so/\/ata

Программа для математических расчётов

Оглавление

Предисловие

so/\/ata - консольная программа для математических расчётов и моделирования, написанная на языке программирования Lua. Она состоит из библиотеки математических модулей *matlib* (может использоваться независимо), 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 после скачивания
- кроссплатформенность
- 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.

I 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 A).

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 явно обозначать перенос строк

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 актуальной 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      # английский текст
lua sonata.lua --doc ru   # русский  текст
```

- **--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     # отображает текущий язык
lua sonata.lua --lang eo  # генерирует/обновляет файл локализации
```

- **--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 аргумент - название тестируемого модуля.

```
lua sonata.lua --test           # протестировать все библиотеки
lua sonata.lua --test complex  # протестировать библиотеку комплексных чисел
```

Аргументами so/\/ata 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 в 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}   -- упорядочены по индексу
a[1]               -- 10, первый элемент таблицы
b = {x=10, y=20, z=30}  -- упорядочены по ключу
b.x                -- 10, элемент с индексом x
c = {x=10, 20, 30}  -- возможна комбинация
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 круглые 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'  -- для строки скобки не обязательны
```

Пример объявление функции представлен 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 “:” (двоеточие), позволяющий неявно передать таблицу в качестве первого аргумента.

```
t = {x = 0}
t.foo = function (var) var.x = 1 end  -- изменяем поле в таблице
t.foo(t)   -- явный вызов
t:foo()     -- неявный вызов
```

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** (бесконечность) 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 необходимо 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 в виду последующее развитие 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 “6” и сохранит результат в переменной ANS.

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 во избежание конфликтов.

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

Если строк больше 2, можно указать первую и последнюю строки знаками “\&”.

```
## 1 + \&
.. 2 +
.. 3 +
.. 4  \&
10
```

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 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'
```

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 или 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     ++
...
```

Можно загрузить один или несколько модулей за раз.

```
## use 'data'  -- загрузить один модуль
## use {'complex', 'matrix'}  -- загрузить перечисленные модули
## use '*'  -- загрузить все модули
```

В настоящее время использовать явно функцию **use**() необязательно, модуль будет загружен при первом вызове соответствующего ему конструктора.

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**('Z') вместо **use**('*complex*'). Для загрузки можно использовать стандартную функцию **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'   -- список функций модуля
## help(Mat.T)  -- справка о конкретной функции
```

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})
## lst
{ 1.0, 2.7182, 7.3890, }
```

В данном примере строится список чисел e^n для n от 0 до 2. Вывод значения в консоль 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 знаками “\” и “\&”.

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 считается 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**
- Выполняет 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 so/\/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 позволяет модулю *asciiplot* использовать Unicode символы для улучшения внешнего вида диаграмм.
- SONATA_PROTECT_ALIAS
  Если данная опция включена, so/\/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”.

II *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.

```
## use 'data'
## 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 индексу 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)
## 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}
## 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}
## D:is(a, odd)
{ 1, 0, 1, 0, }
## D:isNot(a, "x1 % 2 ~= 0")
{ 0, 1, 0, 1, }
```

Поскольку 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")
## b
{ 3, 4, }
## 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 преобразуется 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 визуализации двумерных списков предусмотрена функция **md**(*список*, *[заголовки*, *функция]*), которая выравнивает столбцы и использует Markdown формат для таблиц. Можно опционально задать список 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 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 также может быть выполнен с учётом весов элементов функцией **moment**(*степень*, *список*, *вес*).

```
## 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 - элементы исходного списка, 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)
## sum
{ 1, 6, 1, }
## rng
{ -1.25, 2.25, }
```

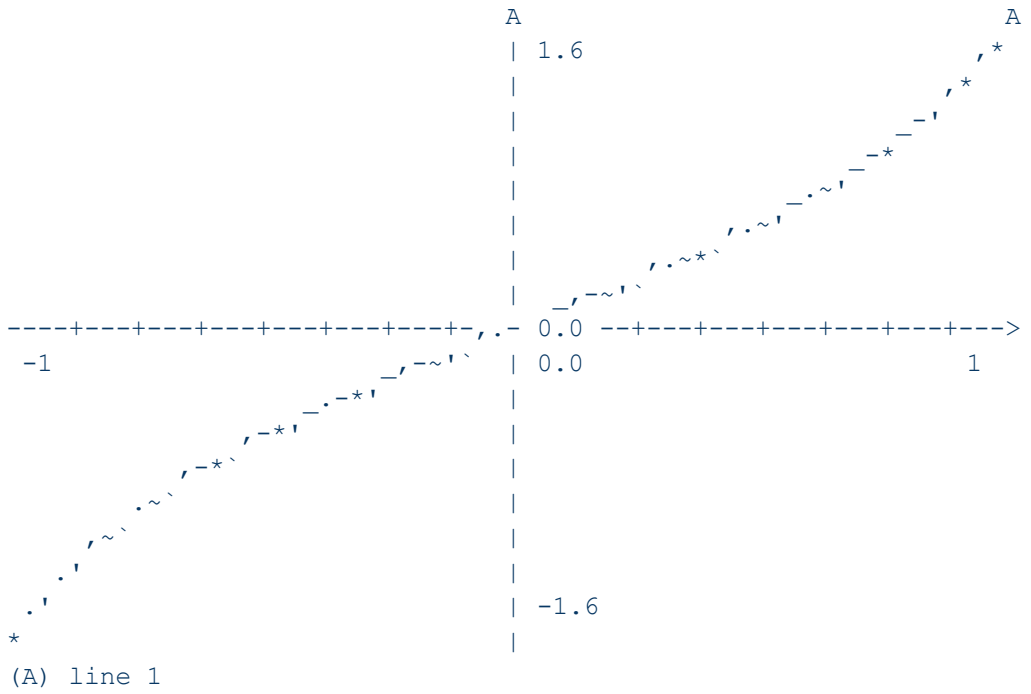
III *asciiplot*: визуализация в консоли

3.1 Введение

Визуализация данных играет важную роль ... С учётом ограниченных возможностей чистого Lua единственной возможностью для визуализации является использование текстовых символов, то есть псевдо-графика. Модуль *asciiplot* предоставляет набор функций для создания таких диаграмм. Безусловно, псевдо-графика накладывает свои ограничения и на разрешение “изображения”, и на форму представления данных. Однако, во многих случаях этого может быть достаточно для отрисовки поведения функции на заданном интервале. Для лучшего визуального восприятия можно использовать цветной вывод и символы Unicode (см. пункт о настройке so/\/ata).

В качестве примера, изобразим график функции **tan**(). Для этого необходимо создать объект *asciiplot* с помощью функции **Ap**(), затем вызвать для него метод **plot**() и вывести на печать полученный результат.

```
## use 'asciiplot'
## fig = Ap()
## fig:plot(math.tan)
## fig
```



3.2 Параметры холста

Под холстом будем понимать массив строк, который выводится на экран при печати объекта *asciiplot*. При этом могут быть настроены размер изображения, диапазон значений, а также положение и тип осей.

3.2.1 Размер изображения

При вызове функции **Ap**() без аргументов используются значения по умолчанию, которые хранятся в переменных *WIDTH*, *HEIGHT*. В общем виде конструктор имеет вид **Ap**(*ширина*, *высота*), где в скобках указывается желаемое число символов. На размеры накладываются следующие требования: число должно быть нечётным и не меньше некоторой минимальной величины (на данный момент это 9). Если будет указано некорректное значение, so/\/ata выведет предупреждение и сама установит ближайшее валидное число. Пропорционально увеличить или уменьшить холст позволяет функция **scale**(*коэффициент*), которая домножает ширину и высоту на заданное число. Если же аргументом является другой холст, то функция устанавливает размеры равными заданному.

```
## fig1 = Ap(65, 19)  -- явно заданный размер
## fig2 = Ap()
## fig2:scale(fig1)   -- установить равный размер
```

3.2.2 Настройка осей

Для настройки каждой из осей (X - горизонтальная, Y - вертикальная, Z - для контуров) используется своя функция **set***(*параметры*), где аргументом является таблица с полями

- range - диапазон значений, определяемый списком из двух чисел, используется для построения функций. По умолчанию равен {-1, 1}.
- fix - булевый флаг, обозначающий возможность изменения заданного диапазона при построении графика. Например, если для оси Y флаг *fix=false*, то диапазон значений будет масштабирован, чтобы уместить все точки на заданном интервале X, иначе часть значений будут отброшены.
- view - строка {'*min*', '*mid*', '*max*'} с указанием, в какой части холста рисовать ось (слева направо, снизу вверх).
- log - булевый флаг, равный *true*, если нужна логарифмическая шкала для данной оси.
- size - ширина оси (количестве символов).

Получить установленные значения можно с помощью функции **axes**(), которая возвращает таблицу параметров для {*x*, *y*, *z*}.

```
## fig3 = Ap()
## src = fig1:axes()
## src.x.range
{ -1, 1, }
## fig2:setX(src.x); fig2:setY(src.y)  -- копирование свойств осей
```

Метки на осях не связаны с единицами измерения и предназначены, прежде всего, для удобства восприятия. Они отображаются тогда, когда ось делится нацело на 2^n частей.

3.3 Виды графиков

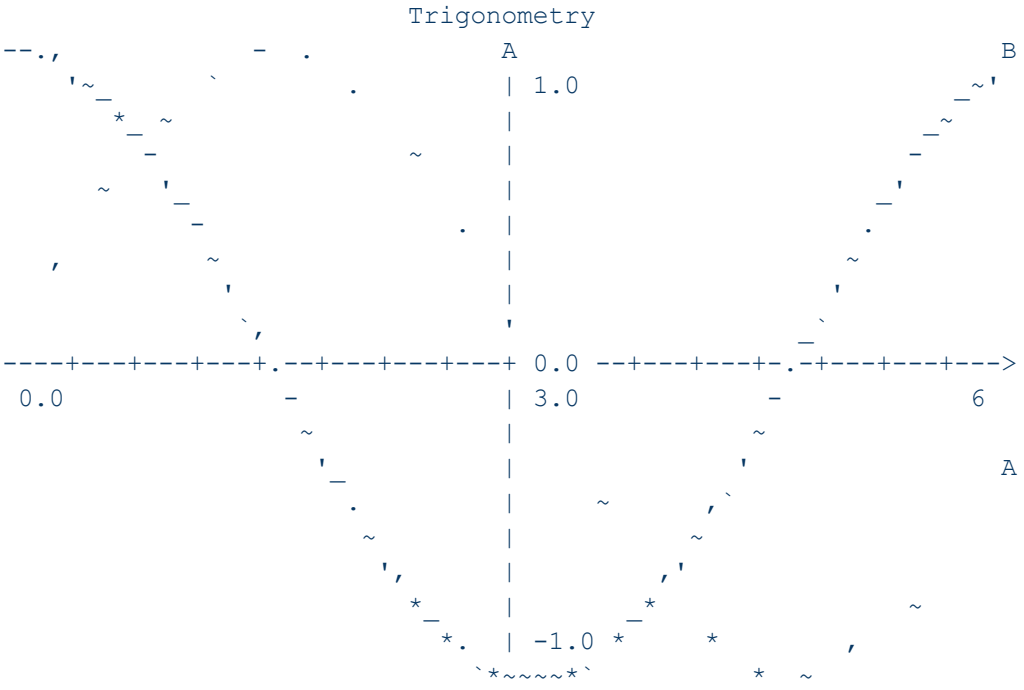
3.3.1 Функция plot

plot(*G1*, ... *Gn*) это универсальная функция для визуализации результатов расчётов. Её вызов похож на работу с одноимёнными функциями Matlab или matplotlib (Python). Каждый из аргументов *G1*, ... *Gn* представляет собой последовательность элементов вида:

- значения Y, название (если это единственный аргумент, название можно опустить)
- значения X, значения Y
- значения X, значения Y, название
- функция
- функция, название

Таким образом, аргументами являются списки чисел, функции и строки. Если значения осей не закреплены, график будет масштабироваться, чтобы уместить все заданные значения.

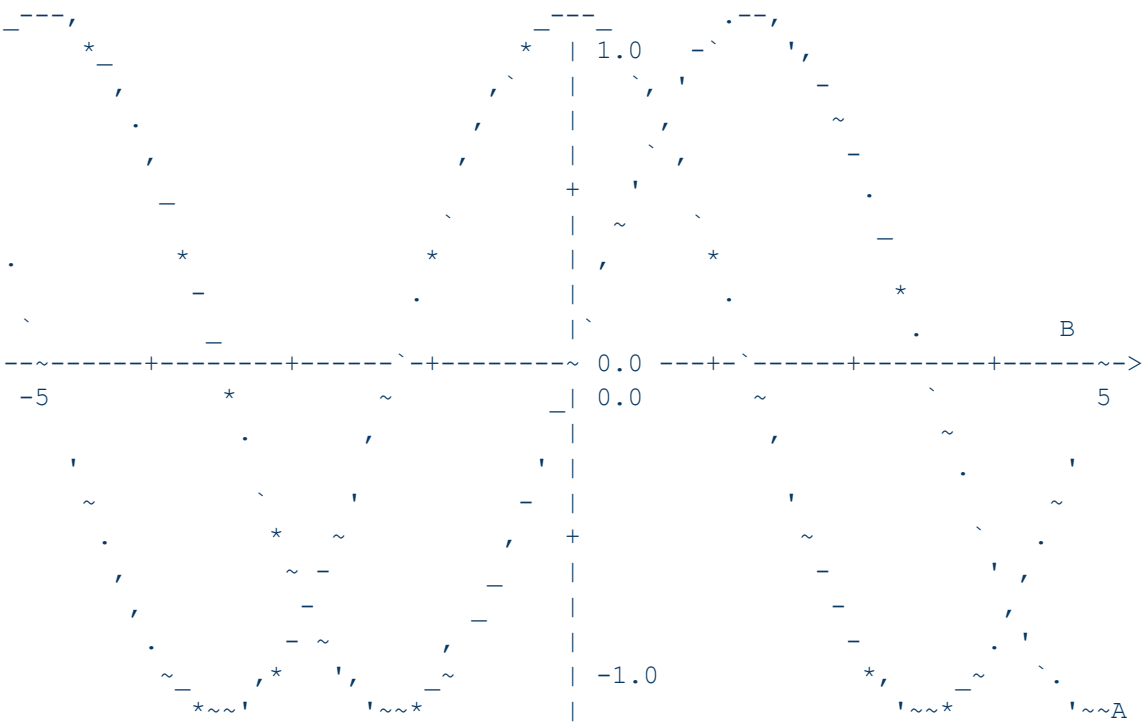
```
## use 'data'
## xs = D:range(0, 6, 0.3)
## ys = xs:map(math.sin)
## fig:plot(xs, ys, 'sin', math.cos, 'cos')
## fig:title 'Trigonometry'
## fig
```



```
(A) sin
(B) cos
```

Функция **title**(*название*) устанавливает наименование графика. Также можно добавить или изменить легенду функцией **legend**(*список*), аргументом которой является таблица с именами кривых. Если хочется быстро построить график, без необходимости явно работать с холстом, можно воспользоваться функцией **Plot**(...), которая принимает тот же набор аргументов, что и **plot**(), но скрывает все действия с *asciiplot*, соответственно, не позволяет его редактировать.

```
## Plot(math.sin, math.cos)
```




```
(A) line 1
(B) line 2
```

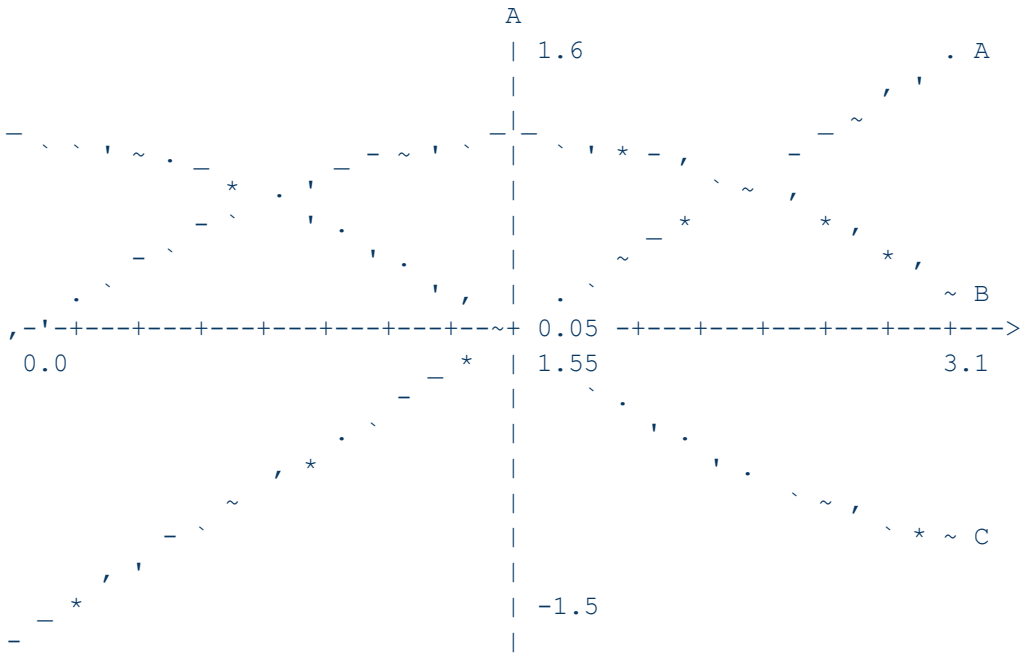
Как видно из графика, диапазон *x* для данной функции составляет от -5 до 5. Если нужно его изменить, придётся использовать списки.

3.3.2 Табличные данные

Если данные представлены в виде двумерного массива, для их визуализации можно воспользоваться функцией **tplot**(*таблица*, *параметры*). В данном случае, массив считается списком строк, независимой переменной по умолчанию является первый элемент строки, остальные - его функции.

```
## tbl = {}
## for x = 0, 3.1, 0.1 do  \
..   tbl[#tbl+1] = {x, x-1.5, sin(x), cos(x)}  \
.. end
## fig:tplot(tbl)
## fig:legend {'line', 'sin', 'cos'}
```

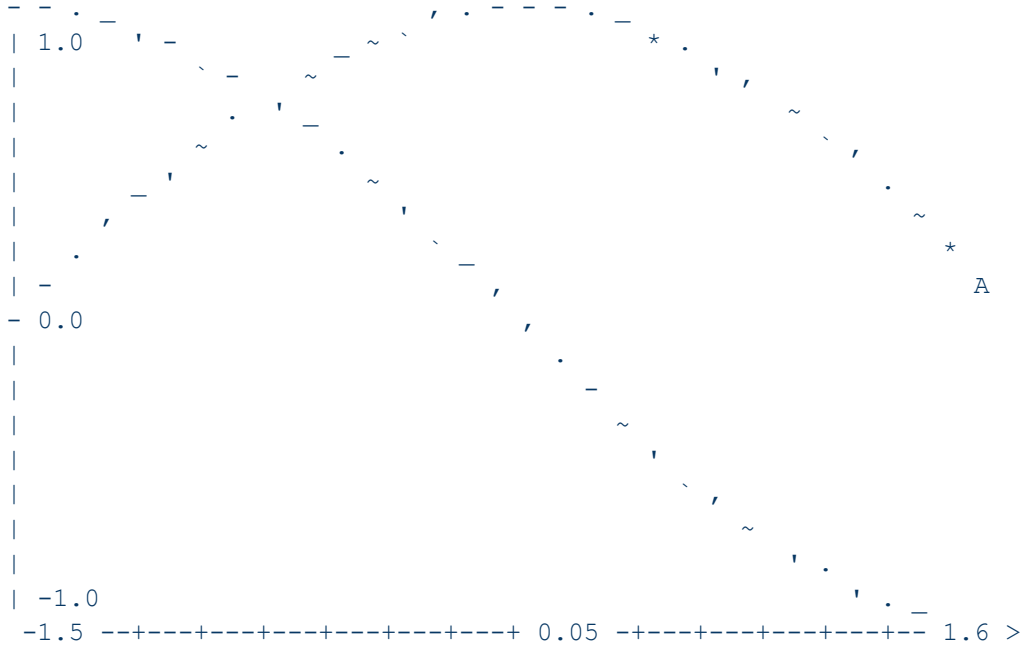
```
## fig
```




```
(A) line
(B) sin
(C) cos
```

Функция **tplot**() строит графики для всех столбцов таблицы. Если же мы хотим визуализировать отдельные столбцы, их номера можно перечислить в таблице с опциями. Здесь же можно указать индекс, который соответствует независимым элементам, его следует присвоить полю с именем *x*.

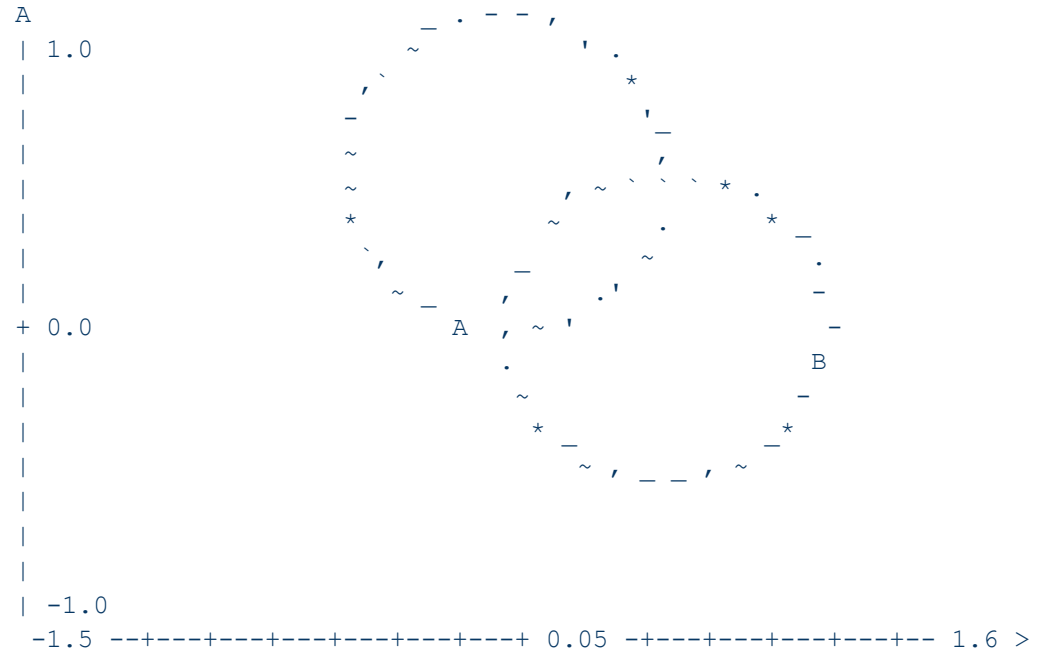
```
## fig:setX {view='min'}
## fig:setY {view='min'}
## fig:tplot(tbl, {3, 4, x=2})
```




```
(A) column 3
(B) column 4
```

Функция **tplot**() позволяет также строить графики в полярных координатах, т.е. независимая переменная считается углом, а её функции - радиусами. Для этого необходимо установить опцию *polar=true*.

```
## fig:tplot(tbl, {3, 4, polar=true})
## fig:legend {'sin', 'cos'}
## fig
```




```
(A) sin
(B) cos
```

3.3.3 Столбчатая диаграмма

Часть данных, особенно представленных в виде таблиц, удобнее изображать столбчатыми диаграммами. С учётом специфики псевдо-графического отображения такие диаграммы ``повёрнуты'' на бок, т.е. столбцы располагаются сверху вниз, при этом слева записывается аргумент, а справа - значение. Построение осуществляет функция **bar**(*таблица*, *iy*, *ix*), первый аргумент которой - двумерный массив, второй и третий аргументы опциональны и обозначают номера столбцов функции и независимой переменной соответственно (по умолчанию *iy* = 2, *ix* = 1).

```
## fig:setX {view='mid'}
## fig:bar(tbl, 4)
## fig
0.0                         ====================  1.0
0.2                         ====================  0.980066
0.4                         ==================    0.921060
0.6                         ================      0.825335
0.8                         ==============        0.696706
1.0                         ==========            0.540302
1.2                         ========              0.362357
1.4                         ====                  0.169967
```

```
1.6                          ==                      -0.02919
1.8                        =====                     -0.22720
2.0                    =========                     -0.41614
2.2                  ============                    -0.58850
2.4              ===============                     -0.73739
2.6            =================                     -0.85688
2.8         ====================                     -0.94222
3.0       ======================                     -0.98999
```

Функция **bar**() пытается вписать данные в заданный размер по *y*. Если таблица длинная или короткая, можно явно задать высоту поля для отрисовки.

```
## t1 = D:ref(tbl, 1, 11)
## fig:setX {view='min'}
## fig:setY {size=#t1}
## fig:bar(t1, 4, 1)
## fig
-1.5  ======================================  1.0
-1.4  ======================================  0.995004
-1.3  ==================================        0.980066
-1.2  ================================          0.955336
-1.1  ==============================            0.921060
-1.0  ============================              0.877582
-0.9  ========================                  0.825335
-0.8  ====================                      0.764842
-0.7  ================                          0.696706
-0.6  ==========                                0.621609
-0.5  ====                                      0.540302
```

3.3.4 Контуры

Модуль *asciiplot* не умеет строить изометрические изображения. Однако, он позволяет отобразить проекции функции двух аргументов на плоскость в виде линий равной высоты. Для этого служит метод **contour**(*функция*, *опции*). Таблица опций определяет число уровней (*level*), используемых для визуализации, а также тип проекции (*view*='XY'|'XZ'|'YZ'). Если указать *view*='*concat*', функция вернёт текст с изображением всех трёх проекций.

```
## fig:setX {range={-5,5}}
## fig:setY {range={-5,5}}
## function saddle (x, y) return x*x - y*y end
## fig:contour(saddle)
                        X-Y view
     b      a               A                a      b
          b        aa       | 5      aa        b
 d     c       b            |             b       c     d
    d                bb     |      bb                d
              c             +              c
                 cc         |          cc
          d         cc       |       cc         d
           d           ccc   |   ccc            d
-----e------d-----------cccc-+ 0.0 -----------d------e---->
```

```
 -5        d           ccc   | 0.0 c           d        5
          d         cc       |       cc         d
                cc           |          cc
             c               +             c
    d               bb       |      bb               d
 d     c       b              |             b       c     d
          b        aa         | -5     aa        b
      b      a                |               a      b
(Z1) a(-16.50)  b(-8.00)  c(0.50)
(Z2) d(9.00)  e(17.50)
```

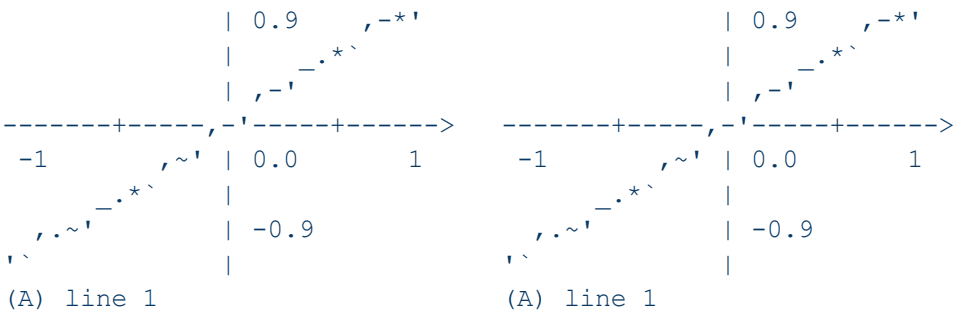
Следует иметь в виду, что в случае симметричных объектов часть линий могут совпадать.

```
## fig:contour(saddle, {view='XZ', level=3})
## fig
                          X-Z view
b                             A
 bb                           | 26.0                      bb
a  bb                         |                         bb
 aa  bb                       |                       bb  aa
   aa  bbb                    +                    bbb  aa
     aa   bbb                 |                 bbb   aa
       aaa   bbb              |              bbb   aaa
          aaa   bbbbb         |         bbbbb   aaa
-------------aaa-----bbbbbbbbb 0.5 bbb-----aaa------------>
 -5             aaaaa         | 0.0    aaaaa             5
                     aaaaaaaaaaaaaaaaa
                              |
                              +
                              |
                              |
                              | -25.0
                              |
(Y1) a(-2.50)  b(0.00)  c(2.50)
```

3.4 Другие операции

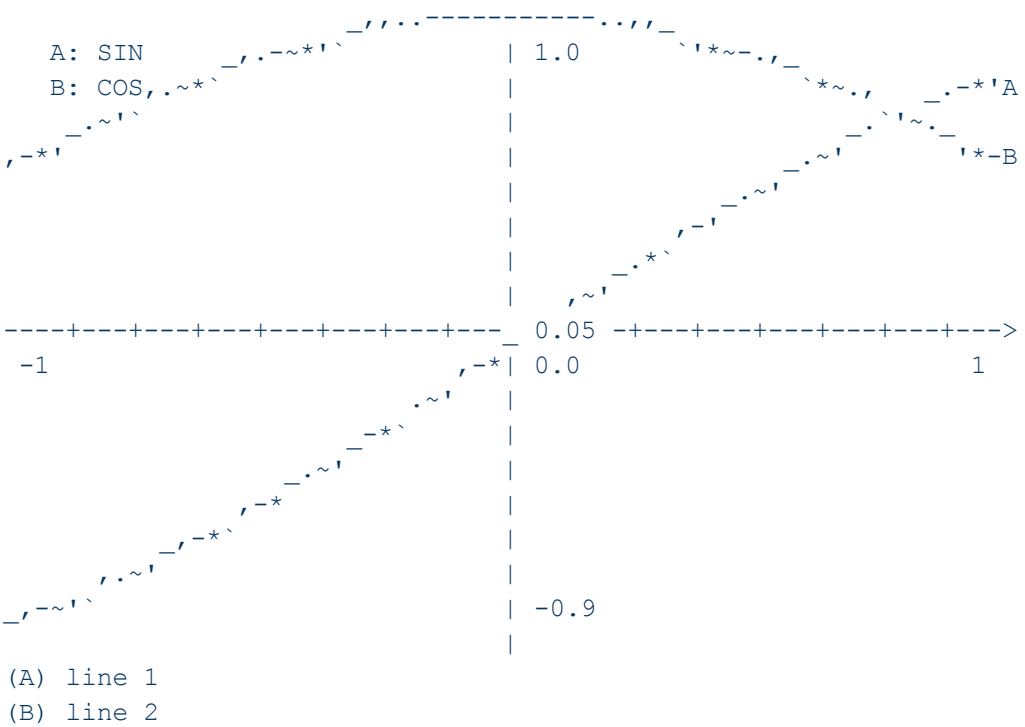
В *asciiplot* имеется ряд методов, для манипуляций с созданными ранее графиками. Во-первых, можно создать полную копию с помощью **copy**(). Во-вторых, можно горизонтально конкатенировать графики функцией **concat**(*график1*, *график2*) или оператором '..'. При этом высоты обоих графиков должны совпадать. Результатом работы функции является текст (строковый объект).

```
## fig1 = Ap():scale(0.4)
## fig1:plot(math.sin)
## fig1:title 'Left'
## fig2 = fig1:copy()
## fig2:title 'Right'
## fig1 .. fig2    -- Ap:concat(fig1, fig2)
           Left                              Right
              A           _A                    A           _A
```



Рад функций позволяют наносить элементы на график вручную: **addPoint**(*x*, *y*, *символ*) отмечает точку (*x*,*y*) заданным символом, **addPose**(*строка*, *столбец*, *символ*) ведёт себя аналогично, но помещает символ в заданную позицию холста, **addString**(*строка*, *столбец*, *текст*) помещает в заданную строку текст. Очистить холст позволяет функция **reset**().

```
## fig:plot(math.sin, math.cos)
## -- добавим легенду в основной области
## fig:addString(2, 4, 'A: SIN')
## fig:addString(3, 4, 'B: COS')
## fig
```



IV *matrix*: операции с матрицами

4.1 Введение

Упрощённо говоря, матрица представляет собой двумерный массив чисел, для которого определены алгебраические операции. Для обращения к элементу матрицы необходимо указать 2 числа: номера строки и столбца.

В so/\/ata матрицы представлены двумерными таблицами, число строк возвращает функция **rows**(), столбцов - **cols**(). Базовым конструктором является **Mat**(*таблица*), который преобразует заданную таблицу в матрицу. Индексация, как и в математике, начинается с 1.

```
## use 'matrix'
## a = Mat {  \
..   {1, 2},  \
..   {3, 4}}
## a
1  2
3  4
## a:cols()
2
## a:rows()
2
## a[2][1]
2
## a[4][5]   -- за пределами матрицы
0
```

В общем случае, стандартная операция определения размера “#” для матрицы не работает, поэтому для обхода массива необходимо использовать число строк и столбцов. При попытке обращения к элементу за пределами допустимого диапазона будет возвращён 0.

Создать нулевую матрицу заданного размера позволяет **zeros**(*строки*, *столбцы*), если число столбцов не указано, строится квадратная матрица. При этом формируется пустая таблица, для неинициализированных явно элементов будет возвращаться 0. Чтобы построить матрицу, заполненную некоторым числом, можно воспользоваться функцией **fill**(*строки*, *столбцы*, *значение*), по умолчанию значение 1. Единичная матрица (т.е. нулевая матрица с единицами на главной диагонали) может быть сформирована функцией **eye**(*строки*, *столбцы*). Функция **D**() формирует квадратную матрицу с заданными диагональными элементами, второй опциональный аргумент обозначает смещение выше или ниже главной диагонали.

```
## b = Mat:zeros(2)
## b[1][1] = 3
## b
3  0
0  0
```

```
## Mat:fill(1, 3)
1  1  1
## Mat:eye(2, 3)
1  0  0
0  1  0
## Mat:D({1,2}, 1)
0  1  0
0  0  2
0  0  0
```

4.2 Основные операции

4.2.1 Арифметические операции

Для матриц в so/\/ata определены основные арифметические операции: сложение, вычитание, умножение, возведение в целую степень, унарный минус. Также возможна проверка матриц на равенство и неравенство. Операции с числами выполняются поэлементно.

```
## a + b
4  2
3  4
## a - b
-2  2
3  4
## a * b
3  0
9  0
## a^3
37   54
81  118
## a == Mat{{1,2},{3,4}}
true
## a + 1
2  3
4  5
```

Деление матриц не определено, необходимо использовать домножение на (псевдо-) обратную матрицу.

Если в результате умножения образуется матрица с одним элементом, результат преобразуется в число. Как и в случае с комплексными числами, можно воспользоваться конструктором **Mat**() чтобы гарантировать, что будут выполнены операции над матрицами.

4.2.2 Методы

Ряд функций определены только для квадратных матриц. Среди них вычисление определителя **det**() и обратной матрицы **inv**(). Дополнительный минор для элемента с координатами *i*, *j* может быть найден с помощью функции **minor**(*i*,

j). Метод **eig**() возвращает две матрицы, элементами первой являются собственные векторы матрицы, элементами второго - соответствующие им собственные числа, расположенные по диагонали.

```
## a:det()
-2
## a:inv()
-2    1
1.500  -0.500
## a:minor(1, 1)
4
## p, q = a:eig()
## p
0.416   0.825
0.909  -0.566
## q
5.372       0
0  -0.372
## p * q * p:inv()
1.000  2.000
3.000  4.000
```

Следующие методы являются универсальными, т.е. могут быть применены к любому типу матрицы. Функция **rank**() определяет ранг матрицы, **norm**() находит евклидову норму. Выразить диагональные элементы в виде вектора-столбца позволяет функция **diag**(), а с помощью **tr**() можно определить след матрицы, т.е. сумму диагональных элементов. Псевдо-обратную матрицу вычисляет функция **pinv**().

```
## b:rank()
1
## a:norm()
5.4772255750517
## a:tr()
5
## c = Mat {{2, 3}}
## ci = c:pinv()
## c * ci
1.0
```

4.3 Объекты-ссылки

Ряд функций возвращают объекты, которые ведут себя как матрицы, но сами данные не хранят, а ссылаются на другой объект. Преимуществами таких объектов является быстрота создания и относительно малый размер потребляемой памяти. К недостаткам можно отнести накладные расходы на вычисление индексов, а также невозможность независимой от оригинала модификации. Чтобы повысить скорость обработки и сделать объект самостоятельным, используйте метод **copy**(), который создаёт матрицу-копию.

Функция **T**() возвращает транспонированную матрицу (ссылку), а **H**() - транспонированную и комплексно сопряженную.

```
## c = Mat{{4,5},{6,7}}
## d = c:T()
## d
4  6
5  7
## d[1][2] = 1
## c
4  5
1  7
```

Для выделения диапазона данных нет отдельной функции, но данный функционал реализуется при вызове матрицы как функции, т.е. при использовании круглых скобок. При этом нужно передать в качестве аргументов два списка, обозначающих диапазоны строк и столбцов соответственно. В общем случае, диапазон обозначается тремя числами: первый индекс, последний индекс, шаг. Пропущенный шаг заменяется единицей, последний индекс - концом списка, а первый индекс - началом, таким образом, пустой список означает все строки (или столбцы). Отрицательные начало или конец означают отсчёт с конца, отрицательный шаг - изменение порядка элементов. Если нас интересует конкретный столбец или строка, вместо диапазона нужно передать в вызов соответствующее число. Если оба аргумента являются числами будет возвращён соответствующий элемент, при этом также можно использовать отрицательные значения для отсчёта с конца.

Все ссылочные объекты содержат поле *data*, но для диапазона значений его использование особенно актуально. При чтении этого поля будет возвращён тот объект, на который выполнена ссылка, при записи происходит копирование элементов объекта справа от знака равенства в соответствующие элементы объекта-ссылки.

```
## c = Mat:eye(3)
1  0  0
0  1  0
0  0  1
## c({1,2},{1,2}).data = a
1  2  0
3  4  0
0  0  1
## c(1, {})
1  2  0
## c(-1,-1)
1
```

Функция **reshape**(*строки*, *столбцы*) позволяет изменить размер матрицы. Перемещение элементов происходит построчно, лишние элементы отбрасываются, вместо недостающих добавляются нули.

```
## c:reshape(4,2)
1  2
0  3
4  0
0  0
```

Для объединения матриц служат функции **hor**(*список*) и **ver**(*список*), первая выполняет горизонтальную конкатенацию, вторая - вертикальную, аргументами являются списки матриц.

```
## Mat:ver {            \
..   Mat:hor {b, a},    \
..   Mat:hor {a:T(), b} \
.. }
3  0  1  2
0  0  3  4
1  3  3  0
2  4  0  0
## a..b
1  2  3  0
3  4  0  0
```

Горизонтальную конкатенацию двух матриц можно осуществить стандартным оператором многоточия, при этом формируется не ссылка, а новая матрица.

4.4 Векторные операции

Вектором можно назвать матрицу, которая состоит из одной строки или столбца. Соответственно, к векторам применимы стандартные арифметические операции и большая часть функций, определённых для матриц. Для упрощения создания вектора-столбца предусмотрена функция **V**(*элементы*).

```
## v1 = Mat:V {1,2,3}  -- вектор-столбец
## v1
1
2
3
## v2 = Mat{{4,5,6}}   -- вектор-строка
## v2
4  5  6
## v2 * v1
32
## v1 * v2
4   5   6
8  10  12
12  15  18
```

Некоторым неудобством является то, что для обращения к элементу нужно указывать 2 индекса, как для обычной матрицы. Частично это проблему решает оператор “()”, но он позволяет только прочитать значение. Чтобы упростить работу с векторами, введён ещё один ссылочный объект, который формируется функцией

vec(). Для трёхэлементных векторов это даёт возможность также использовать индексы x, y, z.

```
## v1(3)  -- чтение значения
3
## p = v1:vec()
## q = v2:vec()
## p[3]   -- чтение значения
3
## p[1] = 0  -- присваивание нового значения
## p
0  2  3
## v1   -- исходная матрица также меняется
0
2
3
## #p
3
## q.z   -- доступ по "индексу"
6
```

В отличие от предыдущих объектов, данный тип не является матрицей, зато содержит ряд специфических методов, таких как векторное (**cross**) и скалярное (**dot**) произведение, нормализацию вектора **normalize**(), а также вычисление нормы для различных метрик (l_1, l_2, l_∞).

```
## p:dot(q)
28
## p:cross(q)
-3
12
-8
## p:norm('l1')
5
## q:normalize()
## q
0.455  0.569  0.683
## q.data = p   -- копирование элементов вектора
## q
0  2  3
```

4.5 Преобразования

В этом разделе будут описаны различные преобразования и разложения матриц, доступные в so/\/ata. Начнём с преобразования Гаусса (функция **rref**()), которое чаще всего используется для решения систем линейных уравнений. При этом модифицируется матрица, переданная в аргументе.

```
## m = a..Mat:V{5,6}
## m
1  2  5
```

```
3  4  6
## m:rref()
1  0  -4
0  1  4.500
```

LU разложение квадратной матрицы выполняет функция **lu**(), которая возвращает нижнюю и верхнюю треугольные матрицы (L, U), а также матрицу перестановок P.

```
## L,U,P = a:lu()
## L*U == P*a
true
## L
    1  0
0.333  1
## U
3      4
0  0.667
```

Для QR разложения служит функция **qr**(), при этом число строк аргумента должно быть не меньше, чем столбцов. Результатом работы являются матрица с ортонормированными столбцами Q и верхняя треугольная R.

```
## Q,R = a:qr()
## Q*R
1.000  2.000
3.000  4.000
## Q
-0.316  -0.949
-0.949   0.316
## R
-3.162  -4.427
0       -0.632
```

Разложение Холецкого выполняет функция **chol**(), которая возвращает нижнюю треугольную матрицу *либо nil*, если исходная матрица не является положительно определённой.

```
## m = Mat {{1,3},{3,10}}
## L = m:chol()
## L
1  0
3  1
## L*L:T()
1   3
3  10
```

Сингулярное разложение осуществляет функция **svd**(), которая возвращает матрицу с сингулярными числами на главной диагонали, а также матрицы левых и правых сингулярный векторов.

```
## U,B,V = a:svd()
```

```
## B
5.465      0
    0  0.366
## U*B*V:T()
1.000  2.000
3.000  4.000
```

Матричную экспоненту можно найти с помощью функции **exp**().

```
## a:exp()
51.98  74.75
112.1  164.1
```

Применить некоторую функцию одного аргумента к каждому элементу матрицы позволяет функция **map**(*функция*, *строка*, *столбец*). Дополнительно здесь можно задать условие, привязанное к номеру строки и столбца. Если же необходимо применить функцию к нескольким матрицам, используйте **zip**(*функция*, ...).

```
## f1 = function (x) return 2*x end
## a:map(f1)
2  4
6  8
## f2 = function (x,y) return x*y end
## Mat:zip(f2, a, b)
3  0
0  0
```

Т.о. **zip**() позволяет выполнить поэлементное произведение и подобные ему действия.

4.6 Другие операции

Произведение и сумму Кронекера можно вычислить с помощью методов **kron**(*матрица*) и **kronSum**(*матрица*) соответственно.

```
## a:kron(a)
1   2   2   4
3   4   6   8
3   6   4   8
9  12  12  16
## a:kronSum(b)
4  0  2  0
0  1  0  2
3  0  7  0
0  3  0  4
```

Функция **vectorize**() преобразует матрицу в вектор-столбец, при этом, в отличие от функции **reshape**(), группировка элементов происходит не по строкам, а по столбцам. Обычно применяется совместно с произведением Кронекера.

```
## (a*b):vectorize()
3
9
0
0
## Mat:eye(2):kron(a) * b:vectorize()
3
9
0
0
```

Иногда бывает полезно визуально отобразить матрицу. Функция stars(критерий) вместо чисел выводит на печать пробелы и звёздочки исходя из заданного булевого условия. По умолчанию происходит проверка на 0.

```
## Mat:eye(3):stars()
*  |
 * |
  *|
```

V *complex*: комплексные числа

5.1 Введение

Работу с комплексными числами в so/\/ata обеспечивает модуль *complex*. Конструктор числа имеет вид

```
## use 'complex'
## re = 1
## im = 2
## Z(re, im)
1+2i
```

т.е. первый аргумент соответствует действительной части, второй - мнимой. Также можно использовать полярную (экспоненциальную) форму представления $r\,e^{i\alpha}$ и конструктор **cis**(*угол*)

```
## 2*Z:cis(PI/4)
1.414+1.414i
```

Комплексные числа являются неизменяемые, т.е. функции не модифицируют существующий объект, а создают новый. Если определён параметр *ROUND*, то действительная и мнимая части будут урезаться до заданной точности с округлением в сторону наименьшей ошибки.

```
## Z.ROUND = 1E-3
## Z(1E-10, 2.34567)
0+2.346i
```

5.2 Основные операции

Комплексные числа в so/\/ata поддерживают основные арифметические операции, результат приводится к обыкновенному числу, если мнимая часть равна нулю. Если нужно выполнить над результатом действия, специфические для комплексных чисел, можно передать его через конструктор.

```
## a, b = Z(1,2), Z(1, -2)
## a + b
2
## Z(a + b):im()
0
## a ^ b
17.94-9.855i
```

Функции **re**() и **im**() возвращают действительную и мнимую части числа соответственно. Модуль числа можно найти функцией **abs**(), а аргумент (угол) для экспоненциальной записи с помощью **arg**(). Комплексно-сопряжённое число

можно получить при помощи функции **conj**() или оператора побитового отрицания “~”:

```
## a:abs()
2.2360679774998
## a:arg()
1.1071487177941
## ~a
1-2i
```

5.3 Функции

Практически все математические функции модуля *main* (кроме **atan2**()) определены и для **complex**, причём, вызов **fn**(x) будет более универсальным чем **x:fn**(), поскольку в первом случае **x** может быть произвольным числовым типом so/\/ata, а во втором - только комплексным числом.

```
## a:exp()
-1.131+2.472i
## exp(a)
-1.131+2.472i
```

При загрузке *complex* происходит переопределение функций **sqrt**() и **log**() для работы с отрицательными числами.

```
## sqrt(-1)
0+1i
## log(-1)
0+3.142i
```

VI *polynomial*: полиномы

6.1 Введение

Полиномы в so/\/ata представляются списком коэффициентов. В математике обычно их принято записывать от старшей степени к младшей, такая же нотация принята в so/\/ata. Т.е. выражение $x^2 + 2x + 1$ может быть представлено в виде **Poly** {1, 2, 1}, где Poly это конструктор. В полученном объекте коэффициенты располагаются таким образом, что индекс элемента соответствует его степени.

```
## use 'polynomial'
## a = Poly {1, 2, 1}
## a
1 2 1
## a[0]   -- свободный коэффициент
1
```

Иногда может быть удобнее использовать “алгебраическую” запись полинома в виде суммы степеней, особенно при наличии нулевых коэффициентов. Для этого можно использовать объект, возвращаемый функцией **x**(). Строку с алгебраической записью полинома возвращает функция **str**(*символ*), по умолчанию переменная обозначается через *x*. Значение полинома для заданного аргумента *t* можно получить функцией **val**(t), либо просто оператором “()”. Копию полинома строит функция **copy**().

```
## x = Poly:x()
## b = 2*x^3 + 3*x^2 + 1
## b
2 3 0 1
## b:str()
2*x^3+3*x^2+1
## b(2)  -- значение при x=2
29
```

6.2 Основные операции

С полиномами в so/\/ata можно выполнять основные арифметические операции: сложение, вычитание, умножение, деление (“нацело”), остаток от деления и возведение в натуральную степень. Также возможна проверка на равенство.

```
## p1 = Poly {1, 1}
## p2 = p1^2 + 1
## p2
1 2 2
## p1 + p2
1 3 3
```

```
## p2 - p1
1 1 1
## p2 / p1
1 1
## p2 % p1
1
## p2 / p1 == p1
true
```

Вычислить производную позволяет функция **der**(), а интеграл - **int**(*p0*), где аргументом является значение свободного коэффициента.

```
## p2:der():str()
2*x+2
## p2:int(7):str()
0.333*x^3+x^2+2*x+7
```

Корни полинома вычисляет функция **roots**(), которая возвращает список, сначала идут действительные корни, затем мнимые. Обратную операцию, построение полинома по известным корням, выполняет функция **R**(*корни*).

```
## rs = p2:roots()
## rs
{ -1-1i, -1+1i, }
## Poly:R(rs)
1 2 2
```

Функция **char**(*матрица*) находит характеристический полином для заданной матрицы.

```
## use "Mat"
## m = Mat{{1,2},{3,4}}
## cp = Poly:char(m)
## cp:str()
x^2-5*x-2
## r = cp:roots()
## r[1] * r[2]
-2.0
## m:det()
-2
```

Если в результате вычислений остаётся только свободный коэффициент, он преобразуется в обычное число. Чтобы сохранить результат в виде полинома, можно передать его в конструктор **Poly**, по аналогии с матрицами и комплексными числами.

6.3 Аппроксимация

6.3.1 Полиномы

Функции для аппроксимации можно разделить на 2 группы: одни проводят кривую через заданные точки, а вторые находят приближение. Представителем первой группы является **lagrange**(*xs*, *ys*), которая строит полином Лагранжа, а аргументами являются списки *x* и *y* значений в узловых точках.

```
## p3 = Poly:lagrange({0,1,2}, {3,5,1})
## p3:str()
-3.0*x^2+5.0*x+3.0
## p3(1)
5.0
```

Если для некоторой точки x_0 известно значение функции и первых производных, значение функции в окрестности x_0 может быть рассчитано с помощью полинома Тейлора, который строит функция **taylor**(*x0*, *f*, *f'*, ...).

```
## x0 = PI/4
## f, f1, f2 = sin(x0), cos(x0), -sin(x0)
## p4 = Poly:taylor(x0, f, f1, f2)
## p4:str()
-0.353*x^2+1.26*x-0.066
## p4(PI/2)
1.0443776422176
```

Функция **fit**(*xs*, *ys*, *степень*) вычисляет наилучшее приближение полиномом заданной степени для исходного множества точек. При этом используется метод наименьших квадратов.

```
## use 'data'
## xs = D:range(0, 10)
## ys = xs:map(function(x) return p2(x)+0.5*math.random() end)
## Poly:fit(xs, ys, 2):str()
0.981*x^2+2.15*x+2.12
```

6.3.2 Сплайны

Если искать единую кривую, которая проходит через все заданные точки, не требуется, можно разбить последовательность узлов на пары и соединить их кривыми по отдельности. Функция **lin**(*xs*, *ys*, *y0*, *yN*) строит интерполяцию прямыми. За пределами диапазона *xs* по умолчанию происходит экстраполяция полученными функциями, но можно использовать константные значения, если установить значения *y0* и *yN*. Значение в заданной точке вычисляется также как для обычного полинома, но возвращается два числа, первое является значением, второе - индексом полинома в списке. При вычислении значения также можно указывать индекс полинома, тогда расчёт будет выполняться быстрее.

```
## pp1 = Poly:lin({0,1,2},{3,4,1}, 0)
## pp1(1.5)
2.5
## pp1(-1)
0
```

Функция **spline**(*xs*, *ys*) выполняет интерполяцию с помощью кубических сплайнов.

```
## pp2 = Poly:spline({0,1,2},{3,4,1})
## pp2(1.5)
2.875
```

Над сплайнами не определены арифметические операции, но к ним можно применять функции **copy**(), **der**() и **int**().

```
## pp1 = Poly:lin({0,1,2},{3,4,1},0)
## pp1:der():val(1.5)
-3.0
## pp1:int():val(1.5)
```

VII *numeric*: численные методы

VIII *random*: случайные числа

8.1 Введение

Модуль *random* содержит различные генераторы случайных чисел, а также функции на их основе. Вызов конструктора **Rand**() возвращает случайное число от 0 до 1, формируемое стандартным генератором Lua на основе равномерного распределения. Функция **new**() создаёт новый генератор равномерно распределённых случайных чисел, не связанный с *math.random*. Такой генератор работает чуть медленнее, но это позволяет получить независимые случайные числа в разных частях кода. Задать ядро генератора позволяет функция **seed**(*число*). Ядро представляет собой целое число и может быть использовано для обеспечения повторяемости возвращаемых значений при разных вызовах. **seed**() без аргумента сбрасывает ядро до “случайного” значения.

```
## use 'random'
## Rand:seed(11)
## Rand()
0.44766006064378
## rnd = Rand:new()
## rnd()
0.84616688955534
## rnd1 = Rand:new():seed(3)
## rnd2 = Rand:new():seed(3)
## rnd1() == rnd2()  -- одинаковые последовательности
true
## rnd() == rnd1()   -- разные последовательности
false
```

Все функции модуля могут вызываться как с генератором по умолчанию (через **Rand**), так и с кастомными генераторами.

8.2 Распределения случайных чисел

В модуле random доступны генераторы для следующих распределений случайных чисел:

- **norm**(μ, *σ*) - распределение Гаусса;
- **rayleigh**(*σ*) - распределение Рэлея;
- **poisson**(*λ*) - распределение Пуассона;
- **cauchy**(μ, *σ*) - распределение Коши;
- **exp**(*λ*) - экспоненциальное распределение;
- **gamma**(*α*, *β*) - гамма распределение;
- **logistic**(μ, *σ*) - логистическое распределение;
- **binomial**(*p*, N) - биномиальное распределение;
- **int**(*от*, *до*) - равномерно распределённые целые числа.

В качестве примера построим гистограмму для нормального распределения и сравним с теоретическим значением $exp(-\ 0.5(x\ -\ \mu)^2/\sigma^2)/(\sigma\ sqrt(2\ \pi))$.

```
## -- генерация чисел
## lst = {}; N = 200
## mu = 1; sig = 0.5
## for i = 1, N do lst[i] = Rand:norm(mu, sig) end
## -- гистограмма
## step = 4*sig/10
## lim = D:range(mu-2*sig, mu+2*sig, step)
## ts, te = D:histcounts(lst, lim)
## -- нормализация
## for i, v in ipairs(ts) do ts[i] = v / (N * step) end
## -- плотность распределения
## pdf = function (x) \
.. return exp(-0.5*(x-mu)^2/sig^2) / (sig*sqrt(2*PI)) end
## -- график
## fig = Ap()
## fig:setX {range={mu-2*sig, mu+2*sig}, view='min'}
## fig:setY {view='min'}
## fig:plot(pdf, 'pdf')
## -- изобразим элементы гистограммы с помощью '@'
## for i, v in ipairs(te) do fig:addPoint(v-0.5*step, ts[i], '@') end
## fig
A                          _.---._
| 0.8                   -'       @'-
|                   @ -`            `-
|                   .`    @           `.
+                 ~                     ~
|               ,`                       `,
|              -                          -
|             @                            *
+ 0.45       ,`                             `,
|           -                                -
|          *                                  *       @
|        .`                                     `.
+       ~@                                     @      ~
|  @  -`                                                `-
|   .'                                                    '.
| 0.1                                                       '-A
0.0 ------------------------+ 1.0 ------------------ 2.0 >
(A) pdf
```

8.3 Другие функции

Наряду с генераторами, модуль random содержит ряд функций, работа которых напрямую связана со случайными числами. Функция **flip**(*p*) возвращает true с заданной вероятностью, по умолчанию равной 0.5. **bytes**(*длина*) генерирует случайную последовательность ascii символов заданного размера. **choice**(*таблица*) возвращает случайный элемент и его индекс, **shuffle**(*таблица*) перемешивает

элементы таблицы, а **ipairs**(*таблица*) возвращает итератор для обхода в случайном порядке (без повторений).

```
## Rand:flip()
true
## rnd:bytes(10)
s+{/axj)*/
## t = {2, 1, 7, 9, 0, 4}
## Rand:choice(t)
9
## for i, v in Rand:ipairs(t) do print(i, v) end
4    9
3    7
5    0
1    2
2    1
6    4
## Rand:shuffle(t)
## t
{ 4, 1, 0, 9, 2, 7, }
```

IX *bigint*: длинные целые числа

9.1 Введение

Модуль *bigint* изначально появился для поддержки чисел произвольной длины в различных системах счисления, однако со временем фокус сместился в сторону функций над целыми числами. Тем не менее, произвольная длина и преобразование систем счисления по-прежнему доступны.

Объект *bigint* создаётся с помощью конструктора **Int**(*значение*), аргументом может выступать целое число, строка и таблица. Использование чисел - самый простой способ, но он допускает ограниченный диапазон значений. Использование строки более гибко. Если число имеет основание 10 либо начинается с ‘0x’ и ‘0b’, основание системы счисления можно не писать, а числа записываются последовательно без разделителей. В общем случае основание записывается в конце и отделяется двоеточием, а знаки числа записываются в десятичном представлении и разделяются запятой. Вместо запятой можно использовать любой знак пунктуации (кроме двоеточия), а также чередовать разделители для большей наглядности. Таблица позволяет перечислить знаки числа, а также задать основание системы счисления *base* и знак *sign*=±1. При этом k-й (справа налево) элемент должен иметь индекс k, поэтому число записывается в обратном порядке.

```
## use 'bigint'
## a = Int(42)
## b = Int '1,2,3:8'
## b    -- значение выводится в десятичном виде
83
## b1 = Int{3,2,1,base=8,sign=1}  -- то же что b
## b1
83
## big = Int '123`456`789`321'
```

Для целых чисел определены стандартные арифметические и булевы операции. Преобразовать значение в обычное число позволяет функция **float**(), знак числа (1 или -1) можно получить с помощью **sign**().

```
## a + b
125
## a - b
-41
## a * b
3486
## b / a
1
## b % a
41
## b ^ a
```

```
399282135658682469334331901689876886678820965774126540592961885129481494196023689
## ANS:float()
3e+80
## a > b
false
## b == b1
true
## a:sign()
1
```

Во время арифметических вычислений объект *bigint* может быть упрощён до обычного числа Lua. Чтобы этого не произошло, результат можно передать в конструктор **Int**().

9.2 Функции

Модуль числа возвращает функция **abs**(). Случайное число от 0 до заданного значения генерирует метод **random**().

```
## c = Int(-50):random()
## c
-41
## c:abs()
41
```

Проверить число на простоту позволяет функция **isPrime**(). По умолчанию, используется перебор множителей, если добавить аргумент *Fermat*, будет использован метод Ферма. Функция **factorize**() раскладывает число на простые множители.

```
## b:isPrime()
true
## a:isPrime('Fermat')
false
## a:factorize()
{ 2, 3, 7, }
```

Наибольший общий делитель находит функция **gcd**(*n1*, *n2*, ...), наименьшее общее кратное - **lcm**(*n1*, *n2*, ...).

```
## Int:lcm(Int(2), Int(6), Int(7))
42
## Int:gcd(a, a*2, a/2)
21
```

9.3 Системы счисления

Функция **digits**(*основание*) позволяет получить представление числа в заданной системе счисления (в десятичной по умолчанию). Результатом является

таблица, где i-й знак стоит в i-й позиции. Эту таблицу можно передать в конструктор **Int**() для преобразования обратно в число. Для данных объектов определены операции сдвига << и >>, функция **group**(*число*) позволяет объединять знаки в группы для лучшей читаемости при выводе на печать.

```
## d = (a * a):digits(6)
## d
12100:6
## d >> 2
121:6
## d:group(3)
12`100:6
```

9.4 Элементы комбинаторики

Факториал может быть найден функцией **F**(), для ускорения расчёта отношения факториалов определена функция **ratF**(a, b). Число перестановок из n по k определяет функция **P**(n, k), число сочетаний - **C**(n, k). Субфакториал рассчитывается функцией **subF**(), двойной факториал - **FF**().

```
## a:F()    -- a!
140500611775287989854314260624451156993638400000000
## Int:ratF(a, a/2)
27500101936481280675682713600000
## Int:P(Int(10), Int(3))
720
## Int:C(Int(10), Int(3))
120
## Int(10):subF()  -- !10
1334961
## Int(10):FF()    -- 10!!
3840
```

X *rational*: рациональные числа

10.1 Введение

Рациональными являются числа, представимые отношением двух целых чисел. С точки зрения программной реализации, они хранят в себе оба значения: числитель и знаменатель. Создать рациональное число можно с помощью конструктора **Rat**(*числитель*, *знаменатель*), если второй аргумент опущен, он считается равным 1. Данные объекты являются “неизменяемые”.

Для рациональных чисел определены основные арифметические операции: сложение, вычитание, умножение, деление, возведение в целую степень.

```
## use 'rational'
## a = Rat(1,2)
## b = Rat(2,3)
## a + b
7/6
## b - a
1/6
## a * b
1/3
## a / b
3/4
## a^3
1/8
```

Также можно выполнять сравнение и проверку на равенство. Однако, следует помнить, что в Lua выражение “число == объект” всегда возвращает *false*, поэтому для сопоставления с числом лучше использовать метаметод **__eq**().

```
## a == b
false
## a < b
true
## a:__eq(0.5)
true
```

Для отображения дроби в смешанном виде необходимо установить флаг MIXED.

```
## Rat.MIXED = true
## a + b
1 1/6
```

10.2 Функции

Специфических функций для рациональных чисел не так много. Это чтение числителя **num**(), знаменателя **denom**(), а также преобразование в число с плавающей точкой с помощью **float**().

```
## a:num()
1.0
## a:denom()
2.0
## a:float()
0.5
```

Обратную операцию, т.е. преобразование числа с плавающей точкой в рациональное (с заданной точностью, по умолчанию 0.001), выполняет функция **from**(*число*, *точность*).

```
## Rat:from(math.pi, 1E-5)
355/113
## 355 / 113
3.141592920354
```

10.3 Цепные дроби

so/\/ata умеет преобразовывать цепные дроби вида $a_0 + 1/(a_1 + 1/(a_2 + 1/...))$ в рациональные числа и обратно. При этом цепная дробь представляется в виде таблицы положительных коэффициентов, где числу (a_0) соответствует индекс 0. Функция **toCF**() формирует данную таблицу коэффициентов, а **fromCF**() выполняет обратное преобразование.

```
## c = b:toCF()
## c
{0+L1+L2}  -- 1/(1+1/2)
## c[1]
1
## Rat:fromCF(c)
2/3
```

Знак *L* символизирует “1/(...)” и используется чтобы не вызывать путаницы с простой суммой дробей.

quaternion: операции с кватернионами

special: функции мат-физики

graph: операции с графами

units: единицы измерения и преобразования

const: коллекция констант

gnuplot: построение графиков в GNUplot

symbolic: элементы компьютерной алгебры

qubit: эмуляция квантовых вычислений

lens: оптика параксиальных лучей

geodesy: преобразования координат