Методическое пособие для выполнения лабораторных работ по оптической информатике на языке Octave

Кириленко М.С.

23 сентября 2023 г.

1 Введение

Данное пособие предназначено для студентов, обучающихся на курсе «Оптическая информатика», и посвящено языку Octave. Предполагается, что студенты уже имеют базовое представление о программировании и знакомы хотя бы с одним из языков программирования (например, C++).

GNU Octave – высокоуровневый язык, предназначенный в первую очередь для численных расчётов. Он предоставляет удобный интерфейс командной строки для решения линейных и нелинейных численных задач и для выполнения других вычислительных экспериментов с использованием языка, по большей части совместимого с MATLAB.

Octave имеет широкий набор средств для решения общих вычислительных задач линейной алгебры, нахождения корней нелинейных уравнений, интегрирования простых функций, работы с полиномами, а также решения обыкновенных дифференциальных уравнений. Octave можно легко расширять и настраивать с использованием заданных пользователем функций, написанных на собственном языке Octave, или с использованием динамически подгружаемых модулей, написанных на C, C++, Fortran или других языках.

GNU Octave также является свободно-распространяемым программным обеспечением. Его можно распространять и/или модифицировать в соответствии со стандартной общественной лицензией GNU (GPL).

Пособие знакомит лишь с основами Octave, которые понадобятся для выполнения лабораторных работ. Полный список возможностей представлен в документации, размещённой на официальном сайте программного обеспечения.

2 Начало работы

После установки GNU Octave рекомендуется проверить его работоспособность. Запустите Octave GUI, и введите в командном окне команду, вычис-

ляющую экспоненту с мнимой степенью:

```
>> exp(i*pi)
```

Согласно формуле Эйлера в результате выполнения должна получиться минус 1 (в пределах точности вычислений):

```
>> exp(i*pi)
ans = -1.0000e+00 + 1.2246e-16i
```

В данном примере присутствует видимая погрешность в комплексной части результата из-за особенностей работы с числами с плавающей точкой, но эта погрешность является незначительной.

Базовыми строительными блоками для вычислительного анализа являются вектора и матрицы. Чтобы создать матрицу и сохранить её в переменную **A** (для того, чтобы можно было обращаться к ней позже), наберите команду (не вводите символы >> - они означают непосредственный ввод команды):

```
>> A = [ 1, 1, 2; 3, 5, 8; 13, 21, 34 ];
```

После выполнения команды матрица будет храниться в переменной **A**, но в консоль ничего не выведется. Это произошло из-за того, что команда заканчивается точкой с запятой. Если её не ставить, результат будет выведен:

```
>> A = [ 1, 1, 2; 3, 5, 8; 13, 21, 34 ]
A =

1     1     2
3     5     8
13    21    34
```

Octave использует запятые или пробелы для разделения записей в строке, а точки с запятой и символ возврата каретки (`Shift + Enter`) для отделения одной строки от другой.

К векторам и матрицам можно применять многие стандартные функции, например, тригонометрические, так, что они будут применены поэлементно. Например, выполните следующие команды:

```
>> x = -pi:0.01:pi;
>> u = sin(x);
>> plot(x, u);
```

Первая команда создаёт вектор-строку, представляющую собой диапазон значений от минус π до π с промежутком 0,01 между значениями. Вторая команда создаёт новый вектор-строку, полученную применением функции синуса к каждому элементу первого вектора. Третья команда строит график по парам значений из первого и второго вектора: таким образом, получается график функции $u = \sin(x)$.

3 Типы данных

Встроенные в Octave числовые объекты включаяют в себя действительные, комплексны и целочисленные скаляры и матрицы. Все встроенные числа с плавающей точкой по умолчанию имеют двойную точность (binary64). Другие типы рассматриваться не будут.

3.1 Числа

Если числовая константа представляет собой действительное целое число, то оно может быть задано в десятичной, шестнадцатеричной или бинарной форме. Цифры числа могут быть разделены символом нижнего подчёркивания. Следующие примеры дейсвтительного целого числа представляют собой одно и то же значение:

```
42              # десятичная нотация
0x2A            # шестнадцатеричная нотация
0b101010        # бинарная нотация
0b10_1010       # бинарная нотация с разделителем
```

В случае, если константа не является целой, она может быть также записана различными способами. Далее приведён пример разной записи одного и того же числа:

```
.105
1.05e-1
.00105e+2
```

Комплексные константы представляют собой сумму действительной и мнимой части. Мнимая часть определяется как действительное число, после которого сразу же ставится один из символов: *i*, *j*, *I* или *J*. Для обозначения мнимой едиинцы рекомендуется добавлять символ «1» перед ней, поскольку символы *i*, *j*, *I* или *J* могут быть переменными сами по себе:

```
5 + 1i
1j
```

3.2 Матрицы

Размер матрицы при её задании опреляется автоматически. Задаются матрицы с помощью квадратных скобок, а в качестве разделителей элементов внутри строки выступают пробелы и запятые, а сами строки отделяются точкой с запятой, либо символом возврата каретки, как было сказано ранее:

```
>> a = [1, 2; 3, 4]
a =
```

```
1    2
3    4
```

Элементы матрицы могут быть произвольными выражениями при условии, что размерности не должны противоречить друг другу. Например:

```
>> [ a, a ]
ans =

1    2    1    2
3    4    3    4

>> [ a; a ]
ans =

1    2
3    4
1    2
3    4
```

Однако, следующее выражение приведёт к ошибке из-за несовпадения размерностей:

```
>> [ a, 1 ]
error: horizontal dimensions mismatch (2x2 vs 1x1)
```

Избегайте лишних пробелов при задании матрицы, иначе они будут трактоваться как разделитель элементов:

```
[ sin (pi) ] будет расценено как [ sin, (pi) ]
```

3.3 Диапазоны

Диапазоны – это удобный способ задать вектор-строку с элементами, равно отстающими друг от друга. Выражение диапазона представляет собой значение первого элемента, необязательное значение инкремента между элементами и максимальное значение, которое не будет превышено. Эти значения разделяются двоеточиями. Если инкремент не задан явно, он принимается равным 1:

```
>> 1:5
ans =

1    2    3    4    5

>> 1:3:5
ans =
```

```
1    4
```

Несмотря на то, что диапазон является вектором-строкой, Octave в общем не хранит каждое значение этого вектора благодаря тому, что их можно легко вычислить.

3.4 Логические типы

Переменная логического типа может принимать значение *true*, либо *false*. Базовые логические операции &, | и ! соответствуют логическому И, логическому ИЛИ и логическому НЕ. Также логические значения можно использовать в контектсе численных расчётов, в этом случае *true* конвертируется в 1, а *false* – в 0.

4 Выражения

Выражения – это базовые строительные блоки языка Octave. Выражение вычисляет значение, которое можно вывести на экран, проверить, сохранить в переменную, передать в функцию или в оператор присваивания.

Как и в других языках, выражения могут включать в себя переменные, ссылки на массивы, константы, вызовы функций и комбинации этих выражений с различными операторами. Мы рассмотрим некоторые из них.

4.1 Индексирование

Выражения индексирования позволяют ссылаться или извлекать выбранные элементы векторов и матриц, а также многомерных массивов. Индексами могут служить скаляры, вектора, матрицы, а также специальный оператор «:», позволяющий выбирать целые строки, колонки или слои более высокой размерности.

Индексирование представляет собой выражение, состоящее из скобок, внутри которых перечисляются M выражений, разделённых запятыми. Каждое индивидуальное значение индекса (или компонента) используется для соответствующего измерения объекта, к которому применяется. Другими словами, первая индексная компонента используется для первого измерения (номер строки) объекта, вторая - для второго измерения (номер столбца), и т.д. Число M соответствет размерности индексирования. Индекс с двумя компонентами является двумерных индексом, поскольку имеет два измерения.

В простом случае все компоненты являются скалярами и размерность индекса M эквивалентна размерности объекта, к которому она применяется. Например, требуется создать трёхмерный массив $2 \times 2 \times 2$, заполненный числами от 1 до 8 в порядке увеличения размерностей:

```
A = reshape(1:8, 2, 2, 2)
A =
```

```
ans(:,:,1) =

1    3
2    4

ans(:,:,2) =

5    7
6    8
```

Обратите внимание, что трёхмерная матрица создаётся из диапазона $1:8$ с помощью функции *reshape*. Octave выводит трёхмерную матрицу по слоям: сначала первый слой, а затем второй, ссылаясь на матрицу с помощью переменной ans. Индекс первого элемента в Octave равен 1, а не 0, как во многих языках программирования.

Чтобы получить элемент во второй строке, первой колонке, на втором слое, к матрице можно обратиться с помощью соответствующего выражения индексирования:

```
>> A(2, 1, 2)
ans =  6
```

Размер возвращаемого объекта в заданном измерении эквивалентен числу элеметов в соответствующем компоненте выражения индексирования. Если все компоненты представляют собой скалярные величины, результатом будет являться одно число. Однако, если представляет собой вектор/-матрицу или диапазон, возвращаемое значение будет являться декартовым произведением индексов в соответствующих измерениях. Например:

```
>> A([1, 2], 1, 2)
ans =

5
6

>> [A(1, 1, 2); A(2, 1, 2)]
ans =

5
6
```

Общее число возвращаемых значений равно произведению числа элементов каждого компонента в выражении индексирования. В примере выше оно равно $2 \times 1 \times 1 = 2$.

Существует возможность использование одномерного индекса для работы с мультиразмерным объектом. В этом случае мультиразмерный объект

рассматривается как линеаризованный, будто это один длинный вектор-столбец, образованный конкатенацией всех столбцов исходного объекта. Например:

```
>> A(5)
ans =  5
>> A(3:5)
ans =

3    4    5
```

Двоеточие «:» может быть использовано в качестве компонента индексирования для выбора всех элементов в заданном измерении. Перезададим матрицу:

```
>> A = [1, 2; 3, 4]
A =

1    2
3    4
```

Все три следующие выражения эквивалентны и выбирают первую строку матрицы:

```
>> A(1, [1, 2])   # строка 1, колонки 1 и 2
>> A(1, 1:2)      # строка 1, колонки в диапазоне 1-2
>> A(1, :)        # строка 1, все колонки
```

Если двоеточие «:» используется в специальном случае одномерного индекса, результат всегда представляет собой вектор-столбец:

```
>> A(:)
ans =

1
3
2
4

>> A(:).'
ans =

1    3    2    4
```

Последняя строка была получена с помощью оператора транспонирования.

В индексировании можно использовать ключевое слово *end*, которое автоматически ссылается на последний элемент соответствующего измерения.

Это слово может быть применено и к заданию диапазона индекса. Примеры:

```
>> A = [1, 2, 3, 4];
>> A(1:end/2)        # первая половина A => [1, 2]
>> A(end + 1) = 5; # добавить элемент
>> A(end) = [];      # удалить элемент
>> A(1:2:end)        # нечётное индексирование A => [1, 3]
>> A(2:2:end)        # чётное индексирование A => [2, 4]
>> A(end:-1:1)       # переворот A => [4, 3, 2, 1]
```

4.2 Вызов функций

Функция – это название для конкретных вычислений. Поскольку она имеет имя, можно обращаться к ней по нему в любой точке программы. Например, функция *sqrt* вычисляет квадратный корень из числа.

Некоторые функции являеются встроенными, поэтому их можно использовать в любой программе Octave. Функция *sqrt* является одной из таких. Более того, можно определять собственные функции.

Для использования функции требуется написать выражение вызова функции, состоящее из имени функции, за которым следует список аргументов, записанных в скобках. Если аргументов больше одного, то они разделяются запятыми. Если аргументов нет, то скобки можно опустить, но лучше их оставить для ясности, что производится вызов функции. Примеры вызовов:

```
>> sqrt (x^2 + y^2)       # Один аргумент
>> ones (n, m)            # Два аргумента
>> rand ()                # Нет аргументов
```

Некоторые встроенные функции могут принимать различное количество аргументов в зависимости от ситуации, из-за этого поведение функции может быть различным при разном наборе числа аргументов.

Как и любое другое выражение, вызов функции возвращает значение, которое рассчитывается на основе аргументов, переданных в функцию. Функция *sqrt* возвращает квадратный корень аргумента. Функции также могут иметь побочные эффекты, такие как присвоение значения конкретным переменным или выполнение входных или выходных операций.

В отличие от других языков, функции в Octave могут возвращать несколько значений. Например:

```
>> A = [ 3, 2, 2; 2, 3, -2 ]
A =

3   2   2
2   3  -2

>> [U, S, V] = svd (A)
```

```
U =

-0.70711   -0.70711
-0.70711    0.70711

S =

Diagonal Matrix

5   0   0
0   3   0

V =

-7.0711e-01   -2.3570e-01   -6.6667e-01
-7.0711e-01    2.3570e-01    6.6667e-01
-6.4793e-17   -9.4281e-01    3.3333e-01
```

В примере выполняется сингулярное разложение матрицы **A**, результат которого записывается в матрицы **U**, **S** и **V**.

Аргументы во время вызова передаются в функцию по значению. Иными словами, если в функцию передать какой-то аргумент, он гарантированно не сможет измениться, поскольку функция работает с его копией. Тем не менее, память на создание копии не тратится до тех пор, пока не будет предпринята попытка изменения аргумента со стороны функции – в этом случае Octave будет обязан создать копию для предотвращения изменения значения переменной за пределами области видимости функции.

4.3 Арифметические операции

Следующие арифметические операции доступны как для скалярных величин, так и для матриц. Для некоторых операций требуются матрицы одинаковых размерностей или их возможность транслироваться в одинаковую форму. Транслирование в общем случае не будет здесь изучаться. Вместо этого ограничимся примером: если требуется сложить матрицу и константу, последняя рассматривается как матрица такой же размерности, каждый элемент которой равен исходной константе.

```
>> A = [1, 2; 3, 4]
A =

1   2
3   4

>> A + 5
ans =
```

```
6     7
8     9
```

Транслировать можно не только константы, но рассматривать такие случаи мы не будем.

В следующем списке перечислены арифметические операции. Большинство операций имеет функцию-аналог, которая приведена рядом с ними.

- $x + y$, **plus** (x, y)

  Сложение. Если оба операнда являются матрицами, число строк и столбцов должно быть одинаковым, либо матрицы должны допускать транслирование к одинаковой форме.

- $x .+ y$

  Поэлементное сложение. Эта операция эквивалентна $+$.

- $x - y$, **minus** (x, y)

  Вычитание. Если оба операнда являются матрицами, число строк и столбцов должно быть одинаковым, либо матрицы должны допускать транслирование к одинаковой форме.

- $x .- y$

  Поэлементное вычитание. Эта операция эквивалентна $-$.

- $x * y$, **mtimes** (x, y)

  Матричное умножение. Число столбцов x должно сочетаться с числом строк y.

- $x .* y$, **times** (x, y)

  Поэлементное умножение. Если оба операнда являются матрицами, число строк и столбцов должно быть одинаковым, либо матрицы должны допускать транслирование к одинаковой форме.

- x / y, **mrdivide** (x, y)

  Правое деление. Для вектора-строки x и квадратной матрицы y операция эквивалентна следующему выражению:

  ```
  (inverse (y') * x')'
  ```

  Но при этом обратная матрица *inverse* не вычисляется. Иными словами, находится решение t системы уравнений $yt = x$. Если система не квадратная или коэффициенты матрицы вырожденные, решается задача минимизации нормы отклонения.

- $x ./ y$, **rdivide** (x, y)

  Поэлементное правое деление. Если оба операнда являются матрицами, число строк и столбцов должно быть одинаковым, либо матрицы должны допускать транслирование к одинаковой форме.

- $x \setminus y$, **mldivide** (x, y)

  Левое деление. Для матрицы x и вектора-столбца y операция эквивалентна следующему выражению:

```
inverse (x) * y
```

  Но при этом обратная матрица *inverse* не вычисляется. Иными словами, находится решение t системы уравнений $xt = y$. Если система не квадратная или коэффициенты матрицы вырожденные, решается задача минимизации нормы отклонения.

- $x .\setminus y$, **ldivide** (x, y)

  Поэлементное левое деление. Каждый элемент y делится на соответсвующий элемент x. Если оба операнда являются матрицами, число строк и столбцов должно быть одинаковым, либо матрицы должны допускать транслирование к одинаковой форме.

- $x \hat{} y$, $x ** y$, **mpower** (x, y)

  Возведение в степень. Если x и y – скалярные величины, то результат – возведение x в степень y. Другие случаи рассматривать не будем.

- $x .\hat{} y$, $x .** y$, **power** (x, y)

  Поэлементное возведение в степень. Если оба операнда являются матрицами, число строк и столбцов должно быть одинаковым, либо матрицы должны допускать транслирование к одинаковой форме. Если существует несколько комплексных решений, выбирается одно с минимальным неотрицательным аргументом (углом). Из-за последнего правила может быть выбран комплексный корень, даже если существует действительный (когда он отрицательный). Следует использовать функции *realpow*, *realsqrt*, *cbrt* или *nthroot*, если предпочтительнее действительный результат.

- $-x$, **uminus** (x)

  Отрицание.

- $+x$, **uplus** (x)

  Унарный плюс. Этот оператор не оказывает никакого эффекта на операнд.

- x', **ctranspose** (x)

  Комплексное сопряжённое транспонирование. Если аргументы действительные, то оператор действует так, как и обычное транспонирование. Для комплексных аргументов эквивалентно следующее выражение:

```
conj (x.')
```

- *x.'*, **transpose** (*x*)

  Транспонирование.

- **plus** (*x1*, *x2*, ...)

  Сложение. Эквивалентно операции +, но может быть применено к переменному числу аргументов.

- **mtimes** (*x1*, *x2*, ...)

  Произведение. Эквивалентно операции ∗, но может быть применено к переменному числу аргументов.

- **times** (*x1*, *x2*, ...)

  Поэлементное произведение. Эквивалентно операции .∗, но может быть применено к переменному числу аргументов.

4.4 Операторы сравнения

Все операторы сравнения возвращают 1, если сравнение истинно, и 0 в противном случае. Для матричных значений операторы применяются поэлементно, при этом остаются актуальными правила транслирования. Примеры сравнения:

```
>> [1, 2; 3, 4] > [1, 3; 2, 4]
ans =

0   0
1   0

>> [1, 2; 3, 4] <= 2
ans =

1   1
0   0
```

- $x < y$, **lt** (x, y)

  Истина, если x меньше y.

- $x <= y$, **le** (x, y)

  Истина, если x меньше, либо равно y.

- $x == y$, **eq** (x, y)

  Истина, если x равно y.

- $x >= y$, **ge** (x, y)

  Истина, если x больше, либо равно y.

- $x > y$, **gt** (x, y)

  Истина, если x больше y.

- x != y, x ~= y, **ne** (x, y)

  Истина, если x не равно y.

- **isequal** (*x1*, *x2*, ...)

  Истина, если все *x1*, *x2*, ... равны.

- **isequaln** (*x1*, *x2*, ...)

  Истина, если все *x1*, *x2*, ... равны, при этом все значения NaN считаются равными между собой (в общем случае NaN != NaN).

В отличие от традиционного комплексного анализа, для комплексных чисел эти операции также определены. В первую очередь у комплексных чисел сравниваются модули, а в случае их равенства сравниваются аргументы (углы).

4.5 Логические операции

Логические (или булевы) выражения представляют собой комбинации операций с использованием булевых операторов ИЛИ («|»), И («&») и НЕ («!», « »), а также скобок для контроля порядка вычисления.

Существуют именованные функции, соответствующие операторам ИЛИ (*and*), И (*or*) и НЕ (*not*). Первые две поддерживают переменное число аргументов.

Поэлементные логические выражения могут быть использованы везде, где ожидаются выражения сравнения. Они могут быть подставлены в выражения *if* и *while*. Однако, при использовании матриц в качестве условия в *if* или *while* рассматривается как *true*, только если все элементы этой матрицы не равны 0.

Как и в случае операторов сравнения, каждый элемент поэлементного логического выражения имеет числовое значение (1, если *true*, или 0, если *false*), когда результат булева выражения записывается в переменную или используется в арифметических рассчётах.

```
>> A = [1, 0; 0, 1] & [1, 0; 2, 3]
A =

1   0
0   1
```

В последнем примере числа 2 и 3 были трактованы как *true* при выполнении логической операции.

Для логических операций действуют правила транслирования, встречавшиеся ранее. Для бинарных поэлементных булевых операторов оба операнда всегда будут вычисляться перед определением результата. Это может привести к побочным эффектам. Например:

```
a & b++
```

В этом выражении для переменной b всегда будет выполнен инкремент, даже если переменная a равна 0. Чтобы избежать вычисления каждого аргумента, если итог выражения можно определить досрочно, существуют логические операторы, выполняющиеся по короткой схеме: ИЛИ («||») и И («&&»). Перепишем пример:

```
a && b++
```

В этом случае значение переменной b увеличится, только если a не равно нулю. Аналогично работает оператор «||».

4.6 Выражения присваивания

Присваивание – это выражение, которое сохраняет значение в переменную:

```
>> z = 1
```

Оператор «=» называется оператором присваивания.

Важно отметить, что переменные не имеют постоянного типа. Тип переменной зависит от того, какое значение оно хранит в данный момент. Пример:

```
>> foo = 1
foo =  1
>> foo = "bar"
foo = bar
```

Второе присвоение передаёт переменной *foo* строковое значение, а тот факт, что до этого переменная хранила числовое значение, забывается.

Присвоение скалярной величины индексированной матрице задаёт все элементы, указанные индексами, равными этой скалярной величине. Например, a – матрица хотя бы с двумя колонками:

```
>> a(:, 2) = 5
```

Это выражение сделает все элементы во второй колонке равными 5.

Присвоение пустой матрицы «[ ]» в большинстве случаев позволяет удалить строки или колонки матриц и векторов. Например, для матрицы A размером 4×5:

```
>> A(3, :) = [];
```

Это выражение удалит третью строку A, в то время как следующее выражение удалит первую, третью и пятую колонки.

```
>> A(:, 1:2:5) = [];
```

Так как присваивание является выражением, то оно возвращает значение. Так, выражение $z = 1$ возвращает значение 1. Благодаря этому можно записать вместе несколько присваиваний:

```
>> x = y = z = 0;
```

Это справедливо и для списка значений:

```
>> [a, b, c] = [u, s, v] = svd (a);
```

Последнее выражение эквивалентно:

```
>> [u, s, v] = svd (a);
>> a = u;
>> b = s;
>> c = v;
```

В подобных выражениях количество значений в каждой части не обязано совпадать:

```
>> [a, b] = [u, s, v] = svd (a);
```

Это эквивалентно:

```
>> [u, s, v] = svd (a);
>> a = u;
>> b = s;
```

Однако, количество значений в левой части выражения не должно превышать количество значений в правой части. Символ ~можно использовать в качестве заполнителя в левой части, чтобы соответствующее возвращаемое значение игнорировалось и нигде не хранилось:

```
>> [~, s, v] = svd (a);
```

Широко распространённой практикой программирования является увеличение существующей переменной на заданное значение, например:

```
>> a = a + 2;
```

Это выражение может быть перезаписано понятнее и в более краткой форме с использованием оператора «+=»:

```
>> a += 2;
```

Похожие операторы существуют для вычитания (−=), произведения (∗=) и деления (/=).

4.7 Операции инкрементирования

Операции инкрементирования увеличивают или уменьшают значение переменной на 1. Оператор для увеличения значения записывается как «++», а для уменьшения – «−−». Для обоих операторов сущесвуют варианты постфиксной и префиксной записи.

Чтобы увеличить значение переменной до того, как оно будет взято, используется префиксная запись $++x$. Если требуется сначала получить значение, а потом изменить переменную, используется запись $x++$.

Для матриц и векторов операции инкрементирования работают с каждым элементом операнда.

5 Инструкции

Инструкции могут быть как простым выражением, так и сложным списком вложенных циклов и условных инструкций.

Управляющие инструкции, такие как *if* и *while*, контролируют поток исполнения в программах Octave. Все управляющие инструкции начинаются с ключевого слова для того, чтобы отличать их от простых выражений. Многие управляющие инструкции содержат другие инструкции, к примеру, инструкция *if* содержит другую инструкцию, которая может исполниться или нет. Для каждой управляющей инструкции существует конечная инструкция.

Здесь будут кратко рассмотрены лишь некоторые инструкции. Другие (такие как *switch* и *do-until*) при желание можно изучить самостоятельно.

Наиболее общая форма инструкции *if* выглядит следущим образом:

```
if ( условие1 )
        # если условие1 истинно
elseif ( условие2 )
        # условие1 ложно, но условие2 истинно
else
        # все условия ложны
endif
```

Условий *elseif* может быть несколько. Условия *elseif* и *else* могут отсутствовать вовсе.

Инструкция *while* является простейшей реализацией цикла в Octave. Она периодически выполняет инструкции до тех пор, пока условие истинно:

```
while ( условие )
        # тело цикла
endwhile
```

Инструкция *for* более удобна для подсчёта числа итераций в цикле. Общая форма цикла *for* выглядит следующим образом:

```
for переменная = выражение
        # тело цикла
endfor
```

Выражение присваивания в операторе *for* отличаются от стандартного присваивания. Вместо того, чтобы присвоить полностью результат *выражения*, оно присваивает по одному столбцу *выражения* в *переменную* за итерацию. Если *выражение* является диапазоном, вектором-строкой или скалярной величиной, значение *переменной* всегда будет скаляром, когда начинается новая итерация цикла.

Следующий пример показывает способ создания вектора, содержащего первые 10 элементов последовательности Фибоначчи:

```
fib = ones (1, 10); # строковый вектор из единиц
for i = 3:10
        fib(i) = fib(i-1) + fib(i-2);
endfor
```

Инструкция *break* прерывает выполнение ближайшего окружающего цикла и заставляет поток выполнения досрочно завершить этот цикл. Инструкция *continue* досрочно завершает выполение текущей итерации цикла, но сам цикл продолжается.

В языке Octave большинство инструкций заканчивают своё выполнение на символе новой строки, поэтому требуется дополнительно сообщить Octave продолжить инструкцию при переходе с одной строки на другую. Для этого существует последовательность символов «...». При этом любой текст после неё игнорируется. Пример:

```
x = long_variable_name ...      # comment one
    + longer_variable_name ...comment two
    - 42                        # last comment
```

Текст внутри двойных кавычек у строковой константы можно перенести на другую строку с помощью символа «\».

Выражения внутри скобок могут быть продолжены на другую строку без каких-либо маркеров.

6 Функции и скрипты

Трудные для понимания программы Octave могут быть существенно упрощены с помощью определения функций. Новые функции могут быть определены напрямую в командной строке или во внешнем файле и могут быть вызваны так же, как и любые встроенные функции. Мы рассмотрим лишь некоторые основные сведения о функциях.

6.1 Определение функции

Для объявления функции используется следующая конструкция:

```
function результат = имя ( аргументы )
        тело функции
endfunction
```

Для *имени* функции применяются те же правила, что для имени переменной - оно должно состоять из букв, цифр и символов подчёркивания и при этом не начинаться с цифры. *Тело функции* состоит из инструкций Octave. Оно представляет собой наиболее важную часть функции, так как сообщает, что же функция должна делать.

Аргументы функции и её возвращаемое значение (*результат*) являются необязательными параметрами. Аргументов может быть больше 1, тогда они записываются через запятую. Возвращаемых значений тоже может быть несколько, но здесь не будут рассматриваться такие случаи. Пример объявления и вызова функции с одним аргументом и без возвращаемых значений:

```
function wakeup (message)
        printf ("\a%s\n", message);
endfunction

wakeup ("Rise and shine!");
```

Пример функции, вычисляющей среднее значение элементов в векторе:

```
function retval = avg (v)
        retval = sum (v) / length (v);
endfunction
```

6.2 Файлы функций

Не считая простых программ для одноразового запуска, не практично определять все функции, которые нужны, каждый раз, когда они нужны. Вместо этого обычно функции сохрают в файлах, чтобы было легко редактировать их и сохранять для дальнейшего использования в будущем.

Octave не требует загружать определения функций перед тех, как использовать их. Вместо этого необходимо поместить определения функций туда, где Octave сможет их найти.

Когда Octave встречает неопределённый идентификатор, он впервую очередь ищет переменные или функции, которые уже скомпилированы и содержатся в его таблице символов. Если поиск не дал результатов, по определённому списку директорий (*path*) начинается новый поиск файлов с расширением «.m» и именем, совпдающим с идентификатором (т.е. для функции с именем «example» ищется файл «example.m»). Этот файл должен содержать единственную функцию, тогда он компилируется и исполняется.

6.3 Скрипты

Файл скрипта представляет собой файл, содержащий почти любую последовательность команд Octave. Он читается и исполняется так, как будто идёт набор каждой из команд в консоль по отдельности, а также предоставляет удобный способ выполнения последовательности команд, которые логически не относятся к одной функции.

В отличии от файла функции, файл скрипта не должен начинаться с ключевого слова function. Иначе Ocatave определит его как файл функции.

Ещё одно отличие файла скрипта от файла функции заключается в том, что переменные в скриптовом файле не являются локальными и могут быть видны из командной строки.

Несмотря на то, что файл скрипта не может начинаться со слова function, есть возможность определить более одной функции в едином скриптовом файле и загрузить (но не выполнить) их все сразу. Для этого первой лексемой в файле (не считая пробелов и комментариев) должно быть что-то другое, например, инструкция, не имеющая эффектов:

```
1;
```

Чтобы запустить файл скрипта, он должен находиться в пути загрузки Octave. Указать путь загрузки и выполнить скрипт можно из графического интерфейса.

7 Графики и изображения

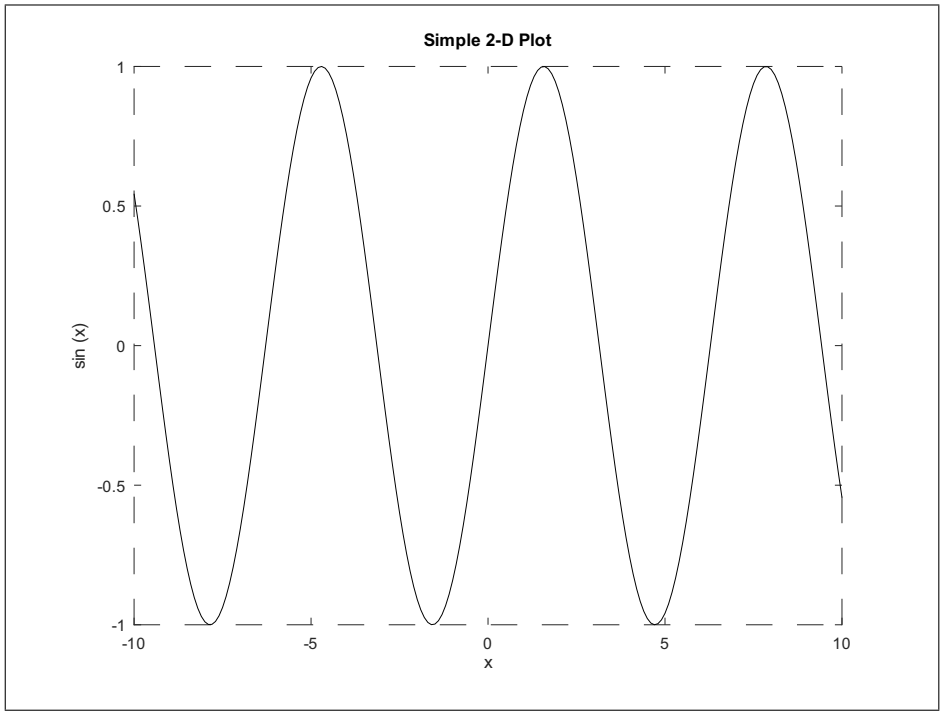
Octave предоставляет простую возможность для создания многих различных типов двумерных и трёхмерных графиков с использованием высокоуровневых функций. Мы рассмотрим некоторые из них.

7.1 Функция plot

Функция plot позволяет создавать простейшие X-Y графики с линейными осями. Например:

```
x = -10:0.1:10;
plot (x, sin (x), "k");
xlabel ("x");
ylabel ("sin (x)");
title ("Simple 2-D Plot");
```

Данный набор команд отображает график синуса на участке $x \in [-10, 10]$, при этом сама линия графика имеет чёрный цвет (из-за того, что в функцию plot передана строка "k"):



plot (*y*)
plot (*x*, *y*)
plot (*x*, *y*, *fmt*)
plot (..., *property*, *value*, ...)
plot (*x1*, *y1*, *x2*, *y2*, ..., *xn*, *yn*)

Возможны многие различные комбинации аргументов. Самый простой пример:

```
plot (y)
```

Если *y* представляет собой вектор действительных чисел, то на построенном графике будет выражена зависимость *y* от номера элемента в этом векторе. Если *y* комплексное, то будет построена зависимость мнимой части от действительной.

Если задано несколько аргументов, они интепретируются одним из следующих способов:

```
plot (y, property, value, ...)
```

```
plot (x, y, property, value, ...)
```

```
plot (x, y, fmt, ...)
```

И так далее. Аргументов может быть сколько угодно. Значения x и y интерпретируются следующим образом:

- Если аргумент единственный, то график строится в зависимости от того, действительный это вектор или комплексный (см. выше).
- Если x и y - скалярные величины, то рисуется только одна точка.
- Если x и y - вектора, строится зависимость y от x.
- Если x - вектор, а y - матрица, тогда строится зависимость столбцов (или строк, если столбцы не сочетаются с вектором) y от x.
- И т.д.

Важно: если в векторах присутствуют комплексные числа, во многих случаях мнимая часть игнорируется.

В параметрах могут передавать значения пар свойств property-value, которые применяются к линиям графика. Например: linestyle, linewidth, color, marker, markersize, markeredgecolor и markerfacecolor. При желании их можно изучить самостоятельно.

Аргумент *fmt* представляет собой параметр форматирования и также может быть использован для изменения стиля графика. Он представляет собой строку, сформированную из четырёх параметров: <linestyle><marker><color><;displayname;> (стиль линии, маркер, цвет, отображаемое имя). Если указан маркер, но не задан стиль линии, рисуются только маркеры. Аналогично, если указан стиль линии, но не задан маркер, рисуются только линии. Если указано и то, и другое, будут нарисованы и линии, и маркеры. Если не заданы ни fmt, ни пары property-values, применяется форматирование по умолчанию, а именно: все линии рисуются сплошными, без маркеров, с цветом, определяемым свойством "colororder"текущей оси.

Аргументы форматирования:

Стиль линии

'-' Сплошная линия (по умолчанию)
'--' Пунктир
':' Линия из точек
'-.' Штрихпунктир

Маркер

'+' Прицел
'o' Круг
'*' Звезда
'.' Точка
'x' Крест
's' Квадрат
'd' Алмаз (ромб)
'^' Стрелка вверх
'v' Стрелка вниз
'>' Стрелка вправо
'<' Стрелка влево

'p' Пятиугольник
'h' Шестиугольник

Цвет
'k' Чёрный (blacK)
'r' Красный (Red)
'g' Зелёный (Greed)
'b' Синий (Blue)
'y' Жёлтый (Yellow)
'm' Пурпурный (Magenta)
'c' Бирюзовый (Cyan)
'w' Белый (White)

";displayname;"

Где «displayname» - название линии графика, которое используется для легенды.

Аргумент *fmt* может быть также использован для задания легенды линий графика. Для этого необходимо включить название у линии между точками с запятой в конце форматирующей последовательности, описанной выше, например: "+b;Key Title;". Последняя точка с запятой обязательна, иначе произойдёт ошибка.

Некоторые примеры:

```
plot (x, y, "or", x, y2, x, y3, "m", x, y4, "+")
```

Данная команда простроит график *y* красными кружками, *y2* - сплошной линией, *y3* - сплошной пурпурной линией и *y4* - точками, отображаемыми как «+».

```
plot (b, "*", "markersize", 10)
```

Данная команда построит график из данных в переменной *b* точками, отображаемыми как «*», размером 10.

```
t = 0:0.1:6.3;
plot (t, cos(t), "-;cos(t);",
      t, sin(t), "-b;sin(t);");
```

После выполнения набора команд построится график функций косинуса и синуса, а также отобразится легенда, содержащая их названия.

График, вообще говоря, представляет собой сложный объект, рисующийся на объекте figure. Каждый раз, когда нужно нарисовать новый график, старый график на figure стирается. Чтобы была возможность рисовать несколько графиков сразу, рекомендуется создавать новый объект figure:

```
fig1 = figure (1);
```

В качестве параметра можно рекомендуется передавать номер figure. Если требуется добавить новый график к осям, которые уже имеются на figure, можно использовать конструкцию «hold on»:

```
hold on;
```

В этом случае при построение очередного графика ничего стираться не будет. Конструкция «hold off» отменяет эту опцию.

7.2 Функция image

image (*img*)
image (*x*, *y*, *img*)
image (..., "*prop*" , val, ...)

Рисует матрицу как индексированное цветное изображение.

Элементы матрицы img отрисовываются с помощью цветовой карты и должны быть действительными. Аргументы x и y - необязательные двухэлементные вектора, [мин, макс], которые задают координаты центров угловых пикселей изображения. Если параметр задан как [макс, мин], то изображение перевернётся относительно соответствующей оси.

Дополнительно могут быть заданы многократные свойства prop-value, но они всегда должны идти в паре.

Начало координат изображения (0, 0) расположено в верхнем левом углу, ось Y направлена вниз.

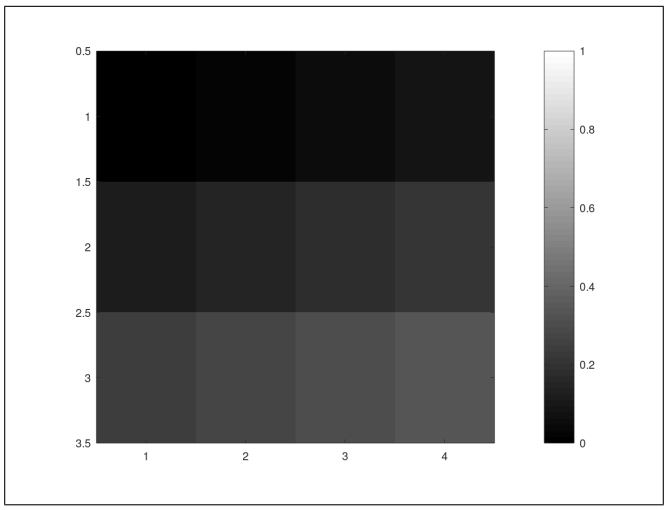
Для задания цветовой карты используется команда colormap. Пример, как настроить отрисовку изображения в оттенках серого:

```
colormap gray;
```

Для того чтобы рядом с изображением отрисовывался столбец со значениями цветовой карты, необходимо применить команду colorbar. Пример изображения в оттенках серого:

```
image1 = figure(1);
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
image(C);
colormap gray;
colorbar;
```

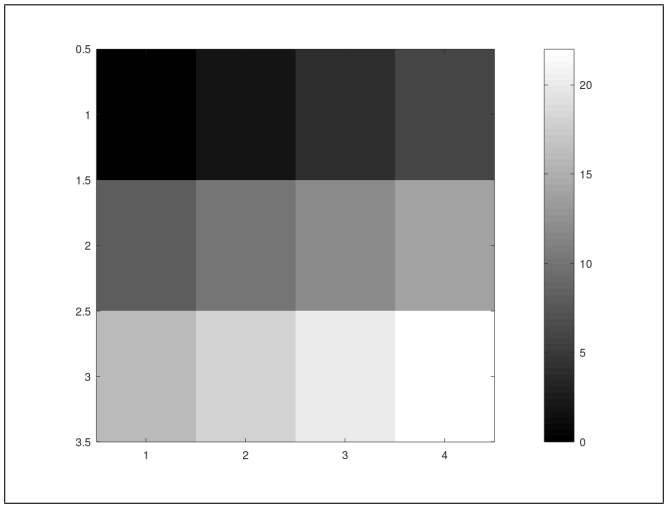
Полученное изображение:



В данном примере в столбце справа представлены значения цветовой карты, а не элементов матрицы, поэтому невозможно сказать, какой цвет какому числу соответствует. Для исправления данной проблемы рекомендуется использовать функцию imagesc вместо функции image:

```
image1 = figure(1);
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
imagesc(C);
colormap gray;
colorbar;
```

Полученное изображение:

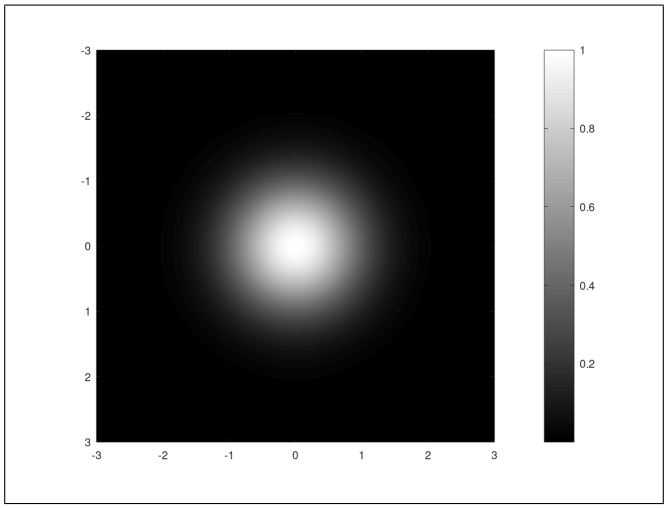


Теперь видно, что чёрному цвету соответствует минимальное значение в матрице - 0, а белому - максимальное значение 22. Существуют различные цветовые карты, которые можно изучить самостоятельно, например: parula, jet, hsv, hot, cool, spring, summer, autumn, winter, gray, bone, copper, pink и т.д.

Рекомендуется самостоятельно изучить пример построения изображения гауссова пучка в области $[-3,3] \times [-3,3]$:

```
x = -3:0.005:3-0.005;
y = x.';
f = exp(-(y .^ 2)) * exp(-(x .^ 2));
image1 = figure(1);
h = imagesc([-3, 3], [-3, 3], f);
colormap gray;
colorbar;
```

Полученное изображение:



8 Операции с матрицами

Мы изучим лишь некоторые операции, с остальными при желании можно ознакомиться самостоятельно.

any (x), **any** (x, dim)

Для векторного аргумента возвращает «истину» (логическое 1), если любой из элементов вектора ненулевой. Для матричного аргумента возвращает строку векторов логических единиц и нулей, где каждый элемент соответствует своему столбцу в матрице. В частности, если третий элемент полученного результата является логической

единицей, то, значит, в матрице в третьем столбце есть хотя бы один ненулевой элемент. Если задан необязательный аргумент *dim*, то работа производится лишь над эти измерением с номером *dim*.

all (x), **all** (x, dim)

Для векторного аргумента возвращает «истину» (логическое 1), если все элементы вектора ненулевые. Для матричного аргумента возвращает строку векторов логических единиц и нулей, где каждый элемент соответствует своему столбцу в матрице. В частности, если третий элемент полученного результата является логической единицей, то, значит, в матрице в третьем столбце все элементы ненулевые. Если задан необязательный аргумент *dim*, то работа производится лишь над эти измерением с номером *dim*.

diag (v)

Возвращает диагональную матрицу с вектором v на главной диагонали.

eye (n)

Возвращает единичную матрицу размерности $n \times n$.

ones (n), **ones** (m, n)

Возвращает матрицу, заполненную единицами, размером $n \times n$ (если аргумент один) или $m \times n$ (если аргументов два).

zeros (n), **zeros** (m, n)

Возвращает матрицу, заполненную нулями, размером $n \times n$ (если аргумент один) или $m \times n$ (если аргументов два). Как правило, используется для создания большинства матриц, которые затем заполняются другими элементами.

$[X, Y]$ = **meshgrid** (x, y)

Возвращает координаты 2D-решётки на основании координат, содержащихся в векторах x и y. X представляет собой матрицу, где каждая строка является копией x, а Y - матрица, где каждая колонка - копия y.

9 Арифметические операции

Как и в предыдущем разделе, здесь рассматриваются только некоторые операции. Они могут быть применимы как к скалярным величинам, так и к векторам и матрицам.

exp (x)

Вычисляет экспоненту e^x для каждого элемента x.

log (x)

Вычисляет натуральный логарифм для каждого элемента x.

log10 (x)

Вычисляет десятичный логарифм для каждого элемента x.

log2 (x)

Вычисляет логарифм по основанию 2 для каждого элемента x.

sqrt (x)

Вычисляет квадратный корень для каждого элемента x. Если x отрицательно, возвращается комплексный результат.

cbrt (x)

Вычисляет кубический корень для каждого элемента x.

nthroot (x, n)

Вычисляет действительный (не комплексный) корень n-ной степени из каждого элемента x. Вектор x должен содержать только действительные числа, а n должен быть скалярной величиной.

abs (z)

Возвращает модуль для каждого комплексного элемента z.

arg (z), **angle** (z)

Возвращает аргумент для каждого комплексного элемента z.

conj (z)

Возвращает комплексное сопряжение для каждого комлпексного элемента z.

real (z)

Возвращает действительную часть для каждого комплексного элемента z.

imag (z)

Возвращает мнимую часть для каждого комплексного элемента z.

sin (x), **cos** (x), **tan** (x), **cot** (x), **sec** (x), **csc** (x)

Любая из этих тригонометрических функций применяется к каждому элементу x в радианах.

asin (x), **acos** (x), **atan** (x), **acot** (x), **asec** (x), **acsc** (x)

Любая из этих обратных тригонометрических функций применяется к каждому элементу x и возвращает результат в радианах.

sinh (*x*), **cosh** (*x*), **tanh** (*x*), **coth** (*x*), **sech** (*x*), **csch** (*x*)

Любая из этих гиперболических функций применяется к каждому элементу *x*.

asinh (*x*), **acosh** (*x*), **atanh** (*x*), **acoth** (*x*), **asech** (*x*), **acsch** (*x*)

Любая из этих обратных гиперболических функций применяется к каждому элементу *x*.

atan2 (*y*, *x*)

Вычисляет atan (*y* / *x*) для соответствующих элементов *y* и *x*. При этом *y* и *x* должны иметь одинаковые размерности. Результат представляет собой полярный угол (в радианах) для декартовой координаты (*x*, *y*).

sum (*x*), **sum** (*x*, *dim*), **prod** (*x*), **prod** (*x*, *dim*)

Вычисляет сумму (или произведение) элементов вдоль измерения *dim*. Если измерение не задано, выбирается первое измерение, не равное единице. В частности, и для вектора-строки, и для вектора-столбца будут проссумированы (или помножены) все их элементы.

sumsq (*x*), **sumsq** (*x*, *dim*)

Вычисляет сумму квадратов модулей элементов вдоль измерения *dim*. Если измерение не задано, выбирается первое измерение, не равное единице.

ceil (*x*)

Округление вверх: возвращает наименьшее целое число, не меньше, чем *x*.

```
>> ceil ([-2.7, 2.7])
ans =

-2   3
```

fix (*x*)

Отбрасывает дробную часть *x* и возвращает целую.

```
>> fix ([-2.7, 2.7])
ans =

-2   2
```

floor (*x*)

Округление вниз: возвращает наибольшее целое число, не больше, чем *x*.

```
floor ([-2.7, 2.7])
ans =

-3   2
```

round (x)

Округляет число x до ближайшего целого.

```
round ([-2.7, 2.7])
ans =

-3   3
```

min (x), **max** (x)

Находит минимальное/максимальное значение в x.

Если x - вектор, находит минимальное/максимальное значение внутри него. Если x - матрица, возвращает вектор-строку с минимальными/максимальными значениями для каждой колонки.

factorial (n)

Возвращает факториал для целого неотрицательного числа n.

airy (k, z)

Возвращает значение функций Эйри первого и второго рода, а также их производных (в зависимости от переданного параметра k), от аргумента z:

$k = 0$ - функция Эйри первого рода $Ai(x)$

$k = 1$ - производная функции Эйри первого рода $Ai(x)$

$k = 2$ - функция Эйри второго рода $Bi(x)$

$k = 3$ - производная функции Эйри второго рода $Bi(x)$

besselj $(alpha, x)$, **bessely** $(alpha, x)$, **besseli** $(alpha, x)$, **besselk** $(alpha, x)$

Возвращает значение функции Бесселя от x. Порядок функции Бесселя *alpha* должен быть действительным. Элементы x могут быть комплексными.

besselj - функция Бесселя первого рода $J_\alpha(x)$

bessely - функция Бесселя второго рода $Y_\alpha(x)$ (функция Неймана)

besseli - модифицированная функция Бесселя первого рода $I_\alpha(x)$ (функция Инфельда)

besselk - модифицированная функция Бесселя второго рода $K_\alpha(x)$ (функция Макдональда)

besselh $(alpha, k, x)$

Возвращает значение функции Бесселя третьего рода (функции Ханкеля). Порядок функции Бесселя *alpha* должен быть действительным. Элементы x могут быть комплексными. Число k определяет род функции Ханкеля:

$k = 1$: функция Ханкеля первого рода $H_\alpha^{(1)}(z)$

$k = 2$: функция Ханкеля второго рода $H_\alpha^{(2)}(z)$

beta (a, b)

Возвращает значение бета-функции для действительных a и b.

gamma (z)

Возвращает значение гамма-функции для комплексного z.

e, **e** (n), **e**$(n,\ m)$
pi, **pi** (n), **pi**$(n,\ m)$
I, **I** (n), **I**$(n,\ m)$
Inf, **Inf** (n), **Inf**$(n,\ m)$
NaN, **NaN** (n), **NaN**$(n,\ m)$
eps, **eps** (n), **eps**$(n,\ m)$
realmax, **realmax** (n), **realmax**$(n,\ m)$
realmin, **realmin** (n), **realmin**$(n,\ m)$

Возвращает число, вектор или матрицу, все элементы которых равны одной из констант:

e, *pi* - фундаментальные постоянные

I - мнимая единица (I можно заменять на i, j, J)

Inf - бесконечность, результат операции $1/0$, может появляться, когда происходит переполнение числа с плавающей точкой

NaN - не число, неопределённость, результат операции $0/0$ (или $Inf - Inf$); *NaN* никогда не равняется самому себе

eps - машинное эпсилон

realmax, *realmin* - максимальное и минимальное представимое положительное число с плавающей точкой

10 Векторизация и быстрое выполнение кода

Векторизацией называется техника программирования, которая использует векторные операции вместо поэлементных операций, основанных на циклах. Помимо того, что векторизация способствует созданию болеее лаконичного кода, она ещё и обеспечивает лучшую оптимизацию в последующей реализации.

Векторизация - это не единственный концепт Octave, но он особенно важен, поскольку Octave - матрично-ориентированный язык. Векторизованный код Octave значительно ускоряет своё выполнение (в десятки и сотни раз) во многих случаях.

Цель векторизации - писать код, стараясь избегать циклов, вместо этого используя операции над целым массивом сразу. Пример:

```
for i = 1:n
        for j = 1:m
                c(i,j) = a(i,j) + b(i,j);
        endfor
endfor
```

Этот код может быть переписан компактнее и эффективнее с помощью матричного сложения:

```
c = a + b;
```

Менее тривиальный пример:

```
for i = 1:n-1
        a(i) = b(i+1) - b(i);
endfor
```

Как видно из кода, в вектор a записывается разность элементов b, из каждого следующего элемента вычитается текущий. С помощью индексирования можно векторизовать и этот код:

```
a = b(2:n) - b(1:n-1);
```

Мы берём вектор b сначала без первого элемента, а затем без последнего. И вычитаем второе из первого, получая тот же результат.

Ещё один пример:

```
for i = 1:n
        if (a(i) > 5)
                a(i) -= 20
        endif
endfor
```

Видно, что из всех элементов вектора a вычитается значение 20, но только если текущее значение этого элемента больше пяти. С помощью булевых индексов можно векторизовать и этот код:

```
a(a>5) -= 20;
```

Используйте встроенные функции там, где это возможно. Зачастую они уже оптимизированы или будут оптимизированы в следующих версиях. К примеру:

```
a = b(2:n) - b(1:n-1);
```

Даже такой код может быть улучшен:

```
a = diff(b);
```

Используйте поэлементные векторные операторы для того, чтобы избегать циклы. Допустим, надо для каждого элемента из массива x рассчитать новый массив в соответствии с функцией $x\sin(x)$:

```
x = -10:10;
n = size(x, 2);
y = zeros(1, n);
for i = 1:n
        y(i) = x(i) * sin(x(i));
endfor
```

После векторизации:

```
x = -10:10;
y = x .* sin(x);
```

А теперь, предположим, что нам нужно на основе двух векторов x и y построить матрицу, элементы которой содержат всевозможные попарные деления элементов x на y:

```
x = -10:10;
y = 15:-1:5;
n = size(x, 2);
m = size(y, 2);
a = zeros(n, m);
for i = 1:n
        for j = 1:m
                a(i, j) = x(i) / y(j);
        endfor
endfor
```

С помощью поэлементного деления и функции meshgrid (читать об этой функции выше) можно всё значительно упростить:

```
x = -10:10;
y = 15:-1:5;
[X, Y] = meshgrid(x, y);
a = X ./ Y;
```

Если в предыдущей задаче требуется не деление, а умножение, можно обойтись даже без meshgrid, воспользовавшись операцией матричного умножения и транспонированием:

```
x = -10:10;
y = 15:-1:5;
a = x.' * y;
```