Working with Arrays

Outline

Numeric arrays

- Numeric data types
- Creating multidimensional arrays
- Dimensions have meaning

Indexing

- Syntax
- Examples with meaningful dimensions
- `squeeze()` function
- Transpose operation

Logicals

- Conditional operators
- Boolean operators
- Logical indexing
- `find()` function
- `nnz()` function

Assignment Overview

Data Types in MATLAB

- Numbers (numeric classes)
- Booleans, aka True/False (logical classes)
- Characters and Strings
- Cell arrays
- Structures
- Classes/Objects

Array Size Examples

Scalar: size is (1, 1) or 1 row, 1 column

23

Row vector: size is (1, 5) or 1 row, 5 columns

23	15	1	2.4	-1.1

Column vector: size is (5, 1) or 5 rows, 1 columns

23
15
1
2.4
-1.1

Colon notation

Useful for creating evenly sampled points on a number line.

Syntax:

```
start:end
```

or

```
start:step:end
```

Colon notation

Useful for creating evenly sampled points on a number line.

Examples:

```
1:5 == [1 2 3 4 5]

12:14 == [12 13 14]

0:2:10 == [0 2 4 6 8 10]

5:-1:1 == [5 4 3 2 1]
```

Syntax for creating 2d arrays

Syntax

- Mainly useful for working on the command line
- Enclose everything in square brackets [ ]

Spaces or commas between values mean put on same row:

[2 3 4] and [2, 3, 4] both mean

2	3	4

Semicolons between values mean put on next row:

[2; 3; 4] means

2
3
4

Syntax for creating 2d arrays

Combine spaces or commas with semicolons to specify a full 2d array:

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

means

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

Just make sure you have the same number of items in each row!

Demo: Numerical Arrays

Array Indexing

Indexing

- Syntax
- Examples with meaningful dimensions
- `squeeze()` function
- Transpose operation

Indexing

Indexing allows you to select specific elements based on their location

```
a = [23 15 1 2.4 -1.1]
```

23	15	1	2.4	-1.1

`a(1) ==` $\boxed{23}$

`a(2) ==` $\boxed{15}$

Indexing

Indexing allows you to select specific elements based on their location

```
a = [23 15 1 2.4 -1.1]
```

23	15	1	2.4	-1.1

```
a([1 2 3]) ==
```

23	15	1

```
a([2 4 5]) ==
```

15	2.4	-1.1

Indexing

Indexing allows you to select specific elements based on their location

```
a = [23 15 1 2.4 -1.1]
```

23	15	1	2.4	-1.1

`a(1:3) ==`

23	15	1

`a(3:end) ==`

1	2.4	-1.1

`a(1:2:end) ==`

23	1	-1.1

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

a =

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

Dim 1

Dim 2

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`a(1,1) ==` 23

`a(4,3) ==` -3

row 4, col 3

`a(end,3) ==` 2

last row, col 3

`a(end,end) ==` 75

last row, last col

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

$\boxed{23}$	0	23	23	0.2
$\boxed{15}$	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`a([1 2],1) ==`

23
15

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`a(3,2:4) ==`

1	1	2

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`a(2:4,3:5) ==`

0	15	122
1	2	1
-3	76	2.4

Indexing on multidimensional arrays

Colon by itself means grab all indices along this dimension

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

```
a(1,:) ==
```

23	0	23	23	0.2

first row, all columns

Indexing on multidimensional arrays

Within the parentheses, include indices for each dimension, separated by commas

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`a(:,2) ==`

all rows, col 2

0
15
1
0
-1.1

Slice physiology data

2-dimensional array: each row is a trial, each column is a timepoint

`data =`

23	0	23	23	0.2	...	23	23	0.2
15	15	0	15	122	...	0	15	122
1	1	1	2	1	...	1	2	1
2.4	0	-3	76	2.4	...	-3	76	2.4
-1.1	-1.1	2	-1.1	75	...	2	-1.1	75

Trials

Time

Slice physiology data

How do we grab trial 1?

Trials

23	0	23	23	0.2	...	23	23	0.2
15	15	0	15	122	...	0	15	122
1	1	1	2	1	...	1	2	1
2.4	0	-3	76	2.4	...	-3	76	2.4
-1.1	-1.1	2	-1.1	75	...	2	-1.1	75

Time

Slice physiology data

Trials

23	0	23	23	0.2	...	23	23	0.2
15	15	0	15	122	...	0	15	122
1	1	1	2	1	...	1	2	1
2.4	0	-3	76	2.4	...	-3	76	2.4
-1.1	-1.1	2	-1.1	75	...	2	-1.1	75

Time

`data(1,:) ==`

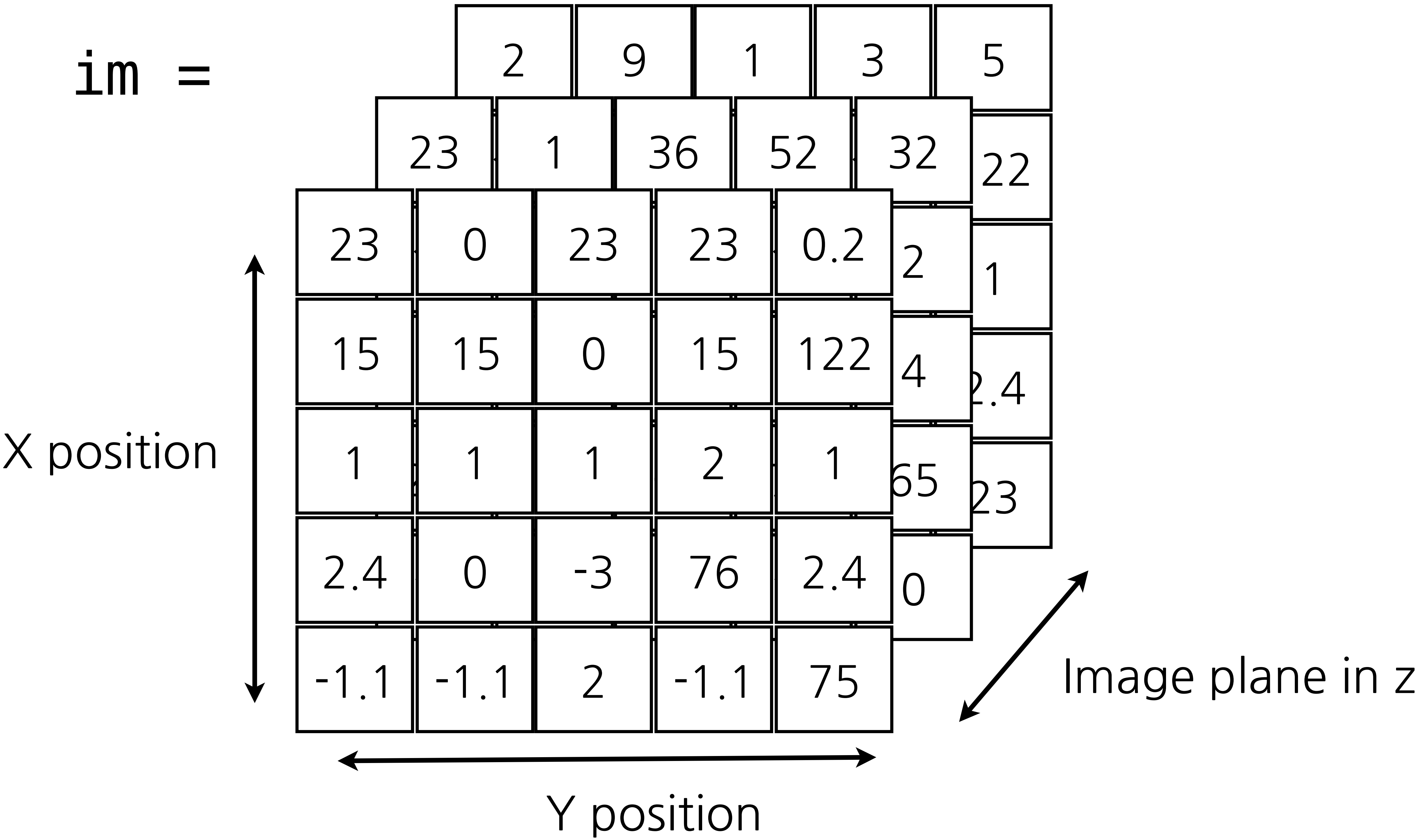
23	0	23	23	0.2	...	23	23	0.2

(trial 1)

3d image example

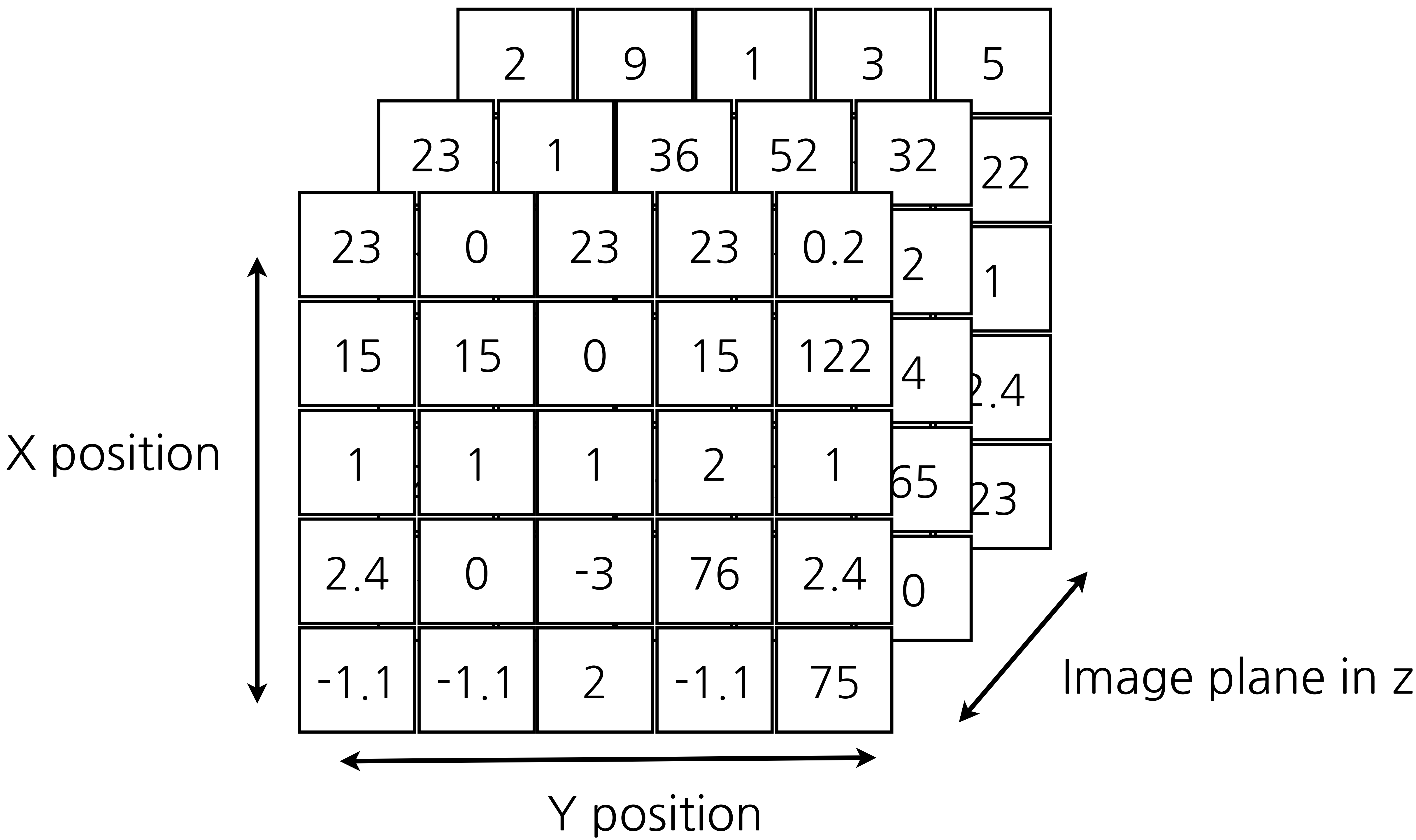
3-dimensional image stack

`im =`



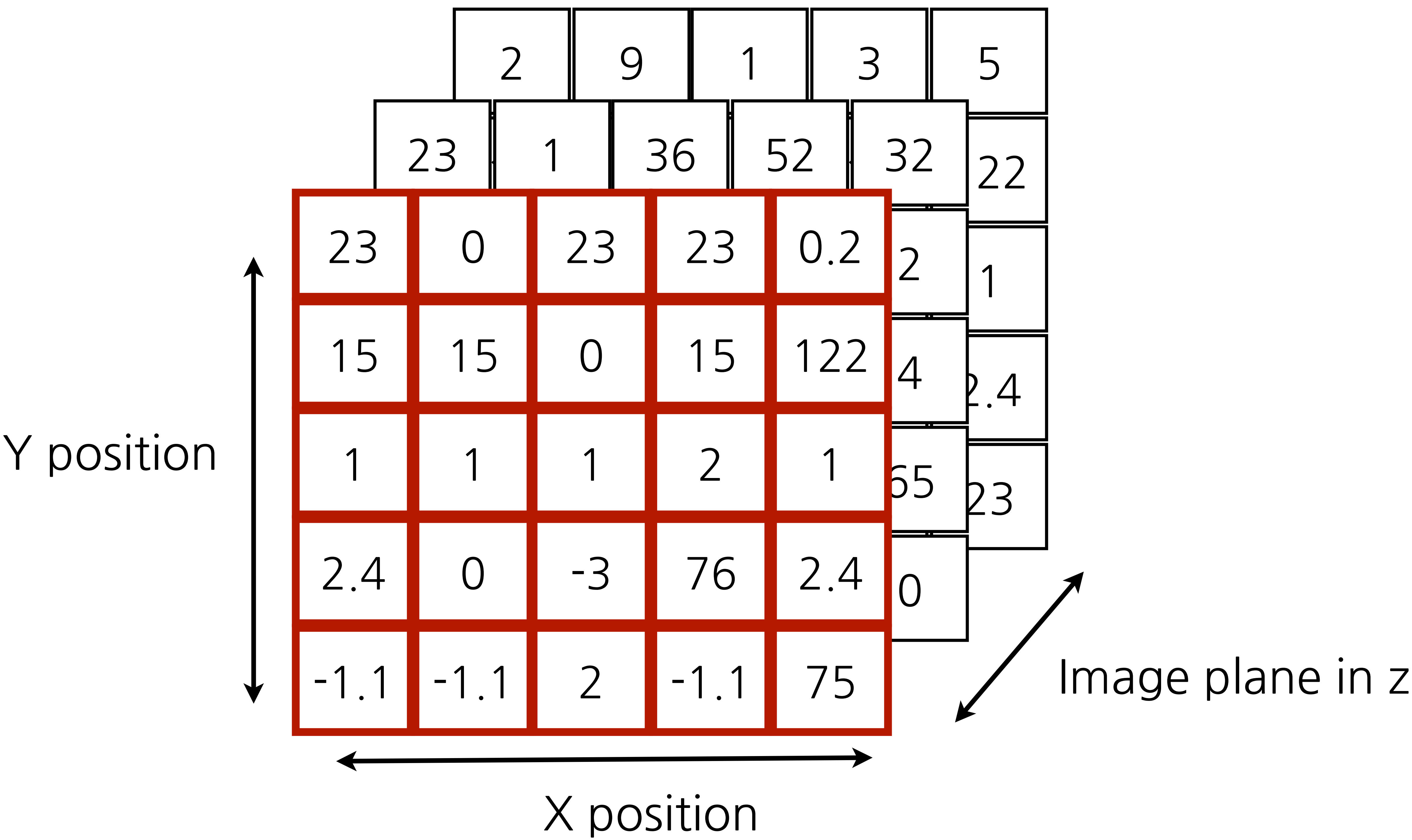
3d image example

How do we grab image 1 of the stack?



3d image example

```
im(:,:,1)
```



3d image example

`im(:,:,1) ==`

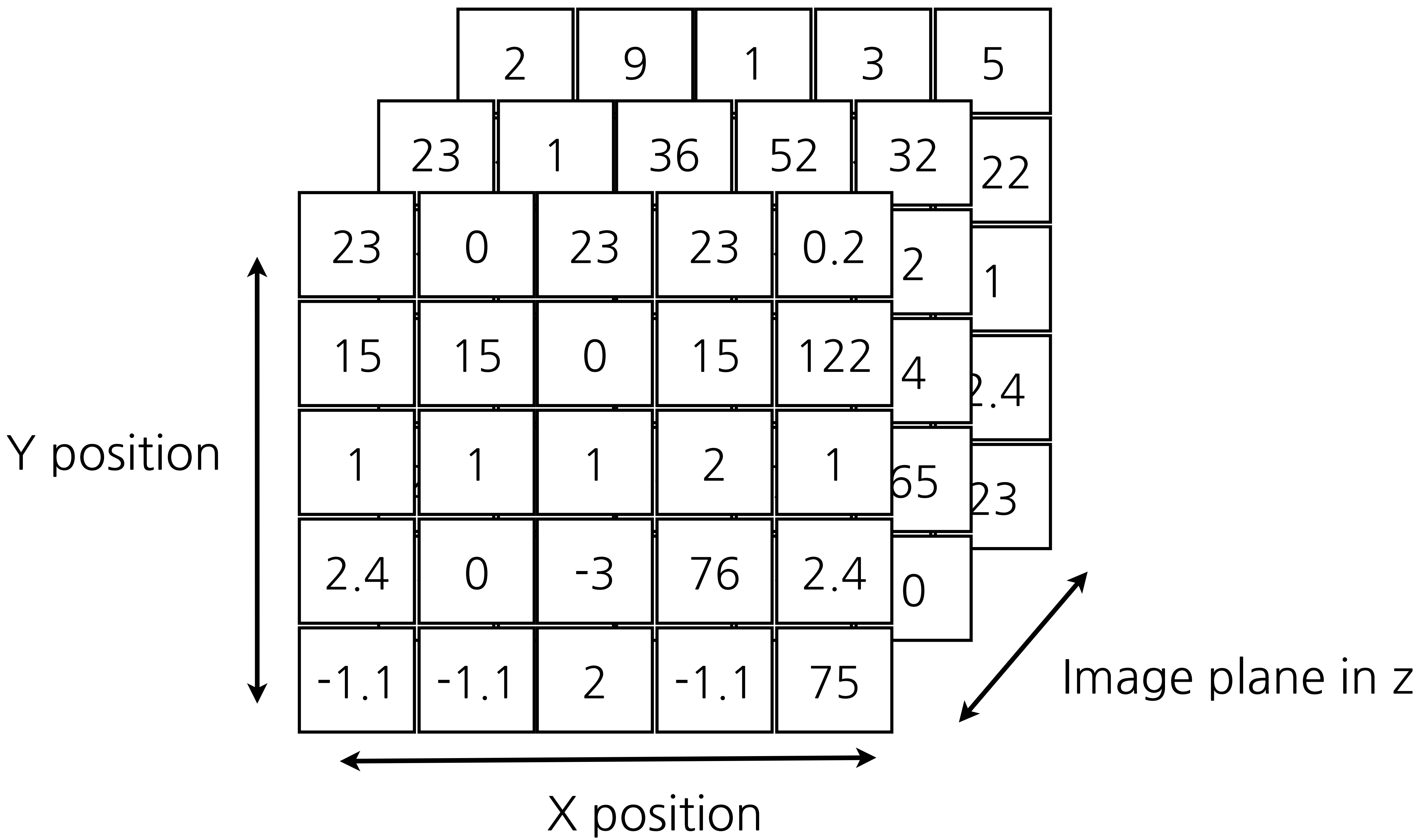
Y position

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

X position

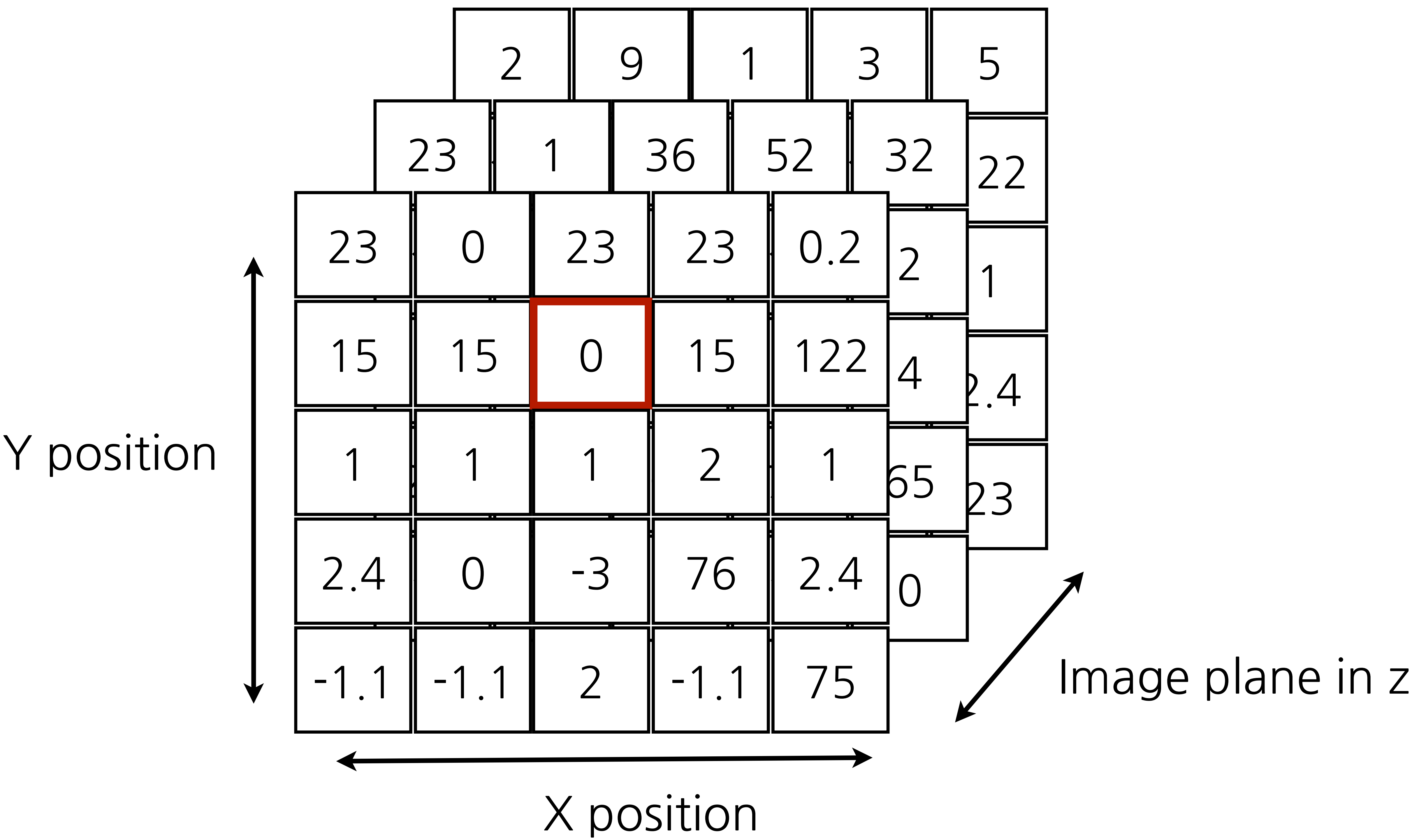
3d image example

How do we grab a z-stack at a particular coordinate?



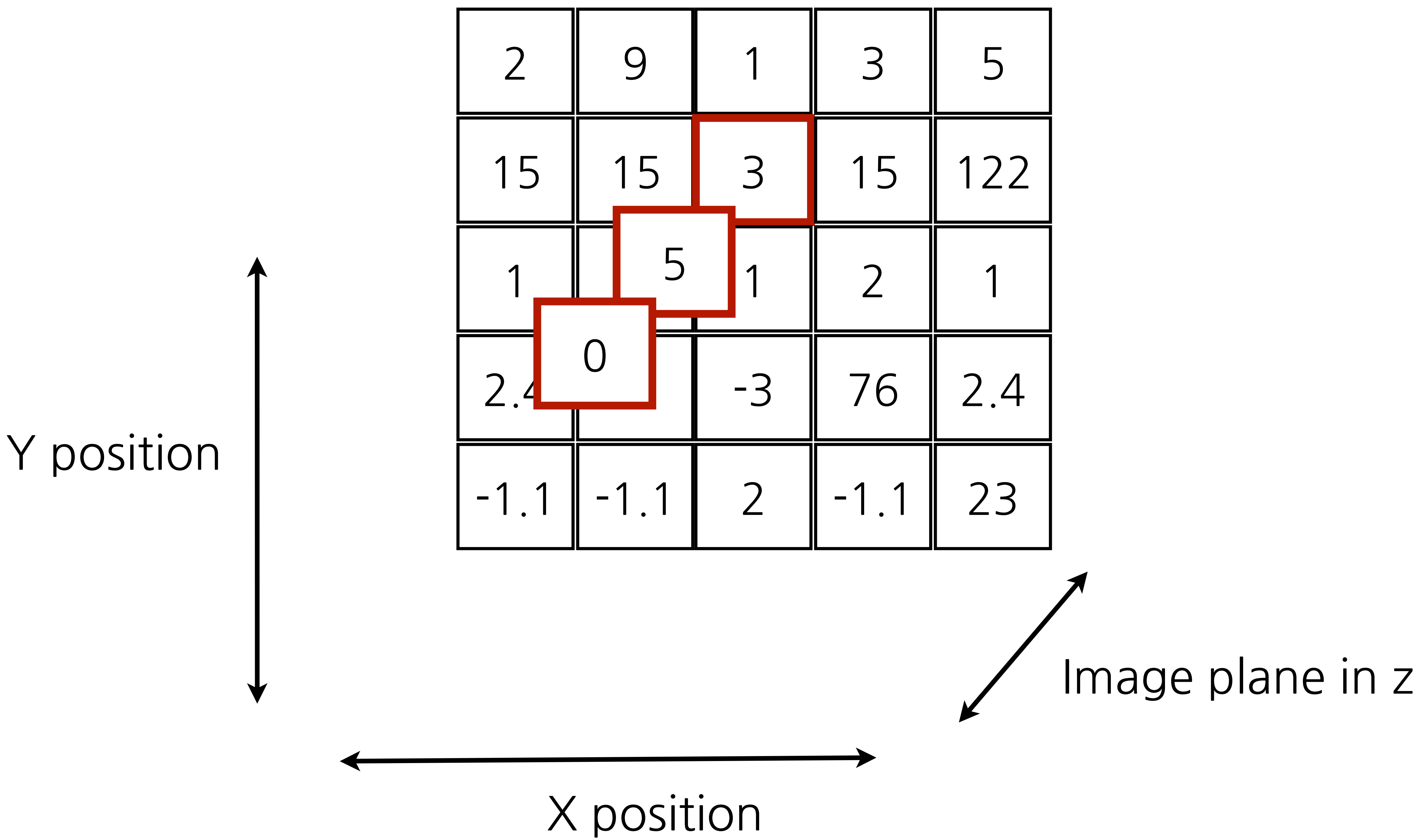
3d image example

`im(2,3,:)`



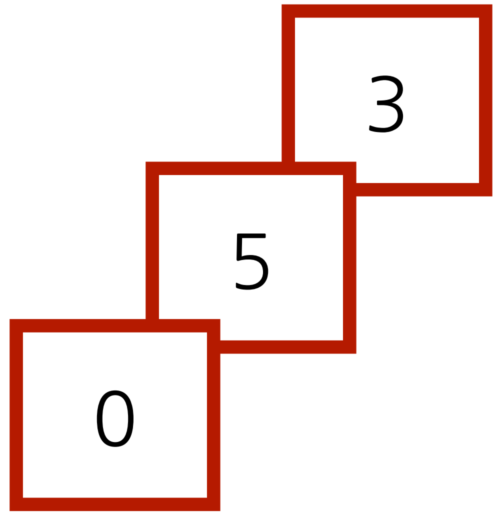
3d image example

```
im(2,3,:)
```



3d image example

```
im(2,3,:) ==
```



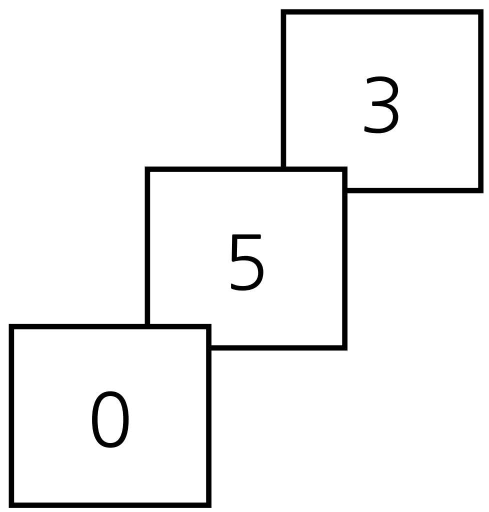

```
zProfile = im(2,3,:);

size(zProfile) == [1 1 3]
```

This is an unwieldy “shape” for this vector…

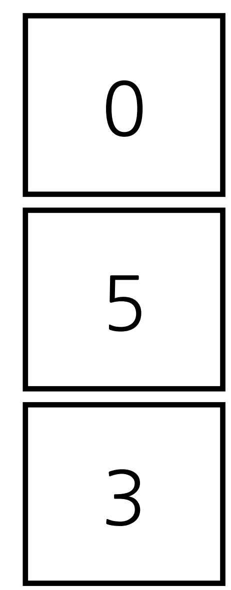
`squeeze()` function

The squeeze function looks at each dimension, and removes dimensions that have length 1. This is useful for reshaping arrays that you've extracted from something that is higher dimensional.

```
zProfile ==
```



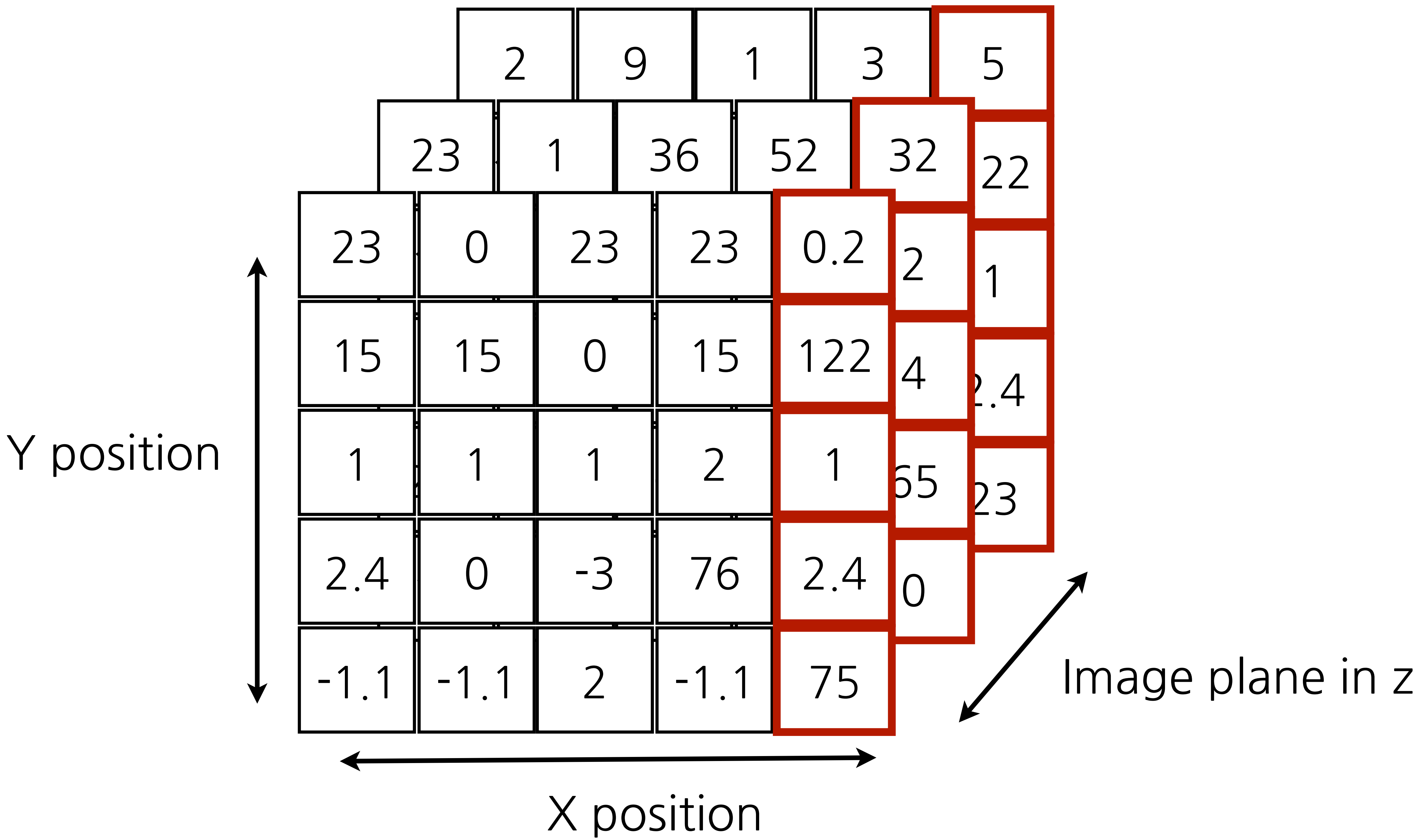

```
zProfile = squeeze(zProfile)
```




```
size(zProfile) == [3 1]
```

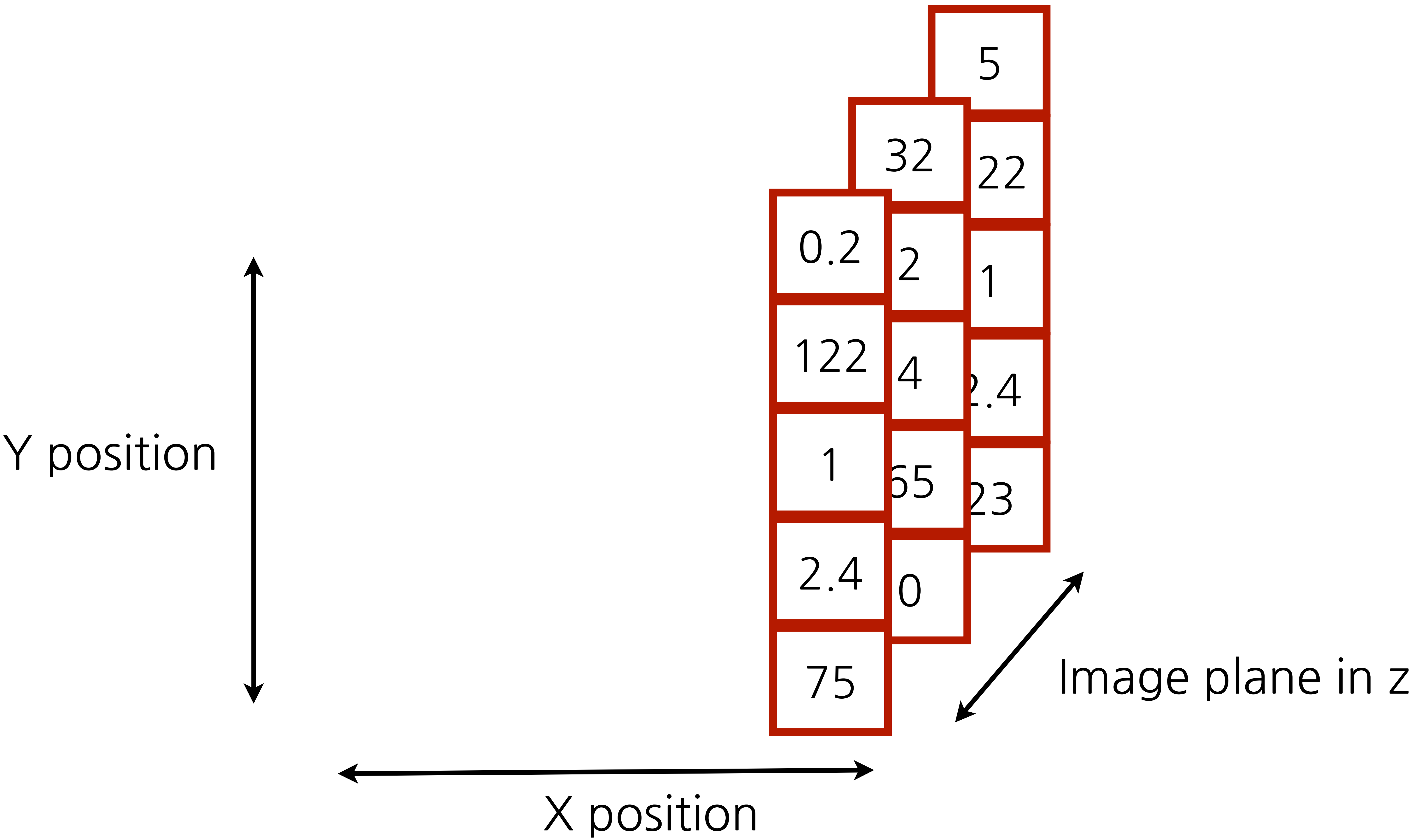
3d image example

How do we grab a side profile of this image stack?



3d image example

```
sideView = im(:,5,:)
```



3d image example

What happens if we run `squeeze()`?

Looks at dimension 1 (Y):

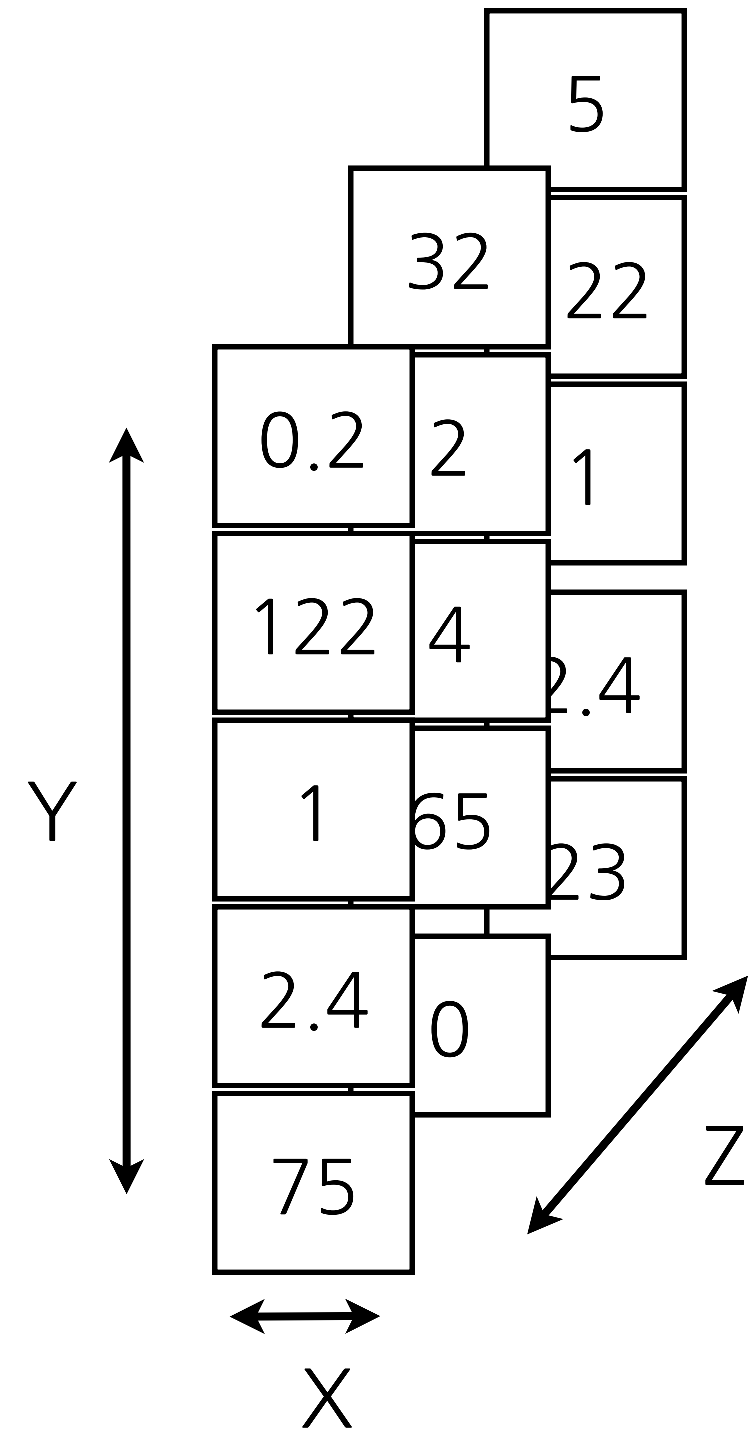
- Not length 1, move on

Looks at dimension 2 (X):

- length 1, get rid of this dimension!

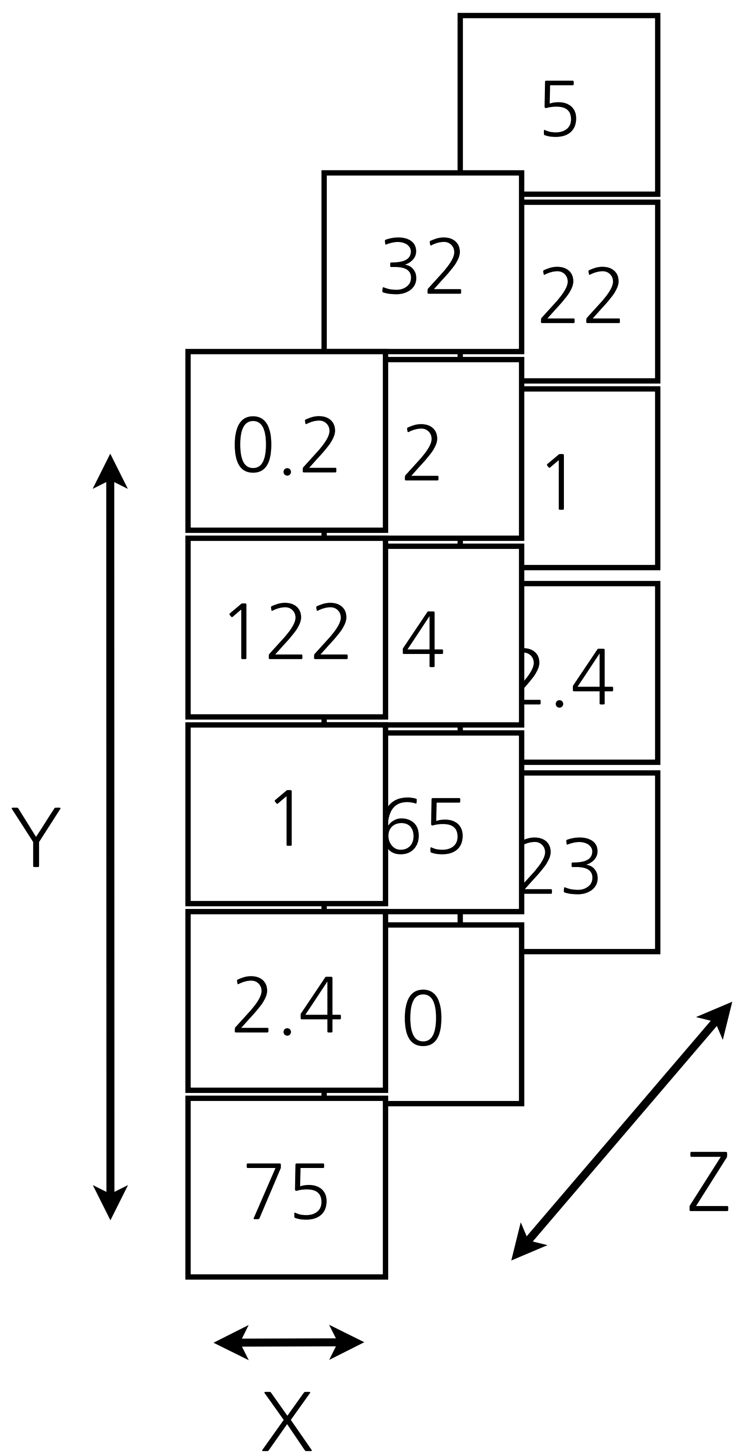
Looks at what was dimension 3 (Z)

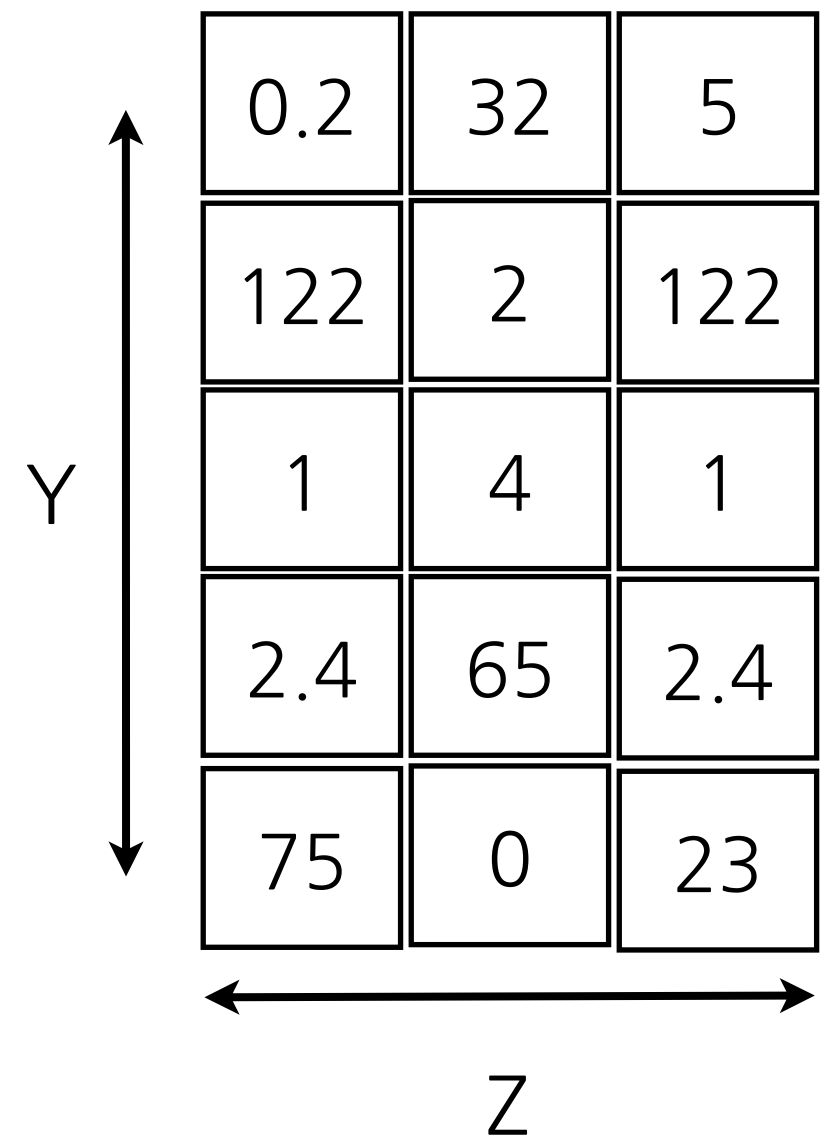
- Not length 1, move on



3d image example

```
sideView = squeeze(sideView)
```





still not quite right...

Transpose operation

Transpose means swap the row and column directions. This can reorient a 2d array, change a row vector into a column vector, or change a column vector into a row vector.

```
sideView = sideView'
```

0.2	32	5
122	2	122
1	4	1
2.4	65	2.4
75	0	23

Y (rows), Z (columns)

0.2	122	1	2.4	75
32	2	4	65	0
5	122	1	2.4	23

Z (rows), Y (columns)

Selecting indices automatically

Often you don't know what indices you want, but want to select them on the basis of some criteria.

A few related topics:

- Conditional operators
- Logical indexing
- find() command

Conditional operators

Tests a condition, evaluates to true (1) or false (0)

`1 < 2` evaluates to `1`

`2 < 2` evaluates to `0`

`2 <= 2` evaluates to `1`

`3 > 2` evaluates to `1`

`1 > 2` evaluates to `0`

`2 >= 2` evaluates to `1`

`2 == 2` evaluates to `1`

`3 == 2` evaluates to `0`

`3 ~= 2` evaluates to `1`

`2 ~= 2` evaluates to `0`

All of these `0` or `1` values that are returned are of class `logical`

Conditional operators

Can operate on each element of an array simultaneously

`[1 2 −1 1 −3] > 0` evaluates to `[1 1 0 1 0]`

`[1 2 −1 1 −3] == 2` evaluates to `[0 1 0 0 0]`

`[1 2 −1 1 −3] >= −1` evaluates to `[1 1 1 1 0]`

All of these `0` or `1` values that are returned are of class `logical`

Conditional operators

Works on multidimensional arrays too

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`== 0` evaluates to

0	1	0	0	0
0	0	1	0	0
0	0	0	0	0
0	1	0	0	0
0	0	0	0	0

All of these `0` or `1` values that are returned are of class `logical`

Conditional operators

Compare equal-size arrays element-wise

$$\begin{bmatrix} 23 & 0 \\ 15 & 15 \\ 1 & 1 \\ 2.4 & 0 \\ -1.1 & -1.1 \end{bmatrix} == \begin{bmatrix} 23 & 5 \\ 15 & 4 \\ 0 & 1 \\ 2.4 & 2 \\ 0 & -1.1 \end{bmatrix} \text{ evaluates to } \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

All of these `0` or `1` values that are returned are of class `logical`

Boolean operators

Allow you to select indices based on multiple conditions.

‘**and**’ operator & requires **both** conditions to be true

‘**or**’ operator | requires **either** condition to be true

And operator

'and' operator & requires **both** conditions to be true

```
vals = [1 2 -1 1 -3];
```

`vals >= 0` evaluates to `[1 1 0 1 0]`

`vals <  2` evaluates to `[1 0 1 1 1]`

`vals >= 0 & vals < 2` evaluates to `[1 0 0 1 0]`

Or operator

'or' operator | requires **either** condition to be true

```
vals = [1 2 -1 1 -3];
```

`vals < 0` evaluates to `[0 0 1 0 1]`

`vals > 1` evaluates to `[0 1 0 0 0]`

`vals < 0 | vals > 1` evaluates to `[0 1 1 0 1]`

Demo: Conditional operators

Logical indexing

Use conditional operators to create a logical array of the same size as the original. Then use the logical array to pick out the indices that satisfy those conditions.

Logical index array must be the same size as the array being indexed into.

Must be of class `logical` (as opposed to double). Conditional operators return `logical` arrays.

Logical indexing

These logical arrays are useful because you can use them directly to index into arrays

```
vals = [1 2 -1 1 -3];
```

`vals >= 0` evaluates to `[1 1 0 1 0]`

```
indsToSelect = vals >= 0;
```

`vals(indsToSelect)` evaluates to `[1 2 1]`

Logical indexing

These logical arrays are useful because you can use them directly to index into arrays

`vals =`

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

Logical indexing

These logical arrays are useful because you can use them directly to index into arrays

`vals == 0` evaluates to

0	1	0	0	0
0	0	1	0	0
0	0	0	0	0
0	1	0	0	0
0	0	0	0	0

`vals(vals == 0)` evaluates to `[0; 0; 0]`

Logical indexing

These logical arrays are useful because you can use them directly to index into arrays

`vals > 15` evaluates to

1	0	1	1	0
0	0	0	0	0
0	0	0	0	0
0	0	0	1	0
0	0	0	0	1

`vals(vals > 15)` evaluates to

`[23; 23; 23; 76; 75]`

Assignment using logical indexing

You can assign over the values selected using logical indexing. Useful for truncation and marking values as invalid

```
vals = [1 2 -1 1 -3];
```

`vals < 0` evaluates to `[0 0 1 0 1]`

Mark values as invalid by replacing with NaN

```
vals(vals < 0) = NaN;
```

`vals` evaluates to `[1 2 NaN 1 NaN]`

Assignment using logical indexing

You can assign over the values selected using logical indexing. Useful for truncation and marking values as invalid

```
vals = [1 2 -1 1 -3];
```

`vals < 0` evaluates to `[0 0 1 0 1]`

Truncate values from below:

```
vals(vals < 0) = 0;
```

`vals` evaluates to `[1 2 0 1 0]`

Assignment using logical indexing

You can assign over the values selected using logical indexing. Useful for truncation and marking values as invalid

```
vals = [1 2 -1 1 -3];
```

`vals < 0` evaluates to `[0 0 1 0 1]`

Remove selected values:

```
vals(vals < 0) = [];
```

`vals` evaluates to `[1 2 1]`

`nnz()` function

Counts the **number of non-zero** elements

Can be used on any array, but with logical arrays, counts the number of elements that satisfy the conditions.

```
vals = [1 2 -1 1 -3];
```

`nnz(vals > 0)` evaluates to `3`

nnz() function

Counts the **number of non-zero** elements

Can be used on any array, but with logical arrays, counts the number of elements that satisfy the conditions.

`vals =`

23	0	23	23	0.2
15	15	0	15	122
1	1	1	2	1
2.4	0	-3	76	2.4
-1.1	-1.1	2	-1.1	75

`nnz(vals == 1)` evaluates to `4`

Demo: Logical indexing

`find()` function

The find command is useful when you are interested in the position of values that satisfy a set of conditions (and not just the values themselves).

At it's simplest, `find()` takes a logical array and returns a list of which indices are `1` (true).

```
idx = logical([1 0 1 0 1]);
```

`find(idx)` evaluates to `[1 3 5]`

`find()` function

Typically, you combine two operations in one line:

- Use conditional operators to create the logical array
- Use find to locate the `1`s, i.e. the positions where the conditions are satisfied

```
vals = [1 2 -1 1 -3];
```

`find(vals > 0)` evaluates to `[1 2 4]`

`find()` function

Use multiple outputs to locate the indices rows, columns, etc.

`vals =`

0	5	0	0	0
0	0	1	0	0
0	0	0	0	0
0	8	0	0	0
0	0	0	0	0

```
[i, j] = find(vals > 0);
```

`i` evaluates to `[1; 2; 4]`

Rows on which the values are found

`j` evaluates to `[2; 3; 2]`

Columns in which the values are found

Demo: `find()` function

Summary

Multiple dimensional arrays can be very useful in managing data

The key is keeping track of what each dimension means, so that extracting what you want is a simple indexing operation.

Use conditional operators to filter data points by certain criteria, then use logical indexing to pull out those data points. Or use `find()` to ask where they're located in the array.

Sophisticated indexing, criteria testing, performing calculations, and assigning into whole chunks of an array simultaneously in one operation is the real advantage of the MATLAB language.

Importing data

- Data can be saved in lots of different formats
- We want to be able to read in data from different programs and formats (CSV, TXT, XLS, XML, ABF, JPG, ...)
- Most common data formats have build in commands to read that data

File importing

MATLAB offers functions that load some common file formats:

- `csvread`: comma separated value .csv files containing only numeric data
- `dlmread`: delimited dat file containing only numeric data separated by a delimiter character (space, tab, newline, etc.)
- `xlsread`: read Excel spreadsheet
- `textscan`: read data in a file with a custom format
- `imread`: numerous image formats
- `fread, fgetl, fscanf, fseek`: low-level line

csvread() function

Reads a file with only numeric data separated by commas and newlines. Returns a matrix of those values. Use row, col, and range to select particular rows and columns.

Not very useful if your data has a mix of numeric and text information in it. In that case, see `textscan()`

```
M = csvread(filename, row, col, range)
```

These three arguments are optional

If filename contained:

```
02, 04, 06, 08, 10, 12
03, 06, 09, 12, 15, 18
05, 10, 15, 20, 25, 30
07, 14, 21, 28, 35, 42
11, 22, 33, 44, 55, 66
```

`M` would evaluate to:

```
 2     4     6     8    10    12
 3     6     9    12    15    18
 5    10    15    20    25    30
 7    14    21    28    35    42
11    22    33    44    55    66
```

csvread() function

Reads a file with only numeric data separated by commas and newlines. Returns a matrix of those values. Use row, col, and range to select particular rows and columns.

If you need to skip a header line, use 1 in the second argument.

```
M = csvread('data.csv', 1)
```

Means skip the first 1 row

If data.csv contained:

```
 a,  b,  c,  d,  e,  f
02, 04, 06, 08, 10, 12
03, 06, 09, 12, 15, 18
05, 10, 15, 20, 25, 30
07, 14, 21, 28, 35, 42
11, 22, 33, 44, 55, 66
```

`M` would evaluate to:

```
 2     4     6     8    10    12
 3     6     9    12    15    18
 5    10    15    20    25    30
 7    14    21    28    35    42
11    22    33    44    55    66
```

xlsread() function

Reads an Excel spreadsheet. Only opens XLS 97-2000 unless you have Excel installed and you're running Windows.

Optional: index of cells, e.g. `'B2:D5'`

```
[num,txt,raw] = xlsread(filename,sheet,range)
```

Optional: Name or number of sheet to load

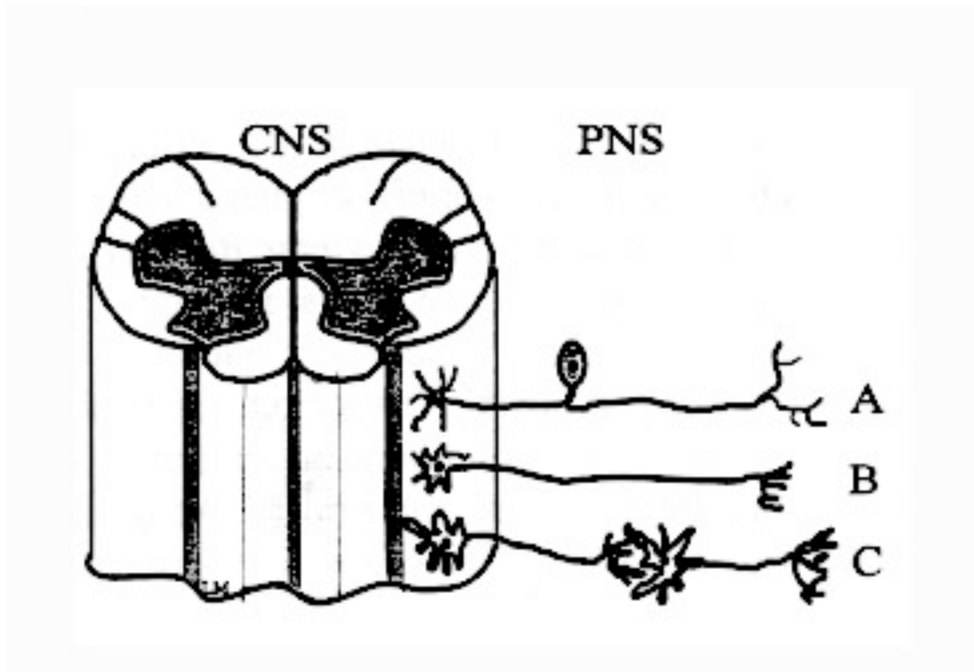
Numeric data as 2d array

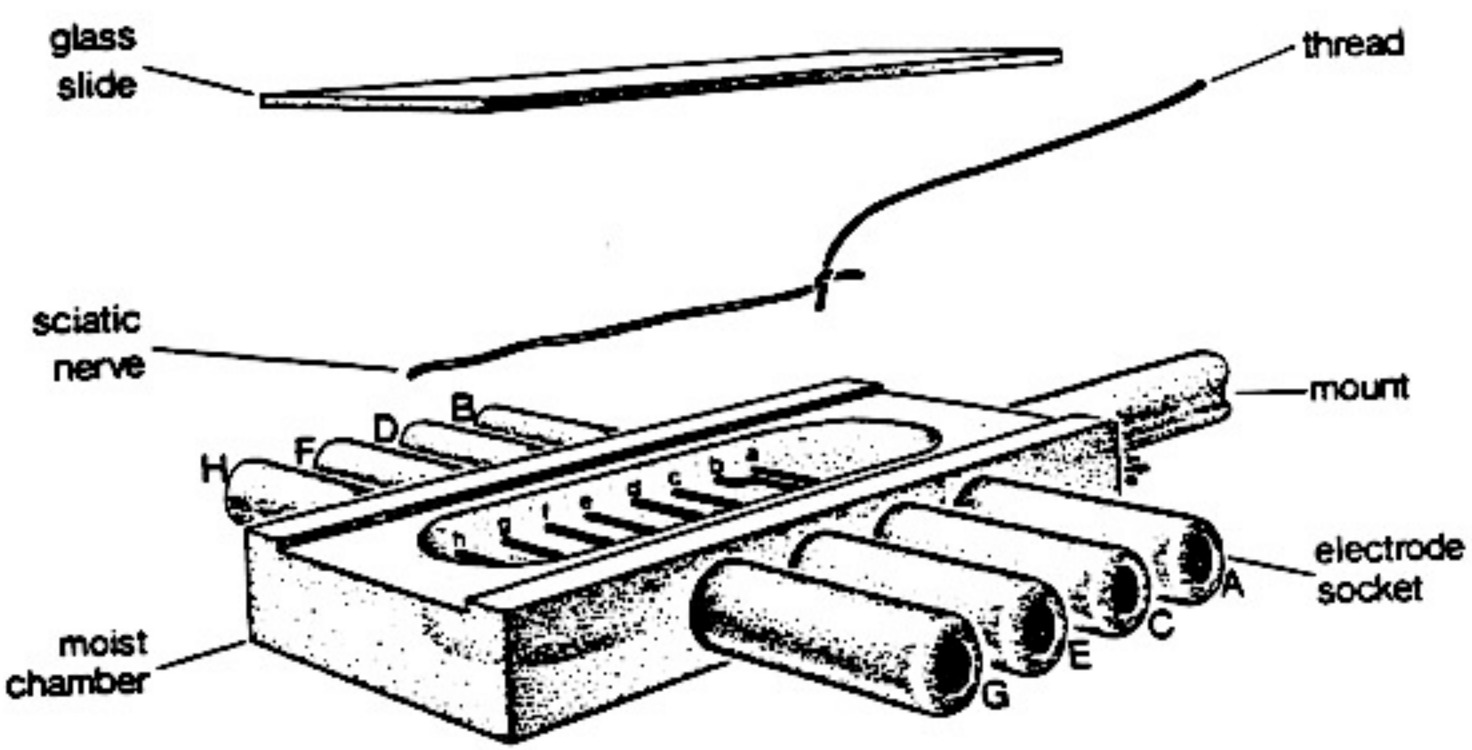
Text data as cell array

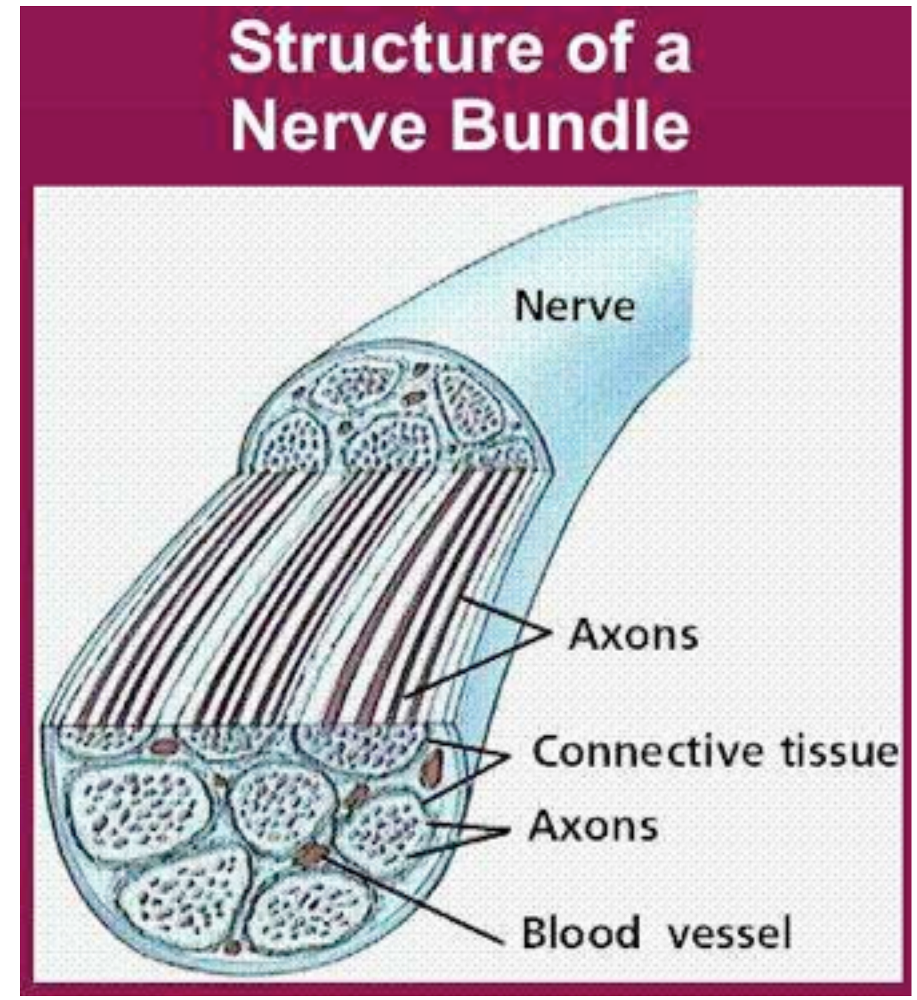
All data (numeric and text) as cell array

Demo:
Data Import

Problem Set #2







source: http://www.utoronto.ca/physio/courses/nrs302/Week3/nrs302_sec3_Compound_Action_Potential.html

Problem Set #2

- Data:
  - Voltage and time (mat)
  - Pulse duration and strength (csv)
  - Electrode distance and delay

- Basic signal processing - remove noise from a trace

source: http://www.utoronto.ca/physio/courses/nrs302/Week3/nrs302_sec3_Compound_Action_Potential.html

Review

Concepts

Data types:

Numerical classes are for storing numbers. Examples of numerical classes are integers, doubles, floats

Strings are for storing text

Logicals are stored as 0 or 1

There are other types for more structured data (structures, classes), but these are the basics

Importing data from other programs and file types using built in commands

Naming conventions are used to help keep code easily readable and consistent

Functions

`+ - / *` arithmetic
`;` suppresses output
`:` incremental indexing
`%` comments
`%%` code blocks
`a'` transpose
`[a b]` concatenates horizontally
`[a; b]` concatenates vertically
`a(3:end-1)` indexing
`a(n)=[]` excises nth element

```
disp
load
save
saveas
clear
clc
mean
length
size
plot
bar
hist
title
xlabel
ylabel
pwd
edit
```