Week 2:
Working with Data

Demo:
Manipulating MRI Data

Outline

Data Types in MATLAB

- Numeric arrays
- Indexing Arrays
- Logicals and logical indexing
- Strings

Importing Data

Assignment 2 Overview

Data Types in MATLAB

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

Numeric data types

Most common data type: **`double`**

- Stores floating point values

$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times 10^{\overbrace{-4}^{\text{exponent}}}$$

Each **double** uses 64 bits or 8 bytes

- Don't worry about this unless you have massive amounts of data, you can store 500 million of these values in 4 GB RAM

Other numeric data types include

- **`single`** - 32 bit floating point
- **`int8, uint8, int16, uint16, int32, uint32, int64, uint64`** - signed and unsigned integers

Arrays / matrices

MATLAB = Matrix Laboratory

- Most variables (regardless of dimensions) are really an array
- Arrays can have arbitrary dimensions
  - 1D array -> "vector"

23	15	1	2.4	-1.1

  - 2D array -> matrix

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

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

Review: Colon notation

Examples:

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

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

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

Syntax for creating 2d arrays

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!

Useful functions for arrays

`size(array)`

- returns the number of elements in each dimension of the array
- Example:

`size(` $\begin{bmatrix} 23 & 0 \\ 15 & 15 \\ 1 & 1 \end{bmatrix}$ `)` would return `[3 2]`

nRows nCols

Demo:
Numerical Arrays

Review: 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

Indices can and usually are also arrays

```
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

```
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 allows you to select specific elements based on their location

Indexing on multidimensional arrays

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

Dim 1

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 2

Indexing on multidimensional arrays

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

Dim 1

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 2

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

23

Indexing on multidimensional arrays

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

Dim 1

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 2

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

`a(4,3) ==` $\boxed{-3}$

row 4, col 3

Indexing on multidimensional arrays

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

Dim 1

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 2

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

23

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

-3

row 4, col 3

```
a(end,3) ==
```

2

last row, col 3

Indexing on multidimensional arrays

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

Dim 1

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 2

```
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

Dim 1

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 2

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

23
15

Indexing on multidimensional arrays

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

Dim 1

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 2

`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

Dim 1

Dim 2

```
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

Dim 1

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 2

`a(1,:) ==`

23	0	23	23	0.2

first row, all columns

Indexing on multidimensional arrays

Colon by itself means grab all indices along this dimension

Dim 1

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 2

`a(:,2) ==`

0
15
1
0
-1.1

all rows, col 2

Array dimensions should mean something to you, the programmer

physiology data

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

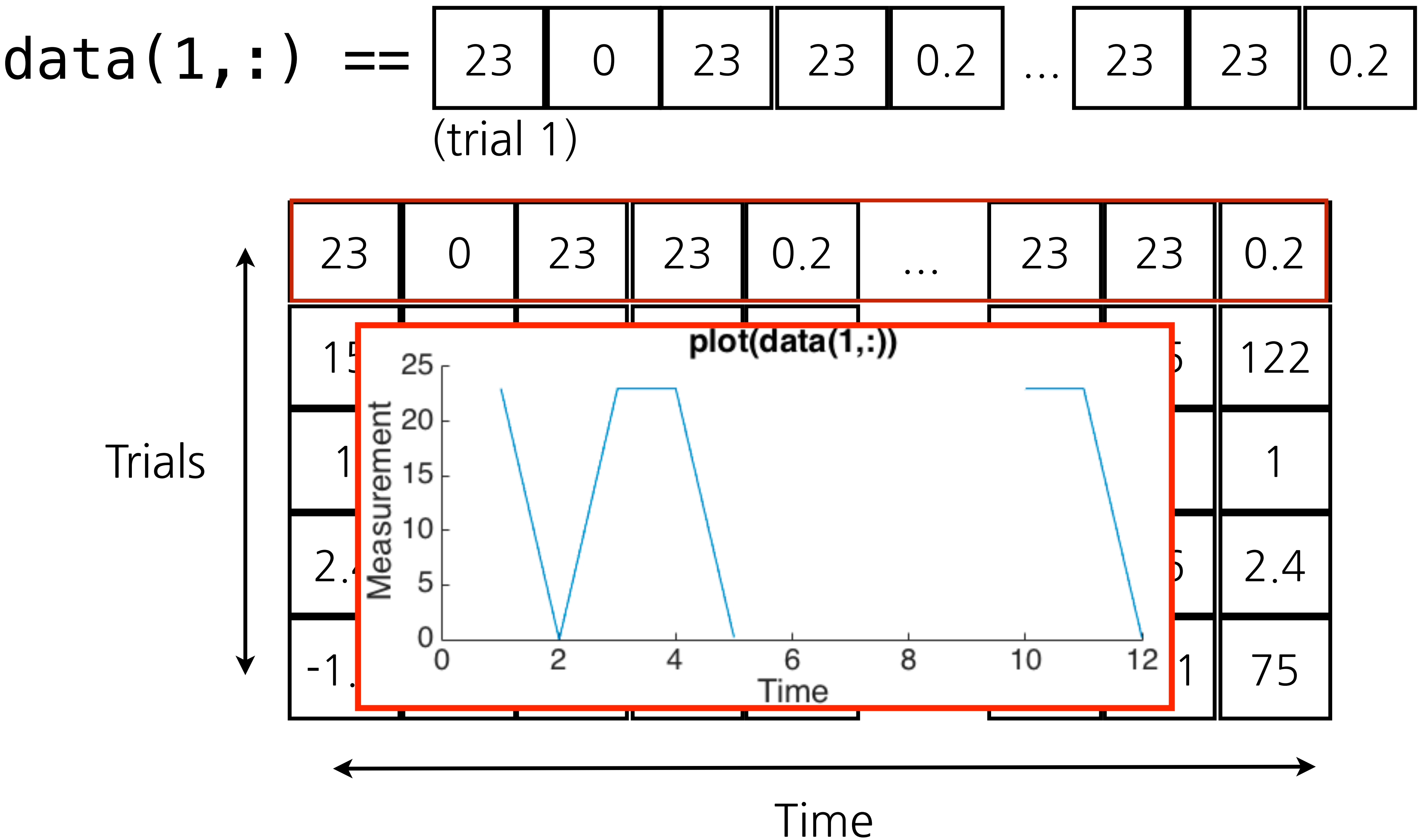
`data =`

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

physiology data

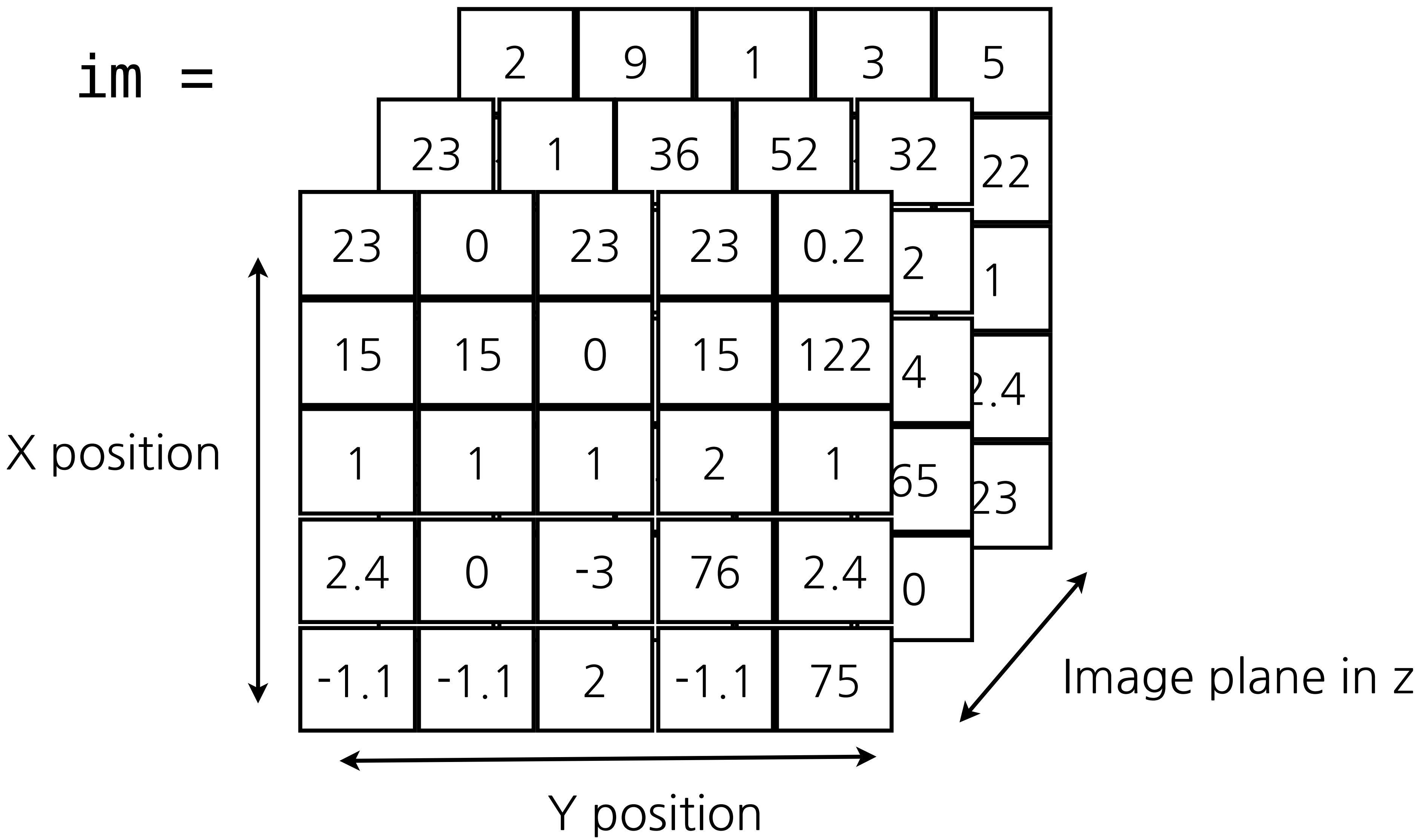


How do we grab trial 1?

3d arrays: image example

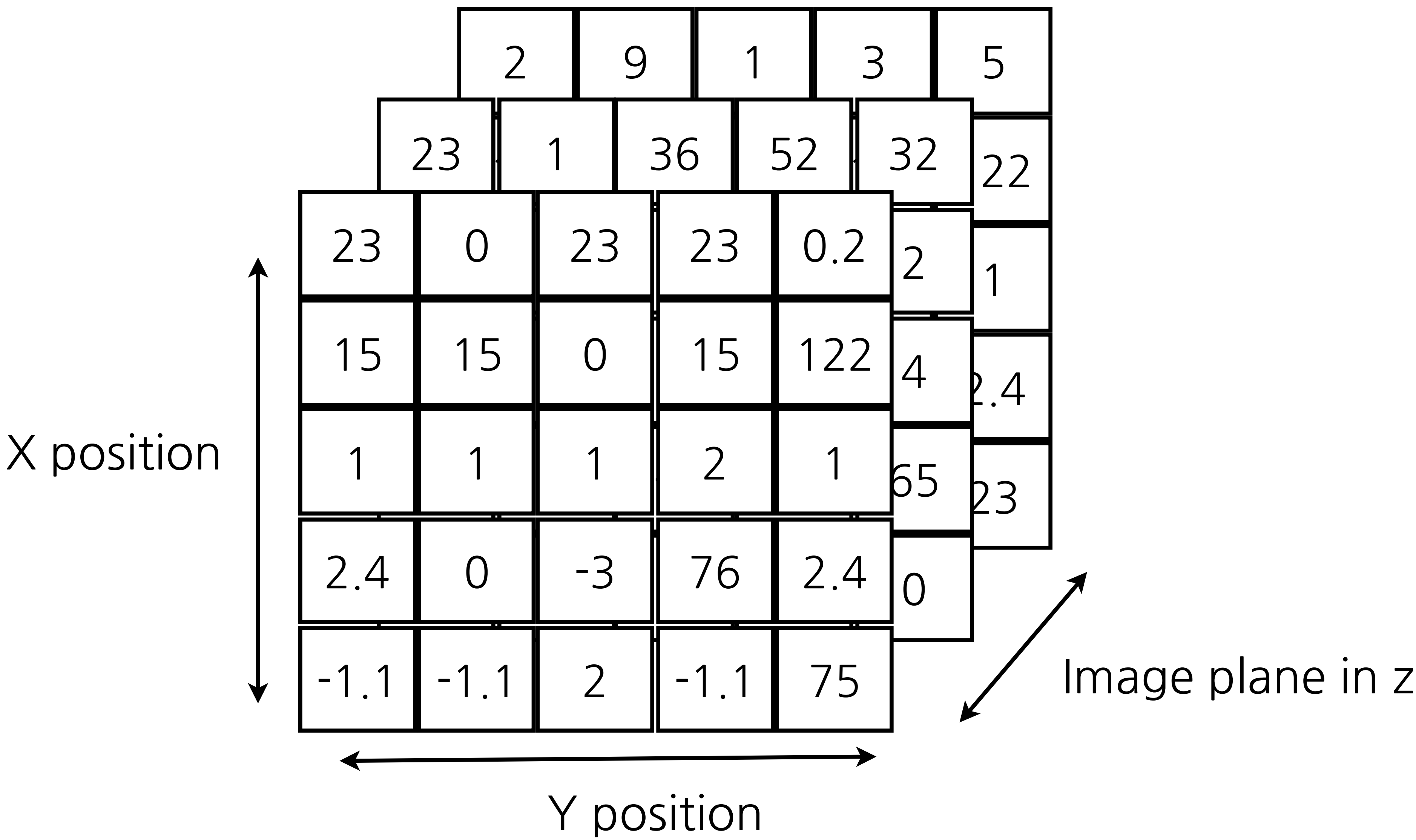
3-dimensional image stack

`size(im) == [5,5,3]`



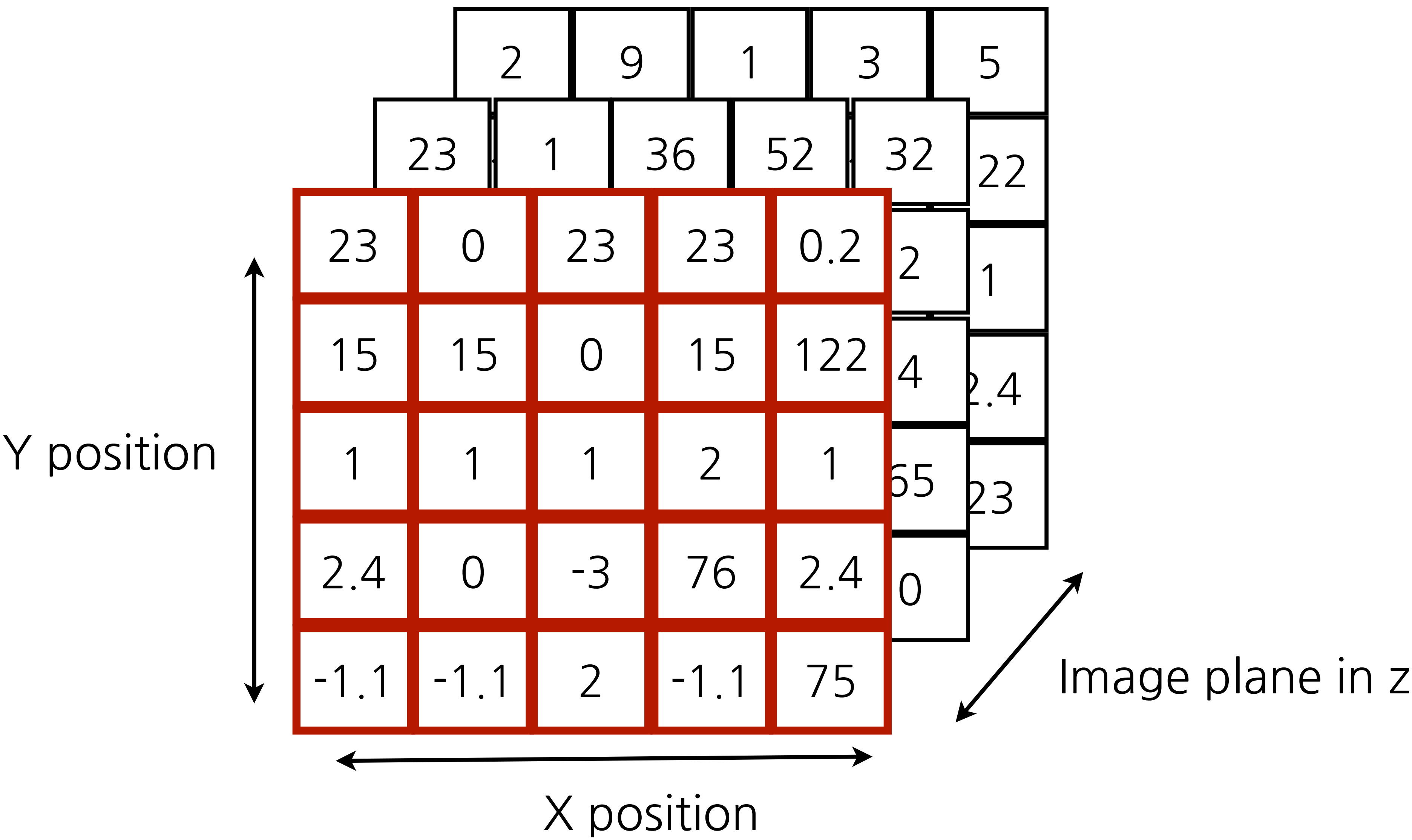
3d arrays: image example

How do we grab image 1 (top) of the stack?



3d arrays: image example

`im(:,:,1)`



3d arrays: image example

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

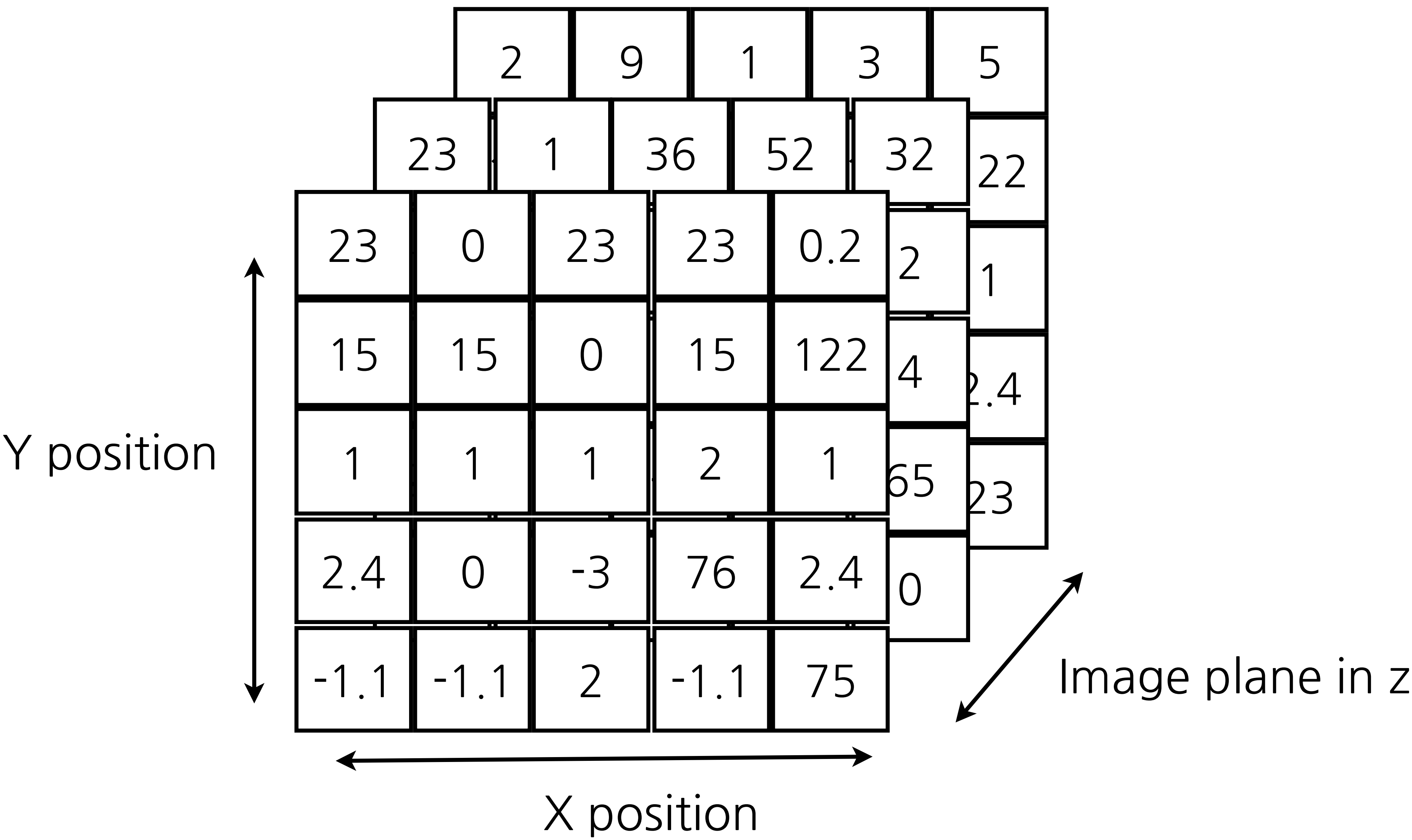
Y position

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

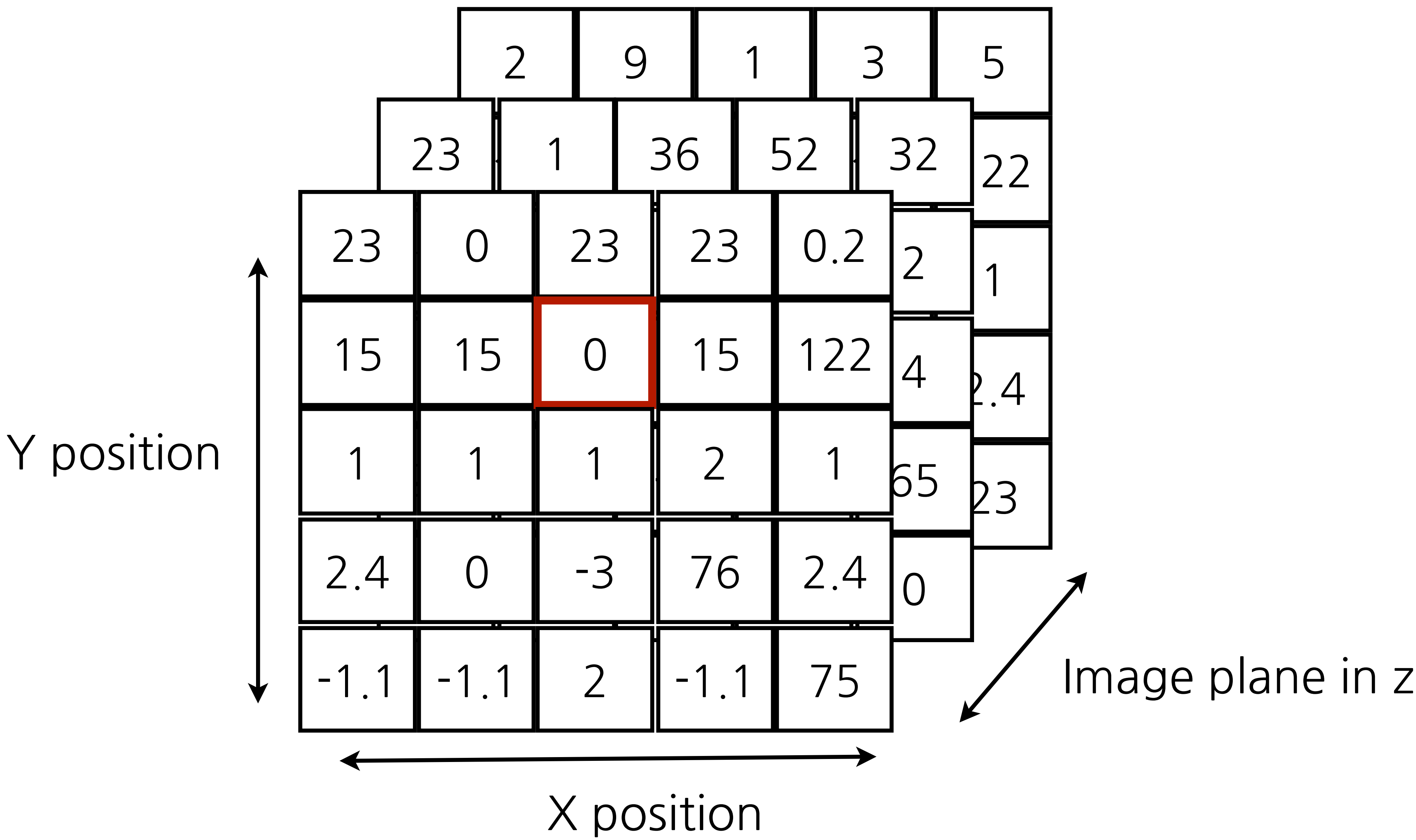
3d arrays: image example

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



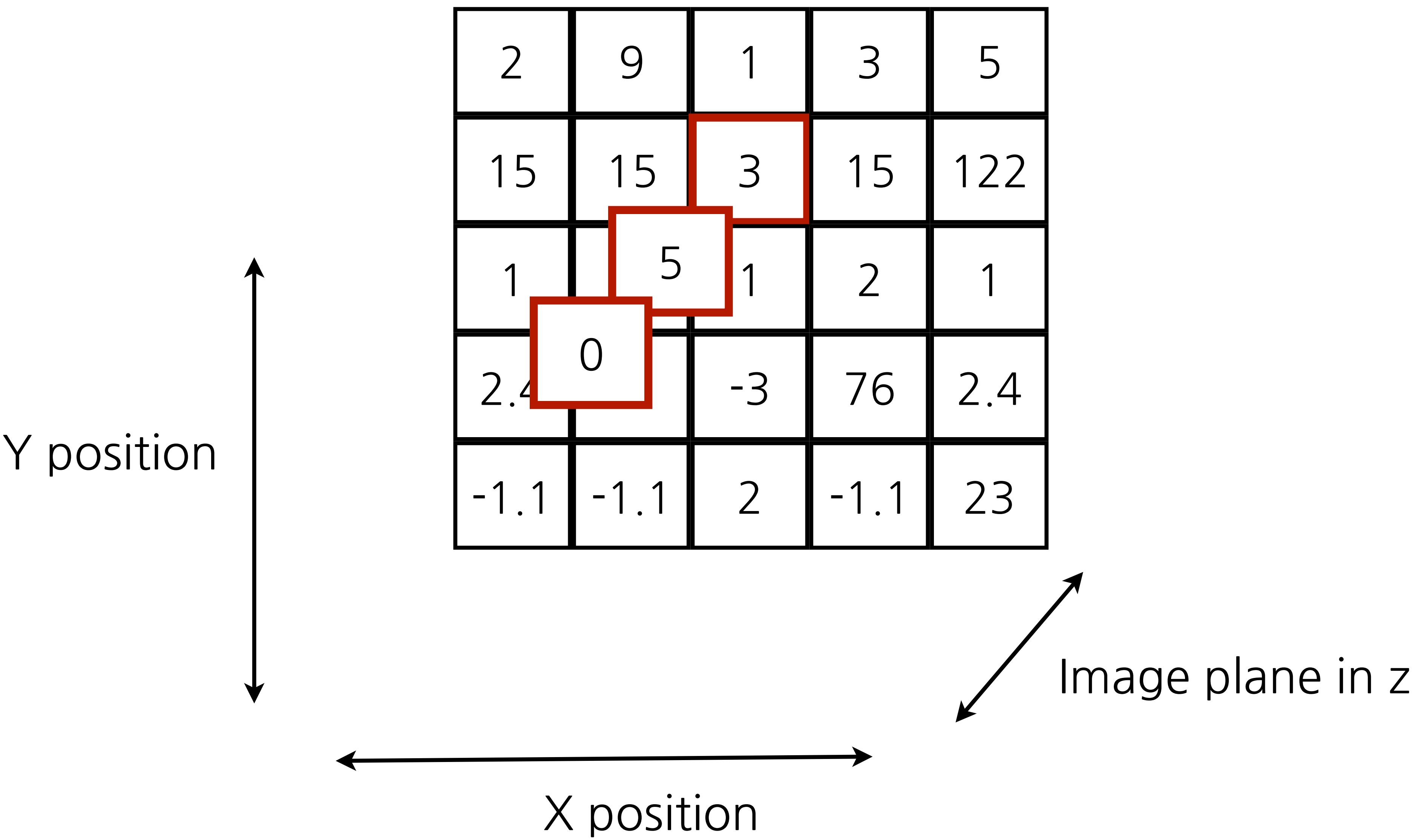
3d arrays: image example

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

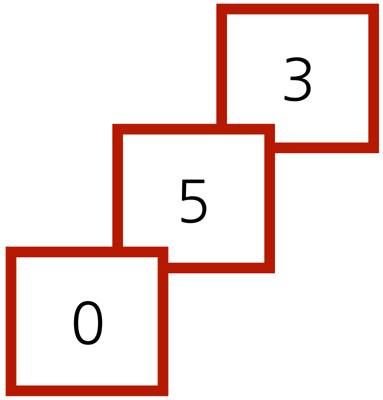


3d arrays: image example

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



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




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

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

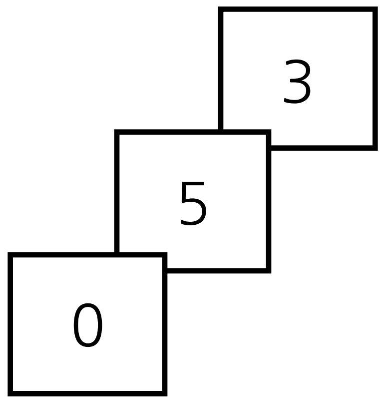
This is an unwieldy “shape” for this vector…

squeeze() function

Removes dimensions that have length 1

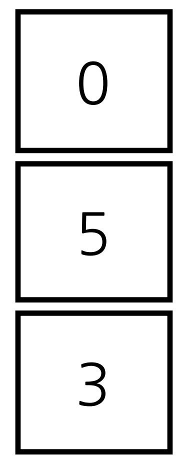
Useful for reshaping “awkward” arrays that you’ve extracted from something that is higher dimensional

```
zProfile ==
```




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

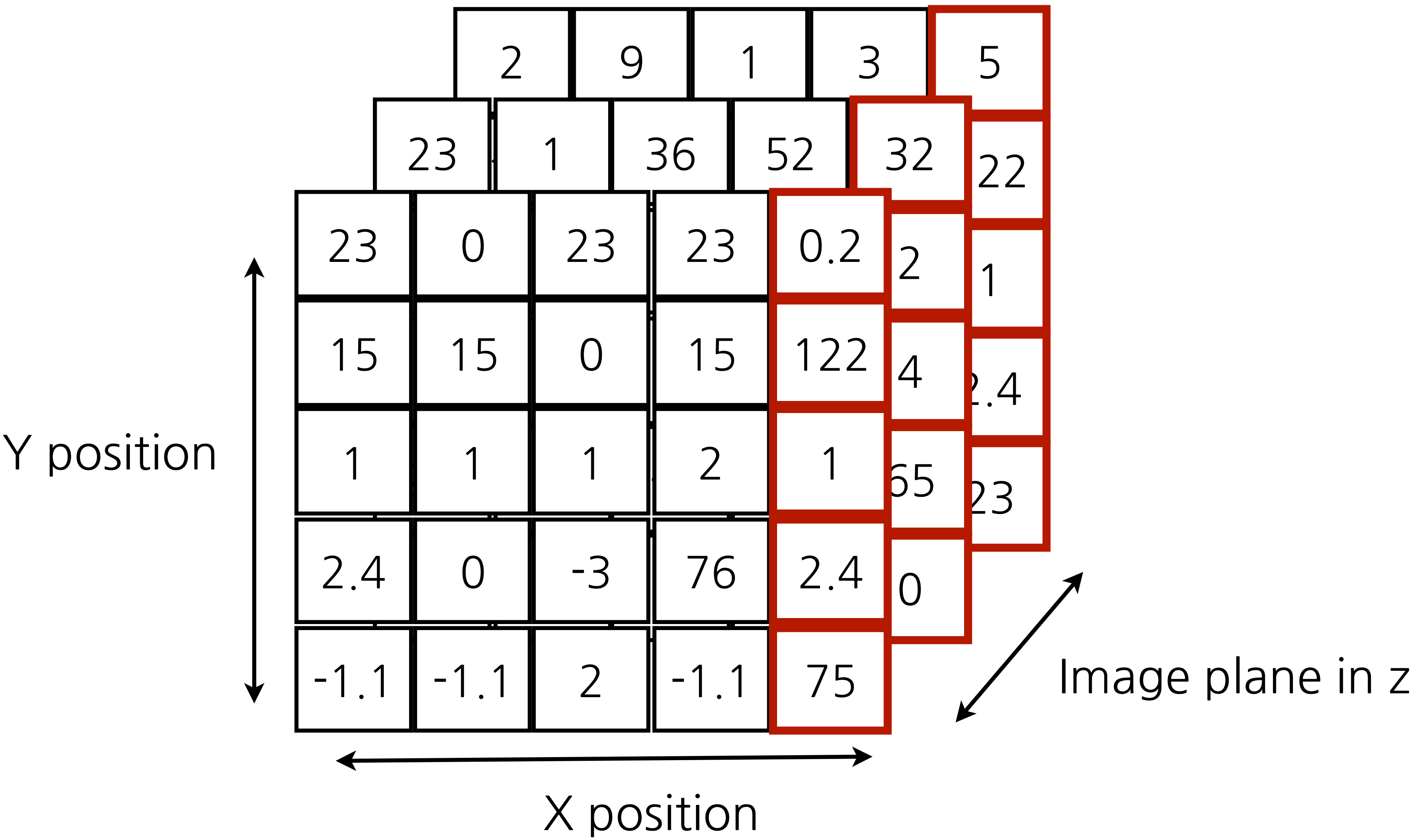
```
zProfile = squeeze(zProfile)
```




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

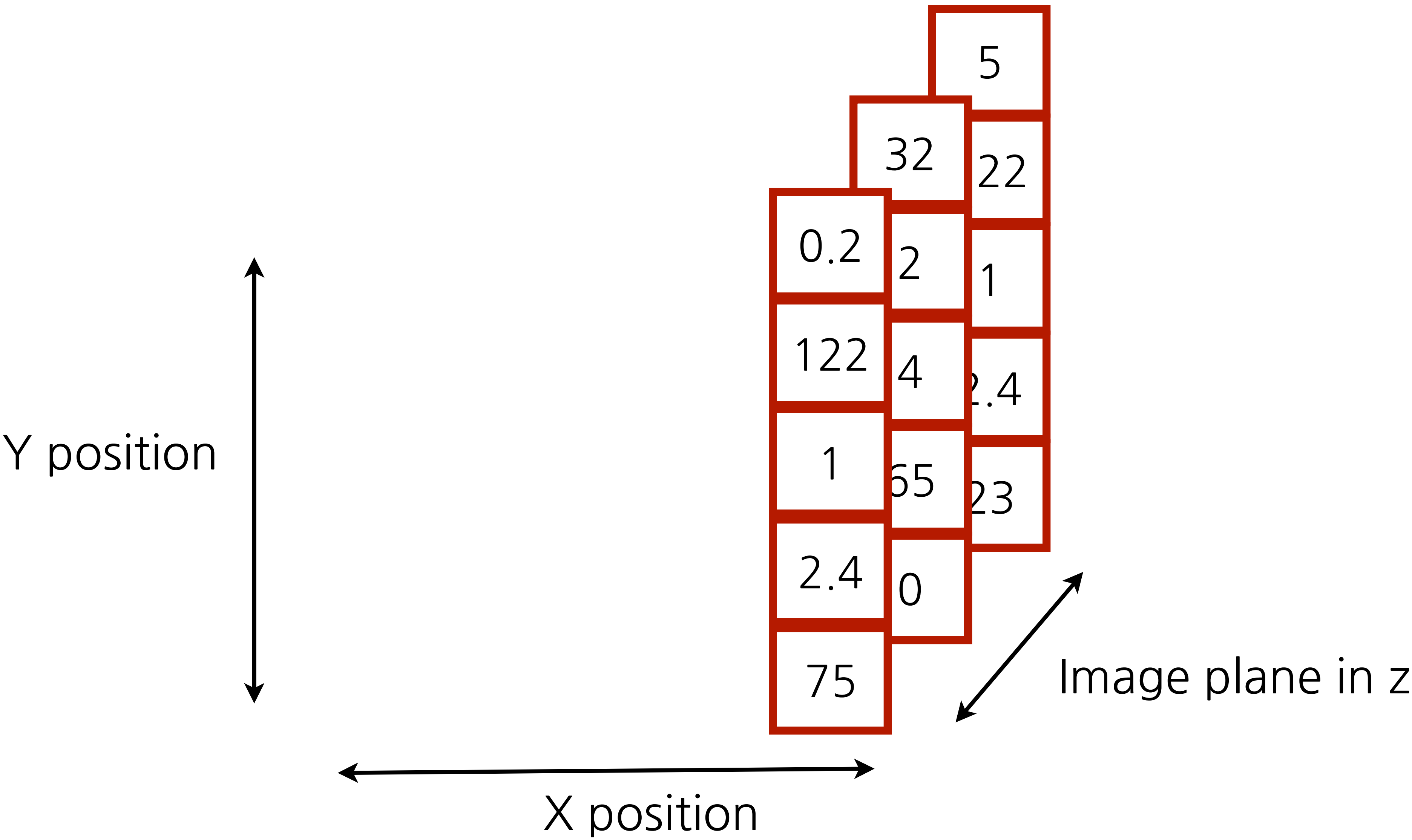
3d image example

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



3d image example

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



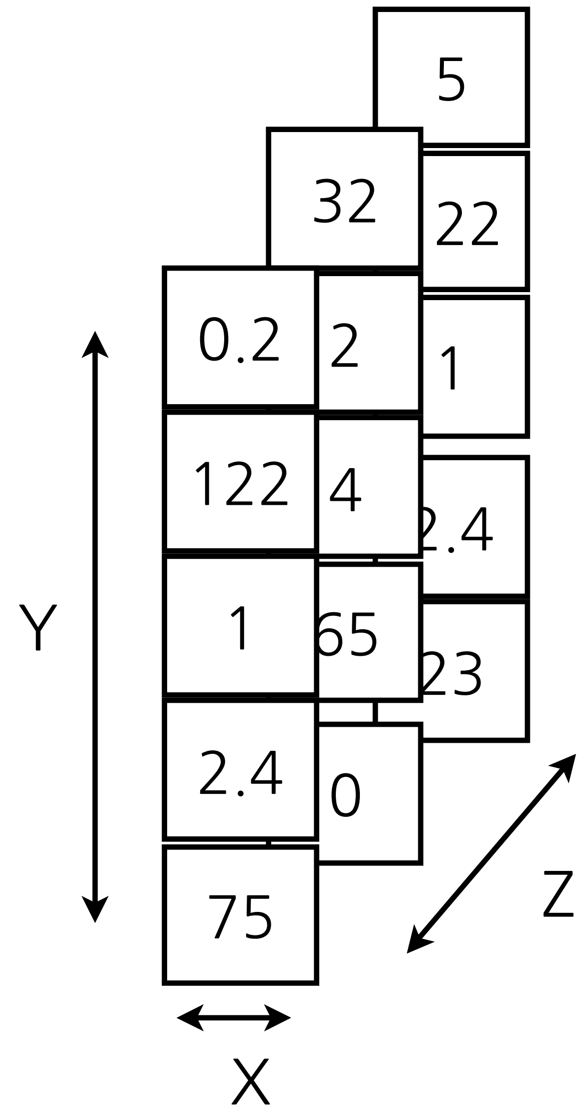
3d image example

What happens if we run `squeeze()`?

Looks at dimension 1 (Y):
- Not length 1, keep it (new dim 1)

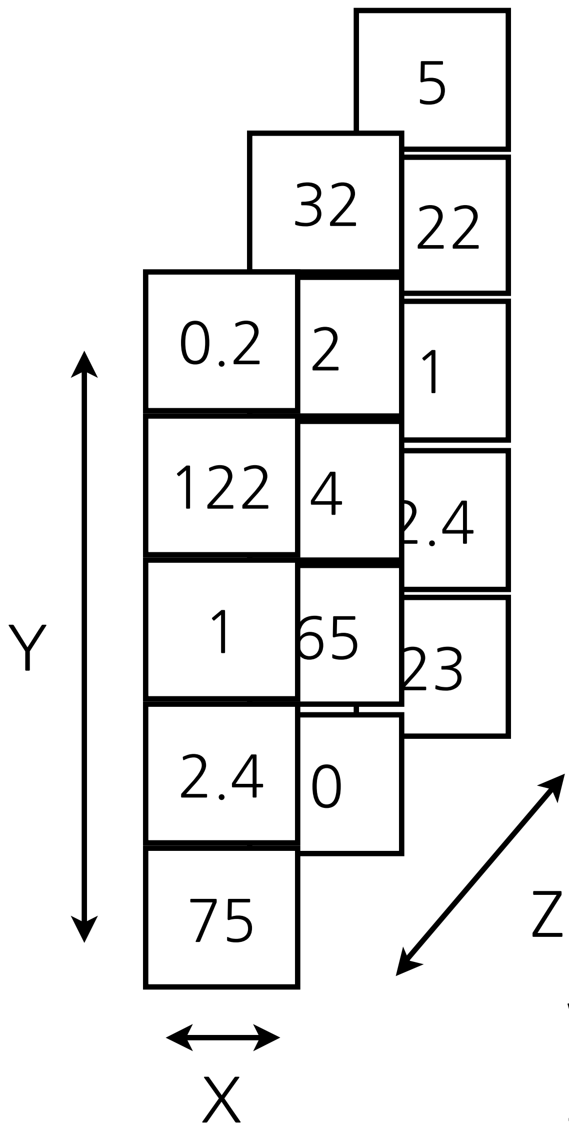
Looks at dimension 2 (X):
- length 1, get rid of this dimension!

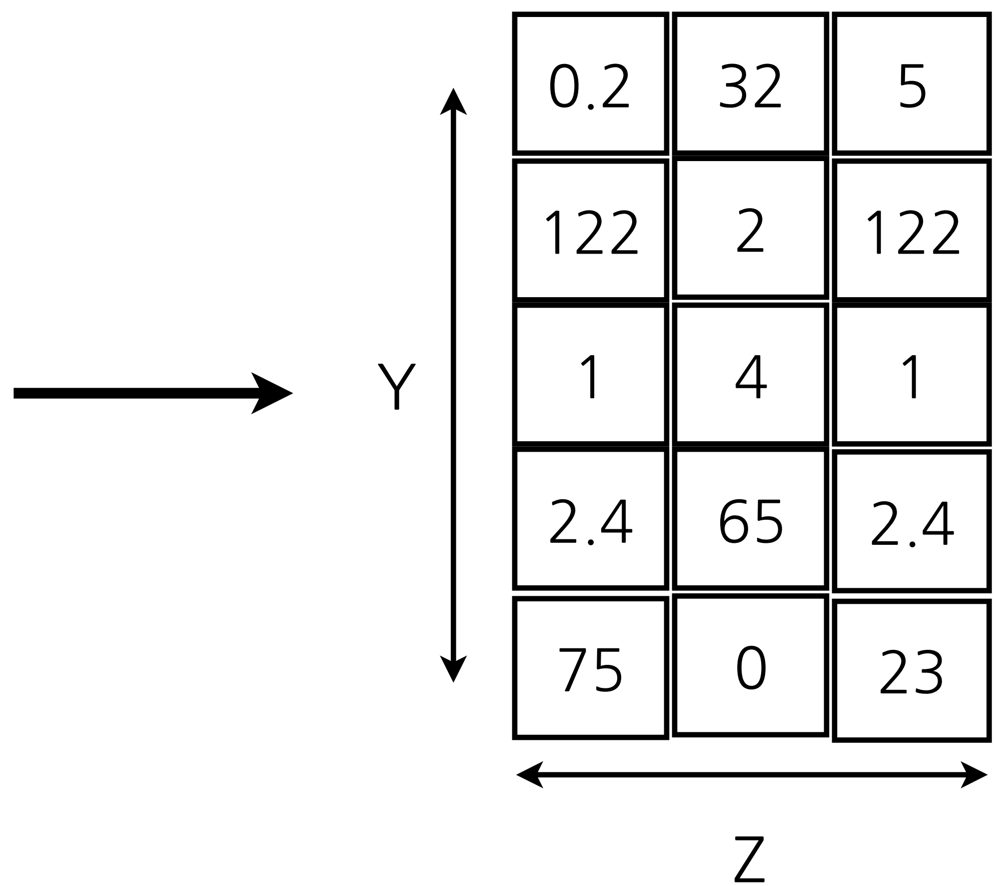
Looks at what was dimension 3 (Z)
- Not length 1, keep it (new dim 2)



3d image example

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



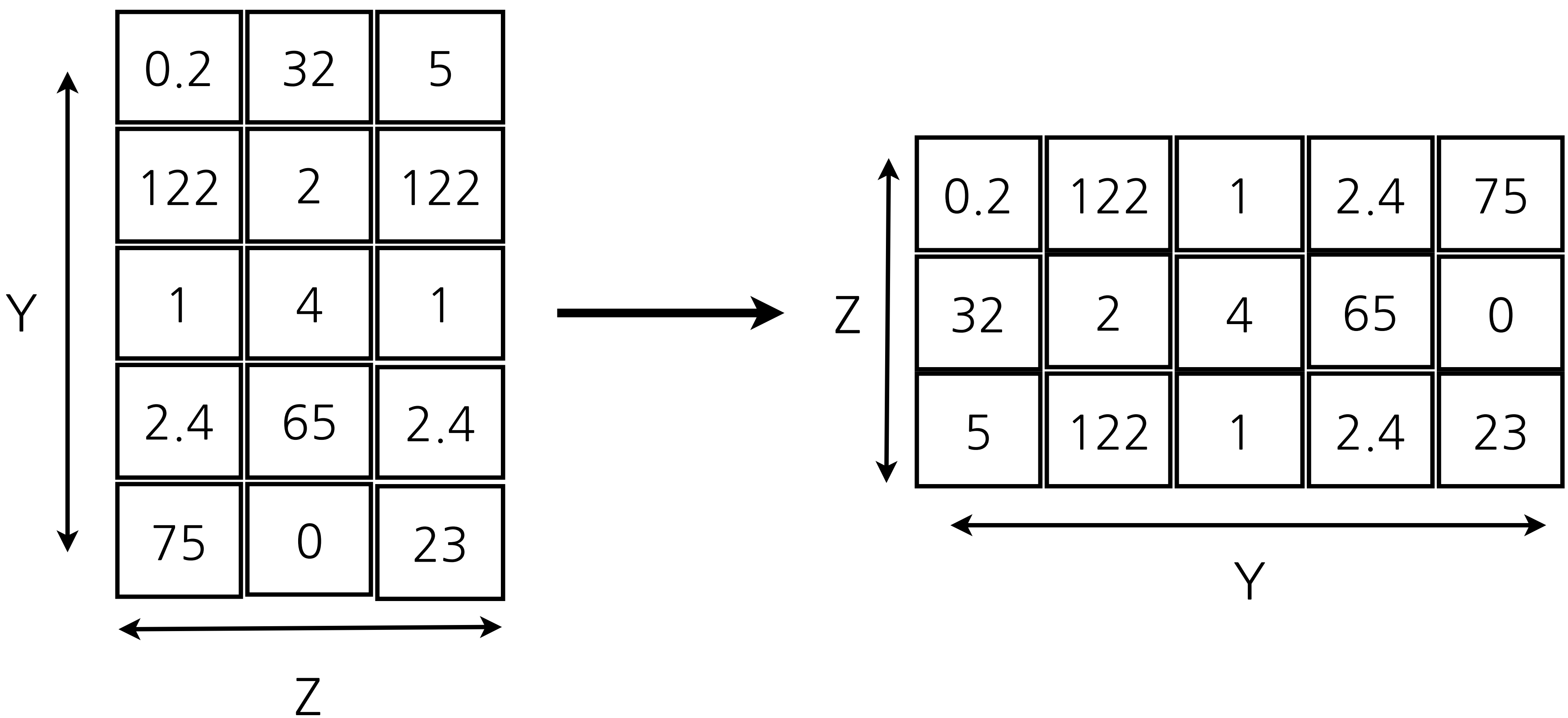


What if we want y on the horizontal,
z on the vertical? i.e. size [z, y] instead

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’
```



Demo revisited: Manipulating MRI Data

Selecting indices automatically

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

We’ll use:

- Booleans
- Conditional operators
- Logical indexing
- find() command

Boolean

Has value true (1) or false (0) and is of class `logical`

`x = true` evaluates to `1`

`x = false` evaluates to `0`

Arrays can consist of booleans

`x = [true, true, false]` evaluates to `1 0 0`

has size `[1, 3]`

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`

double equal means "is equal to?"

`2 == 2` evaluates to `1`

`3 == 2` evaluates to `0`

`3 > 2` evaluates to `1`

`1 > 2` evaluates to `0`

`2 >= 2` evaluates to `1`

"not equal to?"

`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

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

To compare whole matrices, use `isequal(A,B)` function

`isequal(A,B)` returns `0 (false)`

Boolean operators

Allow you to combine multiple logical operations

‘**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]`

Logical indexing

Logical arrays are useful because you can use them directly to index into arrays:

```
vals = [4 2 -1 1 -3];
```

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

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

`vals(indsToSelect)` evaluates to `[4 2 1]`

Logical indexing

STEP 1: Use conditional operators to create a logical array of the same size as the original.

STEP 2: Use logical array to pick out indices that satisfy conditions.

Note: 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 always return `logical` arrays.

Assignment using logical indexing

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

```
vals = [4 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 = [4 2 -1 1 -3];
```

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

Zero values we don't want:

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

`vals` evaluates to `[4 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 = [4 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

`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`

nnz( ) function

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

With logical arrays, useful way to count number of elements that satisfy the conditions:

```
vals = [4 2 -1 1 -3];
```

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

`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]);` note we've *typecast* this vector as a logical

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

`find(idx,1)` evaluates to `[1]`

`find(idx,2,'last')` evaluates to `[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 = [4 2 -1 1 -3];
```

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

`find()` function with higher dim arrays

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; 4; 2]`

Rows on which the values are found

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

Columns in which the values are found

Matched elements in **`i,j`** are indices into the positive element of **`vals`**

Demo: neuron image

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.

Strings

Strings

An array of characters as opposed to numbers

Start and end with single quotes (apostrophe).

```
opsinName = ‘ChR2’;
```

`opsinName(1)` evaluates to `‘C’`

`opsinName(4)` evaluates to `‘2’`

`length(opsinName)` evaluates to `4`

Comparing strings

What happens if we just use the == operator?

Compares the two arrays element-wise

```
channelName = ‘gfp’;
```

`channelName == ‘gfp’` evaluates to `[1 1 1]`

(logical)

`channelName == ‘dapi’`

```
Error using ==> eq
Matrix dimensions must
agree.
```

strcmp() function

Instead, use strcmp to test whether two strings are equal

```
channelName1 = ‘gfp’;
channelName2 = ‘dapi’;
```

`strcmp(channelName1, ‘gfp’)` evaluates to `1` (logical)

`strcmp(channelName2, ‘gfp’)` evaluates to `0` (logical)

Concatenating strings

You can concatenate or join together strings:

- Like you would concatenate a numeric array, by wrapping them in `[ ]` brackets separated by a comma or space

```
prefix = ‘data’;
dayName = ‘20110909’;

fullName = [prefix dayName]
```

evaluates to `‘data20110909’;`

Concatenating strings

You can concatenate or join together strings:

- Like you would concatenate a numeric array, by wrapping them in `[ ]` brackets separated by a comma or space
- Using the `strcat()` function

```
strcat(string1, string2, ...)
```

```
fullName = strcat(prefix, dayName)
```

evaluates to `‘data20110909’;`

Concatenating strings

Be careful with combining strings with numbers.
Use the function `num2str()` to convert numbers to characters before building a string.

```
prefix = ‘data’;
year = 2015; month = 9; day = 29;
```

Use ellipses to continue code on the next line!

```
fullName = [prefix num2str(year) ...
            num2str(month) num2str(day)]
```

evaluates to `‘data20150929’`

num2str() and str2num()

`num2str()` function converts a numeric type (e.g. `double`) into a string representation of that number

`num2str(21)` evaluates to `'21'`

`str2num()` function converts a string representation of a number into a `double`

`str2num('21')` evaluates to `21`

Printing a string

`fprintf()` function prints out a string in the formatting you want.

```
fprintf(‘Hello %s week %i\n’, ‘NENS230’, 2)
```

prints `Hello NENS230 week 2`

`%s` means ‘put a string here’

`%s` means ‘put an integer here’

`\n` means ‘put a new line (“carriage return”) here’

`doc fprintf` is your friend for remembering formatting rules.

Demo: Strings

Data Import

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 by line input

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.

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

These three arguments are optional

If the file 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
```

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

csvread() function

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

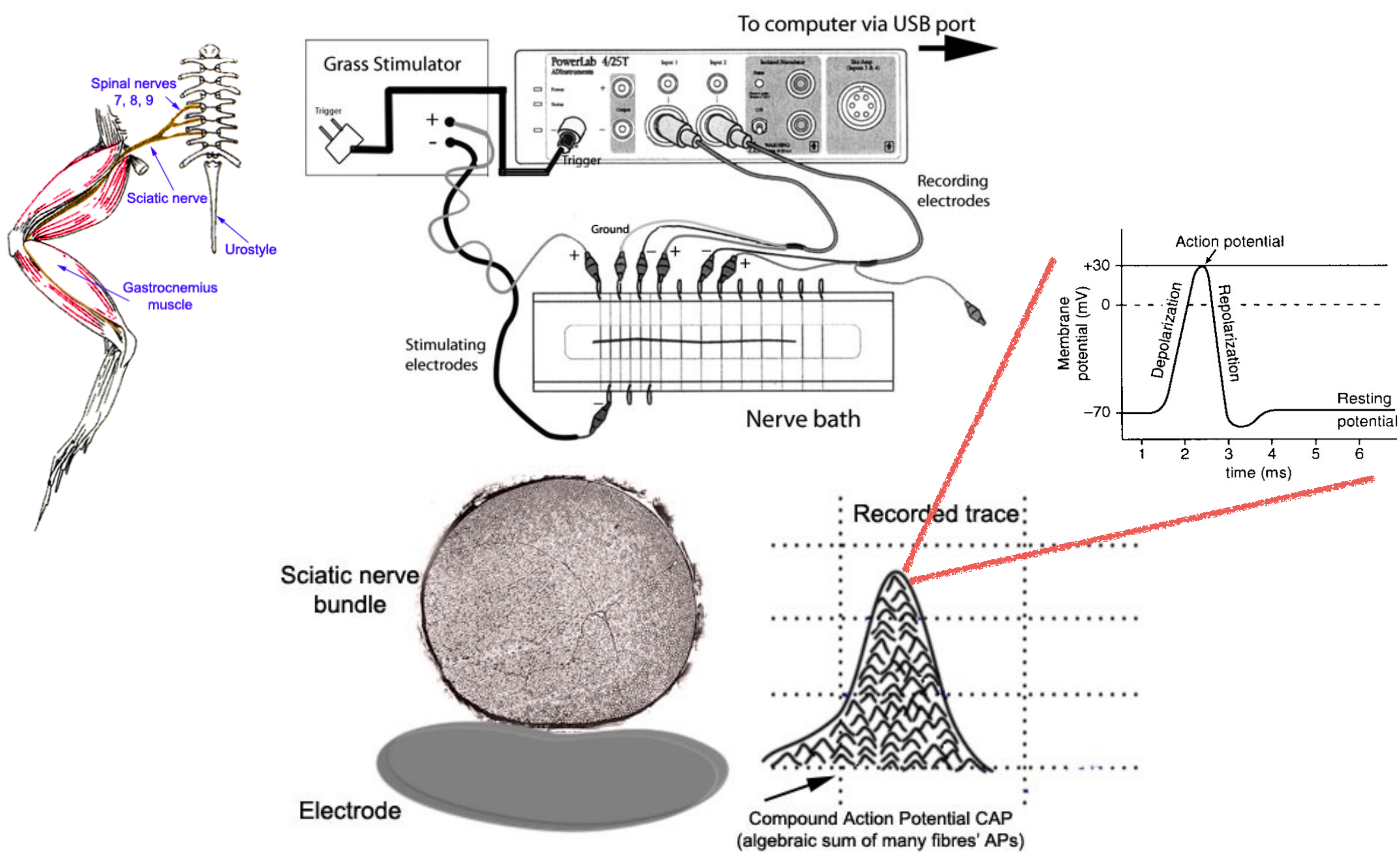
Numeric data as 2d array

Text data as cell array

All data (numeric and text) as cell array

Demo:
Data Import
(stock prices)

Assignment 2: Sciatic Nerve Recordings



Week 2 Assignment

- Data import and processing:
  - Voltage and time (`actionpotential.mat`)
  - Pulse duration and strength (`pulsedata.csv`)
  - Electrode distance and action potential delay (`recordings.mat`)

- Basic signal processing - remove noise from a trace

Review

Concepts	Functions

Data types:
Numerical classes are for storing numbers. Examples of numerical classes are integers, doubles, floats
Logicals are stored as 0 or 1
Strings are for storing text
Elements are accessed/assigned with **indexing rules**
There are other types for more structured data (structures, classes) covered later, 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
	`+ - / *` arithmetic
	`;` suppresses output
	`:` incremental indexing
	`a'` transpose
	`[a b]` concatenates horizontally
	`[a; b]` concatenates vertically
	`a(3:end-1)` indexing
	`a(n)=[]` excises n^{th} element
	`zeros ones` two ways to build a matrix
	`mean`
	`std`
	`plot` basic plot
	`size` dimensions of a variable
	`squeeze` flatten unneeded dimension
	`> < >= <= == ~=` comparators
	`strcmp` compare two strings
	`strcat` concatenate (join) two strings
	`nnz` number of nonzero elements
	`find` find nonzero element(s)
	`load` load a .mat file
	`csvread` read a comma separated file
	`imread` read an image file
	`fprintf` formatted text output
	`num2str` convert number to string
	`str2num` convert string to number