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) и REPL интерпретатора.

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 программы
- интерфейс - командная строка
- отсутствие встроенных 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. 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 из произвольного места). После выполнения команды

```
lua sonata.lua
```

должно отобразиться приглашение к работе:

```
   # #        --======  so/\/ata  ======--      # #
    # #         --===== 0.9.40 =====--        # #

------- 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 функции (командный режим, использование цветов, логирование)
- более информативный вывод
- одинаковый интерфейс взаимодействия во всех версиях Lua 5.x

Недостатки:

- скорость обработки команд пользователя чуть ниже, чем в интерпретаторе Lua
- нужно явно обозначать перенос строк
- нет подстановки введенных ранее команд

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. Описанные в предыдущем пункте преимущества и недостатки при этом поменяются местами.

1.3.3 Библиотека *matlib*

Математическая библиотека *sonata/matlib* может работать без остальных компонентов программы. Однако, если планируется использовать не всю папку, а отдельные файлы, следует помнить, что модули могут зависеть друг от друга.

1.4 Аргументы запуска

Вызов

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

выводит список аргументов, с которыми программа может быть запущена.

- **-e выражение**
  позволяет выполнить команду и отобразить результат.

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

- **--doc [язык]**
  генерирует *html* страницу с описанием актуальной версии программы, файл сохраняется в текущую 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 [язык]**
  формирует файл локализации в папке *sonata/locale*, аргументом является наименование языка, подробности в приложении B. 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 модуль псевдоним [описание]**
  формирует шаблон для новой математической библиотеки, аргументами являются полное имя модуля (название генерируемого файла в *sonata/matlib*), его краткий псевдоним и (опционально) описание. Подробности в приложении C.

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

- **--test [модуль]**
  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 их 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 “--”. Сочетание знаков “--” и “[[ ]]” позволяет получить “длинный” комментарий.

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 без пробелов (и начинается с буквы или “_”), 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 чаще всего используются для описания параметров, а списки - для представления массивов.

Функции это ещё один ключевой компонент Lua. Вызов выглядит так же, как в большинстве языков 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
```

Ниже приведён пример объявления функции. 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 и ускоряет вычисления.

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, которая называется *math*. Данная библиотека является обёрткой над стандартными математическими функциями языка C. Её состав в различных версиях Lua может отличаться, но 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.

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: для каждой введённой команды 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.

Интерпретатор so/\/ata относительно прост, он умеет обрабатывать только текущую строку. Поэтому для продолжения ввода длинного выражения необходимо поставить знак “\” перед переходом на новую строку. Этот символ 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 поставить два обратных слеша, при этом окончание ввода будет задаваться пустой строкой.

```
## 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 всех загруженных модулях и функциях. Аргументом **help**() может быть и произвольный объект, 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
```

Явное использование **use**() для одиночного модуля необязательно, компонент 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.модуль*'), при этом нужно явно указать имя переменной (алиаса) для хранения ссылки на библиотеку.

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)
-- несколько функций
hor, ver = Bind(Mat, 'hor', 'ver')
```

Функция **Round**(*объект*, *позиция*) принимает число или объект (например, матрицу) и округляет его до заданного числа знаков после запятой (по умолчанию до целой части).

```
## a = Mat {{1,2},{3,4}} * PI
## a[1][1]
3.1415926535898
## b = Round(a, 3)
## b[1][1]
3.142
```

Часто работа происходит со списками или другими упорядоченными коллекциями, при этом необходимо получить новый список путём применения какой-либо операции к элементам исходного списка. 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. **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 так, 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 обработка 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 текст комментария отделён от символов “--” с помощью табуляции, он будет дополнительно подсвечен 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 ряд дополнительных команд,

управляющих его работой. Такие 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 ввода-вывода программы. Результат сохраняется в *note*-файл, поэтому может быть использован для повторного воспроизведения команд в будущем.

Работа с note-файлами

- **:o name**
  Открывает файл и загружает данные. Новые блоки добавляются в конец к загруженным ранее.
- **:ls**
  Отображает список блоков команд.
- **:rm**
  Очищает список блоков.
- **:show N**
  Отображает содержимое N-ного или следующего по списку блока
- **: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**
  Оценивает 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 и число вызовов функций при вычислении заданного выражения.

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

1.8 Настройка программы

Конфигурация программы прописана в файле *sonata.lua*. Вы можете менять настройки файла внутри папки *sonata*, либо создать копию файла в удобном для доступа месте.

Настраивать можно 2 раздела: *MODULES* и *CONFIGURATION*. В первом разделе представлены модули, которые входят в *matlib* и могут быть загружены 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, соответствующие параметрам 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_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 в терминале может дополнительно потребоваться включение Unicode с помощью команды

  ```
  chcp 65001
  ```

II *data*: обработка списков данных

2.1 Введение

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. Также в данный модуль включены функции статистической обработки.

2.2 Операции с массивами

2.2.1 Объекты

Часть функций модуля являются конструкторами объектов, которые ведут себя как таблицы Lua, но являются либо генераторами, либо ссылками на другие таблицы.

Функция **range**(*начало*, *конец*, *[шаг]*) позволяет получить объект, который ведёт себя как таблица значений в заданном интервале, шаг по умолчанию равен 1. Если значение для конца больше, чем для начала, список будет построен в обратном порядке. При этом элементы списка нигде не хранятся, они 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 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”.

range() строит 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, возвращающий ссылку на диапазон данных списка. 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 формируется функцией **T**(*таблица*) и представляет собой ссылку на массив, где строки и столбцы поменялись местами, т.е. были “транспонированы”.

```
## 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 вызвать и для **range**(), и для строки транспонированной таблицы.

2.2.2 Фильтрация и преобразование данных

Можно 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 задана стандартным способом, либо с помощью метода **Fn**(*описание*, *число аргументов*), который генерирует функцию с заданным числом аргументов на основе строки описания. При этом аргументы должны иметь имена *x1*, *x2* и т.д. Если их число не превышает 2, 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 интерпретируется как “веса”, его можно использовать в некоторых статистических функциях модуля *data*, а также для фильтрации данных.

Выделить из списка определённые элементы позволяет функция **filter**(*список*, *условие*). Критерий отбора может быть задан списком весов равного размера (нули отбрасываются, остальное сохраняется), функцией, возвращающей *true* и *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 агрументов и формирует список результатов. Размер результата равен длине самого короткого из списков. Как и ранее, операция может быть задана в виде Lua функции или строки.

```
## zip = Bind(D, 'zip')
## c = zip("{x1-x2, x1+x2}", a, b)
## #c
2
## c[1]
{ -2, 4, }
```

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 **reduce**(*функция*, *список*, *[x0]*). Она последовательно применяет функцию двух аргументов к следующему элементу списка и предыдущему результату, и возвращает найденное значение. 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 это возможно. Запись файла выполняет 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 может быть использована функция **md**(*список*, *[заголовки*, *функция]*), которая выравнивает столбцы и использует тип представления Markdown для лучшего восприятия. Можно дополнительно задать список заголовков. Если требуется показать 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*) находят 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 в качестве опционального аргумента принимают список весов. Если таблица весов задана, но значение для некоторого элемента отсутствует, оно принимается равным 1. Таким образом, при пустой таблице учитываются все элементы 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*). Матрица ковариации для произвольного числа списков может быть найдена через **cov**(*таблица*), аргументом которой является таблица из исходных списков.

```
## p = D:range(1, 6)
## q = 1 - p
## D:cov2(p, q)
-2.9166666666667
## D:cov {p, q}
 2.917  -2.917
-2.917   2.917
```

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 сумм 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, }
```

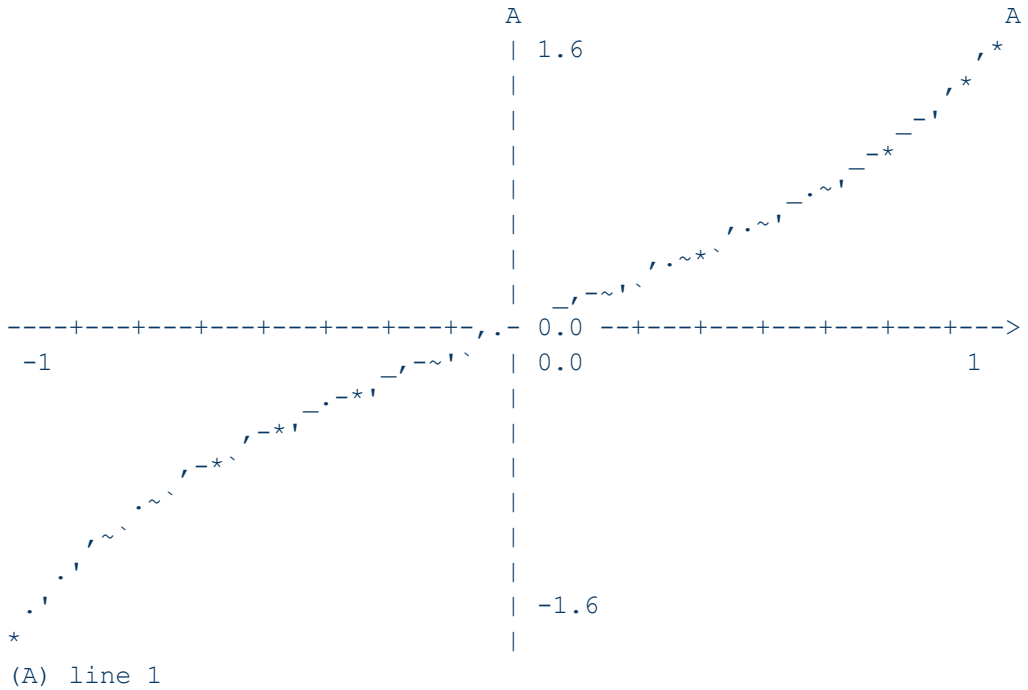
III *asciiplot*: визуализация в консоли

3.1 Введение

При работе в консоли без привлечения дополнительных библиотек единственной возможностью для визуализации является псевдо-графика. Модуль *asciiplot* предоставляет набор функций для создания таких диаграмм. Безусловно, псевдо-графика накладывает 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.

```
## 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 может быть настроен 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 *Ap.WIDTH*, *Ap.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 9 (эмпирический предел “читаемости” графика). Пропорционально увеличить или уменьшить холст позволяет функция **scale**(*коэффициент*), которая умножает ширину и высоту на заданное число. Если же аргументом является другой холст, то функция заимствует его размеры.

```
## 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 диапазона *range* 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. По умолчанию *false*.
- *view* - строка 'min'/'mid'/'max', которая указывает, в какой части холста рисовать ось (слева направо, снизу вверх). При *view=false* ось не отображается. По умолчанию ‘mid’.
- *log* - булевый флаг, равный *true*, если нужна логарифмическая шкала для данной оси. По умолчанию *false*.
- size - ширина оси (количестве символов).

Получить текущие 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 для удобства восприятия. Они отображаются тогда, когда ось делится на 2^n частей, поэтому цена деления равна $(x_{max} - x_{min})/2^n$. По той же причине пересечение осей не обязательно проходит через 0.

3.3 Виды графиков

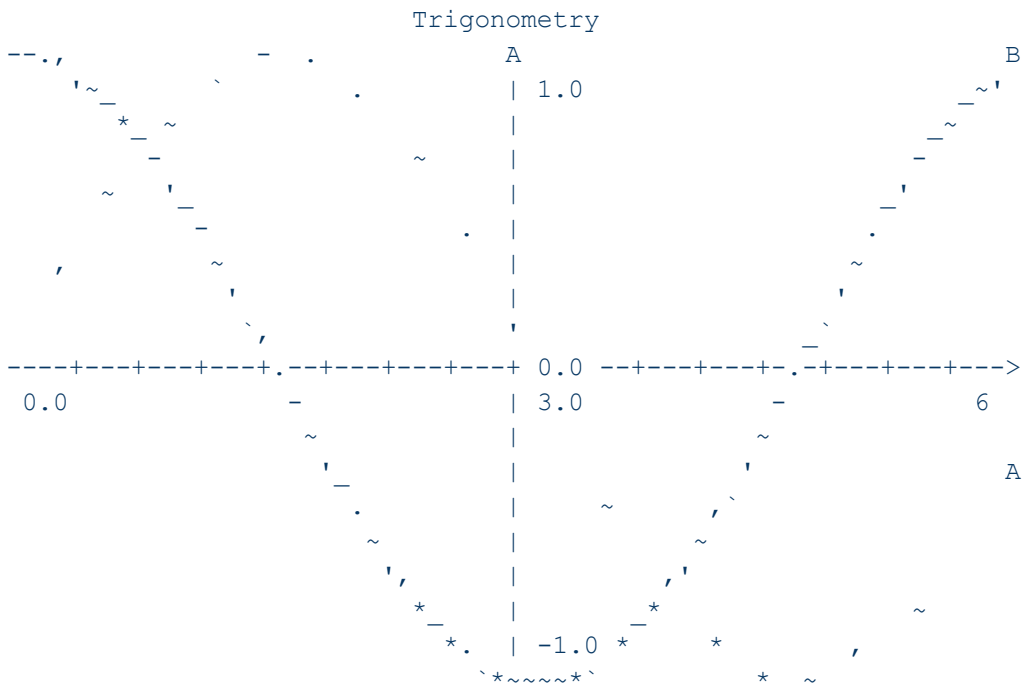
3.3.1 Функция plot

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:

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

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

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

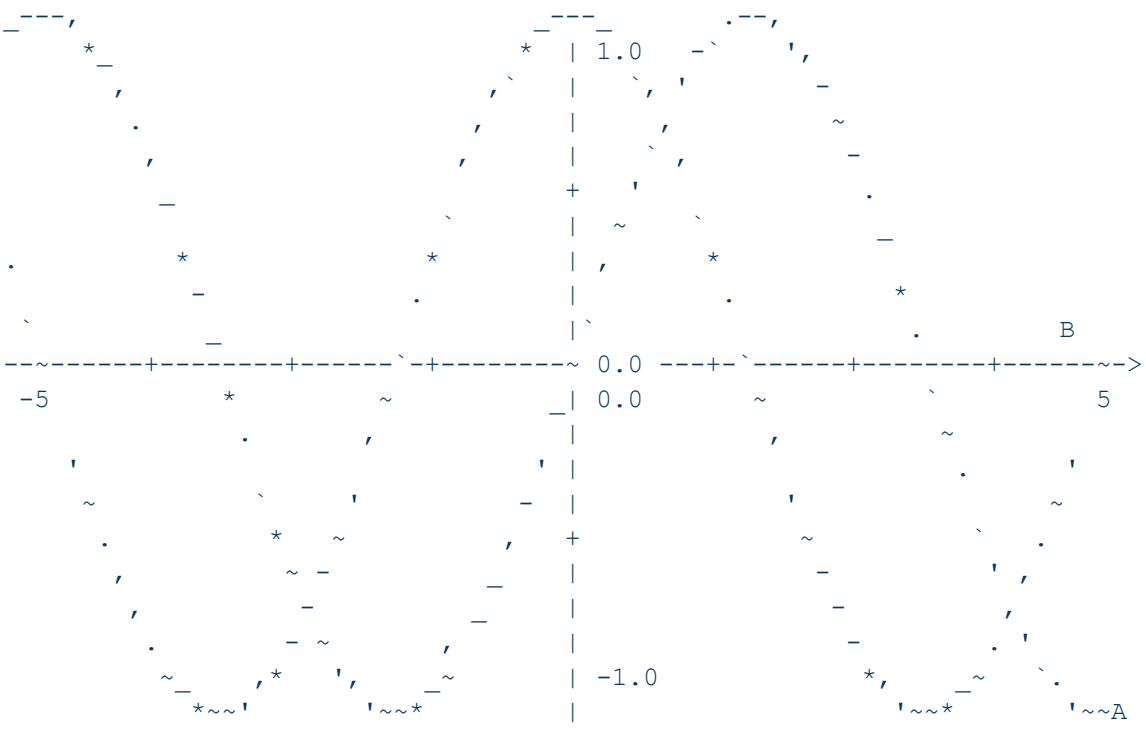



```
(A) sin
(B) cos
```

Функция **title**(*название*) устанавливает наименование графика. Также можно добавить или изменить легенду функцией **legend**(*список*), аргументом которой является таблица с именами кривых. Если нужно 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)
```



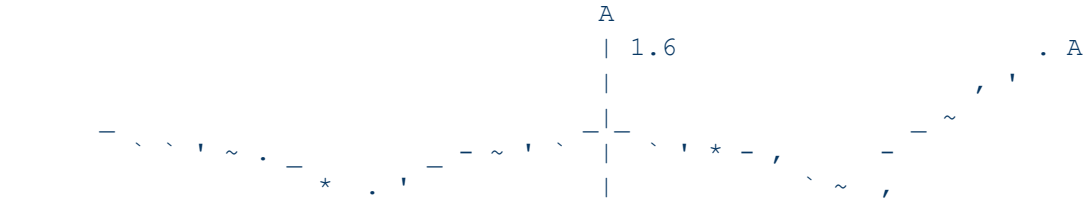

```
(A) line 1
(B) line 2
```

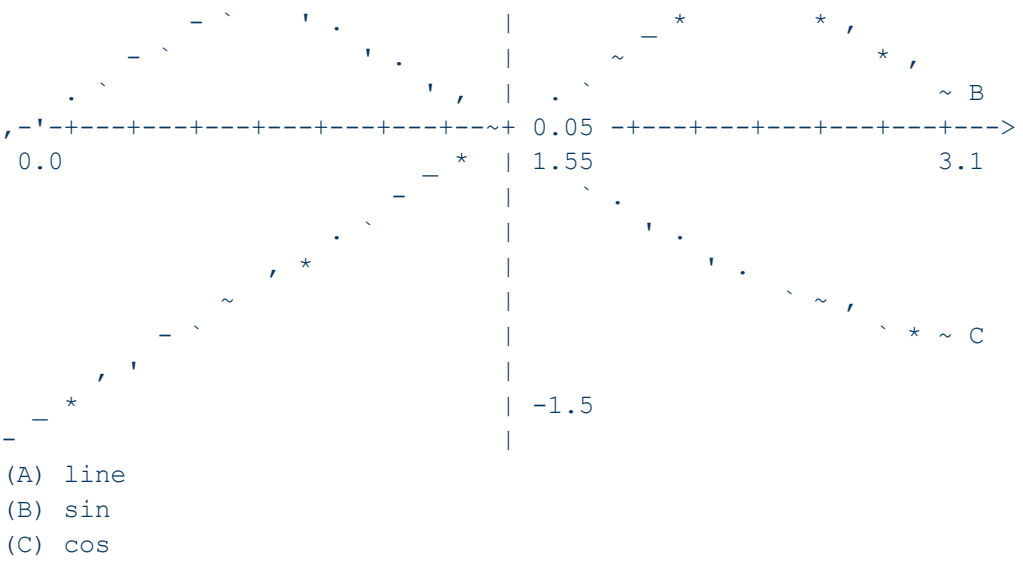
Как видно из графика, диапазон *x* для данной функции составляет от -5 до 5. Если нужно его изменить, придётся использовать списки.

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.

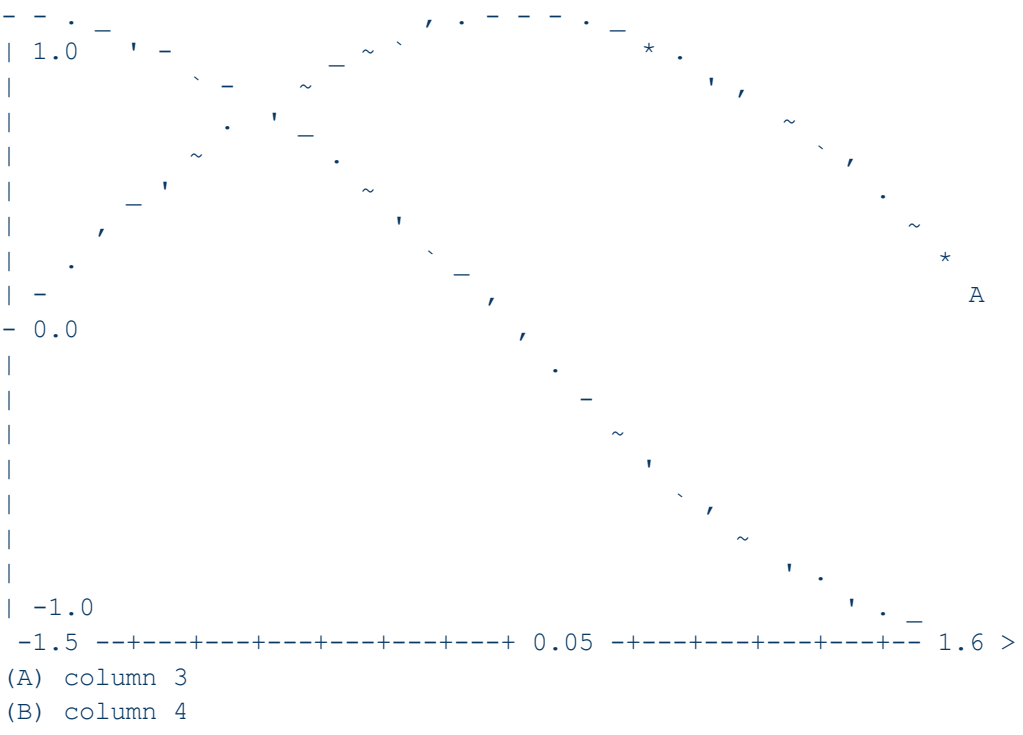
```
      ## tbl = {}
      ## for x = 0, 3.1, 0.1 do     \
      ..   tbl[#tbl+1] = {x, x-1.5, sin(x), cos(x)}  \
      .. end
      ## fig:tplot(tbl)
      ## fig:legend {'line', 'sin', 'cos'}
      ## fig
```





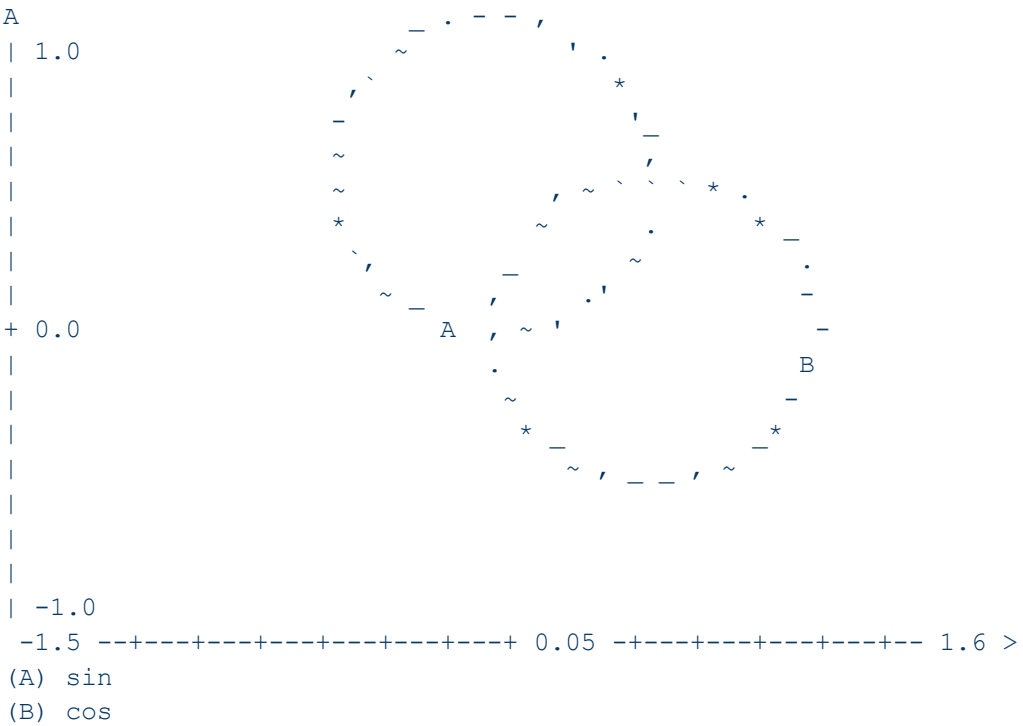
Функция **tplot**() строит графики для всех столбцов таблицы. Если требуется визуализировать отдельные столбцы, их номера нужно перечислить в таблице с опциями. В поле *x* можно указать индекс столбца с независимыми переменными.

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



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
```



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

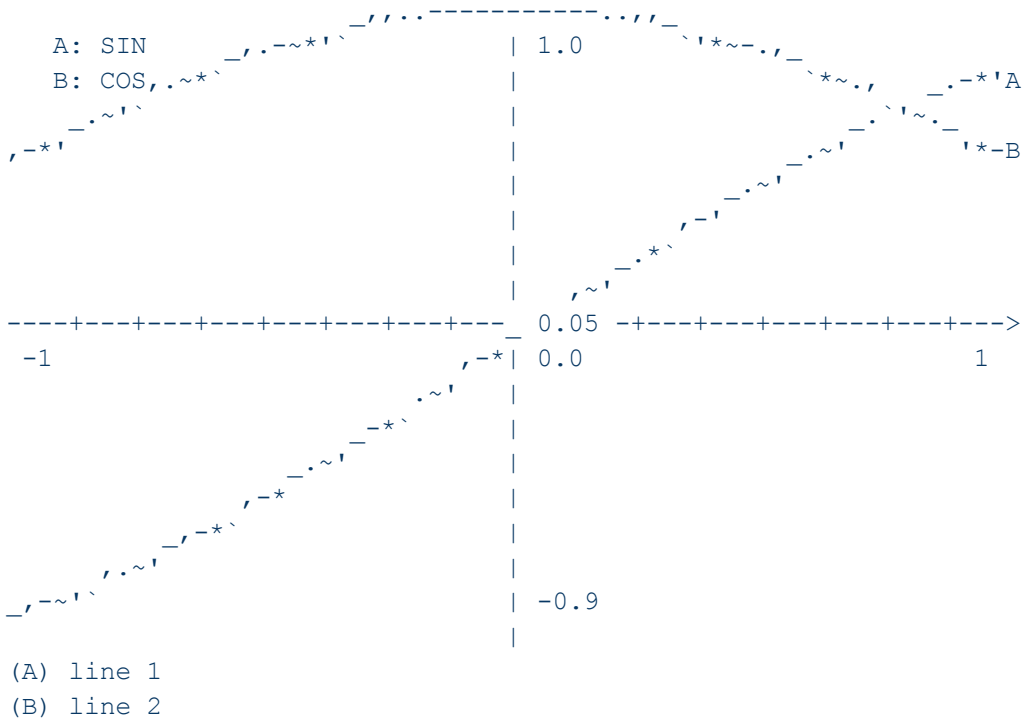
Ряд методов служат для манипуляций с созданными ранее графиками. Во-первых, можно получить полную копию с помощью **copy**(). Во-вторых, можно горизонтально конкатенировать графики функцией **concat**(*график1*, *график2*) или оператором '..'. При этом высоты обоих графиков должны совпадать. Результатом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

```
## fig1 = Ap()
## fig1: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
              | 0.9    ,-*'                     | 0.9    ,-*'
              |    _.*`                         |    _.*`
              | ,-'                             | ,-'
-------+-----,-'-----+------>   -------+-----,-'-----+------>
 -1       ,~' | 0.0       1      -1       ,~' | 0.0       1
      _.*`    |                         _.*`    |
```

```
  ,.~'        | -0.9              ,.~'        | -0.9
'`            |                 '`            |
(A) line 1                      (A) line 1
```

Можно наносить элементы на график вручную с помощью следующих функций: **addPoint**(*x*, *y*, *символ*) отмечает точку (*x*,*y*) заданным символом, **addPose**(*строка*, *столбец*, *символ*) ведёт себя аналогично, но помещает символ в определённую 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 модуле *matrix* 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.

```
## 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
```

Стандартная операция определения размера (“#”) для матрицы не задана, 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 позволяет метод **zeros**(*строки*, *столбцы*), если число столбцов не указано, строится квадратная матрица. При этом формируется пустая таблица, для неинициализированных элементов будет возвращаться 0. Чтобы построить матрицу, все элементы которой равны заданному числу, 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**(*элементы*, *смещение*) 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 определены основные арифметические операции: сложение, вычитание, умножение, возведение в целую степень, унарный минус. Также возможна проверка матриц на равенство и неравенство. Сложение и умножение со сколярными 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 матрица содержит один элемент, она преобразуется в число. 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. Псевдообратную 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 глубокую копию исходной матрицы.

Функция **T**() возвращает транспонированную матрицу (ссылку), а **H**() - транспонированную 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 отдельного метода, но данный функционал реализуется при вызове матрицы как функции, т.е. с использованием круглых скобок. При этом нужно передать в качестве аргументов два списка, обозначающих диапазоны строк и столбцов соответственно. В общем случае, диапазон содержит 3 числа: 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 индексы.

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
## c
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 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 большинство 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 "имени"
6
```

В отличие от предыдущих объектов данный тип не является матрицей, зато содержит ряд специфических методов, таких как векторное (**cross**) и скалярное (**dot**) произведение, нормализация 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

Преобразование Гаусса (функция **rref**()) 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 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
```

Таким образом **zip**() позволяет выполнить поэлементное произведение матриц 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 качественно оценить содержимое матрицы. Функция **stars**(критерий) вместо чисел выводит на печать пробелы и звёздочки исходя из заданного булевого условия. По умолчанию выполняется проверка на равенство нулю.

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

V *complex*: комплексные числа

5.1 Введение

Модуль *complex* содержит функции, необходимые для работы с комплексными числами. Конструктор имеет вид **Z**(*действительная_часть*, *мнимая_часть*), по умолчанию оба аргумента равны нулю. Также можно использовать полярную (экспоненциальную) форму представления $e^{i\alpha}$ и соответствующий ей конструктор **E**(*a*)

```
## Z(1, 2)
1+2i
## 2*Z:E(PI/4)
1.414+1.414i
```

Комплексные числа являются неизменяемыми, т.е. функции не модифицируют существующий объект, а создают новый.

5.2 Основные операции

Комплексные числа 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 модуле *polynomial* представляются списком коэффициентов от старшей степени к младшей. Т.е. выражение $x^2 + 2x + 1$ может быть представлено в виде **Poly** {1, 2, 1}, где **Poly** это конструктор. Однако, в полученном объекте коэффициенты располагаются таким образом, что индекс элемента соответствует его степени.

```
## 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 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. Первыми 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*) строит 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, её 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 в окрестности x_0 может быть рассчитано с помощью полинома Тейлора, который строит функция **taylor**(*x*0, *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 исходного множества точек полиномом заданной степени методом наименьших квадратов.

```
## 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 не требуется искать единую кривую, которая проходит через все заданные точки, можно разбить последовательность узлов на пары и соединить их кривыми по-отдельности. Функция **lin**(*xs*, *ys*, *[y0*, *yN]*) выполняет интерполяцию заданной зависимости прямыми. За пределами диапазона *xs* по умолчанию происходит экстраполяция полученными функциями, но можно использовать константные значения, если установить *y*0 и *y*N. Значение в заданной точке вычисляется так же как для обычного полинома, но возвращаются два числа, первое является результатом, второе - индексом полинома в списке. При вычислении можно указать этот индекс, 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)
3.625
```

VII *numeric*: численные методы

7.1 Введение

Модуль *numeric* представляет собой коллекцию функций для численного решения таких задач, как поиск корня уравнения, интегрирование функции и т.п. Работа модуля регулируется переменными:

- *numeric*.TOL - требуемая точность решения;
- *numeric*.SMALL - минимальный шаг некоторых алгоритмов;
- *numeric*.NEWTON_MAX - максимальное число итераций метода Ньютона;
- *numeric*.INT_MAX - максимальное число итераций интегрирования.

При загрузке модуля в окружение добавляется переменная *INF*, которая является ссылкой на *math.huge*.

7.2 Функции

7.2.1 Поиск корня

Поиск корня функции на заданном интервале [*a*, *b*] выполняет метод **solve**(*функция*, *a*, *b*), который возвращает найденное значение или возбуждает ошибку, если значения функции в крайних точка имеют одинаковый знак. Для поиска используется метод хорд. Для дифференцируемой функции решение может быть найдено методом Ньютона **newton**(*функция*, *x0*), при этом достаточно указать только одно начальное значение *x0*, желательно в окрестностях корня.

```
## -- корень sin(x) = 0, метод хорд
## Num:solve(sin, 0.5*PI, 1.5*PI)
3.1415926535898
## -- метод Ньютона
## Num:newton(sin, 0.8*PI)
3.1415927368139
```

Метод Ньютона может быть также использован для функции, которая принимает на вход вектор и возвращает скаляр.

```
## Num.TOL = 1E-4
## -- аргумент - вектор
## fn = function (v) return v(1)^2 + v(2)^2 end
## Num:newton(fn, Mat:V{1, 1})
4.09E-03
4.09E-03
```

7.2.2 Математический анализ

В этом разделе собраны функции, которые имеют отношение к интегральному и дифференциальному исчислению. Оценить величину предела

функции позволяет метод **lim**(*функция*, *предел_аргумента*, *[положительный]*), вторым аргументом является число *t*, к которому стремится независимая переменная. Опциональный третий аргумент положительный, если $x \to t_+$, и отрицательный когда $x \to t\_$. Предел может быть бесконечным (*math.huge*). При невозможности выполнить оценку выводится предупреждение.

```
## Num:lim(function (x) return sin(x)/x end, 0)
0.99999999999983
## Num:lim(exp, -INF)
0.0
## Num:lim(sqrt, 0)  -- по умолчанию 0-
Warning: limit: not found
-nan
## Num:lim(sqrt, 0, true)  -- теперь 0+
3.1622776601684e-05
```

Оценить величину производной в точке *x* позволяет метод **der**(*функция*, *x*).

```
## Num:der(sin, PI/4)
0.70709499613245
## Num:der(exp, 1.0)
2.7183271333827
```

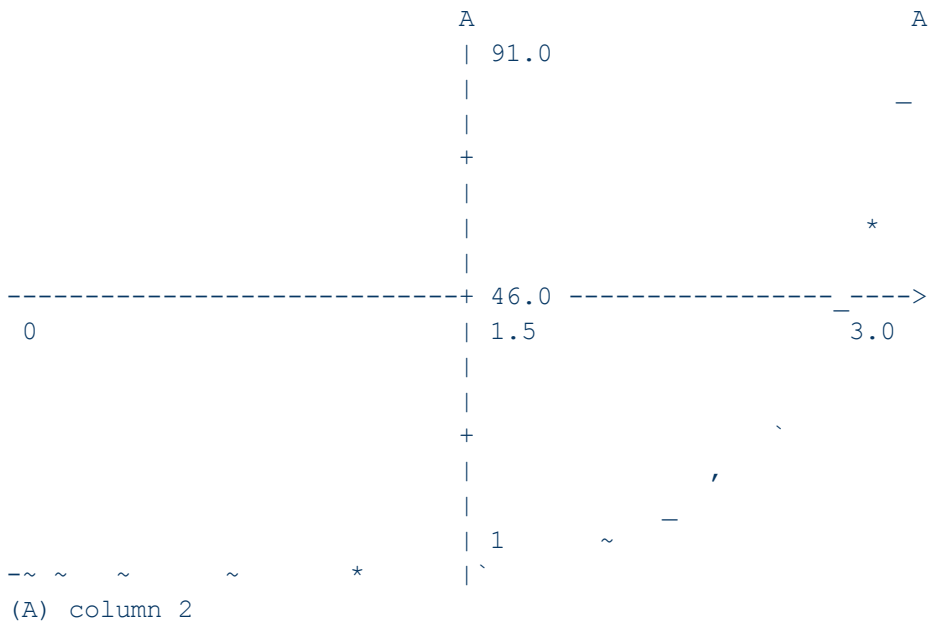
Найти определённый интеграл на интервале *[a, b]* можно с помощью метода **int**(*функция*, *a*, *b*). В ряде случаев метод работает для бесконечных пределов или когда функция в крайних точках не определена.

```
## Num:int(sin, 0, PI)
2.0000165910479
## fn = function (x) return 1/(1+x)/sqrt(x) end
## Num:int(fn, 0, INF)     -- пи
3.1394196640198
## fn = function (x) return exp(-x*x) end
## Num:int(fn, -INF, INF)  -- корень из пи
1.772391289196
```

7.3 Обыкновенные дифференциальные уравнения

Для численного интегрирования обыкновенных дифференциальных уравнений реализован метод **ode**(*функция*, *интервал t*, *начальное y*, *[опции]*), который принимает на вход помимо функции интервал интегрирования и значение в начальной точке этого интервала. Опции будут подробнее рассмотрены ниже. Метод возвращает список пар, где первый элемент - независимая переменная, второй - значение в точке.

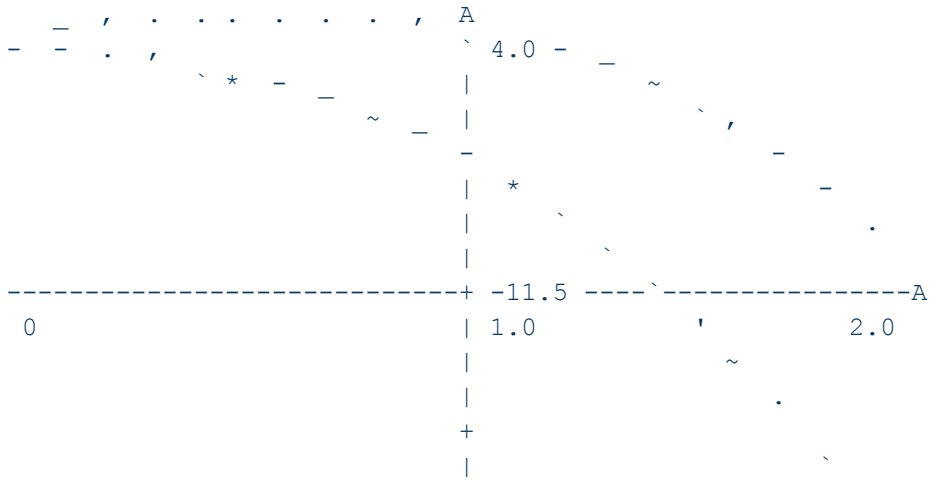
```
# -- уравнение y' = t*y, t in [0, 3], y0 = 1
## ys = Num:ode(function (t,y) return t*y end, {0, 3}, 1)
## fig = Ap()
## fig:tplot(ys)
## fig
```



Функция **ode**() по умолчанию изменяет шаг интегрирования для достижения требуемой точности. Если нужно выполнить расчёт с постоянным шагом, можно задать его в таблице опций, в поле *dt*. Условие прекращения расчёта можно задать в виде функции, которая получает на вход текущие промежуточные результаты и возвращает *true* для завершения. Эту функцию необходимо приравнять полю *exit*.

ode() может работать с функциями, входом и выходом которых являются векторы, т.е. интегрировать систему дифференциальных уравнений. Для примера рассмотрим уравнение второго порядка $y'' - 2y' + 2y = 1$, которое с учётом замены $x_1 = y, x_2 = y'$ может быть представлено в виде системы $x_1' = x_2, x_2' = 1 + 2\,x_2 - 2\,x_1$.

```
## fun = function (t, x) return Mat:V{x(2), 1+2*x(2)-2*x(1)} end
## -- расчёт с постоянным шагом 0.1
## xs = Num:ode(fun, {0, 2}, Mat:V{3,2}, {dt=0.1})
## -- преобразуем {t, Vec{x1, x2}} в {t, x1, x2}
## xs:flat()
## fig:tplot(xs)
## fig
```



```
                                |                              -
                                | -27.0
                                |                                 B
(A) column 2
(B) column 3
## -- прерывание по запросу
## fun = function (t, y) return -y end
## cond = function (ys) return ys[#ys][2] < 0.1 end  --  около нуля
## ys = Num:ode(fun, {0, 100}, 1, {exit=cond})
## ys[#ys]  -- последняя точка
{ 2.35, 0.095415291087094, }
```

В этом примере использована функция **flat**(), определенная специально для таблицы решений ОДУ. Данная функция в каждой строке заменяет вектор на список его элементов, что требуется для отрисовки через **tplot**().

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 для равномерного распределения. Функция **new**() создаёт новый генератор равномерно распределённых случайных чисел, не связанный с *math.random*. Такой генератор работает чуть медленнее, но дает возможность 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 результатов при разных вызовах. **seed**() без аргумента сбрасывает ядро до “случайного” значения.

```
## 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.

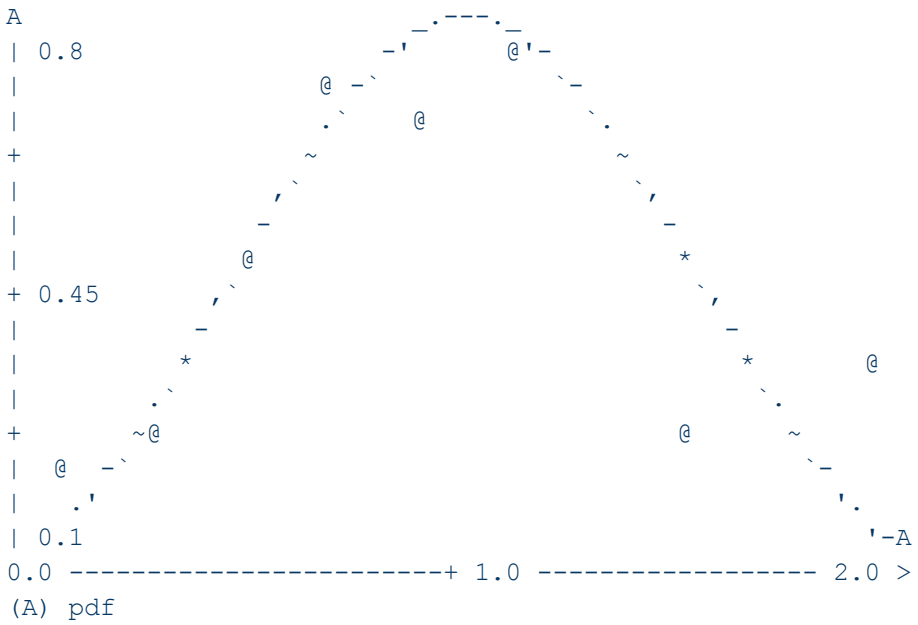
8.2 Распределения случайных чисел

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*) - равномерное распределение целых чисел.

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={lim[1], lim[#lim]}, 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
```



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 списка 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 в десятичной системе либо начинается с ‘0x’ и ‘0b’, основание системы счисления можно не писать, а числа записываются последовательно без разделителей. В общем случае основание помещается 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*, *повторы*), число сочетаний - **C**(*n*, *k*, *повторы*), в обоих случаях опциональный третий аргумент определяет, учитываются ли повторы. 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.

XI *quaternion*: операции с кватернионами

11.1 Введение

Кватернионы являются расширением идеи комплексных чисел на 4 измерения, они состоят из четырёх чисел, одно из которых является действительным, а три оставшихся - мнимыми. Модуль *quaternion* поддерживает базовые арифметические операции и функции, но основное внимание уделено использованию единичного кватерниона для представления ориентации в пространстве. В этом смысле мнимую часть можно рассматривать как направление оси вращения, а действительную - как меру поворота относительно этой оси.

Аргументами конструктора кватерниона **Quat** может быть список {*действительное w*, *мнимое x*, *мнимое y*, *мнимое z*}, словарь {*w = действительное w*, *x = мнимое x*, *y = мнимое y*, *z = мнимое z*} или их комбинация, пропущенные элементы замещаются нулём. Среди арифметических операций не определено деление, вместо него можно использовать умножение на обратный кватернион. Возведение в дробную степень задано только для единичных объектов. Получить единичный кватернион позволяет функция **normalized**().

```
## a = Quat {1, 2, 3, 4}
## b = Quat {w=5, y=2,}
## a + b
6+2i+5j+4k
## a - b
-4+2i+1j+4k
## a * b
-1+2i+17j+24k
## a^3
-86-52i-78j-104k
## c = a:normalized()  -- |c| равен 1
## c^0.5   -- sqrt
0.769+0.237i+0.356j+0.475k
```

Кватернион является неизменяемым объектом. Отдельные компоненты можно получить с помощью функций **x**(), **y**(), **z**(), **w**(). Результатом арифметических операций может быть обыкновенное число, если на каком-либо этапе мнимая часть будет равна нулю. Чтобы гарантированно получить кватернион, можно поместить результат расчёта в конструктор.

11.2 Функции

Модуль кватерниона можно найти с помощью функции **abs**(), сопряженное значение через **conj**(), обратное - **inv**(). Определены некоторые стандартные функции, такие как **exp**(), **log**(). Интерполировать значение между двумя

единичными кватернионами позволяет **slerp**(*Q1*, *Q2*, *часть*), где третьим аргументом является число от 0 до 1.

```
## a:abs()
5.4772255750517
## a:conj()
1-2i-3j-4k
## a * b:inv()
0.379+0.621i+0.448j+0.552k
## d = a:log()
## d:exp()
1.000+2i+3.000j+4k
## c:slerp(b:normalized(), 0.5)
0.670+0.220i+0.555j+0.441k
```

11.3 Ориентация

Единичный кватернион является одним из способов представления ориентации тела в пространстве, наряду с матрицами вращения, углами Эйлера и поворотом относительно заданной оси. Следующие функции определены для преобразования этих форм в кватернион и обратно: **toRot**() и **fromRot**(*матрица*), **toRPY**() и **fromRPY**(*крен*, *тангаж*, *рыскание*), **toAA**() и **fromAA**(*угол*, *ось*). Повернуть вектор с помощью кватерниона позволяет функция **rotate**(*вектор*).

```
## m = c:toRot()
## m
-0.667   0.133  0.733
 0.667  -0.333  0.667
 0.333   0.933  0.133
## Quat:fromRot(m)
0.183+0.365i+0.548j+0.730k
## r, p, y = c:toRPY()
## Quat:fromRPY(r, p, y)
0.183+0.365i+0.548j+0.730k
## angle, axis = c:toAA()
## axis
{ 0.3713906763541, 0.55708601453116, 0.74278135270821, }
## Quat:fromAA(angle, axis)
0.183+0.365i+0.548j+0.730k
## c:rotate {1, 0, 0}
{ -0.66666666666667, 0.66666666666667, 0.33333333333333, }
```

XII *special*: специальные функции

12.1 Введение

К специальный обычно относят функции, которые распространены в математической физике и не выражаются явно через элементарные функции. Часть из них реализована в модуле *special*.

12.2 Список функций

12.2.1 Гамма-функции

Гамма-функция может быть вычислена с помощью **gamma**(*x*), логарифм рассчитывается через **gammaln**(*x*). Неполная (нижняя) гамма-функция вызывается через **gammp**(*степень*, *x*), а сопряжённая с ней (верхняя) функция через **gammq**(*степень*, *x*).

```
## Spec:gamma(-1.5)
2.3632718012074
## Spec:gammaln(10)
12.801827480082
## Spec:gammp(3, 1.5)
0.19115316863236
## Spec:gammq(3, 1.5)
0.80884683136764
```

С неполными гамма-функциями связаны функция ошибки **erf**(*x*) и дополнительная функция ошибки **erfc**(*x*).

```
## Spec:erf(0.5)
0.52049990772324
## Spec:erfc(0.5)
0.47950009227676
```

12.2.2 Бета-функции

Значение бета-функции может быть найдено с помощью **beta**(*z1*, *z2*). Натуральный логарифм возвращает функция **betaln**(*z1*, *z2*). Неполная бета-функция рассчитывается с помощью **betainc**(*x*, *z1*, *z2*), где $x \in [0, 1]$ - верхний предел интегрирования.

```
## Spec:beta(3, 7)
0.0039682539682521
## Spec:betaln(3, 7)
-5.5294290875119
## Spec:betainc(0.5, 3, 7)
0.91015624999996
```

12.2.3 Функции Бесселя

Функции Бесселя первого (J) и второго (Y) рода рассчитываются с помощью **besselj**(*степень*, *x*) и **bessely**(*степень*, *x*). Соответствующие им модифицированные функции первого (I) и второго (K) рода определены в виде **besseli**(*степень*, *x*) и **besselk**(*степень*, *x*).

```
## Spec:besselj(2, 5)
0.046565118685798
## Spec:bessely(2, 5)
0.36766287898699
## Spec:besseli(2, 5)
17.505615012117
## Spec:besselk(2, 5)
0.0053089437352436
```

12.2.4 Другие функции

Интегральная показательная функция может быть вычислена с помощью **expint**(*степень*, *x*). Интеграл Доусона определяется через **dawson**(*x*).

```
## Spec:expint(2, 3.5)
0.0058018938267817
## Spec:dawson(2.5)
0.22308368068315
```

XIII *graph*: операции с графами

13.1 Введение

Модуль *graph* предназначен для работы с ориентированными и неориентированными графами без петель и кратных рёбер. Узлами могут быть произвольные объекты Lua, в то время как весами рёбер должны выступать числа. Пустой граф создаётся конструктором **Graph**(). С помощью дополнительных параметров, представленных в виде таблицы, можно определить тип и способ генерации графа. Список узлов возвращает метод **nodes**(), а рёбер - **edges**().

```
## g1 = Graph()
## g1
Graph {}
## g1:nodes()
{ }
## g1:edges()
{ }
## g2 = Graph {dir=true, C=3}   -- ориентированное кольцо
## g2:nodes()
{ n1, n2, n3, }
## g2:edges()
{ n1 -> n3, n2 -> n1, n3 -> n2, }
## g2
Digraph {n1,n2,n3}
## #g2  -- то же что g2:size(), число узлов
3
```

13.2 Генерация

Как было сказано выше, некоторые частные типы графов могут быть сгенерированы на основе параметров конструктора:

- O - граф без рёбер;
- K - полный граф;
- C - кольцо;
- P - цепь.

Для каждого типа можно задать число узлов либо таблицу имён. По умолчанию, узлы называются ‘n1’, ‘n2’, и т.д. При необходимости, общую часть имени можно заменить с помощью параметра *name*.

Для создания ориентированного графа необходимо установить флаг *dir*, который может сочетаться с другими параметрами.

```
## a = Graph {K=3}
## a:nodes()
{ n1, n2, n3, }
## a:edges()
{ n1 -- n2, n1 -- n3, n2 -- n3, }
```

```
## b = Graph {P=3, name='b'}
## b:nodes()
{ b1, b2, b3, }
## b:edges()
{ b2 -- b1, b3 -- b2, }
## c = Graph {O={'one', 'two', 'three'}}  -- имена узлов
## c:nodes()
{ one, two, three, }
```

Граф можно случайным образом заполнить ребрами с помощью функций **rand**(*число_ребер*) и **randp**(*вероятность*). В первой функции нужно указать желаемое число ребер, во второй - вероятность добавления ребра в процессе генерации. Обе функции предварительно удаляют найденные рёбра.

```
## d = Graph {O=4}
## d:rand(4)
## d:edges()
{ n1 -- n2, n1 -- n3, n4 -- n2, n2 -- n3, }
## d:randp(0.5)
## d:edges()
{ n1 -- n4, n4 -- n2, n2 -- n3, }
```

13.3 Основные действия

13.3.1 Модификация

Основной функцией для добавления элементов является **add**(*узел1*, *узел2*, *вес*). Если указан только первый аргумент, будет добавлен узел. Если заданы оба узла, добавится соответствующее им ребро, для которого можно указать вес с помощью третьего аргумента (по умолчанию 1). Эта же функция позволяет обновить вес. Удаление узла или ребра выполняет функция **remove**(*узел1*, *узел2*). Аналогично, один аргумент удаляет узел, два - ребро.

Для работы с группой элементов служат функции **addNodes**(*узлы*) и **addEdges**(*ребра*). Первый метод принимает на вход список узлов, второй - список рёбер, где каждое ребро представлено таблицей с именами соответствующих узлов и (опционально) весом.

```
## g = Graph()
## g:add 'a'
## g:add('b', 'c')
## g:add('c', 'd', 2)
## g
Graph {b,c,d,a}
## g:edges()
{ b -- c, c -- d, }
## g:remove('c','d')
## g:edges()
{ b -- c, }
## g:remove('b')
```

```
## g
Graph {c,d,a}
## g:addNodes {'x', 'y', 'z'}
## g:nodes()
{ x, a, z, c, d, y, }
## g:addEdges { \
.. {'a', 'x'},  \
.. {'b', 'y'},  \
.. {'c', 'z', 3}}
## g:edges()
{ b -- y, a -- x, c -- z, }
## g:nodes()
{ b, x, a, z, c, d, y, }
```

Узнать вес ребра можно с помощью функции **edge**(*ребро*), а проверить принадлежность узла графу через **has**(*узел*). Для ориентированного графа функция **nin**(*узел*) возвращает список узлов, соответствующих входящим рёбрам, **nout**(*узел*) - соответствующих исходящим, для неориентированного графа обе функции возвращают список смежных узлов.

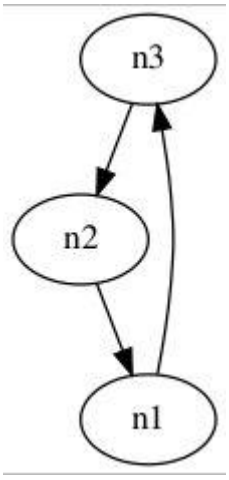
```
## g:has 'x'
true
## g:edge {'c', 'z'}
3
## g:nin 'a'
{ x, }
```

13.3.2 Экспорт

Для визуализации графов может быть использована внешняя программа Graphviz. Если она установлена и настроен вызов утилиты dot из консоли, можно сохранить граф в svg файл с помощью функции **toSvg**(*имя_файла*).

```
## g2:toSvg 'example'
```

В результате вызова будет сгенерирован файл *example.svg*.



Текстовое описание графа в нотации утилиты dot может быть сгенерировано функцией **dot**([*имя_файла*]). Если имя файла не указано, текст будет выведен на экран.

```
## g2:dot()
digraph {
  n3 -> n2;
  n1 -> n3;
  n2 -> n1;
}
```

13.3.3 Другие функции

Функция **matrix**() возвращает матрицу смежности графа и соответствующий строкам список узлов. Получить копию графа можно с помощью метода **copy**(). Объединить несколько графов в один позволяет функция **concat**(*список*), в случае двух объектов можно воспользоваться оператором конкатенации. Операция может быть применена только к графам одного типа (ориентированным или не ориентированным).

```
## m, ns = a:matrix()
## m
0  1  1
1  0  1
1  1  0
## ns
{ n3, n2, n1, }
## a1 = a:copy()
## a1 == a
true
## a == b
false
## Graph:concat{a,b}:edges()
{ b3 -- b2, b2 -- b1, n2 -- n3, n1 -- n2, n1 -- n3, }
## ab = a..b
## ab:edges()
{ b3 -- b2, b2 -- b1, n2 -- n3, n1 -- n2, n1 -- n3, }
```

Разложить граф на отдельные компоненты связности позволяет метод **components**(), который возвращает список найденных подграфов.

```
## ab:components()
{ Graph {b3,b2,b1}, Graph {n3,n2,n1}, }
```

13.4 Алгоритмы

13.4.1 Проверка свойств

Ряд методов предназначен для проверки особенностей графа: **isComplete**() - является ли полным, **isConnected**() - является ли связанным, **isDirected**() - является ли ориентированным, **isEuler**() - содержит ли эйлеров цикл, **isTree**() - является ли деревом, **isWeighted**() - содержит ли рёбра с весами, отличными от 1. Во всех случаях возвращается *true* или *false* в зависимости от результата проверки.

```
## a:isConnected()
true
## a:isEuler()
true
## a:isTree()
false
```

13.4.2 Поиск пути

Для поиска пути между двумя узлами служит функция **search**(*начало*, *конец*, *метод*). Метод представляет собой строку с наименованием алгоритма: 'bfs' - поиск в ширину, 'dfs' - поиск в глубину, 'dijkstra' - поиск кратчайшего пути алгоритмом Дейкстры.

```
## p = Graph {K=6}
## p:search('n1', 'n3', 'bfs')
{ n1, n3, }
## p:search('n1', 'n3', 'dfs')
{ n1, n5, n6, n3, }
-- рандомные веса от 1 до 10
## for _, e in ipairs(p:edges()) do \
..   p:add(e[1], e[2], math.random(1, 10)) end
## p:search('n1', 'n4', 'dijkstra')
{ n1, n6, n4, }
## p:edge {'n1', 'n4'}       -- прямой путь
8
## p:edge {'n1', 'n6'} + p:edge {'n6', 'n4'}  -- найденный путь
5
```

XIV *units*: единицы измерения

14.1 Введение

Модуль *units* предназначен для проведения расчётов с учётом единиц измерений и преобразования одних единиц в другие на основе заложенных правил. Создать объект можно с помощью конструктора **U**(*величина*, *единица_измерения*). Если первый аргумент отсутствует, он считается равным единице. В случае безразмерной величины вторым аргументом можно указать пустую строку. Для объектов определены основные арифметические операции и правила сравнения. При этом следует помнить, что складывать, вычитать и сравнивать можно только величины, единицы измерения которых совпадают или могут быть приведены друг к другу.

```
## a = U(2, 'm')
## b = 1 * U's'  -- то же что 1*U('s') или U(1, 's')
## a / b
2 m/s
## a * b
2 m*s
## 0.5 * a / b^2
1 m/s^2
## a > b
ERROR ./matlib/units.lua:255: Different units!
```

14.2 Преобразование единиц измерения

14.2.1 Список правил

Правила преобразования в модуле *units* можно разделить на неявные и явные. К первым относятся преобразования на основе дольных и кратных приставок, которые определены в таблице **U.prefix**. По умолчанию это стандартные приставки, используемые в науке (кило-, микро- и т.п.). Если при сравнении двух единиц измерения программа обнаруживает, что они отличаются известной ей приставкой, то она домножает один из объектов для приведения к общей размерности.

Явные преобразования выполняются на основе правил, хранящихся в таблице **U.rules**. Правило представляет собой выражение типа

```
1 'A' = x 'B'
```

которое записывается в виде

```
U.rules['A'] = U(x, 'B')
```

При этом величина 'A' должна быть простой единицей измерения, не содержащей арифметических операций.

Список правил и приставок можно отобразить с помощью функции **print**().

14.2.2 Функции

Чтобы определить численное значение при заданных единицах измерения, нужно обратиться к объекту *units* как к функции, передав в качестве аргумента строку с требуемой размерностью. Если аргумент опущен, будет возвращено текущее значение. Единицы измерения могут быть получены с помощью функции **u**().

```
## a = U(2, 'm/s')
## a()  -- текущее значение
2
## a:u()  -- текущие единицы измерения
m/s
## -- чтобы получить км/ч определим 2 промежуточных преобразования
## U.rules['h'] = 60 * U'min'  -- U(60, 'min')
## U.rules['min'] = 60 * U's'  -- U(60, 's')
## a 'km/h'  -- то же что a('km/h')
7.2
## a 'm/min'
120.0
## a 'dm/ms'
0.02
```

Функция **convert**(*единицы*) возвращает новый объект *units*, а не только численное значение. Создать копию можно с помощью метода **copy**().

```
## b = a:convert('km/h')
## b
7.200 km/h
## c = b:copy()
## c == a
true
```

XV *const*: коллекция констант

15.1 Введение

Модуль *const* представляет собой коллекцию постоянных величин, используемых в различных областях науки (на данный момент, преимущественно в физике). Получить значение можно с помощью точечной нотации, с указанием имени раздела. Как и положено константам, эти величины неизменяемые.

```
## use 'C'
## C.phy.G -- гравитационная постоянная
6.672041e-11
## C.math.phi
1.6180339887499
## C.math.phi = 123
ERROR: ./matlib/const.lua:55: Constants are immutable!
```

Физические величины обычно имеют единицы измерения. Представленные в данном модуле константы могут быть преобразованы в объекты *units*, для этого достаточно к имени добавить суффикс '_U'.

```
## e = C.phy.e_U  --  заряд электрона
## e
1.60E-19 C
## e 'nC'  --  нКл
1.602189246e-10
```

15.2 Пользовательские константы

Для постоянного использования лучше добавить константы в соответствующие таблицы файла *matlib/const.lua*. Но если требуется получить неизменяемый объект во время работы с программой, можно воспользоваться функциями **add**(*имя*, *значение*, *[единицы]*) и **remove**(*имя*). Первая из них добавляет константу в список, вторая удаляет.

```
## C:add('aa', 4)
## C:add('bb', 5, 'm/s')
## C.aa
4
## C.bb
5
## C.bb_U
5 m/s
## C.aa = 2
ERROR: ./matlib/const.lua:55: Constants are immutable!
## C:remove('aa')
true
```

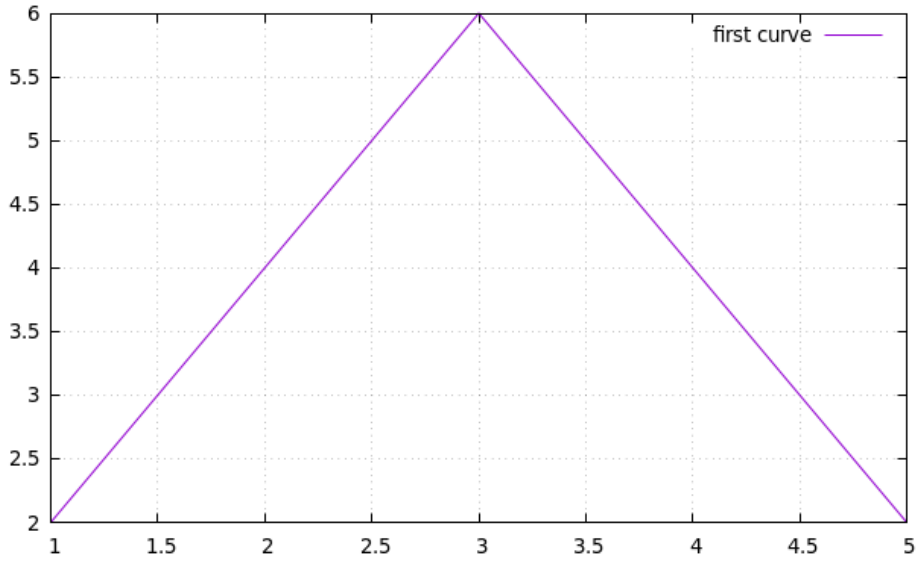
XVI *gnuplot*: построение графиков в GNUplot

16.1 Введение

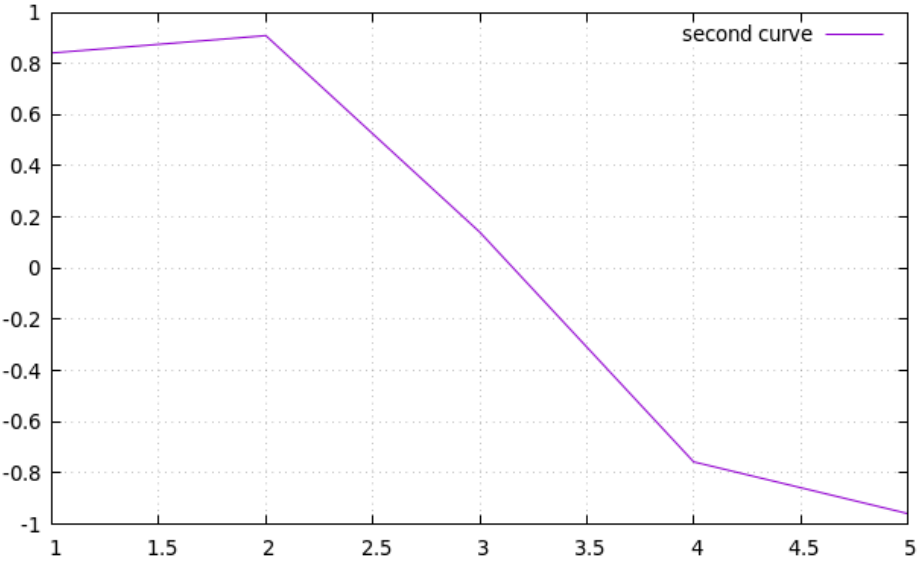
Для визуализации данных могут быть использованы внешние графические инструменты. Одним из них является Gnuplot. Для работы с этой программой реализован одноимённый модуль.

Объект “холста” формируется с помощью конструктора **Gp**(), однако, большая часть функций может быть вызвана без привязки к конкретному объекту.

```
## t1 = {1, 2, 3, 4, 5}
## t2 = {2, 4, 6, 4, 2}
## Gp:plot(t1, t2, 'first curve')
```




```
## Gp:plot(t1, sin, 'second curve')
```



16.2 Основные функции

Функция **plot**() была представлена в предыдущем разделе. Её аргументами являются последовательности вида: *список x [список y или функция] [имя]*, выражения в квадратных скобках могут быть опущены.

```
## t3 = D:range(0, 2*PI, 0.05)
## Gp:plot(t3, sin, 'sin', t3, cos, 'cos')
```

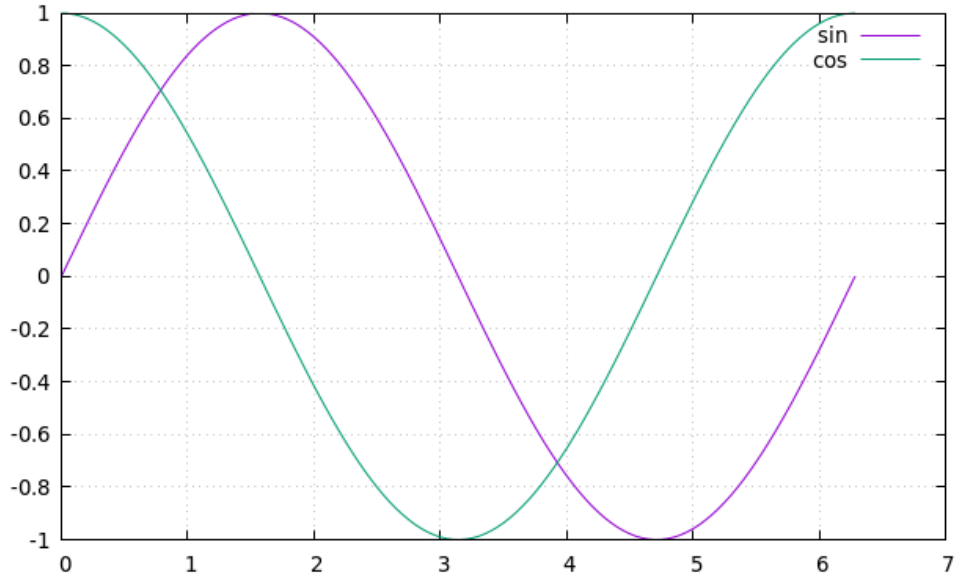
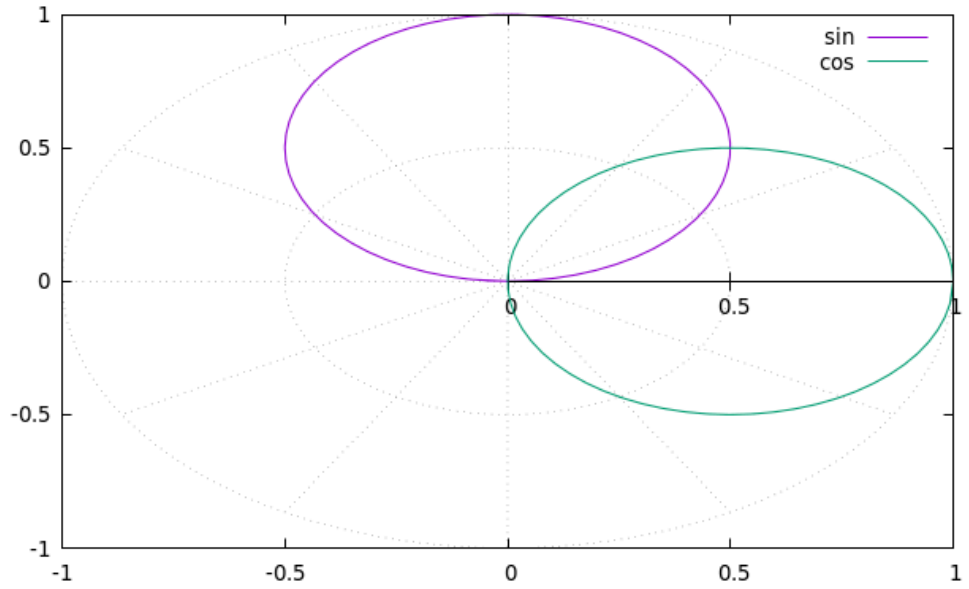


График в полярных координатах строит функция **polarplot**(), её сигнатура аналогична **plot**().

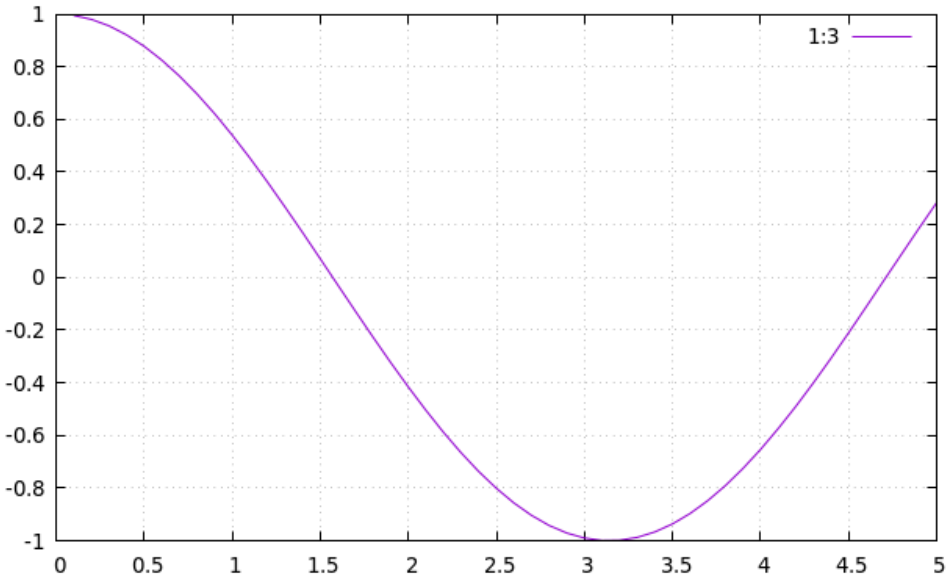
```
## Gp:polarplot(t3, sin, 'sin', t3, cos, 'cos')
```



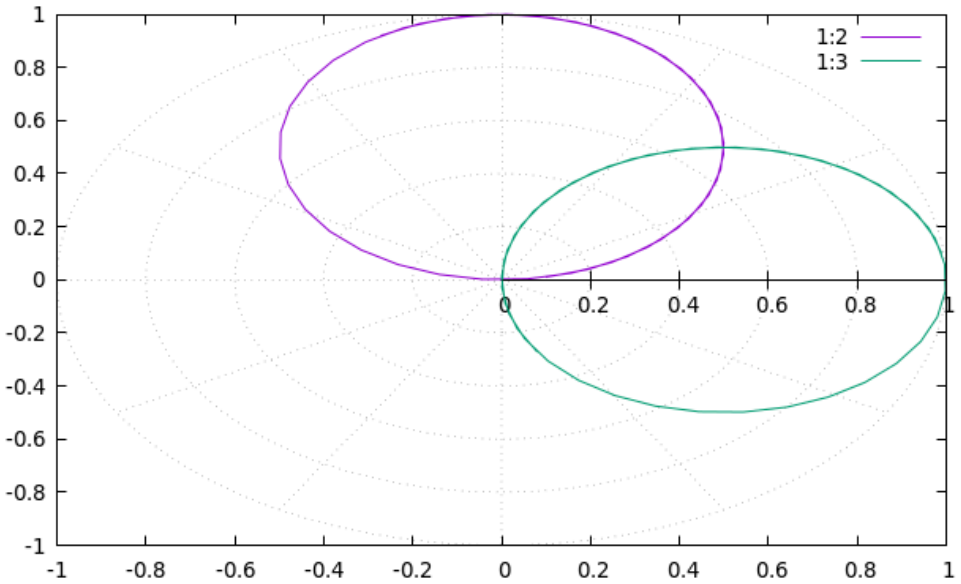
Для визуализации табличных данных служит функция **tplot**(*массив, [индекс x, индекс y1, индекс y2...]*). Если индексы столбцов явно не заданы, отображаются все данные, при этом независимой переменной считается первый столбец.

Аналогичная функция, **tpolar**(), строит график для табличных данных в полярных координатах.

```
## arr = {}
## for i = 1, 50 do  \\
..   local x = 0.1*i
..   arr[i] = {x, sin(x), cos(x)}
.. end
..
## Gp:tplot(arr, 1, 3)
```



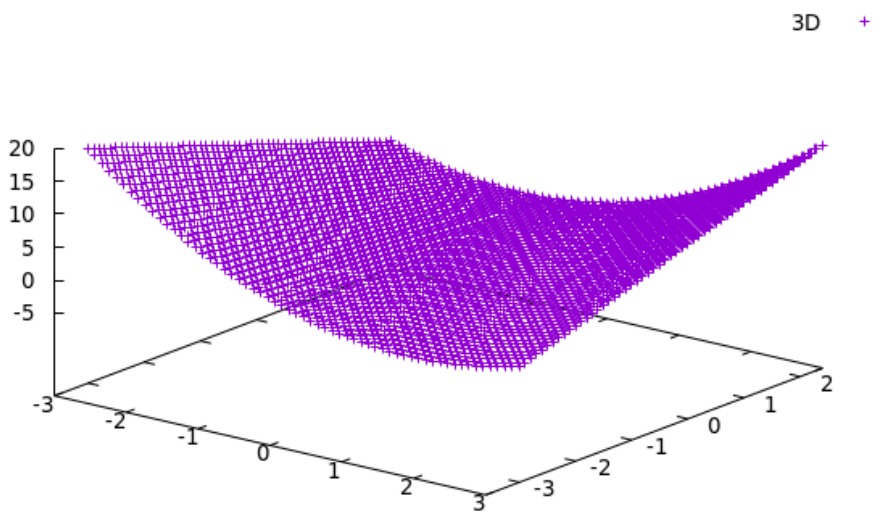

```
## Gp:tpolar(arr)
```



Для построения поверхности служит функция **surfplot**(*список x, список y, функция от (x, y)*). Точки поверхности могут быть заданы таблично, в этом случае для визуализации можно использовать **tsurf**(*таблица*).

```
## function fun(x,y) return x*x + y*x end
## t4 = D:range(-3, 3, 0.1)
```

```
## Gp:surfplot(t4, t4, fun, '3D')
```



16.3 Построение через вызов объекта

Описанные выше функции используют настройки по умолчанию. Если требуется больший контроль над графиком, можно настроить объект, который формирует конструктор **Gp**(). Основные параметры:

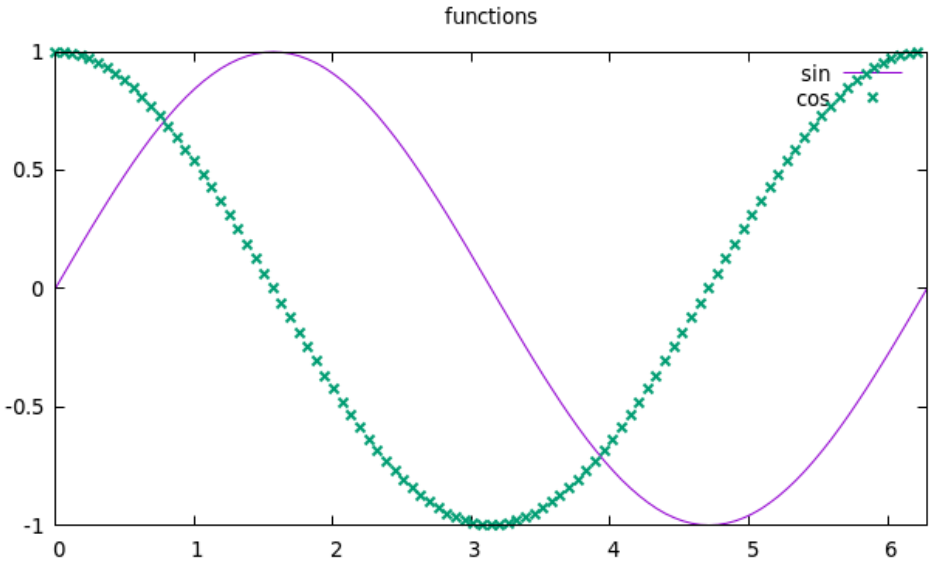
- *title* - заголовок;
- *xlabel*, *ylabel* - названия осей;
- *legend* - отобразить/скрыть легенду;
- *terminal*, *output* - настройка вывода данных;
- *raw* - выполнить “сырую” команду Gnuplot.

Для линии могут быть заданы:

- *title* - название;
- *with* - способ отображения;
- *linetype*, *linestyle*, *linewidth* - параметры линии.

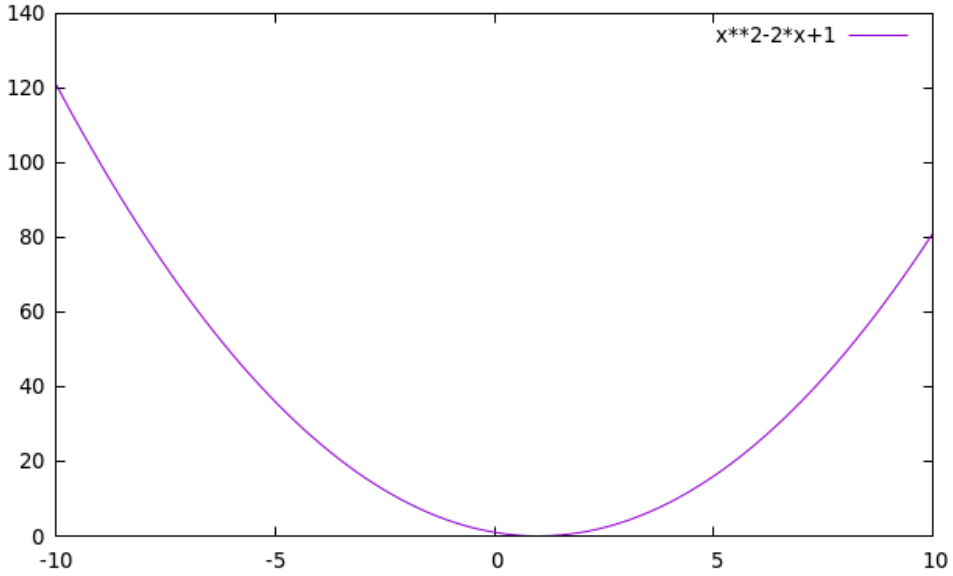
Добавить график позволяет функция **add**(*данные*), её аргументом является таблица с настройками типа линии. Результат отображает функция **show**(). Скопировать объект со всеми настройками можно с помощью функции **copy**().

```
## fig = Gp()
## fig.xrange = {0, 2*PI}
## fig:add {sin, title='sin', with='lines'}
## fig:add {cos, title='cos', lw=2}
## fig.title = 'functions'
## fig:show()
```



Если вы знакомы с синтаксисом Gnuplot, можно указать команду в явном виде.

```
## fig = Gp()
## fig.raw = [[plot x**2-2*x+1; set xlabel "X"; set ylabel "Y"]]
## fig:show()
```



XVII *symbolic*: элементы компьютерной алгебры

17.1 Введение

Модуль *symbolic* дает возможность выполнять манипуляции с символьными математическими выражениями. Хотя возможности данного модуля не идут ни в какое сравнение с такими программами как, например, Maxima, он позволяет выполнять некоторые упрощения и вычислять производные.

Символьную переменную или константу можно создать с помощью конструктора **Sym**(*переменная*). Функция **parse**(*строка*) формирует символьное выражение из строки, при этом можно определить несколько выражений, разделённых запятыми.

```
## a = Sym 'a'
## b = Sym(42)
## a * b
42*a
## p, q = Sym:parse('x-y, x+y')
## p / q
(x-y)/(x+y)
## p + q
2*x
```

17.2 Основные операции

Для символьных переменных определены операции сложения, вычитания, умножения, деления и возведения в степень. Вычислить выражение при некоторых значениях переменных можно с помощью метода **eval**(*значения*), аргументом которого является таблица пар *имя = величина*. Метод **eval**() пытается упростить полученное выражение и может быть вызван без аргументов. Если результатом является константа, преобразовать её в обычное число можно с помощью **val**().

```
## qe = q:eval {x=2, y=3}
## help(qe)
<symbolic>
5
## help(qe:val())
<number>
5
## p:eval {x=1}
1-y
```

Символьную функцию можно определить в виде **def**(*имя*, *аргументы*, *тело*), где аргументы представляют собой список имен или символов, тело это

выражение, которое вычисляется и возвращается при вызове. Аргументы могут быть переменными или именами.

```
## foo = Sym:def('foo', {'x', 'y'}, \
.. Sym:parse('x^2 + y^2'))  -- возвращает
## foo
foo(x,y): x^2+y^2
## foo(p, q)
(x-y)^2+(x+y)^2
```

Стандартные и пользовательские функции могут быть получены через обращение к таблице **fn**. Проверить, является ли символ функцией, можно с помощью методы **isFn**().

```
## Sym.fn.foo
foo(x,y): x^2+y^2
## foo:isFn()
true
```

Раскрыть скобки позволяет метод **expand**(). Вычислить производную по заданной переменной позволяет **diff**(*переменная*), аргументом может быть символьный объект или строка.

```
## (p*q):expand()
x^2-y^2
## Sym:parse('x^2*sin(x/2)'):diff('x')
0.5*x^2*cos(x/2)+2.0*x*sin(x/2)
```

Если выражение является рациональным, выделить числитель и знаменатель можно с помощью **ratNum**() и **ratDenom**() соответственно. Увидеть внутреннее представление символьного объекта позволяет метод **struct**().

```
## r = p / q
## r
(x-y)/(x+y)
## r:ratNum()
x-y
## r:ratDenom()
x+y
```

XVIII *qubit*: эмуляция квантовых вычислений

18.1 Введение

Модуль *qubit* позволяет эмулировать кубиты и гейты (квантовые вентили), на основании которых могут быть реализованы алгоритмы квантовых вычислений. Однако, следует помнить, что эти расчёты выполняются на обычном компьютере, поэтому время вычисления будет расти экспоненциально с увеличением числа кубитов.

18.2 Кубиты

Кубит можно задать в виде суммы состояний в базисе “|0>” и “|1>”. Для этого служит конструктор **Qb**(*строка_состояния*), который умножается на комплексный (или действительный) коэффициент. С вычислительной точки зрения состояние описывается вектором, модуль которого равен единице. Учитывая произвольный характер начальных коэффициентов, результат можно нормализовать функцией **normalize**(). При записи состояния символы “|>” можно опускать.

Вероятность конкретного состояния возвращает метод **prob**(). Функция измерения **meas**(*кубит*) снимает неопределенность, при этом “квантовая” система переходит в одно из возможных состояний. Можно “измерить” состояние отдельного кубита, при этом неопределённость остальных элементов сохранится.

```
## q = 0.4*Qb'|0>' + 0.6*Qb'|1>'
## a:normalize()
## q
(0.555)|0> + (0.832)|1>
## q:prob '|0>'
0.30769230769231  -- квадрат амплитуды
## q:prob '|1>'
0.69230769230769
## q:meas()  -- измерение
|1>
## q:prob '1'
1
## q:prob '0'
0
```

Если передать конструктору число кубитов вместо строки состояния, будет возвращена система в случайном начальном состоянии. Для получения проекции на заданное состояние используется оператор произведения.

```
## q2 = Qb(1)
## q2
(0.715-0.369i)|0> + (0.193-0.562i)|1>
## k = 1 / sqrt(2)
## state = {['+'] = k*Qb'0'+k*Qb'1', ['-'] = k*Qb'0'-k*Qb'1'}
```

```
## state['+'] * q2
0.642-0.658i
## state['-'] * q2
0.369+0.136i
## q2 * q2
1.0
```

Систему из нескольких кубитов можно задать с помощью конструктора, либо получить путём объединения более простых систем функцией **combine**(*Q1*, *Q2*, ...), для двух элементов определена операция конкатенации.

```
## Qb'00' + Qb'10' - Qb'01'  -- нужна нормализация
(1)|00> + (-1)|01> + (1)|10>
## state['+'] .. state['-']
(0.500)|00> + (-0.500)|01> + (0.500)|10> + (-0.500)|11>
```

Создать копию кубита можно с помощью метода **copy**(), а получить представление в виде матрицы (вектора) с помощью **matrix**().

```
## q3 = q2:copy()
## q3:matrix()
0.715-0.369i
0.193-0.562i
```

18.3 Гейты

Гейты представляют собой преобразования, применяемые к квантовой системе. В этом смысле их можно рассматривать как функции (точнее, операторы), на вход которых подаётся система кубитов. С вычислительной точки зрения гейты формируют матрицу, определитель которой равен единице.

Для инициализации гейтов необходимо вызвать функцию **gates**(*входы*), и указать ей требуемое число входов. Затем последовательность преобразований записывается слева направо, с использованием стандартных или заданных пользователем элементов. Гейты Паули (**X**(), **Y**(), **Z**()), Адамара **H**(), а также **S**() и **T**(), применяются только к тем входам, номера которых указаны в списке аргументов, индексация входов с 0. Если список пустой, эти гейты применяются ко всем входам. Гейт **SWAP**(*n1*, *n2*) меняет местами указанные выходы, **CNOT**(*n1*, *n2*) применяет операцию NOT к входу *n1* если *n2* в состоянии |1>. Обратное преобразование можно получить с помощью метода **inverse**().

```
## g1 = Qb:gates(3):X(0,2):Y(1):Z()
## g1
|x2>  X - Z
|x1>  - Y Z
|x0>  X - Z
## g1 = g1:CNOT(1,2):SWAP(0,1) -- гейт можно "наращивать"
## g1
|x2>  X - Z o -
|x1>  - Y Z x \
```

```
|x0>  X - Z - /
## g1(Qb'001')  -- применяем к состоянию |001>
|100>
## g2 = g1:inverse()  -- обратное преобразование
## g2
|x2> - o Z - X
|x1> \ x Z Y -
|x0> / - Z - X
## g2(Qb'100')
|001>
```

Пользователь может создать собственные гейты двумя способами. Если известна единичная матрица преобразований, можно воспользоваться методом **fromMatrix**(*матрица*). Если же известна таблица истинности, её можно передать на вход функции **fromTable**(*таблица*), при этом каждому входному состоянию должно соответствовать единственное выходное состояние, и наоборот. Проверить, является ли гейт унитарным, позволяет метод **isUnitary**().

```
## t = {         \\
.. {'00', '01'},
.. {'01', '00'},
.. {'10', '11'},
.. {'11', '10'}}
..
## g3 = Qb:gates(2):fromTable(t)
## g3
|x1>  U
|x0>  U
## g3:isUnitary()
true
## g3(Qb'01')
|00>
## m = Mat {     \\
.. {1, 0, 0, 0},
.. {0, 0, 1, 0},
.. {0, 1, 0, 0},
.. {0, 0, 0, 1}}
..
## m:det()
-1
## g4 = Qb:gates(2):fromMatrix(m)
## g4(Qb'01')
|10>
```

Для гейтов также определены операции умножения и возведения в степень. Первая позволяет позволяет последовательно соединить два преобразования, вторая - повторить гейт заданное число раз. Матричное представление может быть получено с помощью метода **matrix**().

```
## g5 = g3 * g4^3
## g5
|x1>  U U U U
```

```
|x0>  U U U U
## g5:matrix()
0  1  0  0
0  0  0  1
1  0  0  0
0  0  1  0
```

18.4 Пример

В качестве примера рассмотрим алгоритм Гровера, который решает задачу поиска аргумента, при котором функция возвращает 1 (при остальных допустимых значениях возвращается 0).

В качестве оракула возьмем функцию трех кубитов, которая выдает единицу для состояния ‘101’. Этому условию соответствует матрица, у которой все диагональные элементы 1, кроме строки 6 (бинарное 101 равно 5, но индексация в Lua начинается с 1, поэтому 6). Гейт диффузии Гровера также можно определить в матричном виде, здесь все диагональные элементы равны -1, кроме первого. Объединим оракл с гейтом диффузии.

```
## -- матрицы
## mat_fn = Mat:eye(8)
## mat_fn[6][6] = -1
## mat_df = -Mat:eye(8)
## mat_df[1][1] = 1
## -- оракл + гейт диффузии
## oracle_diffuse = Qb:gates(3) \\
..  :fromMatrix(mat_fn)
..  :H()
..  :fromMatrix(mat_df)
..  :H()
..
```

Равновероятную суперпозицию состояний входа можно получить с помощью гейта Адамара. Вызов записанного выше гейта нужно повторить по крайней мере дважды ($\sqrt{2^3}\frac{\pi}{4} \approx 2$). После этого можно подать в систему начальное состояние ‘000’ и выполнить измерение. Следует помнить, что алгоритм вероятностный и иногда может возвращать неверный результат.

```
## grov = Qb:gates(3):H() * oracle_diffuse^2
## grov(Qb'000'):meas()
|101>
```

XIX *lens*: параксиальная оптика

19.1 Введение

Параксиальными называют лучи, идущие под малым углом к оптической оси (обычно до 5 градусов). Для таких лучей $sin(\alpha) \approx tan(\alpha) \approx \alpha$, что позволяет линеаризовать систему и описать трансформацию луча с помощью матриц.

В основе модуля *lens* лежат методы, описанные в книге “Введение в матричную оптику”, А. Джеррард, Дж. М. Бёрч. Каждое преобразование описывается единичной матрицей {{A, B}, {C, D}}, преобразование системы в целом определяется произведением этих матриц. Состояние луча задаётся его смещением относительно оптической оси y и оптическим углом $V = \alpha \cdot n$, равным произведению геометрического угла на показатель преломления среды.

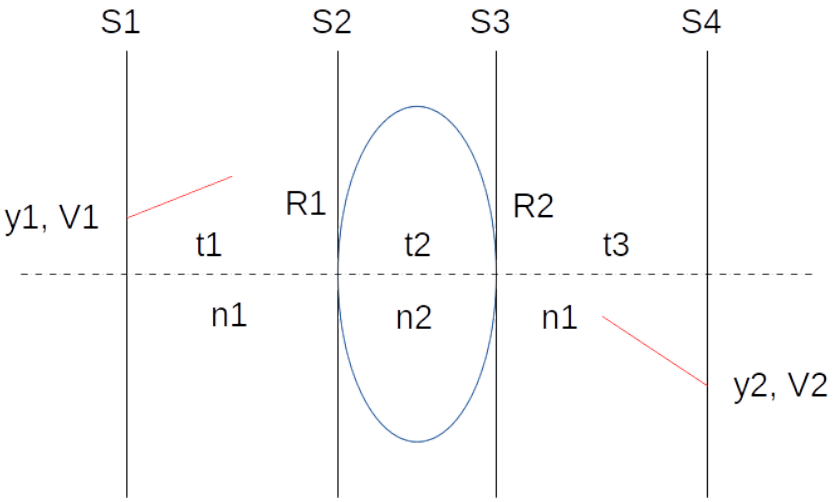
19.2 Оптические системы

19.2.1 Описание

Чтобы описать оптическую систему в терминах матричной оптики, необходимо задать опорные плоскости и определить преобразования между ними. К элементарным преобразованиям относятся перемещение **T**(*расстояние*, *n*), преломление **R**(*n_входное*, *радиус*, *n_выходное*) и отражение **M**(*радиус*, *n*) которые задаются с учётом показателя преломления среды *n*. Также определены матрицы для афокальной системы **afocal**(*увеличение*) и тонкой линзы **thin**(*фокусное_расстояние*). Произвольную матрицу можно создать с помощью конструктора **Lens**(A, B, C, D). Последовательность преобразований может быть задана с помощью последовательного вызова соответствующих методов, либо путём конкатенации отдельных элементов.

В качестве примера рассмотрим систему, представленную на следующем рисунке. Пространство между опорными плоскостями S1 и S2 заполнено однородным веществом с показателем преломления n1, аналогично для пространства между S3 и S4. Допустим, оптическая система между S2 и S3 представляет собой линзу, тогда преобразования между этими опорными плоскостями будут включать в себя преломление луча на первой поверхности, распространение в среде с показателем преломления n2 и последующее преломление на второй поверхности. Пусть n1 = 1, n2 = 1.56, t2 = 10 мм, R1 = 200 мм, R2 = -150 мм (поверхность вогнутая с точки зрения “луча”). Тогда для линзы можно записать

```
## L1 = Lens: R(1, 200, 1.56) : T(10, 1.56) : R(1.56, -150, 1)
## L1
A: 0.982   B: 6.410
C: -0.006   D: 0.976
```



Поскольку в соседних элементах встречаются одинаковые показатели преломления, для упрощения записи было реализовано следующее правило: если в конструкторе элемента указывается показатель преломления, он может быть “унаследован” следующим элементом при вызове через двоеточие. Тогда рассматриваемая система может быть представлена в виде

```
## L2 = Lens:R(1, 200, 1.56) \\
..   :T(10)
..   :R(-150, 1)
..
## L2 == L1
true
```

Ещё одним упрощением является возможность опускать n, если он равен 1, т.е. система находится в воздухе. Однако, в случае элемента **R**() этой возможностью следует пользоваться с осторожностью.

Допустим, t1 = t2 = 20 мм, y1 = 2 мм, V1 = 0.05. Преобразование луча между плоскостями S1 и S4 можно найти следующим образом.

```
## sys = Lens:T(20) .. L1 .. Lens:T(20)
## y2, V2 = sys(2, 0.05)
## print(y2, V2)
3.8547555555556    0.029404444444444
```

19.2.2 Анализ

Обращение элементов матрицы в нуль определяет следующие состояния:

- D = 0 - входная плоскость (S1) соответствует передней фокальной плоскости;
- B = 0 - входная (S1) и выходная (S4) плоскости являются сопряжёнными (плоскости предмета и изображения) с увеличением A;
- C = 0 - система афокальная;
- A = 0 - выходная плоскость (S4) соответствует задней фокальной плоскости.

Функция **solve**(*функция*, *индекс*, *x0*) выполняет поиск параметра *x*, при котором заданный элемент матрицы становится равен нулю. Функция возвращает описание оптической системы с учётом параметра.

```
## -- определим расстояние до плоскости изображения t3 (B = 0)
## -- если расстояние до предмета t1 = 250 мм
## fn = function (d) return Lens:T(250) .. L1 .. Lens:T(d) end
## Lens:solve(fn, 'B', 100)  -- начальное приближение 100 мм
393.31465172138
## -- расстояние до задней фокальной плоскости  (A = 0)
## Lens:solve(fn, 'A', 100)
151.87162948081
```

Кардинальные точки оптической системы могут быть найдены с помощью функции **cardinal**(), которая возвращает таблицу с положением фокальных, главных и узловых точек.

```
## t = L1:cardinal()
## t.F2    -- задний фокус
151.87162948081
## t
From the input plane
F1 at -150.94638891826
H1 at 3.700962250185
N1 at 3.7009622501851
From the output plane
F2 at 151.87162948081
H2 at -2.7757216876388
N2 at -2.7757216876388
```

19.2.3 Другие функции

Создать копию системы позволяет функция **copy**(). Инвертировать преобразования оптической системы можно с помощью метода **inv**(). Значения отдельных элементов могут быть получены с помощью индексов (*A*, *B*, *C*, *D*), а матрица целиком - через метод **matrix**().

```
## L1.C
-0.0064663247863248
## L1:matrix():det()
1.0
## isys = sys:inv()
## print(isys(y2, V2))
2.0    0.05
```

19.3 Лазерное излучение

Одномодовое лазерное излучение обычно описывается функцией Гаусса. В модуле *lens* определены несколько функций для работы с гауссовым (лазерным) излучением. Поскольку результатом являются размерные единицы (длины,

радиусы), все аргументы должны быть одной размерности. Для удобства преобразований можно воспользоваться модулем *units*.

Функция **gParam**(*перетяжка*, *длина_волны*) возвращает угол расходимости в дальней зоне и длину ближней зоны, **gSize**(*перетяжка*, *длина_волны*, *расстояние*) определяет радиус кривизны R волнового фронта и размер пучка w на заданном расстоянии от перетяжки. Метод **beam**(R, w, *длина_волны*) вычисляет R и w на выходе из системы. Если оптическая система описывает резонатор, то метод **emit**(*длина_волны*) вернёт R выходного излучения, а в случае устойчивого резонатора также радиус луча w, перетяжки $w0$, и её смещение влево от опорной плоскости.

```
## lam = 1024 * U('nm')  -- длина волны
## -- радиус перетяжки 1 мм
## print( Lens:gParam(1, lam 'mm') )  -- расходимость, ближняя зона
0.0003259493234522    3067.9615757713
## d = 100 * U('m')  -- расстояние до преграды
## -- радиус кривизны, радиус луча
## print( Lens:gSize(1, lam 'mm', d 'mm') )
100094.1238823    32.610268545191
## -- параметры на выходе системы, радиус кривизны 1000 мм
## print( L1:beam(1000, 1, lam 'mm') )
-180.0620907336    0.98846374677498
## -- резонатор
## air_rod = Lens: T(50) : T(30, 1.56)
## mirror = Lens:M(-300)
## cavity = mirror .. air_rod .. mirror .. \
.. air_rod  -- в общем случае в обратном порядке
## print( cavity:emit(lam 'mm') )
300.0    0.29701991298779   0.22053244527224    134.61538461538
```

XX *geodesy*: преобразования координат

20.1 Введение

Модуль geodesy содержит функции для решения ряда задач, возникающих в геодезии, таких как преобразования координат или прямая и обратная задачи. При этом Земля моделируется эллипсоидом с экваториальным радиусом a (большая полуось) и полярным радиусом b. Сжатие f определяется как $f = (a - b)/a$. Широта, долгота и высота обозначаются через B, L, H соответственно, угловые координаты выражаются в градусах, расстояния в метрах.

Эллипсоид для расчётов может быть получен с помощью конструктора **Geo**(*параметры*). По умолчанию он использует параметры WGS84, однако, можно определить собственные значения a и f. Координаты точки на поверхности эллипсоида могут быть преобразованы в прямоугольную геоцентрическую систему с помощью метода **toXYZ**(*blh*). Обратное преобразование выполняет функция **toBLH**(*xyz*).

```
## wgs = Geo()
## t1 = {B=22.2, L=33.3, H=10}  -- углы в градусах
## xyz = wgs:toXYZ(t1)
## xyz
{
  Y = 3243716.0912491,
  Z = 2394935.8675327,
  X = 4938085.8106046,}
## wgs:toBLH(xyz)
{
  B = 22.2,
  H = 10.0,
  L = 33.3,}
```

20.2 Прямая и обратная задачи

Пусть задана точка в пространстве (широта и долгота). Относительно неё нужно провести линию определённой длины в направлении, заданном азимутом. Прямая задача заключается в том, чтобы найти положение и азимут конца прямой. Обратная задача, соответственно, пытается найти длину и азимуты для отрезка, концы которого заданы.

Для решения этих проблем реализованы функции **solveDir**(*начало*, *азимут*, *длина*), которая возвращает точку конца и азимут, и **solveInv**(*начало*, *конец*), возвращающая расстояние и азимуты.

```
## -- угол 30 градусов, длина 10 км
## t2, a2 = wgs:solveDir(t1, 30, 1e4)
## a2
30.018361900328
```

```
## t2
{
  L = 33.348504398613,
  B = 22.278201086692,}
## print(wgs:solveInv(t1, t2))  -- длина, азимут 1, азимут 2
9999.4126387104    29.994174586851    30.012532175206
```

20.3 Преобразования координат

20.3.1 Разные эллипсоиды

Чтобы создать новый эллипсоид, нужно указать его параметры в аргументе конструктора **Geo**(). Для преобразования координат между разными эллипсоидами нужно определить линейное и угловое смещение между базисами, а также масштабный коэффициент с помощью функции **into**(*эллипсоид 1*, *эллипсоид 2*, *линейное смещение*, *угловое смещение*, *масштаб*). После этого в каждый эллипсоид будут добавлены методы для взаимного преобразования геоцентрических и астрономических координат.

```
## -- российская система "Параметры Земли"
## pz = Geo {a=6378136, f=1/298.25784}
## pz:into(wgs, \
.. {-1.1, -0.3, -0.9}, \
.. {0, 0, math.rad(-0.2/3600)}, \
.. -0.12E-6)
## xyz_wgs2pz = wgs.xyzInto[pz]
## xyz_wgs2pz(xyz)
{
  Z = 2394937.054925,
  X = 4938090.6483711,
  Y = 3243711.9923913,}
## blh_pz2wgs = pz.blhInto[wgs]
## blh_pz2wgs(t1)
{
  B = 22.199996473721,
  L = 33.300003425074,
  H = 7.508917491904,}
```

20.3.2 Один эллипсоид

Для представления развёртки Земля на плоскости широко распространена универсальная поперечная проекция Меркатора (UTM). Функция **bl2utm**(*blh*) для заданных астрономических координат определяет координаты смещения на север N и восток E, номер соответствующей зоны, полушарие (N/S), а также масштабный коэффициент. Обратное преобразование выполняет функция **utm2bl**(*nez*), которая помимо величины смещения (NE) должна знать номер зоны и индекс полушария.

```
## ne, s = wgs:bl2utm(t1)
## ne
{
```

```
  hs = N,
  zone = 36,
  E = 530922.7463073,
  N = 2454995.0126965,}
## -- обратно
## wgs:utm2bl(ne)
{
  L = 33.3,
  H = 0,
  B = 22.2,}
```

Альтернативной формой представления положения является геохэш. Фактически, он задаёт не отдельную координату, а область на карте, размеры которой определяются длиной хэша (чем он короче, тем больше территория). Закодировать положение позволяет функция **hashEncode**(*точка*, *длина хэша*). Второй аргумент опциональный, может принимать значения от 1 до 12 и по умолчанию равен 6. Обратное преобразование выполняет **hashDecode**(*хэш*), результатом являются координаты центральной точки зоны и размеры по широте и долготе.

```
## hash = Geo:hashEncode(t1, 5)
## hash
sezwm
## p, rng = Geo:hashDecode(hash)
## p
{
  B = 22.21435546875,
  L = 33.28857421875,}
## rng
{ 0.0439453125, 0.0439453125, }
```

20.4 Другие функции

Перевести градусы, минуты и секунды в десятичную дробь позволяет функция **dms2deg**(*градусы*, *минуты*, *секунды*), при этом второй и третий аргументы могут быть опущены. Обратное преобразование выполняет функция **deg2dms**(*градусы*). Определить ускорение свободного падения на поверхности эллипсоида с учётом широты можно с помощью функции **grav**(*широта*).

```
## Geo:dms2deg(10, 20)  -- градусы и минуты
10.333333333333
## print(Geo:deg2dms(10.123456))  -- в градусы, минуты, секунды
10    7    24.441599999997
## Geo:grav(50.5)
9.8111495438143
```

Приложения

A. Настройка окружения

В Linux обычно не требуется явно прописывать путь к программе Lua, т.к. она устанавливается стандартными средствами. Если программа, например, собрана из исходных кодов, путь можно прописать в файле bashrc в виде строки

```
alias lua='путь к файлу lua'
```

В случае Windows может потребоваться войти в меню Переменных окружения в Свойствах системы, открыть редактирование для переменной Path, нажать кнопку Добавить и вписать путь к исполняемому файлу lua. После этого программа должна стать доступной для вызова из терминала.

B. Локализация программы

Чтобы добавить поддержку нового языка, необходимо сгенерировать файл локализации, сделать перевод для необходимых функций и добавить имя файла в настройки программы. Например, чтобы добавить перевод на французский язык, нужно перейти в папку с so/\/ata и выполнить команду

```
lua sonata.lua --lang fr
```

После этого в папке *locale* появится файл *fr.lua*, содержащий закомментированный список сообщений на английском языке. Если раскомментировать какое-либо сообщение и сделать для него перевод, то при загрузке файла оно заменит строку по умолчанию. Чем больше сообщений переведено, тем выше локализация программы. Чтобы прочитать файл при загрузке, нужно добавить его в переменную: SONATA_LOCALIZATION = "*fr.lua*".

Команда *lang* также может быть использована для обновления списка сообщений в существующем файле. Т.е. при вызове данной команды в файл будут добавлены новые сообщения (на английском) и удалены неактуальные.

C. Добавление модуля

Чтобы добавить новый модуль, необходимо поместить соответствующий файл в папку *matlib* и добавить ссылку в таблицу *use*.

Проще всего добавить модуль с помощь генерации шаблона. Например, нужно добавить модуль для обработки сигналов. Для этого перейдите в папку so/\/ata и выполните команду

```
lua sonata.lua --new signal Sn "Signal processing"
```

Здесь первый аргумент после *new* это название файла, второй - алиас (используется в юнит тестах), третий аргумент содержит краткое описание. Алиас и

описание могут быть опущены. В результате будет сформирован файл *signal.lua* в папке *matlib*.

Доступные пользователю методы будут находиться в таблице с заданным ранее именем (signal). Рекомендуется использовать стиль наименования camel case, а “скрытые” методы начинать с нижнего подчеркивания.

Каждый модуль содержит раздел, который начинается с ‘--*[[TEST_IT*’ и заканчивается ‘*]]*’. Это юнит тесты, которые состоят из последовательности блоков текста, разделённый пустыми строками. Если необходимо проверить результат функции, его нужно сохранить в переменной *ans* и добавить ожидаемый результат в комментарий, который начинается с ‘-->’ для точного равенства или ‘--n>’ для сравнения чисел с плавающей запятой, где n - порядок точности.

Чтобы добавить описание функции для **help**(), нужно поместить функцию в таблицу about со значением в виде таблицы из трех элементов: первый - сигнатура функции, второй - описание, третий (опциональный) - тэг для группировки сообщений. Например:

```
about[signal.lowPass] = {“:lowPass(freq_d) --> S”, “Make low pass filter”, “filters”}
```

Для сигнатуры используется следующее правило: если строка начинается со знаков препинания, в описании к ней будет добавлен алиас модуля, т.е. это метод класса, а не объекта. Постфикс ‘_d’ служит для обозначения типа данных (digit) и является опциональным.

Чтобы модуль загружался аналогично другим компонентам программы, нужно добавить его в таблицу *use* файла *sonata.lua* в виде *имя_файла* = *Алиас*. После этого можно будет загружать модуль, упоминая в коде его алиас или используя **use**().