Бесплатная электронная книга

УЧУСЬ

MATLAB Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#matlab

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: matlab-language

It is an unofficial and free MATLAB Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official MATLAB Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с языком MATLAB

Версии

Версия	Релиз	Дата выхода
1,0		1984-01-01
2		1986-01-01
3		1987-01-01
3,5		1990-01-01
4		1992-01-01
4.2c		1994-01-01
5.0	Том 8	1996-12-01
5,1	Том 9	1997-05-01
5.1.1	R9.1	1997-05-02
5,2	R10	1998-03-01
5.2.1	R10.1	1998-03-02
5,3	R11	1999-01-01
5.3.1	r11.1	1999-11-01
6,0	R12	2000-11-01
6,1	R12.1	2001-06-01
6,5	R13	2002-06-01
6.5.1	R13SP2	2003-01-01
6.5.2	R13SP2	2003-01-02
7	R14	2006-06-01
7.0.4	R14SP1	2004-10-01
7,1	R14SP3	2005-08-01

Версия	Релиз	Дата выхода
7,2	R2006a	2006-03-01
7,3	R2006b	2006-09-01
7,4	R2007a	2007-03-01
7,5	R2007b	2007-09-01
7,6	R2008a	2008-03-01
7,7	R2008b	2008-09-01
7,8	R2009a	2009-03-01
7,9	R2009b	2009-09-01
7,10	R2010a	2010-03-01
7,11	R2010b	2010-09-01
7,12	R2011a	2011-03-01
7,13	R2011b	2011-09-01
7,14	R2012a	2012-03-01
8,0	R2012b	2012-09-01
8,1	R2013a	2013-03-01
8,2	R2013b	2013-09-01
8,3	R2014a	2014-03-01
8,4	R2014b	2014-09-01
8,5	R2015a	2015-03-01
8,6	R2015b	2015-09-01
9,0	R2016a	2016-03-01
9,1	R2016b	2016-09-14
9,2	R2017a	2017-03-08

См. Также: История выпуска MATLAB в Википедии .

Examples

Привет, мир

Откройте новый пустой документ в редакторе MATLAB (в последних версиях MATLAB сделайте это, выбрав вкладку «Главная» на панели инструментов и нажав «Новый скрипт»). Клавиатура по умолчанию для создания нового скрипта - `Ctrl-n` .

В качестве альтернативы, набрав `edit myscriptname.m` вы откроете файл `myscriptname.m` для редактирования или предложите создать файл, если он не существует на пути MATLAB.

В редакторе введите следующее:

```
disp('Hello, World!');
```

Перейдите на вкладку «Редактор» на панели инструментов и нажмите «Сохранить как». Сохраните документ в файл в текущем каталоге `helloworld.m` . Сохранение файла без названия приведет к отображению диалогового окна с именем файла.

В окне команд MATLAB введите следующее:

```
>> helloworld
```

Вы должны увидеть следующий ответ в командном окне MATLAB:

```
Hello, World!
```

Мы видим, что в окне команд мы можем ввести имена функций или файлов сценариев, которые мы написали или которые поставляются вместе с MATLAB, для их запуска.

Здесь мы запустили скрипт helloworld. Обратите внимание, что набирать расширение (`.m`) не нужно. Инструкции, хранящиеся в файле сценария, выполняются MATLAB, здесь печать «Hello, World!» используя функцию `disp` .

Файлы сценариев могут быть записаны таким образом, чтобы сохранить ряд команд для последующего (повторного) использования.

Матрицы и массивы

В MATLAB самым основным типом данных является числовой массив. Это может быть скаляр, одномерный вектор, двумерная матрица или многомерный массив ND.

```
% a 1-by-1 scalar value
x = 1;
```

Чтобы создать вектор строки, введите элементы внутри скобок, разделенные пробелами или запятыми:

```
% a 1-by-4 row vector
v = [1, 2, 3, 4];
v = [1 2 3 4];
```

Чтобы создать вектор-столбец, выделите элементы с точкой с запятой:

```
% a 4-by-1 column vector
v = [1; 2; 3; 4];
```

Чтобы создать матрицу, мы вводим строки, как и раньше, разделенные точкой с запятой:

```
% a 2 row-by-4 column matrix
M = [1 2 3 4; 5 6 7 8];

% a 4 row-by-2 column matrix
M = [1 2; ...
     4 5; ...
     6 7; ...
     8 9];
```

Обратите внимание: вы не можете создать матрицу с неравным размером строки / столбца. Все строки должны иметь одинаковую длину, и все столбцы должны иметь одинаковую длину:

```
% an unequal row / column matrix
M = [1 2 3 ; 4 5 6 7]; % This is not valid and will return an error

% another unequal row / column matrix
M = [1 2 3; ...
     4   5; ...
     6 7 8; ...
     9   10];     % This is not valid and will return an error
```

Чтобы транспонировать вектор или матрицу, мы используем `.'` -оператором, или `'` оператор взять его эрмитово сопряжение, которое комплексно сопряженное транспонированной. Для реальных матриц эти два одинаковы:

```
% create a row vector and transpose it into a column vector
v = [1 2 3 4].';              % v is equal to [1; 2; 3; 4];

% create a 2-by-4 matrix and transpose it to get a 4-by-2 matrix
M = [1 2 3 4; 5 6 7 8].';     % M is equal to [1 5; 2 6; 3 7; 4 8]

% transpose a vector or matrix stored as a variable
A = [1 2; 3 4];
B = A.';                      % B is equal to [1 3; 2 4]
```

Для массивов более двух измерений нет прямого синтаксиса языка, чтобы вводить их

буквально. Вместо этого мы должны использовать функции для их построения (такие как `ones` , `zeros` , `rand`) или манипулировать другими массивами (используя такие функции, как `cat` , `reshape` , `permute`). Некоторые примеры:

```
% a 5-by-2-by-4-by-3 array (4-dimensions)
arr = ones(5, 2, 4, 3);

% a 2-by-3-by-2 array (3-dimensions)
arr = cat(3, [1 2 3; 4 5 6], [7 8 9; 0 1 2]);

% a 5-by-4-by-3-by-2 (4-dimensions)
arr = reshape(1:120, [5 4 3 2]);
```

Матрицы индексирования и массивы

MATLAB позволяет несколько методов индексировать (получать доступ) элементы матриц и массивов:

- **Индексация индексов** - где вы указываете позицию элементов, которые вы хотите в каждом измерении матрицы, отдельно.
- **Линейное индексирование** - где матрица рассматривается как вектор, независимо от ее размеров. Это означает, что вы указываете каждую позицию в матрице с одним номером.
- **Логическая индексация** - где вы используете логическую матрицу (и матрицу `true` и `false` значений) с одинаковыми размерами матрицы, которую вы пытаетесь индексировать в качестве маски, чтобы указать, какое значение нужно вернуть.

Эти три метода теперь описано более подробно с помощью следующего 3 на 3 матрицы `M` в качестве примера:

```
>> M = magic(3)

ans =

       8    1    6
       3    5    7
       4    9    2
```

Индексация индексов

Самый простой способ доступа к элементу - указать его индекс столбца строки. Например, доступ к элементу во второй строке и третьем столбце:

```
>> M(2, 3)

ans =

      7
```

Количество предоставленных индексов точно соответствует числу размеров `M` (два в этом примере).

Обратите внимание, что порядок индексов совпадает с порядком математического соглашения: индекс строки является первым. Более того, индексы MATLAB **начинаются с** `1` а **не** `0` как большинство языков программирования.

Вы можете индексировать сразу несколько элементов, передавая вектор для каждой координаты вместо одного числа. Например, чтобы получить всю вторую строку, мы можем указать, что нам нужны первый, второй и третий столбцы:

```
>> M(2, [1,2,3])

ans =

       3    5    7
```

В MATLAB вектор `[1,2,3]` легче создать с помощью оператора двоеточия, т. Е. `1:3` . Вы можете использовать это и в индексировании. Чтобы выбрать целую строку (или столбец), MATLAB предоставляет ярлык, позволяя вам просто указать `:` Например, следующий код также вернет всю вторую строку

```
>> M(2, :)

ans =

       3    5    7
```

MATLAB также предоставляет ярлык для указания последнего элемента измерения в форме ключевого слова `end` . Ключевое слово `end` будет работать точно так же, как если бы это был номер последнего элемента в этом измерении. Поэтому, если вы хотите, чтобы все столбцы от столбца `2` до последнего столбца, вы можете использовать следующее:

```
>> M(2, 2:end)

ans =

       5    7
```

Индексация индексирования может быть ограничительной, поскольку она не позволит извлекать отдельные значения из разных столбцов и строк; он будет извлекать комбинацию всех строк и столбцов.

```
>> M([2,3], [1,3])
ans =

       3    7
       4    2
```

Например, индексирование индексов не может извлекать только элементы `M(2,1)` или `M(3,3)` . Для этого мы должны рассмотреть линейную индексацию.

Линейная индексация

MATLAB позволяет обрабатывать n-мерные массивы как одномерные массивы при индексировании с использованием только одного измерения. Вы можете напрямую получить доступ к первому элементу:

```
>> M(1)

ans =

       8
```

Обратите внимание, что массивы хранятся в основном порядке в MATLAB, что означает, что вы получаете доступ к элементам, сначала спустившись по столбцам. Таким образом, `M(2)` является вторым элементом первого столбца, который равен `3` а `M(4)` будет первым элементом второго столбца, т.е.

```
>> M(4)

ans =

        1
```

Существуют встроенные функции в MATLAB для преобразования индексов индексов в линейные индексы и наоборот: `sub2ind` и `ind2sub` соответственно. Вы можете вручную преобразовать индексы (`r` , `c`) в линейный индекс на

```
idx = r + (c-1)*size(M,1)
```

Чтобы понять это, если мы находимся в первом столбце, линейный индекс будет просто индексом строки. Вышеприведенная формула верна для этого, поскольку для `c == 1` , `(c-1) == 0` . В следующих столбцах линейный индекс - это номер строки и все строки предыдущих столбцов.

Обратите внимание, что ключевое слово `end` прежнему применяется и теперь относится к самому последнему элементу массива, т.е. `M(end) == M(end, end) == 2` .

Вы также можете индексировать несколько элементов с помощью линейной индексации. Обратите внимание: если вы это сделаете, возвращаемая матрица будет иметь ту же форму, что и матрица индексных векторов.

`M(2:4)` возвращает вектор строки, потому что `2:4` представляет вектор строки `[2,3,4]` :

```
>> M(2:4)
```

```
ans =

        3     4     1
```

В качестве другого примера `M([1,2;3,4])` возвращает матрицу 2 на 2, потому что `[1,2;3,4]` является матрицей 2 на 2. Посмотрите приведенный ниже код, чтобы убедить себя:

```
>> M([1,2;3,4])

ans =

       8         3
       4         1
```

Обратите внимание, что индексирование с помощью `:` alone *всегда* будет возвращать вектор-столбец:

```
>> M(:)

ans =

        8
        3
        4
        1
        5
        9
        6
        7
        2
```

Этот пример также иллюстрирует порядок, в котором MATLAB возвращает элементы при использовании линейной индексации.

Логическая индексация

Третий метод индексирования - использовать логическую матрицу, т. Е. Матрицу, содержащую только `true` или `false` значения, в качестве маски для фильтрации элементов, которые вы не хотите. Например, если мы хотим найти все элементы `M` , которые больше `5` мы можем использовать логическую матрицу

```
>> M > 5

ans =

    1     0     1
    0     0     1
    0     1     0
```

для индекса `M` и возвращать только значения, превышающие `5` следующим образом:

```
>> M(M > 5)

ans =

        8
        9
        6
        7
```

Если вы хотите, чтобы этот номер оставался на месте (т. Е. Сохранял форму матрицы), тогда вы могли бы назначить логический комплимент

```
>> M(~(M > 5)) = NaN

ans =

    8      NaN    6
    NaN    NaN    7
    NaN    9      Nan
```

Мы можем уменьшить сложные кодовые блоки, содержащие операторы `if` и `for` , используя логическую индексацию.

Возьмите не-векторизованный (уже сокращенный до одного цикла с помощью линейной индексации):

```
for elem = 1:numel(M)
  if M(elem) > 5
    M(elem) = M(elem) - 2;
  end
end
```

Это можно сократить до следующего кода, используя логическую индексацию:

```
idx = M > 5;
M(idx) = M(idx) - 2;
```

Или даже короче:

```
M(M > 5) = M(M > 5) - 2;
```

Подробнее об индексировании

Более высокие размерные матрицы

Все упомянутые выше методы обобщаются на n-мерные. Если мы используем в качестве примера трехмерную матрицу `M3 = rand(3,3,3)` , то вы можете получить доступ ко всем строкам и столбцам второго среза третьего измерения, написав

```
>> M(:,:,2)
```

Вы можете получить доступ к первому элементу второго среза с помощью линейной индексации. Линейное индексирование будет двигаться только ко второму срезу после всех строк и всех столбцов первого среза. Таким образом, линейный индекс для этого элемента равен

```
>> M(size(M,1)*size(M,2)+1)
```

Фактически, в MATLAB *каждая* матрица n-мерна: просто случается так, что размер большинства других n-измерений один. Итак, если `a = 2` то `a(1) == 2` (как и следовало ожидать), *но также* `a(1, 1) == 2` , как и `a(1, 1, 1) == 2` , `a(1, 1, 1, ..., 1) == 2` и т. Д. Эти «дополнительные» размеры (размером `1`) называются *одноэлементными размерами* . `squeeze` команды удалит их, и можно использовать `permute` чтобы поменять порядок размеров вокруг (и при необходимости ввести размеры синглтона).

N-мерную матрицу можно также индексировать, используя m индексов (где m <= n). Правило состоит в том, что первые индексы m-1 ведут себя обычно, в то время как последний (m'th) индекс ссылается на оставшиеся (n-m + 1) измерения, точно так же, как линейный индекс будет ссылаться на (n-m + 1) меру массив. Вот пример:

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

Возвращаемые диапазоны элементов

При индексировании индексов, если вы указываете более одного элемента в нескольких измерениях, MATLAB возвращает каждую возможную пару координат. Например, если вы попробуете M ([1,2], [1,3]), MATLAB вернет `M(1,1)` и `M(2,3)` но **также** вернет `M(1,3)` и `M(2,1)` . Это может показаться неинтуитивными , когда вы ищете элементы для списка пар координат , но рассмотрим пример большей матрицы `A = rand(20)` (заметим , теперь `A 20` матрицу с размерностью `20`), где вы хотите получить верхний правый квадрант. В этом случае вместо того, чтобы указывать каждую пару координат в этом квадранте (и этот случай равен `100` парам), вы просто указываете `10` строк и `10` столбцов, которые вы хотите, так что `A(1:10, 11:end)` . *Нарезка* такой матрицы гораздо более распространена, чем запрос списка пар координат.

В случае, если вы хотите получить список пар координат, самым простым решением является преобразование в линейную индексацию. Рассмотрим проблему, в которой у вас есть вектор индексов столбцов, который вы хотите вернуть, где каждая строка вектора

содержит номер столбца, который вы хотите вернуть для *соответствующей* строки матрицы. Например

```
colIdx = [3;2;1]
```

Поэтому в этом случае вы действительно хотите вернуть элементы в `(1,3)` , `(2,2)` и `(3,1)` . Таким образом, используя линейное индексирование:

```
>> colIdx = [3;2;1];
>> rowIdx = 1:length(colIdx);
>> idx = sub2ind(size(M), rowIdx, colIdx);
>> M(idx)

ans =

        6     5     4
```

Возвращение элемента несколько раз

С индексом и линейной индексацией вы также можете вернуть элемент несколько раз, повторяя его индекс так

```
>> M([1,1,1,2,2,2])

ans =

        8     8     8     3     3     3
```

Вы можете использовать это, чтобы дублировать целые строки и столбец, например, чтобы повторить первую строку и последний столбец

```
>> M([1, 1:end], [1:end, end])

ans =

        8     1     6     6
        8     1     6     6
        3     5     7     7
        4     9     2     2
```

Для получения дополнительной информации см. Здесь .

Помощь себе

MATLAB поставляется со многими встроенными сценариями и функциями, которые варьируются от простых умножений до наборов инструментов распознавания изображений. Чтобы получить информацию о функции, которую вы хотите использовать, введите: `help functionname` в командной строке. Давайте возьмем функцию `help` в качестве примера.

Информацию о том, как ее использовать, можно получить, набрав:

```
>> help help
```

в окне команд. Это вернет информацию об использовании функции `help` . Если информация, которую вы ищете, все еще неясна, вы можете попробовать страницу **документации** этой функции. Просто введите:

```
>> doc help
```

в окне команд. Это откроет доступную для просмотра документацию на странице `help` по функциям, в которой содержится вся информация, необходимая для понимания того, как работает «помощь».

Эта процедура работает для всех встроенных функций и символов.

При разработке собственных функций вы можете позволить им иметь свою секцию помощи, добавляя комментарии в верхней части функционального файла или сразу после объявления функции.

Пример для простой функции `multiplyby2` сохраненной в файле `multiplyby2.m`

```
function [prod]=multiplyby2(num)
% function MULTIPLYBY2 accepts a numeric matrix NUM and returns output PROD
% such that all numbers are multiplied by 2

    prod=num*2;
end
```

или же

```
% function MULTIPLYBY2 accepts a numeric matrix NUM and returns output PROD
% such that all numbers are multiplied by 2

function [prod]=multiplyby2(num)
    prod=num*2;
end
```

Это очень полезно, когда вы получаете код недели / месяцев / лет после его написания.

Функция `help` и `doc` предоставляет много информации, чтобы узнать, как использовать эти функции, поможет вам быстро развиваться и эффективно использовать MATLAB.

Чтение ввода и записи

Как и весь язык программирования, Matlab предназначен для чтения и записи в самых разных форматах. Собственная библиотека поддерживает большое количество форматов Text, Image, Video, Audio, Data с большим количеством форматов, включенных в каждое обновление версии. Здесь вы можете посмотреть полный список поддерживаемых форматов файлов и какую функцию использовать для их импорта.

Прежде чем пытаться загрузить файл, вы должны спросить себя, какими должны быть данные, и как вы ожидаете, что компьютер будет организовывать данные для вас. Скажем, у вас есть файл txt / csv в следующем формате:

```
Fruit,TotalUnits,UnitsLeftAfterSale,SellingPricePerUnit
Apples,200,67,$0.14
Bananas,300,172,$0.11
Pineapple,50,12,$1.74
```

Мы видим, что первый столбец находится в формате строк, а второй, третий - Numeric, последний столбец находится в форме Currency. Предположим, мы хотим найти, сколько дохода мы сделали сегодня, используя Matlab, и сначала мы хотим загрузить этот файл txt / csv. После проверки ссылки мы видим, что текстовые и текстовые типы файлов txt обрабатываются с помощью `textscan` . Поэтому мы могли бы попробовать:

```
fileID = fopen('dir/test.txt'); %Load file from dir
C = textscan(fileID,'%s %f %f %s','Delimiter',',','HeaderLines',1); %Parse in the txt/csv
```

где `%s` полагают, что этот элемент является строковым типом, `%f` предполагает, что этот элемент является типом Float и что файл разделен на «,». Опция HeaderLines запрашивает у Matlab пропустить первые N строк, а 1 сразу после этого означает пропустить первую строку (строку заголовка).

Теперь C - это данные, которые мы загрузили, которые находятся в форме массива ячеек из 4 ячеек, каждый из которых содержит столбец данных в файле txt / csv.

Поэтому сначала мы хотим рассчитать, сколько фруктов мы продали сегодня, вычитая третий столбец из второго столбца, это можно сделать:

```
sold = C{2} - C{3}; %C{2} gives the elements inside the second cell (or the second column)
```

Теперь мы хотим умножить этот вектор на цену за единицу, поэтому сначала нам нужно преобразовать этот столбец строк в столбец Numbers, а затем преобразовать его в `cell2mat` матрицу, используя Matlab's `cell2mat` первое, что нам нужно сделать, с значком «$» есть много способов сделать это. Самый прямой способ - использовать простое регулярное выражение:

```
D = cellfun(@(x)(str2num(regexprep(x, '\$',''))), C{4}, 'UniformOutput', false);%cellfun
allows us to avoid looping through each element in the cell.
```

Или вы можете использовать цикл:

```
for t=1:size(C{4},1)
   D{t} = str2num(regexprep(C{4}{t}, '\$',''));
end

E = cell2mat(D)% converts the cell array into a Matrix
```

Функция `str2num` превращает строку, в которой знаки «$» `cell2mat` на числовые типы, а `cell2mat` превращает ячейку числовых элементов в матрицу чисел

Теперь мы можем размножать единицы, продаваемые по цене за единицу:

```
revenue = sold .* E; %element-wise product is denoted by .* in Matlab

totalrevenue = sum(revenue);
```

Ячеистые массивы

Элементы одного и того же класса часто могут быть объединены в массивы (с несколькими редкими исключениями, например, с помощью функций). Числовые скаляры, по умолчанию класса `double` , могут храниться в матрице.

```
>> A = [1, -2, 3.14, 4/5, 5^6; pi, inf, 7/0, nan, log(0)]
A =
   1.0e+04 *
    0.0001   -0.0002    0.0003    0.0001    1.5625
    0.0003       Inf       Inf       NaN      -Inf
```

Символы, которые имеют класс `char` в MATLAB, также могут храниться в массиве с использованием аналогичного синтаксиса. Такой массив похож на строку во многих других языках программирования.

```
>> s = ['MATLAB ','is ','fun']
s =
MATLAB is fun
```

Обратите внимание, что, несмотря на то, что оба они используют скобки `[` и `]` , классы результатов отличаются. Поэтому операции, которые могут быть сделаны на них, также различны.

```
>> whos
  Name      Size            Bytes  Class     Attributes

  A         2x5                80  double
  s         1x13               26  char
```

На самом деле массив `s` не является массивом строк `'MATLAB '` , `'is '` и `'fun'` , это всего лишь одна строка - массив из 13 символов. Вы получите те же результаты, если бы они были определены одним из следующих:

```
>> s = ['MAT','LAB ','is f','u','n'];
>> s = ['M','A','T','L','A','B,' ','i','s',' ','f','u','n'];
```

Обычный вектор MATLAB не позволяет хранить сочетание переменных разных классов или несколько разных строк. Здесь массив `cell` пригодится. Это массив ячеек, каждый из

которых может содержать некоторый объект MATLAB, класс которого может быть различным в каждой ячейке, если это необходимо. Используйте фигурные скобки `{` и `}` вокруг элементов для хранения в массиве ячеек.

```
>> C = {A; s}
C =
    [2x5 double]
    'MATLAB is fun'
>> whos C
  Name      Size             Bytes  Class    Attributes

  C         2x1                330  cell
```

Стандартные объекты MATLAB любых классов могут храниться вместе в массиве ячеек. Обратите внимание, что массивы ячеек требуют больше памяти для хранения их содержимого.

Доступ к содержимому ячейки осуществляется с помощью фигурных скобок `{` и `}` .

```
>> C{1}
ans =
   1.0e+04 *
    0.0001   -0.0002    0.0003    0.0001    1.5625
    0.0003       Inf       Inf       NaN      -Inf
```

Заметим, что `C(1)` отличается от `C{1}` . Принимая во внимание, что последний возвращает содержимое ячейки (и имеет пример с `double` примером), первый возвращает массив ячеек, который является подматрицей `C` Точно так же, если `D` было массивом из 10 на 5 ячеек, тогда `D(4:8,1:3)` вернет подматрицу `D` , размер которой равен 5 на 3, а класс - `cell` . И синтаксис `C{1:2}` не имеет одного возвращенного объекта, но rater он возвращает 2 разных объекта (аналогично функции MATLAB с несколькими возвращаемыми значениями):

```
>> [x,y] = C{1:2}
x =
                         1                        -2                      3.14
0.8                    15625
          3.14159265358979                       Inf                       Inf
NaN                     -Inf
y =
MATLAB is fun
```

Скрипты и функции

Код MATLAB можно сохранить в m-файлах для повторного использования. m-файлы имеют расширение `.m` которое автоматически связано с MATLAB. M-файл может содержать скрипт или функции.

Сценарии

Скрипты - это просто программные файлы, которые выполняют последовательность

команд MATLAB в предопределенном порядке.

Скрипты не принимают входные данные, и сценарии не возвращают выходные данные. Функционально сценарии эквивалентны вводу команд непосредственно в командное окно MATLAB и их повторному воспроизведению.

Пример сценария:

```
length = 10;
width = 3;
area = length * width;
```

Этот скрипт будет определять `length` , `width` и `area` в текущей рабочей области со значениями `10` , `3` и `30` соответственно.

Как указано выше, приведенный выше сценарий функционально эквивалентен вводу одних и тех же команд непосредственно в командное окно.

```
>> length = 10;
>> width = 3;
>> area = length * width;
```

функции

Функции по сравнению со сценариями гораздо более гибкие и расширяемые. В отличие от сценариев, функции могут принимать входные и выходные данные для вызывающего. Функция имеет собственное рабочее пространство, это означает, что внутренние операции функций не изменят переменные от вызывающего.

Все функции определяются с одинаковым форматом заголовков:

```
function [output] = myFunctionName(input)
```

Ключевое слово `function` начинается с каждого заголовка функции. Далее следует список результатов. Список выходов также может быть списком переменных для запятой.

```
function [a, b, c] = myFunctionName(input)
```

Далее следует имя функции, которая будет использоваться для вызова. Это, как правило, то же имя, что и имя файла. Например, мы сохранили бы эту функцию как `myFunctionName.m` .

Следующим именем функции является список входов. Подобно выводам, это также может быть список, разделенный запятыми.

```
function [a, b, c] = myFunctionName(x, y, z)
```

Мы можем переписать пример скрипта из ранее как функцию многократного

использования, например:

```
function [area] = calcRecArea(length, width)
   area = length * width;
end
```

Мы можем вызывать функции из других функций или даже из файлов сценариев. Ниже приведен пример нашей функции, используемой в файле сценария.

```
l = 100;
w = 20;
a = calcRecArea(l, w);
```

Как и раньше, мы создаем `l` , `w` и `a` в рабочей области со значениями `100` , `20` и `2000` соответственно.

Типы данных

В MATLAB имеется 16 основных типов данных или классов. Каждый из этих классов имеет форму матрицы или массива. За исключением дескрипторов функций, эта матрица или массив является как минимум размером 0 на 0 и может вырасти до n-мерного массива любого размера. Функциональный дескриптор всегда скалярный (1 на 1).

Важным моментом в MATLAB является то, что вам не нужно использовать объявления типа или инструкции по размеру по умолчанию. Когда вы определяете новую переменную, MATLAB создает ее автоматически и выделяет соответствующее пространство памяти.

Пример:

```
a = 123;
b = [1 2 3];
c = '123';

>> whos
  Name      Size             Bytes  Class     Attributes

  a         1x1                  8  double
  b         1x3                 24  double
  c         1x3                  6  char
```

Если переменная уже существует, MATLAB заменяет исходные данные на новую и при необходимости выделяет новое пространство для хранения.

Основные типы данных

Основные типы данных: числовые, `logical` , `char` , `cell` , `struct` , `table` и `function_handle` .

Числовые типы данных :

- Номера с плавающей запятой (по *умолчанию*)

  MATLAB представляет числа с плавающей точкой в формате с двойной точностью или с одной точностью. По умолчанию используется двойная точность, но вы можете сделать любое число одинарной точности с простой функцией преобразования:

```
a = 1.23;
b = single(a);

>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x1                 4  single
```

- Целые

  MATLAB имеет четыре подписанных и четыре беззнаковых целочисленных класса. Подписанные типы позволяют работать с отрицательными целыми и позитивными, но не могут представлять собой широкий диапазон чисел, как неподписанные, потому что один бит используется для обозначения положительного или отрицательного знака для числа. Неподписанные типы дают вам более широкий диапазон чисел, но эти цифры могут быть только нулевыми или положительными.

  MATLAB поддерживает 1-, 2-, 4- и 8-байтовые хранилища для целочисленных данных. Вы можете сохранить память и время выполнения для своих программ, если вы используете наименьший целочисленный тип, который поддерживает ваши данные. Например, для хранения значения 100 вам не нужно 32-разрядное целое число.

```
a = int32(100);
b = int8(100);

>> whos
  Name      Size            Bytes  Class    Attributes

  a         1x1                 4  int32
  b         1x1                 1  int8
```

  Чтобы хранить данные как целое число, вам необходимо преобразовать из двойного значения в желаемый целочисленный тип. Если число, преобразованное в целое число, имеет дробную часть, MATLAB округляется до ближайшего целого числа. Если дробная часть равна `0.5` , то из двух одинаково близких целых чисел MATLAB выбирает ту, для которой абсолютное значение больше по величине.

```
a  = int16(456);
```

- `char`

  Массивы символов обеспечивают хранение текстовых данных в MATLAB. В

соответствии с традиционной терминологией программирования массив (последовательность) символов определяется как строка. В розничных выпусках MATLAB нет явного строкового типа.

- логические: логические значения 1 или 0, соответственно соответствуют true и false. Используется для реляционных условий и индексации массивов. Поскольку это только TRUE или FALSE, размер 1 байт.

```
a = logical(1);
```

- состав. Массив структуры - это тип данных, который группирует переменные разных типов данных с использованием контейнеров данных, называемых *полями* . Каждое поле может содержать любые типы данных. Доступ к данным в структуре с использованием точечной нотации формы structName.fieldName.

```
field1 = 'first';
field2 = 'second';
value1 = [1 2 3 4 5];
value2 = 'sometext';
s = struct(field1,value1,field2,value2);
```

Чтобы получить доступ к значению 1, каждый из следующих синтаксисов эквивалентен

```
s.first or s.(field1) or s.('first')
```

Мы можем явно получить доступ к полю, которое, как мы знаем, будут существовать с помощью первого метода или передать строку или создать строку для доступа к полю во втором примере. Третий пример - это демонстрация того, что нотация дескрипторов точек принимает строку, которая является той же самой, что и в переменной поля1.

- переменные таблицы могут быть разных размеров и типов данных, но все переменные должны иметь одинаковое количество строк.

```
Age = [15 25 54]';
Height = [176 190 165]';
Name = {'Mike', 'Pete', 'Steeve'}';
T = table(Name,Age, Height);
```

- клетка. Это очень полезный тип данных MATLAB: массив ячеек - это массив, каждый из которых может иметь разные типы и размер данных. Это очень сильный инструмент для управления данными по вашему желанию.

```
a = { [1 2 3], 56, 'art'};
```

или же

```
a = cell(3);
```

- Функция handles хранит указатель на функцию (например, на анонимную функцию). Он позволяет передать функцию другой функции или вызвать локальные функции извне основной функции.

Существует множество инструментов для работы с каждым типом данных, а также встроенные функции преобразования типа данных (`str2double` , `table2cell`).

Дополнительные типы данных

Существует несколько дополнительных типов данных, которые полезны в некоторых конкретных случаях. Они есть:

- Дата и время: массивы для представления дат, времени и продолжительности. `datetime('now')` возвращается `21-Jul-2016 16:30:16` .

- Категориальные массивы: это тип данных для хранения данных со значениями из набора дискретных категорий. Полезно для хранения нечетных данных (эффективная память). Может использоваться в таблице для выбора групп строк.

```
a = categorical({'a' 'b' 'c'});
```

- Контейнеры карт представляют собой структуру данных, которая обладает уникальной способностью индексировать не только любые скалярные числовые значения, но и вектор символов. Индексы в элементы Карты называются ключами. Эти ключи вместе со значениями данных, связанными с ними, сохраняются в пределах Карты.

- Временные ряды - это векторы данных, отобранные во времени, по порядку, часто через регулярные промежутки времени. Полезно хранить данные, связанные с timesteps, и у него есть много полезных методов для работы.

Анонимные функции и функции

ОСНОВЫ

Анонимные функции являются мощным инструментом языка MATLAB. Это функции, которые существуют локально, то есть: в текущей рабочей области. Однако они не существуют на пути MATLAB, как регулярная функция, например, в m-файле. Вот почему они называются анонимными, хотя они могут иметь имя как переменную в рабочей области.

Оператор `@`

Используйте оператор `@` для создания анонимных функций и функций. Например, чтобы создать дескриптор функции `sin` (sine) и использовать его как `f` :

```
>> f = @sin
f =
    @sin
```

Теперь `f` является дескриптором функции `sin` . Как и в реальной жизни, дверная ручка - это способ использовать дверь, функциональная рукоятка - это способ использовать функцию. Для использования `f` аргументы передаются ему, как если бы это была функция `sin` :

```
>> f(pi/2)
ans =
     1
```

`f` принимает любые входные аргументы, которые принимает функция `sin` . Если `sin` будет функцией, которая принимает нулевые входные аргументы (а это не так, но другие делают, например, функцию `peaks`), `f()` будет использоваться для вызова без входных аргументов.

Пользовательские анонимные функции

Анонимные функции одной переменной

Очевидно, не полезно создавать дескриптор существующей функции, такой как `sin` в приведенном выше примере. В этом примере он является избыточным. Тем не менее, полезно создавать анонимные функции, которые выполняют пользовательские вещи, которые в противном случае нужно было бы повторять несколько раз или создавать отдельную функцию. В качестве примера пользовательской анонимной функции, которая принимает одну переменную в качестве ее ввода, суммируйте синус и косинус в квадрате сигнала:

```
>> f = @(x) sin(x)+cos(x).^2
f =
    @(x)sin(x)+cos(x).^2
```

Теперь `f` принимает один входной аргумент, называемый `x` . Это было задано с помощью круглых скобок `(...)` непосредственно после оператора `@` . `f` теперь является анонимной функцией `x` : `f(x)` . Он используется, передавая значение `x` в `f` :

```
>> f(pi)
ans =
    1.0000
```

Вектор значений или переменная также может быть передан в `f` , если они действительным образом используются внутри `f` :

```
>> f(1:3) % pass a vector to f
ans =
    1.1334    1.0825    1.1212
>> n = 5:7;
>> f(n) % pass n to f
ans =
   -0.8785    0.6425    1.2254
```

Анонимные функции более чем одной переменной

Таким же образом анонимные функции могут быть созданы, чтобы принимать более одной переменной. Пример анонимной функции, которая принимает три переменные:

```
>> f = @(x,y,z) x.^2 + y.^2 - z.^2
f =
    @(x,y,z)x.^2+y.^2-z.^2
>> f(2,3,4)
ans =
    -3
```

Параметрирование анонимных функций

Переменные в рабочей области могут использоваться в определении анонимных функций. Это называется параметризацией. Например, для использования константы `c = 2` в анонимной функции:

```
>> c = 2;
>> f = @(x) c*x
f =
    @(x)c*x
>> f(3)
ans =
     6
```

`f(3)` использовала переменную `c` как параметр для умножения с предоставленным `x` . Обратите внимание, что если значение `c` в этой точке задано на что-то другое, тогда вызывается `f(3)` , результат **не** будет отличаться. Значение `c` является значением *во время создания* анонимной функции:

```
>> c = 2;
>> f = @(x) c*x;
>> f(3)
ans =
     6
>> c = 3;
>> f(3)
ans =
     6
```

Входные аргументы для анонимной функции не относятся к переменным

рабочего пространства

Обратите внимание, что использование имени переменных в рабочей области в качестве одного из входных аргументов анонимной функции (т. Е. С помощью `@(...)`) **не** будет использовать значения этих переменных. Вместо этого они рассматриваются как разные переменные в рамках анонимной функции, то есть: анонимная функция имеет свое личное рабочее пространство, где входные переменные никогда не ссылаются на переменные из основного рабочего пространства. Основное рабочее пространство и рабочая область анонимной функции не знают друг о друге. Пример, иллюстрирующий это:

```
>> x = 3 % x in main workspace
x =
     3
>> f = @(x) x+1; % here x refers to a private x variable
>> f(5)
ans =
     6
>> x
x =
     3
```

Значение `x` из основного рабочего пространства не используется в `f` . Кроме того, в основной рабочей области `x` осталось нетронутым. В пределах области `f` имена переменных между круглыми скобками после оператора `@` не зависят от основных переменных рабочей области.

Анонимные функции хранятся в переменных

Анонимная функция (или, точнее, дескриптор функции, указывающая на анонимную функцию) сохраняется как любое другое значение в текущем рабочем пространстве: в переменной (как мы это делали выше) в массиве ячеек (`{@(x)x.^2,@(x)x+1}`) или даже в свойстве (например, `h.ButtonDownFcn` для интерактивной графики). Это означает, что анонимную функцию можно рассматривать как любое другое значение. Когда он хранится в переменной, он имеет имя в текущей рабочей области и может быть изменен и очищен так же, как переменные, содержащие числа.

По-разному: дескриптор функции (будь то в форме `@sin` или для анонимной функции) - это просто значение, которое может быть сохранено в переменной, подобно цифровой матрице.

Расширенное использование

Передающая функция обрабатывает другие функции

Поскольку дескрипторы функций обрабатываются как переменные, они могут быть переданы в функции, которые принимают дескрипторы функций в качестве входных аргументов.

Пример: функция создается в m-файле, который принимает дескриптор функции и скалярное число. Затем он вызывает дескриптор функции, передавая ему `3` а затем добавляет скалярное число к результату. Результат возвращается.

Содержание `funHandleDemo.m` :

```
function y = funHandleDemo(fun,x)
y = fun(3);
y = y + x;
```

Сохраните его где-нибудь на пути, например, в текущей папке MATLAB. Теперь `funHandleDemo` можно использовать следующим образом, например:

```
>> f = @(x) x^2; % an anonymous function
>> y = funHandleDemo(f,10) % pass f and a scalar to funHandleDemo
y =
    19
```

Рукоятка другой существующей функции может быть передана `funHandleDemo` :

```
>> y = funHandleDemo(@sin,-5)
y =
   -4.8589
```

Обратите внимание, что `@sin` был быстрым способом получить доступ к функции `sin` без предварительного ее хранения в переменной, используя `f = @sin` .

Использование `bsxfun` , `cellfun` и подобных функций с анонимными функциями

MATLAB имеет встроенные функции, которые принимают анонимные функции в качестве входных данных. Это способ выполнить многие вычисления с минимальным количеством строк кода. Например, `bsxfun` , который выполняет `bsxfun` двоичные операции, то есть: он применяет функцию по двум векторам или матрицам `bsxfun` . Обычно для этого требуется использование `for` -loops, для которого часто требуется предварительное распределение скорости. Используя `bsxfun` этот процесс ускоряется. Следующий пример иллюстрирует это, используя `tic` и `toc` , две функции, которые можно использовать для временного использования кода. Он вычисляет разницу между каждым элементом матрицы из среднего значения столбца.

```
A = rand(50); % 50-by-50 matrix of random values between 0 and 1
```

```
% method 1: slow and lots of lines of code
tic
meanA = mean(A); % mean of every matrix column: a row vector
% pre-allocate result for speed, remove this for even worse performance
result = zeros(size(A));
for j = 1:size(A,1)
    result(j,:) = A(j,:) - meanA;
end
toc
clear result % make sure method 2 creates its own result

% method 2: fast and only one line of code
tic
result = bsxfun(@minus,A,mean(A));
toc
```

Выполнение приведенного выше примера приводит к двум выходам:

```
Elapsed time is 0.015153 seconds.
Elapsed time is 0.007884 seconds.
```

Эти строки поступают от функций `toc` , которые печатают прошедшее время со времени последнего вызова функции `tic` .

`bsxfun` применяет функцию в первом аргументе ввода к двум другим входным аргументам. `@minus` - это длинное имя для той же операции, что и знак минуса. Возможно, была указана другая анонимная функция или дескриптор (`@`) для любой другой функции, если она принимает `A` и `mean(A)` качестве входных данных для создания значимого результата.

Специально для больших объемов данных в больших матрицах `bsxfun` может значительно ускорить работу. Он также делает код более чистым, хотя его может быть труднее интерпретировать для людей, которые не знают MATLAB или `bsxfun` . (Обратите внимание, что в MATLAB R2016a и более поздних версиях многие операции, которые ранее использовали `bsxfun` больше не нуждаются в них, `A-mean(A)` работает напрямую и может в некоторых случаях быть еще быстрее.)

Прочитайте Начало работы с языком MATLAB онлайн:
https://riptutorial.com/ru/matlab/topic/235/начало-работы-с-языком-matlab

глава 2: Введение в MEX API

Examples

Проверить количество входов / выходов в MEX-файле C ++

В этом примере мы напишем базовую программу, которая проверяет количество входов и выходов, переданных MEX-функции.

В качестве отправной точки нам нужно создать файл C ++, реализующий «шлюз MEX». Это функция, выполняемая при вызове файла из MATLAB.

testinputs.cpp

```
// MathWorks provided header file
#include "mex.h"

// gateway function
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // This function will error if number of inputs its not 3 or 4
    // This function will error if number of outputs is more than 1

    // Check inputs:
    if (nrhs < 3 || nrhs > 4) {
        mexErrMsgIdAndTxt("Testinputs:ErrorIdIn",
            "Invalid number of inputs to MEX file.");
    }

    // Check outputs:
    if (nlhs > 1) {
        mexErrMsgIdAndTxt("Testinputs:ErrorIdOut",
                "Invalid number of outputs to MEX file.");
    }
}
```

Во-первых, мы `mex.h` заголовок `mex.h` который содержит определения всех необходимых функций и типов данных для работы с API MEX. Затем мы реализуем функцию `mexFunction` как показано на рисунке, где ее подпись не должна меняться независимо от реально используемых входов / выходов. Параметры функции следующие:

- `nlhs` : количество запрошенных результатов.
- `*plhs[]` : массив, содержащий все выходы в формате API MEX.
- `nrhs` : Количество пропущенных данных.
- `*prhs[]` : массив, содержащий все входы в формате MEX API.

Затем мы проверяем количество аргументов ввода / вывода, и если проверка не выполняется, возникает ошибка с использованием функции `mexErrMsgIdAndTxt` (она ожидает

идентификатор имени `somename:iD` , простой «ID» не будет работать).

После того как файл скомпилирован как `mex testinputs.cpp` , функцию можно вызвать в MATLAB как:

```
>> testinputs(2,3)
Error using testinputs. Invalid number of inputs to MEX file.

>> testinputs(2,3,5)

>> [~,~] = testinputs(2,3,3)
Error using testinputs. Invalid number of outputs to MEX file.
```

Введите строку, измените ее на C и выведите ее

В этом примере мы иллюстрируем манипуляции с строками в MATLAB MEX. Мы создадим MEX-функцию, которая принимает строку как входную информацию от MATLAB, скопирует данные в C-строку, `mxArray` ее и преобразует обратно в `mxArray` возвращенный стороне MATLAB.

Основная цель этого примера - показать, как строки могут быть преобразованы в C / C ++ из MATLAB и наоборот.

stringIO.cpp

```
#include "mex.h"
#include <cstring>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // check number of arguments
    if (nrhs != 1 || nlhs > 1) {
        mexErrMsgIdAndTxt("StringIO:WrongNumArgs", "Wrong number of arguments.");
    }

    // check if input is a string
    if (mxIsChar(prhs[0])) {
        mexErrMsgIdAndTxt("StringIO:TypeError", "Input is not a string");
    }

    // copy characters data from mxArray to a C-style string (null-terminated)
    char *str = mxArrayToString(prhs[0]);

    // manipulate the string in some way
    if (strcmp("theOneString", str) == 0) {
        str[0] = 'T';  // capitalize first letter
    } else {
        str[0] = ' ';  // do something else?
    }

    // return the new modified string
    plhs[0] = mxCreateString(str);
```

```
    // free allocated memory
    mxFree(str);
}
```

Соответствующие функции в этом примере:

- `mxIsChar` чтобы проверить, имеет `mxArray` `mxCHAR` тип `mxCHAR` .
- `mxArrayToString` для копирования данных строки `mxArray` в буфер `char *` .
- `mxCreateString` для создания строки `mxArray` из `char*` .

В качестве примечания, если вы хотите только прочитать строку и не изменять ее, не забудьте объявить ее как `const char*` для скорости и надежности.

Наконец, после компиляции мы можем назвать это из MATLAB следующим образом:

```
>> mex stringIO.cpp

>> strOut = stringIO('theOneString')
strOut =
TheOneString

>> strOut = stringIO('somethingelse')
strOut=
omethingelse
```

Передайте трехмерную матрицу от MATLAB до C

В этом примере мы проиллюстрируем, как взять двойную трехмерную матрицу реального типа из MATLAB и передать ее в массив C `double*` .

Основные цели этого примера - показать, как получить данные из массивов MATLAB MEX и выделить некоторые мелкие детали в хранении и обработке матриц.

matrixIn.cpp

```
#include "mex.h"

void mexFunction(int  nlhs , mxArray *plhs[],
        int nrhs, mxArray const *prhs[]){
   // check amount of inputs
   if (nrhs!=1) {
        mexErrMsgIdAndTxt("matrixIn:InvalidInput", "Invalid number of inputs to MEX file.");
    }

   // check type of input
   if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])){
        mexErrMsgIdAndTxt("matrixIn:InvalidType",  "Input matrix must be a double, non-complex
array.");
   }
```

```
   // extract the data
   double const * const matrixAux= static_cast<double const *>(mxGetData(prhs[0]));

   // Get matrix size
   const mwSize *sizeInputMatrix= mxGetDimensions(prhs[0]);

   // allocate array in C. Note: its 1D array, not 3D even if our input is 3D
   double*  matrixInC= (double*)malloc(sizeInputMatrix[0] *sizeInputMatrix[1]
*sizeInputMatrix[2]* sizeof(double));


   // MATLAB is column major, not row major (as C). We need to reorder the numbers
   // Basically permutes dimensions

   // NOTE: the ordering of the loops is optimized for fastest memory access!
   // This improves the speed in about 300%

    const int size0 = sizeInputMatrix[0]; // Const makes compiler optimization kick in
    const int size1 = sizeInputMatrix[1];
    const int size2 = sizeInputMatrix[2];

    for (int j = 0; j < size2; j++)
    {
        int jOffset = j*size0*size1; // this saves re-computation time
        for (int k = 0; k < size0; k++)
        {
            int kOffset = k*size1; // this saves re-computation time
            for (int i = 0; i < size1; i++)
            {
                int iOffset = i*size0;
                matrixInC[i + jOffset + kOffset] = matrixAux[iOffset + jOffset + k];
            }
        }
    }

    // we are done!

    // Use your C matrix here

    // free memory
    free(matrixInC);
    return;
}
```

Соответствующие концепции, которые необходимо знать:

- Матрицы MATLAB все 1D в памяти, независимо от того, сколько измерений они имеют при использовании в MATLAB. Это также справедливо для большинства (если не всех) основных матричных представлений в библиотеках C / C ++, что позволяет оптимизировать и ускорить выполнение.
- Вам нужно явно скопировать матрицы из MATLAB в C в цикле.
- Матрицы MATLAB хранятся в главном порядке столбцов, как в Fortran, но C / C ++ и большинство современных языков являются основными. Важно переставить входную

матрицу, иначе данные будут выглядеть совершенно иначе.

Соответствующая функция в этом примере:

- `mxIsDouble` проверяет, является ли вход `double` .
- `mxIsComplex` проверяет, является ли вход реальным или мнимым.
- `mxGetData` возвращает указатель на реальные данные во входном массиве. `NULL` если реальных данных нет.
- `mxGetDimensions` возвращает указатель на массив `mwSize` с размером измерения в каждом индексе.

Передача структуры по именам полей

В этом примере показано, как читать записи типа MATLAB с различными типами и передавать их переменным типа C.

Хотя из примера можно легко и легко вычислить, как загружать поля по номерам, это достигается путем сравнения имен полей со строками. Таким образом, поля struct могут быть адресованы их именами полей, а переменные в нем могут быть прочитаны C.

structIn.c

```
#include "mex.h"
#include <string.h> // strcmp


void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  // helpers
  double* double_ptr;
  unsigned int i; // loop variable

  // to be read variables
  bool optimal;
  int randomseed;
  unsigned int desiredNodes;

  if (!mxIsStruct(prhs[0])) {
    mexErrMsgTxt("First argument has to be a parameter struct!");
  }
  for (i=0; i<mxGetNumberOfFields(prhs[0]); i++) {
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"randomseed")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      randomseed = *mxGetPr(p);
    }
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"optimal")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      optimal = (bool)*mxGetPr(p);
    }
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"numNodes")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      desiredNodes = *mxGetPr(p);
```

```
    }
  }
}
```

Цикл над `i` пробегает все поля в данной структуре, а `if(0==strcmp)` сравнивают имя поля matlab с данной строкой. Если это совпадение, соответствующее значение извлекается в переменную C.

Прочитайте Введение в MEX API онлайн: https://riptutorial.com/ru/matlab/topic/680/введение-в-mex-api

глава 3: Векторизация

Examples

Элементные операции

MATLAB поддерживает (и поощряет) векторизованные операции над векторами и матрицами.
Например, предположим, что мы имеем `A` и `B` , две `n m` матрицы и хотим, чтобы `C` являлось элементарным произведением соответствующих элементов (т. Е. `C(i,j) = A(i,j)*B(i,j)`).

Не-векторизованный способ с использованием вложенных циклов выглядит следующим образом:

```
C = zeros(n,m);
for ii=1:n
    for jj=1:m
        C(ii,jj) = A(ii,jj)*B(ii,jj);
    end
end
```

Однако векторизованный способ выполнения этого заключается в использовании элементарного оператора `.*` :

```
C = A.*B;
```

- Для получения дополнительной информации об умножении элемента в MATLAB см. Документацию о `times` .
- Дополнительные сведения о различиях между операциями массива и матрицы см. В разделе « Массивные и матричные операции» в документации MATLAB.

Сумма, средняя, prod & co

Для случайного вектора

```
v = rand(10,1);
```

если вы хотите получить сумму своих элементов, **НЕ** используйте цикл

```
s = 0;
for ii = 1:10
    s = s + v(ii);
end
```

но использовать векторизованную способность функции `sum()`

```
s = sum(v);
```

Функции, такие как `sum()` , `mean()` , `prod()` и другие, имеют возможность работать непосредственно по строкам, столбцам или другим измерениям.

Например, учитывая случайную матрицу

```
A = rand(10,10);
```

среднее значение для каждого **столбца** равно

```
m = mean(A,1);
```

среднее значение для каждой **строки** равно

```
m = mean(A,2)
```

Все вышеперечисленные функции работают только с одним измерением, но что, если вы хотите суммировать всю матрицу? Вы можете использовать:

```
s = sum(sum(A))
```

Но что, если есть ND-массив? применяя `sum` по `sum` на `sum` ... не похоже на лучший вариант, вместо этого используйте оператор `:` vectorize для вашего массива:

```
s = sum(A(:))
```

и это приведет к одному числу, которое является суммой всего вашего массива, не имеет значения, сколько у него размеров.

Использование bsxfun

Довольно часто причина, по которой код был написан в цикле `for` заключается в вычислении значений из «ближайших». Функция `bsxfun` часто может использоваться для этого более сжатым способом.

Например, предположим, что вы хотите выполнить операцию по столбцу на матрице `B` , вычитая из нее среднее значение каждого столбца:

```
B = round(randn(5)*10);                  % Generate random data
A = zeros(size(B));                      % Preallocate array
for col = 1:size(B,2);                    % Loop over columns
    A(:,col) = B(:,col) - mean(B(:,col));   % Subtract means
```

```
end
```

Этот метод неэффективен, если `B` большой, часто из-за того, что MATLAB должен перемещать содержимое переменных в памяти. Используя `bsxfun` , можно выполнить одну и ту же работу аккуратно и легко только в одной строке:

```
A = bsxfun(@minus, B, mean(B));
```

Здесь `@minus` является дескриптором функции для оператора `minus` (`-`) и будет применяться между элементами двух матриц `B` и `mean(B)` . Возможны и другие функции, даже пользовательские.

Затем предположим, что вы хотите добавить вектор строки `v` в каждую строку в матрице `A` :

```
v = [1,  2,  3];

A = [8,  1,  6
     3,  5,  7
     4,  9,  2];
```

Наивный подход - использовать цикл (***не*** *делайте этого*):

```
B = zeros(3);
for row = 1:3
    B(row,:) = A(row,:) + v;
end
```

Другим вариантом будет репликация `v` с помощью `repmat` (***не*** *делайте этого*):

```
>> v = repmat(v,3,1)
v =
     1     2     3
     1     2     3
     1     2     3

>> B = A + v;
```

Вместо этого используйте `bsxfun` для этой задачи:

```
>> B = bsxfun(@plus, A, v);
B =
     9     3     9
     4     7    10
     5    11     5
```

Синтаксис

```
bsxfun(@fun, A, B)
```

где `@fun` - одна из поддерживаемых функций, а два массива `A` и `B` соответствуют двум условиям ниже.

Имя `bsxfun` помогает понять, как работает эта функция, и это означает, что **B** inary **FUN** ction с **S** ingleton e **X** pansion. Другими словами, если:

1. два массива имеют одинаковые размеры, кроме одного
2. и несогласованный размер является одноточечным (т.е. имеет размер `1`) в любом из двух массивов

то массив с размером Singleton будет расширен, чтобы соответствовать размерности другого массива. После расширения двоичная функция применяется по-разному на двух массивах.

Например, пусть `A` - массив `M` -by- `N` -by `K` а `B` - массив `M` -by- `N` . Во-первых, их первые два измерения имеют соответствующие размеры. Во-вторых, `A` имеет `K` слоев, в то время как `B` имеет неявно только `1` , поэтому он является одиночным. Все **условия** выполнены, и `B` будет реплицироваться в соответствии с 3-м измерением `A`

На других языках это обычно называют *вещанием* и происходит автоматически в Python (numpy) и Octave.

Функция `@fun` должна быть двоичной функцией, означающей, что она должна принимать ровно два входа.

замечания

Внутри `bsxfun` не реплицирует массив и выполняет эффективный цикл.

Логическая маскировка

MATLAB поддерживает использование логической маскировки для выполнения выбора по матрице без использования для циклов или операторов if.

Логическая маска определяется как матрица, состоящая только из `1` и `0` .

Например:

```
mask = [1 0 0; 0 1 0; 0 0 1];
```

является логической матрицей, представляющей единичную матрицу.

Мы можем создать логическую маску, используя предикат для запроса матрицы.

```
A = [1 2 3; 4 5 6; 7 8 9];
B = A > 4;
```

Сначала мы создаем матрицу 3x3, `A` , содержащие цифры от 1 до 9. Затем запрос для значений, которые больше , чем 4 и сохранить результат в виде новой матрицы под названием `A B` .

`B` - логическая матрица формы:

```
B = [0 0 0
     0 1 1
     1 1 1]
```

Или `1` когда предикат `A > 4` был истинным. И `0` когда это было ложно.

Мы можем использовать логические матрицы для доступа к элементам матрицы. Если для выбора элементов используется логическая матрица, в матрице, которую вы выбираете, будут выбраны индексы, в которых `1` появляется в логической матрице.

Используя тот же `B` сверху, мы могли бы сделать следующее:

```
C = [0 0 0; 0 0 0; 0 0 0];
C(B) = 5;
```

Это выберет все элементы `C` где `B` имеет `1` в этом индексе. Затем эти индексы в `C` устанавливаются в `5` .

Наш `C` теперь выглядит так:

```
C = [0 0 0
     0 5 5
     5 5 5]
```

Мы можем уменьшить сложные кодовые блоки, содержащие `if` и `for` , используя логические маски.

Возьмите не-векторизованный код:

```
A = [1 3 5; 7 9 11; 11 9 7];
for j = 1:length(A)
  if A(j) > 5
    A(j) = A(j) - 2;
  end
end
```

Это можно сократить с помощью логической маскировки до следующего кода:

```
A = [1 3 5; 7 9 11; 11 9 7];
B = A > 5;
```

```
A(B) = A(B) - 2;
```

Или даже короче:

```
A = [1 3 5; 7 9 11; 11 9 7];
A(A > 5) = A(A > 5) - 2;
```

Неявное расширение массива (широковещательная передача) [R2016b]

MATLAB R2016b характеризовался обобщением своего механизма скалярного расширения [1], [2], чтобы также поддерживать определенные элементарные операции между *массивами* разных размеров, если их размерность совместима. Операторы, которые поддерживают неявное расширение, равны [1]:

- **Элементно-арифметические операторы:** `+`, `-` `.^` `.*` `.^`, `./` `.^` `.\`.
- **Операторы отношения:** `<`, `<=`, `>`, `>=`, `==`, `~=`.
- **Логические операторы:** `&`, `|`, `xor`.
- **Битовые функции:** `bitand`, `bitor`, `bitxor`.
- **Элементарные математические функции:** `max`, `min`, `mod`, `rem`, `hypot`, `atan2`, `atan2d`.

Вышеупомянутые двоичные операции разрешены между массивами, если они имеют «совместимые размеры». Размеры считаются «совместимыми», когда каждое измерение в одном массиве либо точно равно одной и той же размерности в другом массиве, либо равно `1`. Обратите внимание, что конечные синглтоны (т. Е. Размера `1`) не учитываются MATLAB, хотя теоретически их бесконечное количество. Другими словами - размеры, которые появляются в одном массиве и не отображаются в другом, неявно подходят для автоматического расширения.

Например, в версиях MATLAB **до R2016b** это произойдет:

```
>> magic(3) + (1:3)
Error using  +
Matrix dimensions must agree.
```

Принимая во внимание, что **начиная с R2016b** предыдущая операция будет успешной:

```
>> magic(3) + (1:3)
ans =

     9     3     9
     4     7    10
     5    11     5
```

Примеры совместимых размеров:

Описание	1-й размер массива	2-й размер массива	Размер результата
Вектор и скаляр	`[3x1]`	`[1x1]`	`[3x1]`
Векторы строк и столбцов	`[1x3]`	`[2x1]`	`[2x3]`
Векторная и 2D матрица	`[1x3]`	`[5x3]`	`[5x3]`
Матрицы ND и KD	`[1x3x3]`	`[5x3x1x4x2]`	`[5x3x3x4x2]`

Примеры несовместимых размеров:

Описание	1-й размер массива	2-й размер массива	Возможное обходное решение
Векторы, где размерность является кратной одной и той же размерности в другом массиве.	`[1x2]`	`[1x8]`	`transpose`
Массивы с размерами, кратными друг другу.	`[2x2]`	`[8x8]`	`repmat` , `reshape`
ND, которые имеют правильное количество одноэлементных измерений, но они находятся в неправильном порядке (№ 1).	`[2x3x4]`	`[2x4x3]`	`permute`
ND, которые имеют правильное количество одноэлементных измерений, но они находятся в неправильном порядке (# 2).	`[2x3x4x5]`	`[5x2]`	`permute`

ВАЖНЫЙ:
Код, основанный на этом соглашении, **НЕ** обратно совместим с *любыми* более старыми версиями MATLAB. Поэтому явный вызов `bsxfun` [1] , [2] (который достигает такого же эффекта) должен использоваться, если код должен запускаться на старых версиях MATLAB. Если такой проблемы не существует, заметки о выпуске MATLAB R2016 побуждают пользователей переключаться с `bsxfun` :

По сравнению с использованием `bsxfun` неявное расширение обеспечивает более быструю скорость выполнения, лучшее использование памяти и улучшенную читаемость кода.

Связанные чтения:

- Документация MATLAB на « *Совместимые размеры массивов для основных операций* ».
- Широковещательная передача NumPy [1] , [2] .
- Сравнение скорости вычислений с использованием `bsxfun` против неявного расширения массива .

Получить значение функции из двух или более аргументов

Во многих приложениях необходимо вычислить функцию двух или более аргументов.

Традиционно мы используем `for` -loops. Например, если нам нужно вычислить `f = exp(-x^2-y^2)` (не используйте это, если вам нужно **быстрое моделирование**):

```
% code1
x = -1.2:0.2:1.4;
y = -2:0.25:3;
for nx=1:lenght(x)
   for ny=1:lenght(y)
      f(nx,ny) = exp(-x(nx)^2-y(ny)^2);
   end
end
```

Но векторная версия более элегантная и быстрая:

```
% code2
[x,y] = ndgrid(-1.2:0.2:1.4, -2:0.25:3);
f = exp(-x.^2-y.^2);
```

чем мы можем это визуализировать:

```
surf(x,y,f)
```

Примечание1 - Сетки: Обычно хранилище матриц организовано *по очереди* . Но в MATLAB это *столбцовое* хранилище, как в FORTRAN. Таким образом, есть два Двойники функции `ndgrid` и `meshgrid` в MATLAB для реализации двух вышеупомянутых моделей. Чтобы визуализировать функцию в случае `meshgrid` , мы можем использовать:

```
surf(y,x,f)
```

Примечание2 - Потребление памяти: пусть размер `x` или `y` равен 1000. Таким образом, нам нужно хранить `1000*1000+2*1000 ~ 1e6` элементов для не-векторизованного ***кода1*** . Но нам

нужно `3*(1000*1000) = 3e6` элементов в случае векторизованного ***кода2*** . В 3D-случае (пусть `z` имеет тот же размер, что и `x` или `y`), потребление памяти резко возрастает: `4*(1000*1000*1000)` (~ 32 ГБ для удвоений) в случае векторизованного ***кода2*** против `~1000*1000*1000` (всего ~ 8 ГБ) в случае ***кода1*** . Таким образом, мы должны выбрать либо память, либо скорость.

Прочитайте Векторизация онлайн: https://riptutorial.com/ru/matlab/topic/750/векторизация

глава 4: Графика: 2D и 3D преобразования

Examples

2D-преобразования

В этом примере мы возьмем строчную линию, построенную с использованием `line` и выполняем преобразования на ней. Затем мы будем использовать одни и те же преобразования, но в другом порядке и посмотреть, как это влияет на результаты.

Сначала мы открываем фигуру и устанавливаем некоторые начальные параметры (координаты квадратных точек и параметры преобразования)

```
%Open figure and create axis
Figureh=figure('NumberTitle','off','Name','Transformation Example',...
    'Position',[200 200 700 700]); %bg is set to red so we know that we can only see the axes
Axesh=axes('XLim',[-8 8],'YLim',[-8,8]);

 %Initializing Variables
    square=[-0.5 -0.5;-0.5 0.5;0.5 0.5;0.5 -0.5]; %represented by its vertices
    Sx=0.5;
    Sy=2;
    Tx=2;
    Ty=2;
    teta=pi/4;
```

Затем мы построим матрицы преобразования (масштаб, поворот и перевод):

```
%Generate Transformation Matrix
S=makehgtform('scale',[Sx Sy 1]);
R=makehgtform('zrotate',teta);
T=makehgtform('translate',[Tx Ty 0]);
```

Затем мы рисуем синий суаре:

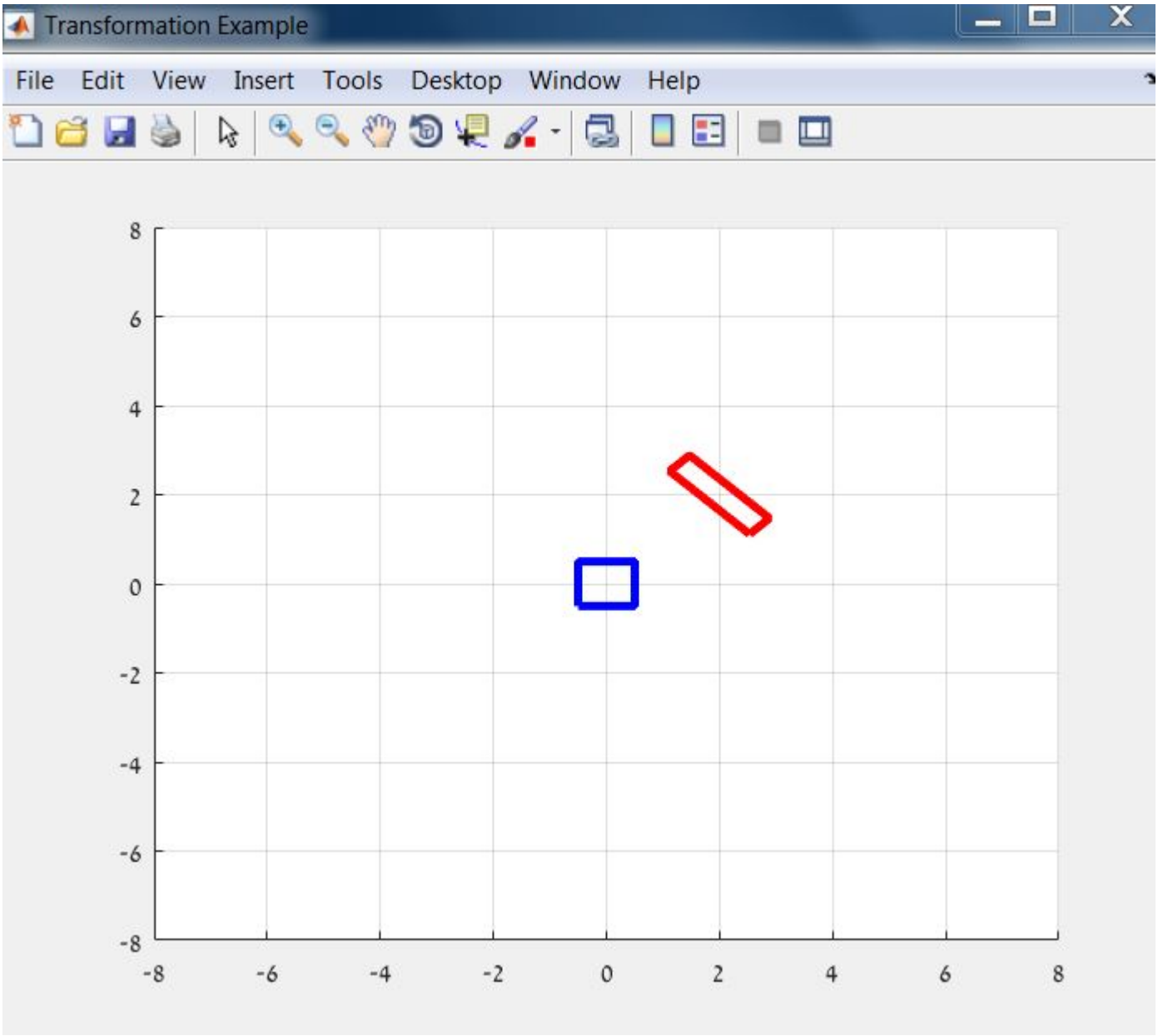
```
%% Plotting the original Blue Square
OriginalSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','b','LineWidth',3);

grid on;    % Applying grid on the figure
hold all;   % Holding all Following  graphs to current axes
```

Затем мы заново построим его в другом цвете (красный) и применим преобразования:

```
%% Plotting the Red Square
%Calculate rectangle vertices
HrectTRS=T*R*S;
RedSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','r','LineWidth',3);
%transformation of the axes
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectTRS);
%seting the line to be a child of transformed axes
```

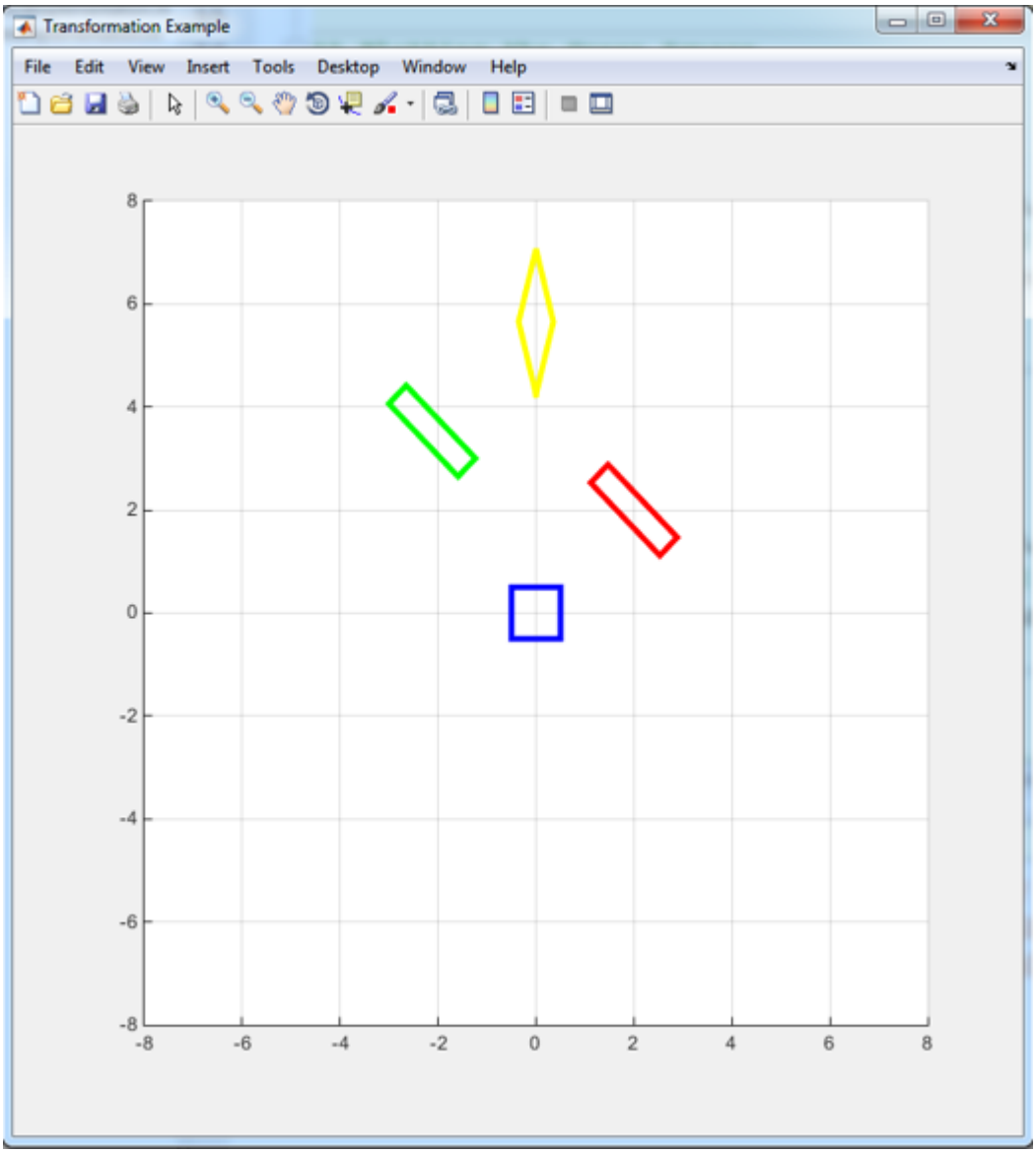
```
set(RedSQ,'Parent',AxesTransformation);
```

Результат должен выглядеть так:



Теперь давайте посмотрим, что произойдет, когда мы изменим порядок преобразования:

```
%% Plotting the Green Square
HrectRST=R*S*T;
GreenSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','g','LineWidth',3);
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectRST);
set(GreenSQ,'Parent',AxesTransformation);

%% Plotting the Yellow Square
HrectSRT=S*R*T;
YellowSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','y','LineWidth',3);
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectSRT);
set(YellowSQ,'Parent',AxesTransformation);
```



Прочитайте Графика: 2D и 3D преобразования онлайн:
https://riptutorial.com/ru/matlab/topic/7418/графика--2d-и-3d-преобразования

глава 5: Графика: 2D-линии

Синтаксис

- участок (Y)
- участок (Y, указатель строки)
- участок (X, Y)
- участок (X, Y, указатель строки)
- график (X1, Y1, X2, Y2, ..., Xn, Yn)
- (X1, Y1, LineSpec1, X2, Y2, LineSpec2, ..., Xn, Yn, LineSpecn)
- plot (___, Name, Value)
- h = график (___)

параметры

параметр	подробности
Икс	x-значение
Y	y-значения
указатель строки	Стиль линии, символ маркера и цвет, указанный в виде строки
Name, Value	Необязательные пары аргументов name-value для настройки свойств линии
час	обрабатывать графический объект

замечания

http://www.mathworks.com/help/matlab/ref/plot.html

Examples

Несколько строк в одном сюжете

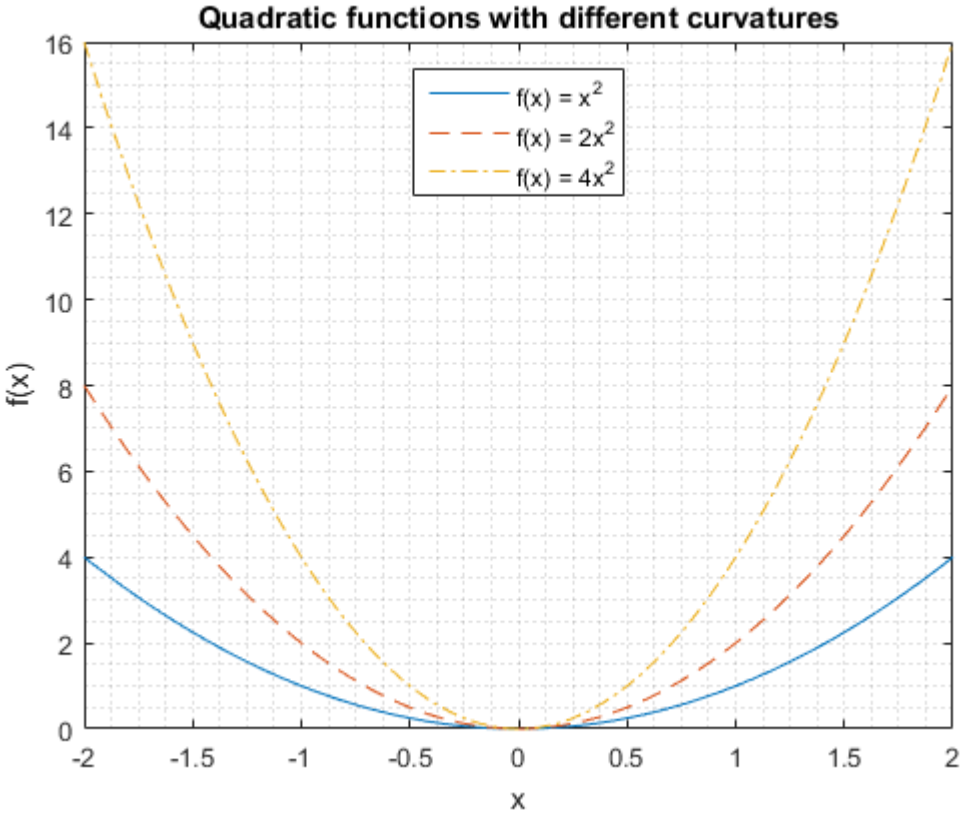
В этом примере мы собираемся построить несколько строк на одну ось. Кроме того, мы выбираем внешний вид линий и создаем легенду.

```
% create sample data
x = linspace(-2,2,100);                 % 100 linearly spaced points from -2 to 2
y1 = x.^2;
y2 = 2*x.^2;
y3 = 4*x.^2;

% create plot
figure;                                 % open new figure
plot(x,y1, x,y2,'--', x,y3,'-.');       % plot lines
grid minor;                             % add minor grid
title('Quadratic functions with different curvatures');
xlabel('x');
ylabel('f(x)');
legend('f(x) = x^2', 'f(x) = 2x^2', 'f(x) = 4x^2', 'Location','North');
```

В приведенном выше примере мы построили линии с помощью одной команды- `plot` . Альтернативой является использование отдельных команд для каждой строки. Нам нужно *удерживать* содержимое фигуры с `hold on` последней, прежде чем добавить вторую строку. В противном случае ранее построенные линии исчезнут из рисунка. Чтобы создать тот же сюжет, что и выше, мы можем использовать следующие команды:

```
figure; hold on;
plot(x,y1);
plot(x,y2,'--');
plot(x,y3,'-.');
```

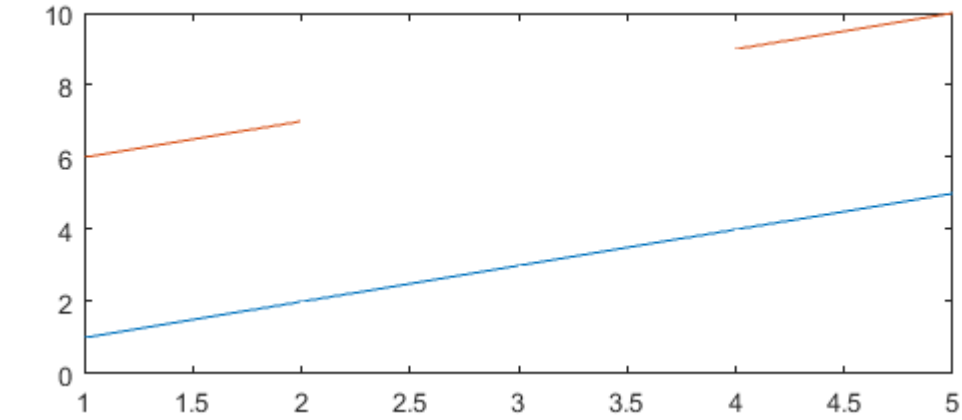
Полученная фигура выглядит так в обоих случаях:



Разделительная линия с NaN

Перемешивайте свои значения `y` или `x` с помощью `NaNs`

```
x = [1:5; 6:10]';

x(3,2) = NaN
x =
     1     6
     2     7
     3   NaN
     4     9
     5    10

plot(x)
```

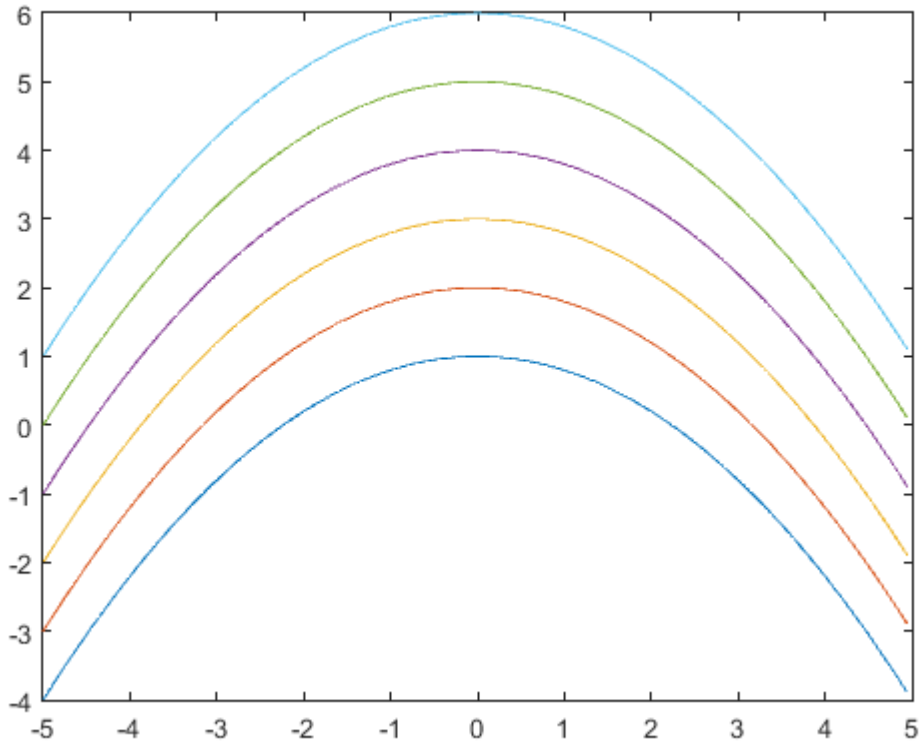


Пользовательские заказы цветов и линий

В MATLAB мы можем установить новые пользовательские заказы по *умолчанию* , такие как порядок цветов и порядок стиля линии. Это означает, что новые заказы будут применены к любой фигуре, созданной после применения этих параметров. Новые настройки сохраняются до тех пор, пока сеанс MATLAB не будет закрыт или не будут выполнены новые настройки.

Цвет по умолчанию и стиль линии

По умолчанию MATLAB использует несколько разных цветов и только сплошной стиль линии. Поэтому, если `plot` вызывается для рисования нескольких строк, MATLAB чередует цветный порядок, чтобы рисовать линии разных цветов.



Мы можем получить порядок цветов по умолчанию, вызвав `get` с глобальным дескриптором `0` за которым следует этот атрибут `DefaultAxesColorOrder` :

```
>> get(0, 'DefaultAxesColorOrder')
ans =
         0    0.4470    0.7410
    0.8500    0.3250    0.0980
    0.9290    0.6940    0.1250
    0.4940    0.1840    0.5560
    0.4660    0.6740    0.1880
    0.3010    0.7450    0.9330
    0.6350    0.0780    0.1840
```

Пользовательский порядок цветов и линий

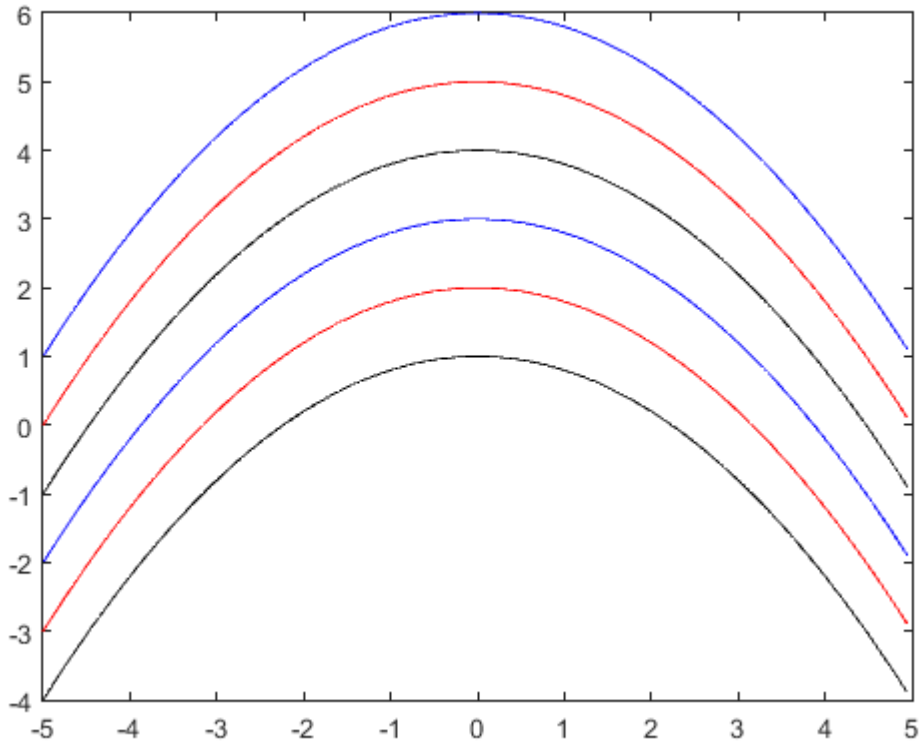
После того, как мы решили установить пользовательский порядок порядка цветов и стиль линии, MATLAB должен чередовать оба. Первое изменение MATLAB применяется в цвете. Когда все цвета исчерпаны, MATLAB применяет следующий стиль линии из определенного порядка стиля линии и устанавливает индекс цвета на 1. Это означает, что MATLAB начнет чередовать все цвета снова, но используя следующий стиль линии в своем порядке. Когда все стили линий и цвета исчерпаны, очевидно, что MATLAB начинает цикл с самого начала, используя первый цвет и стиль первой линии.

В этом примере я определил входной вектор и анонимную функцию, чтобы сделать рисунки немного проще:

```
F = @(a,x) bsxfun(@plus, -0.2*x(:).^2, a);
x = (-5:5/100:5-5/100)';
```

Чтобы установить новый цвет или новые порядки стиля линии, мы вызываем функцию `set` с глобальным дескриптором `0` за которым следует атрибут `DefaultAxesXXXXXXX` ; `XXXXXXX` может быть либо `ColorOrder` либо `LineStyleOrder` . Следующая команда устанавливает новый порядок цветов в черный, красный и синий соответственно:

```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
plot(x, F([1 2 3 4 5 6],x));
```

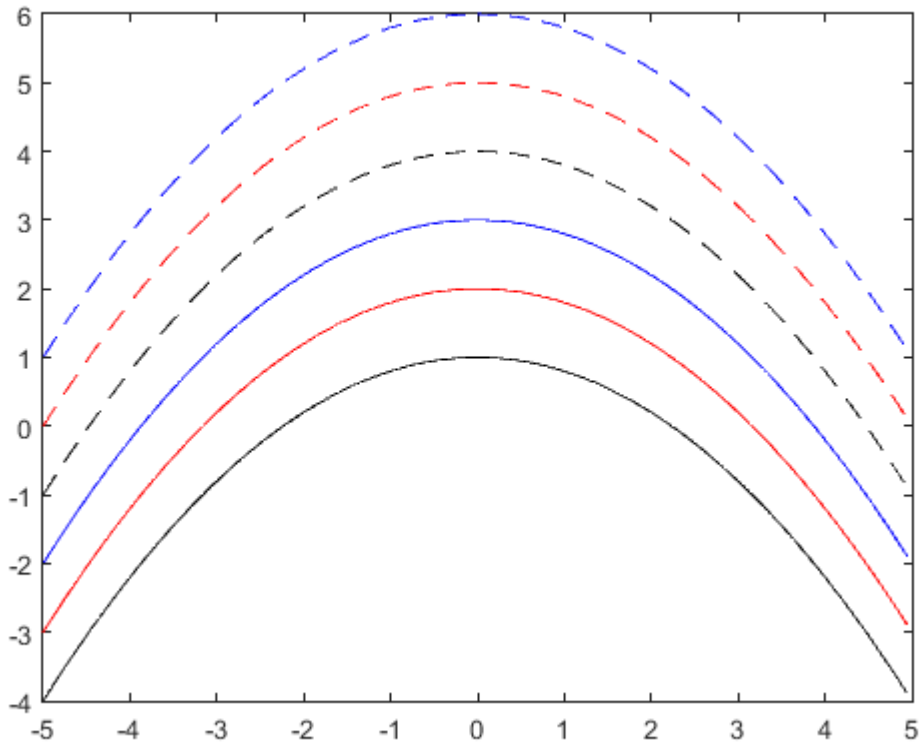


Как вы можете видеть, MATLAB чередуется только через цвета, потому что порядок

строкового стиля по умолчанию задан сплошной линией. Когда набор цветов исчерпан, MATLAB начинается с первого цвета в цветовом порядке.

Следующие команды задают как цветные, так и строковые порядки:

```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
set(0, 'DefaultAxesLineStyleOrder', {'-' '--'});
plot(x, F([1 2 3 4 5 6],x));
```



Теперь MATLAB чередуется с различными цветами и разными стилями линий, используя цвет как наиболее часто используемый атрибут.

Прочитайте Графика: 2D-линии онлайн: https://riptutorial.com/ru/matlab/topic/426/графика--2d-линии

глава 6: Для петель

замечания

Итерация над вектором столбца

Общий источник ошибок пытается перебрать элементы вектора столбца. Вектор столбца обрабатывается как матрица с одним столбцом. (В Matlab фактически нет различий.) Цикл `for` выполняется один раз, когда переменная цикла устанавливается в столбец.

```
% Prints once: [3, 1]
my_vector = [1; 2; 3];
for i = my_vector
    display(size(i))
end
```

Изменение переменной итерации

Изменение переменной итерации изменяет ее значение для текущей итерации, но не влияет на ее значение в последующих итерациях.

```
% Prints 1, 2, 3, 4, 5
for i = 1:5
    display(i)
    i = 5; % Fail at trying to terminate the loop
end
```

Специальные характеристики `a:b` в правой части

В базовом примере рассматривается `1:n` как обычный экземпляр создания вектора строки, а затем итерации по нему. По соображениям производительности Matlab фактически обрабатывает любые `a:b` или `a:c:b` специально, не создавая вектор строки полностью, а вместо этого создавая каждый элемент по одному за раз.

Это можно обнаружить, слегка изменив синтаксис.

```
% Loops forever
for i = 1:1e50
end
```

```
% Crashes immediately
for i = [1:1e50]
end
```

Examples

Петля от 1 до n

Самый простой случай - это просто выполнить задачу для фиксированного известного количества раз. Скажем, мы хотим отображать числа от 1 до n, мы можем написать:

```
n = 5;
for k = 1:n
    display(k)
end
```

Цикл будет выполнять внутренний оператор (ы), все между символами `for` и `end` , в течение `n` раз (5 в этом примере):

```
1
2
3
4
5
```

Вот еще один пример:

```
n = 5;
for k = 1:n
    disp(n-k+1:-1:1) % DISP uses more "clean" way to print on the screen
end
```

на этот раз мы используем как `n` и `k` в цикле, чтобы создать «вложенный» дисплей:

```
 5     4     3     2     1

 4     3     2     1

 3     2     1

 2     1

 1
```

Итерации над элементами вектора

Правая часть задания в цикле `for` может быть любым вектором строки. Левая часть задания может быть любым допустимым именем переменной. Цикл `for` присваивает другому элементу этого вектора переменной каждый пробег.

```
other_row_vector = [4, 3, 5, 1, 2];
for any_name = other_row_vector
    display(any_name)
end
```

Вывод будет отображаться

```
4
3
5
1
2
```

(Версия `1:n` является нормальным случаем этого, потому что в Matlab `1:n` - просто синтаксис для построения вектор-строки из `[1, 2, ..., n]` .)

Следовательно, два следующих блока кода идентичны:

```
A = [1 2 3 4 5];
for x = A
  disp(x);
end
```

а также

```
for x = 1:5
  disp(x);
end
```

И следующие идентичны:

```
A = [1 3 5 7 9];
for x = A
  disp(x);
end
```

а также

```
for x = 1:2:9
  disp(x);
end
```

Любой вектор строки будет делать. Они не должны быть цифрами.

```
my_characters = 'abcde';
for my_char = my_characters
    disp(my_char)
end
```

выйдет

```
a
b
c
d
e
```

Итерация по столбцам матрицы

Если правая часть присваивания является матрицей, то в каждой итерации переменной присваиваются последующие столбцы этой матрицы.

```
some_matrix = [1, 2, 3; 4, 5, 6]; % 2 by 3 matrix
for some_column = some_matrix
    display(some_column)
end
```

(Версия вектора строки является нормальным случаем этого, потому что в Matlab вектор строки является просто матрицей, столбцы которой имеют размер 1.)

Вывод будет отображаться

```
1
4
2
5
3
6
```

т.е. каждый столбец отображенной итерационной матрицы отображается, каждый столбец печатается при каждом вызове `display` .

Перечислять индексы

```
my_vector = [0, 2, 1, 3, 9];
for i = 1:numel(my_vector)
    my_vector(i) = my_vector(i) + 1;
end
```

Наиболее простые вещи, выполняемые `for` циклов, могут выполняться быстрее и проще с помощью векторизованных операций. Например, приведенный выше цикл можно заменить на `my_vector = my_vector + 1` .

Вложенные петли

Петли могут быть вложенными, чтобы преформировать итерированную задачу в другую итерационную задачу. Рассмотрим следующие петли:

```
ch = 'abc';
m = 3;
```

```
for c = ch
    for k = 1:m
        disp([c num2str(k)]) % NUM2STR converts the number stored in k to a charachter,
                             % so it can be concataneted with the letter in c
    end
end
```

мы используем 2 итератора для отображения всех комбинаций элементов из `abc` и `1:m` , что дает:

```
a1
a2
a3
b1
b2
b3
c1
c2
c3
```

Мы также можем использовать вложенные циклы для объединения между задачами, которые должны выполняться каждый раз, и задачи, которые необходимо выполнить один раз в нескольких итерациях:

```
N = 10;
n = 3;
a1 = 0; % the first element in Fibonacci series
a2 = 1; % the secound element in Fibonacci series
for j = 1:N
    for k = 1:n
        an = a1 + a2; % compute the next element in Fibonacci series
        a1 = a2;      % save the previous element for the next iteration
        a2 = an;      % save ht new element for the next iteration
    end
    disp(an) % display every n'th element
end
```

Здесь мы хотим вычислить все ряды Фибоначчи , но каждый раз отображать только `n` й элемент, поэтому мы получаем

```
   3
   13
   55
   233
   987
   4181
   17711
   75025
   317811
   1346269
```

Другое дело, что мы можем использовать первый (внешний) итератор во внутреннем цикле. Вот еще один пример:

```
N = 12;
gap = [1 2 3 4 6];
for j = gap
    for k = 1:j:N
        fprintf('%d ',k) % FPRINTF prints the number k proceeding to the next the line
    end
    fprintf('\n')        % go to the next line
end
```

На этот раз мы используем вложенный цикл для форматирования вывода и торможем линию только тогда, когда был введен новый пробел (`j`) между элементами. Мы пробиваем ширину зазора во внешнем контуре и используем его внутри внутреннего цикла для итерации по вектору:

```
1 2 3 4 5 6 7 8 9 10 11 12
1 3 5 7 9 11
1 4 7 10
1 5 9
1 7
```

Примечание. Странные те же самые вложенные контуры.

Это не то, что вы увидите в других средах программирования. Я наткнулся на это несколько лет назад, и я не мог понять, почему это происходит, но после некоторого времени работы с MATLAB я смог понять это. Посмотрите на фрагмент кода ниже:

```
for x = 1:10
    for x = 1:10
        fprintf('%d,', x);
    end
    fprintf('\n');
end
```

вы не ожидаете, что это будет работать должным образом, но это произойдет, производя следующий вывод:

```
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
```

Причина в том, что, как и все остальное в MATLAB, счетчик `x` также является матрицей - вектором, который будет точным. Таким образом, `x` является только ссылкой на «массив» (когерентная, последовательная структура памяти), которая соответствующим образом ссылается на каждый последующий цикл (вложенный или нет). Тот факт, что вложенный

цикл использует один и тот же идентификатор, не имеет никакого отношения к тому, как ссылаются значения из этого массива. Единственная проблема заключается в том, что внутри вложенного цикла внешний `x` скрывается вложенным (локальным) `x` и поэтому не может быть ссылкой. Однако функциональность структуры вложенных циклов остается неповрежденной.

Прочитайте Для петель онлайн: https://riptutorial.com/ru/matlab/topic/1927/для-петель

глава 7: Инициализация матриц или массивов

Вступление

Matlab имеет три важные функции для создания матриц и установки их элементов в нули, единицы или единичную матрицу. (Единичная матрица имеет одну на главной диагонали и нули в другом месте).

Синтаксис

- Z = нули (sz, тип данных, тип массива)
- X = единицы (sz, тип данных)
- I = глаз (sz, тип данных)

параметры

параметр	подробности
С.З.	n (для nxn-матрицы)
С.З.	n, m (для матрицы nxm)
С.З.	m, n, ..., k (для матрицы m-by-n-by -...-by-k)
тип данных	'double' (по умолчанию), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64' или 'uint64'
arraytype	«Распределенная»
arraytype	'Codistributed'
arraytype	'GpuArray'

замечания

По умолчанию эти функции создадут матрицу двойников.

Examples

Создание матрицы из 0s

```
z1 = zeros(5); % Create a 5-by-5 matrix of zeroes
z2 = zeros(2,3); % Create a 2-by-3 matrix
```

Создание матрицы из 1s

```
o1 = ones(5); % Create a 5-by-5 matrix of ones
o2 = ones(1,3); % Create a 1-by-3 matrix / vector of size 3
```

Создание единичной матрицы

```
i1 = eye(3); % Create a 3-by-3 identity matrix
i2 = eye(5,6); % Create a 5-by-6 identity matrix
```

Прочитайте Инициализация матриц или массивов онлайн:
https://riptutorial.com/ru/matlab/topic/8049/инициализация-матриц-или-массивов

глава 8: интеграция

Examples

Интеграл, интеграл 2, интеграл 3

1 мерный

Интегрировать одномерную функцию

```
f = @(x) sin(x).^3 + 1;
```

в пределах диапазона

```
xmin = 2;
xmax = 8;
```

можно вызвать функцию

```
q = integral(f,xmin,xmax);
```

также возможно установить границы для относительных и абсолютных ошибок

```
q = integral(f,xmin,xmax, 'RelTol',10e-6, 'AbsTol',10-4);
```

2-мерный

Если нужно интегрировать двумерную функцию

```
f = @(x,y) sin(x).^y ;
```

в пределах диапазона

```
xmin = 2;
xmax = 8;
ymin = 1;
ymax = 4;
```

один вызывает функцию

```
q = integral2(f,xmin,xmax,ymin,ymax);
```

Как и в другом случае, можно ограничить допуски

```
q = integral2(f,xmin,xmax,ymin,ymax, 'RelTol',10e-6, 'AbsTol',10-4);
```

Трехмерный

Интегрирование трехмерной функции

```
f = @(x,y,z) sin(x).^y - cos(z) ;
```

в пределах диапазона

```
xmin = 2;
xmax = 8;
ymin = 1;
ymax = 4;
zmin = 6;
zmax = 13;
```

выполняется путем вызова

```
q = integral3(f,xmin,xmax,ymin,ymax, zmin, zmax);
```

Опять же, можно ограничить допуски

```
q = integral3(f,xmin,xmax,ymin,ymax, zmin, zmax, 'RelTol',10e-6, 'AbsTol',10-4);
```

Прочитайте интеграция онлайн: https://riptutorial.com/ru/matlab/topic/7022/интеграция

глава 9: Интерполяция с MATLAB

Синтаксис

1. zy = interp1 (x, y);
2. zy = interp1 (x, y, 'method');
3. zy = interp1 (x, y, 'метод', 'экстраполяция');
4. zy = interp1 (x, y, zx);
5. zy = interp1 (x, y, zx, 'method');
6. zy = interp1 (x, y, zx, 'метод', 'экстраполяция');

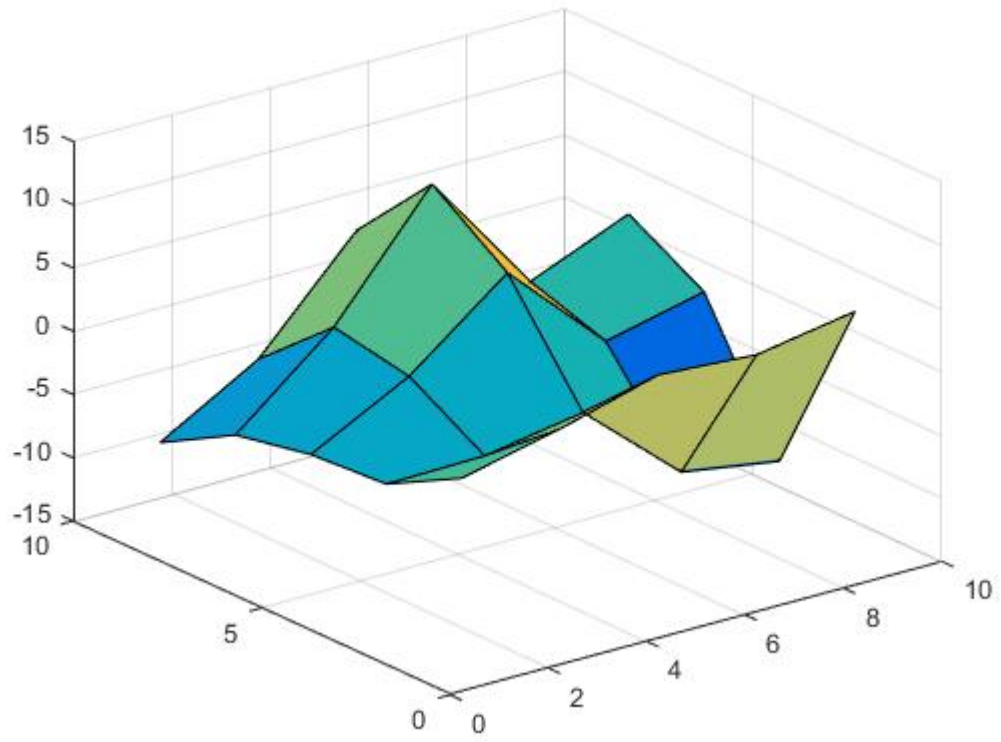
Examples

Кусочно-интерполяционная 2-мерная

Мы инициализируем данные:

```
[X,Y] = meshgrid(1:2:10);
Z = X.*cos(Y) - Y.*sin(X);
```

Поверхность выглядит следующим образом.

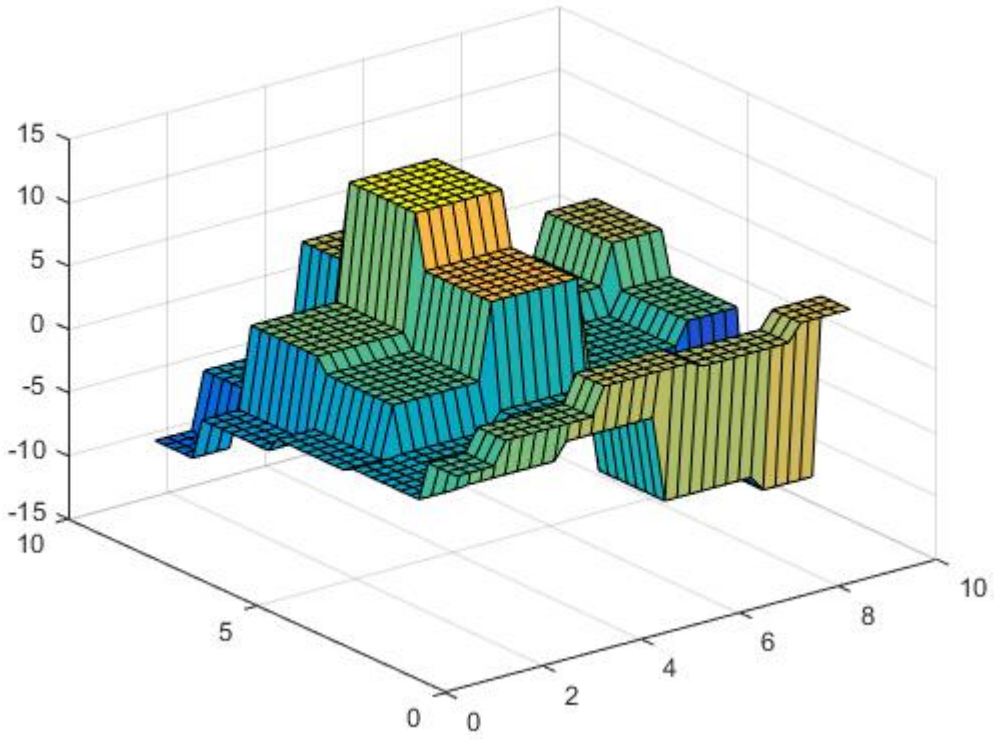


Теперь мы устанавливаем точки, в которые мы хотим интерполировать:

```
[Vx,Vy] = meshgrid(1:0.25:10);
```

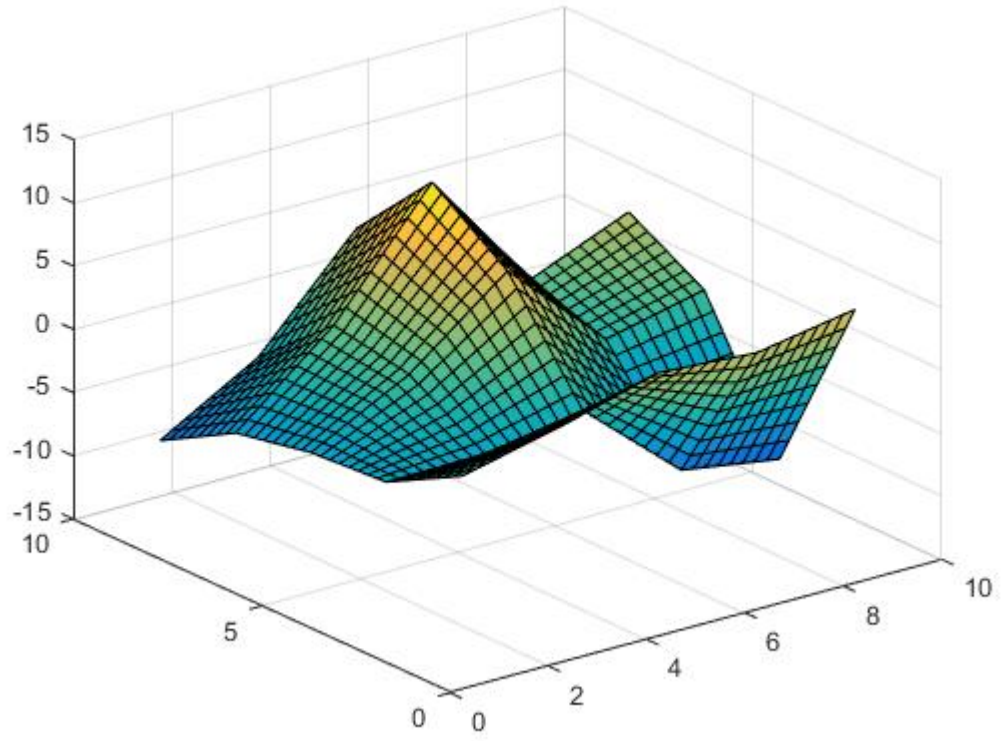
Теперь мы можем выполнить ближайшую интерполяцию,

```
Vz = interp2(X,Y,Z,Vx,Vy,'nearest');
```



линейная интерполяция,

```
Vz = interp2(X,Y,Z,Vx,Vy,'linear');
```

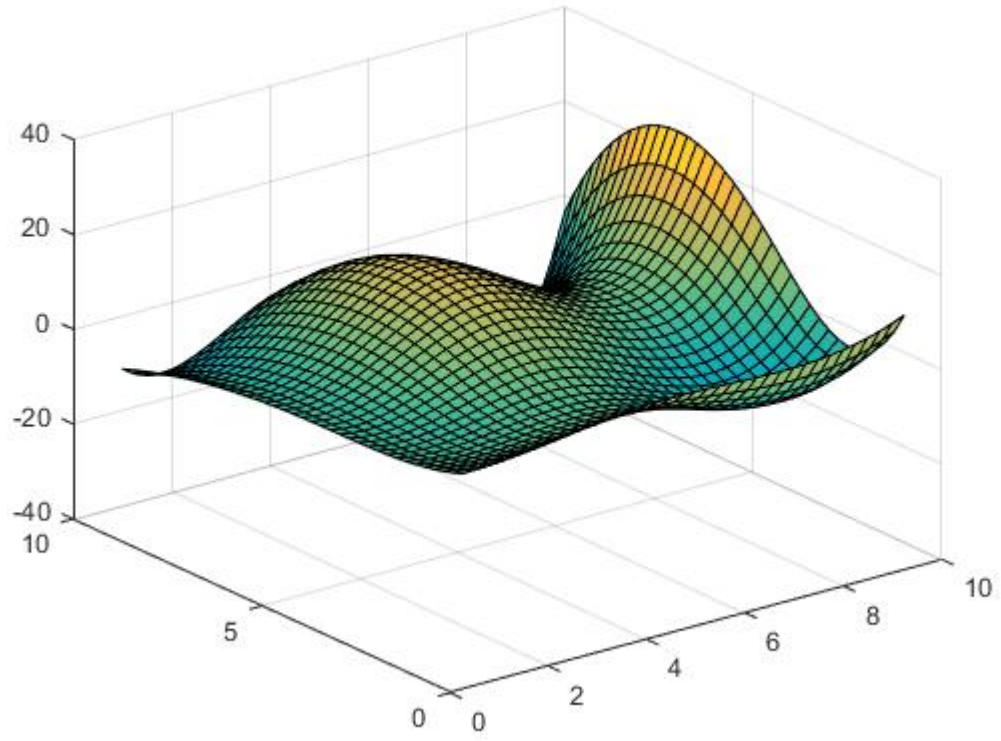


кубическая интерполяция

```
Vz = interp2(X,Y,Z,Vx,Vy,'cubic');
```

или сплайн-интерполяция:

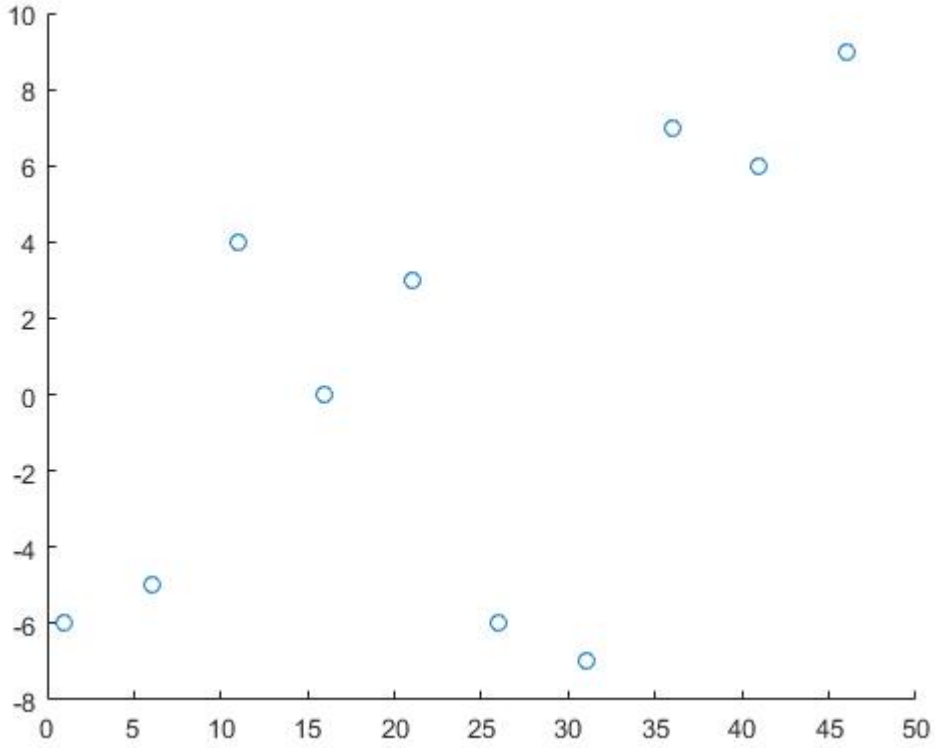
```
Vz = interp2(X,Y,Z,Vx,Vy,'spline');
```



Кусочная интерполяция 1 мерная

Мы будем использовать следующие данные:

```
x = 1:5:50;
y = randi([-10 10],1,10);
```

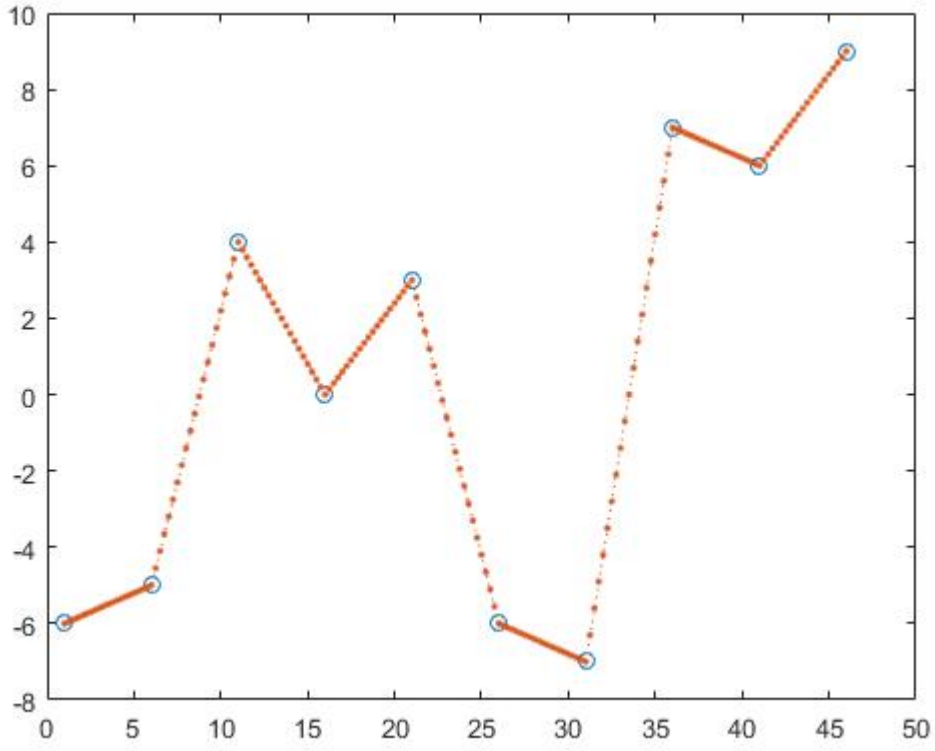


Таким образом, `x` и `y` являются координатами точек данных, а `z` - это те точки, в которых нам нужна информация.

```
z = 0:0.25:50;
```

Одним из способов найти y-значения z является кусочно-линейная интерполяция.

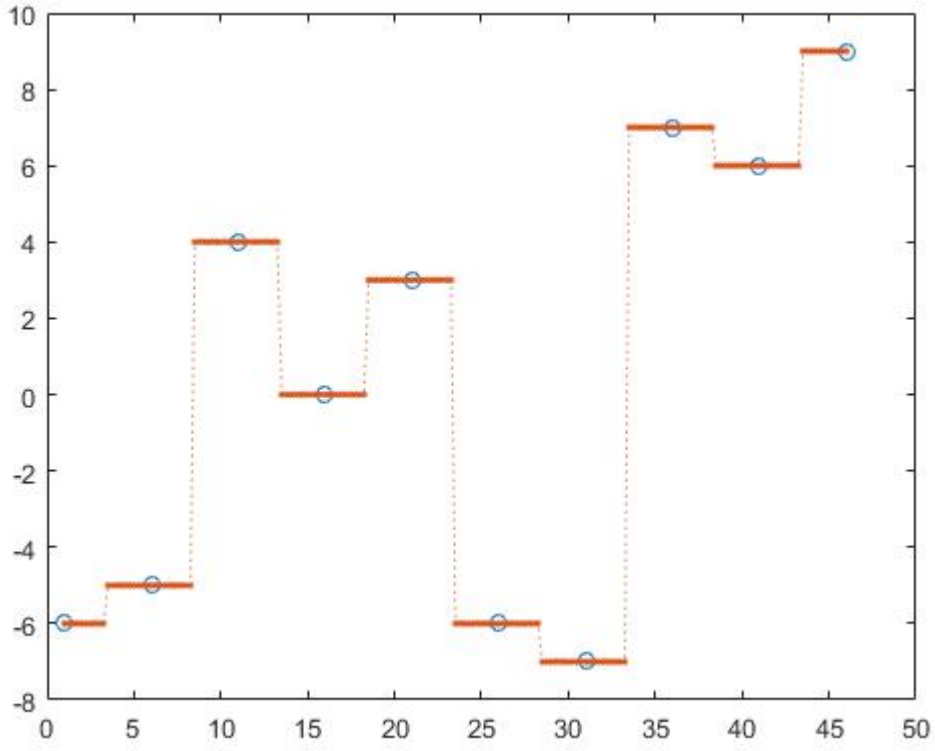
```
z_y = interp1(x,y,z,'linear');
```



Таким образом, мы вычисляем линию между двумя соседними точками и получаем `z_y` , предполагая, что точка будет элементом этих линий.

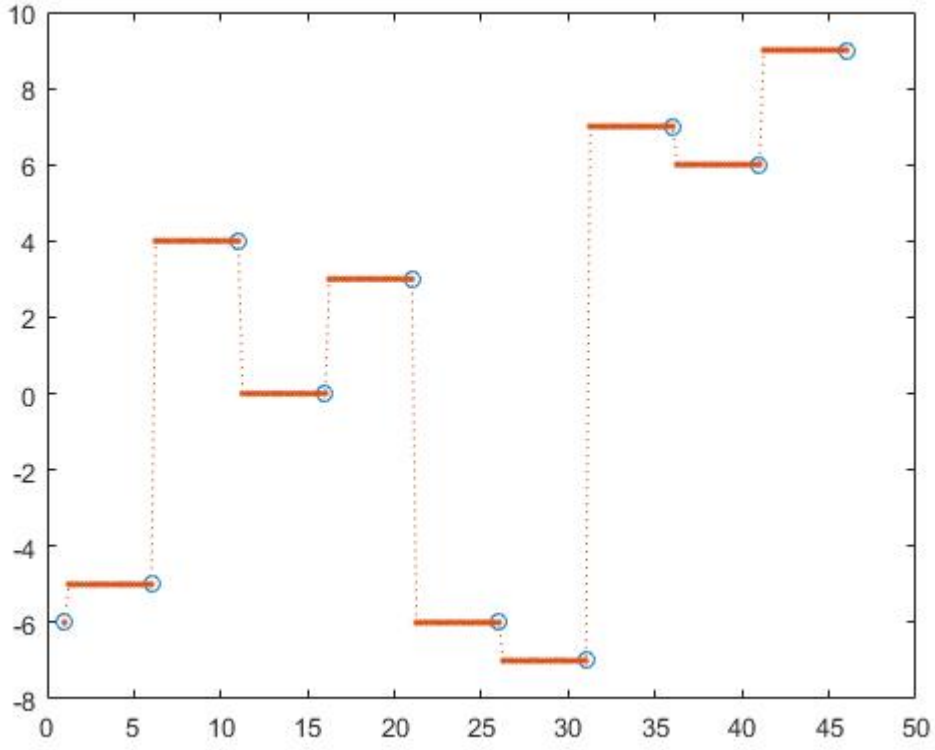
`interp1` предоставляет другие параметры, например, ближайшую интерполяцию,

```
z_y = interp1(x,y,z, 'nearest');
```



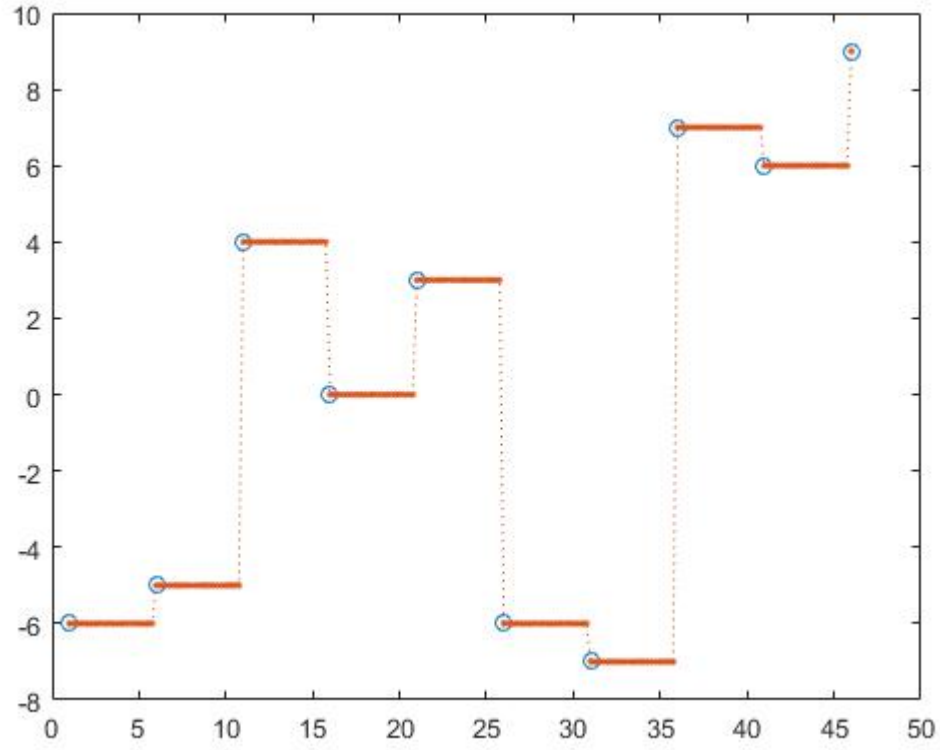
следующая интерполяция,

```
z_y = interp1(x,y,z, 'next');
```



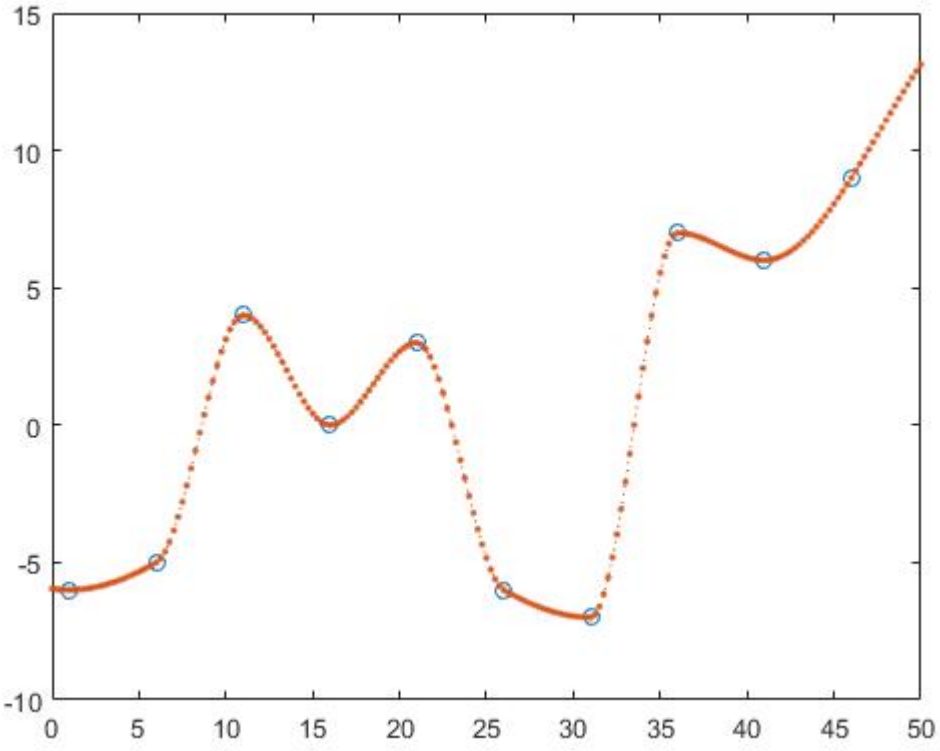
предыдущая интерполяция,

```
z_y = interp1(x,y,z, 'previous');
```

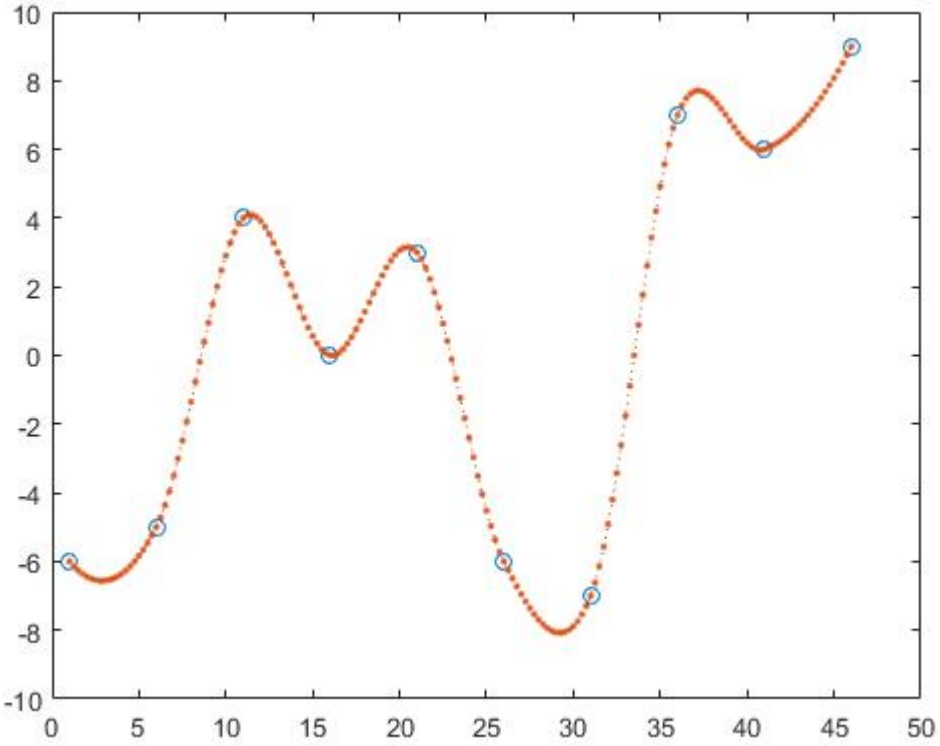


Сохранение формы кусочно-кубической интерполяцией,

```
z_y = interp1(x,y,z, 'pchip');
```

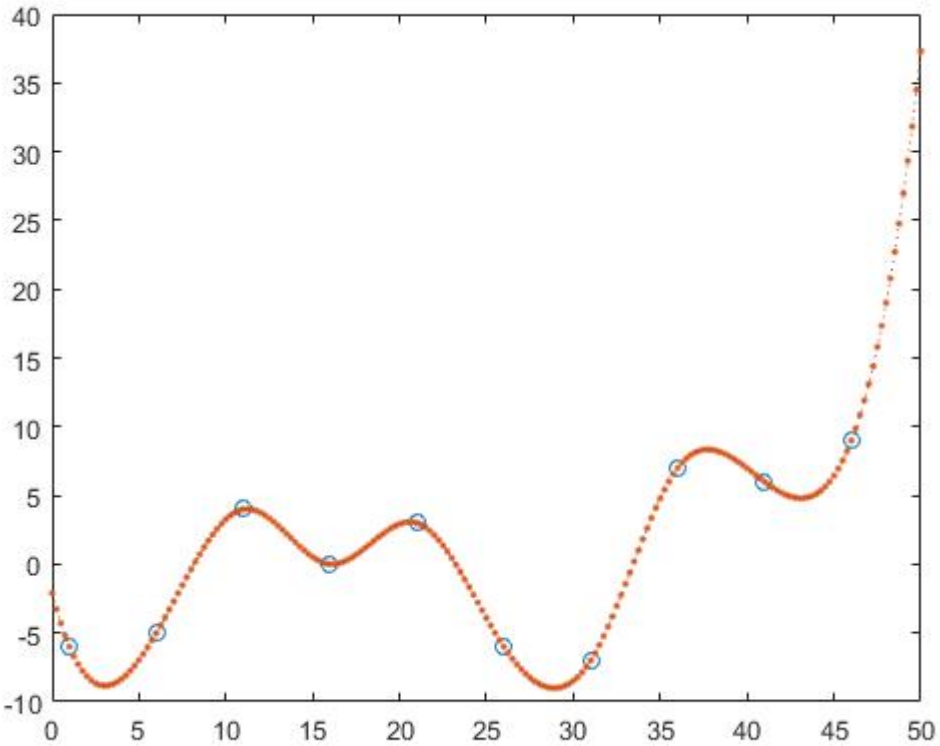


кубическая свертка, z_y = interp1 (x, y, z, 'v5cubic');



и сплайн-интерполяция

```
z_y = interp1(x,y,z, 'spline');
```



Ниже приведены приближенные, последующие и предыдущие интерполяционные кусочно-постоянные интерполяции.

Полиномиальная интерполяция

Мы инициализируем данные, которые мы хотим интерполировать:

```
x = 0:0.5:10;
y = sin(x/2);
```

Это означает, что основная функция для данных в интервале [0,10] является синусоидальной. Теперь вычисляются коэффициенты аппроксимирующих полиномов:

```
p1 = polyfit(x,y,1);
p2 = polyfit(x,y,2);
p3 = polyfit(x,y,3);
p5 = polyfit(x,y,5);
p10 = polyfit(x,y,10);
```

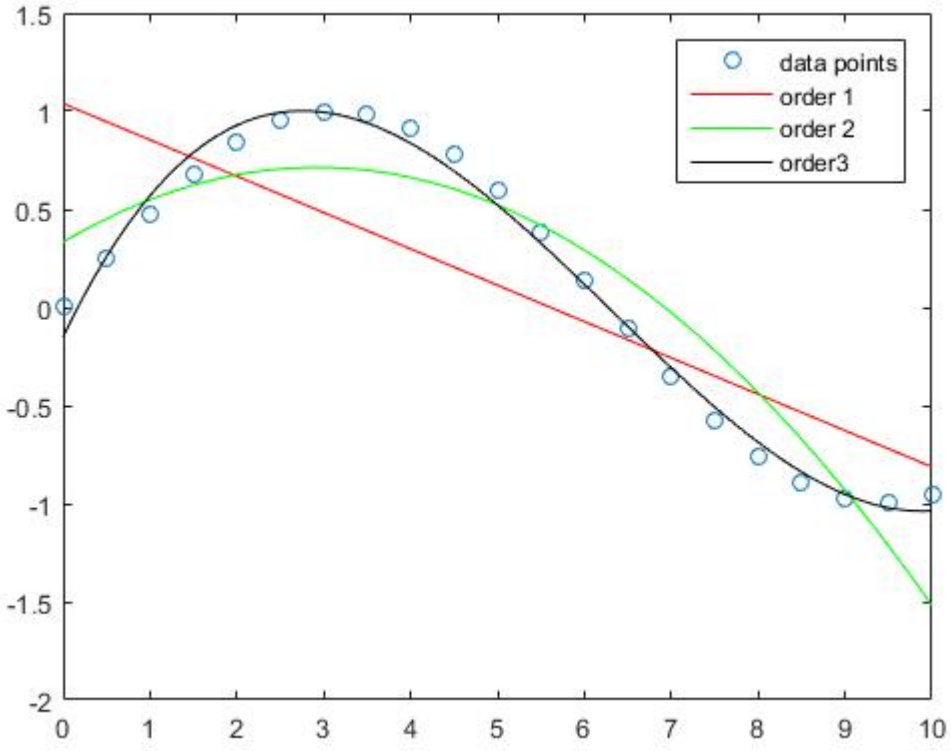
Таким образом это `x` величин x и `y` y-значение наших точек данных и третье число является порядок / степенью многочлена. Теперь мы устанавливаем сетку, которую хотим вычислить нашу интерполяционную функцию:

```
zx = 0:0.1:10;
```

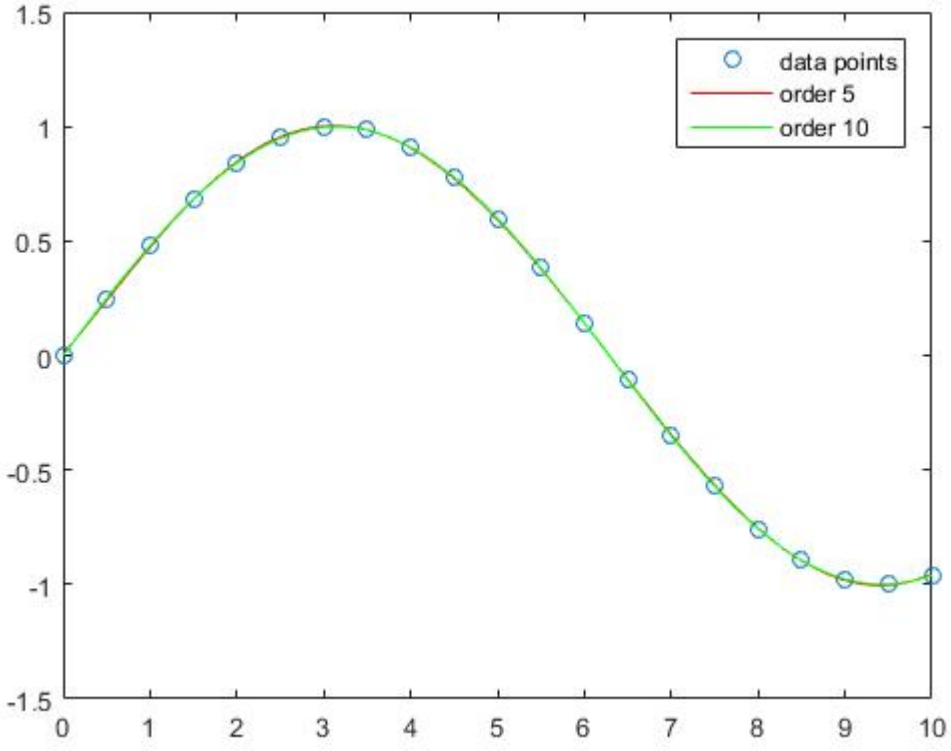
и вычислить значения y:

```
zy1 = polyval(p1,zx);
zy2 = polyval(p2,zx);
zy3 = polyval(p3,zx);
zy5 = polyval(p5,zx);
zy10 = polyval(p10,zx);
```

Видно, что погрешность аппроксимации для образца уменьшается при увеличении степени полинома.



Хотя приближение прямой в этом примере имеет более крупные ошибки, полином порядка 3 аппроксимирует синусовую функцию в этой взаимосвязи относительно хорошо.



Интерполяция с полиномами порядка 5 и порядка 10 почти не имеет погрешности apprroximation.

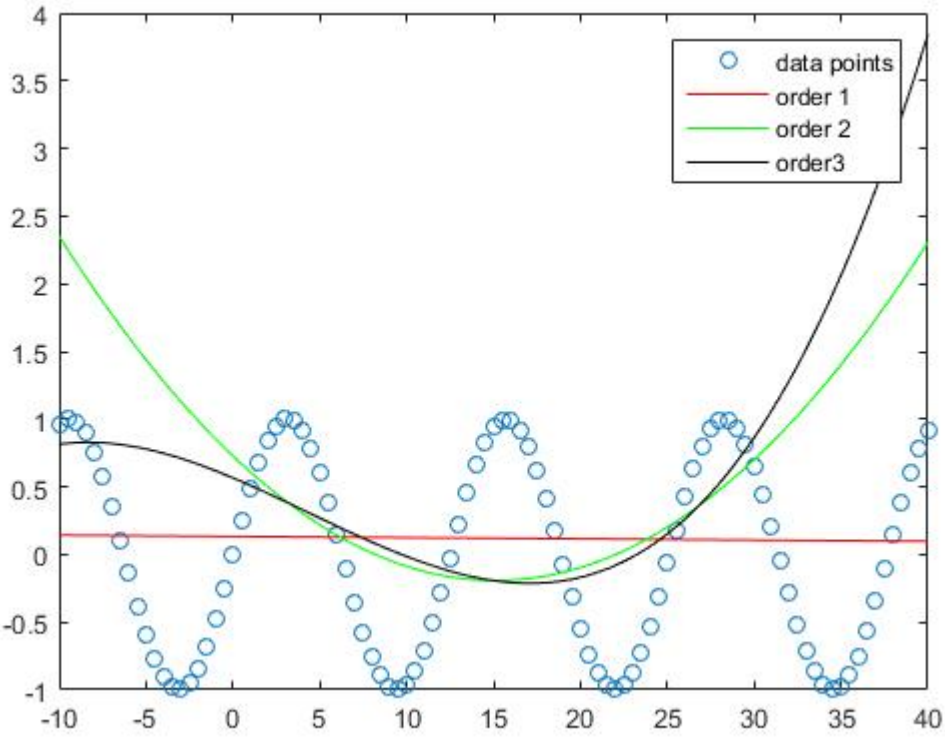
Однако, если мы рассмотрим эффективность выборки, мы увидим, что слишком высокие заказы имеют тенденцию перегружать и, следовательно, плохо работают с образцом. Мы установили

```
zx = -10:0.1:40;
p10 = polyfit(X,Y,10);
p20 = polyfit(X,Y,20);
```

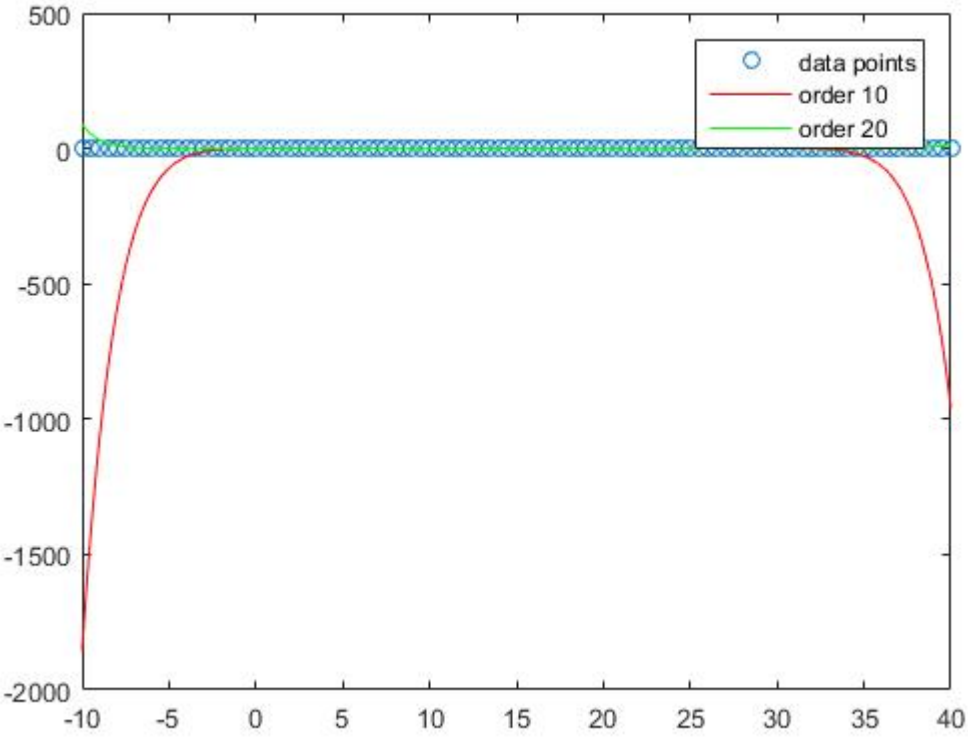
а также

```
zy10 = polyval(p10,zx);
zy20 = polyval(p20,zx);
```

Если мы посмотрим на график, мы увидим, что выборка производительности лучше всего подходит для заказа 1



и все более ухудшается.



Прочитайте Интерполяция с MATLAB онлайн: https://riptutorial.com/ru/matlab/topic/6997/интерполяция-с-matlab

глава 10: Использование последовательных портов

Вступление

Последовательные порты представляют собой общий интерфейс для связи с внешними датчиками или встроенными системами, такими как Arduinos. Современная последовательная связь часто реализуется через USB-соединения с использованием USB-последовательных адаптеров. MATLAB обеспечивает встроенные функции для последовательной связи, включая протоколы RS-232 и RS-485. Эти функции могут использоваться для аппаратных последовательных портов или «виртуальных» USB-последовательных соединений. Примеры здесь иллюстрируют последовательную связь в MATLAB.

параметры

Параметр последовательного порта	что оно делает
`BaudRate`	Устанавливает скорость. Наиболее распространенным сегодня является 57600, но 4800, 9600 и 115200 также часто видны
`InputBufferSize`	Количество байтов, хранящихся в памяти. Matlab имеет FIFO, что означает, что новые байты будут отброшены. Значение по умолчанию - 512 байт, но его можно легко установить на 20 МБ без проблем. Есть только несколько краевых случаев, когда пользователь хотел бы, чтобы это было маленьким
`BytesAvailable`	Количество байтов, ожидающих чтения
`ValuesSent`	Количество байтов, отправленных с момента открытия порта
`ValuesReceived`	Число байтов, считанных с момента открытия порта
`BytesAvailableFcn`	Укажите функцию обратного вызова, которая будет выполняться, когда указанное количество байтов доступно во входном буфере, или считывается терминатор
`BytesAvailableFcnCount`	Укажите количество байтов, которое должно быть доступно во входном буфере для генерации `bytes-available` события

Параметр последовательного порта	что оно делает
`BytesAvailableFcnMode`	Укажите, будет ли генерируемое `bytes-available` событие генерироваться после того, как указанное количество байтов будет доступно во входном буфере или после чтения терминатора

Examples

Создание последовательного порта на Mac / Linux / Windows

```
% Define serial port with a baud rate of 115200
rate = 115200;
if ispc
    s = serial('COM1', 'BaudRate',rate);
elseif ismac
    % Note that on OSX the serial device is uniquely enumerated. You will
    % have to look at /dev/tty.* to discover the exact signature of your
    % serial device
    s = serial('/dev/tty.usbserial-A104VFT7', 'BaudRate',rate);
elseif isunix
    s = serial('/dev/ttyusb0', 'BaudRate',rate);
end

% Set the input buffer size to 1,000,000 bytes (default: 512 bytes).
s.InputBufferSize = 1000000;

% Open serial port
fopen(s);
```

Чтение из последовательного порта

Предполагая, что вы создали объект последовательного порта `s` как в этом примере, тогда

```
% Read one byte
data = fread(s, 1);

% Read all the bytes, version 1
data = fread(s);

% Read all the bytes, version 2
data = fread(s, s.BytesAvailable);

% Close the serial port
fclose(s);
```

Закрытие последовательного порта, даже если он потерян, удален или перезаписан

Предполагая, что вы создали объект последовательного порта `s` как в этом примере, затем закройте его

```
fclose(s)
```

Однако иногда вы можете случайно потерять порт (например, очистить, перезаписать, изменить область действия и т. Д.), А `fclose(s)` больше не будет работать. Решение легко

```
fclose(instrfindall)
```

Дополнительная информация в `instrfindall()` .

Запись на последовательный порт

Предполагая, что вы создали объект последовательного порта `s` как в этом примере, тогда

```
% Write one byte
fwrite(s, 255);

% Write one 16-bit signed integer
fwrite(s, 32767, 'int16');

% Write an array of unsigned 8-bit integers
fwrite(s,[48 49 50],'uchar');

% Close the serial port
fclose(s);
```

Выбор режима общения

Matlab поддерживает *синхронную* и *асинхронную* связь с последовательным портом. Важно выбрать правильный режим связи. Выбор будет зависеть от:

- как инструмент, с которым вы общаетесь, ведет себя.
- какие другие функции ваша основная программа (или графический интерфейс) придется делать помимо управления последовательным портом.

Я опишу 3 различных случая, чтобы проиллюстрировать, от самых простых до самых требовательных. Для 3-х примеров инструмент, к которому я подключаюсь, представляет собой печатную плату с инклинометром, который может работать в трех режимах, которые я буду описывать ниже.

Режим 1: синхронный (ведущий / ведомый)

Этот режим является самым простым. Это соответствует случаю, когда ПК является *Мастером,* а инструмент является *подчиненным* . Инструмент ничего не отправляет на

последовательный порт, он **отвечает** только после ответа на задание / команду мастера (ПК, ваша программа). Например:

- ПК отправляет команду: «Дайте мне измерение сейчас»,
- Прибор получает команду, выполняет измерение, а затем возвращает значение измерения в последовательную линию: «Значение инклинометра - XXX».

ИЛИ ЖЕ

- ПК отправляет команду: «Переход из режима X в режим Y»
- Инструмент получает команду, выполняет ее, а затем отправляет подтверждающее сообщение в последовательную строку: « *Выполненная команда* » (или « *Команда НЕ выполнена* »). Обычно это называется ответом ACK / NACK (для «Acknowledge (d)» / «NOT Acknowledged»).

Резюме: в этом режиме инструмент (*ведомый*) только отправляет данные в последовательную линию **сразу же после** того, как компьютер спросил (*мастер*)

Режим 2: асинхронный

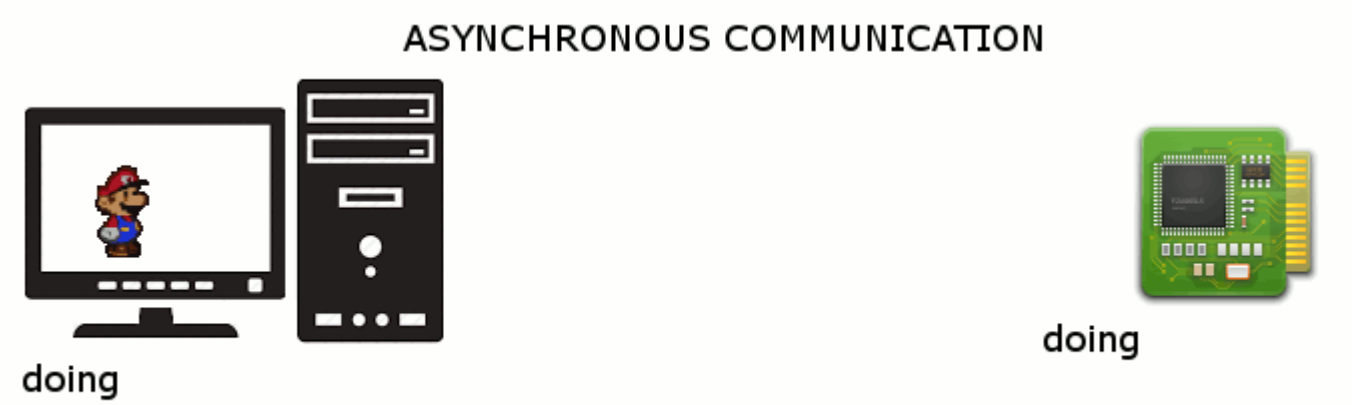
Теперь предположим, что я начал свой инструмент, но это больше, чем просто глупый сенсор. Он постоянно контролирует собственный склон и до тех пор, пока он вертикальный (в пределах допуска, скажем, +/- 15 градусов), он остается тихим. Если устройство наклонено более чем на 15 градусов и приближается к горизонтали, оно отправляет аварийное сообщение на последовательную линию, сразу же после чего считывается наклон. Пока наклон выше порога, он продолжает посылать показания наклона каждые 5 секунд.

Если ваша основная программа (или графический интерфейс) постоянно «ждет» сообщения, поступающего на последовательную линию, это может сделать это хорошо ... но пока это не может сделать ничего другого. Если основной программой является графический интерфейс, очень сложно сделать графический интерфейс

«замороженным», потому что он не будет принимать какие-либо данные от пользователя. По существу, он стал *рабом,* а инструмент - *Мастером* . Если у вас нет причудливого способа управления вашим графическим интерфейсом от инструмента, этого можно избежать. К счастью, *асинхронный* режим связи позволит вам:

- определить отдельную функцию, которая сообщает вашей программе, что делать, когда сообщение получено
- удерживайте эту функцию в углу, она будет вызываться и выполняться только **при поступлении сообщения** на последовательную линию. В остальное время GUI может выполнять любой другой код, который должен выполнить.

Описание: В этом режиме инструмент может отправлять сообщение на последовательную линию в любое время (но не обязательно *все* время). Компьютер не *ждать* постоянно для сообщения для обработки. Разрешено запускать любой другой код. Только когда приходит сообщение, оно активирует функцию, которая будет читать и обрабатывать это сообщение.



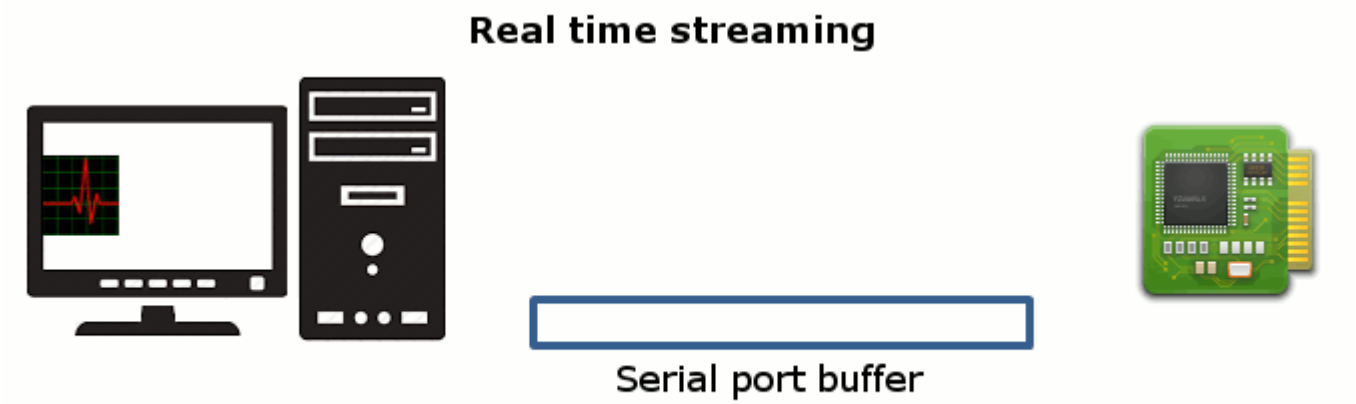
Режим 3: Потоковое (в *реальном времени*)

Теперь давайте развяжем всю мощь моего инструмента. Я помещаю его в режим, когда он будет постоянно отправлять измерения в последовательную линию. Моя программа хочет получить эти пакеты и отобразить их на кривой или цифровом дисплее. Если он отправляет только значение каждые 5 секунд, как указано выше, не проблема, сохраните вышеуказанный режим. Но мой инструмент с полным ударом отправляет точку данных в последовательную линию на частоте 1000 Гц, то есть посылает новое значение каждые миллисекунды. Если я останусь в *асинхронном режиме,* описанном выше, существует высокий риск (на самом деле гарантированная уверенность), что специальная функция, которую мы определили для обработки каждого нового пакета, займет более 1 мс (если вы хотите построить или отобразить значение, графические функции довольно медленные, даже не учитывая фильтрацию или FFT-сигнал). Это означает, что функция начнет выполнение, но прежде чем она закончится, появится новый пакет и снова активирует

функцию. Вторая функция помещается в очередь для выполнения и запускается только тогда, когда первая выполняется ... но к этому времени появилось несколько новых пакетов, и каждая из них помещала функцию в очередь. Вы можете быстро предвидеть результат: к тому моменту, когда я рисую 5 баллов, у меня уже сотни ожиданий, которые тоже будут заложены ... gui замедляется, в конце концов зависает, стек растет, буферы заполняются, пока что-то не дает. В конце концов вы остаетесь полностью замороженной программой или просто разбитой.

Чтобы преодолеть это, мы еще больше отключим связь синхронизации между ПК и инструментом. Мы позволяем прибору отправлять данные в своем собственном темпе, без немедленного запуска функции при каждом приходе пакета. Буфер последовательного порта просто накапливает полученные пакеты. ПК будет собирать данные только в буфере с темпом, которым он может управлять (регулярный интервал, настроенный на стороне ПК), что-то делать с ним (пока буфер заполняется инструментом), а затем собирайте новую партию данные из буфера ... и так далее.

Описание: В этом режиме прибор непрерывно посылает данные, которые собираются буфером последовательного порта. На регулярной основе ПК собирает данные из буфера и что-то делает с ним. Между ПК и прибором нет жесткой синхронизации. Оба выполняют свои задачи самостоятельно.



Автоматическая обработка данных, полученных от последовательного порта

Некоторые устройства, подключенные через последовательный порт, отправляют данные в вашу программу с постоянной скоростью (потоковые данные) или отправляют данные с непредсказуемыми интервалами. Вы можете настроить последовательный порт для выполнения функции автоматически для обработки данных всякий раз, когда он прибывает. Это называется «функцией обратного вызова» для объекта последовательного порта.

Существует два свойства последовательного порта, которые должны быть установлены

для использования этой функции: имя функции, которую вы хотите для обратного вызова (`BytesAvailableFcn`), и условие, которое должно инициировать выполнение функции обратного вызова (`BytesAvailableFcnMode`).

Существует два способа запуска функции обратного вызова:

1. Когда определенное количество байтов было получено на последовательном порту (обычно используется для двоичных данных)
2. Когда определенный символ принимается на последовательном порту (обычно используется для текстовых или ASCII-данных)

Функции обратного вызова имеют два обязательных входных аргумента, называемых `obj` и `event` . `obj` - последовательный порт. Например, если вы хотите распечатать данные, полученные от последовательного порта, определите функцию для печати данных, называемых `newdata` :

```
function newdata(obj,event)
    [d,c] = fread(obj);  % get the data from the serial port
    % Note: for ASCII data, use fscanf(obj) to return characters instead of binary values
    fprintf(1,'Received %d bytes\n',c);
    disp(d)
end
```

Например, чтобы выполнить функцию `newdata` всякий раз, когда `newdata` 64 байта данных, настройте последовательный порт следующим образом:

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcnCount = 64;
s.BytesAvailableFcn = @newdata;
```

С текстовыми или ASCII-данными данные обычно делятся на строки с «символом терминатора», как текст на странице. Чтобы выполнить функцию `newdata` всякий раз, когда получен символ возврата каретки, настройте последовательный порт следующим образом:

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'terminator';
s.Terminator = 'CR';  % the carriage return, ASCII code 13
s.BytesAvailableFcn = @newdata;
```

Прочитайте Использование последовательных портов онлайн:
https://riptutorial.com/ru/matlab/topic/1176/использование-последовательных-портов

глава 11: Использование функции `accumarray ()`

Вступление

`accumarray` позволяет агрегировать элементы массива различными способами, потенциально применяя некоторую функцию к элементам процесса. `accumarray` можно рассматривать как легкий редуктор (см. также: Введение в MapReduce).

Этот `accumarray` будет содержать общие сценарии, в которых особенно полезен `accumarray` .

Синтаксис

- accumarray (subscriptArray, valuesArray)
- accumarray (subscriptArray, valuesArray, sizeOfOutput)
- accumarray (subscriptArray, valuesArray, sizeOfOutput, funcHandle)
- accumarray (subscriptArray, valuesArray, sizeOfOutput, funcHandle, fillVal)
- accumarray (subscriptArray, valuesArray, sizeOfOutput, funcHandle, fillVal, isSparse)

параметры

параметр	подробности
`subscriptArray`	Матрица индексов, заданная как вектор индексов, матрица индексов или массив ячеек индексных векторов.
`valuesArray`	Данные, заданные как вектор или скаляр.
`sizeOfOutput`	Размер выходного массива, заданный как вектор положительных целых чисел.
`funcHandle`	Функция, применяемая к каждому набору элементов во время агрегации, указанная как дескриптор функции или `[]` .
`fillVal`	Заполнить значение, когда `subs` не ссылается на каждый элемент на выходе.
`isSparse`	Должен ли выход быть разреженным массивом?

замечания

- Представлен в MATLAB v7.0.

Ссылки :

1. « `accumarray` », *Лорен Шуре* , 20 февраля 2008 года .
2. `accumarray` в официальной документации MATLAB.

Examples

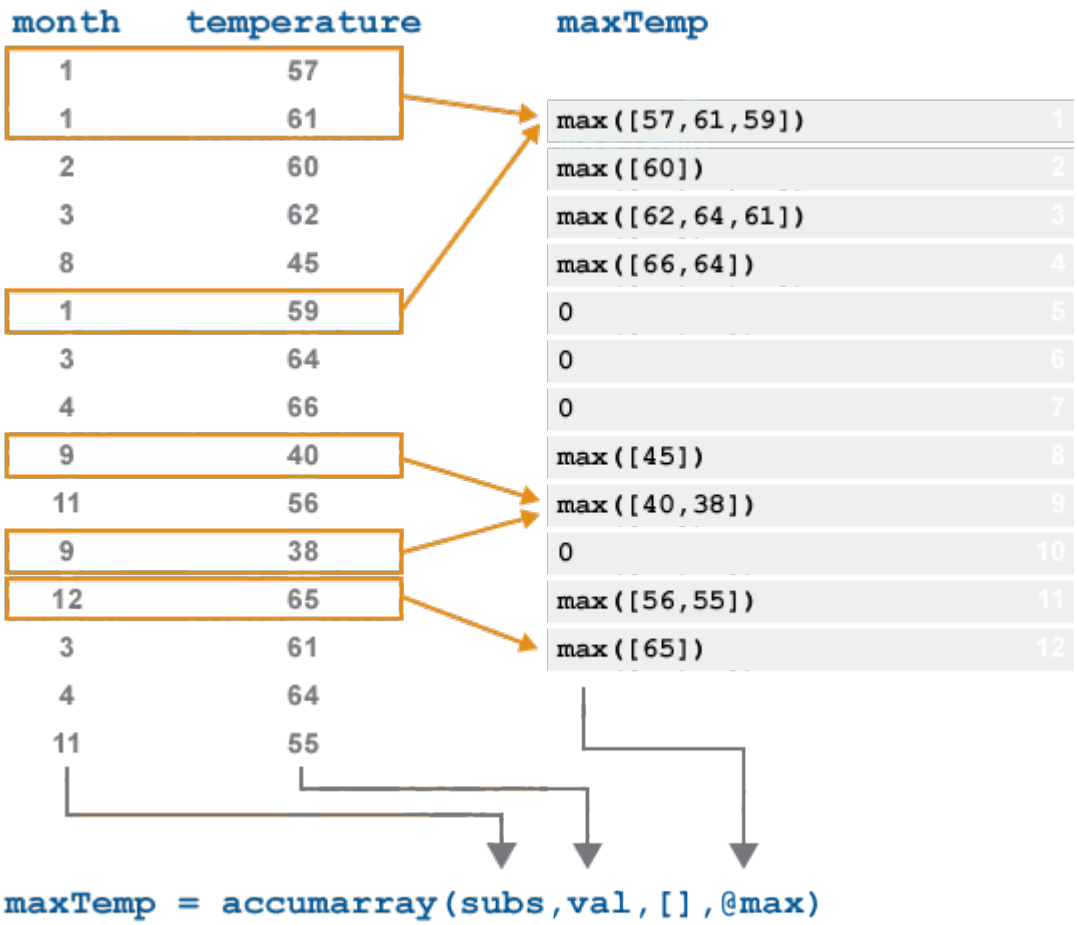
Поиск максимального значения среди элементов, сгруппированных другим вектором

Это официальный пример MATLAB

Рассмотрим следующий код:

```
month = [1;1;2;3;8;1;3;4;9;11;9;12;3;4;11];
temperature = [57;61;60;62;45;59;64;66;40;56;38;65;61;64;55];
maxTemp = accumarray(month,temperature,[],@max);
```

На приведенном ниже `accumarray` процесс вычисления, выполненный с помощью `accumarray` в этом случае:



В этом примере сначала собираются все значения, имеющие один и тот же `month` , а затем функция, указанная 4- ᴍ входом для `accumarray` (в данном случае, `@max`), применяется к каждому из таких наборов.

Примените фильтр к изображениям и установите каждый пиксель как среднее для результата каждого патча

Многие современные алгоритмы обработки изображений используют патчи - их основной элемент для работы.
Например, можно было бы разложить патчи (см. Алгоритм BM3D).

Однако при создании формы изображения обработанных патчей у нас есть много результатов для одного и того же пикселя.
Один из способов борьбы с ним - взять среднее (эмпирическое среднее) всех значений одного и того же пикселя.

Следующий код показывает, как разбить изображение на патчи, и они восстанавливают изображение из патчей, используя среднее значение, используя `[accumarray()][1]` :

```
numRows = 5;
numCols = 5;

numRowsPatch = 3;
numColsPatch = 3;

% The Image
mI = rand([numRows, numCols]);

% Decomposing into Patches - Each pixel is part of many patches (Neglecting
% boundariwes, each pixel is part of (numRowsPatch * numColsPatch) patches).
mY = ImageToColumnsSliding(mI, [numRowsPatch, numColsPatch]);

% Here one would apply some operation which work on patches

% Creating image of the index of each pixel
mPxIdx = reshape(1:(numRows * numCols), [numRows, numCols]);

% Creating patches of the same indices
mSubsAccu = ImageToColumnsSliding(mPxIdx, [numRowsPatch, numColsPatch]);

% Reconstruct the image - Option A
mO = accumarray(mSubsAccu(:), mY(:)) ./ accumarray(mSubsAccu(:), 1);

% Reconstruct the image - Option B
mO = accumarray(mSubsAccu, mY(:), [(numRows * numCols), 1], @(x) mean(x));

% Rehsape the Vector into the Image
mO = reshape(mO, [numRows, numCols]);
```

Прочитайте Использование функции `accumarray ()` онлайн:
https://riptutorial.com/ru/matlab/topic/9321/использование-функции--accumarray----

глава 12: Использование функций с логическим выходом

Examples

Все и все с пустым массивом

Особая осторожность должна быть предпринята, когда существует вероятность того, что массив станет пустым массивом, когда дело доходит до логических операторов. Часто ожидается, что если `all(A)` истинны, то `any(A)` должно быть истинным, а если `any(A)` ложно, `all(A)` также должны быть ложными. Это не так в MATLAB с пустым массивом.

```
>> any([])
ans =
     0
>> all([])
ans =
     1
```

Поэтому, если вы, например, сравниваете все элементы массива с определенным порогом, вам нужно знать о том, где массив пуст:

```
>> A=1:10;
>> all(A>5)
ans =
     0
>> A=1:0;
>> all(A>5)
ans =
     1
```

Использовать встроенную функцию `isempty` для проверки пустых массивов:

```
a = [];
isempty(a)
ans =
1
```

Прочитайте Использование функций с логическим выходом онлайн:
https://riptutorial.com/ru/matlab/topic/5608/использование-функций-с-логическим-выходом

глава 13: Лучшие практики MATLAB

замечания

В этом разделе представлены лучшие практики, которые сообщество узнало с течением времени.

Examples

Стройте короткие строки

Используйте символ продолжения (многоточие) `...` для продолжения длинного оператора.

Пример:

```
MyFunc( parameter1,parameter2,parameter3,parameter4, parameter5, parameter6,parameter7,
parameter8, parameter9)
```

могут быть заменены на:

```
MyFunc( parameter1, ...
        parameter2, ...
        parameter3, ...
        parameter4, ...
        parameter5, ...
        parameter6, ...
        parameter7, ...
        parameter8, ...
        parameter9)
```

Код отступа правильно

Правильный отступ дает не только эстетический вид, но и повышает читаемость кода.

Например, рассмотрим следующий код:

```
%no need to understand the code, just give it a look
n = 2;
bf = false;
while n>1
for ii = 1:n
for jj = 1:n
if ii+jj>30
bf = true;
break
end
end
if bf
break
```

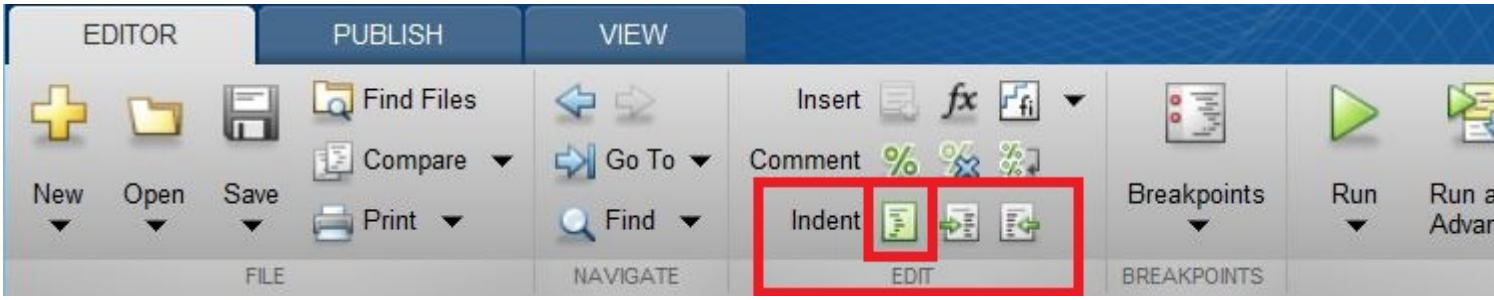
```
end
end
if bf
break
end
n = n + 1;
end
```

Как вы можете видеть, вам нужно внимательно посмотреть, какой цикл и `if` утверждения заканчиваются где.
С умным отступом вы получите этот вид:

```
n = 2;
bf = false;
while n>1
    for ii = 1:n
        for jj = 1:n
            if ii+jj>30
                bf = true;
                break
            end
        end
        if bf
            break
        end
    end
    if bf
        break
    end
    n = n + 1;
end
```

Это явно указывает начало и конец инструкции loop / `if` .

Вы можете делать интеллектуальные отступы:
• выбор всего кода (`Ctrl` + `A`)
• а затем нажмите `Ctrl` + `I` или нажмите  из панели редактирования.



Использовать assert

Matlab позволяет совершать очень простые ошибки, что может привести к тому, что ошибка будет поднята намного позже в ходе выполнения, что затруднит отладку. Если вы **примете** что-то о своих переменных, **подтвердите** его.

```
function out1 = get_cell_value_at_index(scalar1,cell2)
assert(isscalar(scalar1),'1st input must be a scalar')
assert(iscell(cell2),'2nd input must be a cell array')

assert(numel(cell2) >= scalar1),'2nd input must have more elements than the value of the 1st
input')
assert(~isempty(cell2{scalar1}),'2nd input at location is empty')

out1 = cell2{scalar1};
```

Избегайте циклов

Большую часть времени, петли вычислительно дорого с Matlab. При использовании векторизации ваш код будет быстрее на порядок. Он также часто делает ваш код более модульным, легко изменяемым и легче отлаживать. Основной недостаток заключается в том, что вам нужно потратить время на планирование структур данных, а ошибки измерения легче найти.

Примеры

Не пишите

```
for t=0:0.1:2*pi
    R(end+1)=cos(t);
end
```

но

```
t=0:0.1:2*pi;
R=cos(t)
```

Не пишите

```
for i=1:n
    for j=1:m
        c(i,j)=a(i)+2*b(j);
    end
end
```

Но что-то похожее

```
c=repmat(a.',1,m)+2*repmat(b,n,1)
```

Для получения дополнительной информации см. Векторизация

Создание уникального имени для временного файла

При кодировании скрипта или функции может потребоваться один или несколько

временных файлов, чтобы, например, хранить некоторые данные.

Чтобы избежать перезаписи существующего файла или тени функции MATLAB, функция tempname может использоваться для создания **уникального имени** временного файла в системной временной папке.

```
my_temp_file=tempname
```

Имя файла генерируется без расширения; его можно добавить, объединяя желаемое расширение с именем, созданным `tempname`

```
my_temp_file_with_ext=[tempname '.txt']
```

Локатор временной папки системы можно восстановить, выполнив функцию tempdir .

Если во время выполнения функции / скрипта временный файл больше не нужен, его можно удалить, используя функцию delete

Поскольку `delete` не требует подтверждения, это может быть полезно установить `on` опцию , чтобы переместить файл , который будет удален в `recycle` папке.

Это можно сделать, используя функцию recycle следующим образом:

```
recycle('on')
```

В следующем примере предлагается возможное использование функций `tempname` , `delete` и `recycle` .

```
%
% Create some example data
%
theta=0:.1:2*pi;
x=cos(theta);
y=sin(theta);
%
% Generate the temporary filename
%
my_temp_file=[tempname '.mat'];
%
% Split the filename (path, name, extension) and display them in a message box
[tmp_file_path,tmp_file_name, tmp_file_ext]=fileparts(my_temp_file)
uiwait(msgbox(sprintf('Path= %s\nName= %s\nExt= %s', ...
              tmp_file_path,tmp_file_name,tmp_file_ext),'TEMPORARY FILE'))
%
% Save the varaibles in a temporary file
%
save(my_temp_file,'x','y','theta')
%
% Load the varaibles from the temporary file
%
load(my_temp_file)
%
```

```
% Set the reclycle option on
%
recycle('on')
%
% Delete the temporary file
%
delete(my_temp_file)
```

Предостережение

Временное имя файла создается с помощью метода `java.util.UUID.randomUUID` (randomUUID).

Если MATLAB запускается без JVM, временное имя файла генерируется с помощью `matlab.internal.timing.timing` на основе счетчика и времени процессора. В этом случае временное имя файла не гарантируется быть уникальным.

Использовать validateattributes

Функция validateattributes может использоваться для проверки массива по набору спецификаций

Поэтому он может использоваться для проверки ввода, предоставляемого функции.

В следующем примере функция `test_validateattributes` требует три ввода

```
function test_validateattributes(input_1,input_2,input_3)
```

Спецификация входа:

- array_1:
    - класс: двойной
    - размер: [3,2]
    - значения: элементы должны быть не NaN
- char_array:
    - class: char
    - value: строка не должна быть пустой
- array_3
    - класс: двойной
    - размер: [5 1]
    - значения: элементы должны быть реальными

Чтобы проверить три входа, функцию `validateattributes` можно вызвать со следующим синтаксисом:

```
validateattributes(A,classes,attributes,funcName,varName,argIndex)
```

где:

- `A` - это массив, подлежащий обложению
- `classes` : `type` массива (например, `single` , `double` , `logical`)
- `attributes` : это `attributes` которые должен иметь входной массив (например, `[3,2]`, `size` чтобы указать размер массива, `nonnan` чтобы указать, что массив не должен иметь значения NaN)
- `funcName` : имя функции, в которой происходит проверка. Этот аргумент используется при генерации сообщения об ошибке (если есть)
- `varName` : имя массива при проверке. Этот аргумент используется при генерации сообщения об ошибке (если есть)
- `argIndex` : позиция массива inpurt в списке ввода. Этот аргумент используется при генерации сообщения об ошибке (если есть)

Если один или несколько входных данных не соответствуют спецификации, генерируется сообщение об ошибке.

В случае более одного недопустимого ввода проверка прекращается, когда обнаружено первое несоответствие.

Это `function test_validateattributes` в которой реализована проверка ввода.

Так как функция требует трех входных сигналов, первая проверка количества вводимых данных выполняется с использованием nargin .

```
function test_validateattributes(array_1,char_array_1,array_3)
%
% Check for the number of expected input: if the number of input is less
% than the require, the function exits with an error message
%
if(nargin ~= 3)
   error('Error: TEST_VALIDATEATTRIBUTES requires 3 input, found %d',nargin)
end
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the first input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #1 name (only used in the generation of the error message)
%
input_1_name='array_1';
%
% INPUT #1 position (only used in the generation of the error message)
%
input_1_position=1;
%
% Expected CLASS of the first input MUST BE "double"
%
input_1_class={'double'};
%
```

```
% Expected ATTRIBUTES of the first input
%   SIZE: MUST BE [3,2]
%
input_1_size_attribute='size';
input_1_size=[3,2];
%
%   VALUE CHECK: the element MUST BE NOT NaN
%
input_1_value_type='nonnan';
%
% Build the INPUT 1 attributes
%
input_1_attributes={input_1_size_attribute,input_1_size, ...
                    input_1_value_type};
%
% CHECK THE VALIDITY OF THE FIRST INPUT
%
validateattributes(array_1, ...
                   input_1_class,input_1_attributes,'', ...
                   input_1_name,input_1_position);

%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the second input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #1 name (only used in the generation of the error message)
%
input_2_name='char_array_1';
%
% INPUT #2 position (only used in the generation of the error message)
%
input_2_position=2;
%
% Expected CLASS of the first input MUST BE "string"
%
input_2_class={'char'};
%
%   VALUE CHECK: the element must be not NaN
%
input_2_size_attribute='nonempty';
%
% Build the INPUT 2 attributes
%
input_2_attributes={input_2_size_attribute};
%
% CHECK THE VALIDITY OF THE SECOND INPUT
%
validateattributes(char_array_1, ...
                   input_2_class,input_2_attributes,'', ...
                   input_2_name,input_2_position);
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the third input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #3 name (only used in the generation of the error message)
%
input_3_name='array_3';
%
% INPUT #3 position (only used in the generation of the error message)
```

```
%
input_3_position=3;
%
% Expected CLASS of the first input MUST BE "double"
%
input_3_class={'double'};
%
% Expected ATTRIBUTES of the first input
%   SIZE: must be [5]
input_3_size_attribute='size';
input_3_size=[5 1];
%   VALUE CHECK: the elements must be real
input_3_value_type='real';
%
% Build the INPUT 3 attributes
%
input_3_attributes={input_3_size_attribute,input_3_size, ...
                    input_3_value_type};
%
% CHECK THE VALIDITY OF THE FIRST INPUT
%
validateattributes(array_3, ...
                   input_3_class,input_3_attributes,'', ...
                   input_3_name,input_3_position);

disp('All the three input are OK')
```

Следующий сценарий можно использовать для проверки реализации процедуры проверки.

Он генерирует три необходимых ввода и, случайно, делает их недействительными.

```
%
% Generate the first input
%
n_rows=randi([2 3],1);
n_cols=2;
input_1=randi([20 30],n_rows,n_cols);
%
% Generate the second input
%
if(rand > 0.5)
   input_2='This is a string';
else
   input_2='';
end
%
% Generate the third input
%
input_3=acos(rand(5,1)*1.2);
%
% Call the test_validateattributes function with the above generated input
%
input_1
input_2
input_3
%
test_validateattributes(input_1,input_2,input_3)
```

Это несколько примеров неправильного ввода, обнаруженного функцией `validateattributes`

:

Неверный ввод

```
input_1 =

    23    22
    26    28

input_2 =

     ''

input_3 =

   0.0000 + 0.4455i
   1.2420 + 0.0000i
   0.4063 + 0.0000i
   1.3424 + 0.0000i
   1.2186 + 0.0000i

Error using test_validateattributes (line 44)
Expected input number 1, array_1, to be of size 3x2 when it is actually
size 2x2.
```

Неверный ввод

```
input_1 =

    22    24
    21    25
    26    27

input_2 =

This is a string

input_3 =

   1.1371 + 0.0000i
   0.6528 + 0.0000i
   1.0479 + 0.0000i
   0.0000 + 0.1435i
   0.0316 + 0.0000i

Error using test_validateattributes (line 109)
Expected input number 3, array_3, to be real.
```

Действительный ввод

```
input_1 =

    20    25
    25    28
    24    23

input_2 =
```

```
This is a string

input_3 =

    0.9696
    1.5279
    1.3581
    0.5234
    0.9665

All the three input are OK
```

Оператор блокировки комментариев

Хорошая практика - добавлять комментарии, описывающие код. Это полезно для других и даже для кодера при возврате позже. Одну строку можно прокомментировать с помощью символа `%` или с помощью сочетания `Ctrl+R` Чтобы раскомментировать ранее прокомментированную строку, удалите символ `%` или используйте короткий ключ `Crtl+T`

Комментируя блок кода, можно сделать, добавив символ `%` в начале каждой строки, более новые версии MATLAB (после 2015a) позволяют использовать **блок-комментарий оператора** `%{ code %}` . Этот оператор повышает читаемость кода. Он может использоваться как для комментирования кода, так и для справочной документации по функциям. Блок можно **складывать** и **разворачивать** для повышения удобочитаемости кода.

```
function y = myFunction(x)
%{
myFunction  Binary Singleton Expansion Function
y = myFunction(x) applies the element-by-element binary operation
specified by the function handle FUNC to arrays A and B, with impli
expansion enabled.
%}

%%  Compute z(x, y) = x.*sin(y) on a grid:
% x = 1:10;
y = x.';

%{
z = zeros(numel(x),numel(y));
for ii=1:numel(x)
    for jj=1:numel(y)
        z(ii,jj) = x(ii)*sin(y(jj));
    end
end
%}

z = bsxfun(@(x, y) x.*sin(y), x, y);
y = y + z;

end
```

Как видно, операторы `%{` и `%}` должны отображаться только в строках. Не включайте другой текст в эти строки.

```
function y = myFunction(x)
%{
myFunction  Binary Singleton Expansion Function
y = myFunction(x) applies the element-by-element binary operation
specified by the function handle FUNC to arrays A and B, with implicit
expansion enabled.
%}

%%  Compute z(x, y) = x.*sin(y) on a grid:
% x = 1:10;
y = x.';

%{
z = zeros(numel(x),numel(y));
for ii=1:numel(x)
    for jj=1:numel(y)
        z(ii,jj) = x(ii)*sin(y(jj));
    end
end
```

```
%}

z = bsxfun(@(x, y) x.*sin(y), x, y);
y = y + z;

end
```

Прочитайте Лучшие практики MATLAB онлайн: https://riptutorial.com/ru/matlab/topic/2887/лучшие-практики-matlab

глава 14: Матричные разложения

Синтаксис

1. R = chol (A);
2. [L, U] = lu (A);
3. R = qr (A);
4. T = schur (A);
5. [U, S, V] = svd (A);

Examples

Разложение Холецкого

Разложение Холецкого представляет собой метод разложения гермита, положительно определенной матрицы в верхнюю треугольную матрицу и ее транспонирование. Он может быть использован для решения систем линейных уравнений и примерно в два раза быстрее, чем LU-разложение.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = chol(A);
```

Это возвращает верхнюю треугольную матрицу. Нижняя получается транспозицией.

```
L = R';
```

Наконец, мы можем проверить правильность разложения.

```
b = (A == L*R);
```

QR-разложение

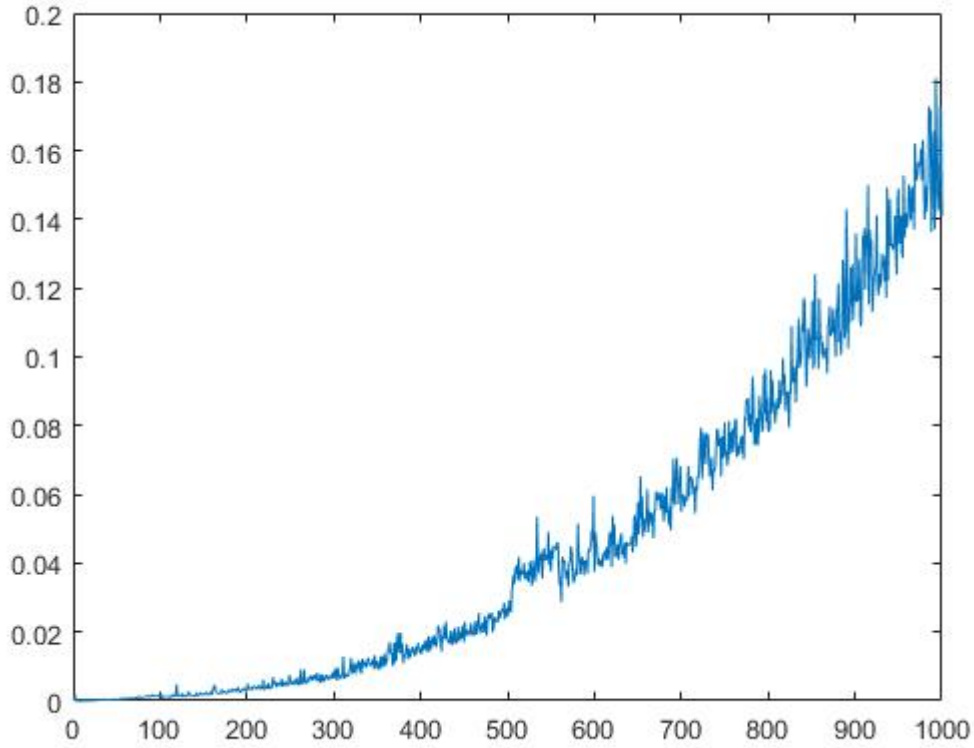
Этот метод разложит матрицу в верхнюю треугольную и ортогональную матрицу.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = qr(A);
```

Это вернет верхнюю треугольную матрицу, а следующее вернет обе матрицы.

```
[Q,R] = qr(A);
```

На следующем графике будет отображаться время выполнения `qr` зависящее от квадратного корня элементов матрицы.



Разложение LU

Таким образом, матрица будет разложена на верхнюю треугольную и нижнюю треугольную матрицу. Часто он будет использоваться для повышения производительности и стабильности (если он выполняется с перестановкой) устранения Gauß.

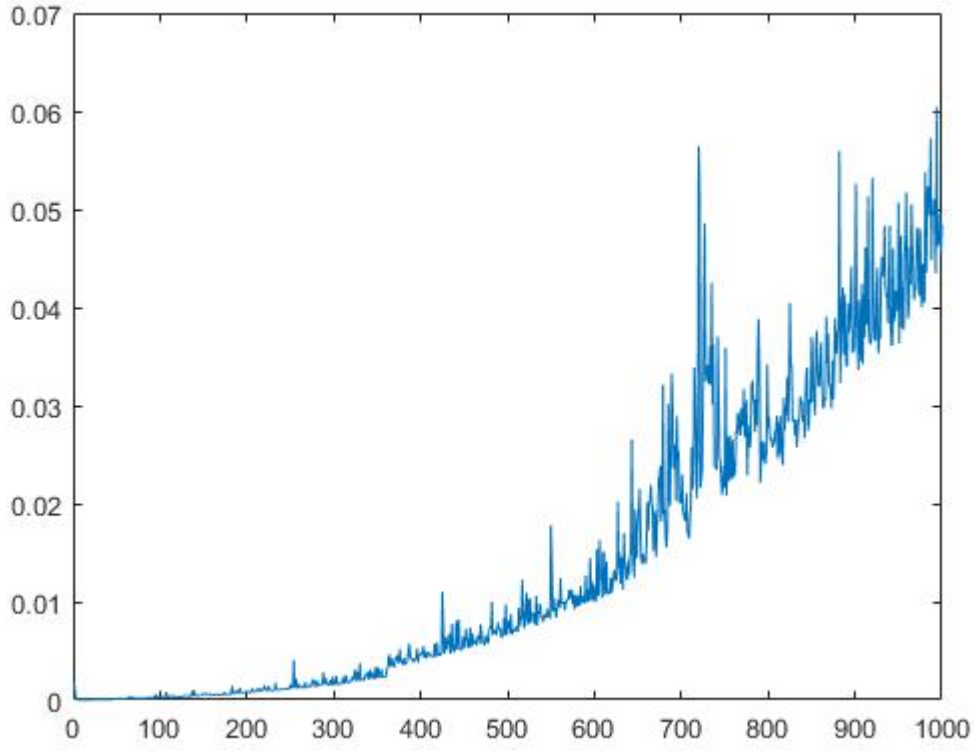
Однако довольно часто этот метод не работает или плохо работает, поскольку он нестабилен. Например

```
A = [8 1 6
     3 5 7
     4 9 2];
[L,U] = lu(A);
```

Достаточно добавить матрицу перестановок, для которой PA = LU:

```
[L,U,P]=lu(A);
```

Далее мы построим время выполнения `lu ', зависящее от квадратного корня элементов матрицы.

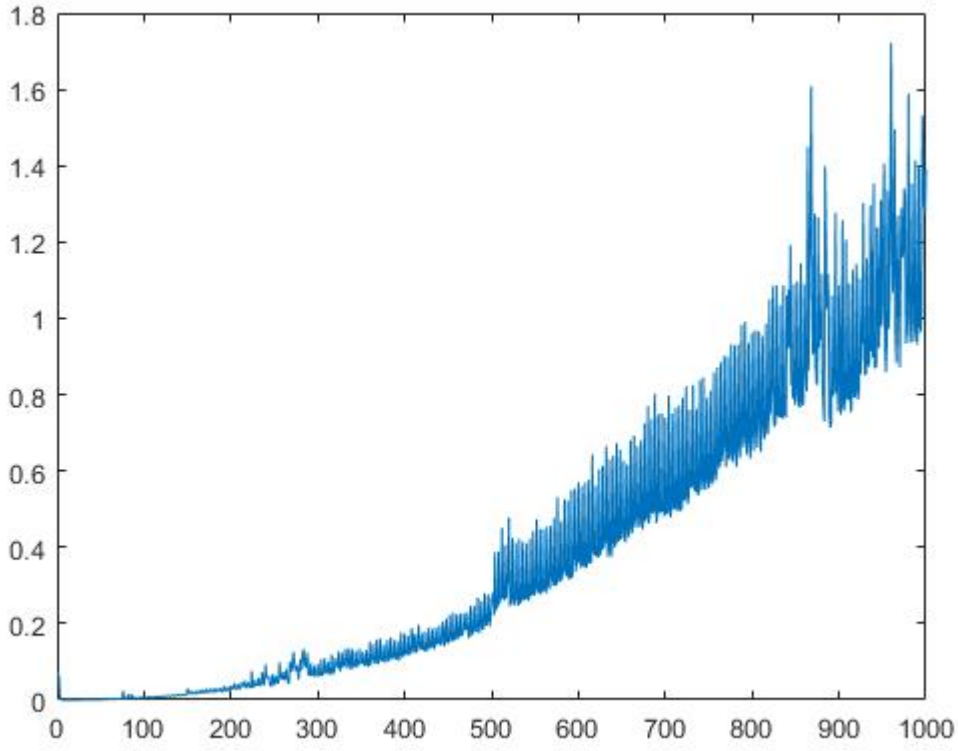


Разложение Шура

Если A - комплексная и квадратичная матрица, то существует унитарная Q такая, что Q * AQ = T = D + N, где D - диагональная матрица, состоящая из собственных значений, а N - строго верхний тридиагональный.

```
A = [3 6 1
    23 13 1
    0 3 4];
T = schur(A);
```

Мы также показываем время выполнения `schur` зависящее от квадратного корня из матричных элементов:



Разложение сингулярного значения

Учитывая m раз n матрицу A с n больше m. Разложение сингулярного значения

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

вычисляет матрицы U, S, V.

Матрица U состоит из левых сингулярных собственных векторов, являющихся собственными векторами `A*A.'` в то время как V состоит из правых сингулярных собственных значений, являющихся собственными векторами оператора `A.'*A` Матрица `S` имеет квадратные корни из собственных значений `A*A.'` и `A.'*A` на его диагонали.

Если m больше n, можно использовать

```
[U,S,V] = svd(A,'econ');
```

для выполнения разложения особых величин с экономичным размером.

Прочитайте Матричные разложения онлайн: https://riptutorial.com/ru/matlab/topic/6163/матричные-разложения

глава 15: Многопоточность

Examples

Использование parfor для параллелизации цикла

Вы можете использовать `parfor` для выполнения итераций цикла параллельно:

Пример:

```
poolobj = parpool(2);       % Open a parallel pool with 2 workers

s = 0;                      % Performing some parallel Computations
parfor i=0:9
    s = s + 1;
end
disp(s)                     % Outputs '10'

delete(poolobj);            % Close the parallel pool
```

Примечание: `parfor` не может быть вложен напрямую. Для `parfor` nesting используйте функцию в parse `parfor` и добавьте второй `parfor` в эту функцию.

Пример:

```
parfor i = 1:n
[op] = fun_name(ip);
end

function [op] = fun_name(ip)
parfor j = 1:length(ip)
% Some Computation
end
```

Когда использовать парфор

В принципе, `parfor` рекомендуется в двух случаях: много итераций в вашем цикле (например, `1e10`), или если каждая итерация занимает очень много времени (например, `eig(magic(1e4))`). Во втором случае вы можете рассмотреть возможность использования `spmd` . Причина `parfor` медленнее , чем `for` цикла для коротких диапазонов или быстрых итераций накладных расходов , необходимых для управления всех работников правильно, а не просто делать расчет.

Также много функций имеют неявная многопоточность встроенной , делая `parfor` цикл не более эффективным, при использовании этих функций, чем последовательная `for` цикла, так как уже используются все ядра. `parfor` действительно будет иметь вред, так как он имеет накладные расходы распределения, в то время как он параллелен функции, которую

вы пытаетесь использовать.

Рассмотрим следующий пример , чтобы увидеть поведение `for` в противоположность тому , что из `parfor` . Сначала откройте параллельный пул, если вы еще этого не сделали:

```
gcp; % Opens a parallel pool using your current settings
```

Затем выполните пару больших циклов:

```
n = 1000; % Iteration number
EigenValues = cell(n,1); % Prepare to store the data
Time = zeros(n,1);
for ii = 1:n
tic
    EigenValues{ii,1} = eig(magic(1e3)); % Might want to lower the magic if it takes too long
Time(ii,1) = toc; % Collect time after each iteration
end

figure; % Create a plot of results
plot(1:n,t)
title 'Time per iteration'
ylabel 'Time [s]'
xlabel 'Iteration number[-]';
```

Затем сделайте то же самое с `parfor` а не `for` . Вы заметите, что среднее время на итерацию увеличивается. Помните, однако, что `parfor` использует всех доступных работников, поэтому общее время (`sum(Time)`) должно быть разделено на количество ядер на вашем компьютере.

Таким образом, в то время как время , чтобы сделать каждую отдельную итерацию идет вверх , используя `parfor` относительно использования `for` , общее время значительно снижается.

Выполнение команд параллельно с помощью оператора «Single Program, Multiple Data» (SPMD)

В отличие от параллельного for-loop (`parfor`), который принимает итерации цикла и распределяет их между несколькими потоками, оператор одной программы, несколько данных (`spmd`) принимает серию команд и распределяет их по **всем** потокам, так что каждый thread выполняет команду и сохраняет результаты. Учти это:

```
poolobj = parpool(2);    % open a parallel pool with two workers

spmd
    q = rand(3);         % each thread generates a unique 3x3 array of random numbers
end

q{1}             % q is called like a cell vector
q{2}             % the values stored in each thread may be accessed by their index

delete(poolobj)  % if the pool is closed, then the data in q will no longer be accessible
```

Важно отметить, что каждый поток может быть доступен во время блока `spmd` по индексу потока (также называемому лабораторным индексом или `labindex`):

```
poolobj = parpool(2);        % open a parallel pool with two workers

spmd
    q = rand(labindex + 1);  % each thread generates a unique array of random numbers
end

size(q{1})                   % the size of q{1} is 2x2
size(q{2})                   % the size of q{2} is 3x3

delete(poolobj)              % q is no longer accessible
```

В обоих примерах `q` представляет собой составной объект , который может быть инициализирован командой `q = Composite()` . Важно отметить, что составные объекты доступны только во время работы пула.

Используя командную команду для параллельного выполнения различных вычислений

Для того, чтобы использовать многопоточность в MATLAB можно использовать `batch` команду. Обратите внимание, что у вас должен быть установлен инструментарий Parallel Computing.

Например, для трудоемкого сценария,

```
for ii=1:1e8
   A(ii)=sin(ii*2*pi/1e8);
end
```

для запуска его в пакетном режиме можно использовать следующее:

```
job=batch("da")
```

который позволяет MATLAB работать в пакетном режиме и позволяет одновременно использовать MATLAB для выполнения других задач, таких как добавление более пакетных процессов.

Чтобы получить результаты после завершения задания и загрузить массив `A` в рабочую область:

```
load(job, 'A')
```

И, наконец, откройте «GUID для монитора» из « *Домашняя»* → « *Среда»* → « *Параллельный»* → « *Задания монитора»* и удалите задание с помощью:

```
delete(job)
```

Чтобы загрузить функцию для пакетной обработки, просто используйте этот оператор, где `fcn` - имя функции, `N` - количество выходных массивов, а `x1` , ... , `xn` - входные массивы:

```
j=batch(fcn, N, {x1, x2, ..., xn})
```

Прочитайте Многопоточность онлайн: https://riptutorial.com/ru/matlab/topic/4378/многопоточность

глава 16: Недокументированные функции

замечания

- Использование недокументированных функций считается рискованной практикой [1] , так как эти функции могут меняться без предупреждения или просто работать по-разному в разных версиях MATLAB. По этой причине рекомендуется использовать защитные методы программирования , такие как включение недокументированных фрагментов кода в блоки `try/catch` с документально подтвержденными резервными копиями.

Examples

Совместимые с C ++ вспомогательные функции

Использование **Matlab Coder** иногда отрицает использование некоторых очень общих функций, если они не совместимы с C ++. Относительно часто существуют **недокументированные вспомогательные функции** , которые могут использоваться как замены.

Ниже приведен полный список поддерживаемых функций. ,

И после сбора альтернатив для не поддерживаемых функций:

Функции `sprintf` и `sprintfc` весьма схожи, первый возвращает *массив символов* , последний - *строку ячейки* :

```
str = sprintf('%i',x)    % returns '5' for x = 5
str = sprintfc('%i',x)   % returns {'5'} for x = 5
```

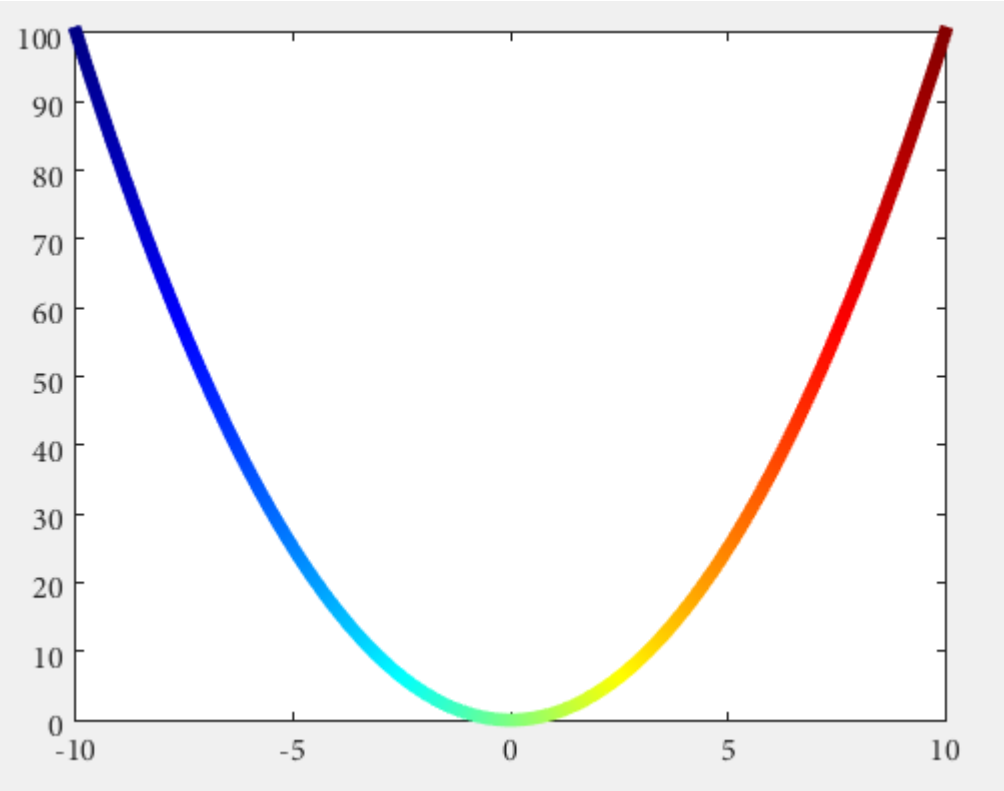
Однако `sprintfc` совместим с C ++, поддерживаемым Matlab Coder, а `sprintf` - нет.

Цветные 2D-линии с цветовыми данными в третьем измерении

В версиях MATLAB до **R2014b** , используя старый графический движок HG1, было не очевидно, как создавать цветные 2D-графики . С выпуском нового графического движка HG2 возникла новая **недокументированная функция, представленная Яиром Альтманом** :

```
n = 100;
x = linspace(-10,10,n); y = x.^2;
p = plot(x,y,'r', 'LineWidth',5);
```

```
% modified jet-colormap
cd = [uint8(jet(n)*255) uint8(ones(n,1))].';

drawnow
set(p.Edge, 'ColorBinding','interpolated', 'ColorData',cd)
```



Полупрозрачные маркеры в линиях и разбросах

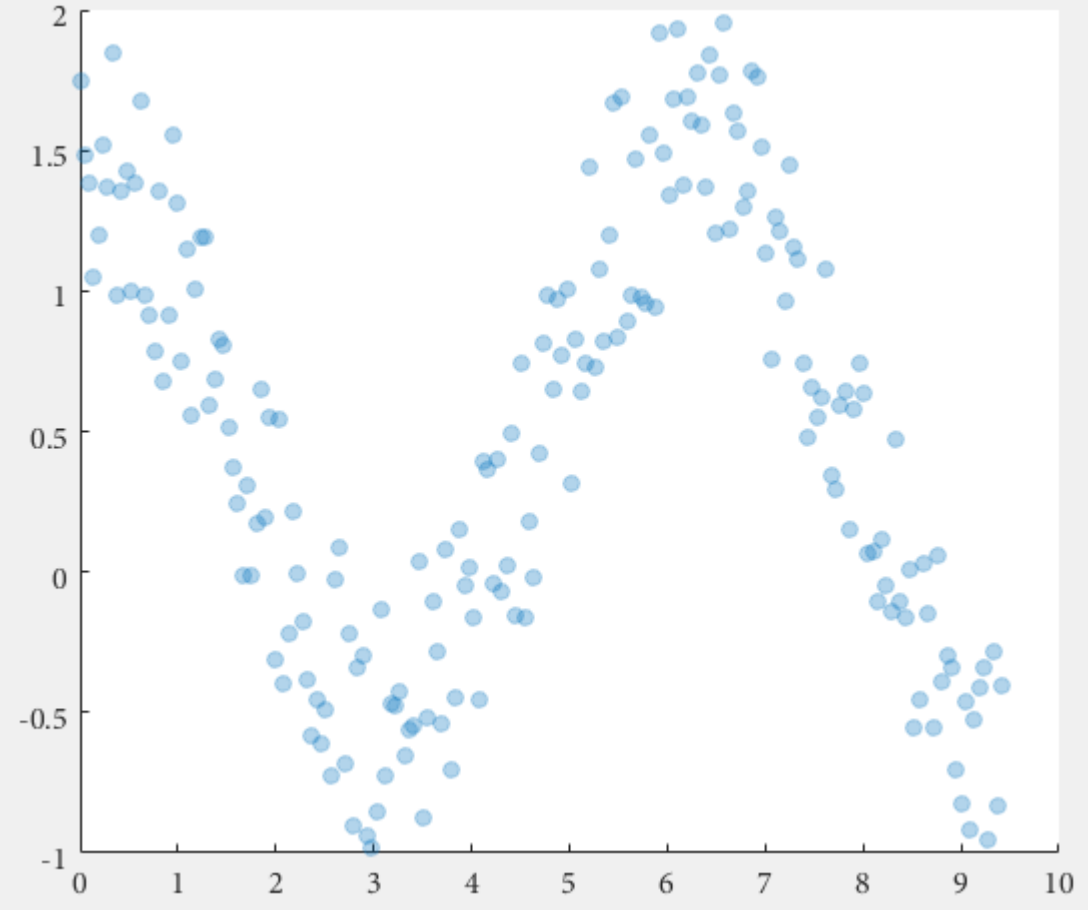
Поскольку **Matlab R2014b** легко получить полупрозрачные маркеры для линейных и рассеивающих графиков, используя **недокументированные функции, представленные Яиром Альтманом** .

Основная идея состоит в том, чтобы получить скрытый дескриптор маркеров и применить значение **<1** для последнего значения в `EdgeColorData` для достижения желаемой прозрачности.

Здесь мы идем за `scatter` :

```
%// example data
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);

%// plot scatter, get handle
h = scatter(x,y);
drawnow; %// important

%// get marker handle
hMarkers = h.MarkerHandle;

%// get current edge and face color
```

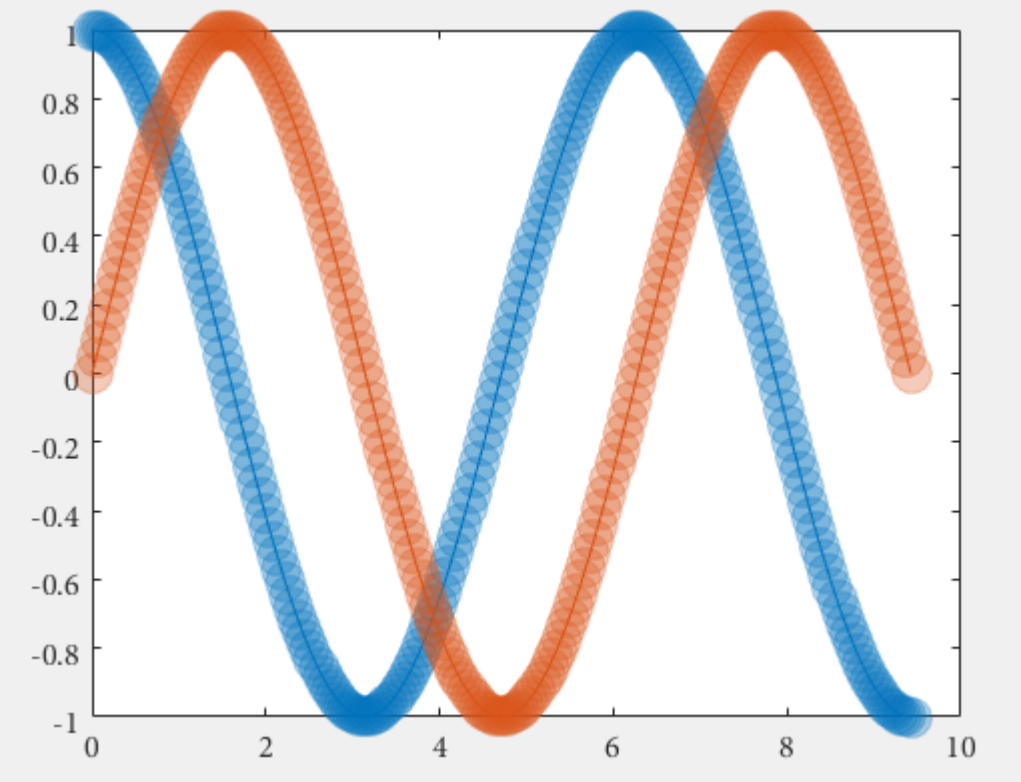
```
edgeColor = hMarkers.EdgeColorData
faceColor = hMarkers.FaceColorData

%// set face color to the same as edge color
faceColor = edgeColor;

%// opacity
opa = 0.3;

%// set marker edge and face color
hMarkers.EdgeColorData = uint8( [edgeColor(1:3); 255*opa] );
hMarkers.FaceColorData = uint8( [faceColor(1:3); 255*opa] );
```



и для линейного `plot`

```
%// example data
x = linspace(0,3*pi,200);
y1 = cos(x);
y2 = sin(x);

%// plot scatter, get handle
h1 = plot(x,y1,'o-','MarkerSize',15); hold on
h2 = plot(x,y2,'o-','MarkerSize',15);
drawnow; %// important

%// get marker handle
h1Markers = h1.MarkerHandle;
h2Markers = h2.MarkerHandle;

%// get current edge and face color
```

```
edgeColor1 = h1Markers.EdgeColorData;
edgeColor2 = h2Markers.EdgeColorData;

%// set face color to the same as edge color
faceColor1 = edgeColor1;
faceColor2 = edgeColor2;

%// opacity
opa = 0.3;

%// set marker edge and face color
h1Markers.EdgeColorData = uint8( [edgeColor1(1:3); 255*opa] );
h1Markers.FaceColorData = uint8( [faceColor1(1:3); 255*opa] );
h2Markers.EdgeColorData = uint8( [edgeColor2(1:3); 255*opa] );
h2Markers.FaceColorData = uint8( [faceColor2(1:3); 255*opa] );
```



Маркерные ручки, которые используются для манипуляции, создаются с помощью рисунка. Команда **`drawnow`** обеспечивает создание фигуры перед **`drawnow`** последующих команд и позволяет избежать ошибок в случае задержек.

Контурные сюжеты - настройка текстовых меток

При отображении меток на контурах Matlab не позволяет вам контролировать формат чисел, например, чтобы перейти к научной нотации.

Отдельные текстовые объекты являются обычными текстовыми объектами, но то, как вы их получаете, недокументировано. Вы `TextPrims` доступ к ним из свойства `TextPrims` контура.

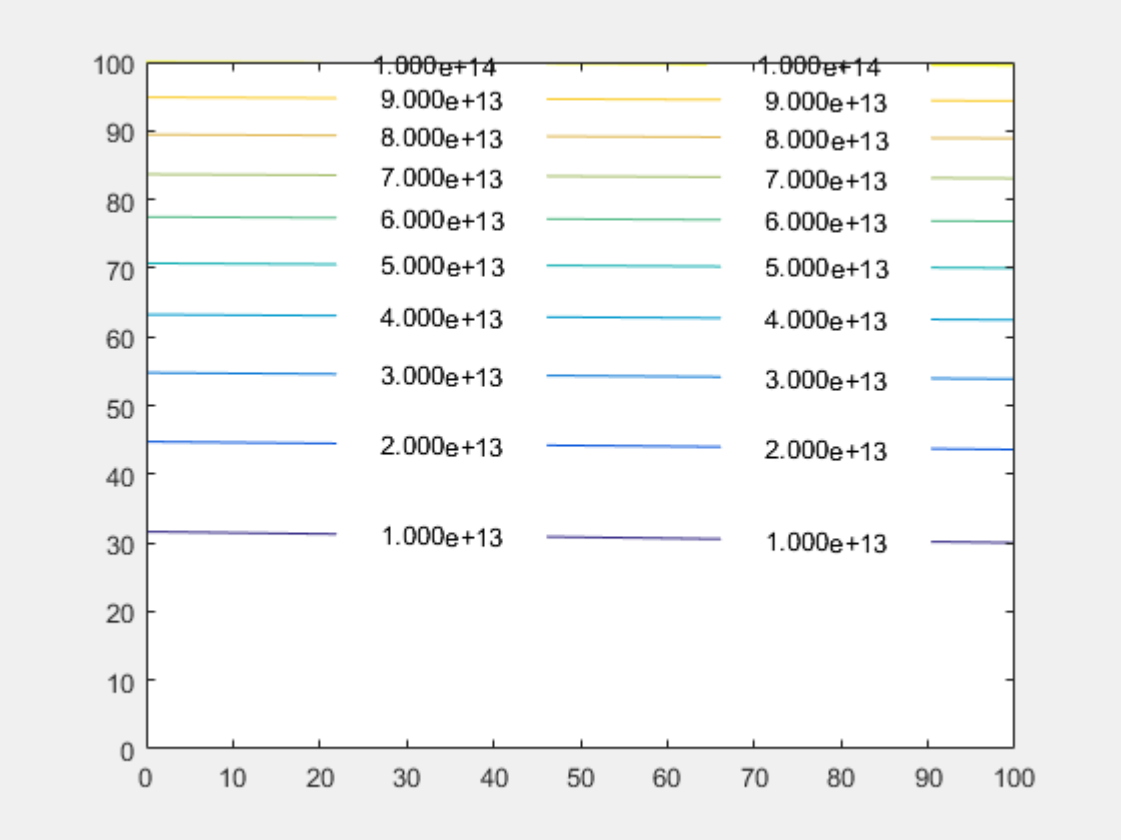
```
figure
[X,Y]=meshgrid(0:100,0:100);
Z=(X+Y.^2)*1e10;
```

```
  [C,h]=contour(X,Y,Z);
  h.ShowText='on';
  drawnow();
  txt = get(h,'TextPrims');
  v = str2double(get(txt,'String'));
  for ii=1:length(v)
    set(txt(ii),'String',sprintf('%0.3e',v(ii)))
  end
```

Обратите внимание : что вы должны добавить команду `drawnow` чтобы заставить Matlab рисовать контуры, число и местоположение объектов txt определяются только тогда, когда контуры фактически нарисованы, поэтому текстовые объекты создаются тогда.

Тот факт, что объекты txt создаются при рисовании контуров, означает, что они пересчитываются каждый раз, когда график перерисовывается (например, изменение размера фигуры). Чтобы управлять этим, вам необходимо прослушать `undocumented event MarkedClean` :

```
function customiseContour
  figure
  [X,Y]=meshgrid(0:100,0:100);
  Z=(X+Y.^2)*1e10;
  [C,h]=contour(X,Y,Z);
  h.ShowText='on';
  % add a listener and call your new format function
  addlistener(h,'MarkedClean',@(a,b)ReFormatText(a))
end
function ReFormatText(h)
  % get all the text items from the contour
  t = get(h,'TextPrims');
  for ii=1:length(t)
    % get the current value (Matlab changes this back when it
    %   redraws the plot)
    v = str2double(get(t(ii),'String'));
    % Update with the format you want - scientific for example
    set(t(ii),'String',sprintf('%0.3e',v));
  end
end
```



Пример, протестированный с использованием Matlab r2015b в Windows

Добавление / добавление записей в существующую легенду

Существующие легенды трудно справиться. Например, если ваш сюжет имеет две строки, но только один из них имеет запись с легендой, и это должно остаться таким образом, то добавление третьей строки с записью легенды может быть затруднено. Пример:

```
figure
hold on
fplot(@sin)
fplot(@cos)
legend sin  % Add only a legend entry for sin
hTan = fplot(@tan);  % Make sure to get the handle, hTan, to the graphics object you want to
add to the legend
```

Теперь, чтобы добавить запись легенды для `tan` , но не для `cos` , любая из следующих строк не будет делать трюк; они все так или иначе терпят неудачу:

```
legend tangent  % Replaces current legend -> fail
legend -DynamicLegend  % Undocumented feature, adds 'cos', which shouldn't be added -> fail
legend sine tangent  % Sets cos DisplayName to 'tangent' -> fail
legend sine '' tangent  % Sets cos DisplayName, albeit empty -> fail
legend(f)
```

К счастью, недокументированное свойство легенды, называемое `PlotChildren` отслеживает детей родительской фигуры [1] . Таким образом, путь заключается в том, чтобы явно установить детей легенды с помощью свойства `PlotChildren` следующим образом:

```
hTan.DisplayName = 'tangent';  % Set the graphics object's display name
l = legend;
l.PlotChildren(end + 1) = hTan;  % Append the graphics handle to legend's plot children
```

Легенда обновляется автоматически, если объект добавлен или удален из свойства `PlotChildren` .

[1] Действительно: цифра. Вы можете добавить ребенка любой фигуры с свойством `DisplayName` к любой легенде на рисунке, например, из другого подзаголовка. Это потому, что легенда сама по себе представляет собой объект осей.

Протестировано на MATLAB R2016b

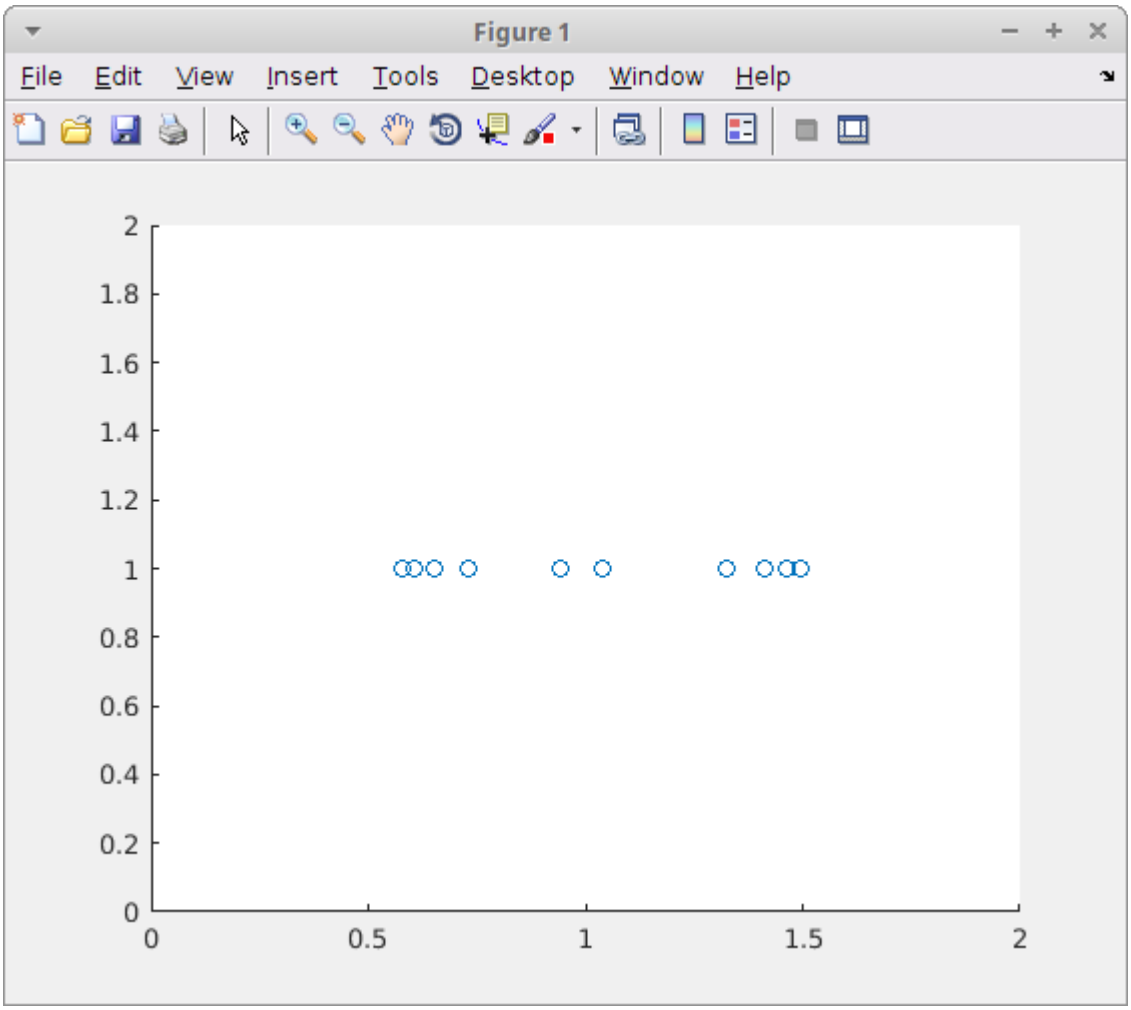
Фрагмент дрожания графика

Функция `scatter` имеет два недокументированных свойства `'jitter'` и `'jitterAmount'` которые позволяют `'jitterAmount'` только данные по оси x. Это относится к Matlab 7.1 (2005) и, возможно, ранее.

Чтобы включить эту функцию установки `'jitter'` свойство `'on'` и установить `'jitterAmount'` свойства до требуемого абсолютного значения (по умолчанию `0.2`).

Это очень полезно, когда мы хотим визуализировать перекрывающиеся данные, например:

```
scatter(ones(1,10), ones(1,10), 'jitter', 'on', 'jitterAmount', 0.5);
```



Подробнее о недокументированной Matlab

Прочитайте Недокументированные функции онлайн:
https://riptutorial.com/ru/matlab/topic/2383/недокументированные-функции

глава 17: Обработка изображения

Examples

Основные изображения ввода / вывода

```
>> img = imread('football.jpg');
```

Используйте `imread` для чтения файлов изображений в матрицу в MATLAB.
Как только вы `imread` изображение, оно хранится в памяти как ND-массив:

```
>> size(img)
ans =
 256   320     3
```

Изображение `'football.jpg'` имеет 256 строк и 320 столбцов и имеет 3 цветовых канала: красный, зеленый и синий.

Теперь вы можете отразить это:

```
>> mirrored = img(:, end:-1:1, :);  %// like mirroring any ND-array in Matlab
```

И, наконец, напишите это как изображение, используя `imwrite` :

```
>> imwrite(mirrored, 'mirrored_football.jpg');
```

Извлечение изображений из Интернета

Пока у вас есть подключение к Интернету, вы можете читать изображения из гиперссылки

```
I=imread('https://cdn.sstatic.net/Sites/stackoverflow/company/img/logos/so/so-logo.png');
```

Фильтрация с использованием 2D FFT

Как и для сигналов 1D, можно фильтровать изображения, применяя преобразование Фурье, умножая его на фильтр в частотной области и преобразовывая обратно в пространственную область. Вот как вы можете применять фильтры с высоким или низким пропусканием к изображению с помощью Matlab:

Пусть `image` будет оригинальным, нефильтрованное изображение, вот как вычислить его 2 D FFT:

```
ft = fftshift(fft2(image));
```

Теперь, чтобы исключить часть спектра, нужно установить его значения пикселей равными 0. Пространственная частота, содержащаяся в исходном изображении, отображается от центра к краям (после использования `fftshift`). Чтобы исключить низкие частоты, мы установим центральную круговую область на 0.

Вот как создать двоичную маску в форме диска с радиусом `D` используя встроенную функцию:

```
[x y ~] = size(ft);
D = 20;
mask = fspecial('disk', D) == 0;
mask = imresize(padarray(mask, [floor((x/2)-D) floor((y/2)-D)], 1, 'both'), [x y]);
```

Маскирование изображения частотной области может быть выполнено путем умножения точки БПФ на двоичную маску, полученную выше:

```
masked_ft = ft .* mask;
```

Теперь давайте вычислим обратный БПФ:

```
filtered_image = ifft2(ifftshift(masked_ft), 'symmetric');
```

Высокими частотами изображения являются острые края, поэтому этот фильтр верхних частот может использоваться для резкости изображений.

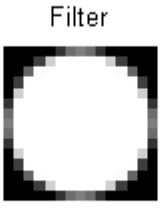
Фильтрация изображений

Допустим, у вас есть изображение `rgbImg` , например, чтение с использованием `imread` .

```
>> rgbImg = imread('pears.png');
>> figure, imshow(rgbImg), title('Original Image')
```

Используйте `fspecial` для создания 2D-фильтра:

```
>> h = fspecial('disk', 7);
>> figure, imshow(h, []), title('Filter')
```



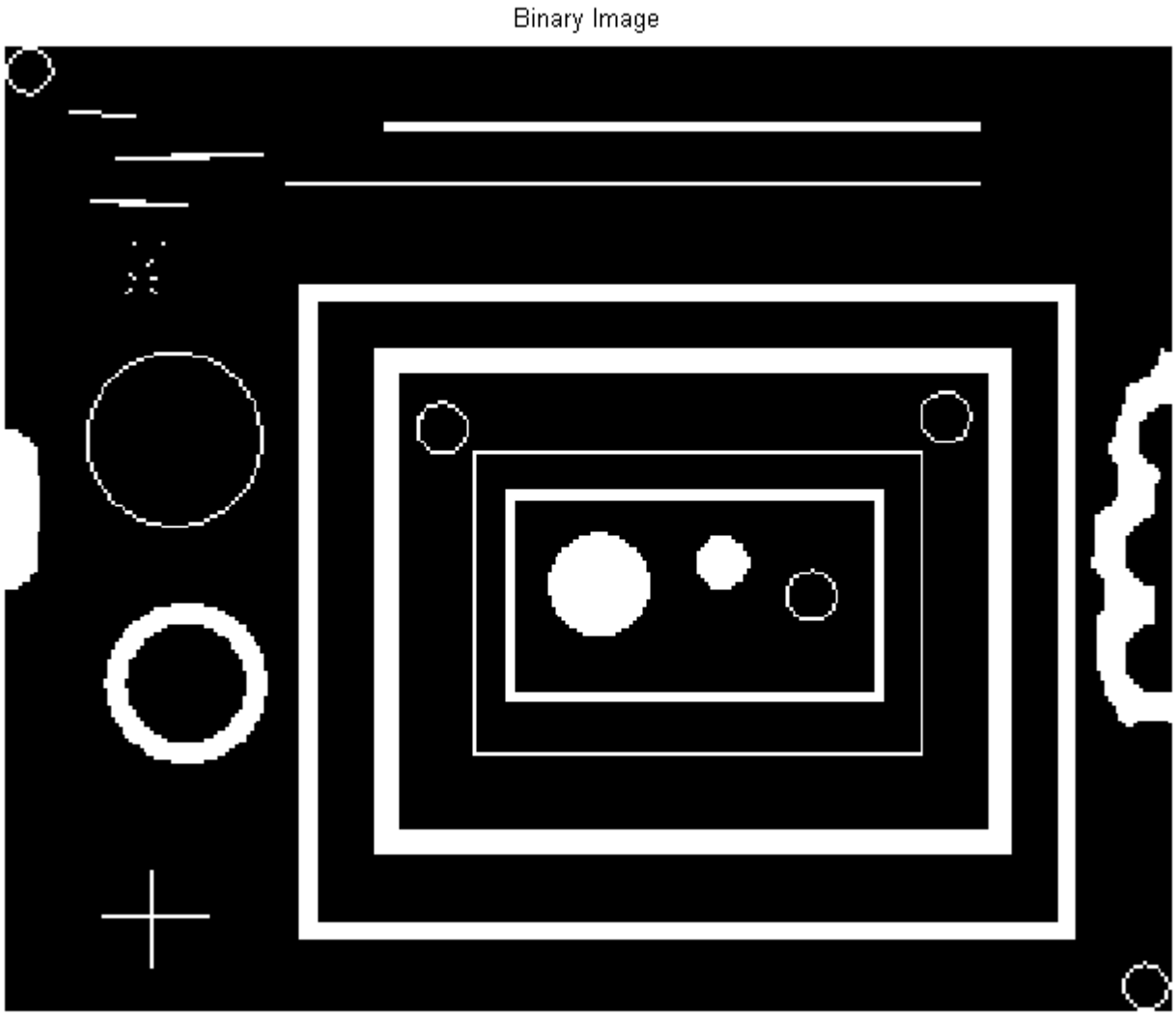
Используйте `imfilter` чтобы применить фильтр к изображению:

```
>> filteredRgbImg = imfilter(rgbImg, h);
>> figure, imshow(filteredRgbImg), title('Filtered Image')
```

Измерение свойств подключенных регионов

Начиная с бинарного изображения, `bwImg` , который содержит несколько связанных объектов.

```
>> bwImg = imread('blobs.png');
>> figure, imshow(bwImg), title('Binary Image')
```



Чтобы измерить свойства (например, область, центр и т. Д.) `regionprops` объекта на изображении, используйте `regionprops` :

```
>> stats = regionprops(bwImg, 'Area', 'Centroid');
```

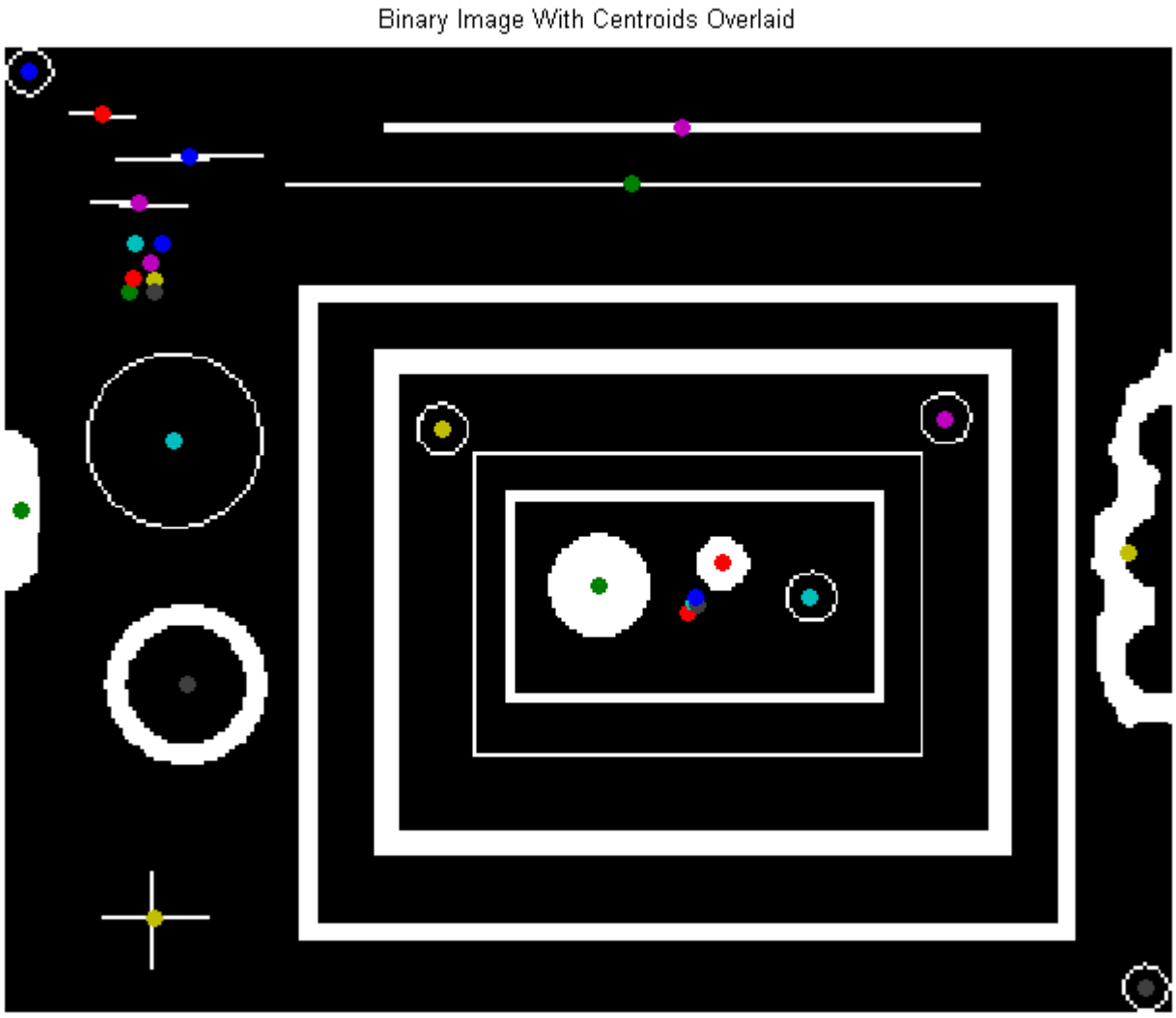
`stats` - это массив структур, который содержит структуру для каждого объекта изображения. Доступ к измеренному свойству объекта прост. Например, чтобы отобразить область первого объекта, просто,

```
>> stats(1).Area

ans =

    35
```

Визуализируйте центроиды объектов, наложив их на исходное изображение.

```
>> figure, imshow(bwImg), title('Binary Image With Centroids Overlaid')
>> hold on
>> for i = 1:size(stats)
```

```
scatter(stats(i).Centroid(1), stats(i).Centroid(2), 'filled');
end
```



Прочитайте Обработка изображения онлайн: https://riptutorial.com/ru/matlab/topic/3274/обработка-изображения

глава 18: Общие ошибки и ошибки

Examples

Не называйте переменную с именем существующей функции

Уже существует функция `sum()` . В результате, если мы будем называть переменную с тем же именем

```
sum = 1+3;
```

и если мы попытаемся использовать эту функцию, пока переменная все еще существует в рабочей области

```
A = rand(2);
sum(A,1)
```

мы получим загадочную **ошибку** :

```
Subscript indices must either be real positive integers or logicals.
```

сначала `clear()` переменную, а затем используйте функцию

```
clear sum

sum(A,1)
ans =
       1.0826       1.0279
```

Как мы можем проверить, существует ли функция, чтобы избежать этого конфликта?

Используйте `which()` с флагом `-all` :

```
which sum -all
sum is a variable.
built-in (C:\Program Files\MATLAB\R2016a\toolbox\matlab\datafun\@double\sum)    % Shadowed
double method
...
```

Этот вывод говорит нам, что `sum` является первой переменной и что следующие методы (функции) затенены ею, то есть MATLAB сначала попытается применить наш синтаксис к переменной, а не использовать этот метод.

То, что вы видите, НЕ является тем, что вы получаете: char vs cellstring в окне команд

Это основной пример, нацеленный на новых пользователей. Он не фокусируется на объяснении разницы между `char` и `cellstring` .

Может случиться так, что вы хотите избавиться от `'` в ваших строках», хотя вы никогда их не добавляли. Фактически, это *артефакты,* которые используется в **командном окне** для различения некоторых типов.

Строка будет печатать

```
s = 'dsadasd'
s =
dsadasd
```

Ячейка будет печатать

```
c = {'dsadasd'};
c =
    'dsadasd'
```

Обратите внимание, что **одиночные кавычки** и **отступы** являются артефактами, чтобы уведомить нас о том, что `c` - это `cellstring` а не `char` . Строка фактически содержится в ячейке, т. Е.

```
c{1}
ans =
dsadasd
```

Операторы транспонирования

- `.'` является правильным способом **транспонировать** вектор или матрицу в MATLAB.
- `'` - правильный способ взять **комплексно-сопряженную транспозицию** (aka эрмитово сопряженную) вектора или матрицы в MATLAB.

Обратите внимание, что для транспонирования `.'` , перед апострофом существует **период** . Это согласуется с синтаксисом для других элементарных операций в MATLAB: `*` умножает *матрицы* , `.*` умножает *элементы матриц* вместе. Обе команды очень похожи, но концептуально очень разные. Как и другие команды MATLAB, эти операторы являются «синтаксическим сахаром», который во время выполнения превращается в «правильный» вызов функции. Подобно тому, как `==` становится оценкой функции eq , подумайте `.'` как сокращенное обозначение для `transpose` . Если бы вы только пишете `'` (без точки), вы фактически используете команду `ctranspose` , которая вычисляет сложную сопряженную транспозицию , которая также известна как эрмитова сопряженная , часто используемая в физике. Пока транспонированный вектор или матрица вещественны, оба оператора производят одинаковый результат. Но как только мы будем заниматься сложными числами , мы неизбежно столкнемся с проблемами, если не будем использовать «правильную»

стенографию. Что «правильно» зависит от вашего приложения.

Рассмотрим следующий пример матрицы `C` содержащей комплексные числа:

```
>> C = [1i, 2; 3*1i, 4]
C =
   0.0000 + 1.0000i   2.0000 + 0.0000i
   0.0000 + 3.0000i   4.0000 + 0.0000i
```

Возьмем *транспонирование,* используя сокращенное выражение `.'` (с периодом). Выход такой, как ожидалось, транспонированная форма `C`

```
>> C.'
ans =
   0.0000 + 1.0000i   0.0000 + 3.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i
```

Теперь давайте использовать `'` (без периода). Мы видим, что в дополнение к транспозиции комплексные значения также трансформируются в их *комплексные сопряжения* .

```
>> C'
ans =
   0.0000 - 1.0000i   0.0000 - 3.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i
```

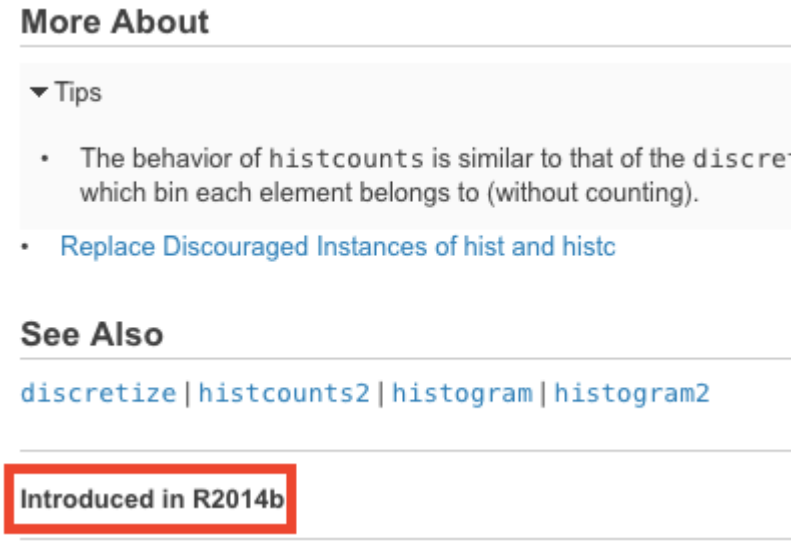
Подводя итог, если вы намереваетесь рассчитать эрмитовское сопряженное, комплексно-сопряженное транспонирование, то используйте `'` (без периода). Если вы просто хотите вычислить транспонирование без комплексного сопряжения значений, используйте `.'` (с периодом).

Неопределенная функция или метод X для входных аргументов типа Y

Это длинный способ MATLAB сказать, что он не может найти функцию, которую вы пытаетесь вызвать. Существует несколько причин, по которым вы можете получить эту ошибку:

Эта функция была введена *после* вашей текущей версии MATLAB

Онлайн-документация MATLAB обеспечивает очень приятную функцию, которая позволяет вам определить, в какой версии была введена данная функция. Он расположен в левом нижнем углу каждой страницы документации:

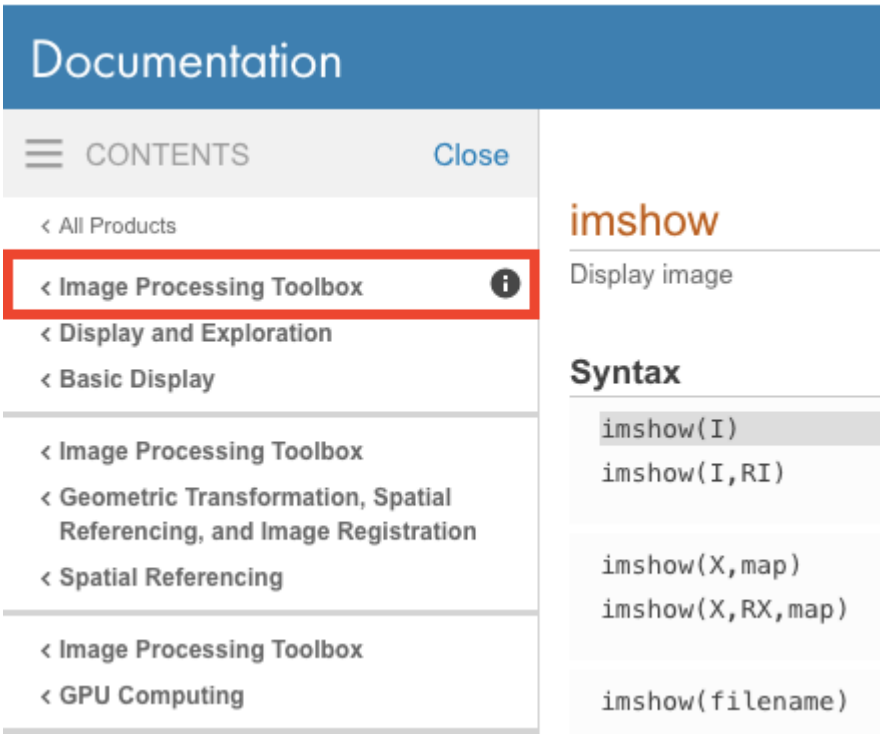


Сравните эту версию с вашей текущей версией (`ver`), чтобы определить, доступна ли эта функция в вашей конкретной версии. Если это не так, попробуйте найти архивные версии документации, чтобы найти подходящую альтернативу в своей версии.

У вас нет этого инструментария!

Базовая установка MATLAB имеет большое количество функций; однако более специализированные функциональные возможности упакованы в панели инструментов и продаются отдельно Mathworks. Документация для *всех* панелей инструментов видна, есть ли у вас панель инструментов или нет, поэтому обязательно проверьте и посмотрите, есть ли у вас соответствующий набор инструментов.

Чтобы проверить, к какой панели инструментов принадлежит данная функция, посмотрите в левом верхнем углу интерактивной документации, чтобы узнать, указан ли конкретный набор инструментов.



Затем вы можете определить, какие панели инструментов установлены вашей версией MATLAB, выпустив команду `ver` которая выведет список всех установленных наборов инструментов.

Если у вас нет установленного набора инструментов и вы хотите использовать эту функцию, вам нужно будет приобрести лицензию для этого конкретного инструментария из The Mathworks.

MATLAB не может найти функцию

Если MATLAB все еще не может найти вашу функцию, то она должна быть определяемой пользователем. Возможно, что он живет в другом каталоге и этот каталог должен быть добавлен в путь поиска для запуска вашего кода. Вы можете проверить , может ли MATLAB найти вашу функцию с помощью `which` , которая должна возвращать путь к исходному файлу.

Помните о неточности с плавающей запятой

Числа с плавающей запятой не могут представлять все действительные числа. Это известно как неточность с плавающей запятой.

Существует бесконечно много чисел с плавающей запятой, и они могут быть бесконечно длинными (например, `π`), поэтому возможность их представления идеально потребует бесконечного количества памяти. Увидев это, возникла проблема, было разработано специальное представление для хранения «реального числа» в компьютере, стандарт IEEE 754 . Короче говоря, он описывает, как компьютеры хранят этот тип чисел с

показателем и мантиссой,

```
floatnum = sign * 2^exponent * mantissa
```

При ограниченном количестве бит для каждого из них может быть достигнута только конечная точность. Чем меньше число, тем меньше разрыв между возможными числами (и наоборот!). Вы можете попробовать свои реальные цифры в этом онлайн-демо .

Помните об этом и старайтесь избегать сравнения с плавающей точкой и их использования в качестве условий остановки в циклах. Ниже приведены два примера:

Примеры: сравнение плавающих точек WRONG:

```
>> 0.1 + 0.1 + 0.1  == 0.3

ans =

  logical

   0
```

Плохая практика заключается в использовании сравнения с плавающей запятой, как показано в примере с прецедентом. Вы можете преодолеть это, взяв абсолютную величину их разницы и сравнив ее с (малым) уровнем допуска.

Ниже приведен еще один пример, где число с плавающей запятой используется как условие остановки в цикле while: **

```
k = 0.1;
while k <= 0.3
  disp(num2str(k));
  k = k + 0.1;
end

% --- Output: ---
0.1
0.2
```

Он пропускает последний ожидаемый цикл (`0.3 <= 0.3`).

Пример: Выполнено сравнение с плавающей запятой. RIGHT:

```
x = 0.1 + 0.1 + 0.1;
y = 0.3;
tolerance = 1e-10; % A "good enough" tolerance for this case.

if ( abs( x - y ) <= tolerance )
  disp('x == y');
else
  disp('x ~= y');
end
```

```
% --- Output: ---
x == y
```

Несколько замечаний:

- Как и ожидалось, теперь `x` и `y` рассматриваются как эквивалентные.
- В приведенном выше примере выбор допуска выполнялся произвольно. Таким образом, выбранное значение может не подходить для всех случаев (особенно при работе с гораздо меньшими номерами). Выбор метода *разумно* может быть выполнен с использованием функции `eps` , т. `N*eps(max(x,y))` , где `N` - некоторое конкретное заданное число. Разумным выбором для `N` , который также является достаточно разрешительным, является `1E2` (хотя в приведенной выше проблеме `N=1` также будет достаточно).

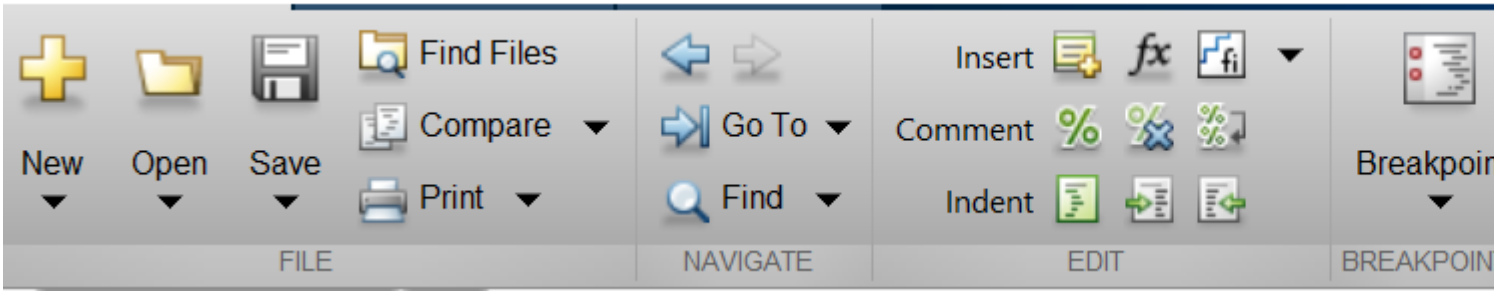
Дальнейшее чтение:

См. Эти вопросы для получения дополнительной информации о неточности с плавающей запятой:

- Почему 24.0000 не равно 24.0000 в MATLAB?
- Сбита ли математика с плавающей запятой?

Недостаточно входных аргументов

Часто начинающие разработчики MATLAB будут использовать редактор MATLAB для написания и редактирования кода, в частности пользовательских функций с входами и выходами. В верхней части есть кнопка « *Выполнить»* , которая доступна в последних версиях MATLAB:



Как только разработчик заканчивается кодом, они часто испытывают желание нажать кнопку « *Запустить»* . Для некоторых функций это будет работать нормально, но для других они получат ошибку Недопустимые `Not enough input arguments` и будут озадачены тем, почему возникает ошибка.

Причина, по которой эта ошибка может не произойти, заключается в том, что вы написали сценарий MATLAB или функцию, которая не принимает входных аргументов. С помощью кнопки *Run* запускается тестовый скрипт или запускается функция, не принимающая

входных аргументов. Если для вашей функции требуются входные аргументы, ошибка « `Not enough input arguments` будет возникать, поскольку вы написали функции, которые ожидают ввода входных данных внутри функции. Поэтому вы не можете ожидать, что функция будет работать, просто нажав кнопку « *Выполнить»* .

Чтобы продемонстрировать эту проблему, предположим, что у нас есть функция `mult` которая просто умножает две матрицы вместе:

```
function C = mult(A, B)
    C = A * B;
end
```

В последних версиях MATLAB, если вы написали эту функцию и нажали кнопку « *Запустить»* , она даст вам ожидаемую ошибку:

```
>> mult
Not enough input arguments.

Error in mult (line 2)
    C = A * B;
```

Существует два способа решить эту проблему:

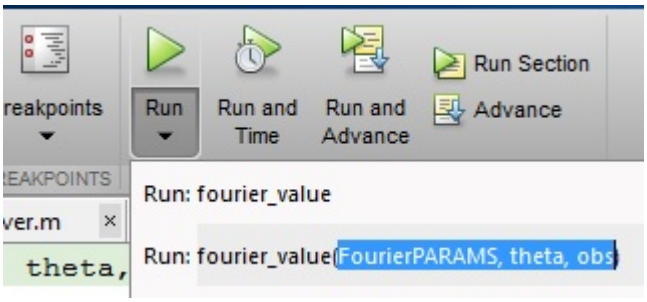
Метод №1 - с помощью командной строки

Просто создайте нужные вам данные в командной строке, затем запустите функцию, используя созданные вами входы:

```
A = rand(5,5);
B = rand(5,5);
C = mult(A,B);
```

Метод № 2 - Интерактивно через редактор

Под кнопкой « *Выполнить»* находится темная черная стрелка. Если вы нажмете на эту стрелку, вы можете указать переменные, которые хотите получить из рабочего пространства MATLAB, набрав способ, которым вы хотите вызвать функцию, точно так же, как вы видели в методе # 1. Убедитесь, что переменные, которые вы указываете внутри функции, существуют в рабочей области MATLAB:



Следите за изменениями размера массива

Некоторые общие операции в MATLAB, такие как **дифференциация** или **интеграция** , выводят результаты, которые имеют различное количество элементов, чем входные данные. Этот факт можно легко упускать из виду, что, как правило, вызывает ошибки, такие как `Matrix dimensions must agree` . Рассмотрим следующий пример:

```
t = 0:0.1:10;         % Declaring a time vector
y = sin(t);           % Declaring a function

dy_dt = diff(y);      % calculates dy/dt for y = sin(t)
```

Предположим, мы хотим построить эти результаты. Мы рассмотрим размеры массива и посмотрим:

```
size(y) is 1x101
size(t) is 1x101
```

Но:

```
size(dy_dt) is 1x100
```

Массив на один элемент короче!

Теперь представьте, что у вас есть данные измерений позиций с течением времени и вы хотите вычислить *jerk (t)* , вы получите массив на 3 элемента меньше, чем временный массив (потому что рывок является дифференцированным положением 3 раза).

```
vel = diff(y);        % calculates velocity vel=dy/dt for y = sin(t)  size(vel)=1x100
acc = diff(vel);      % calculates acceleration acc=d(vel)/dt         size(acc)=1x99
jerk = diff(acc);     % calculates jerk jerk=d(acc)/dt                size(jerk)=1x98
```

И тогда операции вроде:

```
x = jerk .* t;          % multiplies jerk and t element wise
```

возвращают ошибки, поскольку размеры матрицы не согласуются.

Чтобы вычислить такие операции, как описано выше, вам необходимо настроить размер

массива большего размера, чтобы он соответствовал меньшему. Вы также можете запустить регрессию (`polyfit`) с вашими данными, чтобы получить многочлен для ваших данных.

Ошибки несоответствия размеров

Ошибки **несоответствия** размеров обычно появляются, когда:

- Не обращая внимания на форму возвращаемых переменных из вызовов функций / методов. Во многих встроенных функциях MATLAB матрицы преобразуются в векторы для ускорения вычислений, а возвращаемая переменная может быть вектором, а не ожидаемой матрицей. Это также распространенный сценарий, когда задействована логическая маскировка .
- Использование несовместимых размеров массива при вызове неявного расширения массива .

Использование «i» или «j» в качестве мнимой единицы, индексов цикла или общей переменной.

Рекомендация

Поскольку символы `i` и `j` могут представлять существенно разные вещи в MATLAB, их использование в качестве индексов цикла разделило сообщество пользователей MATLAB с возрастом. Хотя некоторые исторические соображения производительности могут помочь балансу опираться на одну сторону, это уже не так, и теперь выбор полностью зависит от вас и методов кодирования, которые вы выбираете.

В настоящее время официальными рекомендациями Mathworks являются:

- Поскольку `i` является функцией, ее можно переопределить и использовать в качестве переменной. Однако лучше избегать использования `i` и `j` для имен переменных, если вы собираетесь использовать их в сложной арифметике.
- Для скорости и улучшенной устойчивости в сложной арифметике используйте `1i` и `1j` вместо `i` и `j` .

По умолчанию

В MATLAB по умолчанию буквы `i` и `j` являются встроенными именами `function` , которые оба относятся к мнимой единице в сложном домене.

Таким образом, по умолчанию `i = j = sqrt(-1)` .

```
>> i
ans =
   0.0000 + 1.0000i
>> j
ans =
   0.0000 + 1.0000i
```

и как и следовало ожидать:

```
>> i^2
ans =
    -1
```

Используя их как переменную (для индексов цикла или другой переменной)

MATLAB позволяет использовать встроенное имя функции в качестве стандартной переменной. В этом случае используемый символ больше не будет указывать на встроенную функцию, а на вашу собственную пользовательскую переменную. Однако эта практика обычно не рекомендуется, так как это может привести к путанице, сложной отладке и обслуживанию (*см. Другой пример* *do-not-name-a-variable-with-an-existing-function-name*).

Если вы глубоко педантичны относительно соблюдения конвенций и лучших практик, вы избежите использовать их в качестве индексов цикла на этом языке. Тем не менее, это разрешено компилятором и совершенно функционально, поэтому вы также можете сохранить старые привычки и использовать их в качестве итераторов цикла.

```
>> A = nan(2,3);
>> for i=1:2        % perfectly legal loop construction
       for j = 1:3
        A(i, j) = 10 * i + j;
       end
   end
```

Обратите внимание, что индексы цикла не заканчиваются в конце цикла, поэтому они сохраняют новое значение.

```
>> [ i ; j ]
ans =
     2
     3
```

Если вы используете их как переменную, убедитесь, что **они инициализированы** до их использования. В цикле над MATLAB они автоматически инициализируют их при подготовке цикла, но если они не инициализированы правильно, вы можете быстро увидеть, что вы можете непреднамеренно ввести в свой результат `complex` числа.

Если позже, вам нужно отменить затенение встроенной функции (=, например, вы хотите, чтобы `i` и `j` отображали мнимую единицу снова), вы можете `clear` переменные:

```
>> clear i j
```

Теперь вы понимаете, что резервирование Mathworks позволяет использовать их в качестве индексов цикла, *если вы собираетесь использовать их в сложной арифметике* . Ваш код будет пронизан переменными инициализациями и `clear` командами, лучшим способом сбить с толку самого серьезного программиста (*да, вы там! ...*) и аварийных программ, ожидающих своего появления.

Если ожидаемая сложная арифметика не ожидается, использование `i` и `j` вполне функционально, и нет штрафа за производительность.

Используя их как мнимую единицу:

Если ваш код должен иметь дело со `complex` числами, то `i` и `j` , безусловно, пригодится. Однако для устранения неоднозначности и даже для выступлений рекомендуется использовать полную форму вместо сокращенного синтаксиса. Полная форма равна `1i` (или `1j`).

```
>> [ i ; j ; 1i ; 1j]
ans =
   0.0000 + 1.0000i
   0.0000 + 1.0000i
   0.0000 + 1.0000i
   0.0000 + 1.0000i
```

Они представляют одно и то же значение `sqrt(-1)` , но более поздняя форма:

- более явным, семантическим образом.
- (кто-то, смотрящий на ваш код позже, не должен будет читать код, чтобы узнать, является ли `i` или `j` переменной или мнимой единицей).
- быстрее (источник: Mathworks).

Обратите внимание, что полный синтаксис `1i` действителен с любым числом, предшествующим символу:

```
>> a = 3 + 7.8j
a =
   3.0000 + 7.8000i
```

Это единственная функция, с которой вы можете придерживаться номера без оператора между ними.

Ловушки

Хотя их использование в качестве *мнимой единицы* **ИЛИ** *переменной* является совершенно законным, вот лишь небольшой пример того, как путаница могла бы возникнуть, если оба метода получаются смешанными:

Давайте переопределим `i` и сделаем его переменной:

```
>> i=3
i =
     3
```

Теперь `i` - *переменная* (удерживающая значение `3`), но мы только преувеличиваем *сокращенную* нотацию мнимой единицы, полная форма по-прежнему интерпретируется правильно:

```
>> 3i
ans =
   0.0000 + 3.0000i
```

Что теперь позволяет нам строить самые неясные формулировки. Позвольте вам оценить читаемость всех следующих конструкций:

```
>> [ i ; 3i ; 3*i ; i+3i ; i+3*i ]
ans =
   3.0000 + 0.0000i
   0.0000 + 3.0000i
   9.0000 + 0.0000i
   3.0000 + 3.0000i
  12.0000 + 0.0000i
```

Как вы можете видеть, каждое значение в массиве выше возвращает другой результат. Хотя каждый результат действителен (при условии, что это было первоначальное намерение), большинство из вас признают, что было бы хорошим кошмаром прочитать код, пронизанный такими конструкциями.

Использование `length` для многомерных массивов

В распространенной ошибке кодеры MATLAB используют функцию `length` для матриц (в отличие от **векторов** , для которых она предназначена). Функция `length` , как указано в ее документации , « *возвращает длину наибольшего размера массива* » ввода.

Для векторов возвращаемое значение `length` имеет два разных значения:

1. Общее число элементов в векторе.
2. Наибольшая размерность вектора.

В отличие от векторов, приведенные выше значения не будут равны для массивов более чем одного не-одиночного (т. Е. Размера которого больше `1`). Вот почему использование `length` для матриц неоднозначно. Вместо этого рекомендуется использовать одну из следующих функций, даже при работе с векторами, чтобы сделать код совершенно понятным:

1. `size(A)` - возвращает вектор строки, элементы которого содержат количество элементов вдоль соответствующего размера `A`
2. `numel(A)` - возвращает количество элементов в `A` Эквивалентен `prod(size(A))` .
3. `ndims(A)` - возвращает количество измерений в массиве `A` Эквивалент `numel(size(A))` .

Это особенно важно при написании «перспективных», векторизованных библиотечных функций, входы которых неизвестны заранее и могут иметь различные размеры и формы.

Прочитайте Общие ошибки и ошибки онлайн: https://riptutorial.com/ru/matlab/topic/973/общие-ошибки-и-ошибки

глава 19: Объектно-ориентированное программирование

Examples

Определение класса

Класс можно определить с помощью `classdef` в файле `.m` с тем же именем, что и класс. Файл может содержать `classdef` ... `end` блока и локальные функции для использования в методах класса.

Наиболее общее определение класса MATLAB имеет следующую структуру:

```
classdef (ClassAttribute = expression, ...) ClassName < ParentClass1 & ParentClass2 & ...

   properties (PropertyAttributes)
      PropertyName
   end

   methods (MethodAttributes)
      function obj = methodName(obj,arg2,...)
         ...
      end
   end

   events (EventAttributes)
      EventName
   end

   enumeration
      EnumName
   end

end
```

Документация MATLAB: атрибуты класса, атрибуты свойств, атрибуты метода, атрибуты событий , ограничения класса перечисления .

Пример:

Класс под названием `Car` можно определить в файле `Car.m` как

```
classdef Car < handle % handle class so properties persist
    properties
        make
        model
        mileage = 0;
    end
```

```
    methods
        function obj = Car(make, model)
            obj.make = make;
            obj.model = model;
        end
        function drive(obj, milesDriven)
            obj.mileage = obj.mileage + milesDriven;
        end
    end
end
```

Обратите внимание, что конструктор - это метод с тем же именем, что и класс. <Конструктор - это специальный метод класса или структуры в объектно-ориентированном программировании, который инициализирует объект этого типа. Конструктор - это метод экземпляра, который обычно имеет то же имя, что и класс, и может использоваться для установки значений элементов объекта либо по умолчанию, либо по определенным пользователем значениям.>

Экземпляр этого класса может быть создан путем вызова конструктора;

```
>> myCar = Car('Ford', 'Mustang'); //creating an instance of car class
```

Вызов метода `drive` увеличит пробег

```
>> myCar.mileage

    ans =
             0

>> myCar.drive(450);

>> myCar.mileage

   ans =
             450
```

Классы Value vs Handle

Классы в MATLAB подразделяются на две основные категории: классы значений и классы дескрипторов. Основное различие заключается в том, что при копировании экземпляра класса значений базовые данные копируются в новый экземпляр, тогда как для классов дескрипторов новый экземпляр указывает на исходные данные и изменение значений в новом экземпляре меняет их в оригинале. Класс может быть определен как дескриптор путем наследования класса `handle` .

```
classdef valueClass
    properties
        data
    end
end
```

а также

```
classdef handleClass < handle
    properties
        data
    end
end
```

затем

```
>> v1 = valueClass;
>> v1.data = 5;
>> v2 = v1;
>> v2.data = 7;
>> v1.data
ans =
     5

>> h1 = handleClass;
>> h1.data = 5;
>> h2 = h1;
>> h2.data = 7;
>> h1.data
ans =
     7
```

Наследование классов и абстрактных классов

Отказ от ответственности: примеры, представленные здесь, предназначены только для того, чтобы показать использование абстрактных классов и наследования и, возможно, не обязательно иметь практическое применение. Кроме того, в MATLAB нет ничего, кроме полиморфного, и поэтому использование абстрактных классов ограничено. В этом примере показано, кто должен создавать класс, наследовать от другого класса и применять абстрактный класс для определения общего интерфейса.

Использование абстрактных классов довольно ограничено в MATLAB, но оно все же может пригодиться несколько раз.

Предположим, нам нужен регистратор сообщений. Мы могли бы создать класс, подобный приведенному ниже:

```
classdef ScreenLogger
    properties(Access=protected)
        scrh;
    end

    methods
        function obj = ScreenLogger(screenhandler)
            obj.scrh = screenhandler;
        end

        function LogMessage(obj, varargin)
            if ~isempty(varargin)
```

```
                varargin{1} = num2str(varargin{1});
                fprintf(obj.scrh, '%s\n', sprintf(varargin{:}));
            end
        end
    end
end
```

Свойства и методы

Короче говоря, свойства имеют состояние объекта, тогда как методы похожи на интерфейс и определяют действия над объектами.

Свойство `scrh` защищено. Вот почему он должен быть инициализирован в конструкторе. Существуют другие методы (getters) для доступа к этому свойству, но это не соответствует этому примеру. Свойства и методы могут быть доступны через переменную, содержащую ссылку на объект, с использованием точечной нотации, за которой следует имя метода или свойства:

```
mylogger = ScreenLogger(1);                          % OK
mylogger.LogMessage('My %s %d message', 'very', 1); % OK
mylogger.scrh = 2;                                   % ERROR!!! Access denied
```

Свойства и методы могут быть общедоступными, частными или защищенными. В этом случае protected означает, что я получаю доступ к `scrh` из унаследованного класса, но не извне. По умолчанию все свойства и методы являются общедоступными. Поэтому `LogMessage()` может свободно использоваться вне определения класса. Также `LogMessage` определяет интерфейс, означающий, что это то, что мы должны вызывать, когда мы хотим, чтобы объект регистрировал наши пользовательские сообщения.

заявка

Предположим, у меня есть сценарий, в котором я использую свой регистратор:

```
clc;
% ... a code
logger = ScreenLogger(1);
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
```

Если у меня есть несколько мест, где я использую один и тот же журнал, а затем хочу изменить его на нечто более сложное, например, написать сообщение в файле, мне придется создать другой объект:

```
classdef DeepLogger
```

```
    properties(SetAccess=protected)
        FileName
    end
    methods
        function obj = DeepLogger(filename)
            obj.FileName = filename;
        end

        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

и просто измените одну строку кода на это:

```
clc;
% ... a code
logger = DeepLogger('mymessages.log');
```

Вышеупомянутый метод просто откроет файл, добавит сообщение в конец файла и закроет его. На данный момент, чтобы быть в согласии с моим интерфейсом, мне нужно помнить, что имя метода - `LogMessage()` но в равной степени это может быть что угодно. MATLAB может заставить разработчиков придерживаться одного и того же имени, используя абстрактные классы. Предположим, мы определяем общий интерфейс для любого регистратора:

```
classdef MessageLogger
    methods(Abstract=true)
        LogMessage(obj, varargin);
    end
end
```

Теперь, если оба `ScreenLogger` и `DeepLogger` наследуются от этого класса, MATLAB будет генерировать ошибку, если `LogMessage()` не определен. Абстрактные классы помогают создавать похожие классы, которые могут использовать один и тот же интерфейс.

Ради этого примера я сделаю несколько другое изменение. Я собираюсь предположить, что DeepLogger будет делать как сообщение регистрации на экране, так и в файле одновременно. Поскольку `ScreenLogger` уже регистрирует сообщения на экране, я собираюсь наследовать `DeepLogger` из `ScreenLoggger` чтобы избежать повторения. `ScreenLogger` не изменяется вообще, кроме первой строки:

```
classdef ScreenLogger < MessageLogger
// the rest of previous code
```

Тем не менее, `DeepLogger` нуждается в дополнительных изменениях в методе `LogMessage` :

```
classdef DeepLogger < MessageLogger & ScreenLogger
    properties(SetAccess=protected)
        FileName
        Path
    end
    methods
        function obj = DeepLogger(screenhandler, filename)
            [path,filen,ext] = fileparts(filename);
            obj.FileName = [filen ext];
            pbj.Path     = pathn;
            obj = obj@ScreenLogger(screenhandler);
        end
        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                LogMessage@ScreenLogger(obj, varargin{:});
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

Во-первых, я просто инициализирую свойства в конструкторе. Во-вторых, поскольку этот класс наследуется от `ScreenLogger` мне также нужно инициализировать этот объект parrent. Эта строка еще важнее, потому что конструктор `ScreenLogger` требует одного параметра для инициализации собственного объекта. Эта строка:

```
obj = obj@ScreenLogger(screenhandler);
```

просто говорит: «Назовите конструктора ScreenLogger и активируйте его с помощью обработчика экрана». Здесь стоит отметить, что я определил `scrh` как защищенный. Поэтому я мог бы получить доступ к этому свойству от `DeepLogger` . Если свойство было определено как личное. Единственный способ его инициализации - использовать консуктора.

Другое изменение - в `methods` раздела. Опять же, чтобы избежать повторения, я вызываю `LogMessage()` из родительского класса для регистрации сообщения на экране. Если бы мне пришлось что-то менять, чтобы улучшить внесение изменений в экранный журнал, теперь я должен сделать это в одном месте. Код `DeepLogger` такой же, как и часть `DeepLogger` .

Поскольку этот класс также наследуется от абстрактного класса `MessageLogger` я должен был убедиться, что `LogMessage()` внутри `DeepLogger` также определен. Наследовать от `MessageLogger` здесь немного сложно. Я думаю, что случаи переопределения `LogMessage` обязательны - думаю.

Что касается кода, в котором применяется регистратор, благодаря общему интерфейсу в классах, я могу быть уверенным, что это изменение в одной строке во всем коде не

вызовет никаких проблем. Те же сообщения будут отображаться на экране, как и раньше, но дополнительно код будет записывать такие сообщения в файл.

```
clc;
% ... a code
logger = DeepLogger(1, 'mylogfile.log');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
```

Надеюсь, что эти примеры объясняют использование классов, использование наследования и использование абстрактных классов.

PS. Решение этой проблемы является одним из многих. Другим решением, менее сложным, было бы заставить `ScreenLoger` быть компонентом другого регистратора, такого как `FileLogger` и т. Д. `ScreenLogger` будет храниться в одном из свойств. Его `LogMessage` просто вызовет `LogMessage ScreenLogger` и покажет текст на экране. Я выбрал более сложный подход, чтобы показать, как классы работают в MATLAB. Пример кода ниже:

```
classdef DeepLogger < MessageLogger
    properties(SetAccess=protected)
        FileName
        Path
        ScrLogger
    end
    methods
        function obj = DeepLogger(screenhandler, filename)
            [path,filen,ext] = fileparts(filename);
            obj.FileName     = [filen ext];
            obj.Path         = pathn;
            obj.ScrLogger    = ScreenLogger(screenhandler);
        end
        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                obj.LogMessage(obj.ScrLogger, varargin{:}); % <-------- thechange here
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

Конструкторы

Конструктор - это особый метод в классе, который вызывается при создании экземпляра объекта. Это обычная функция MATLAB, которая принимает входные параметры, но также

должна следовать определенным правилам .

Конструкторы не требуются, так как MATLAB создает по умолчанию. На практике, однако, это место для определения состояния объекта. Например, свойства могут быть ограничены указанием атрибутов . Затем конструктор может инициализировать такие свойства по умолчанию или пользовательские значения, которые на самом деле могут быть отправлены входными параметрами конструктора.

Вызов конструктора простого класса

Это простой класс `Person` .

```
classdef Person
    properties
        name
        surname
        address
    end

    methods
        function obj = Person(name,surname,address)
            obj.name = name;
            obj.surname = surname;
            obj.address = address;
        end
    end
end
```

Имя конструктора совпадает с именем класса. Следовательно, конструкторы вызываются именем своего класса. Класс `Person` может быть создан следующим образом :

```
>> p = Person('John','Smith','London')
p =
  Person with properties:

       name: 'John'
    surname: 'Smith'
    address: 'London'
```

Вызов конструктора дочернего класса

Классы могут быть унаследованы от родительских классов, если общие свойства или методы общего доступа. Когда класс унаследован от другого, вполне вероятно, что должен быть вызван конструктор родительского класса.

`Member` класса наследуется от класса `Person` поскольку `Member` использует те же свойства, что и класс Person, но также добавляет `payment` к его определению.

```
classdef Member < Person
    properties
        payment
    end
```

```
    methods
        function obj = Member(name,surname,address,payment)
            obj = obj@Person(name,surname,address);
            obj.payment = payment;
        end
    end
end
```

Аналогично классу `Person` , `Member` создается путем вызова его конструктора:

```
>> m = Member('Adam','Woodcock','Manchester',20)
m =
  Member with properties:

    payment: 20
       name: 'Adam'
    surname: 'Woodcock'
    address: 'Manchester'
```

Для конструктора `Person` требуются три входных параметра. `Member` должен уважать этот факт и поэтому вызывать конструктор класса `Person` с тремя параметрами. Это выполняется по строке:

```
obj = obj@Person(name,surname,address);
```

В приведенном выше примере показан случай, когда дочерний класс нуждается в информации для своего родительского класса. Вот почему конструктор `Member` требует четыре параметра: три для его родительского класса и один для себя.

Прочитайте Объектно-ориентированное программирование онлайн:
https://riptutorial.com/ru/matlab/topic/1028/объектно-ориентированное-программирование

глава 20: отладка

Синтаксис

- dbstop в файле в месте, если выражение

параметры

параметр	подробности
файл	Имя файла `.m` (без расширения), например, `fit` . Этот параметр является *(обязательно),* если не `dbstop if error` специальные условные типы точек останова, такие как `dbstop if error` или `dbstop if naninf` .
место нахождения	Номер строки, где должна быть установлена точка останова. Если указанная строка не содержит исполняемый код, точка останова будет помещена в первую допустимую строку **после** указанной.
выражение	Любое выражение или комбинация, которая вычисляется по логическому значению. Примеры: `ind == 1` , `nargin < 4 && isdir('Q:\')` .

Examples

Работа с точками прерывания

Определение

> *При разработке программного обеспечения **точка останова** является преднамеренной остановкой или приостановкой в программе, созданной для целей отладки.*
>
> *В более общем плане точка останова является средством получения знаний о программе во время ее выполнения. Во время перерыва программист проверяет тестовую среду (регистры общего назначения, память, журналы, файлы и т. Д.), Чтобы выяснить, работает ли программа, как ожидалось. На практике точка останова состоит из одного или нескольких условий, которые определяют, когда выполнение программы должно быть прервано.*
>
> -Wikipedia

Точки останова в MATLAB

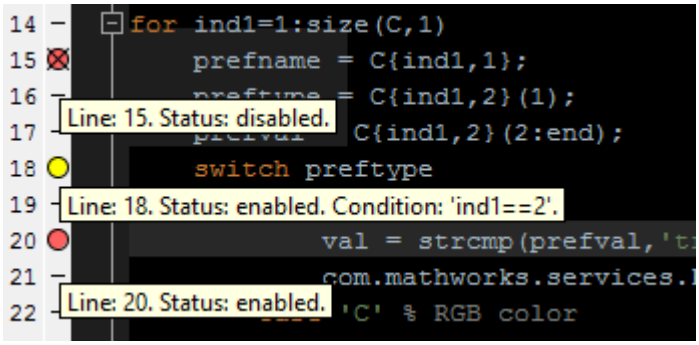
мотивация

В MATLAB, когда выполнение приостанавливается в точке останова, переменные, существующие в текущей рабочей области (aka *scope*) или любое из рабочих областей вызова, могут быть проверены (и обычно также изменены).

Типы точек останова

MATLAB позволяет пользователям размещать два типа точек останова в файлах `.m` :

- Стандартные (или « *неограниченные* ») точки останова (показаны красным цветом) - приостанавливать выполнение всякий раз, когда отмеченная линия достигнута.
- «Условные» точки останова (показаны желтым цветом) - приостанавливать выполнение всякий раз, когда отмеченная линия достигнута, и условие, определенное в точке останова, оценивается как `true` .



Размещение точек останова

Оба типа точек останова могут быть созданы несколькими способами:

- Используя графический интерфейс редактора MATLAB, щелкните правой кнопкой мыши по горизонтальной линии рядом с номером строки.

- Используя команду `dbstop` :

```
% Create an unrestricted breakpoint:
dbstop in file at location
% Create a conditional breakpoint:
dbstop in file at location if expression

% Examples and special cases:
dbstop in fit at 99 % Standard unrestricted breakpoint.

dbstop in fit at 99 if nargin==3 % Standard conditional breakpoint.
```

```
dbstop if error % This special type of breakpoint is not limited to a specific file, and
                % will trigger *whenever* an error is encountered in "debuggable" code.

dbstop in file % This will create an unrestricted breakpoint on the first executable line
               % of "file".

dbstop if naninf % This special breakpoint will trigger whenever a computation result
                 % contains either a NaN (indicates a division by 0) or an Inf
```

- Использование сочетаний клавиш: ключ по умолчанию для создания стандартной точки останова в Windows - `F12` ; ключ по умолчанию для условных точек останова не *установлен* .

Отключение и повторное включение точек останова

Отключите точку останова, чтобы временно ее игнорировать: отключенные точки останова не приостанавливают выполнение. Отключение точки останова может быть выполнено несколькими способами:

- Щелкните правой кнопкой мыши по красному / желтому кругу точек останова> Отключить точку останова.
- Щелкните левой кнопкой мыши на условной (желтой) точке останова.
- На вкладке «Редактор»> «Точки останова»> «Включить \ Отключить».

Удаление точек останова

Все точки останова остаются в файле до удаления, либо вручную, либо автоматически. Точки останова *автоматически* очищаются при завершении сеанса MATLAB (т.е. завершение программы). Очистка контрольных точек вручную выполняется одним из следующих способов:

- Использование команды `dbclear` :

```
dbclear all
dbclear in file
dbclear in file at location
dbclear if condition
```

- Щелкните левой кнопкой мыши значок стандартной точки останова или значок условного прерывания.
- Щелкните правой кнопкой мыши по любой точке останова> Clear Breakpoint.
- На вкладке «Редактор»> «Точки останова»> «Очистить все».
- В версиях MATLAB до R2015b, используя команду `clear` .

Возобновление исполнения

Когда выполнение приостановлено в точке останова, есть два способа продолжить выполнение программы:

- Выполните текущую строку и повторите паузу перед следующей строкой.

  `F10` [1] в редакторе, `dbstep` в окне команд, «Шаг» в ленте> Редактор> DEBUG.

- Выполните до следующей точки останова (если больше не осталось контрольных точек, выполнение продолжается до конца программы).

  `F12` [1] в редакторе, `dbcont` в окне команд, «Продолжить» в ленте> Редактор> DEBUG.

---

[1] - по умолчанию в Windows.

Отладка Java-кода, вызванного MATLAB

обзор

Чтобы отлаживать классы Java, вызываемые во время выполнения MATLAB, необходимо выполнить два шага:

1. Запустите MATLAB в режиме отладки JVM.
2. Присоедините отладчик Java к процессу MATLAB.

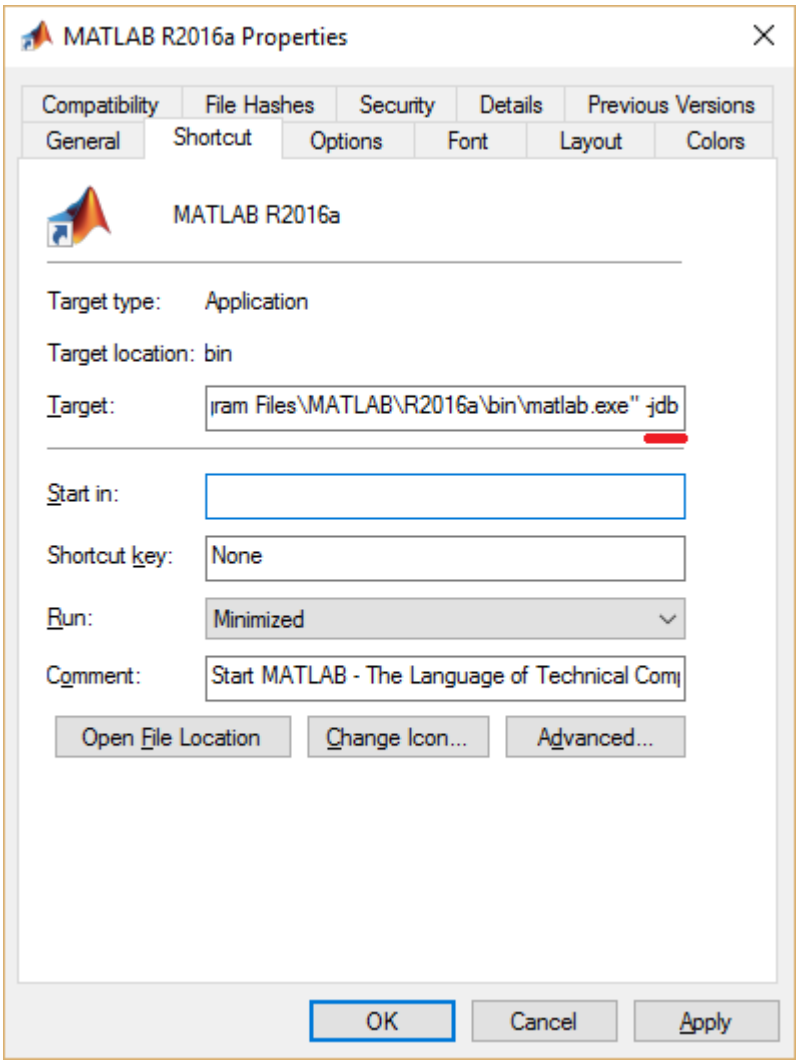
Когда MATLAB запускается в режиме отладки JVM, в окне команд появляется следующее сообщение:

```
JVM is being started with debugging enabled.
Use "jdb -connect com.sun.jdi.SocketAttach:port=4444" to attach debugger.
```

Конец MATLAB

Окна:

Создайте ярлык для исполняемого файла MATLAB (`matlab.exe`) и добавьте флаг `-jdb` в конец, как показано ниже:



При запуске MATLAB с использованием этого ярлыка будет включена отладка JVM.

Кроме того, файл `java.opts` может быть создан / обновлен. Этот файл хранится в «matlab-root \ bin \ arch», где «matlab-root» - это директива установки MATLAB, а «arch» - это архитектура (например, «win32»).

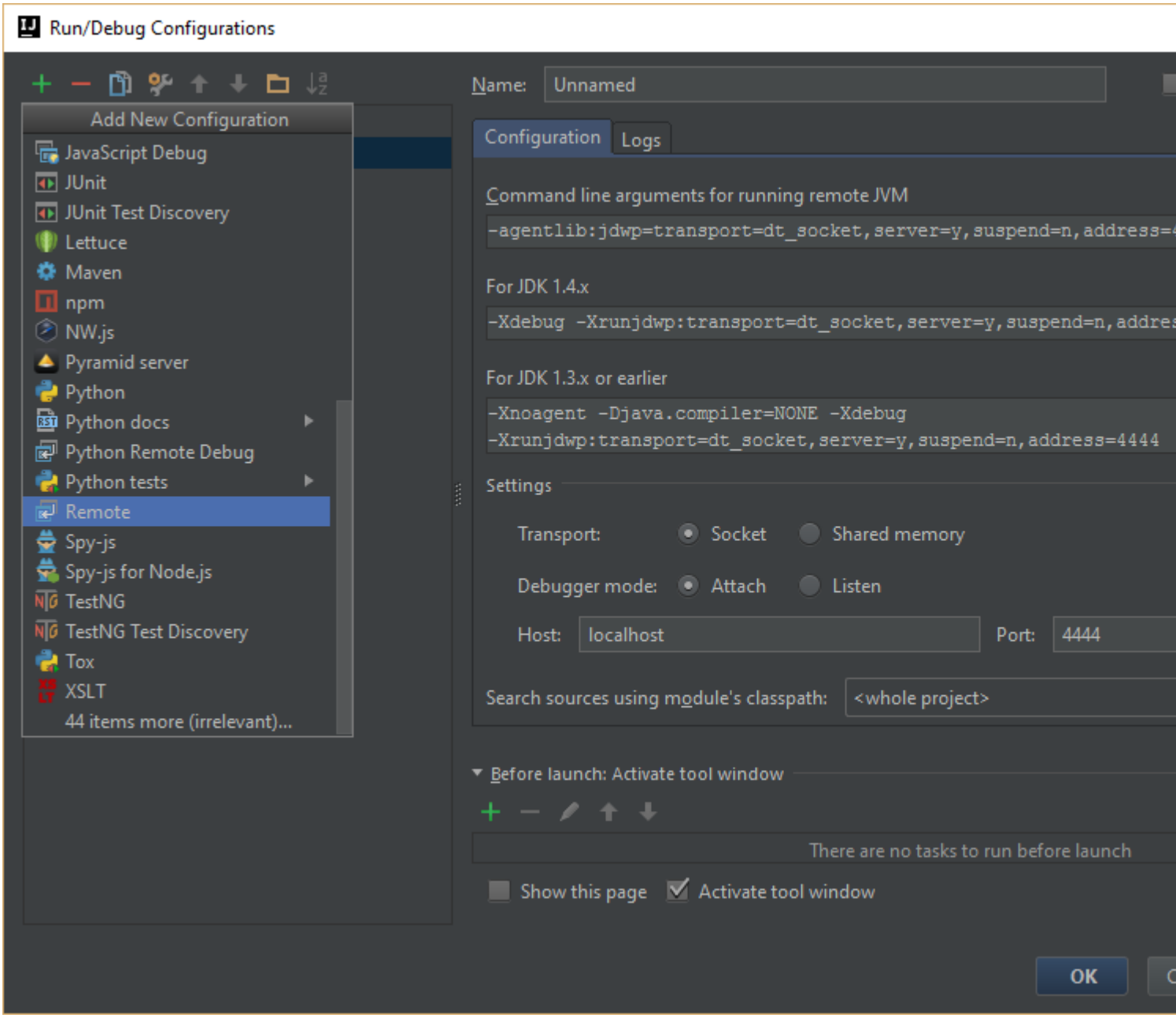
В файл необходимо добавить следующее:

```
-Xdebug
-Xrunjdwp:transport=dt_socket,address=1044,server=y,suspend=n
```

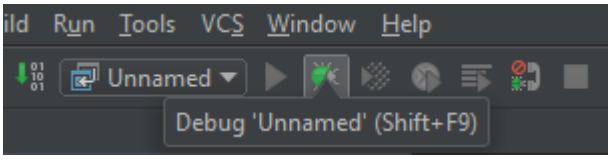
Окончание отладчика

IntelliJ IDEA

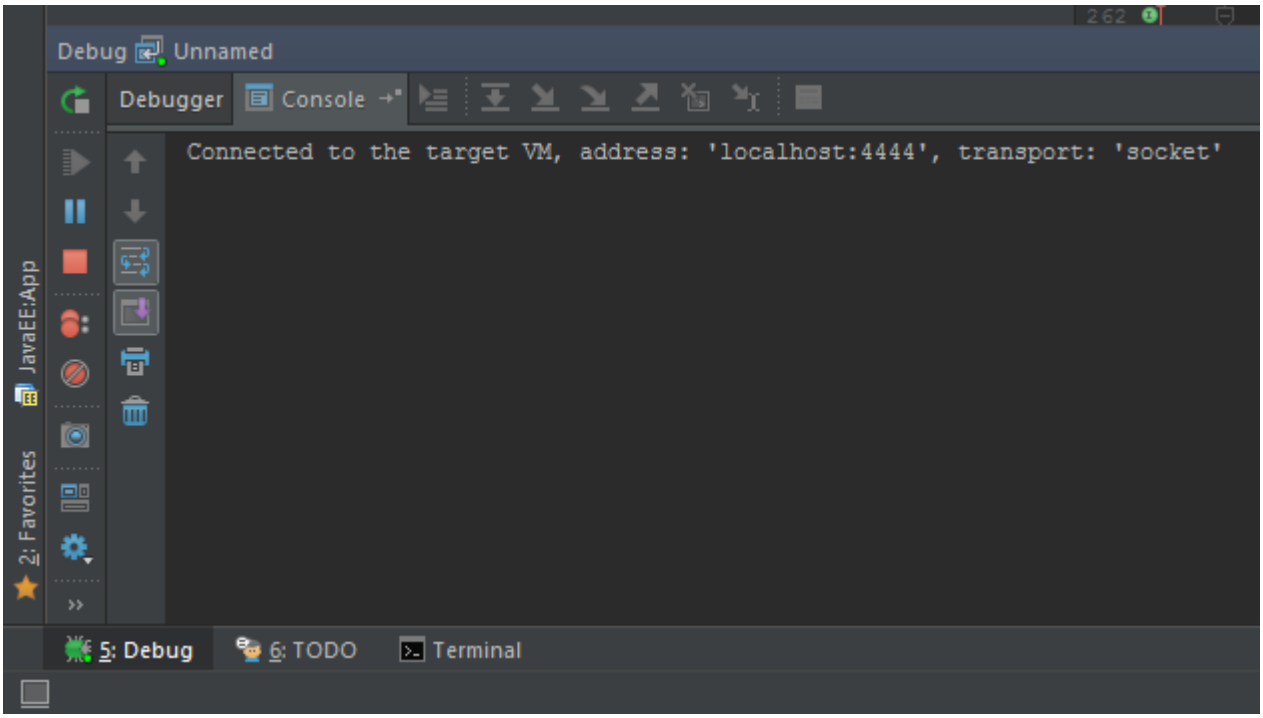
Прикрепление этого отладчика требует создания конфигурации «удаленной отладки» с портом, открытым MATLAB:



Затем запускается отладчик:



Если все работает должным образом, в консоли появится следующее сообщение:



Прочитайте отладка онлайн: https://riptutorial.com/ru/matlab/topic/1045/отладка

глава 21: Полезные трюки

Examples

Полезные функции, которые работают с ячейками и массивами

Этот простой пример дает объяснение некоторых функций, которые я нашел чрезвычайно полезными, так как я начал использовать MATLAB: `cellfun` , `arrayfun` . Идея состоит в том, чтобы взять переменную класса массива или ячейки, перебрать все ее элементы и применить выделенную функцию для каждого элемента. Приложенная функция может быть анонимной, что обычно является случаем, или любая регулярная функция определяется в файле * .m.

Начнем с простой проблемы и скажем, что нам нужно найти список файлов * .mat, заданных в этой папке. В этом примере сначала создадим некоторые файлы * .mat в текущей папке:

```
for n=1:10; save(sprintf('mymatfile%d.mat',n)); end
```

После выполнения кода должно быть 10 новых файлов с расширением * .mat. Если мы запустим команду для перечисления всех файлов * .mat, например:

```
mydir = dir('*.mat');
```

мы должны получить массив элементов структуры dir; MATLAB должен дать аналогичный результат:

```
10x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Как вы можете видеть, каждый элемент этого массива представляет собой структуру с несколькими полями. Вся информация действительно важна для каждого файла, но в 99% меня больше интересуют имена файлов и ничего больше. Чтобы извлечь информацию из массива структуры, я использовал для создания локальной функции, которая включала бы создание временных переменных правильного размера, для циклов, извлечения имени из каждого элемента и сохранения его в созданной переменной. Более простой способ добиться точно такого же результата - использовать одну из вышеупомянутых функций:

```
mydirlist = arrayfun(@(x) x.name, dir('*.mat'), 'UniformOutput', false)
mydirlist =
```

```
    'mymatfile1.mat'
    'mymatfile10.mat'
    'mymatfile2.mat'
    'mymatfile3.mat'
    'mymatfile4.mat'
    'mymatfile5.mat'
    'mymatfile6.mat'
    'mymatfile7.mat'
    'mymatfile8.mat'
    'mymatfile9.mat'
```

Как эта функция работает? Обычно это два параметра: дескриптор функции как первый параметр и массив. Затем функция будет работать с каждым элементом данного массива. Третий и четвертый параметры являются необязательными, но важными. Если мы знаем, что выход не будет регулярным, он должен быть сохранен в ячейке. Это должно указывать на значение `false` для `UniformOutput` . По умолчанию эта функция пытается вернуть регулярный вывод, такой как вектор чисел. Например, давайте выберем информацию о том, сколько места на диске занимает каждый файл в байтах:

```
mydirbytes = arrayfun(@(x) x.bytes, dir('*.mat'))
mydirbytes =
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
```

или килобайт:

```
mydirbytes = arrayfun(@(x) x.bytes/1024, dir('*.mat'))
mydirbytes =
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
```

На этот раз выход является регулярным вектором double. `UniformOutput` было установлено значение `true` .

`cellfun` - аналогичная функция. Разница между этой функцией и `arrayfun` заключается в том, что `cellfun` работает с переменными класса ячеек. Если мы хотим извлечь только имена с указанием списка имен файлов в ячейке «mydirlist», нам просто нужно будет

запустить эту функцию следующим образом:

```
mydirnames = cellfun(@(x) x(1:end-4), mydirlist, 'UniformOutput', false)
mydirnames =
    'mymatfile1'
    'mymatfile10'
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

Опять же, поскольку выход не является регулярным вектором чисел, выход должен быть сохранен в переменной ячейки.

В приведенном ниже примере я объединяю две функции в одном и возвращаю только список имен файлов без расширения:

```
cellfun(@(x) x(1:end-4), arrayfun(@(x) x.name, dir('*.mat'), 'UniformOutput', false),
'UniformOutput', false)
ans =
    'mymatfile1'
    'mymatfile10'
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

Это безумие, но очень возможно, потому что `arrayfun` возвращает ячейку, которая является ожидаемым вводом `cellfun` ; примечание к этому состоит в том, что мы можем заставить любую из этих функций вернуть результаты в переменную ячейки, установив `UniformOutput` в false, явно. Мы всегда можем получать результаты в ячейке. Возможно, мы не сможем получить результаты в регулярном векторе.

Существует еще одна аналогичная функция, которая работает с полями: `structfun` . Я не особо нашел его полезным, как два других, но в некоторых ситуациях он будет сиять. Если, например, хотелось бы знать, какие поля являются числовыми или нечисловыми, следующий код может дать ответ:

```
structfun(@(x) ischar(x), mydir(1))
```

Первое и второе поля структуры dir имеют тип char. Следовательно, выход:

```
 1
```

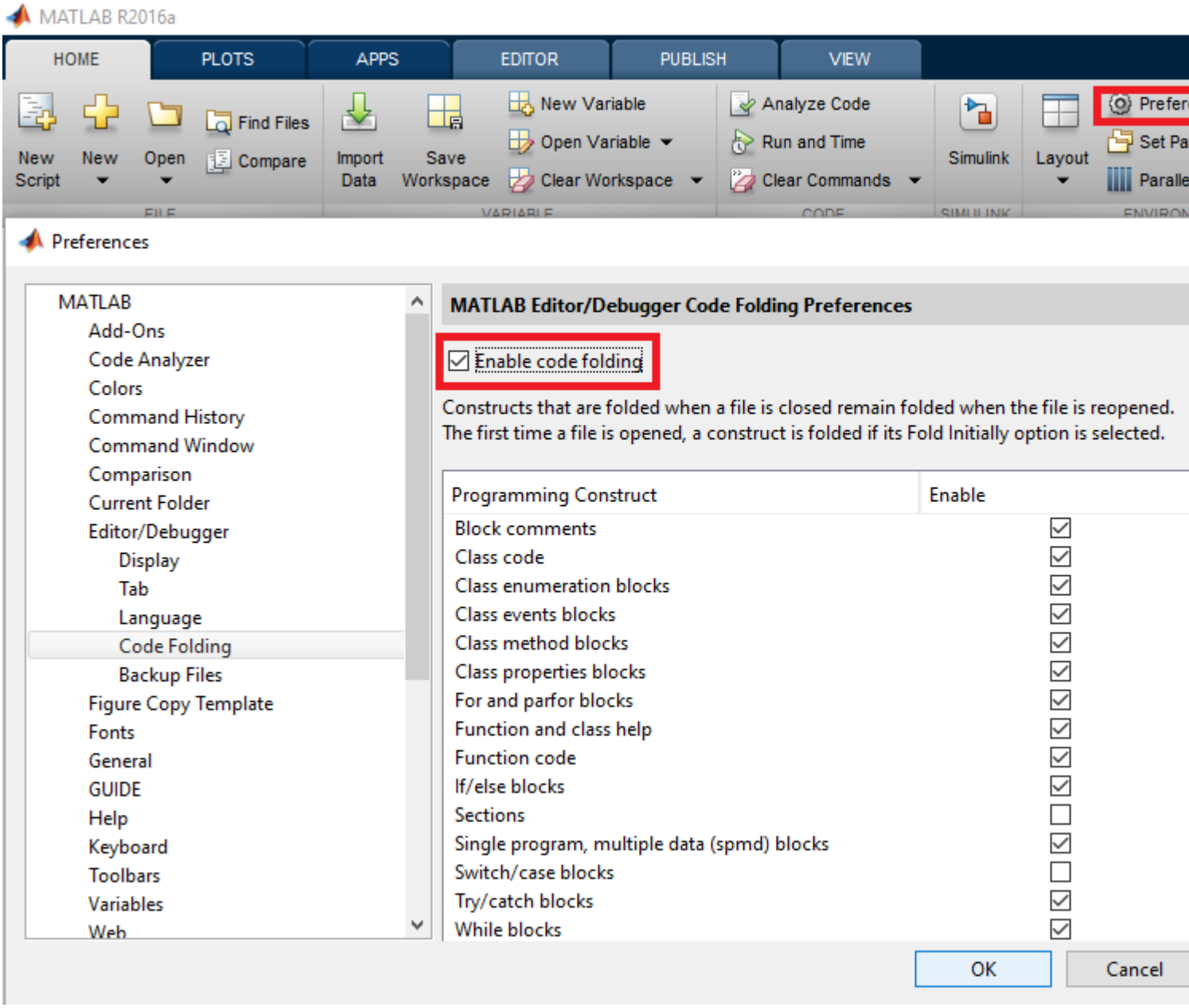
```
1
0
0
0
```

Кроме того, вывод является логическим вектором `true` / `false` . Следовательно, он является регулярным и может быть сохранен в векторе; нет необходимости использовать класс ячеек.

Настройки форматирования кода

Вы можете изменить предпочтение «Складка кода» в соответствии с вашими потребностями. Таким образом, сгибание кода можно установить enable / able для конкретных конструкций (например: `if block` , `for loop` , `Sections` ...).

Чтобы изменить параметры сгиба, перейдите в «Настройки» -> «Сгиб кода»:



Затем вы можете выбрать, какую часть кода можно сложить.

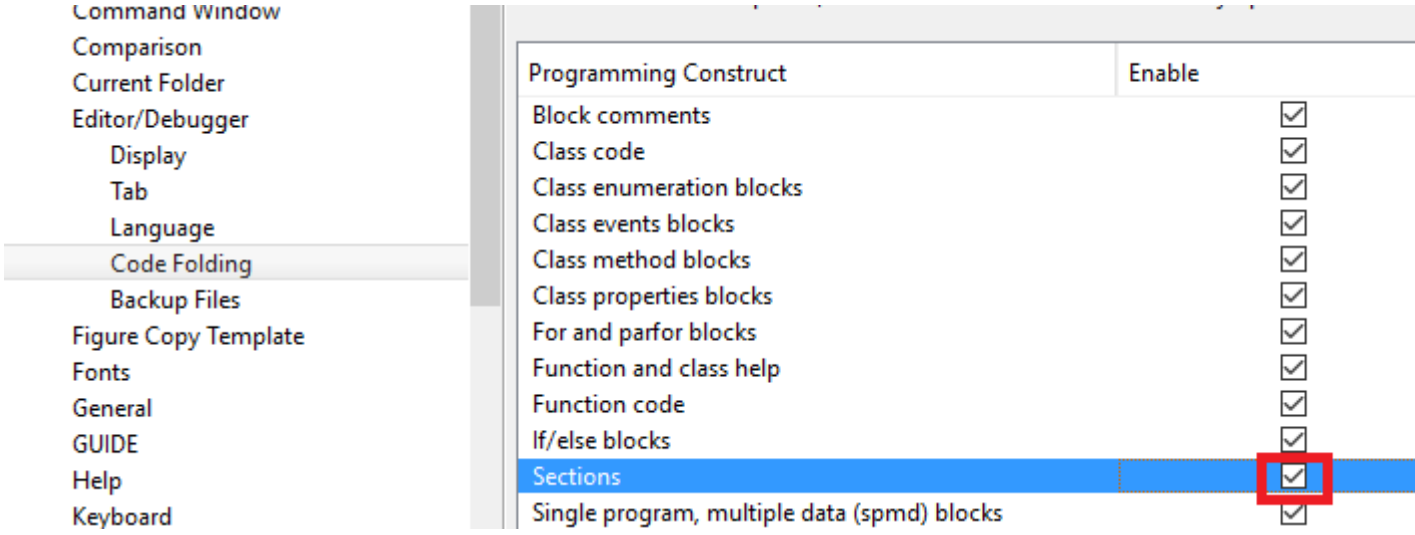
Некоторая информация:

- Обратите внимание, что вы также можете развернуть или свернуть весь код в файле, поместив курсор в любом месте файла, щелкнув правой кнопкой мыши, а затем выберите «Складка кода»> «Развернуть все» или «Сложить код»> «Сложить все» из контекстного меню.
- Обратите внимание, что сгибание является постоянным, в том смысле, что часть кода, который был расширен / свернут, сохранит свой статус после того, как Matlab или m-файл будут закрыты и снова открыты.

Пример. Чтобы включить фальцовку для разделов:

Интересным вариантом является возможность сброса разделов. Разделы разделены знаком двух процентов (`%%`).

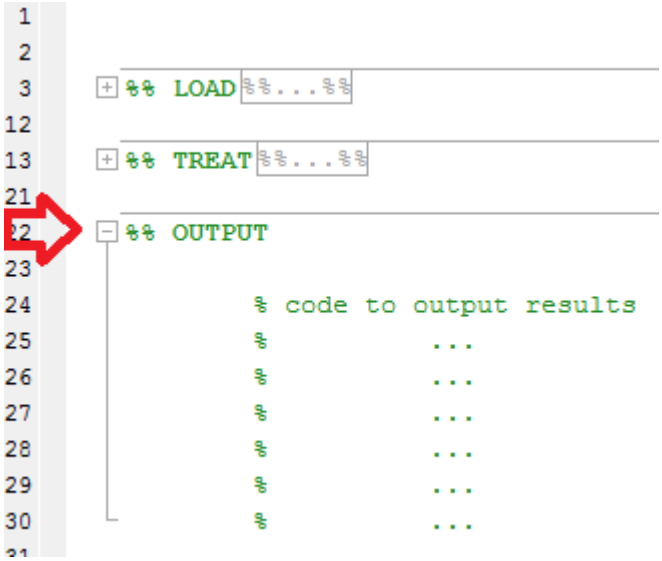
Пример. Чтобы включить его, установите флажок «Разделы»:



Затем вместо длинного исходного кода, похожего на:

```
%% LOAD

        %code to load data
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...

%% TREAT
        % code to run the model
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...

%% OUTPUT

        % code to output results
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
```

Вы сможете сворачивать разделы, чтобы иметь общий обзор вашего кода:



Извлечь фигурные данные

В нескольких случаях у меня была интересная фигура, которую я сохранил, но я потерял доступ к ее данным. В этом примере показано, как получить информацию об извлечении из фигуры.

Ключевыми функциями являются findobj и get . findobj возвращает обработчик объекту, которому присвоены атрибуты или свойства объекта, такие как `Type` или `Color` и т. д. После того, как объект линии найден, get может вернуть любое значение, удерживаемое свойствами. Оказывается, объекты `Line` сохраняют все данные в следующих свойствах: `XData` , `YData` и `ZData` ; последний обычно равен 0, если фигура не содержит 3D-график.

Следующий код создает примерный рисунок, который показывает две строки: функцию sin и порог и легенду

```
t = (0:1/10:1-1/10)';
y = sin(2*pi*t);
plot(t,y);
hold on;
plot([0 0.9],[0 0], 'k-');
hold off;
legend({'sin' 'threshold'});
```

Первое использование findobj возвращает два обработчика в обе строки:

```
findobj(gcf, 'Type', 'Line')
ans =
  2x1 Line array:

  Line    (threshold)
  Line    (sin)
```

Чтобы сузить результат, findobj также может использовать комбинацию логических операторов `-and` , `-or` и имена свойств. Например, я могу найти объект линии, чье имя `DiplayName` является `sin` и читать его `XData` и `YData` .

```
lineh = findobj(gcf, 'Type', 'Line', '-and', 'DisplayName', 'sin');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

и проверьте, равны ли данные.

```
isequal(t(:),xdata(:))
ans =
     1
isequal(y(:),ydata(:))
ans =
     1
```

Аналогично, я могу сузить результаты, исключив черную линию (порог):

```
lineh = findobj(gcf, 'Type', 'Line', '-not', 'Color', 'k');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

и последняя проверка подтверждает, что данные, извлеченные из этого рисунка,

совпадают:

```
isequal(t(:),xdata(:))
ans =
     1
isequal(y(:),ydata(:))
ans =
     1
```

Функциональное программирование с использованием анонимных функций

Анонимные функции могут использоваться для функционального программирования. Основная проблема, которую нужно решить, заключается в том, что для привязки рекурсии нет встроенного способа, но это может быть реализовано в одной строке:

```
if_ = @(bool, tf) tf{2-bool}();
```

Эта функция принимает логическое значение и массив ячеек из двух функций. Первая из этих функций оценивается, если логическое значение оценивается как true, а второе, если логическое значение оценивается как false. Теперь мы легко можем записать факториальную функцию:

```
fac = @(n,f) if_(n>1, {@()n*f(n-1,f), @()1});
```

Проблема здесь в том, что мы не можем напрямую вызывать рекурсивный вызов, так как функция еще не назначена переменной при оценке правой стороны. Однако мы можем завершить этот шаг, написав

```
factorial_ = @(n)fac(n,fac);
```

Теперь `@(n)fac(n,fac)` эвакуирует факториальную функцию рекурсивно. Другой способ сделать это в функциональном программировании с помощью y-combinator, который также может быть легко реализован:

```
y_ = @(f)@(n)f(n,f);
```

С помощью этого инструмента факториальная функция еще короче:

```
factorial_ = y_(fac);
```

Или напрямую:

```
factorial_ = y_(@(n,f) if_(n>1, {@()n*f(n-1,f), @()1}));
```

Сохранить несколько фигур в том же файле .fig

Поместив несколько графических фигур в графический массив, несколько фигур можно сохранить в том же файле .fig

```
h(1) = figure;
scatter(rand(1,100),rand(1,100));

h(2) = figure;
scatter(rand(1,100),rand(1,100));

h(3) = figure;
scatter(rand(1,100),rand(1,100));

savefig(h,'ThreeRandomScatterplots.fig');
close(h);
```

Это создает 3 диаграммы рассеяния случайных данных, каждая часть графического массива h. Затем графический массив можно сохранить с помощью savefig, как с обычной фигурой, но с дескриптором графического массива в качестве дополнительного аргумента.

Интересное замечание состоит в том, что цифры будут оставаться так же, как и при их открытии.

Блоки комментариев

Если вы хотите прокомментировать часть своего кода, тогда комментарии могут быть полезными. Блок комментариев начинается с `%{` в новой строке и заканчивается `%}` в другой новой строке:

```
a = 10;
b = 3;
%{
c = a*b;
d = a-b;
%}
```

Это позволяет вам складывать разделы, которые вы прокомментировали, чтобы сделать код более чистым и компактным.

Эти блоки также полезны для включения / выключения частей вашего кода. Все, что вам нужно сделать, чтобы раскомментировать блок, - это добавить еще `%` до того, как оно будет выглядеть:

```
a = 10;
b = 3;
%%{ <-- another % over here
c = a*b;
d = a-b;
```

```
%}
```

Иногда вы хотите прокомментировать раздел кода, но не затрагивая его отступ:

```
for k = 1:a
    b = b*k;
    c = c-b;
    d = d*c;
    disp(b)
end
```

Обычно, когда вы отмечаете блок кода и нажимаете Ctrl + r для его комментирования (тем самым добавляя `%` автоматически ко всем строкам, тогда, когда вы нажимаете Ctrl + i для автоматического отступа, блок кода перемещается из его правильной иерархической место и переместился слишком сильно вправо:

```
for k = 1:a
    b = b*k;
    %     c = c-b;
    %     d = d*c;
    disp(b)
end
```

Способ решить это - использовать блоки комментариев, поэтому внутренняя часть блока остается правильно отступом:

```
for k = 1:a
    b = b*k;
    %{
    c = c-b;
    d = d*c;
    %}
    disp(b)
end
```

Прочитайте Полезные трюки онлайн: https://riptutorial.com/ru/matlab/topic/4179/полезные-трюки

глава 22: Пользовательские интерфейсы MATLAB

Examples

Передача данных вокруг пользовательского интерфейса

Большинство современных пользовательских интерфейсов требуют, чтобы пользователь мог передавать информацию между различными функциями, составляющими пользовательский интерфейс. Для этого MATLAB предлагает несколько различных методов.

---

guidata

Собственная среда разработки GUI MATLAB (GUIDE) предпочитает использовать `struct` named `handles` для передачи данных между обратными вызовами. Эта `struct` содержит все графические дескрипторы для различных компонентов пользовательского интерфейса, а также данные, заданные пользователем. Если вы не используете созданный GUIDE обратный вызов, который автоматически передает `handles` , вы можете получить текущее значение с помощью `guidata`

```
% hObject is a graphics handle to any UI component in your GUI
handles = guidata(hObject);
```

Если вы хотите изменить значение, хранящееся в этой структуре данных, вы можете изменить, но затем вы должны сохранить его обратно в `hObject` чтобы изменения отображались другими обратными вызовами. Вы можете сохранить его, указав второй входной аргумент для `guidata` .

```
% Update the value
handles.myValue = 2;

% Save changes
guidata(hObject, handles)
```

Значение `hObject` не имеет значения до тех пор, пока оно является компонентом пользовательского интерфейса *на той же* `figure` потому что в конечном итоге данные сохраняются в фигуре, содержащей `hObject` .

Лучше всего:

- Хранение структуры `handles` , в которой вы можете хранить все ручки ваших

компонентов графического интерфейса.
- Хранение «малых» других переменных, к которым нужно обращаться большинством обратных вызовов.

Не рекомендуется для :

- Хранение больших переменных, к которым не нужно обращаться всеми обратными вызовами и `setappdata` (используйте для них `setappdata` / `getappdata`).

`setappdata` / `getappdata`

Подобно подходу `guidata` , вы можете использовать `setappdata` и `getappdata` для хранения и извлечения значений из графического дескриптора. Преимущество использования этих методов состоит в том, что вы можете получить *только* требуемое *значение,* а не целую `struct` содержащую *все* сохраненные данные. Он похож на хранилище ключей / значений.

Для хранения данных в графическом объекте

```
% Create some data you would like to store
myvalue = 2

% Store it using the key 'mykey'
setappdata(hObject, 'mykey', myvalue)
```

И для получения того же значения из другого обратного вызова

```
value = getappdata(hObject, 'mykey');
```

Примечание. Если до вызова `getappdata` не было сохранено `getappdata` , оно вернет пустой массив (`[]`).

Подобно `guidata` , данные хранятся на рисунке, который содержит `hObject` .

Лучше всего:

- Хранение больших переменных, к которым не нужно обращаться всеми обратными вызовами и подфункциями.

`UserData`

Каждый графический дескриптор имеет специальное свойство `UserData` которое может содержать любые данные. Он может содержать массив ячеек, `struct` или даже скаляр. Вы можете воспользоваться этим свойством и сохранить любые данные, которые вы хотите связать с данным графическим дескриптором в этом поле. Вы можете сохранить и

получить значение с помощью стандартных методов `get` / `set` для графических объектов или точечной нотации, если вы используете R2014b или новее.

```
% Create some data to store
mydata = {1, 2, 3};

% Store it within the UserData property
set(hObject, 'UserData', mydata)

% Of if you're using R2014b or newer:
% hObject.UserData = mydata;
```

Затем изнутри другого обратного вызова вы можете получить эти данные:

```
their_data = get(hObject, 'UserData');

% Or if you're using R2014b or newer:
% their_data = hObject.UserData;
```

Лучше всего:

- Хранение переменных с ограниченным объемом (переменные, которые могут использоваться только объектом, в котором они хранятся, или объектами, имеющими прямое отношение к нему).

Вложенные функции

В MATLAB вложенная функция может читать и изменять любую переменную, определенную в родительской функции. Таким образом, если вы укажете обратный вызов как вложенную функцию, он может извлекать и изменять любые данные, хранящиеся в основной функции.

```
function mygui()
    hButton = uicontrol('String', 'Click Me', 'Callback', @callback);

    % Create a counter to keep track of the number of times the button is clicked
    nClicks = 0;

    % Callback function is nested and can therefore read and modify nClicks
    function callback(source, event)
        % Increment the number of clicks
        nClicks = nClicks + 1;

        % Print the number of clicks so far
        fprintf('Number of clicks: %d\n', nClicks);
    end
end
```

Лучше всего:

- Маленькие, простые графические интерфейсы. (для быстрого прототипирования, чтобы не было необходимости применять `guidata` и / или `set/getappdata`).

Не рекомендуется для :

- Средний, большой или сложный графический интерфейс.
- GUI создан с помощью `GUIDE` .

Явные входные аргументы

Если вам нужно отправить данные в функцию обратного вызова и не нужно изменять данные в обратном вызове, вы всегда можете рассмотреть возможность передачи данных в обратный вызов с использованием тщательно разработанного определения обратного вызова.

Вы можете использовать анонимную функцию, которая добавляет входные данные

```
% Create some data to send to mycallback
data = [1, 2, 3];

% Pass data as a third input to mycallback
set(hObject, 'Callback', @(source, event)mycallback(source, event, data))
```

Или вы можете использовать синтаксис массива ячеек, чтобы указать обратный вызов, снова указав дополнительные входы.

```
set(hObject, 'Callback', {@mycallback, data})
```

Лучше всего:

- Когда обратный вызов требует `data` для выполнения некоторых операций, но переменную `data` не нужно изменять и сохранять в новом состоянии.

Создание кнопки в пользовательском интерфейсе, которая приостанавливает выполнение обратного вызова

Иногда мы хотели бы приостановить выполнение кода, чтобы проверить состояние приложения (см. Раздел «Отладка»). При запуске кода через редактор MATLAB это можно сделать, используя кнопку «Пауза» в пользовательском интерфейсе или нажав `Ctrl` + `c` (в Windows). Однако, когда вычисление было инициировано из графического интерфейса (через обратный вызов некоторого `uicontrol`), этот метод больше не работает, и обратный вызов должен быть *прерван* посредством другого обратного вызова. Ниже

приведена демонстрация этого принципа:

```
function interruptibleUI
dbclear in interruptibleUI % reset breakpoints in this file
figure('Position',[400,500,329,160]);

uicontrol('Style', 'pushbutton',...
          'String', 'Compute',...
          'Position', [24 55 131 63],...
          'Callback', @longComputation,...
          'Interruptible','on'); % 'on' by default anyway

uicontrol('Style', 'pushbutton',...
          'String', 'Pause #1',...
          'Position', [180 87 131 63],...
          'Callback', @interrupt1);

uicontrol('Style', 'pushbutton',...
          'String', 'Pause #2',...
          'Position', [180 12 131 63],...
          'Callback', @interrupt2);

end

function longComputation(src,event)
  superSecretVar = rand(1);
  pause(15);
  print('done!'); % we'll use this to determine if code kept running "in the background".
end

function interrupt1(src,event) % depending on where you want to stop
  dbstop in interruptibleUI at 27 % will stop after print('done!');
  dbstop in interruptibleUI at 32 % will stop after **this** line.
end

function interrupt2(src,event) % method 2
  keyboard;
  dbup; % this will need to be executed manually once the code stops on the previous line.
end
```

Чтобы убедиться, что вы понимаете этот пример, выполните следующие действия:

1. Вставьте вышеуказанный код в новый файл и сохраните его как `interruptibleUI.m` , чтобы код начинался в **самой первой строке** файла (это важно для работы первого метода).
2. Запустите скрипт.
3. Нажмите « `Вычислить`» и вскоре после этого нажмите « `Пауза №1`» или « `Пауза № 2`» .
4. Убедитесь, что вы можете найти значение `superSecretVar` .

Передача данных с использованием структуры «ручек»

Это пример базового графического интерфейса с двумя кнопками, которые изменяют значение, хранящееся в структуре `handles` GUI.

```
function gui_passing_data()
```

```
    % A basic GUI with two buttons to show a simple use of the 'handles'
    % structure in GUI building

    %  Create a new figure.
    f = figure();

    % Retrieve the handles structure
    handles = guidata(f);

    % Store the figure handle
    handles.figure = f;

    % Create an edit box and two buttons (plus and minus),
    % and store their handles for future use
    handles.hedit  = uicontrol('Style','edit','Position',[10,200,60,20] , 'Enable',
'Inactive');

    handles.hbutton_plus  = uicontrol('Style','pushbutton','String','+',...
               'Position',[80,200,60,20] , 'Callback' , @ButtonPress);

    handles.hbutton_minus  = uicontrol('Style','pushbutton','String','-',...
               'Position',[150,200,60,20] , 'Callback' , @ButtonPress);

    % Define an initial value, store it in the handles structure and show
    % it in the Edit box
    handles.value = 1;
    set(handles.hedit , 'String' , num2str(handles.value))

    % Store handles
    guidata(f, handles);



function  ButtonPress(hObject, eventdata)
    % A button was pressed
    % Retrieve the handles
    handles = guidata(hObject);

    % Determine which button was pressed; hObject is the calling object
    switch(get(hObject , 'String'))
        case '+'
            % Add 1 to the value
            handles.value = handles.value + 1;
            set(handles.hedit , 'String', num2str(handles.value))
        case '-'
            % Substract 1 from the value
            handles.value = handles.value - 1;
    end

    % Display the new value
    set(handles.hedit , 'String', num2str(handles.value))

    % Store handles
    guidata(hObject, handles);
```

Чтобы проверить пример, сохраните его в файле `gui_passing_data.m` и запустите его с помощью F5. Обратите внимание, что в таком простом случае вам даже не нужно будет сохранять значение в структуре дескрипторов, потому что вы можете напрямую обращаться к нему из свойства `String` окна редактирования.

Проблемы с производительностью при передаче данных вокруг пользовательского интерфейса

Два основных метода позволяют передавать данные между функциями GUI и обратными вызовами: setappdata / getappdata и guidata (подробнее об этом). Первый должен использоваться для больших переменных, поскольку он более эффективен по времени. В следующем примере проверяется эффективность двух методов.

Создается графический интерфейс с простой кнопкой, и большая переменная (10000 x10000 double) сохраняется как с guidata, так и с setappdata. Кнопка перезагружает и сохраняет обратно переменную, используя два метода, в то время как время их выполнения. Время выполнения и процентное улучшение с использованием setappdata отображаются в окне команд.

```
function gui_passing_data_performance()
    % A basic GUI with a button to show performance difference between
    % guidata and setappdata

    %  Create a new figure.
    f = figure('Units' , 'normalized');

    % Retrieve the handles structure
    handles = guidata(f);

    % Store the figure handle
    handles.figure = f;

    handles.hbutton =
uicontrol('Style','pushbutton','String','Calculate','units','normalized',...
                'Position',[0.4 , 0.45 , 0.2 , 0.1] , 'Callback' , @ButtonPress);

    % Create an uninteresting large array
    data = zeros(10000);

    % Store it in appdata
    setappdata(handles.figure , 'data' , data);

    % Store it in handles
    handles.data = data;

    % Save handles
    guidata(f, handles);



function  ButtonPress(hObject, eventdata)

    % Calculate the time difference when using guidata and appdata
    t_handles = timeit(@use_handles);
    t_appdata = timeit(@use_appdata);

    % Absolute and percentage difference
    t_diff = t_handles - t_appdata;
    t_perc = round(t_diff / t_handles * 100);
```

```
    disp(['Difference: ' num2str(t_diff) ' ms / ' num2str(t_perc) ' %'])



function  use_appdata()

    % Retrieve the data from appdata
    data = getappdata(gcf , 'data');

    % Do something with data %

    % Store the value again
    setappdata(gcf , 'data' , data);


function use_handles()

    % Retrieve the data from handles
    handles = guidata(gcf);
    data = handles.data;

    % Do something with data %

    % Store it back in the handles
    handles.data = data;
    guidata(gcf, handles);
```

На моем Xeon W3530@2.80 ГГц я получаю `Difference: 0.00018957 ms / 73 %` , поэтому используя getappdata / setappdata, я получаю повышение производительности на 73%! Обратите внимание, что результат не изменяется, если используется двойная переменная 10x10, однако результат будет изменяться, если в `handles` содержится много полей с большими данными.

Прочитайте Пользовательские интерфейсы MATLAB онлайн:
https://riptutorial.com/ru/matlab/topic/2883/пользовательские-интерфейсы-matlab

глава 23: Преобразования Фурье и обратные преобразования Фурье

Синтаксис

1. Y = fft (X)% вычисляет БПФ вектора или матрицы X, используя длину преобразования по умолчанию 256 (будет подтверждено для версии)
2. Y = fft (X, n)% вычисляет БПФ X, используя n в качестве длины преобразования, n должно быть числом на 2 степени. Если длина X меньше n, то Matlab автоматически вставит X с нулями, так что длина (X) = n
3. Y = fft (X, n, dim)% вычисляет БПФ X, используя n в качестве длины преобразования по размеру dim (может быть 1 или 2 для горизонтальной или вертикальной соответственно)
4. Y = fft2 (X)% Вычислить 2D БПФ X
5. Y = fftn (X, dim). Вычислить dim-мерный БПФ X относительно вектора размерностей dim.
6. y = ifft (X)% вычисляет Обратный БПФ X (который является матрицей / вектором чисел) с использованием 256 Transform Length
7. y = ifft (X, n)% вычисляет IFFT X, используя n в качестве длины преобразования
8. y = ifft (X, n, dim)% вычисляет IFFT X, используя n в качестве длины преобразования по размеру dim (может быть 1 или 2 для горизонтальной или вертикальной соответственно)
9. y = ifft (X, n, dim, 'симметричный).% Симметричный параметр заставляет ifft трактовать X как сопряженное симметричное по активной размерности. Эта опция полезна, когда X не является точно сопряженной симметричной, просто из-за ошибки округления.
10. y = ifft2 (X)% Вычислить обратный 2D ft X
11. y = ifftn (X, dim). Вычислить обратное dim-мерное fft пространства X.

параметры

параметр	Описание
Икс	это ваш входной сигнал Time-Domain, он должен быть вектором цифр.
N	это параметр NFFT, известный как Transform Length, подумайте об этом как о разрешении вашего результата FFT, он ДОЛЖЕН быть числом, которое имеет мощность 2 (т.е. 64,128,256 ... 2 ^ N)
тусклый	это размер, который вы хотите вычислить БПФ, используйте 1, если вы хотите вычислить свой БПФ в горизонтальном направлении и 2, если вы хотите вычислить свой БПФ в вертикальном направлении - Обратите внимание, что этот параметр обычно остается пустым, так как функция способный определять направление вашего вектора.

замечания

Matlab FFT - это очень параллельный процесс, способный обрабатывать большие объемы данных. Он также может использовать GPU для огромного преимущества.

```
ifft(fft(X)) = X
```

Вышеприведенное утверждение верно, если ошибки округления опущены.

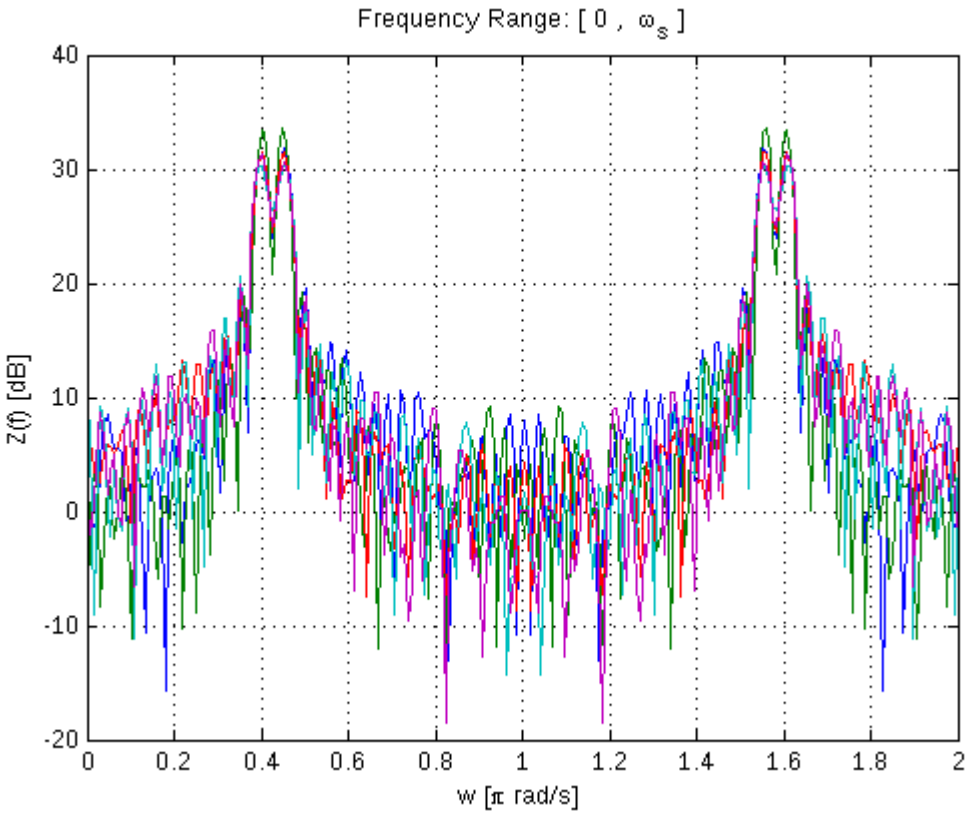
Examples

Внедрение простого преобразования Фурье в Matlab

Преобразование Фурье, вероятно, является первым уроком в цифровой обработке сигналов, это приложение повсеместно, и это мощный инструмент, когда дело доходит до анализа данных (во всех секторах) или сигналов. Matlab имеет набор мощных наборов инструментов для преобразования Фурье. В этом примере мы будем использовать преобразование Фурье для анализа основного синусоидального сигнала и генерировать то, что иногда называют периодограммой с использованием БПФ:

```
%Signal Generation
A1=10;                % Amplitude 1
A2=10;                % Amplitude 2
w1=2*pi*0.2;          % Angular frequency 1
w2=2*pi*0.225;        % Angular frequency 2
Ts=1;                 % Sampling time
N=64;                 % Number of process samples to be generated
K=5;                  % Number of independent process realizations
sgm=1;                % Standard deviation of the noise
n=repmat([0:N-1].',1,K);             % Generate resolution
phi1=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 1
phi2=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 2
x=A1*sin(w1*n*Ts+phi1)+A2*sin(w2*n*Ts+phi2)+sgm*randn(N,K);   % Resulting Signal
```

```
NFFT=256;             % FFT length
F=fft(x,NFFT);        % Fast Fourier Transform Result
Z=1/N*abs(F).^2;      % Convert FFT result into a Periodogram
```



Обратите внимание, что дискретное преобразование Фурье реализовано быстрым преобразованием Фурье (fft) в Matlab, оба будут давать тот же результат, но FFT - это быстрая реализация DFT.

```
figure
w=linspace(0,2,NFFT);
plot(w,10*log10(Z)),grid;
xlabel('w [\pi rad/s]')
ylabel('Z(f) [dB]')
title('Frequency Range: [ 0 ,  \omega_s ]')
```

Обратные преобразования Фурье

Одним из основных преимуществ преобразования Фурье является его способность обратно обратно в Time Domain без потери информации. Рассмотрим тот же Сигнал, который мы использовали в предыдущем примере:

```
A1=10;                % Amplitude 1
A2=10;                % Amplitude 2
w1=2*pi*0.2;          % Angular frequency 1
w2=2*pi*0.225;        % Angular frequency 2
Ts=1;                 % Sampling time
N=64;                 % Number of process samples to be generated
```

```
K=1;                  % Number of independent process realizations
sgm=1;                % Standard deviation of the noise
n=repmat([0:N-1].',1,K);             % Generate resolution
phi1=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 1
phi2=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 2
x=A1*sin(w1*n*Ts+phi1)+A2*sin(w2*n*Ts+phi2)+sgm*randn(N,K);   % Resulting Signal

NFFT=256;            % FFT length
F=fft(x,NFFT);       % FFT result of time domain signal
```

Если мы откроем `F` в Matlab, мы найдем, что это матрица комплексных чисел, вещественная часть и мнимая часть. По определению, чтобы восстановить исходный сигнал временной области, нам нужны как Real (который представляет вариацию Magnitude), так и Imaginary (который представляет изменение фазы), поэтому, чтобы вернуться во временную область, можно просто хотеть:

```
TD = ifft(F,NFFT);   %Returns the Inverse of F in Time Domain
```

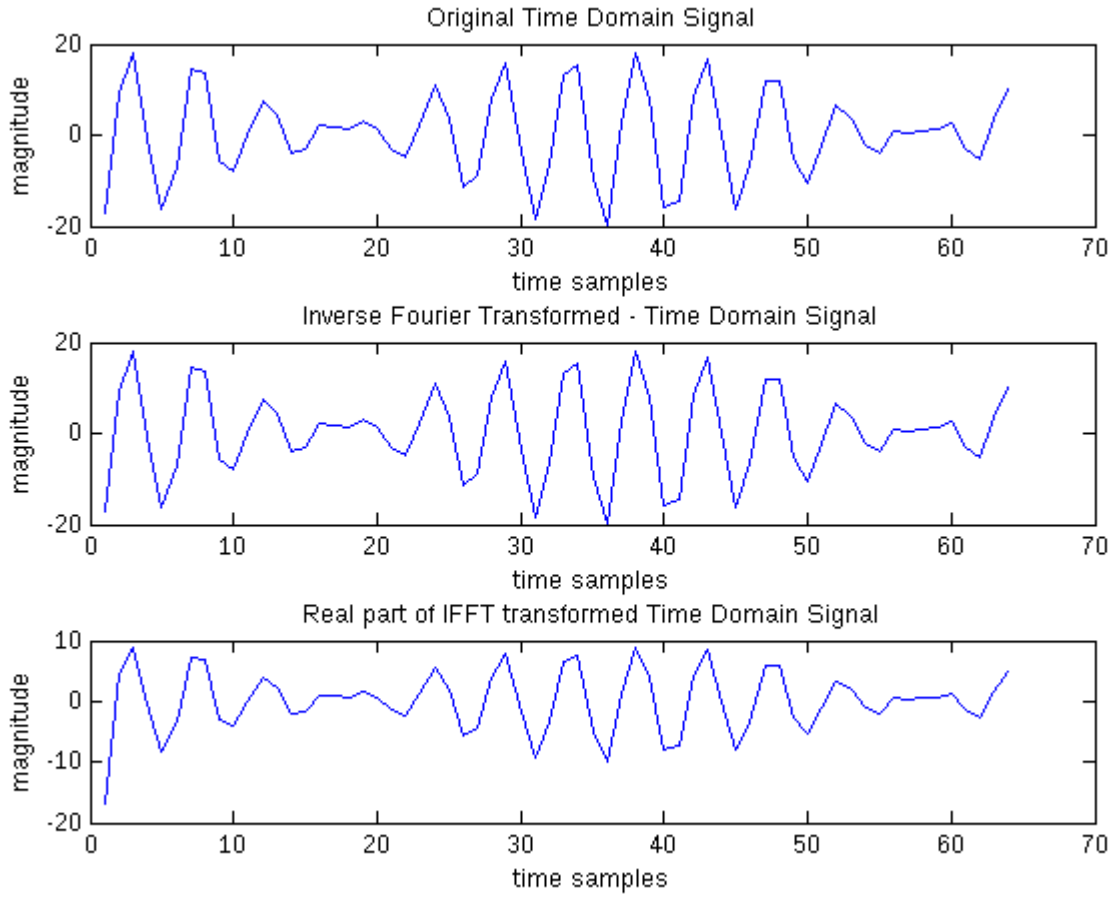
Обратите внимание, что возвращаемый TD будет длиной 256, поскольку мы устанавливаем NFFT на 256, однако длина x равна только 64, поэтому Matlab будет заполнять нули до конца TD-преобразования. Так, например, если NFFT равнялся 1024, а длина равнялась 64, то возвращаемое TD было равно 64 + 960 нулей. Также обратите внимание, что из-за округления с плавающей запятой вы можете получить что-то вроде 3.1 * 10e-20, но для общего назначения: для любого X ifft (fft (X)) равно X с точностью до округлой ошибки.

Скажем на мгновение, что после преобразования мы сделали что-то и остались только с РЕАЛЬНОЙ частью БПФ:

```
R = real(F);         %Give the Real Part of the FFT
TDR = ifft(R,NFFT);  %Give the Time Domain of the Real Part of the FFT
```

Это означает, что мы теряем мнимую часть нашего БПФ, и поэтому мы теряем информацию в этом обратном процессе. Чтобы сохранить оригинал без потери информации, вы всегда должны сохранять мнимую часть БПФ с помощью `imag` и применять свои функции как к обеим, так и к реальной части.

```
figure
subplot(3,1,1)
plot(x);xlabel('time samples');ylabel('magnitude');title('Original Time Domain Signal')
subplot(3,1,2)
plot(TD(1:64));xlabel('time samples');ylabel('magnitude');title('Inverse Fourier Transformed -
Time Domain Signal')
subplot(3,1,3)
plot(TDR(1:64));xlabel('time samples');ylabel('magnitude');title('Real part of IFFT
transformed Time Domain Signal')
```



Изображения и многомерные ФТ

В медицинской визуализации, спектроскопии, обработке изображений, криптографии и других областях науки и техники часто бывает так, что хочется вычислить многомерные преобразования Фурье изображений. Это довольно просто в Matlab: (многомерные) образы - это просто n-мерные матрицы, в конце концов, и преобразования Фурье являются линейными операторами: одно просто итеративно преобразование Фурье по другим измерениям. Matlab обеспечивает `fft2` и `ifft2` чтобы сделать это в 2-d, или `fftn` в n-измерениях.

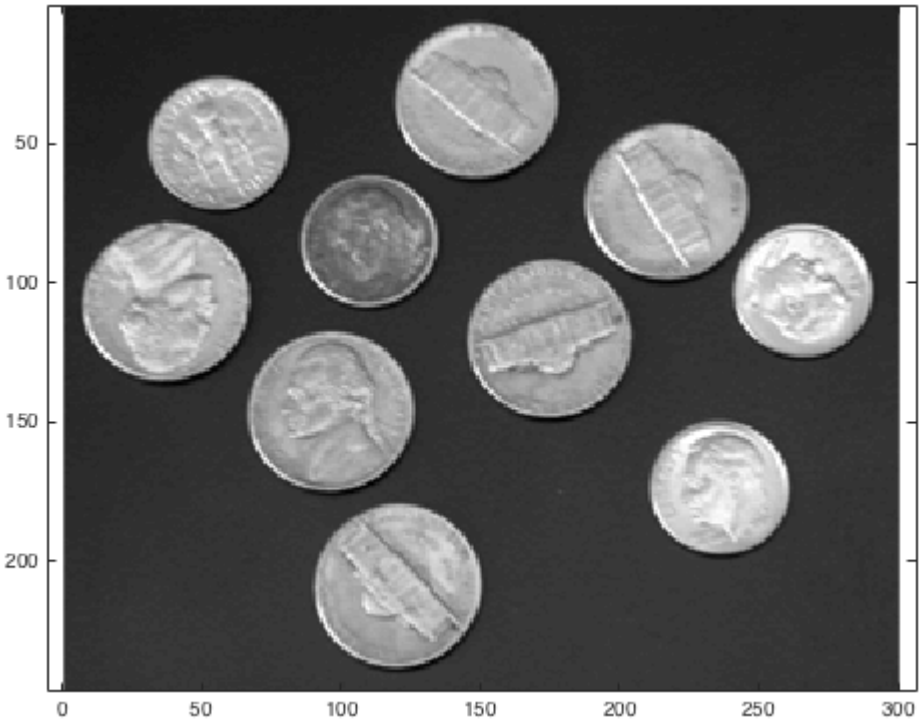
Одна потенциальная ошибка заключается в том, что преобразование Фурье изображений обычно показано «ориентированным по центру», т. Е. С началом k-пространства в середине изображения. Matlab предоставляет команду `fftshift` для правильной замены местоположения DC-компонентов преобразования Фурье. Это упорядочивающее обозначение значительно упрощает выполнение общих методов обработки изображений, один из которых проиллюстрирован ниже.

Нулевое заполнение

Один «быстрый и грязный» способ интерполировать небольшое изображение на больший

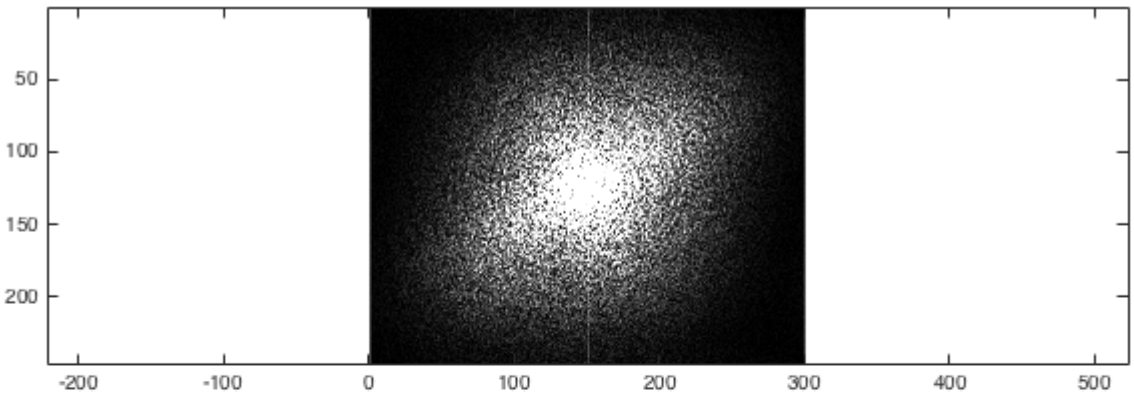
размер - преобразовать его в Фурье, наложить преобразование Фурье на нули и затем принять обратное преобразование. Это эффективно интерполирует между каждым пикселем с основанной на сердцевиной базовой функцией и обычно используется для масштабирования медицинских изображений с низким разрешением. Начнем с загрузки примера встроенного изображения.

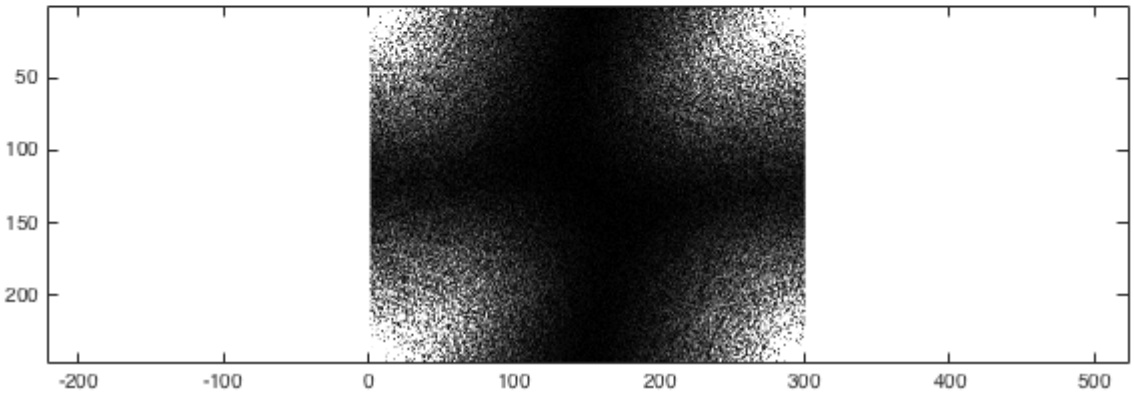
```
%Load example image
I=imread('coins.png'); %Load example data -- coins.png is builtin to Matlab
I=double(I); %Convert to double precision -- imread returns integers
imageSize = size(I); % I is a 246 x 300 2D image

%Display it
imagesc(I); colormap gray; axis equal;
%imagesc displays images scaled to maximum intensity
```



Теперь мы можем получить преобразование Фурье I. Чтобы проиллюстрировать, что делает `fftshift` , давайте сравним два метода:

```
% Fourier transform
%Obtain the centric- and non-centric ordered Fourier transform of I
k=fftshift(fft2(fftshift(I)));
kwrong=fft2(I);

%Just for the sake of comparison, show the magnitude of both transforms:
figure; subplot(2,1,1);
imagesc(abs(k),[0 1e4]); colormap gray; axis equal;
subplot(2,1,2);
imagesc(abs(kwrong),[0 1e4]); colormap gray; axis equal;
%(The second argument to imagesc sets the colour axis to make the difference clear).
```

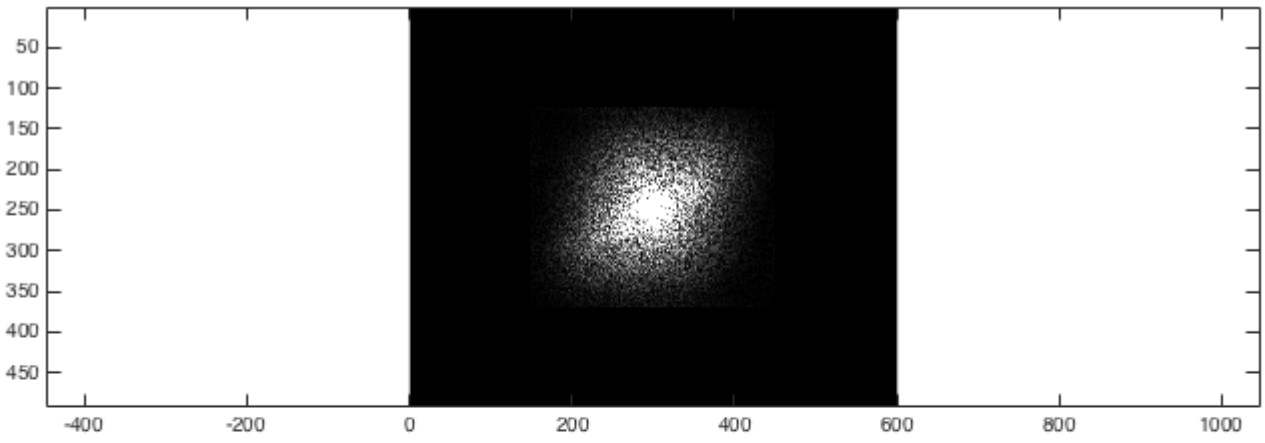


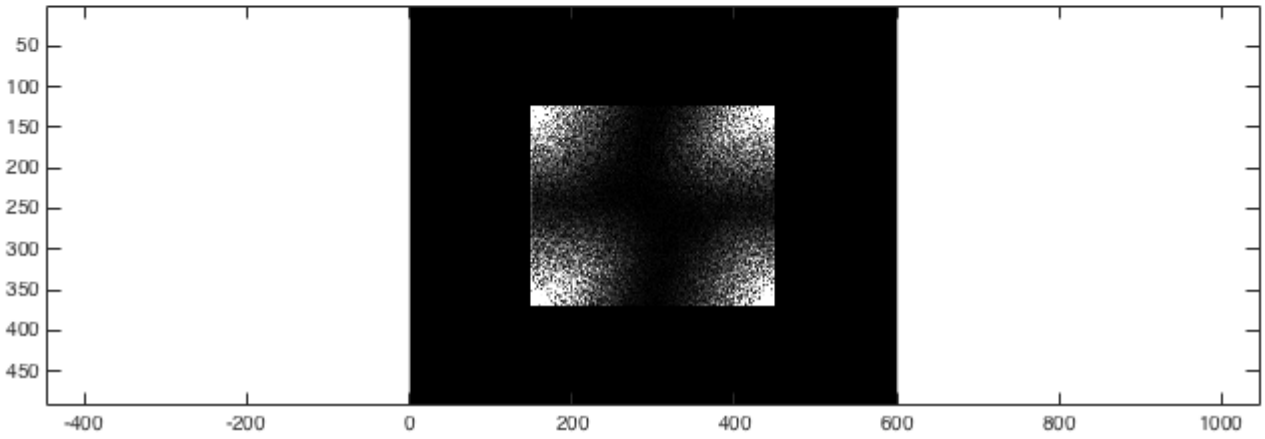


Теперь мы получили 2D FT примерного изображения. Чтобы нуль заполнить его, мы хотим взять каждое k-пространство, напасть на края нулями, а затем взять обратное преобразование:

```
%Zero fill
kzf = zeros(imageSize .* 2); %Generate a  492x600 empty array to put the result in
kzf(end/4:3*end/4-1,end/4:3*end/4-1) = k; %Put k in the middle
kzfwrong = zeros(imageSize .* 2); %Generate a  492x600 empty array to put the result in
kzfwrong(end/4:3*end/4-1,end/4:3*end/4-1) = kwrong; %Put k in the middle

%Show the differences again
%Just for the sake of comparison, show the magnitude of both transforms:
figure; subplot(2,1,1);
imagesc(abs(kzf),[0 1e4]); colormap gray; axis equal;
subplot(2,1,2);
imagesc(abs(kzfwrong),[0 1e4]); colormap gray; axis equal;
%(The second argument to imagesc sets the colour axis to make the difference clear).
```

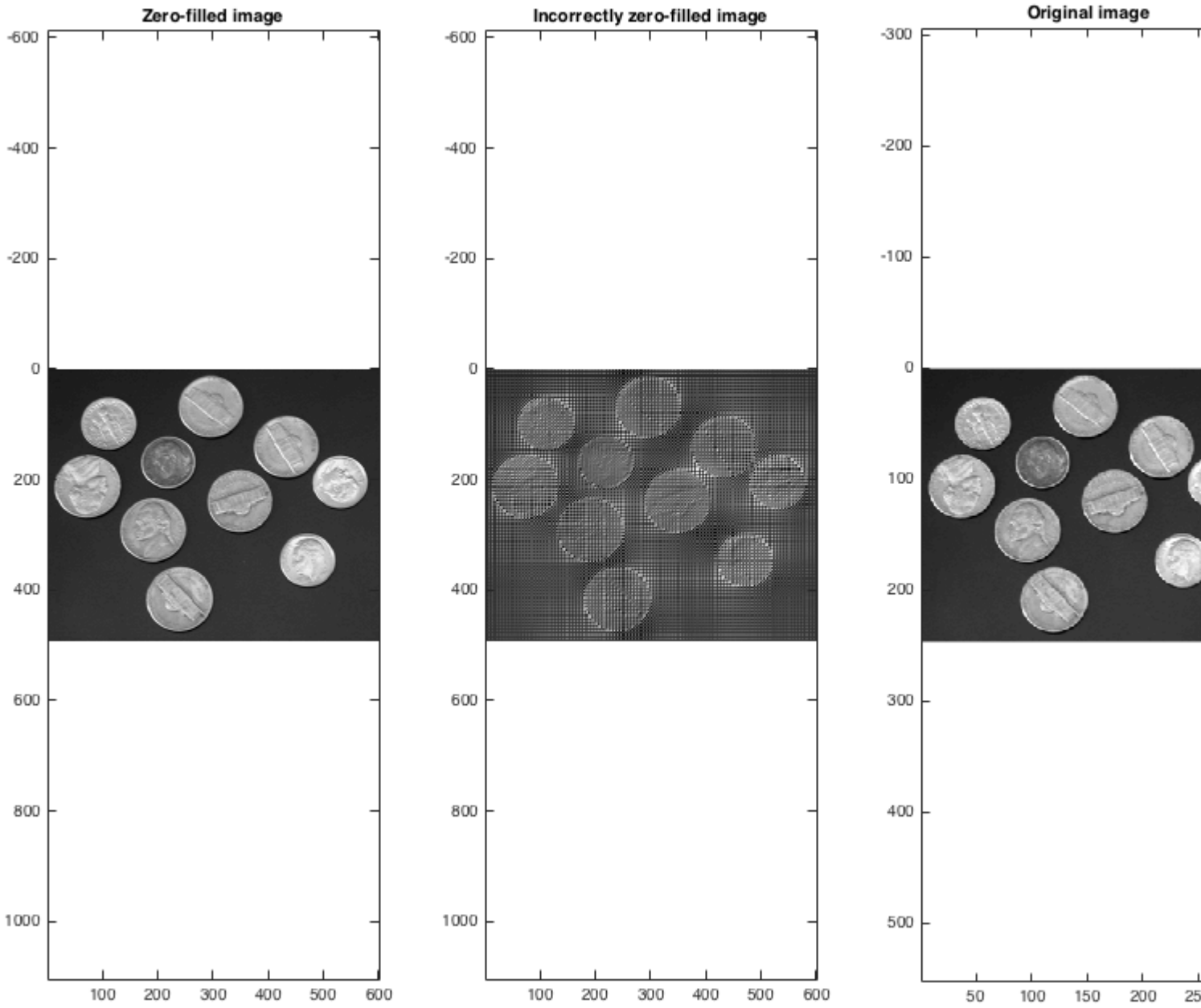
На данный момент результат довольно ничем не примечателен:





Как только мы возьмем обратные преобразования, мы можем видеть, что (правильно!) Данные с нулевым заполнением обеспечивают разумный метод интерполяции:

```
% Take the back transform and view
Izf = fftshift(ifft2(ifftshift(kzf)));
Izfwrong = ifft2(kzfwrong);

figure; subplot(1,3,1);
imagesc(abs(Izf)); colormap gray; axis equal;
title('Zero-filled image');
subplot(1,3,2);
imagesc(abs(Izfwrong)); colormap gray; axis equal;
title('Incorrectly zero-filled image');
subplot(1,3,3);
imagesc(I); colormap gray; axis equal;
title('Original image');
set(gcf,'color','w');
```



Обратите внимание, что размер заполненного нулем изображения вдвое больше, чем у оригинала. В каждом измерении может быть заполнено нулевое значение более чем в два раза, хотя, очевидно, это не приводит к произвольному увеличению размера изображения.

Советы, подсказки, 3D и дальше

Вышеприведенный пример выполняется для трехмерных изображений (как это часто генерируется методами медицинской визуализации или конфокальной микроскопии, например), но требует, `fft2` замены `fftn(I, 3)` на `fftn(I, 3)` . Из-за несколько громоздкой природы написания `fftshift(fft(fftshift(...` несколько раз, довольно часто для определения функций, таких как `fft2c` локально, для обеспечения более легкого синтаксиса локально - например:

```
function y = fft2c(x)
```

```
y = fftshift(fft2(fftshift(x)));
```

Обратите внимание, что FFT является быстрым, но большие многомерные преобразования Фурье по-прежнему занимают время на современном компьютере. Кроме того, он по своей сути является сложным: величина k-пространства была показана выше, но фаза абсолютно необходима; переводы в области изображения эквивалентны фазовой рампе в области Фурье. Есть несколько более сложных операций, которые, возможно, пожелают сделать в области Фурье, такие как фильтрация высоких или низких пространственных частот (путем умножения их на фильтр) или маскирование дискретных точек, соответствующих шуму. Соответственно, имеется большое количество кода, созданного сообществом для обработки общих операций Фурье, доступных на основном сайте репозитория сообщества Matlab, File Exchange .

Прочитайте Преобразования Фурье и обратные преобразования Фурье онлайн: https://riptutorial.com/ru/matlab/topic/2181/преобразования-фурье-и-обратные-преобразования-фурье

глава 24: Производительность и бенчмаркинг

замечания

- Профилирующий код - это способ избежать страшной практики « преждевременной оптимизации », сосредоточив внимание разработчиков на тех частях кода, которые *фактически* оправдывают усилия по оптимизации.
- Документация MATLAB под названием « Измерение эффективности вашей программы ».

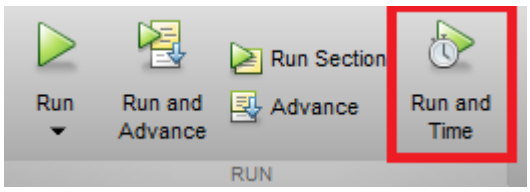
Examples

Определение узких мест производительности с помощью Profiler

MATLAB Profiler - это инструмент для профилирования программного кода MATLAB. Используя Profiler, можно получить визуальное представление как времени выполнения, так и потребления памяти.

Запуск Profiler можно выполнить двумя способами:

- Нажатие кнопки «Выполнить и время» в графическом интерфейсе MATLAB при открытии некоторого файла `.m` в редакторе (добавлено в **R2012b**).



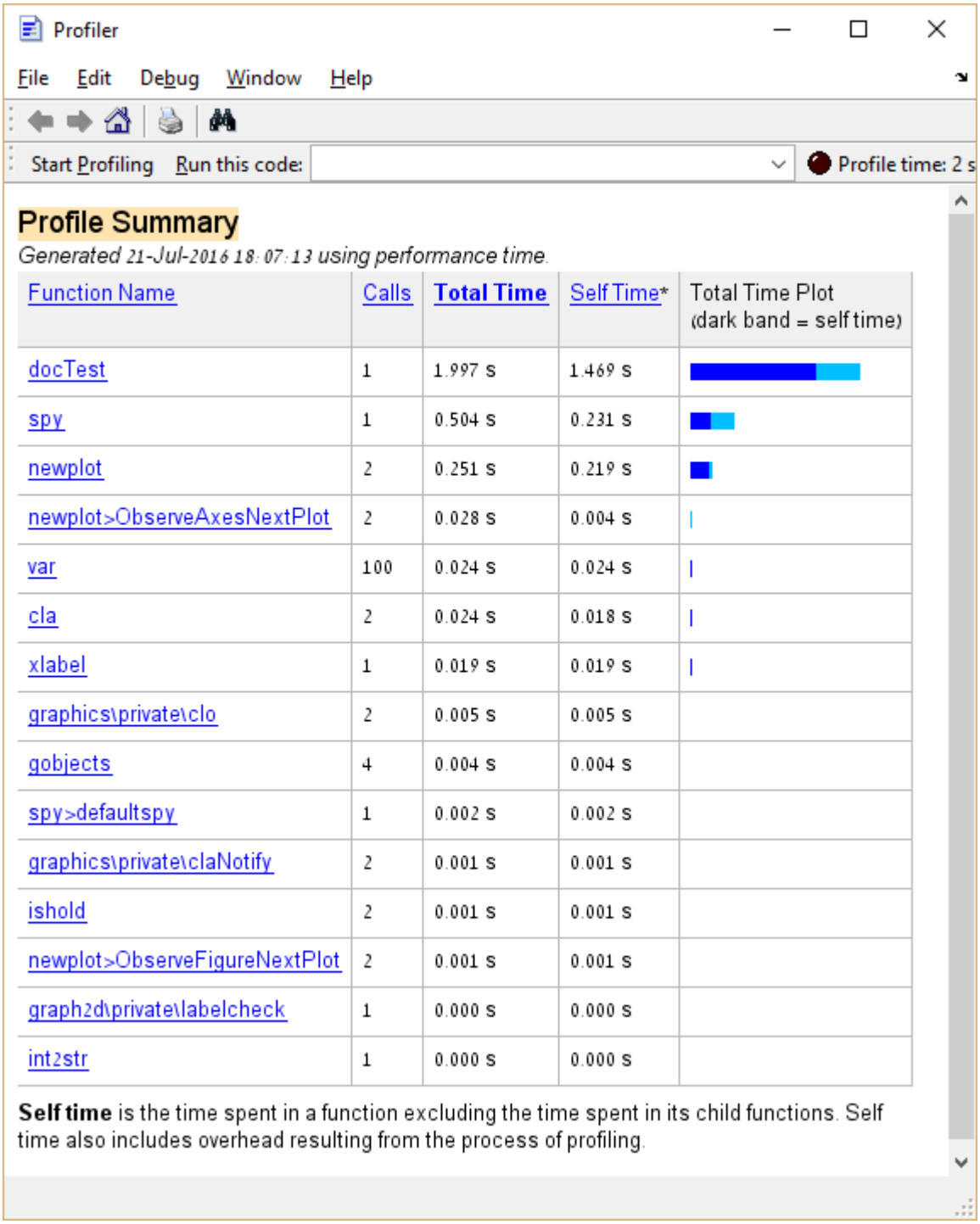
- Программно, используя:

```
profile on
<some code we want to test>
profile off
```

Ниже приведен пример кода примера и результат его профилирования:

```
function docTest

for ind1 = 1:100
  [~] = var(...
            sum(...
                randn(1000)));
end
```

```
spy
```



Profile Summary

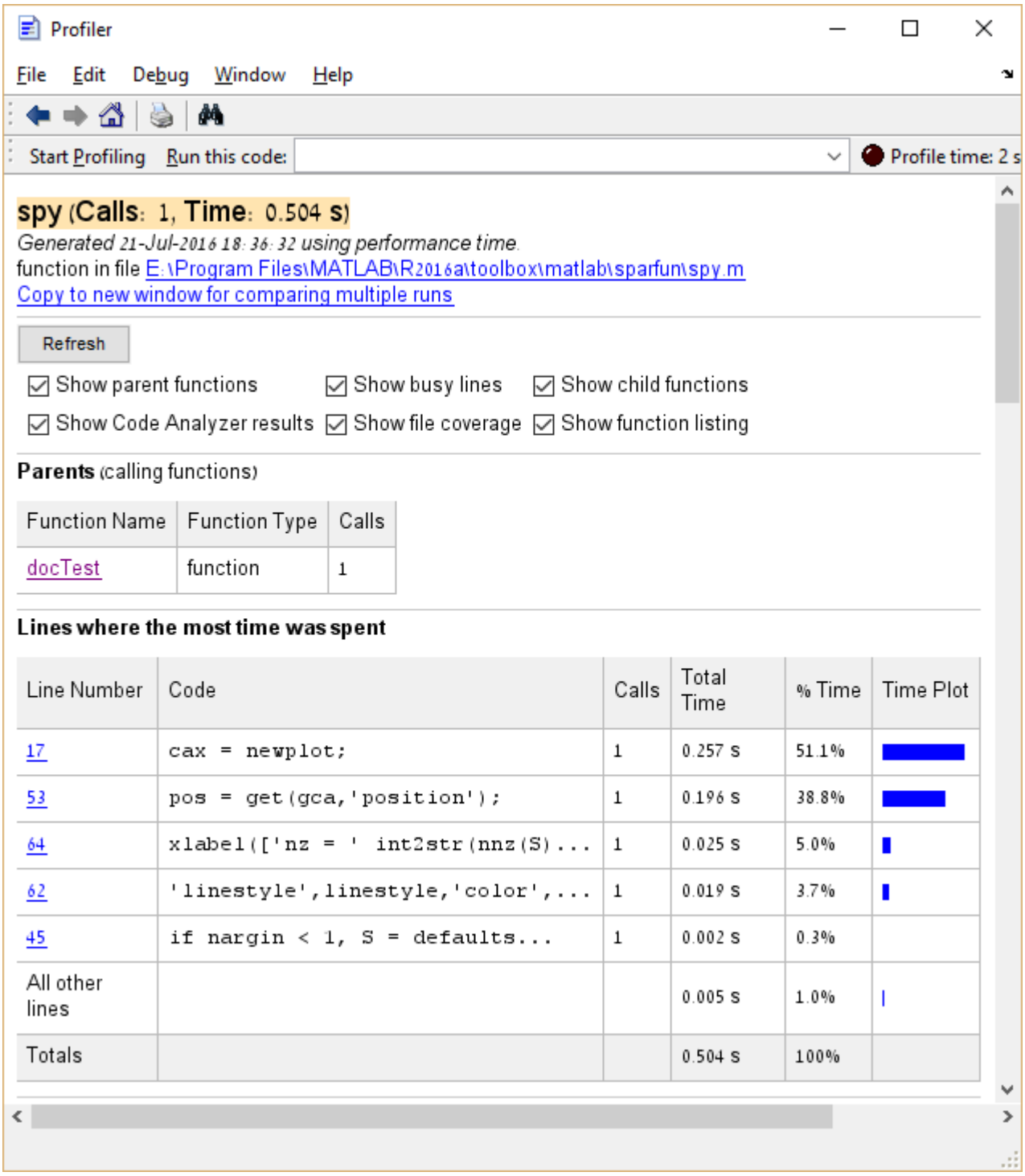
Generated 21-Jul-2016 18:07:13 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
docTest	1	1.997 s	1.469 s	
spy	1	0.504 s	0.231 s	
newplot	2	0.251 s	0.219 s	
newplot>ObserveAxesNextPlot	2	0.028 s	0.004 s	
var	100	0.024 s	0.024 s	
cla	2	0.024 s	0.018 s	
xlabel	1	0.019 s	0.019 s	
graphics\private\clo	2	0.005 s	0.005 s	
gobjects	4	0.004 s	0.004 s	
spy>defaultspy	1	0.002 s	0.002 s	
graphics\private\claNotify	2	0.001 s	0.001 s	
ishold	2	0.001 s	0.001 s	
newplot>ObserveFigureNextPlot	2	0.001 s	0.001 s	
graph2d\private\labelcheck	1	0.000 s	0.000 s	
int2str	1	0.000 s	0.000 s	

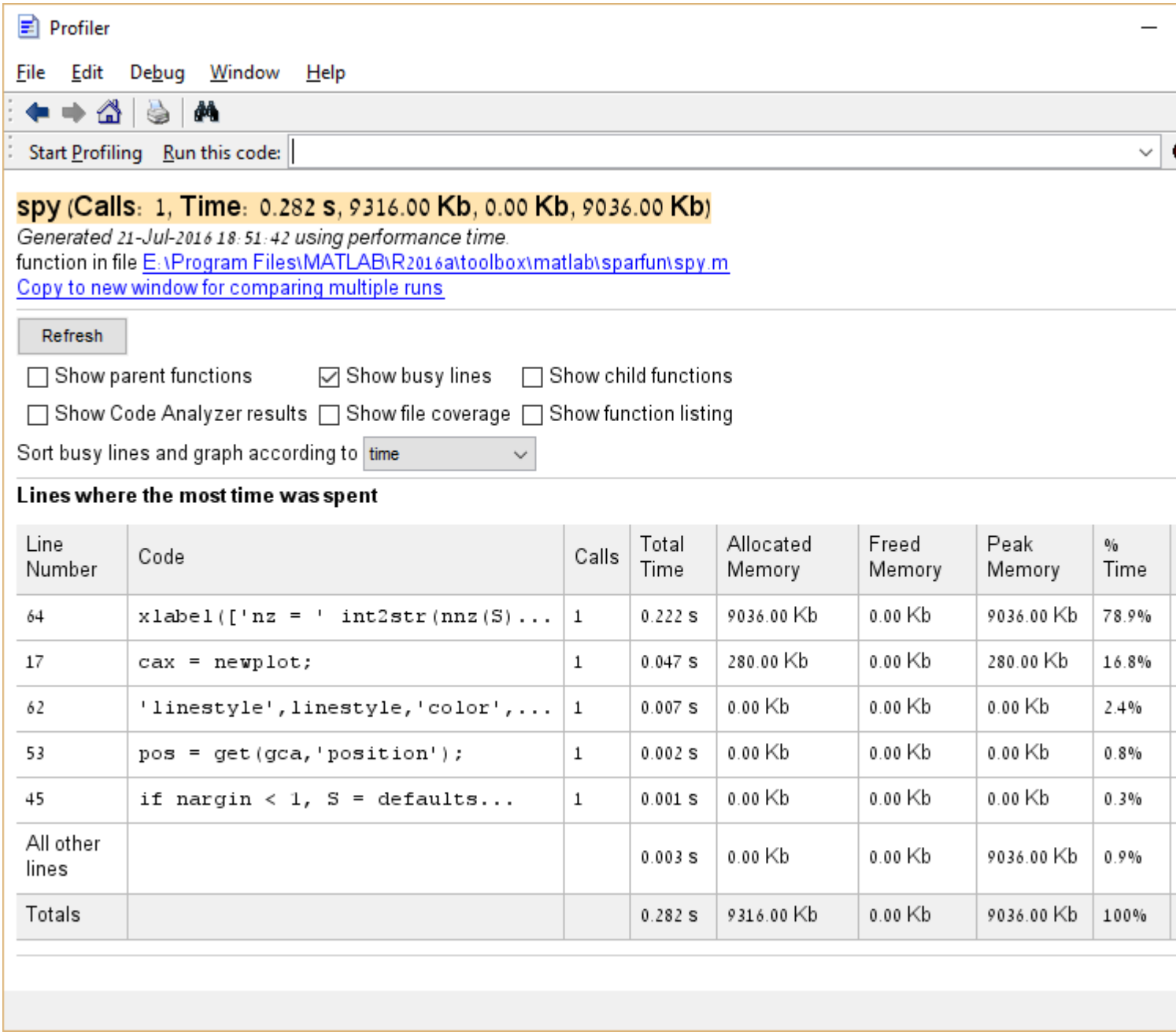
Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Из вышесказанного мы узнаем, что функция `spy` занимает около 25% от общего времени выполнения. В случае «реального кода» функция, которая занимает такой большой процент времени выполнения, будет хорошим кандидатом на оптимизацию, в отличие от функций, аналогичных `var` и `cla` , оптимизация которых следует избегать.

Кроме того, можно щелкнуть записи в столбце « *Имя функции»,* чтобы просмотреть подробную разбивку времени выполнения для этой записи. Вот пример щелчка `spy` :



Также возможно профилировать потребление памяти, выполняя `profile('-memory')` перед запуском Profiler.



Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time
64	`xlabel(['nz = ' int2str(nnz(S)...`	1	0.222 s	9036.00 Kb	0.00 Kb	9036.00 Kb	78.9%
17	`cax = newplot;`	1	0.047 s	280.00 Kb	0.00 Kb	280.00 Kb	16.8%
62	`'linestyle',linestyle,'color',...`	1	0.007 s	0.00 Kb	0.00 Kb	0.00 Kb	2.4%
53	`pos = get(gca,'position');`	1	0.002 s	0.00 Kb	0.00 Kb	0.00 Kb	0.8%
45	`if nargin < 1, S = defaults...`	1	0.001 s	0.00 Kb	0.00 Kb	0.00 Kb	0.3%
All other lines			0.003 s	0.00 Kb	0.00 Kb	9036.00 Kb	0.9%
Totals			0.282 s	9316.00 Kb	0.00 Kb	9036.00 Kb	100%

Сравнение времени выполнения нескольких функций

Широко используемая комбинация `tic` и `toc` может дать приблизительное представление о времени выполнения функции или фрагментов кода.

Для сравнения нескольких функций он не должен использоваться. Зачем? Почти невозможно обеспечить *равные условия* для всех фрагментов кода для сравнения в скрипте с использованием вышеприведенного решения. Возможно, функции имеют одно и то же функциональное пространство и общие переменные, поэтому более поздние функции и фрагменты кода уже используют ранее инициализированные переменные и функции. Также нет никакой информации о том, будет ли компилятор JIT обрабатывать эти впоследствии называемые фрагменты одинаково.

`timeit` функция для эталонных тестов - `timeit` . Следующий пример иллюстрирует его использование.

Есть массив `A` и матрица `B` Следует определить, какая строка `B` наиболее похожа на `A` , подсчитывая количество разных элементов.

```
function t = bench()
    A = [0 1 1 1 0 0];
    B = perms(A);

    % functions to compare
    fcns = {
        @() compare1(A,B);
        @() compare2(A,B);
        @() compare3(A,B);
        @() compare4(A,B);
    };

    % timeit
    t = cellfun(@timeit, fcns);
end

function Z = compare1(A,B)
    Z = sum(  bsxfun(@eq,  A,B) , 2);
end
function Z = compare2(A,B)
    Z = sum(bsxfun(@xor, A, B),2);
end
function Z = compare3(A,B)
    A = logical(A);
    Z = sum(B(:,~A),2) + sum(~B(:,A),2);
end
function Z = compare4(A,B)
     Z = pdist2( A, B, 'hamming', 'Smallest', 1 );
end
```

Этот метод тестирования был впервые замечен в этом ответе .

Это нормально быть «единственным»!

Обзор:

Тип данных по умолчанию для числовых массивов в MATLAB является `double` . `double` - это число чисел с плавающей запятой , и этот формат занимает 8 байтов (или 64 бит) за каждое значение. В **некоторых** случаях, когда, например, работа с целыми числами или когда числовая неустойчивость не является непосредственной проблемой, такая высокая глубина бит может не потребоваться. По этой причине рекомендуется учитывать преимущества `single` точности (или других подходящих типов):

- Более быстрое время выполнения (особенно заметно на графических процессорах).
- Половина потребления памяти: может преуспеть, если `double` сбой из-за ошибки из-за

памяти; более компактный при хранении в виде файлов.

Преобразование переменной из любого поддерживаемого типа данных в `single` выполняется с помощью:

```
sing_var = single(var);
```

Некоторые обычно используемые функции (такие как: `zeros` , `eye` , `ones` и т. Д.), Которые по умолчанию выдают `double` значения, позволяют указать тип / класс вывода.

Преобразование переменных в сценарий в нестандартную точность / тип / класс:

По состоянию на июль 2016 года не существует документального способа изменения типа данных MATLAB по *умолчанию* из `double` .

В MATLAB новые переменные обычно имитируют типы данных переменных, используемых при их создании. Чтобы проиллюстрировать это, рассмотрим следующий пример:

```
A = magic(3);
B = diag(A);
C = 20*B;
>> whos C
  Name      Size             Bytes  Class     Attributes
  C         3x1                 24  double
```

```
A = single(magic(3)); % A is converted to "single"
B = diag(A);
C = B*double(20);     % The stricter type, which in this case is "single", prevails
D = single(size(C));  % It is generally advised to cast to the desired type explicitly.
>> whos C
  Name      Size             Bytes  Class     Attributes
  C         3x1                 12  single
```

Таким образом, может показаться достаточным для перевода / преобразования нескольких начальных переменных, чтобы изменение пронизывало всюду по коду - однако это не **рекомендуется** (см. Ниже « *Оговорки и ошибки»*).

Предостережения и ошибки:

1. Повторные конверсии не **рекомендуется** из-за введения числового шума (при литье из `single` в `double`) или потери информации (при кастинге от `double` до `single` или между определенными целыми типами), например:

```
double(single(1.2)) == double(1.2)
ans =
     0
```

Это можно немного смягчить, используя `typecast` . См. Также Будьте в курсе неточности с плавающей запятой .

2. Опираясь исключительно на неявное типирование данных (то есть, что MATLAB догадывается, какой тип вывода вычисления должен быть) не **рекомендуется** из-за нескольких нежелательных эффектов, которые могут возникнуть:

   - *Потеря информации* : когда ожидается `double` результат, но небрежное сочетание `single` и `double` операндов дает `single` точность.
   - *Неожиданно высокая потребляемая память* : когда ожидается `single` результат, но небрежное вычисление приводит к `double` выходу.
   - *Ненужные накладные расходы при работе с графическими процессорами* : при смешивании типов `gpuArray` (т. `gpuArray` Переменных, хранящихся в VRAM) с переменными `gpuArray` (то есть, которые *обычно* хранятся в ОЗУ) данные должны быть переданы так или иначе, прежде чем вычисление может быть выполнено. Эта операция требует времени и может быть очень заметна при повторных вычислениях.
   - *Ошибки при смешивании типов с плавающей запятой с целыми типами* : такие функции, как `mtimes` (`*`), не определены для смешанных входов целых чисел и типов с плавающей точкой - и будут ошибки. Функции типа `times` (`.*`) Вообще не определены для входов целых типов - и снова будут ошибки.

     ```
     >> ones(3,3,'int32')*ones(3,3,'int32')
     Error using  *
     MTIMES is not fully supported for integer classes. At least one input must be
     scalar.

     >> ones(3,3,'int32').*ones(3,3,'double')
     Error using  .*
     Integers can only be combined with integers of the same class, or scalar doubles.
     ```

   Для лучшей читаемости коды и снижения риска нежелательных видов, оборонительный подход **рекомендуются,** где переменные *явно* приводятся к нужному типу.

Смотрите также:

- Документация MATLAB: номера с плавающей запятой .
- Техническая статья Mathworks: лучшие практики для преобразования кода MATLAB в

фиксированную точку .

переупорядочить ND-массив может улучшить общую производительность

В некоторых случаях нам нужно применять функции к набору ND-массивов. Давайте посмотрим на этот простой пример.

```
A(:,:,1) = [1 2; 4 5];
A(:,:,2) = [11 22; 44 55];
B(:,:,1) = [7 8; 1 2];
B(:,:,2) = [77 88; 11 22];

A =

ans(:,:,1) =

   1   2
   4   5

ans(:,:,2) =

   11   22
   44   55

>> B
B =

ans(:,:,1) =

   7   8
   1   2

ans(:,:,2) =

   77   88
   11   22
```

Обе матрицы 3D, скажем, мы должны вычислить следующее:

```
result= zeros(2,2);
...
for k = 1:2
   result(i,j) = result(i,j) + abs( A(i,j,k) - B(i,j,k) );
...

if k is very large, this for-loop can be a bottleneck since MATLAB order the data in a column
major fashion. So a better way to compute "result" could be:

% trying to exploit the column major ordering
Aprime = reshape(permute(A,[3,1,2]), [2,4]);
Bprime = reshape(permute(B,[3,1,2]), [2,4]);


>> Aprime
Aprime =
```

```
    1    4    2    5
   11   44   22   55

>> Bprime
Bprime =

    7    1    8    2
   77   11   88   22
```

Теперь мы заменим вышеприведенный цикл следующим образом:

```
result= zeros(2,2);
....
temp = abs(Aprime - Bprime);
for k = 1:2
    result(i,j) = result(i,j) + temp(k, i+2*(j-1));
...
```

Мы перегруппировали данные, чтобы мы могли использовать кэш-память. Пермутация и изменение могут быть дорогостоящими, но при работе с большими ND-массивами вычислительная стоимость, связанная с этими операциями, намного ниже, чем работа с не организованными массивами.

Важность предварительного распределения

Массивы в MATLAB хранятся в виде непрерывных блоков в памяти, которые автоматически выделяются и выпускаются MATLAB. MATLAB скрывает операции управления памятью, такие как изменение размера массива за простым в использовании синтаксисом:

```
a = 1:4

a =

     1     2     3     4

a(5) = 10  % or alternatively a = [a, 10]

a =

     1     2     3     4    10
```

Важно понимать, что вышеизложенное не является тривиальной операцией, `a(5) = 10` заставит MATLAB выделить новый блок памяти размером 5, скопировать первые 4 числа и установить 5'-10. Это операция `O(numel(a))` , а не `O(1)` .

Рассмотрим следующее:

```
clear all
n=12345678;
a=0;
tic
for i = 2:n
```

```
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 3.004213 seconds.
```

`a` перераспределяется `n` раз в этом цикле (исключая некоторые оптимизации, предпринятые MATLAB)! Обратите внимание, что MATLAB дает нам предупреждение:

> «Переменная« a », по-видимому, меняет размер на каждой итерации цикла. Подумайте о предварительном распределении для скорости».

Что происходит, когда мы предварительно распределяем?

```
a=zeros(1,n);
tic
for i = 2:n
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 0.410531 seconds.
```

Мы видим, что время выполнения сокращается на порядок.

Методы предварительного распределения:

MATLAB предоставляет различные функции для размещения векторов и матриц в зависимости от конкретных требований пользователя. К ним относятся: `zeros` , `ones` , `nan` , `eye` , `true` и т. Д.

```
a = zeros(3)       % Allocates a 3-by-3 matrix initialized to 0
a =

     0     0     0
     0     0     0
     0     0     0

a = zeros(3, 2)     % Allocates a 3-by-2 matrix initialized to 0
a =

     0     0
     0     0
     0     0

a = ones(2, 3, 2)      % Allocates a 3 dimensional array (2-by-3-by-2) initialized to 1
a(:,:,1) =

     1     1     1
     1     1     1


a(:,:,2) =

     1     1     1
```

```
     1     1     1

a = ones(1, 3) * 7  % Allocates a row vector of length 3 initialized to 7
a =

     7     7     7
```

Также может быть указан тип данных:

```
a = zeros(2, 1, 'uint8');  % allocates an array of type uint8
```

Также легко клонировать размер существующего массива:

```
a = ones(3, 4);       % a is a 3-by-4 matrix of 1's
b = zeros(size(a));  % b is a 3-by-4 matrix of 0's
```

И клонируйте тип:

```
a = ones(3, 4, 'single');       % a is a 3-by-4 matrix of type single
b = zeros(2, 'like', a);        % b is a 2-by-2 matrix of type single
```

обратите внимание, что «как» также клонирует *сложность* и *разреженность* .

Предварительное выделение неявно достигаются с помощью какой - либо функции , которая возвращает массив конечного требуемого размера, например, `rand` , `gallery` , `kron` , `bsxfun` , `colon` и многие другие. Например, общий способ выделения векторов с линейно изменяющимися элементами - это использование оператора двоеточия (с вариантом 2 или 3-операнда [1]):

```
a = 1:3
a =

     1     2     3

a = 2:-3:-4
a =

     2    -1    -4
```

Ячейные массивы могут быть выделены с помощью функции `cell()` почти так же, как и `zeros()` .

```
a = cell(2,3)
a =

    []    []    []
    []    []    []
```

Обратите внимание, что массивы ячеек работают, удерживая указатели в ячейках памяти в содержимом ячейки. Таким образом, все подсказки preallocation применяются также к

отдельным элементам массива ячеек.

Дальнейшее чтение:

- Официальная документация MATLAB « **Предварительная память** ».
- Официальная документация MATLAB « **Как MATLAB выделяет память** ».
- Производительность предварительного распределения на **недокументированной матрице** .
- Понимание массива Prelocation на **Loren по искусству MATLAB**

Прочитайте Производительность и бенчмаркинг онлайн:
https://riptutorial.com/ru/matlab/topic/1141/производительность-и-бенчмаркинг

глава 25: Реальные решатели дифференциальных уравнений (ОДУ)

Examples

Пример для odeset

Сначала мы инициализируем нашу начальную задачу, которую хотим решить.

```
odefun = @(t,y) cos(y).^2*sin(t);
tspan = [0 16*pi];
y0=1;
```

Затем мы используем функцию ode45 без каких-либо определенных опций для решения этой проблемы. Чтобы сравнить это позже, мы построим траекторию.

```
[t,y] = ode45(odefun, tspan, y0);
plot(t,y,'-o');
```

Теперь мы устанавливаем узкий относительный и узкий абсолютный предел терпимости к нашей проблеме.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Мы установили жесткий относительный и узкий абсолютный предел толерантности.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Мы устанавливаем узкий относительный и плотный абсолютный предел толерантности. Как и в предыдущих примерах с узкими пределами толерантности, мы видим, что траектория полностью отличается от первого графика без каких-либо конкретных опций.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Мы установили жесткий относительный и плотный абсолютный предел допуска. Сравнивая результат с другим графиком, мы будем подчеркивать ошибки, сделанные с использованием узких пределов допуска.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Следующее должно продемонстрировать компромисс между точностью и временем выполнения.

```
tic;
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
time1 = toc;
plot(t,y,'-o');
```

Для сравнения мы затягиваем предел допуска для абсолютной и относительной погрешности. Теперь мы можем видеть, что без большого выигрыша потребуется значительно больше времени для решения нашей начальной задачи.

```
tic;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
[t,y] = ode45(odefun, tspan, y0, options);
time2 = toc;
plot(t,y,'-o');
```

Прочитайте Реальные решатели дифференциальных уравнений (ОДУ) онлайн: https://riptutorial.com/ru/matlab/topic/6079/реальные-решатели-дифференциальных-уравнений--оду-

глава 26: Рисование

Examples

круги

Самый простой способ рисовать круг - это, очевидно, функция `rectangle` .

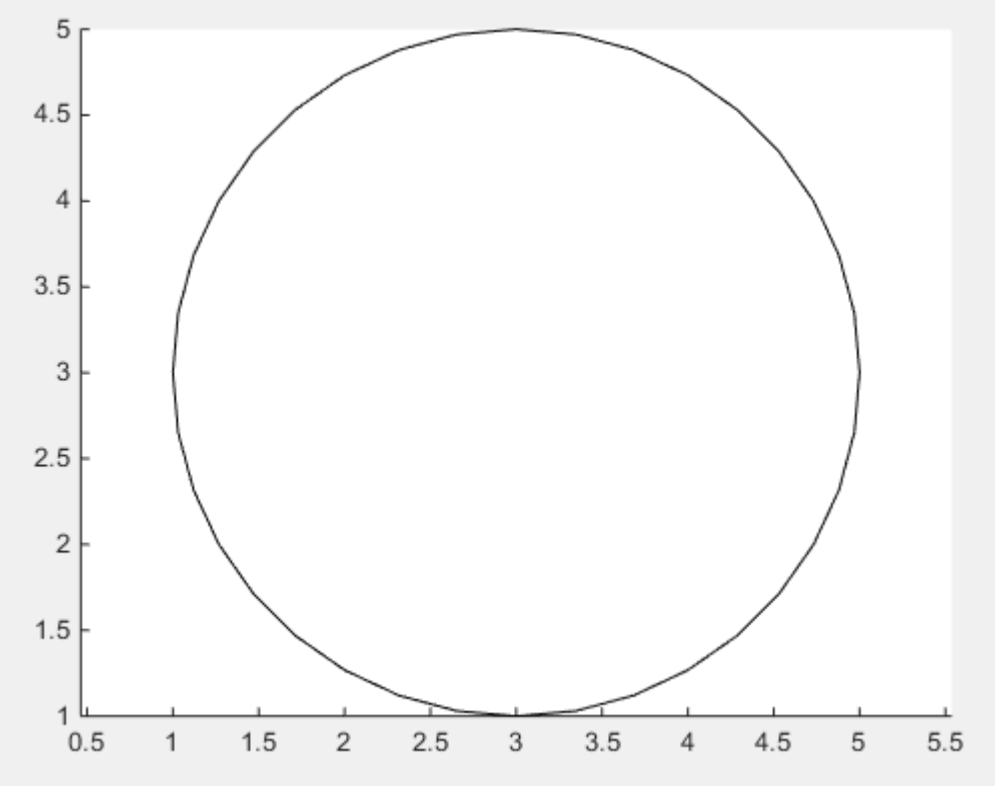
```
%// radius
r = 2;

%// center
c = [3 3];

pos = [c-r 2*r 2*r];
rectangle('Position',pos,'Curvature',[1 1])
axis equal
```

но кривизна прямоугольника должна быть равна **1** !

Вектор `position` определяет прямоугольник, первые два значения `x` и `y` - нижний левый угол прямоугольника. Последние два значения определяют ширину и высоту прямоугольника.

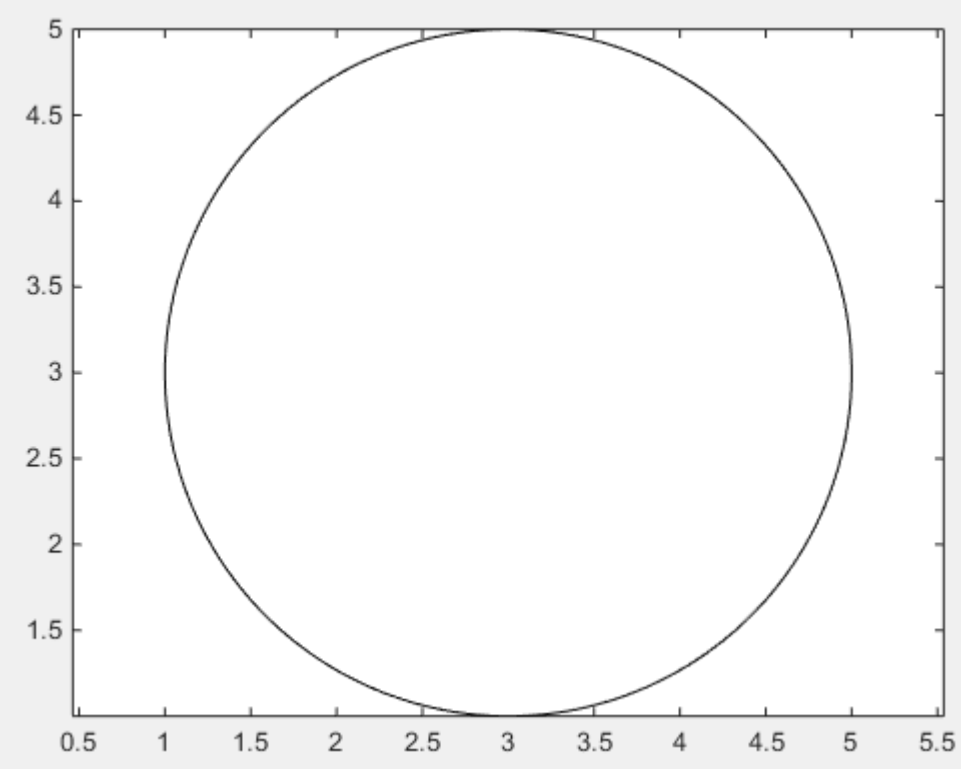
```
pos = [ [x y] width height ]
```

В нижнем левом *углу* круга - да, этот круг имеет углы, мнимые, хотя - это **центр** `c = [3 3]` **минус радиус** `r = 2` который `[xy] = [1 1]` . **Ширина** и **высота** равны **диаметру** круга, поэтому `width = 2*r; height = width;`



В случае, если гладкость вышеуказанного решения недостаточна, нет возможности использовать очевидный способ рисования реального круга с помощью **тригонометрических функций** .

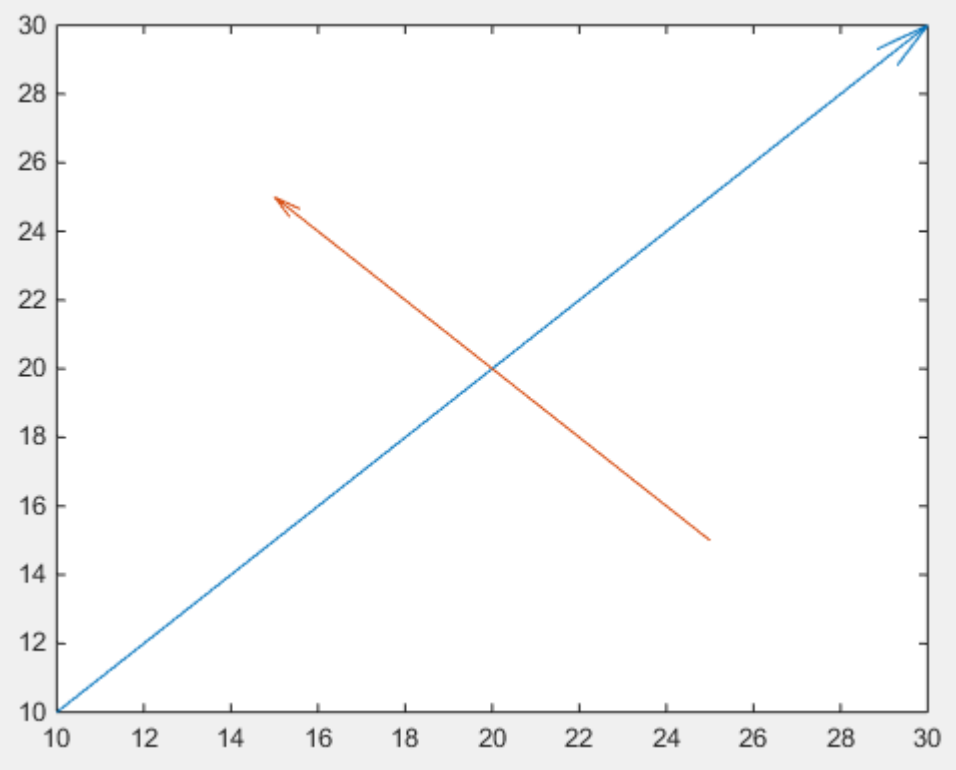
```
%// number of points
n = 1000;

%// running variable
t = linspace(0,2*pi,n);

x = c(1) + r*sin(t);
y = c(2) + r*cos(t);

%// draw line
line(x,y)

%// or draw polygon if you want to fill it with color
%// fill(x,y,[1,1,1])
axis equal
```



Стрелы

Во-первых, можно использовать `quiver` , где не нужно иметь дело с неуправляемыми нормализованными фигурами с помощью `annotation`

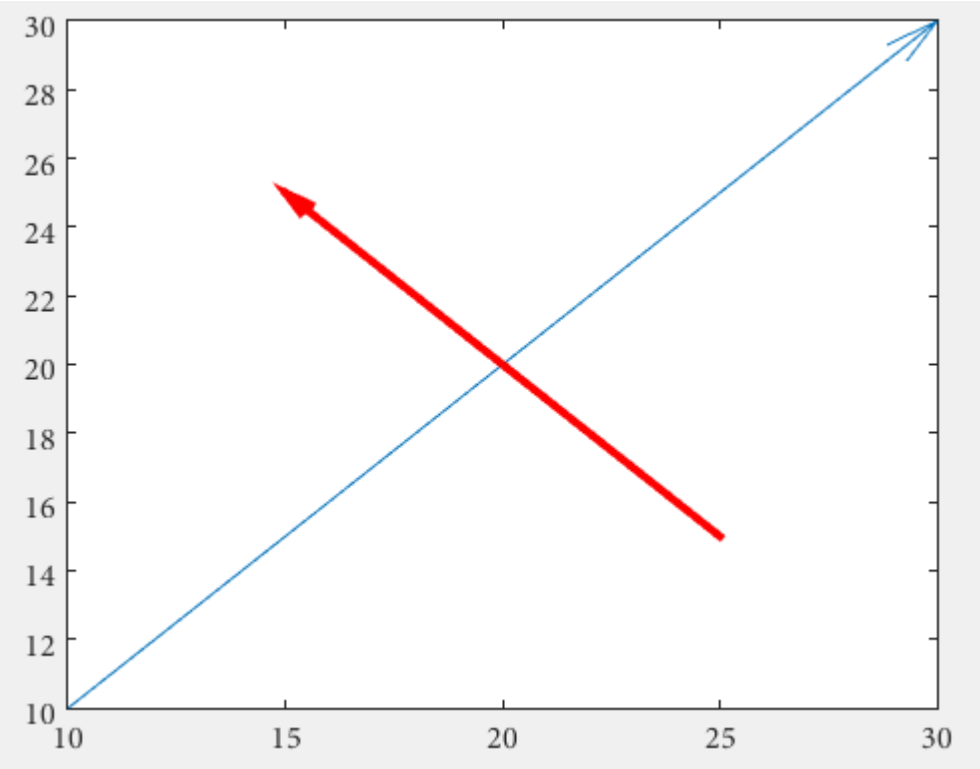
```
drawArrow = @(x,y) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0 )

x1 = [10 30];
y1 = [10 30];

drawArrow(x1,y1); hold on

x2 = [25 15];
y2 = [15 25];

drawArrow(x2,y2)
```



Важным является 5-й аргумент `quiver` : 0, который отключает масштабирование по умолчанию по умолчанию, поскольку эта функция обычно используется для построения векторных полей. (или использовать пару значений свойств `'AutoScale','off'`)

Можно также добавить дополнительные функции:

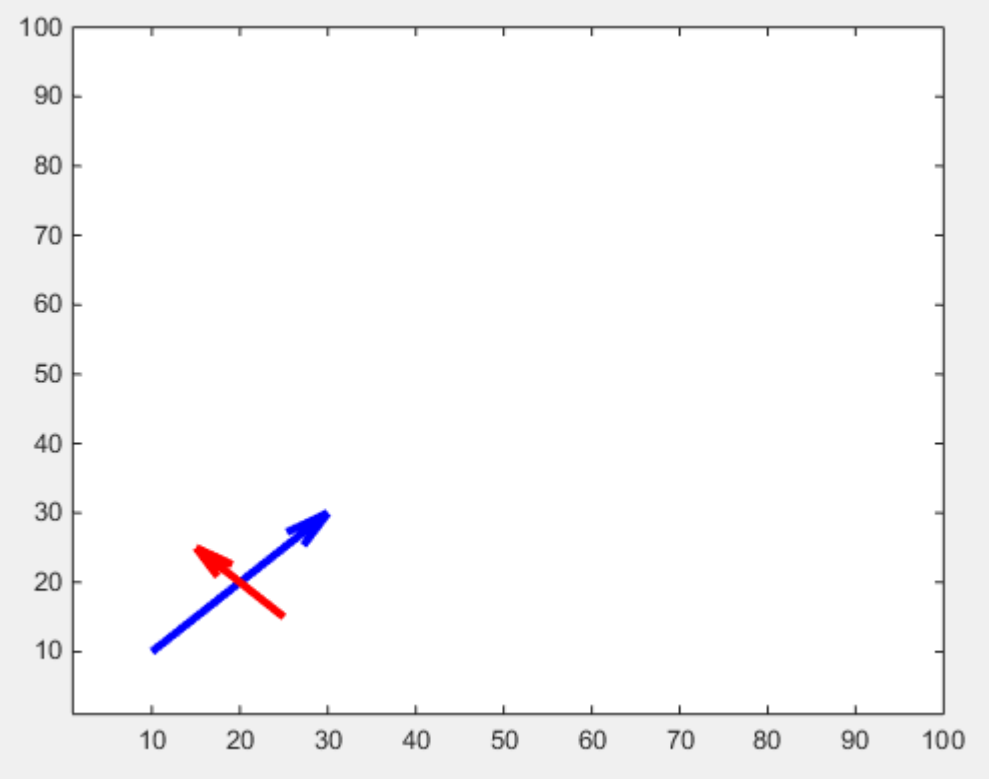
```
drawArrow = @(x,y,varargin) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0, varargin{:} )
drawArrow(x1,y1); hold on
drawArrow(x2,y2,'linewidth',3,'color','r')
```



Если желательны разные стрелки, нужно использовать аннотации (этот ответ может быть полезен. Как изменить стиль стрелочной головки в квадрате колчана?).

Размер головки стрелки можно настроить с помощью `'MaxHeadSize'` . К сожалению, это непротиворечиво. После этого необходимо установить ограничения по осям.

```
x1 = [10 30];
y1 = [10 30];
drawArrow(x1,y1,{'MaxHeadSize',0.8,'Color','b','LineWidth',3}); hold on

x2 = [25 15];
y2 = [15 25];
drawArrow(x2,y2,{'MaxHeadSize',10,'Color','r','LineWidth',3}); hold on

xlim([1, 100])
ylim([1, 100])
```

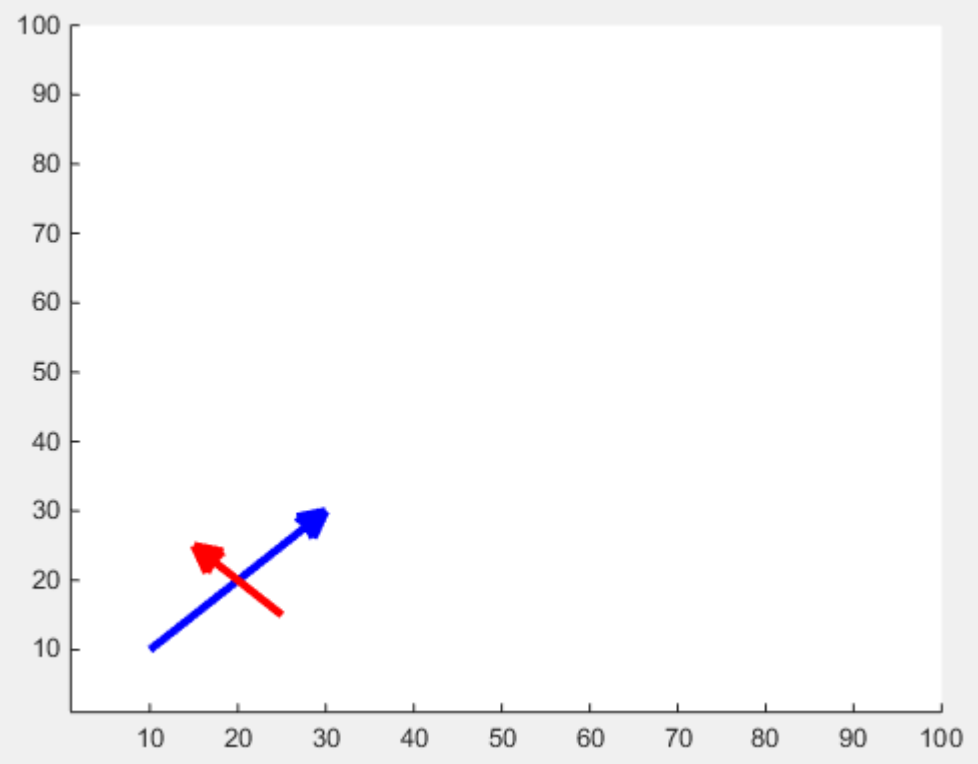


Существует еще одна настройка для регулируемых головок стрелок:

```
function [ h ] = drawArrow( x,y,xlimits,ylimits,props )

xlim(xlimits)
ylim(ylimits)

h = annotation('arrow');
set(h,'parent', gca, ...
    'position', [x(1),y(1),x(2)-x(1),y(2)-y(1)], ...
    'HeadLength', 10, 'HeadWidth', 10, 'HeadStyle', 'cback1', ...
    props{:} );

end
```

который вы можете вызвать из своего скрипта следующим образом:

```
drawArrow(x1,y1,[1, 100],[1, 100],{'Color','b','LineWidth',3}); hold on
drawArrow(x2,y2,[1, 100],[1, 100],{'Color','r','LineWidth',3}); hold on
```



Эллипс

Чтобы построить эллипс, вы можете использовать его уравнение . Эллипс имеет основную и малую оси. Также мы хотим иметь возможность построить эллипс в разных центральных точках. Поэтому мы пишем функцию, входы и выходы которой:

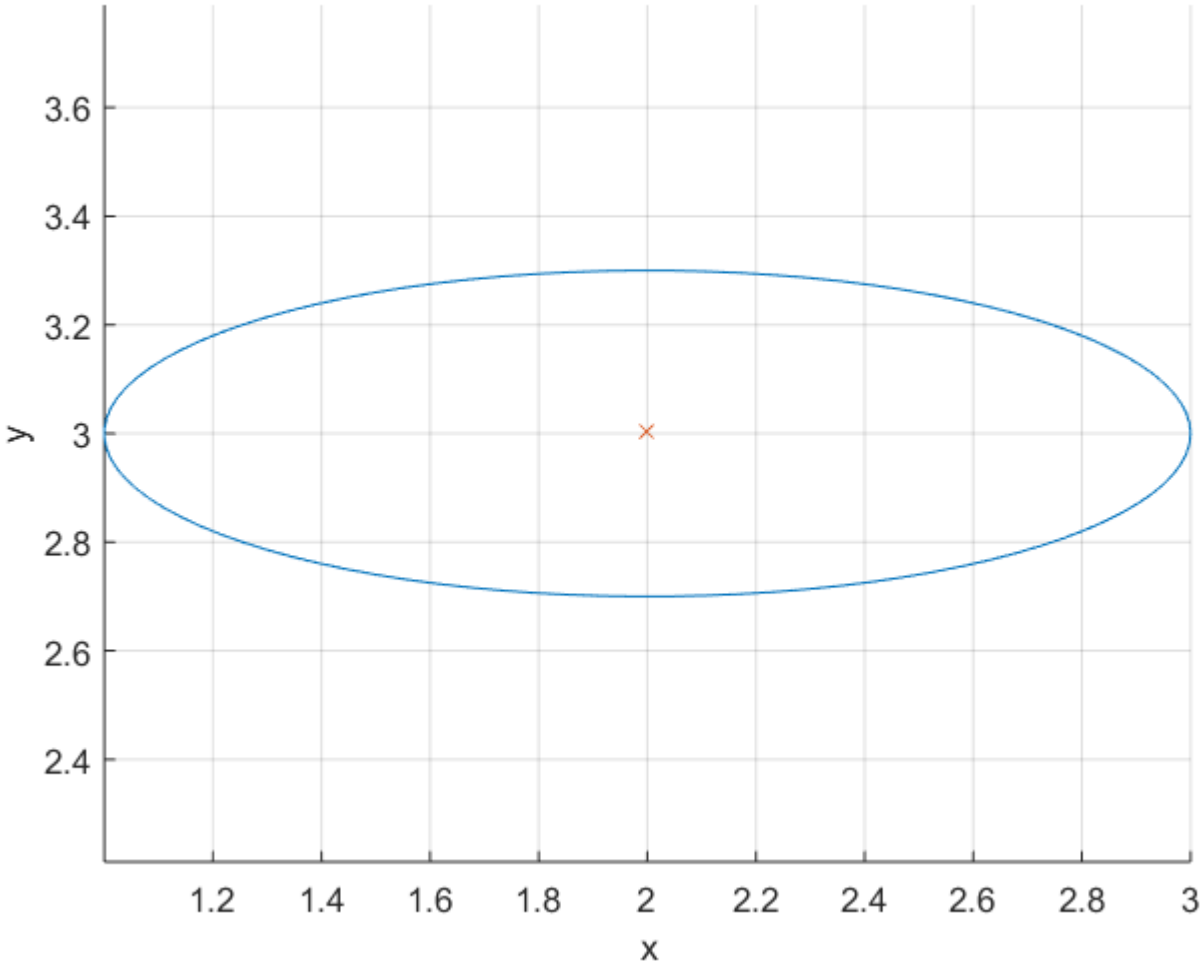
```
Inputs:
    r1,r2: major and minor axis respectively
    C: center of the ellipse (cx,cy)
Output:
    [x,y]: points on the circumference of the ellipse
```

Вы можете использовать следующую функцию, чтобы получить точки на эллипсе, а затем построить эти точки.

```
function [x,y] = getEllipse(r1,r2,C)
beta = linspace(0,2*pi,100);
x = r1*cos(beta) - r2*sin(beta);
y = r1*cos(beta) + r2*sin(beta);
x = x + C(1,1);
y = y + C(1,2);
end
```

Exmaple:

```
[x,y] = getEllipse(1,0.3,[2 3]);
plot(x,y);
```

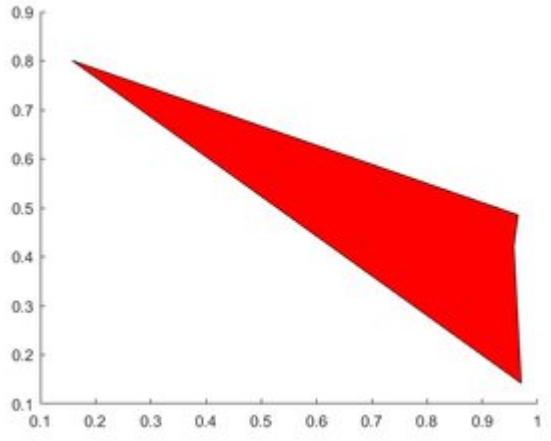


Многоугольник (ы)

Создайте векторы для хранения x- и y-расположений вершин, передайте их в `patch` .

Одиночный полигон

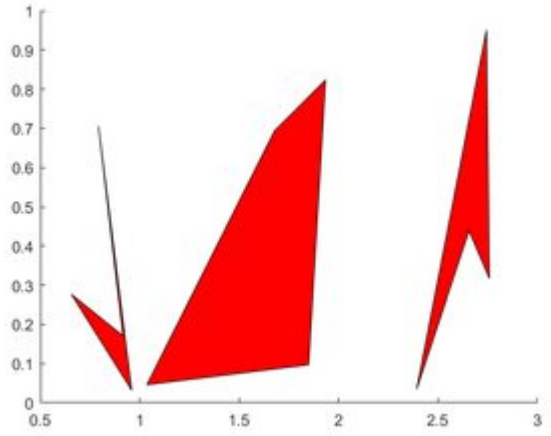
```
X=rand(1,4); Y=rand(1,4);
h=patch(X,Y,'red');
```



Многоугольники

Каждая вершина полигона занимает один столбец каждого из `X` , `Y`

```
X=rand(4,3); Y=rand(4,3);
for i=2:3
  X(:,i)=X(:,i)+(i-1); % create horizontal offsets for visibility
end

h=patch(X,Y,'red');
```



Псевдо 4D сюжет

`(mxn)` A `(mxn)` может представлять собой поверхность с помощью surf ;

Цвет поверхности автоматически устанавливается как функция значений в матрице `(mxn)` . Если цветовая палитра не указана, применяется по умолчанию.

Цветную панель можно добавить для отображения текущей цветовой карты и указать отображение значений данных в цветовой карте .

В следующем примере матрица `z (mxn)` генерируется функцией:

```
z=x.*y.*sin(x).*cos(y);
```

над интервалом `[-pi,pi]` . Значения `x` и `y` могут быть сгенерированы с использованием функции meshgrid, и поверхность отображается следующим образом:

```
% Create a Figure
figure
% Generate the `x` and `y` values in the interval `[-pi,pi]`
[x,y] = meshgrid([-pi:.2:pi],[-pi:.2:pi]);
% Evaluate the function over the selected interval
z=x.*y.*sin(x).*cos(y);
% Use surf to plot the surface
S=surf(x,y,z);
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

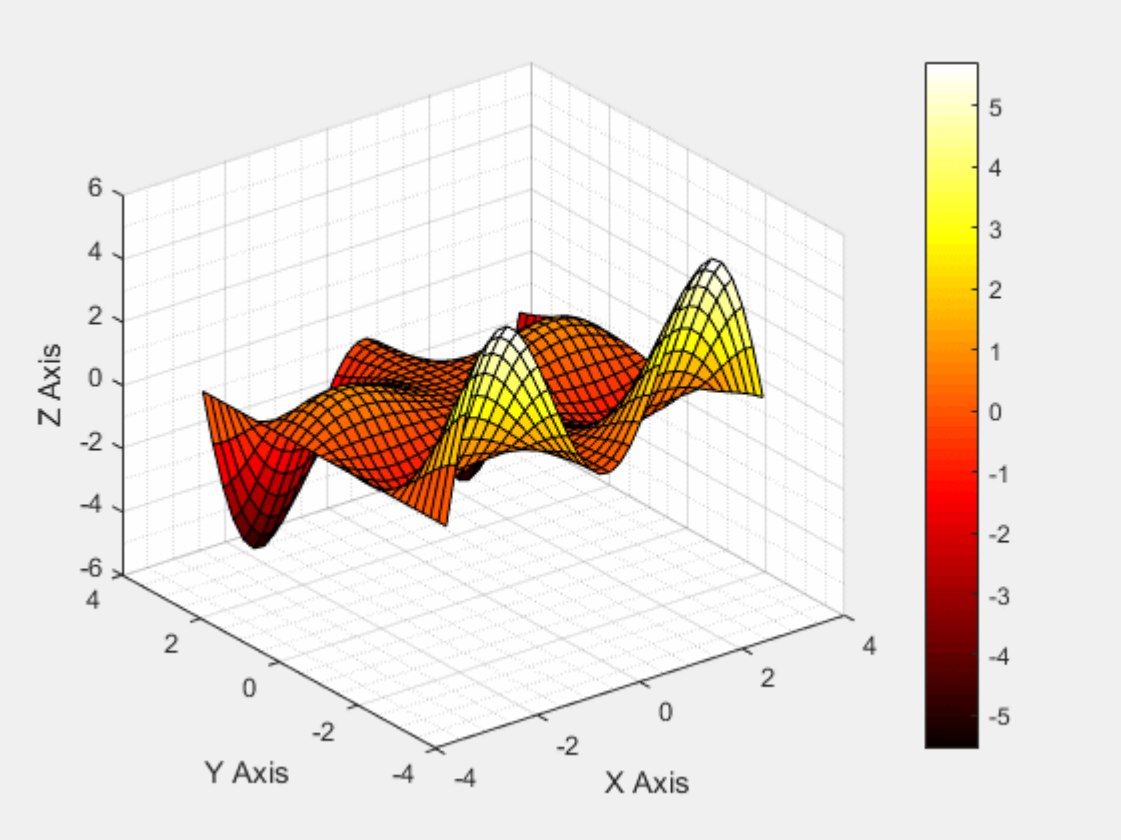


Рисунок 1

Теперь может случиться, что дополнительная информация связана со значениями `z` матрицы, и они хранятся в другой `(mxn)` матрице

Можно добавить эту дополнительную информацию на графике, изменив цвет поверхности.

Это позволит иметь вид 4D-графика: к трехмерному представлению поверхности,

сгенерированной первой `(mxn)` матрицей, четвертое измерение будет представлено данными, содержащимися во второй `(mxn)` матрице.

Такой сюжет можно создать, позвонив по `surf` с 4 входами:

```
surf(x,y,z,C)
```

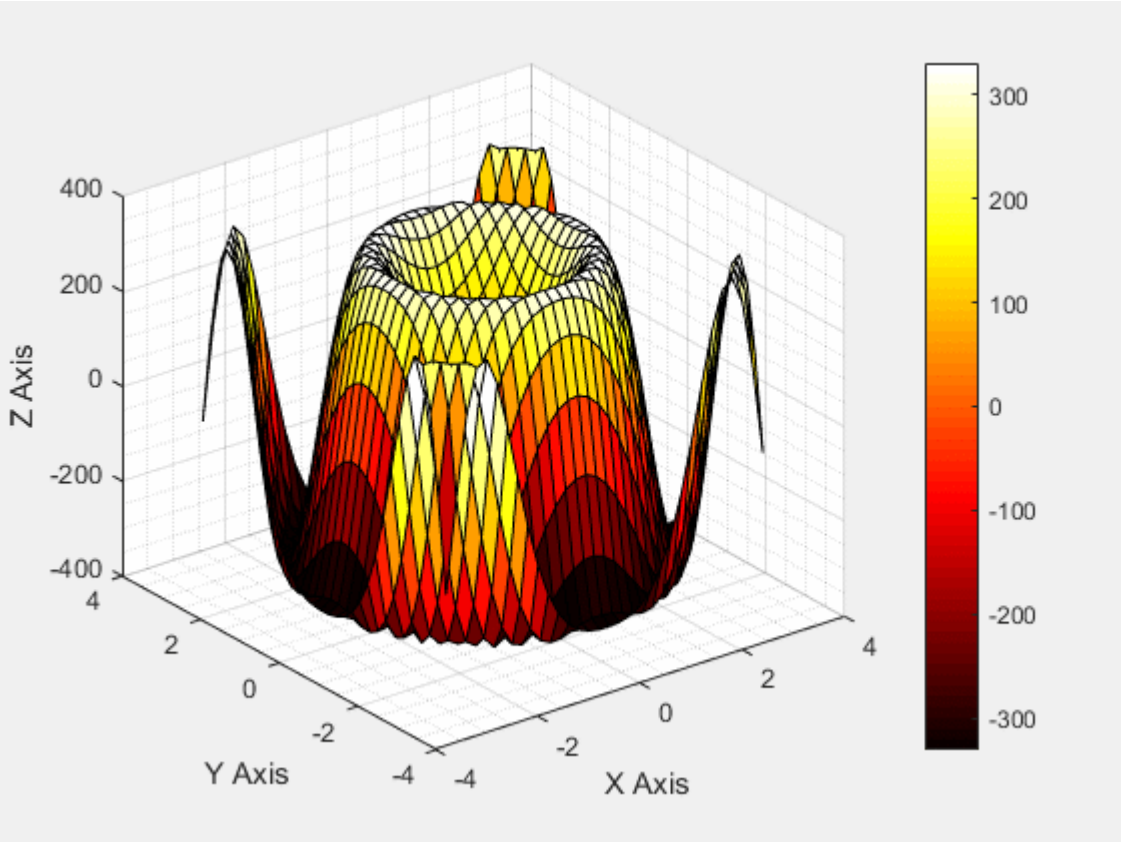
где параметр `C` является второй матрицей (которая должна быть одного размера `z`) и используется для определения цвета поверхности.

В следующем примере матрица `C` генерируется функцией:

```
C=10*sin(0.5*(x.^2.+y.^2))*33;
```

над интервалом `[-pi,pi]`

Поверхность, порожденная `C` ,



фигура 2

Теперь мы можем назвать `surf` с четырьмя входами:

```
figure
surf(x,y,z,C)
% shading interp
xlabel('X Axis');
ylabel('Y Axis');
```

```
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

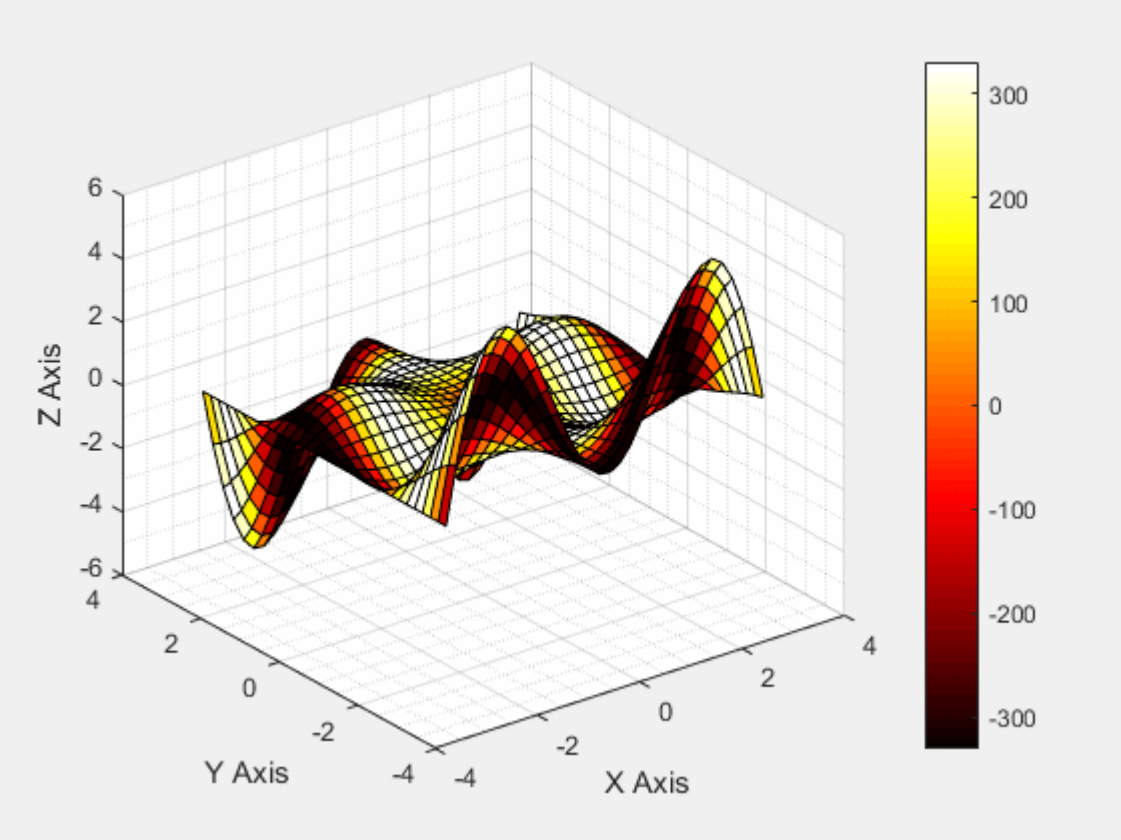


Рисунок 3

Сравнивая рисунки 1 и 3, мы можем заметить, что:

- форма поверхности соответствует значениям `z` (первая `(mxn)` матрица)
- цвет поверхности (и ее диапазон, заданный цветной панелью) соответствуют значениям `C` (первая `(mxn)` матрица)

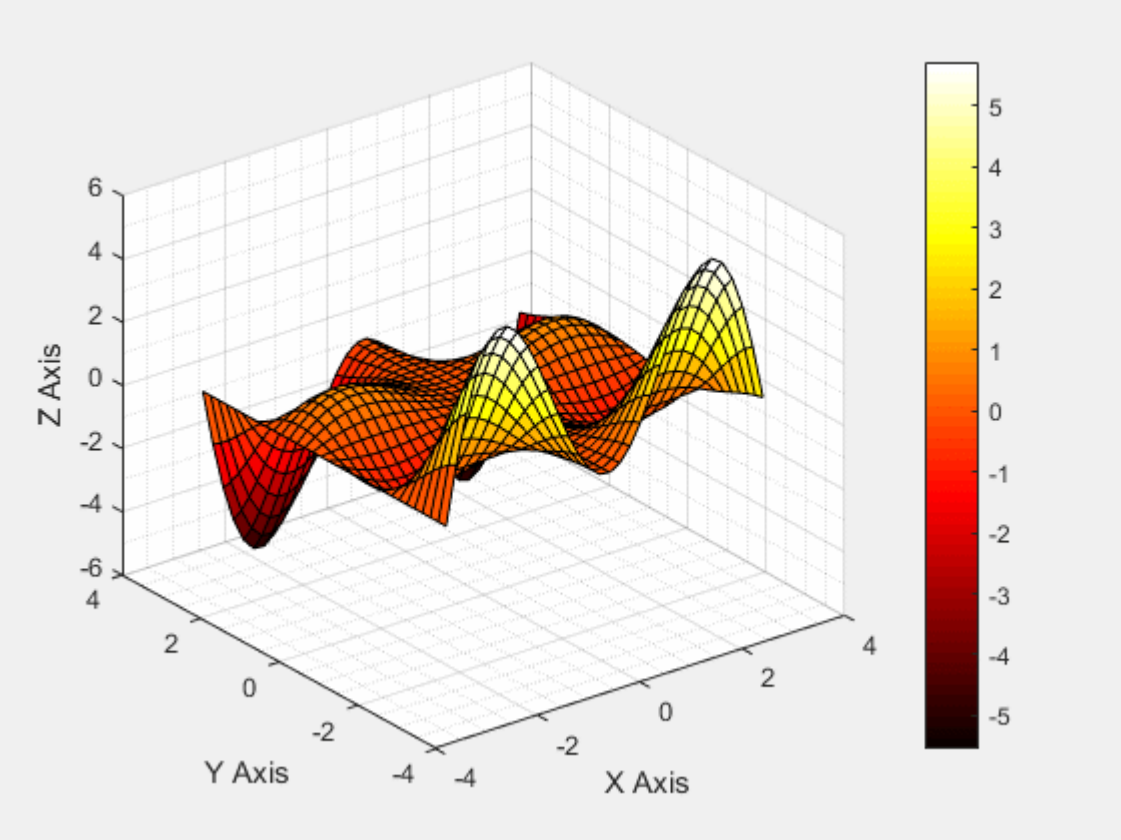
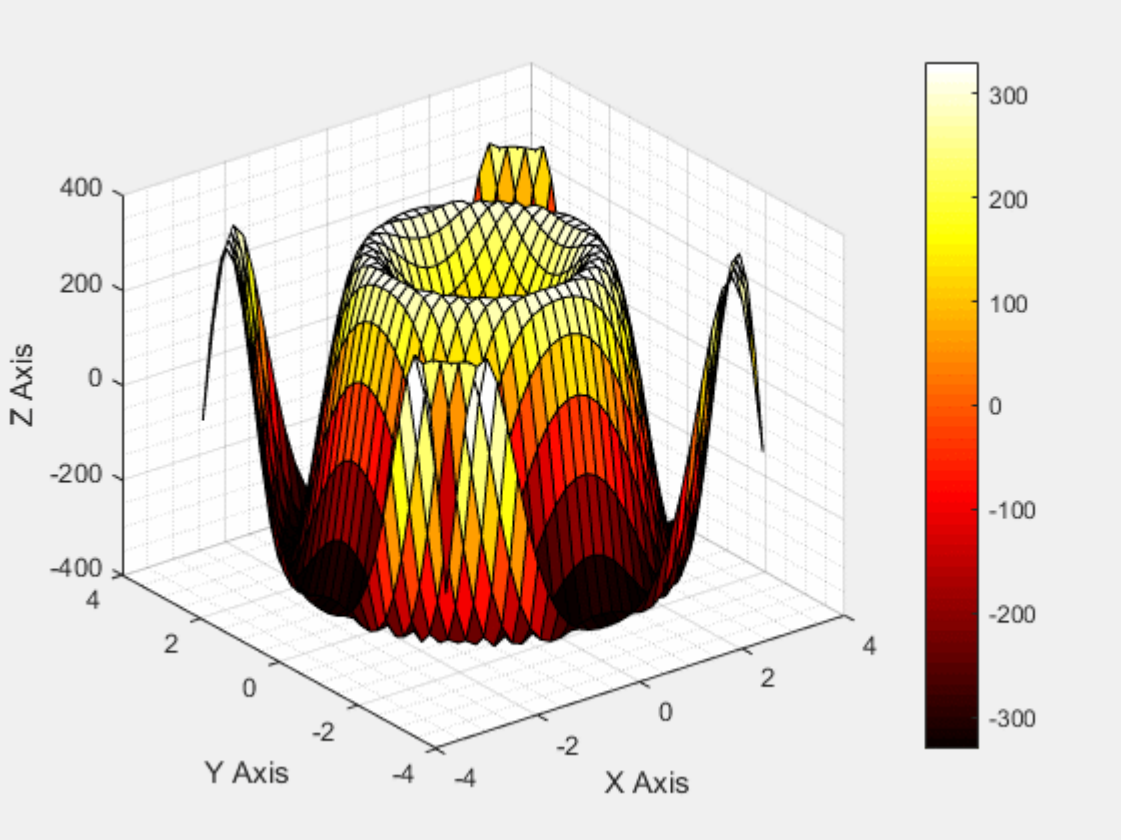


Рисунок 4

Конечно, можно поменять `z` и `C` на графике, чтобы иметь форму поверхности, заданную матрицей `C` и цвет, заданный матрицей `z` :

```
figure
surf(x,y,C,z)
% shading interp
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

и сравнить рисунок 2 с рисунком 4



Быстрый чертеж

Существует три основных способа сделать последовательный сюжет или анимацию: `plot(x,y)` , `set(h , 'XData' , y, 'YData' , y)` и `animatedline` `set(h , 'XData' , y, 'YData' , y)` . Если вы хотите, чтобы ваша анимация была гладкой, вам нужен эффективный рисунок, и три метода не эквивалентны.

```
% Plot a sin with increasing phase shift in 500 steps
x = linspace(0 , 2*pi , 100);

figure
tic
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);
    plot(x,y)
    drawnow
end
toc
```

Я получаю `5.278172 seconds` . Функция plot в основном удаляет и воссоздает объект линии каждый раз. Более эффективным способом обновления графика является использование свойств `XData` и `YData` объекта `Line` .

```
tic
h = [];   % Handle of line object
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);

    if isempty(h)
```

```
        % If Line still does not exist, create it
        h = plot(x,y);
    else
        % If Line exists, update it
        set(h , 'YData' , y)
    end
    drawnow
end
toc
```

Теперь я получаю `2.741996 seconds` , намного лучше!

`animatedline` - относительно новая функция, введенная в 2014b. Посмотрим, как это работает:

```
tic
h = animatedline;
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);
    clearpoints(h)
    addpoints(h , x , y)
    drawnow
end
toc
```

`3.360569 seconds` , не так хорошо, как обновление существующего сюжета, но все же лучше `plot(x,y)` .

Конечно, если вам нужно построить одну строку, как в этом примере, три метода почти эквивалентны и дают плавные анимации. Но если у вас более сложные сюжеты, обновление существующих объектов `Line` будет иметь значение.

Прочитайте Рисование онлайн: https://riptutorial.com/ru/matlab/topic/3978/рисование

глава 27: Управление окраской подзаголовков в Matlab

Вступление

Поскольку я боролся с этим более одного раза, и в Интернете не совсем ясно, что делать, я решил взять то, что там, добавив некоторые из моих собственных, чтобы объяснить, как создавать подзаголовки, которые имеют один цветной барабан, и они масштабируются в соответствии с ним.

Я тестировал это, используя последний Matlab, но я уверен, что он будет работать в более старых версиях.

замечания

Единственное, что вам нужно выработать самостоятельно - это позиционирование цветной панели (если вы хотите ее вообще отобразить). это будет зависеть от количества имеющихся у вас графиков и ориентации бара.

Позиция и размер определяются с помощью 4 параметров - x_start, y_start, x_width, y_width. График обычно масштабируется до нормализованных единиц, так что нижний левый угол соответствует (0,0), а верхний правый - (1,1).

Examples

Как это сделано

Это простой код, создающий 6 3d-подсетей и, в конце концов, синхронизирующий цвет, отображаемый в каждом из них.

```
c_fin = [0,0];
[X,Y] = meshgrid(1:0.1:10,1:0.1:10);

figure; hold on;
for i = 1 : 6
    Z(:,:,i) = i * (sin(X) + cos(Y));

    ax(i) = subplot(3,2,i); hold on; grid on;
    surf(X, Y, Z(:,:,i));
    view(-26,30);
    colormap('jet');
    ca = caxis;
    c_fin = [min(c_fin(1),ca(1)), max(c_fin(2),ca(2))];
end
```

```
%%you can stop here to see how it looks before we color-manipulate

c = colorbar('eastoutside');
c.Label.String = 'Units';
set(c, 'Position', [0.9, 0.11, 0.03, 0.815]); %%you may want to play with these values
pause(2); %%need this to allow the last image to resize itself before changing its axes
for i = 1 : 6
    pos=get(ax(i), 'Position');
    axes(ax(i));
    set(ax(i), 'Position', [pos(1) pos(2) 0.85*pos(3) pos(4)]);
    set(ax(i),'Clim', c_fin); %%this is where the magic happens
end
```

Прочитайте Управление окраской подзаголовков в Matlab онлайн:
https://riptutorial.com/ru/matlab/topic/10913/управление-окраской-подзаголовков-в-matlab

глава 28: условия

Синтаксис

- если *выражение* ... end
- если *выражение* ... else ... end
- если *выражение* ... elseif *выражение* ... end
- if *выражение* ... elseif *expression* ... else ... end

параметры

параметр	Описание
выражение	выражение, имеющее логический смысл

Examples

Условие IF

Условия являются фундаментальной частью почти любой части кода. Они используются для выполнения некоторых частей кода только в некоторых ситуациях, но не для других. Давайте рассмотрим базовый синтаксис:

```
a = 5;
if a > 10    % this condition is not fulfilled, so nothing will happen
    disp('OK')
end

if a < 10    % this condition is fulfilled, so the statements between the if...end are
executed
    disp('Not OK')
end
```

Выход:

```
Not OK
```

В этом примере мы видим, что `if` состоит из двух частей: условия и кода для запуска, если условие истинно. Код - это все, написанное после условия и до его `end if` . Первое условие не было выполнено, и, следовательно, код внутри него не был выполнен.

Вот еще один пример:

```
a = 5;
```

```
if a ~= a+1        % "~=" means "not equal to"
    disp('It''s true!') % we use two apostrophes to tell MATLAB that the ' is part of the
string
end
```

Приведенное выше условие всегда будет истинным и будет отображать вывод. `It's true!` ,

Мы также можем написать:

```
a = 5;
if a == a+1    % "==" means "is equal to", it is NOT the assignment ("=") operator
    disp('Equal')
end
```

На этот раз условие всегда ложно, поэтому мы никогда не получим результат `Equal` .

Хотя условия, которые всегда бывают истинными или ложными, мало используются, потому что, если они всегда являются ложными, мы можем просто удалить эту часть кода, и если они всегда верны, то условие не требуется.

Условие IF-ELSE

В некоторых случаях мы хотим запустить альтернативный код, если условие ложно, для этого мы используем необязательную часть `else` :

```
a = 20;
if a < 10
    disp('a smaller than 10')
else
    disp('a bigger than 10')
end
```

Здесь мы видим, что, поскольку `a` не меньше `10` вторая часть кода, после выполнения `else` и мы получаем выход `a bigger than 10` . Теперь давайте посмотрим на другую попытку:

```
a = 10;
if a > 10
    disp('a bigger than 10')
else
    disp('a smaller than 10')
end
```

В этом примере показано, что мы не проверяли, действительно ли `a` действительно меньше 10, и мы получаем неправильное сообщение, потому что условие проверяет только выражение как оно есть, а ЛЮБОЙ случай, который не равен true (`a = 10`), приведет к вторая часть должна быть выполнена.

Этот тип ошибок является очень распространенной ловушкой для начинающих и опытных программистов, особенно когда условия становятся сложными и их всегда следует помнить

Условие IF-ELSEIF

Используя `else` мы можем выполнить некоторую задачу, когда условие не выполняется. Но что, если мы хотим проверить второе условие в случае, если первый был ложным. Мы можем сделать это так:

```
a = 9;
if mod(a,2)==0   % MOD - modulo operation, return the remainder after division of 'a' by 2
    disp('a is even')
else
    if mod(a,3)==0
        disp('3 is a divisor of a')
    end
end

OUTPUT:
3 is a divisor of a
```

Это также называется «вложенным условием» , но здесь мы имеем спецификацию, которая может улучшить читаемость кода и уменьшить вероятность ошибки anr - мы можем написать:

```
a = 9;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
end

OUTPUT:
3 is a divisor of a
```

используя `elseif` мы можем проверить другое выражение в одном блоке условий, и это не ограничивается одной попыткой:

```
a = 25;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
end

OUTPUT:
5 is a divisor of a
```

Следует проявлять `elseif` осторожность при выборе использования `elseif` в строке, так как **только один** из них будет выполнен из всего блока `if` до `end` . Итак, в нашем примере, если мы хотим отобразить все делители `a` (из тех, которые мы явно проверяем), приведенный выше пример не будет хорош:

```
a = 15;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
end

OUTPUT:
3 is a divisor of a
```

не только 3, но и 5 - делитель 15, но часть, которая проверяет деление на 5, не достигается, если какое-либо из выражений выше оно было истинным.

Наконец, мы можем добавить `else` один (и только **один**) после всех условий `elseif` для выполнения кода, когда ни одно из приведенных выше условий не выполняется:

```
a = 11;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
else
    disp('2, 3 and 5 are not divisors of a')
end

OUTPUT:
2, 3 and 5 are not divisors of a
```

Вложенные условия

Когда мы используем условие в другом условии, мы говорим, что условия «вложены». Один частный случай вложенных условий задается опцией `elseif` , но существует множество других способов использования вложенных условий. Давайте рассмотрим следующий код:

```
a = 2;
if mod(a,2)==0    % MOD - modulo operation, return the remainder after division of 'a' by 2
    disp('a is even')
    if mod(a,3)==0
        disp('3 is a divisor of a')
        if mod(a,5)==0
            disp('5 is a divisor of a')
        end
    end
else
    disp('a is odd')
end
```

При `a=2` выход будет `a is even` , что является правильным. При `a=3` выход будет `a is odd` , что тоже правильно, но пропускает проверку, если 3 является делителем `a` . Это связано с тем, что условия вложены, так что **только** если первое `true` , то мы переходим во внутреннее, а

если `a` нечетно, ни одно из внутренних условий не проверяется. Это несколько противоположно использованию `elseif` где только если первое условие `false` чем мы проверяем следующий. Как насчет проверки деления на 5? только число, которое имеет 6 как делитель (оба и 2), будет проверено для деления на 5, и мы можем проверить и увидеть, что при `a=30` выход:

```
a is even
3 is a divisor of a
5 is a divisor of a
```

Мы также должны заметить две вещи:

1. Положение `end` в нужном месте для каждого , `if` имеет решающее значение для набора условий для работы , как и следовало ожидать, так отступы больше , чем хорошая рекомендация здесь.
2. Позиция инструкции `else` также имеет решающее значение, поскольку нам нужно знать, в каком `if` (и их может быть несколько) мы хотим что-то сделать, если выражение if `false` .

Давайте посмотрим на другой пример:

```
for a = 5:10    % the FOR loop execute all the code within it for every a from 5 to 10
    ch = num2str(a);    % NUM2STR converts the integer a to a character
    if mod(a,2)==0
        if mod(a,3)==0
            disp(['3 is a divisor of ' ch])
        elseif  mod(a,4)==0
            disp(['4 is a divisor of ' ch])
        else
            disp([ch ' is even'])
        end
    elseif mod(a,3)==0
        disp(['3 is a divisor of ' ch])

    else
        disp([ch ' is odd'])
    end
end
```

И выход будет:

```
5 is odd
3 is a divisor of 6
7 is odd
4 is a divisor of 8
3 is a divisor of 9
10 is even
```

мы видим, что у нас есть только 6 строк для 6 чисел, потому что условия вложены таким образом, чтобы обеспечить только одну печать на номер, а также (хотя не видно непосредственно из вывода) никаких дополнительных проверок не выполняется, поэтому

если число даже не имеет смысла проверять, является ли 4 одним из его делителей.

Прочитайте условия онлайн: https://riptutorial.com/ru/matlab/topic/3806/условия

глава 29: Установить операции

Синтаксис

1. C = объединение (A, B);
2. C = пересекаются (A, B);
3. C = setdiff (A, B);
4. a = ismember (A, x);

параметры

параметр	подробности
A, B	множеств, возможно, матриц или векторов
Икс	возможный элемент множества

Examples

Операции с элементарным множеством

Операции с элементарным множеством можно выполнять с помощью Matlab. Предположим, мы дали два вектора или массивы

```
A = randi([0 10],1,5);
B = randi([-1 9], 1,5);
```

и мы хотим найти все элементы, которые находятся в `A` и `B` Для этого мы можем использовать

```
C = intersect(A,B);
```

`C` будет включать все числа, которые являются частью `A` и частью `B` Если мы также хотим найти позицию этих элементов, мы называем

```
[C,pos] = intersect(A,B);
```

`pos` - это положение этих элементов, такое, что `C == A(pos)` .

Другой основной операцией является объединение двух множеств

```
D = union(A,B);
```

Herby содержит `D` всех элементов `A` и `B`

Обратите внимание, что `A` и `B` здесь рассматриваются как наборы, что означает, что не имеет значения, как часто элемент является частью `A` или `B` Чтобы уточнить это, можно проверить `D == union(D,C)` .

Если мы хотим получить данные, находящиеся в «A», но не в «B», мы можем использовать следующую функцию

```
E = setdiff(A,B);
```

Мы хотим еще раз отметить, что это такие множества, что выполняется следующее утверждение `D == union(E,B)` .

Предположим, мы хотим проверить, если

```
x = randi([-10 10],1,1);
```

является элементом либо `A` либо `B` мы можем выполнить команду

```
a = ismember(A,x);
b = ismember(B,x);
```

Если `a==1` то `x` является элементом `A` а `x` является элементом `a==0` . То же самое касается `B` Если `a==1 && b==1` `x` также является элементом `C` Если `a == 1 || b == 1` `x` является элементом `D` и если `a == 1 || b == 0` это также элемент `E`

Прочитайте Установить операции онлайн: https://riptutorial.com/ru/matlab/topic/3242/установить-операции

глава 30: Утилиты для программирования

Examples

Простой таймер в MATLAB

Ниже приведен таймер, который срабатывает с фиксированным интервалом. `Timerfcn` -аут определяется `Period` и вызывает обратный вызов, определенный `Timerfcn` по `Timerfcn` таймаута.

```
t = timer;
t.TasksToExecute = Inf;
t.Period = 0.01; % timeout value in seconds
t.TimerFcn = @(myTimerObj, thisEvent)disp('hello'); % timer callback function
t.ExecutionMode = 'fixedRate';
start(t)
pause(inf);
```

Прочитайте Утилиты для программирования онлайн:
https://riptutorial.com/ru/matlab/topic/1655/утилиты-для-программирования

глава 31: Финансовые приложения

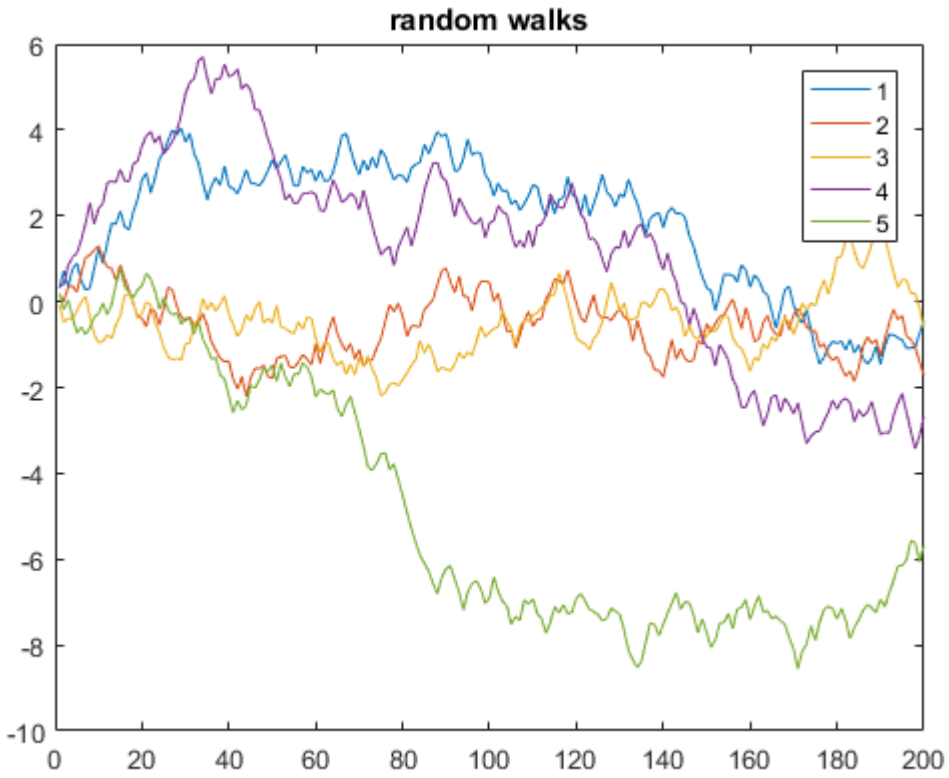
Examples

Случайная прогулка

Ниже приведен пример, который отображает 5 одномерных случайных блужданий по 200 шагов:

```
y = cumsum(rand(200,5) - 0.5);

plot(y)
legend('1', '2', '3', '4', '5')
title('random walks')
```

В приведенном выше коде `y` представляет собой матрицу из 5 столбцов, каждая из которых имеет длину 200. Так как `x` опущен, по умолчанию они соответствуют номерам строк `y` (что эквивалентно использованию `x=1:200` в качестве оси x). Таким образом, функция `plot` отображает несколько y-векторов в отношении одного и того же вектора x, каждый из которых автоматически использует другой цвет.



Одномерное геометрическое броуновское движение

Динамика геометрического броуновского движения (GBM) описывается следующим

стохастическим дифференциальным уравнением (SDE):

$$dS_t = \mu S_t\,dt + \sigma S_t\,dW_t$$

Я могу использовать **точное** решение для SDE

$$S_t = S_0 \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right)$$

для генерации путей, следующих за GBM.

Учитывая суточные параметры для моделирования в течение года

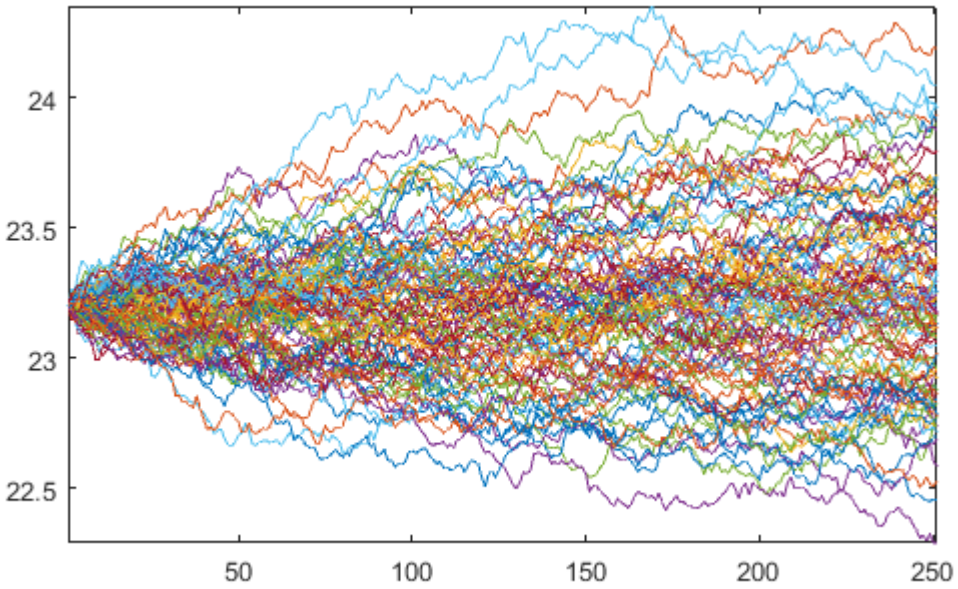
```
mu     = 0.08/250;
sigma  = 0.25/sqrt(250);
dt     = 1/250;
npaths = 100;
nsteps = 250;
S0     = 23.2;
```

мы можем получить броуновское движение (BM) `W` начинающееся с 0, и использовать его для получения GBM, начиная с `S0`

```
% BM
epsilon = randn(nsteps, npaths);
W       = [zeros(1,npaths); sqrt(dt)*cumsum(epsilon)];

% GBM
t = (0:nsteps)'*dt;
Y = bsxfun(@plus, (mu-0.5*sigma.^2)*t, sigma*W);
Y = S0*exp(Y);
```

Который создает пути

```
plot(Y)
```



Прочитайте Финансовые приложения онлайн: https://riptutorial.com/ru/matlab/topic/1197/финансовые-приложения

глава 32: функции

Examples

Базовый пример

Следующий сценарий MATLAB показывает, как определить и вызвать базовую функцию:

myFun.m :

```
function [out1] = myFun(arg0, arg1)
    out1 = arg0 + arg1;
end
```

терминал :

```
>> res = myFun(10, 20)

res =

    30
```

Несколько выходов

Следующий сценарий MATLAB показывает, как возвращать несколько выходов в одну функцию:

myFun.m :

```
function [out1, out2, out3] = myFun(arg0, arg1)
    out1 = arg0 + arg1;
    out2 = arg0 * arg1;
    out3 = arg0 - arg1;
end
```

терминал :

```
>> [res1, res2, res3] = myFun(10, 20)

res1 =

        30

res2 =

        200

res3 =
        -10
```

Однако MATLAB вернет только первое значение при присвоении одной переменной

```
    >> res = myFun(10, 20)

    res =

            30
```

В следующем примере показано, как получить определенный результат

```
    >> [~, res] = myFun(10, 20)

    res =

            200
```

Наргин, Наргут

В теле функции `nargin` и `nargout` указываются соответственно фактическое количество входных и выходных данных, `nargout` в вызов.

Мы можем, например, управлять исполнением функции на основе количества предоставленных входных данных.

myVector.m :

```
function [res] = myVector(a, b, c)
    % Roughly emulates the colon operator

    switch nargin
        case 1
            res = [0:a];
        case 2
            res = [a:b];
        case 3
            res = [a:b:c];
        otherwise
            error('Wrong number of params');
    end
end
```

Терминал:

```
>> myVector(10)

ans =

    0    1    2    3    4    5    6    7    8    9   10

>> myVector(10, 20)

ans =

   10   11   12   13   14   15   16   17   18   19   20
```

```
>> myVector(10, 2, 20)

ans =

   10   12   14   16   18   20
```

Аналогичным образом мы можем управлять выполнением функции на основе количества выходных параметров.

myIntegerDivision :

```
function [qt, rm] = myIntegerDivision(a, b)
    qt = floor(a / b);

    if nargout == 2
        rm = rem(a, b);
    end
end
```

терминал :

```
>> q = myIntegerDivision(10, 7)

q = 1

>> [q, r] = myIntegerDivision(10, 7)

q = 1
r = 3
```

Прочитайте функции онлайн: https://riptutorial.com/ru/matlab/topic/5659/функции

глава 33: Функции документирования

замечания

- Текст справки может быть расположен до или после `function` строки, если не существует кода между функциональной линией и началом текста справки.
- Капитализация имени функции смещает только имя и не требуется.
- Если строка добавляется вместе с `See also` , то все имена в строке, которые соответствуют имени класса или функции на пути поиска, будут автоматически ссылаться на документацию этого класса / функции.
    - Глобальные функции можно отнести здесь, прежде чем их имя будет обозначаться символом `\` . В противном случае имена сначала будут пытаться и разрешать функции-члены.
- Разрешены гиперссылки формы `<a href="matlab:web('url')">Name</a>` .

Examples

Простая функциональная документация

```
function output = mymult(a, b)
% MYMULT Multiply two numbers.
%   output = MYMULT(a, b) multiplies a and b.
%
%   See also fft, foo, sin.
%
%   For more information, see <a href="matlab:web('https://google.com')">Google</a>.
    output = a * b;
end
```

`help mymult` затем предоставляет:

> **mymult** Умножьте два числа.
>
> output = **mymult** (a, b) умножает a и b.
>
> См. Также fft, foo, sin.
>
> Для получения дополнительной информации см. Google .

`fft` и `sin` автоматически ссылаются на соответствующий текст справки, а Google - это ссылка на google.com . `foo` не будет ссылаться на какую-либо документацию, если нет документальной функции / класса по имени `foo` на пути поиска.

Локальная функциональная документация

В этом примере документацию для локальной функции `baz` (определенную в `foo.m`) можно получить либо по полученной ссылке в `help foo` , либо через `help foo>baz` .

```
function bar = foo
%This is documentation for FOO.
% See also foo>baz

% This wont be printed, because there is a line without % on it.
end

function baz
% This is documentation for BAZ.
end
```

Получение сигнатуры функции

Часто бывает полезно, чтобы MATLAB печатал 1- ю строку функции, так как обычно она содержит подпись функции, включая входы и выходы:

```
dbtype <functionName> 1
```

Пример:

```
>> dbtype fit 1

1  function [fitobj,goodness,output,warnstr,errstr,convmsg] =
fit(xdatain,ydatain,fittypeobj,varargin)
```

Документирование функции с помощью сценария примера

Чтобы документировать функцию, часто полезно иметь пример скрипта, который использует вашу функцию. Функция публикации в Matlab затем может быть использована для создания файла справки со встроенными изображениями, кодом, ссылками и т. Д. Синтаксис для документирования вашего кода можно найти здесь .

Функция Эта функция использует «исправленный» FFT в Matlab.

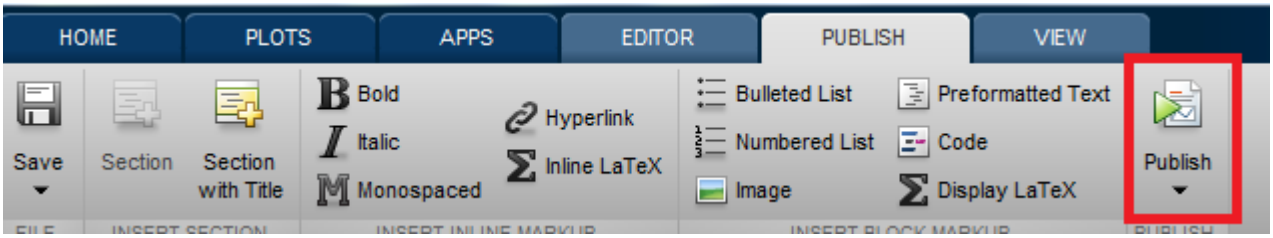
```
function out_sig = myfft(in_sig)

out_sig = fftshift(fft(ifftshift(in_sig)));

end
```

Пример сценария Это отдельный сценарий, который объясняет входы, выходы и дает пример, объясняющий, почему необходима коррекция. Благодаря У, Кан, оригинальному автору этой функции.

```
%% myfft
% This function uses the "proper" fft in matlab. Note that the fft needs to
```

```
% be multiplied by dt to have physical significance.
% For a full description of why the FFT should be taken like this, refer
% to: Why_use_fftshift(fft(fftshift(x)))__in_Matlab.pdf included in the
% help folder of this code. Additional information can be found:
% <https://www.mathworks.com/matlabcentral/fileexchange/25473-why-use-fftshift-fft-fftshift-x-
---in-matlab-instead-of-fft-x-->
%
%% Inputs
% *in_sig* - 1D signal
%
%% Outputs
% *out_sig* - corrected FFT of *in_sig*
%
%% Examples
% Generate a signal with an analytical solution. The analytical solution is
% then compared to the fft then to myfft. This example is a modified
% version given by Wu, Kan given in the link aboce.
%%
% Set parameters
fs = 500;            %sampling frequency
dt = 1/fs;           %time step
T=1;                 %total time window
t = -T/2:dt:T/2-dt; %time grids
df = 1/T;            %freq step
Fmax = 1/2/dt;       %freq window
f=-Fmax:df:Fmax-df; %freq grids, not used in our examples, could be used by plot(f, X)
%%
% Generate Gaussian curve
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);     %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                    %Create Gaussian Curve
%%
% Generate Analytical solution
Xan = A*sqrt(2*pi)*exp(-2*pi^2*f.^2*A^2); %X(f), real part of the analytical Fourier transform
of x(t)

%%
% Take FFT and corrected FFT then compare
Xfft = dt *fftshift(fft(x));     %FFT
Xfinal = dt * myfft(x);          %Corrected FFT
hold on
plot(f,Xan);
plot(f,real(Xfft));
plot(f,real(Xfinal),'ro');
title('Comparison of Corrected and Uncorrected FFT');
legend('Analytical Solution','Uncorrected FFT','Corrected FFT');
xlabel('Frequency'); ylabel('Amplitude');
DT = max(f) - min(f);
xlim([-DT/4,DT/4]);
```

Выходной параметр «Публикация» можно найти на вкладке «Опубликовать», выделенной на рисунке « Простая функциональная документация» ниже.



Matlab запустит сценарий и сохранит отображаемые изображения, а также текст, сгенерированный командной строкой. Вывод может быть сохранен во многих форматах, включая HTML, Latex и PDF.

Результат приведенного выше примера скрипта можно увидеть на изображении ниже.

myfft

This function uses the "proper" fft in matlab. Note that the fft needs to be multiplied by dt to have physical significance. For a full description of why the FFT should be taken like this, refer to: Why_use_fftshift(fft(fftshift(x)))__in_Matlab.pdf included in the help folder of this code. Additional information can be found: https://www.mathworks.com/matlabcentral/fileexchange/25473-why-use-fftshift-fft-fftshift-x----in-matlab-instead-of-fft-x--

Contents

Inputs

in_sig - 1D signal

Outputs

out_sig - corrected FFT of **in_sig**

Examples

Generate a signal with an analytical solution. The analytical solution is then compared to the fft then to myfft. This example is a modified version given by Wu, Kan given in the link aboce.

Set parameters

```
fs = 500;            %sampling frequency
dt = 1/fs;           %time step
T=1;                 %total time window
t = -T/2:dt:T/2-dt; %time grids
df = 1/T;            %freq step
Fmax = 1/2/dt;       %freq window
f=-Fmax:df:Fmax-df; %freq grids, not used in our examples, could be used by plot(f, X)
```

Generate Gaussian curve

```
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);    %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                   %Create Gaussian Curve
```

Generate Analytical solution

```
Xan = A*sqrt(2*pi)*exp(-2*pi^2*f.^2*A^2); %X(f), real part of the analytical Fourier transform of x(t)
```

Take FFT and corrected FFT then compare

```
Xfft = dt *fftshift(fft(x));     %FFT
Xfinal = dt * myfft(x);          %Corrected FFT
hold on
plot(f,Xan);
plot(f,real(Xfft));
plot(f,real(Xfinal),'ro');
title('Comparison of Corrected and Uncorrected FFT');
legend('Analytical Solution','Uncorrected FFT','Corrected FFT');
xlabel('Frequency'); ylabel('Amplitude');
DT = max(f) - min(f);
xlim([-DT/4,DT/4]);
```


функции-документирования

глава 34: Чтение больших файлов

Examples

TextScan

Предположим, что вы форматировали данные в большом текстовом файле или строке, например

```
Data,2015-09-16,15:41:52;781,780.000000,0.0034,2.2345
Data,2015-09-16,15:41:52;791,790.000000,0.1255,96.5948
Data,2015-09-16,15:41:52;801,800.000000,1.5123,0.0043
```

можно использовать `textscan` чтобы прочитать это довольно быстро. Для этого нужно получить идентификатор файла текстового файла с помощью функции `fopen` :

```
fid = fopen('path/to/myfile');
```

Предположим, что для данных в этом примере мы хотим игнорировать первый столбец «Данные», читать дату и время в виде строк и читать остальные столбцы как удвоенные, т. Е.

```
 Data   ,   2015-09-16   , 15:41:52;801   , 800.000000   , 1.5123   ,   0.0043
ignore       string          string          double       double      double
```

Для этого позвоните:

```
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

Звездочка в `%*s` означает «игнорировать этот столбец». `%s` означает «интерпретировать как строку». `%f` означает «интерпретировать как удваивает (плавает)». Наконец, `'Delimiter',','` указывает, что все запятые должны интерпретироваться как разделитель между каждым столбцом.

Подводить итоги:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

теперь `data` содержат массив ячеек с каждым столбцом в ячейке.

Строки даты и времени для числового массива быстро

Преобразование строк даты и времени в числовые массивы можно выполнять с помощью `datenum`

, хотя это может занять половину времени при чтении большого файла данных.

Рассмотрим данные в примере **Textscan** . Кроме того, используя textscan и интерпретируя дату и время как целые числа, они могут быть быстро преобразованы в числовой массив.

Т.е. строка в данных примера будет интерпретироваться как:

```
 Data , 2015  - 09  -  16  ,  15  :  41  :  52  ;  801 , 800.000000 , 1.5123 , 0.0043
ignore double double double double double double double    double     double   double
```

который будет читаться так:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %f %f %f %f %f %f %f %f %f','Delimiter',',-:;');
fclose(fid);
```

Сейчас:

```
y = data{1};           % year
m = data{2};           % month
d = data{3};           % day
H = data{4};           % hours
M = data{5};           % minutes
S = data{6};           % seconds
F = data{7};           % milliseconds

% Translation from month to days
ms = [0,31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334];

n = length(y);         % Number of elements
Time = zeros(n,1);     % Declare numeric time array

% Algorithm for calculating numeric time array
for k = 1:n
    Time(k) = y(k)*365 + ms(m(k)) + d(k) + floor(y(k)/4)...
              - floor(y(k)/100) + floor(y(k)/400) + (mod(y(k),4)~=0)...
              - (mod(y(k),100)~=0) + (mod(y(k),400)~=0)...
              + (H(k)*3600 + M(k)*60 + S(k) + F(k)/1000)/86400 + 1;
end
```

Использование `datenum` на 566 678 элементах потребовало 6626570 секунд, тогда как для метода выше требовалось 0,043434 секунды, т.е. 0,73% времени для `datenum` или ~ 137 раз быстрее.

Прочитайте Чтение больших файлов онлайн: https://riptutorial.com/ru/matlab/topic/9023/чтение-больших-файлов

кредиты

S. No	Главы	Contributors
1	Начало работы с языком MATLAB	adjpayot, Amro, chrisb2244, Christopher Creutzig, Community, Dan, Dev-iL, DVarga, EBH, Erik, excaza, flawr, Franck Dernoncourt, fyrepenguin, GameOfThrows, H. Pauwelyn, honi, Landak, Lior, Matt, Mikhail_Sam, Mohsen Nosratinia, Sam Roberts, Shai, Tyler
2	Введение в MEX API	Amro, Ander Biguri, Kamiccolo, Matt, mike
3	Векторизация	Alexander Korovin, Ander Biguri, Dan, Dev-iL, EBH, Landak, Matt, Oleg, Shai, Tyler
4	Графика: 2D и 3D преобразования	itzik Ben Shabat, Royi
5	Графика: 2D-линии	Amro, Celdor, EBH, Erik, Matt, Oleg, StefanM
6	Для петель	agent_C.Hdj, drhagen, EBH, Tyler
7	Инициализация матриц или массивов	Parag S. Chandakkar, rajah9, Théo P.
8	интеграция	StefanM
9	Интерполяция с MATLAB	Royi, StefanM
10	Использование последовательных портов	Abdul Rehman, Ander Biguri, Hoki, Kenn Sebesta, mhopeng, Oleg
11	Использование функции `accumarray()`	Dev-iL, Royi
12	Использование функций с логическим выходом	Landak, Mikhail_Sam, Mohsen Nosratinia, S. Radev, Trilarion
13	Лучшие практики MATLAB	il_raffa, Malick, MayeulC, McLemon, mnoronha, NKN, rayryeng, Sardar Usama, Thierry Dalon

14	Матричные разложения	StefanM
15	Многопоточность	Adriaan, daren shan, Franck Dernoncourt, Hardik_Jain, Jim, Peter Mortensen
16	Недокументированные функции	Amro, codeaviator, Dev-iL, Erik, matlabgui, thewaywewalk
17	Обработка изображения	A.Youssouf, Ander Biguri, Cape Code, girish_m, Royi, Shai, Trilarion
18	Общие ошибки и ошибки	Amro, Ander Biguri, Dev-iL, EBH, edwinksl, erfan, Franck Dernoncourt, Hoki, Landak, Malick, Matt, Mikhail_Sam, Mohsen Nosratinia, nahomyaja, NKN, Oleg, R. Joiny, rafa, rayryeng, Rody Oldenhuis, S. Radev, Sardar Usama, strpeter , Suever, Thierry Dalon, Tim, Umar
19	Объектно-ориентированное программирование	alexforrence, Celdor, daren shan, Dev-iL, jenszvs, Mohsen Nosratinia, Trogdor
20	отладка	Dev-iL, DVarga, jenszvs, Justin
21	Полезные трюки	Celdor, EBH, Erik, fyrepenguin, Malick
22	Пользовательские интерфейсы MATLAB	Dev-iL, Hoki, Royi, Suever, Zep
23	Преобразования Фурье и обратные преобразования Фурье	GameOfThrows, Landak
24	Производительность и бенчмаркинг	Celdor, daleonpz, Dev-iL, il_raffa, Oleg, pseudoDust, thewaywewalk
25	Реальные решатели дифференциальных уравнений (ОДУ)	Royi, StefanM
26	Рисование	il_raffa, nitsua60, NKN, thewaywewalk, Trilarion, Zep
27	Управление окраской подзаголовков в Matlab	Noa Regev
28	условия	ammportal, EBH

29	Установить операции	Shai, StefanM
30	Утилиты для программирования	Celdor, The Vivandiere
31	Финансовые приложения	Amro, Franck Dernoncourt, Mikhail_Sam, Oleg, Royi, StefanM
32	функции	Batsu
33	Функции документирования	alexforrence, Ander Biguri, ceiltechbladhm, Dev-iL, Eric, excaza
34	Чтение больших файлов	JCKaz