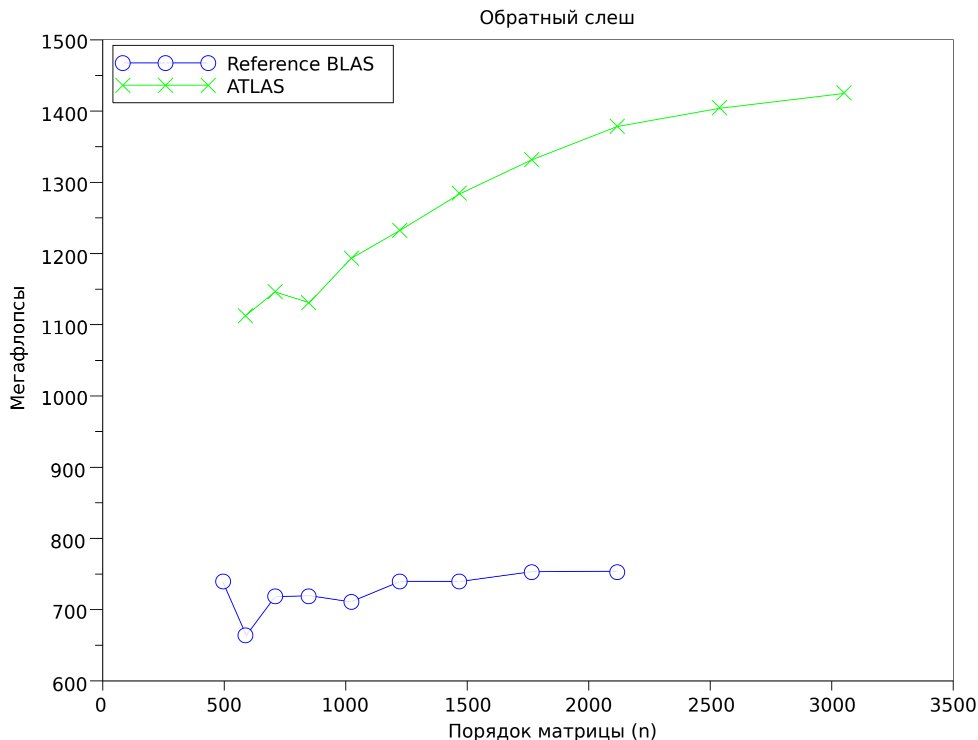

Рис. 42: Производительность решения вещественной, квадратной, плотной, линейной системы уравнений оператором обратного слэша с помощью различных библиотек на Linux.

Как мы можем видеть, библиотека ATLAS улучшает производительность оператора обратного слеша примерно в два раза. Более того, мегафлопсы сильно возрастают с размером матрицы, а производительность библиотеки Reference BLAS кажется остаётся одной и той же. Частота процессора 2 ГГц, это предполагает, что мы можем ожидать пиковую производительность свыше 2000 мегафлопсов. Действительная производительность ATLAS'а на

этих матрицах в пределах [1000,1400], что несколько разочаровывает. Поскольку эта производительность возрастает, то пиковая производительность может быть достигнута для бóльших матриц, но это здесь не проверялось, поскольку времена возрастают очень быстро, делая эксперименты долгими для выполнения.

5.5.3 Многоядерные вычисления

В этом разделе мы покажем, что Scilab может выполнять многоядерные вычисления с плавающей запятой, если мы используем, например, библиотеку линейной алгебры Intel MKL, которая в Windows предоставляется вместе с Scilab'ом.

Мы использовали для этого теста Scilab v5.3.0-beta-4 на операционной системе Windows Vista Ultimate 32 бита. Процессор Intel Xeon E5410 с 4 ядрами на 2,33 ГГц [25, 53]. Процессор может управлять 4 потоками (то есть один поток на ядро) и имеет L2 кэш на 12 МБ. Операционная система может иметь только 4 ГБ физической памяти. Рисунок 43 показывает результаты проверки производительности на перемножении матрицы на матрицу, которое мы уже представили в разделе 5.5.1.

Библиотека	N	Пользовательское время (с)	мегафлопсы
Intel MKL	4766	8,172	26494
ATLAS	1595	3,698	2194
Ref. BLAS	444	0,125	1400

Рис. 43: Производительность произведения матрицы на матрицу на 4-х ядерной системе под Windows с различными библиотеками.

Учитывая, что этот процессор имеет 4 ядра на 2,33 ГГц, мы ожидаем производительность больше 9,33 гигафлопсов. Действительная производительность Intel MKL равна 26,494 гигафлопса, что указывает на то, что эта библиотека использовала 4 ядра, выполняя 2,8 операции с плавающей запятой за цикл. С другой стороны, библиотеки ATLAS и Ref-BLAS не используют 4 ядра этой машины и дают значительно меньшие результаты.

5.6 Замечания и ссылки

В разделе 5.2.3 мы анализировали производительность алгоритма суммирования. Можно было бы указать, что точность суммы зависит от размера данных и конкретных значений в наборе данных. Это проанализировано, например, Уилкинсоном (Wilkinson)[54] и Хайэмом (Higham) [23]. Точнее, можно указать, что набор данных может быть плохообусловлен по отношению к сумме, например, если значения смешанных знаков (то есть, некоторые положительные, а некоторые отрицательные) и очень разных амплитуд. В этом случае мы можем наблюдать, что относительная ошибка суммы может быть большой.

В этом документе мы сфокусировались главным образом на производительности и ссылаемся на предыдущие ссылки на вопросы точности. На практике сложно полностью разделить эти два вопроса. Действительно, гораздо легче получить лучшую производительность, если мы полностью проигнорируем вопросы точности. Поэтому можно рассматривать оптимизацию только после того, как мы будем иметь достаточно широкий базис тестов, включая и корректность и точность. Это гарантирует, что во время процесса «оптимизации» мы не введём программную ошибку или, хуже того, ненужную ошибку в результате.

Тема, которая тесно связана с векторизацией алгоритмов Scilab'а — это индексирование матриц, то есть, выбор набора элементов в матрице. Статья Эддинса (Eddins) и Шура (Shure) [18] фокусируется на этой теме в контексте Matlab®, где авторы представляют индексацию векторов, индексацию матриц с помощью двух подпрограмм, линейную индексацию и логическую индексацию на основе логических выражений. Matlab является зарегистрированной торговой маркой Mathworks.

В разделе 5.2.1 мы представили несколько методов, связанных с принципами компиляции. Классической ссылкой по вопросу компиляторов является «Книга дракона[3]» [5]. Эта книга вводит в проектирование компиляторов, включая лексический анализ, регулярные выражения, машины конечных состояний и методы проверки синтаксиса.

В разделе 5.4.1 мы кратко представили проект ATLAS. Уоли (Whaley) и Донгарра (Dongarra) [51] описывают метод автоматического формирования и оптимизации числового программного обеспечения для процессоров с глубокой иерархией памяти и конвейерных блоков реализации функций.

В разделе 5.1.3 мы представили алгоритм метода исключения Гаусса и заявили, что этот алгоритм (но не конкретная реализация, которую мы представили) используется в Scilab'е в основе оператора обратного слеша. Действительное поведение оператора обратного слэша в Scilab'е следующее. Сначала мы выполним вызов процедуры `DGETRF` из LAPACK, которая раскладывает матрицу $\mathbf{A}$ на $\mathbf{PA} = \mathbf{LU}$, где $\mathbf{P}$ — матрица перестановок, $\mathbf{L}$ — нижняя треугольная матрица, а $\mathbf{U}$ — верхняя треугольная матрица. Диагональные элементы $\mathbf{L}$ являются единицами. Затем мы вызываем процедуру `DGECON` из LAPACK, которая аппроксимирует инвертирование числа обусловленности 1-нормы матрицы $\mathbf{A}$. В этом месте есть два варианта. Если число обусловленности матрицы не слишком велико, то система линейных уравнений не является плохобусловленной (это не точно, поскольку обусловленность только аппроксимирована, но алгоритмы делают гипотезу о том, что оценка может быть достоверной). В этом случае мы вызываем процедуру `DGETRS` из LAPACK, которая решает систему линейных уравнений, используя разложение $\mathbf{PA} = \mathbf{LU}$. Точнее, этот шаг использует перестановку строк, а затем прямой и обратный ход для нахождения решения $\mathbf{x}$ в зависимости от правостороннего $\mathbf{b}$. В другом варианте, то есть, если матрица является плохообусловленной, мы используем модифицированную версию процедуры `DGELSY` из LAPACK, которая решает соответствующую задачу наименьших квадратов. Этот выбор может уменьшить точность матрицы промежуточного условия [10].

В [39] Клив Молер (Cleve Moler) анализирует производительность Matlab'а в различных ситуациях, включая решение системы уравнений с помощью оператора обратного слэша. Чтобы измерить производительность вычисления, он подсчитывает число флопсов, измеряя время выполнения в секундах с помощью функций Matlab'а `tic` и `toc` и оценивая число операций с плавающей запятой для этого конкретного вычисления. Автор использует матрицу случайных чисел с элементами, полученными из функции нормального распределения и использует функцию `randn` в Matlab'е. Увеличивая размер матрицы, он наблюдает пик производительности, который соответствует размеру кэша компьютера SPARC-10, который он использует для этого теста. Эта машина имеет кэш в один мегабайт, что соответствует квадратной матрицы порядка

```
-->floor(sqrt(1.e6/8))
 ans  =
    353.
```

[3] Название «Книга дракона» происходит от того, что на обложке книги изображены сражающиеся рыцарь и дракон. (*прим. перев.*)

Точнее, Молер использует определение, которое связывает один мегабайт с 2^{20} байтами, но результат грубо тот же. Пиковая скорость в мегафлопсах получена для матрицы, которая точно укладывается в кэш. Он заявляет, что этот эффект кэша является следствием библиотеки LINPACK. Он подчёркивает, что это одна из главных мотиваций для проекта LAPACK. Этот документ был написан в 1994: Matlab использует библиотеку LAPACK с 2000 [40]. В [16] Джек Донгарра (Jack Dongarra) собирает различные производительности на этом тесте производительности. Источники по этому вопросу доступны на netlib.org [42].

Общие источники, связанные с улучшением производительности Matlab'а, доступны на [38].

В разделе 5.4.3 мы обсудили установку Scilab'а на Windows. Эта тема обсуждается подробнее на [14], где Аллан Корнэ (Allan Cornet), разработчик из Scilab Consortium, представляет полный процесс установки Scilab 5.2.0 на Windows.

В разделе 5.4.4 мы представили установку двоичной версии ATLAS для Scilab'а на Linux/Ubuntu. Поскольку ATLAS оптимизирован на время сборки, то лучше компилировать собственный ATLAS на той машине, которую планируется использовать. Это гарантирует, что двоичный файл, который мы используем, действительно оптимизирован в соответствии с параметрами текущей машины (например, размер кэша). В [48] Сильвестр Ледрю (Sylvestre Ledru), который является разработчиком из Scilab Consortium и поддерживает пакет Debian Science, предоставляет информацию об установке ATLAS'а на Debian. Этот вопрос представлен более глубоко в [30].

Есть другие ссылки, которые могут помочь пользователям Scilab'а под Linux. В [31] Ледрю описывает процесс установки Scilab 5.1.1 на Linux. В [32] Ледрю обсуждает обновление управления пакетами BLAS и LAPACK в Debian. В [29] Ледрю представляет метод, который позволяет переключаться между различными оптимизированными библиотеками линейной алгебры.

В разделе 5.1.4 мы представили функцию `benchfun`, которая измеряет производительность указанной функции. В качестве альтернативы мы можем использовать модуль Scibench [6], который предоставляется в ATOMS. Целью проекта Scibench является предоставление инструментов для измерения производительности функции. Более того, модуль предоставляет коллекцию программ измерения производительности для измерения производительности Scilab'а. Внутри модуля Scibench функция `scibench_benchfun` имеет ту же цель, что и `benchfun`, которая была представлена в разделе 5.1.4.

5.7 Упражнения

Упражнение 5.1 (*Метод исключения Гаусса с перестановкой строк*) В разделе 5.1.3 мы представили две функции `gausspivotalnaive` и `gausspivotal`, которые включают в себя алгоритм метода исключения Гаусса с перестановкой строк. Используйте функцию `benchfun` и сравните действительную производительность этих двух функций.

Упражнение 5.2 (*Комбинации*) Рассмотрим две матрицы чисел типа double размером $1 \times n$ `x` и `Y`. Допустим, что нам хотелось бы создать функцию, формирующую матрицу `c` размером $2 \times m$, содержащую все комбинации векторов `x` и `y`, с $m = n^2$. То есть, мы хотели бы получить пары `[x(1);y(2)]`, `[x(1);y(3)]`, ..., `[x(n);y(n)]`. Например, рассмотрим следующие матрицы `x` и `y`.

```
-->x = [10 11 12];
-->y = [13 14 15];
```

Сформируем следующую матрицу `c`:

```
-->c
 c  =
   10.    10.    10.    11.    11.    11.    12.    12.    12.
   13.    14.    15.    13.    14.    15.    13.    14.    15.
```

- Создайте простую реализацию, формирующую матрицу `c`.
- Используйте произведение Кронекера `.*.` и создайте векторизованную функцию для получения матрицы `c`.
- Используйте выделения матриц, чтобы создать векторизованную функцию для формирования матрицы `c`.

6 Благодарности

Я хочу выразить благодарность Аллану Корнэ (Allan Cornet), Сильвестру Ледрю (Sylvestre Ledru), Бруно Жофрэ (Bruno Jofret) и Бернарду Хугуэнею (Bernard Hugueney) за помощь в подготовке этого документа. Я благодарю Сильвестра Ледрю за информацию, которой он поделился относительно пакета ATLAS на Debian. Бруно Жофрэ ввёл меня в управление памятью в стеке Scilab'a. Я благодарю Аллана Корнэ за информацию, которой он поделился относительно пакетов ATLAS и Intel MKL на Windows. Я благодарю Бенуа Жепфера (Benoit Goepfert) за его комментарии к этому документу. Я благодарю Сержа Стира (Serge Steer) и Майка Пэйджеса (Mike Pages) за их обсуждения гиперматриц.

7 Ответы к упражнениям

7.1 Ответы к разделу 2

Ответ к упражнению 2.1 (*Максимальный размер стека*) Результат алгоритма, который устанавливает размер стека, зависит от операционной системы. Это результат на Windows-машине:

```
-->format(25)
-->stacksize("max")
-->stacksize()
 ans  =

    110054096.    34881.
```

а это результат на Linux-машине

```
-->format(25)
-->stacksize("max")
-->stacksize()
 ans  =
    28176384.    35077.
```

□

Ответ к упражнению 2.2 (*who_user*) Следующий пример — результат на Linux-машине.

```
-->who_user()
 Пользовательские переменные:
 home
 Используется 7 элементов из 4990798
-->A=ones(100,100);
-->who_user()
 Пользовательские переменные:
 A          home
 Используется 10009 элементов из 4990796
-->home
 home  =
 /home/mb
```

На самом деле могут быть иные переменные, определённые пользователем, которые могут быть распечатаны функцией who_user. Например, я установил модуль ATOMS «uncprb», и вот что я получаю при запуске:

```
-->who_user()
 Пользовательские переменные:
 uncprblib
 Используется 217 элементов из 4990785
```

Это из-за того, что переменная uncprblib содержит библиотеку, связанную с этим конкретным модулем. □

Ответ к упражнению 2.3 (*whos*) Следующий пример представляет вывод функции whos, которая распечатывает имя, тип, размер и байты, требуемые всеми переменными в текущей среде.

```
-->whos
Name                    Тип            Размер          Байт
$                       polynomial     1 на 1          56
%driverName             string*        1 на 1          40
%e                      constant       1 на 1          24
%eps                    constant       1 на 1          24
[...]
guilib                  library                        488
helptoolslib            library                        728
```

```
home                      string          1 на 1          56
integerlib                library                          1416
interpolationlib          library                          336
[...]
```

□

7.2 Ответы к разделу 3

Ответ к упражнению 3.1 (*Поиск файлов*) Следующая функция `searchSciFilesInDir` ищет эти файлы в заданной директории.

```
function filematrix = searchSciFilesInDir ( directory , funname )
  filematrix = []
  lsmatrix = ls ( directory )';
  for f = lsmatrix
    issci = regexp(f,"/(.*).sci/");
    if ( issci <> [] ) then
      scifile = fullfile ( directory , f )
      funname ( scifile )
      filematrix($+1) = scifile
    end
  end
endfunction
```

В нашем примере мы просто распечатываем имя файла. Следующая функция `mydisplay` использует функцию `mprintf` распечатывает имя файла в консоли. Для того, чтобы напечатать короткое сообщение, мы убираем имя директории из отображаемой строки. Для того, чтобы выделить название директории из имени файла, мы используем функцию `fileparts`.

```
function mydisplay ( filename )
  [path,fname,extension]=fileparts(filename)
  mprintf ("%s%s\n",fname,extension)
endfunction
```

Далее мы используем функцию `searchSciFilesInDir` для распечатки файлов-сценариев Scilab из директории Scilab'а, содержащую макросы, связанные с многочленами. Переменная `SCI` содержит название директории, в которой находится директория Scilab'а. Для того, чтобы определить абсолютное имя файла, содержащее имя директории, содержащей макросы, мы используем функцию `fullfile`, которая собирает свои входные аргументы в единую строку, отделяя директории разделителем, соответствующим текущей операционной системе (слеш «/» под Linux и обратный слеш «\» под Windows).

```
-->directory = fullfile(SCI,"modules","polynomials","macros")
 directory  =
 /home/username/scilab-5.2.2/share/scilab/modules/polynomials/macros
-->filematrix = searchSciFiles ( directory , mydisplay );
invr.sci
polfact.sci
cmndred.sci
[...]
horner.sci
rowcompr.sci
htrianr.sci
```

□

Ответ к упражнению 3.2 (*Запрос типизированных списков*) Мы уже видели, что функция `definedfields` возвращает целые числа с плавающей запятой, связанные с расположением полей в структуре данных. Это не совсем то, что нам здесь нужно, но легко построить нашу собственную функцию на этом строительном блоке. Следующая функция `isfielddef` принимает типизированный список `tl` и строку

fieldname в качестве входных аргументов и выдаёт логическую переменную bool, которая равна *истине*, если поле определено, и *лжи*, если нет. Сначала мы вычисляем ifield, которая является индексом поля fieldname. Чтобы это сделать, мы ищем строку fieldname в полном списке полей, определённом в tl(1) и используем find для поиска требуемого индекса. Затем мы используем функцию definedfields для получения матрицы определённых индексов df. Затем мы ищем индекс ifield в матрице df. Если поиск удался, то переменная k не будет пустой, что значит, что поле определено. Если поиск не удался, то переменная k будет пустой, то означает, что поле не определено.

```
function bool = isfielddef ( tl , fieldname )
  ifield = find(tl(1)==fieldname)
  df = definedfields ( tl )
  k = find(df==ifield)
  bool = ( k <> [] )
endfunction
```

Несмотря на то, что программа довольно абстрактна, она остаётся простой и эффективной. Её главное преимущество состоит в том, чтобы показать как типизированные списки могут вести к очень динамичным структурам данных □

7.3 Ответы к разделу 5

Ответ к упражнению 5.1 (*Метод исключения Гаусса с перестановкой строк*) Мы используем следующий файл-сценарий, который создаёт матрицу размером 100×100 и использует её для сравнения производительности функций gausspivotalnaive и gausspivotal.

```
stacksize("max");
n = 100;
A = grand(n,n,"def");
e = ones(n,1);
b = A * e;
benchfun("naive",gausspivotalnaive,list(A,b),1,10);
benchfun("fast",gausspivotal,list(A,b),1,10);
```

Этот файл-сценарий производит следующий вывод:

```
naive: 10 iterations, mean=1.541300, min=1.482000, max=2.03
fast: 10 iterations, mean=0.027000, min=0.010000, max=0.070000
```

В этом случае для этого размера матрицы среднее улучшение производительности равно $1{,}5413/0{,}027 = 57{,}08$. Фактически, оно могло бы быть больше, при условии, что мы используем достаточно большие матрицы. Для $n = 200$ мы наблюдали отношение производительностей больше 100. □

Ответ к упражнению 5.2 (*Комбинации*) Мы рассматриваем два вектора x и y, с количеством элементов n. Целью этого упражнения является создание функции, способной генерировать матрицу c размером $2 \times m$, содержащую все комбинации векторов x и y, с количеством элементов $m = n^2$.

Наша первая идея заключается в использовании двух вложенных циклов: эта идея реализована в следующей функции. Первый цикл по ix перебирает элементы в x, в то время, как цикл по iy перебирает элементы в y. Алгоритм устанавливает начальное значение k=1. Каждый раз, когда мы создаём новую комбинацию, мы сохраняем её в k-том столбце матрицы c, а затем мы увеличиваем k.

```
function c = combinatevectorsSlow ( x , y )
   cx=size(x,"c")
   cy=size(y,"c")
   c=zeros(2,cx*cy)
   k = 1
   for ix = 1 : cx
     for iy = 1 : cy
       c(1:2,k) = [x(ix);y(iy)]
       k = k + 1
     end
```

```
    end
endfunction
```

Показанная функция работает, но она медленная, поскольку требуется выполнить n^2 циклов. Следовательно, целью является удалить эти два вложенных цикла и сформировать комбинации с помощью векторизованных инструкций.

Нашей второй идеей является использование произведения Кронекера для формирования всех комбинаций одной-единственной инструкцией. Следующий пример показывает основную идею формирования всех комбинаций векторов [10 11 12] и [13 14 15].

```
-->[10 11 12] .*. [1 1 1]
 ans  =
   10.   10.   10.   11.   11.   11.   12.   12.   12.
-->[1 1 1] .*. [13 14 15]
 ans  =
   13.   14.   15.   13.   14.   15.   13.   14.   15.
```

Следующая реализация является прямым обобщением предыдущего примера.

```
function c = combinatevectorsKron ( x , y )
    cx=size(x,"c")
    cy=size(y,"c")
    c(1,:) = x .*. ones(1,cy)
    c(2,:) = ones(1,cx) .*. y
endfunction
```

Наша третья идея заключается в использовании операций выделения матрицы. Действительно, первый ряд в c — это просто утроенная копия вектора x. Вот почему мы можем создавать матрицу повторяемых индексов на основе выделения матрицы, и выделять соответствующие элементы за одну-единственную инструкцию. Повторенные индексы также созданы с помощью индексов, повторенных выделением матрицы. Эти идеи представлены в следующем примере, который вычисляет первую строку матрицы c. В этом примере мы сначала создаём индексы i1 от 1 до 3. Затем мы создаём индексы j1 многократно выделяя строку. Затем мы создаём k1 используя функцию matrix, которая преобразует матрицу j1 в вектор-строку индексов. Наконец, мы используем k1 в качестве индексов столбцов в x.

```
-->x = [10 11 12];
-->i1 = 1:3
 i1  =
    1.    2.    3.
-->j1 = i1([1;1;1],:)
 j1  =
    1.    2.    3.
    1.    2.    3.
    1.    2.    3.
-->k1 = matrix(j1,1,9)
 j1  =
   1.   1.   1.   2.   2.   2.   3.   3.   3.
-->x(:,k1)
 ans  =
   10.   10.   10.   11.   11.   11.   12.   12.   12.
```

Мы можем использовать те же принципы, чтобы формировать вторую строку в c. Но эти элементы не просто повторяются, они, к тому же, чередуются. Для того, чтобы это сделать, мы используем ту же самую идею, что и прежде, но транспонируем индексы перед тем, как преобразовать матрицу индексов в вектор индексов.

```
-->y = [13 14 15];
-->i2 = 1:3
 i2  =
    1.    2.    3.
-->j2 = i2([1;1;1],:)
```

```
 j2  =
    1.    2.    3.
    1.    2.    3.
    1.    2.    3.
-->k2 = matrix(j2',1,9)
 k2  =
   1.   2.   3.   1.   2.   3.   1.   2.   3.
-->y(:,k2)
 ans  =
   13.   14.   15.   13.   14.   15.   13.   14.   15.
```

В предыдущем примере важно заметить оператор одиночная скобка в выражении `matrix(j2',1,9)`. Следующий файл-сценарий является прямым обобщением предыдущих идей.

```
function c = combinatevectorsExtr ( x , y )
   // Результаты всех комбинаций двух векторов x и y
   cx=size(x,"c")
   cy=size(y,"c")
   //
   i1 = 1:cx
   j1 = i1(ones(cy,1),:)
   k1 = matrix(j1,1,cx*cy)
   //
   i2 = 1:cy
   j2 = i2(ones(cx,1),:)
   k2 = matrix(j2',1,cx*cy)
   //
   c = [
      x(:,k1)
      y(:,k2)
   ]
endfunction
```

В следующем файле-сценарии мы используем функцию `benchfun` для сравнения производительности предыдущих функций.

```
n = 300;
x=(1:n);
y=(1:n);
benchfun("Slow",combinatevectorsSlow,list(x,y),1,10);
benchfun("Kron.",combinatevectorsKron,list(x,y),1,10);
benchfun("Extr.",combinatevectorsExtr,list(x,y),1,10);
```

Этот файл-сценарий обычно производит следующий вывод:

```
Slow: 10 iterations, mean=0.24010, min=0.23700, max=0.25000
Kron.: 10 iterations, mean=0.01440, min=0.01300, max=0.02200
Extr.: 10 iterations, mean=0.00960, min=0.00700, max=0.01500
```

Мы видим, что реализация на основе произведения Кронекера больше, чем в 10 раз быстрее в среднем. Заметим, что нет значительной разницы между функциями `combinatevectorsKron` и `combinatevectorsExtr`, хотя `combinatevectorsExtr` кажется немного быстрее. Это можно объяснить тем, что `combinatevectorsKron` выполняет много умножений с плавающей запятой, которые не нужны поскольку один из операндов является единицей. □

Предметный указатель

clear, 16
det, 25
fullfile, 13
getos, 13
horner, 24
list, 31
poly, 23
regexp, 22
roots, 25
stacksize, 6
string, 19
tlist, 33
typeof, 17
type, 17
who, 12
MSDOS, 13
SCIHOME, 13
SCI, 13
TMPDIR, 13

[], 67
add_param, 76
argn, 59
check_param, 81
deff, 54
error, 71
get_function_path, 54
get_param, 76
gettext, 71
init_param, 76
tic, 103
timer, 103
toc, 103
typeof, 54
type, 54
varargin, 59
varargout, 59
warning, 71

ATLAS, 128

BLAS, 128

callback, 57, 58
cell, 48

DGEMM, 129

flops, 135
funcprot, 56, 57

Intel, 128
interpreter, 114

LAPACK, 128

macros, 54
MKL, 128
mlist, 44

PCRE, 19
polynomials, 22
primitive, 54

regular expression, 19

struct, 45

ТАД, 38
векторизация, 102
макрос, 54
многочлен, 22
примитив, 54
профиль функции, 105

регулярное выражение, 19
тип абстрактных данных, 38
флопс, 135
функции обратного вызова, 58
число операций с плавающей запятой за секунду, 135

Список литературы

[1] ATLAS – Automatically Tuned Linear Algebra Software. http://math-atlas.sourceforge.net.

[2] Blas – Basic Linear Algebra Subprograms. http://www.netlib.org/blas.

[3] Lapack – Linear Algebra PACKage. http://www.netlib.org/lapack.

[4] The Linpack benchmark. http://www.top500.org/project/linpack.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] Michael Baudin. Scibench. http://atoms.scilab.org/toolboxes/scibench.

[7] Michael Baudin. How to emulate object oriented programming in Scilab. http://wiki.scilab.org/Emulate_Object_Oriented_in_Scilab, 2008.

[8] Michael Baudin. Apifun - Check input arguments in macros. http://forge.scilab.org/index.php/p/apifun/, 2010.

[9] Michael Baudin. The derivative function does not protect against conflicts between user-defined functions and developper-defined variables. http://bugzilla.scilab.org/show_bug.cgi?id=7104, 2010.

[10] Michaël Baudin. The help page of backslash does not print the singularity level - bug report #7497. http://bugzilla.scilab.org/show_bug.cgi?id=7497, July 2010.

[11] Michael Baudin. The integrate function does not protect against conflicts between user-defined functions and developper-defined variables. http://bugzilla.scilab.org/show_bug.cgi?id=7103, 2010.

[12] Michael Baudin. The neldermead component does not protect under particular user-defined objective functions. http://bugzilla.scilab.org/show_bug.cgi?id=7102, 2010.

[13] Michaël Baudin. There is no pascal function - bug report #7670. http://bugzilla.scilab.org/show_bug.cgi?id=7670, August 2010.

[14] Allan Cornet. Scilab installation on Windows. http://wiki.scilab.org/howto/install/windows, 2009.

[15] J. J. Dongarra, Jermey Du Cruz, Sven Hammerling, and I. S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1):18–28, 1990.

[16] Jack J. Dongarra. Performance of various computers using standard linear equations software. http://www.netlib.org/benchmark/performance.ps, 2013.

[17] William H. Duquette. Snit's not incr tcl. http://www.wjduquette.com/snit/.

[18] Steve Eddins and Loren Shure. Matrix indexing in Matlab. http://www.mathworks.com.

[19] Jean-Luc Fontaine. Stooop. http://jfontain.free.fr/stooop.html.

[20] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[21] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[22] Philip Hazel. Pcre – Perl Compatible Regular Expressions. http://www.pcre.org/.

[23] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[24] Intel. Intel Math Kernel Library. http://software.intel.com/en-us/intel-mkl/.

[25] Intel. Intel Xeon Processor E5410. http://ark.intel.com/Product.aspx?id=33080, 2010.

[26] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Third Edition, Addison Wesley, Reading, MA, 1998.

[27] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[28] Sylvestre Ledru. Localization. http://wiki.scilab.org/Localization.

[29] Sylvestre Ledru. Handle different versions of BLAS and LAPACK. http://wiki.debian.org/DebianScience/LinearAlgebraLibraries, 2010.

[30] Sylvestre Ledru. Linear algebra libraries in Debian. http://people.debian.org/~sylvestre/presentation-linear-algebra.pdf, August 2010.

[31] Sylvestre Ledru. Scilab installation under linux. http://wiki.scilab.org/howto/install/linux, 2010.

[32] Sylvestre Ledru. Update of the linear algebra libraries in Debian. http://sylvestre.ledru.info/blog, April 2010.

[33] Pierre Maréchal. Code conventions for the Scilab programming language. http://wiki.scilab.org, 2010.

[34] The Mathworks. Matlab - Product Support - 1106 - Memory Management Guide. http://www.mathworks.com/support/tech-notes/1100/1106.html.

[35] The Mathworks. Matlab - Product Support - 1107 - Avoiding Out of Memory Errors. http://www.mathworks.com/support/tech-notes/1100/1107.html.

[36] The Mathworks. Matlab - Product Support - 1110 - Maximum Matrix Size by Platform. http://www.mathworks.com/support/tech-notes/1100/1110.html.

[37] The Mathworks. Matlab - Technical Solution - What are the benefits of 64-bit Matlab versus 32-bit Matlab? http://www.mathworks.com/support/solutions/en/data/1-YVO5H/index.html.

[38] The Mathworks. Techniques for improving performance. http://www.mathworks.com/help/techdoc/matlab_prog/f8-784135.html.

[39] Cleve Moler. Benchmarks: Linpack and Matlab - Fame and fortune from megaflops. http://www.mathworks.com/company/newsletters/news_notes/pdf/sumfall94cleve.pdf, 1994.

[40] Cleve Moler. Matlab incorporates LAPACK. http://www.mathworks.com, 2000.

[41] Cleve Moler. The origins of Matlab, December 2004. http://www.mathworks.fr.

[42] netlib.org. Benchmark programs and reports. http://www.netlib.org/benchmark/.

[43] Mathieu Philippe, Djalel Abdemouche, Fabrice Leray, Jean-Baptiste Silvy, and Pierre Lando. modules/graphics/includes/ObjectStructure.h. http://gitweb.scilab.org.

[44] Bruno Pinçon. Quelques tests de rapidité entre différents logiciels matriciels. http://wiki.scilab.org/Emulate_Object_Oriented_in_Scilab, 2008.

[45] pqnelson. Object oriented C. http://pqnelson.blogspot.com/2007/09/object-oriented-c.html.

[46] W. H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, Second Edition*. Cambridge University Press, 1992.

[47] Axel-Tobias Schreiner. Object-oriented programming with ANSI C. www.cs.rit.edu/~ats/books/ooc.pdf.

[48] Debian Science. README.Debian. http://anonscm.debian.org/viewvc/debian-science/packages/atlas/trunk/debian, 2010.

[49] Enrico Segre. Scilab function variables: representation, manipulation. http://wiki.scilab.org, 2007.

[50] Enrico Segre. Scoping of variables in Scilab. http://wiki.scilab.org/howto/global_and_local_variables, 2007.

[51] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[52] R. Clint Whaley, Antoine Petitet, R. Clint, Whaley Antoine, Petitet Jack, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project, 2000.

[53] Wikipedia. Intel Xeon — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Xeon, 2010.

[54] James Hardy Wilkinson. *Rounding errors in algebraic processes*. Prentice Hall, 1963.