so/\/ata

Программа для математических расчётов

Станислав Михель

Предисловие

so/\/ata - консольная программа для математических расчётов и моделирования, написанная на языке программирования **Lua**. Она состоит из библиотеки математических модулей *matlib*, которая может использоваться самостоятельно, а также REPL интерпретатора.

Первоначально, данная программа возникла из потребности Автора в консольном калькуляторе. Были опробованы некоторые из доступных в **Linux** специализированных программ, но они не удовлетворили ожиданий либо из-за специфического синтаксиса, либо из-за ограниченных возможностей. Сравнение интерпретаторов языков программирования **Python** и **Lua** показало, что последний более эффективно выполняет некоторые вычисления (без использования специализированных библиотек), а также обладает меньшей задержкой при запуске, поэтому выбор пал на него. Вследствие интереса Автора к различным приложениям математики функционал программы увеличился и со временем перерос возможности калькулятора, в том числе, хочется верить, и с точки зрения удобства.

Использование чистого **Lua** без сторонних зависимостей для so/\/ata имеет как свои плюсы, так и минусы. К последним относятся:

- быстродействие определяется эффективностью интерпретации кода
- возможности программы ограничены стандартной библиотекой **Lua**
- однопоточное исполнение кода
- интерфейс - командная строка
- отсутствие графических инструментов

Из плюсов можно выделить:

- готовность к использованию сразу просле скачивания
- кросплатформенность
- простота расширения и кастомизации

С учётом перечисленных ограничений, so/\/ata пытается быть, прежде всего, подручным инструментом для выполнения простых расчётов и проверки гипотез, по результатам которых могут быть задействованы более мощные программы, такие как **Matlab**.

Оглавление

Глава 1

Начало работы

1.1 Установка

Для работы с so/\/ata необходимо, чтобы был установлен интерпретатор **Lua**. Для работы подойдёт **Lua 5.1** и выше, однако, последние версии будут работать более эффективно. Также должна быть настроена переменная окружения *PATH* (см. приложение А).

so/\/ata не требует дополнительной установки, достаточно скачать последнюю релизную версию с Github и распаковать в удобном для вас месте.

1.2 Быстрый старт

Если у вас уже есть опыт использования **Lua**, **Python** или подобных языков программирования, интерпретатор которых запускается из консоли, то можно ускорить знакомство с so/\/ata. Для этого перейдите в папку программы и выполните следующие команды

```
lua sonata.lua notes/intro_ru.note   # основы ‘ работы
lua sonata.lua notes/modules.note    # описание ‘ модулей
```

1.3 Вызов программы

Для обращения к программе необходимо открыть терминал и перейти в папку с программой (далее будет показано, как настроить вызов программы из произвольного места). После выполнения команды

```
lua sonata.lua
```

должно отобразиться приглашение к работе:

```
  # #       --=====   so/\/ata   =====--       # #
   # #          --===== 0.9.38 ====--         # #

------- help([function]) = get help -------------
---------- use([module]) = expand functionality -
----------------- quit() = exit -----------------

## _
```

Возможны три способа работы с программой: вызов программы в интерпретаторе so/\/ata, вызов в стандартном интерпретаторе **Lua** и самостоятельное использование математической библиотеки.

1.3.1 Интерпретатор so/\/ata

Это режим по умолчанию, который запускается командой

```
lua sonata.lua
```

Интерпретатор написан на **Lua**, поэтому сам "интерпретируется" в процессе работы, но при этом расширяет функциональные возможности программы.

Преимущества:

- дополнительные возможности программы (командный режим, использование цветов, логирование)
- одинаковый интерфейс взаимодействия во всех версиях **Lua 5.x**

Недостатки:

- скорость обработки пользовательских команд чуть ниже, чем в интерпретаторе **Lua**
- нужно обозначать перенос строки знаком '\'

1.3.2 Интерпретатор Lua

Для использования стандартного интерпретатора **Lua** выполните команду

```
lua -i sonata.lua
```

Произойдёт загрузка необходимых библиотек, после чего so/\/ata передаст управление стандартной программе. При этом описанные в предыдущем пункте преимущества и недостатки поменяются местами.

1.3.3 Библиотека *matlib*

Математическая библиотека so/\/ata может быть использована самостоятельно. Пользователь может выбрать необходимые модули на своё усмотрение. Однако, следует помнить, что модули

могут зависеть друг от друга, поэтому перенос отдельных файлов в другой проект может привести к проблемам.

1.4 Аргументы запуска

Вызов

```
lua sonata.lua -h
```

позволяет увидеть список аргументов, с которыми программа может быть запущена. Во-первых, это набор флагов, одним из которых (-h) мы сейчас воспользовались для получения справки. Рассмотрим остальные.

- **-e expr**
  позволяет выполнить последующую команду и отобразить результат.

  ```
  lua sonata.lua -e "1 + 2 + 3 + 4"
  ```

- **–doc [lang]**
  генерирует *html* страницу с описанием текущей версии программы, файл сохраняется в текущу директорию в терминале. Если есть файлы локализации, можно дополнительно указать имя файла и получить транслированный текст.

  ```
  lua sonata.lua --doc      # английскийтекст
  lua sonata.lua --doc ru   # русскийтекст
  ```

- **–lang [name]**
  формирует файл локализаци в папке *sonata/locale*, аргументом является наименование языка, подробности см. далее. При отсутствии аргумента будет отображена текущая настройка языка.

  ```
  lua sonata.lua --lang      # отображаеттекущийязык
  lua sonata.lua --lang eo   # генерируетобновляет/ файллокализации
  ```

- **–new name alias [description]**
  формирует шаблон для новой математической библиотеки, аргументами являются полное имя модуля (название генерируемого файла в *sonata/matlib*), его краткий псевдоним и (опционально) описание. Подробности см. далее.

  ```
  lua sonata.lua matrices Mat "Matrix operations"
  ```

- **–test [name]**
  запускает модульные тесты библиотеки *matlib*, опциональный агрумент - название тестируемого модуля.

  ```
  lua sonata.lua --test           # протестироватьвсебиблиотеки
  lua sonata.lua --test complex   # протестироватьбиблиотекукомплексныхчисел
  ```

Аргументами *sonata.lua* могут быть один или несколько *.lua* или *.note* файлов. В этом случае, программа выполнит их последовательно, после чего завершит работу.

```
lua sonata.lua fileA.lua fileB.note
```

1.5 Основы работы

1.5.1 Синтаксис Lua

Основными типами данных для so/\/ata являются числа, строки и таблицы. Числа обычно делят на целые и с плавающей запятой, но **Lua** все числа рассматривает и обрабатывает как "действительные", т.е. результатом операции "1/2" будет "0.5" (для целочисленного деления в последних версиях **Lua** введён оператор //). Строкой является последовательность символов, заключённая в одинарные или двойные кавычки. Если строка длинная (содержит переносы), вместо кавычек используются знаки [[и]]. Строчные комментарии начинаются с символа –.

Таблица - ключевой элемент **Lua** который представляет собой контейнер, содержащий другие объекты. Элементы таблицы могут быть упорядочены как по ключу (словари), так и по индексу (списки), если ключ не был указан. Индексация начинается с 1. Чтобы получить элемент, необходимо написать его индекс/ключ в квадратных скобках. Если ключ является строкой, достаточно указать его через точку после имени таблицы.

```
a = {10, 20, 30}      -- упорядоченыпоиндексу
a[1]                  -- 10, первыйэлементтаблицы
b = {x=10, y=20, z=30}  -- упорядоченыпоключу
b.x                   -- 10, элементсиндексом x
c = {x=10, 20, 30}    -- возможнакомбинация
c.x                   -- 10
c["x"]                -- 10, эквивалентно c.x
c[1]                  -- 20
```

Словари в so/\/ata используются, чаще всего, для описания параметров, а списки - для представления массивов.

Функции это ещё один ключевой компонент **Lua**. Вызов функций выглядит точно так же, как в других языках программирования: имя функции и список аргументов, перечисленных через запятую и заключённых в скобки. Число возвращаемых элементов может быть больше одного. Если функция принимает единственный аргумент, который является строкой или таблицей, то скобки можно опустить.

```
print(1, 2, 3)   -- напечатает 1 2 3
print('abc')     -- напечатает abc
```

```
print 'abc'      -- длястрокискобкинеобязательны
```

Пример объявление функции преставлен в следующем листинге. Определение находится между ключевыми словами **function** и **end**, если требуется вернуть результат, он указывается после **return**.

```
function foo(x, y)
  local x2, y2 = x*x, y*y
  return x2+y2, x2-y2
end
a, b = foo(3, 4)
```

В **Lua** все переменные по-умолчанию являются глобальными, т.е. доступны для изменения и чтения во всём коде. Чтобы ограничить область видимости, используют ключевое слово **local**. Объявление локальных переменных не обязательно, но позволяет избежать многих неочевидных ошибок.

Поскольку функция является также и типом данных, с ней можно обращаться как с другими переменными, в том числе, хранить в таблице. Если функция должна обработать данные из той же таблицы, в которой лежит она сама, предусмотрен оператор ':', позволяющий неявно передать таблицу в качестве первого аргумента.

```
t = {x = 0}
t.foo = function (var) var.x = 1 end  -- изменяемполевтаблице
t.foo(t)    -- явныйвызов
t:foo()     -- неявныйвызов
```

Следует сказать также несколько слов о стандартной библиотеке математических функций **Lua** с названием *math*. Данная библиотека является обёрткой над стандартными стандартными математическими функциями из **C**. Её состав в различных версиях языка может отличаться, но в обычно включает в себя стандартные функции типа **exp**(), **sqrt**(), **log**(), тригонометрические функции, генератор случайных чисел, константы **pi**, **huge** и ряд вспомогательных функций. Если для вас важно быстродействие и вы работаете с обыкновенными числами, используйте функции стандартной библиотеки, если же вам нужна гибкость, используйте переопределённые функции из модуля *main*.

Если вы хотите загрузить **Lua** файл из кода или интерпретатора, используйте функцию **dofile**(*имя_файла*). Также в работе могут пригодиться функции **tonumber**(), которая пытается конвертировать строку в число, и **tostring**(), которая возвращает строковое представление для заданного объекта.

Узнать больше о возможностях языка **Lua** можно из официального описания manual.html, на русском - руководство.html. Желающим углубиться в язык можно порекомендовать Программирование на языке Lua, однако нужно иметь ввиду последующее развите и изменение **Lua**.

1.5.2 Функции so/\/ata

После запуска интерпретатора so/\/ata вы оказываетесь в интерактивном режиме работы с программой, который не сильно отличается от работы в интерпретаторе **Lua**: на каждую введённую команду программа возвращает результат (если он не *nil*) и переходит к ожиданию следующей команды. Например, если ввести выражение “1 + 2 + 3” и нажать *Enter*, программа напечатает результат сложения.

Для обозначения вводимых пользователем команд далее будет использоваться знак ##. Интерпретатор so/\/ata относительно прост, он умеет обрабатывать только текущую вводимую строку. Поэтому для продолжения ввода длинного выражения необходимо поставить знак ‘\’ перед переходом на новую строку. Это знак не является стандартным для **Lua**, поэтому можно указывать его в комментарии воизбежание конфликтов.

```
## 1 + 2 + \
## 3 + 4
-- или
## 1 + 2 + --\
## 3 + 4
```

При запуске программы загружается модуль *Main*, который содержит ряд стандартных математических функций, а также некоторые дополнительные процедуры. Получить список доступных в данный момент функций позволяет вызов

```
## help()
```

Функции в *Main* можно разбить на 2 категории. К первой относятся широко используемые математические функции, такие как синус или логарифм. Все они переопределены для работы с типами данных, используемыми в so/\/ata, например, с комплексными или рациональными числами. Получить дополнительную информацию о конкретной функции можно с помощью выражения **help**(*функция*), например,

```
## help(sin)
```

Вторая категория включает в себя вспомогательные (auxiliary) функции so/\/ata. С одной из них, **help**(), мы уже познакомились. Если её аргументом является другая функция, и для неё имеется справочная информация, то эта справка выводится на печать. Аргументом может быть строка с именем модуля, например,

```
## help 'Main'
```

также отображает содержимое указанного модуля. Вызов

```
## help '*'
```

выводит информацию о функциях для всех загруженных на данный момент модулей. Аргументом функции может быть и произвольный объект *Lua*/so/\/ata, в этом случае будут выведены его тип и значение (если возможно).

Загрузить новые модули позволяет функция **use**(). При вызове без аргументов на экран будет выведен список доступных библиотек и их статус (загружена или нет).

```
## use()
MODULE        ALIAS      USED
...
main          Main        ++
...
polynomial    Poly
```

Можно загрузить один или несколько модулей за раз.

```
## use 'data'  -- загрузитьодинмодуль
## use {'complex', 'matrix'}  -- загрузитьперечисленныемодули
## use '*'     -- загрузитьвсемодули
```

Ссылка на модуль сохраняется в переменную с определённым именем (alias), которое обычно короче по длине и начинается с заглавной буквы. Алиасы также можно использовать в функции **use**(), т.е. допустимо писать **use**('C') вместо **use**('complex'). Для загрузки можно использовать стандартную фукнцию **require**('matlib.*имя_модуля*'), однако, **use**() более гибко работает с аргументами и дополнительно собирает справочную информацию о функциях.

В качестве примера рассмотрим модуль *matrix*, он может быть загружен через алиас командой

```
## use 'Mat'
```

Для получения информации о модуле выполните

```
## help 'Mat'    -- списокфункциймодуля
## help(Mat.T)   -- справкаоконкретнойфункции
```

Следует обратить внимание, что все функции модулей вызываются через ':', это сделано для того, чтобы избежать неопределённости относительно используемого знака (точка или двоеточие). Тем не менее, при работе с **help**() используется точечная нотация.

В некоторых случаях вызов функции модуля через двоеточие может вызывать неудобство, например, если мы хотим передать её в качестве аргумента другой функции. Сделать "обёртку" для упрощения вызова позволяет функция **Bind**(*объект*, *метод*):

```
## eye = Bind(Mat, 'eye')
## eye(3)  -- эквивалентно Mat:eye(3)
```

Часто работа происходит со списками или другими упорядоченными коллекциями, при этом необходимо получить новый список путём применения какой-либо операции к элементам исходного списка. В so/\/ata для этого определена функция **Map**(*функция*, *список*):

```
## lst = Map(exp, {0, 1, 2})
## print(lst)
{ 1.0, 2.7182, 7.3890, }
```

В данном примере строится список чисел e^n для n от 0 до 2. Функция **print**() в интерпретаторе so/\/ata перегружена и умеет печатать списки. **Map**() может использовать и для некоторых типов данных, таких как матрицы. Например, построить случайную матрицу можно следующим способом:

```
## function rnd() return math.random() end
## Map(rnd, Mat:zeros(2, 2))
```

Для завершения работы программы выполните

```
## quit()
```

1.6 note файлы

Файлы с расширением *note* обрабатываются в so/\/ata так же, как если бы это были команды, вводимые пользователем вручную. То есть файл интерпретируется последовательно, строка за строкой, если выражение возвращает отличное от **nil** значение, оно выводится на печать, перенос длинных строк осуществляется знаком '\'.

Для удобства работы *note* файл может быть разделён на блоки, разделителем является строка "– PAUSE", при достижении которой интерпретатор переходит в интерактивный режим. В данном режиме интерпретация исходного файла приостанавливается, и so/\/ata выполняет команды пользователя, пока они отличаются от пустой строки. После этого начинается интерпретация следующего по списку блока.

Строчные комментарии рассматриваются как часть общего описания и выводятся на печать. Если нужно исключить какой-то фрагмент файла из обработки, используйте многострочные комментарии. Если в комментарии текст отделён от символов "- -" с помощью табуляции, он будет выделен по цвету при включенной опции работы с цветами. В общем случае, заголовком блока считатется его первая строка.

1.7 Команды интерпретатора so/\/ata

Интерпретатора so/\/ata умеет не только работать со стандартными выражениями **Lua**, но также предоставляет "командный" режим работы, т.е. содержит ряд дополнительных команд,

управляющих его работой. Данные команды начинаются со знака ':' (по аналогии с **Vim**).

Общие команды

- **:help [name]**
  Отображает список команд или выводит информацию о команде.
- **:q**
  Завершение работы и выход из программы. Эквивалентно вызову **quit**().
- **:log on/off**
  Включение/выключение логирования команд пользователя. Результат сохраняется в *note*-файл, поэтому может быть использован для повторного исполнения команд в будущем.

Работа с *note*-файлами

- **:o name**
  Открывает файл и загружает данные. Новые блоки команд добавляются в конец к загруженным ранее.
- **:ls**
  Отображает список блоков команд.
- **:rm**
  Очищает список блоков.
- :1, 2, 7:9, -3
  Выполяет команды из указанных блоков. Знак ':' разделяет начало и конец диапазона, отрицательные индексы отсчитываются с конца.

Средства отладки

- **:time func**
  Вычисляет среднее время работы функции в миллисекундах.

```
## function sumAB(a,b) return sin(a)*cos(b) + cos(a)*sin(b) end
## :time function () sumAB(pi/3, pi/4) end
```

- **:trace func**
  Выводит список функций, вызываемых при вычислении заданного выражения, и число вызовов.

```
## :trace function () sumAB(pi/3, pi/4) end
```

1.8 Настройка программы

До сих пор мы использовали so/\/ata с настройками по умолчанию. В этом разделе будет показано, как поменять локализацию, путь доступа к файлам, а также другие параметры программы.

Конфигурация программы прописана в файле *sonata.lua*. Вы можете менять настройки файла внутри папки *sonata*, либо положить копию файла в удобное для доступа место и работать с ней.

Настраивать можно 2 раздела: *CONFIGURATION* и *MODULES*. Начнём со второго раздела, здесь представлены модули, которые программа умеет загружать через функцию **use**(). Здесь можно добавить или удалить (закомментировать) какой-либо модуль. Если же вам не нравятся предложенные сокращения имён, можно определить собственные алиасы, например, переименовать “Z” в “Cx” или “Mat” в “M”.

Раздел *CONFIGURATION* содержит переменные с параметрами программы. Чтобы выполнить настройку, необходимо раскомментировать соответствующий параметр и установить требуемое значение.

- SONATA_ADD_PATH
  Путь к модулям программы. Если вы хотите запускать $\mathrm{so/\backslash/ata}$ из произвольного места, укажите в данной переменной абсолютный путь к папке *sonata*. В Unix системах можно также настроить команду запуска программы, прописав в *bashrc* алиас для пути к файлу *sonata.lua*.
- SONATA_USE_COLOR
  Выделение цветом строки приглашения, сообщения об ошибках, справки и т.д. Данная опция зависит от операционной системы, но в Unix, как правило, работает нормально.
- SONATA_DEFAULT_MODULES
  Определяет модули, загружаемые при запуске программы. Если вы часто работаете с какими-либо модулями, укажите их здесь в виде списка.
- SONATA_ASCIIPLOT_UNICODE
  Данная опция повзоляет модулю *asciiplot* использовать Unicode символы для улучшения внешнего вида диаграмм.
- SONATA_PROTECT_ALIAS
  Если данная опция включена, $\mathrm{so/\backslash/ata}$ не позволит создавать переменные, имена которых совпадают с алиасами модулей. Чем короче длина алиаса, тем больше вероятность его случайно “затереть”.
- SONATA_LOCALIZATION Содержит имя файла локализации. В частности, для включения русского языка укажите здесь “ru.lua”. В ОС Windows для корректного отображения нелатинских букв может дополнительно потребоваться включение Unicode с помощью опции “chcp 65001”.

Глава 2

data: обработка списков данных

2.1 Описание

Модуль *data* содержит методы для обработки списков (таблиц) данных, прежде всего, чисел. Часть функций позволяют работать со списком списков, т.е. с двумерным массивом. Определены классы, упрощающие генерацию последовательностей и доступ к элементам массивов. В данный модуль также включены функции для статистической обработки данных.

2.2 Основные операции

В данном разделе будут описаны функции, позволяющие получить новый список из исходного. Данная задача решается либо созданием объекта, который имитирует свойства таблицы, либо формированием непосредственно новых таблиц.

2.2.1 Объекты

Функция **range**(*начало, конец, [шаг]*) позволяет получить объект, который ведёт себя как таблица значений в заданном интервале, шаг является опциональным и равен 1 по умолчанию. Если значение для конца больше, чем для начала, список будет построен в обратном порядке. При этом сам список физически не создаётся, его элементы рассчитываются во время обращения к соответствующему индексу.

```
## rng = D:range(-10, 10)
## #rng
21
## for i, v in ipairs(rng) do print(i, v) end
```

```
1    -10
...
21    10
```

С объектом *range* можно выполнять линейные преобразования, такие как сложение с числом и умножение на число.

```
## rng2 = 2*rng + 1
## rng2
{-19, -17 .. 21}
```

Также можно применить функцию ко всем элементам списка с помощью **map**(). Например, получить значение синуса на заданном интервале можно следующим способом

```
## rng3 = D:range(-3, 3, 0.3):map(math.sin)
## rng3[21]
0.14112
```

Допускается последовательный вызов **map**(), однако следует помнить, что расчёт будет выполнен в момент обращения к интексу объекта, т.е. он ведёт себя “лениво”.

Объект **range** создаёт неизменяемый список, если же необходима обычная таблица, пригодная для записи, можно получить её с помощью функции **copy**(*таблица*), которая создаёт глубокую копию для заданного списка. Например, следующий код позволяет сгенерировать таблицу, из заданного числа нулей.

```
## zeros = D:copy(D:range(1,8) * 0)
## print(zeros)
{ 0, 0, 0, 0, 0, 0, 0, 0, }
```

В некоторых случаях может быть полезен объект so/\/ata, возвращающий ссылку на определённый диапазон данных другой таблицы. Он формируется функцией **ref**(*таблица, [начало, конец]*). Первым и последним индексами по умолчанию являются начало и конец таблицы.

```
## src = {0, -1, -2, -3, -4, -5}
## ref = D:ref(src, 2, 4)
## #ref
3
## for i, v in ipairs(ref) do print(i, v) end
1    -1
2    -2
3    -3
```

Данный объект позволяет менять данные в исходной таблице, как по отдельности, так и в диапазоне значений. Чтобы заменить некоторое подмножество списка, нужно приравнять его другой ссылке.

```
## ref[1] = 10
## ref[2] = D:ref {20, 30}
## print(src)
{ 0, 10, 20, 30, -4, -5, }
```

Следующий объект предназначен для работы с двумерными таблицами. Он представляет собой ссылку на массив, где строки и столбцы поменяли местами, т.е. “транспонировали”, и формируется функцией **T**(*таблица*).

```
## a = {     \
.. {1,2,3}, \
.. {4,5,6}, \
.. {7,8,9}}
## at = D:T(a)
## at[1][2] == a[2][1]
true
```

Данные исходной таблицы не копируются, что может быть удобно при работе с большими массивами. При этом ссылка доступна как для чтения, так и для записи.

Поскольку рассмотренные объекты ведут себя как таблицы (возвращают размер, осуществляют доступ по индексу), их можно использовать совместно, в частности, **ref**() можно вызывать и для **range**(), и для строки транспонированной таблицы.

2.2.2 Фильтрация и преобразование данных

so/\/ata позволяет применить некоторый критерий ко всем элементам списка, а также выполнить фильтрацию на его основе. Критерием является функция, которая принимает на вход элемент списка, выполняет проверку и возвращает true или false. Эта функция может быть определена стандартным способом, либо с помощью метода **Fn**(*описание, число аргументов*), который генерирует функцию с заданным числом аргументов на основе строки описания. Аргументы должны иметь имена *x1, x2* и т.д. Если число аргументов не превышает двух, второй параметр можно опустить.

```
## sum = D:Fn("x1 + x2", 2)
-- эквивалентно
-- sum = function (x1, x2) return x1 + x2 end
## sum(1, 2)
```

```
3
## odd = D:Fn "x1 % 2 ~= 0"
## odd(3)
true
```

Данный способ упрощает запись, но если важно быстродействие, лучше использовать стандартное определение функции.

Функция **is(*список, условие*)** возвращает список весов: 1 если элемент удовлетворяет критерию и 0 в противном случае (**isNot(*список, условие*)** наоборот). Условие может быть как функцией, так и строкой, по аналогии с **Fn**().

```
## a = {1, 2, 3, 4}
## print(D:is(a, odd))
{ 1, 0, 1, 0, }
## print(D:isNot(a, "x1 % 2 ~= 0"))
{ 0, 1, 0, 1, }
```

Поскльку результат может интерпретироваться как “веса”, его можно использовать в некоторых статистических функциях модуля *data*, а также для фильтрации данных.

Выделить из списка требуемые элементы позволяет функция **filter**(*список, условие*). Критерий отбора может быть задан списком весов равного размера (нули отбрасываются, остальное сохраняется), функцией, возвращающей true или false, а также строкой, как в предыдущих случаях.

```
## b = D:filter(a, "x1 > 2")
## print(b)
{ 3, 4, }
## print(D:filter(a, D:is(a, odd)))
{ 1, 3, }
```

Если **Map**() из модуля *main* применяет функцию к одному списку, то её обобщённой версией является **zip**(*функция, список1, список2, ...*), которая применяет функцию нескольких аргументов поэлементно к каждому из входных списков и формирует список результатов. Как обычно, применяемая операция может быть задана в виде **Lua** функции или строки.

```
## b = D:zip(function (x) return 2*x end, a)
## zip = Bind(D, 'zip')
## c = zip("{x1-x2, x1+x2}", a, b)
{ -4, 12, }
```

При инициализации функции с помощью строки индексация переменных *x* должна соответствовать числу списков.

Ещё одна операция, позаимствованная из функционального программирования наряду с *map* и *filter*, это функция **reduce**(*функция, список, [x0]*). Она последовательно применяет функция двух аргументов к каждому элементу списка и результату предыдущей операции, и возвращает найденный результат. Как и раньше, функция может быть задана с помощью строки, если быстродействие не требуется. Если начальное значение *x0* не указано, используется первый элемент списка.

```
## D:reduce("x1*x2", a)
24
```

2.3 Чтение, сохранение и печать

Модуль *data* позволяет работать с *csv*-файлами, а именно, считывать данные в **Lua** таблицу и сохранять двумерные таблицы в файл. Чтение осуществляется функцией **csvread**(*файл, разделитель*), причём разделителем может быть не только запятая (значение по умолчанию), но и любой другой одиночный символ. Элемент преобразовывается в число, если возможно. Обратную операцию, запись, осуществляет функция **csvwrite**(*файл, таблица, разделитель*).

```
## a = {       \
.. {1, 2, 3}, \
.. {4, 5, 6}, \
.. {7, 8, 9}}
## csvwrite('data.csv', a)
## b = csvread('data.csv')
## b[2][2]
5
```

Отобразить элементы списка можно разными способами. Самый универсальный - обход в цикле и вывод на печать поэлементно. Для одномерных списков подойдёт перегруженная в so/\/ata функция **print**. Для визуализации двумертных списков предусмотрена функция **md**(*список, [заголовки, функция]*), которая выравнивает стролбцы и использует *Markdown* формат для таблиц. Можно опционально задать таблицу заголовков. Если требуется распечатать отдельные столбцы или результат применения некоторой операции, можно указать функцию, которая преобразует каждую строку в заданный вид.

```
## D:md(a, {'c1', 'c2', 'c3'})
| c1 | c2 | c3 |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
```

```
| 7  | 8  | 9  |
## D:md(a, nil, \
..    function (v) return {v[1]+v[2], v[1]-v[3]} end)
|----|----|
| 3  | -2 |
| 9  | -2 |
| 15 | -2 |
```

Сочетание **csvread**() и **md**() может быть использовано для того, чтобы прочитать некоторый файл, выполнить операции над строками и вывести результат в табличном виде.

2.4 Элементы статистики

В данном разделе будут описаны функции, которые принимают на вход массив данных и возвращают, как правило, некоторую его характеристику. **max**(*список*) и **min**(*список*) возвращают как максимальный/минимальный элемент списка, а также его индекс. Медиану вычисляет **median**(*список*), сумму - **sum**(*список*).

```
## a = {2, 0, 1, -3, 4, 1, 2, 1}
## v, ind = D:min(a)
## a[ind] == v
true
## D:max(a)
4
## D:median(a)
1
## D:sum(a)
8
```

Ряд функций принимают в качестве опционального аргумента список весов. Если таблица весов задана, а значение для некоторого элемента отсутствует, оно принимается равным 1. Таким образом, пустая таблица весов означает учёт всех элементов списка, а для исключения отдельных значений достаточно установить нули в соответствующих индексах. С весами работают функции вычисления среднего (**mean**(*список, вес*), **geomean**(*список, вес*), **harmmean**(*список, вес*)) и среднеквадратичного отклонения **std**(*список, вес*). Расчёт момента **moment**(*степень, список, вес*) также может быть выполнен с учётом весов элементов.

```
## D:mean(a)
1
## w = {[2]=0, [4]=0}    -- skip 0, -3
```

```
## D:mean(a, w)
1.8333333333333
## D:geomean(a, w)
1.5874010519682
## D:harmmean(a, w)
1.4117647058824
## D:moment(2, a) - D:std(a)^2
0
```

Связь двух списков можно оценить с помощью функции ковариации **cov2**(*список1, список2*). Матрицу ковариации для произвольного числа списков можно получить с помощью функции **cov**(*таблица*), аргументом которой является таблица из исходных списков.

Анализ частот элементов выполняет функция **freq**(*список*), которая возвращает словарь, ключи которого - элементы исходного списка, а значения - соответствующие частоты. Для подсчёта распределения элементов по диапазонам используется функция **histconuts**(*список, границы*). Границы могут быть заданы таблицей возрастающих чисел, либо числом интервалов, равным по умолчанию 10. Функция возвращает список суммарных значений элементов в каждом диапазоне, а также список границ.

```
## b = D:freq(a)
## b[1]
3
## sum, rng = D:histcounts(a, 3)
## print(sum)
{ 1, 6, 1, }
## print(rng)
{ -1.25, 2.25, }
```

Глава 3

asciiplot: визуализация данных псевдо-графикой

Глава 4

matrix: операции с матрицами

Глава 5

complex: комплексные числа

Глава 6

polynomial: полиномы

Глава 7

numeric: численные методы решения задач

Глава 8

random: случайные числа

Глава 9

bigint: целые числа произвольной длины

Глава 10

rational: рациональные числа

Глава 11

quaternion: операции с кватернионами

Глава 12

special: функции мат-физики

Глава 13

graph: операции с графами

Глава 14

units: единицы измерения и преобразования

Глава 15

const: коллекция констант

Глава 16

gnuplot: построение графиков в GNUplot

Глава 17

symbolic: элементы компьютерной алгебры

Глава 18

qubit: эмуляция квантовых вычислений

Глава 19

lens: оптика параксиальных лучей

Глава 20

geodesy: преобразования координат