Multidimensional Arrays and Indexing

NENS 230: Analysis Techniques in Neuroscience

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

Outline

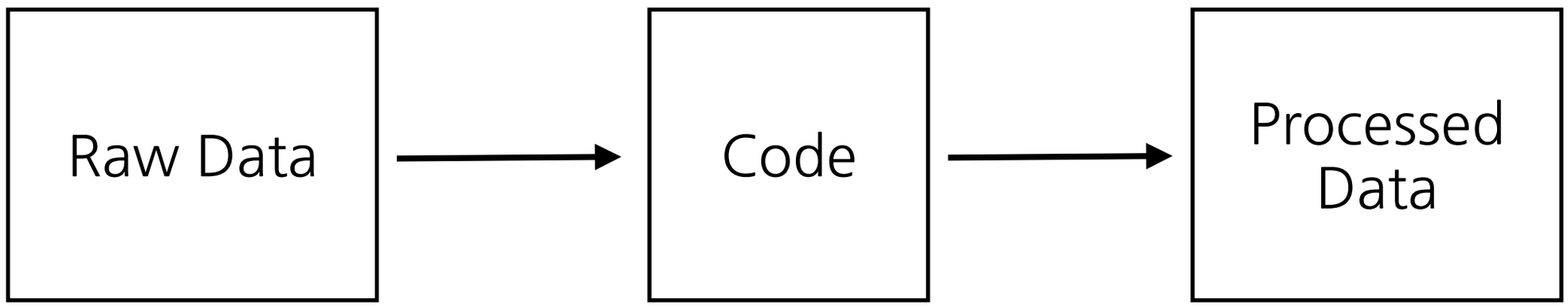
Numeric arrays
- Numeric data types
- Creating multidimensional arrays
- Dimensions have meaning

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

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

Assignment Overview

Why are data types so important?



Examples:

- Voltage clamp traces
  - Current/signal, organized by channel, time
- Image file
  - Intensity, organized by channel, x pos, y pos, z pos

Processed form:

- Opsin tracking by frequency
  - Spikes evoked, organized by pulse frequency
- Cell positions
  - List of x,y,z coordinates for each cell detected

How to organize your data

Input data

- Structure usually determined by source
  - ABF files: signal organized by channel number, time

Intermediate data / final output

- Structure determined by what is most convenient to use in subsequent analyses, plotting, sharing
  - Spike generated (0/1) by frequency, by pulse number
- Often multiple ways to organize things, some better for different purposes

Numeric data types

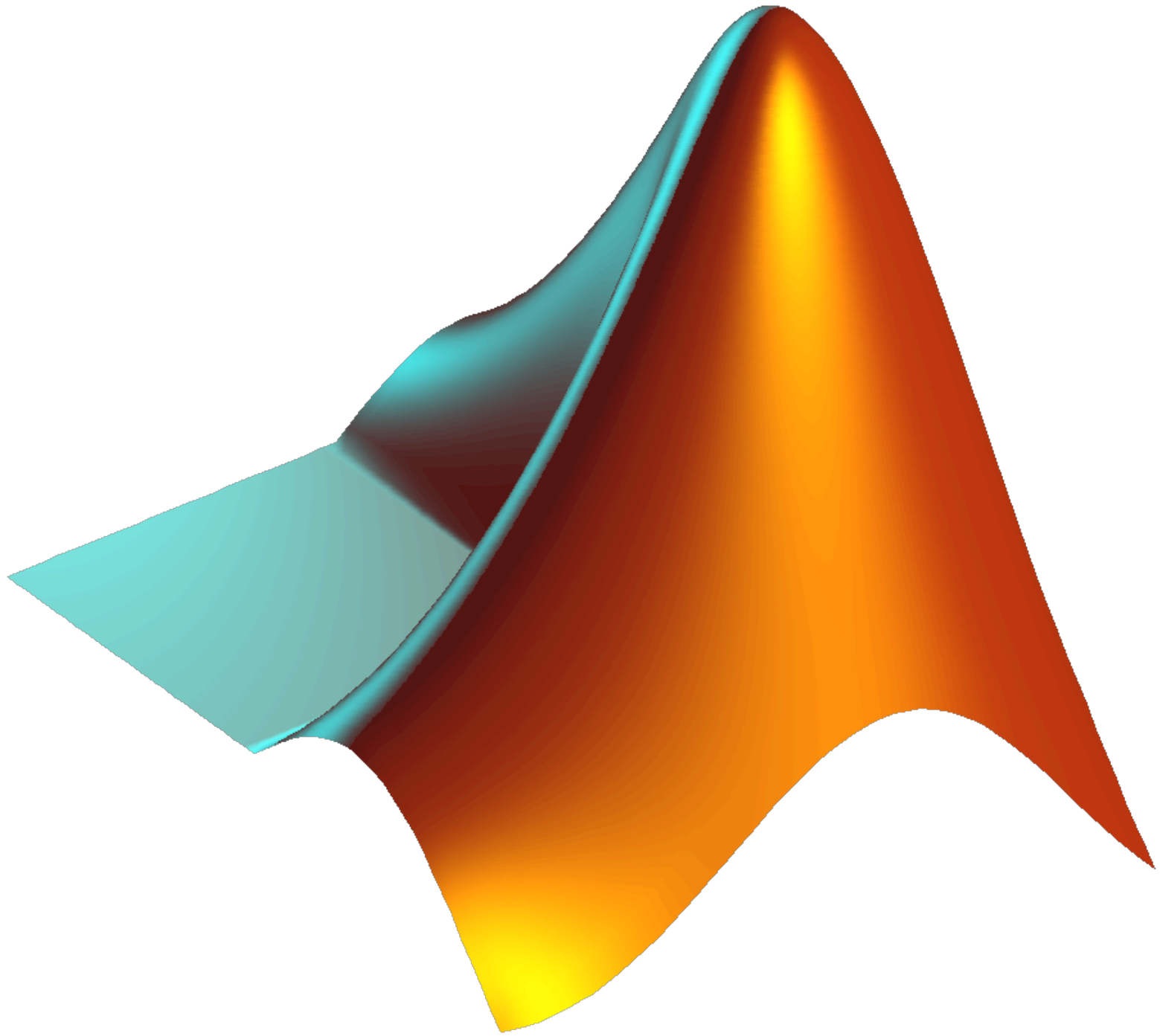
Most common data type: `double`

- Stores floating point values
  - Something like scientific notation
  - Precision varies
- Each value uses 64 bits or 8 bytes
  - But, 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 byte floating point
- `int8, uint8, int16, uint16, int32, uint32, int64, uint64` - signed and unsigned integers

Demo: Assigning numeric values



Arrays / matrices

MATLAB = Matrix Laboratory

- Every data type is actually a matrix
- Here, this means that you can have multiple, identical data “slots” extending along multiple dimensions
- Size is always listed as rows (dim 1), columns (dim 2), size in dim 3, size in dim 4, size in dim 5, etc.

Data 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

Data size examples

Matrix: size is 5, 5 or 5 rows, 5 columns

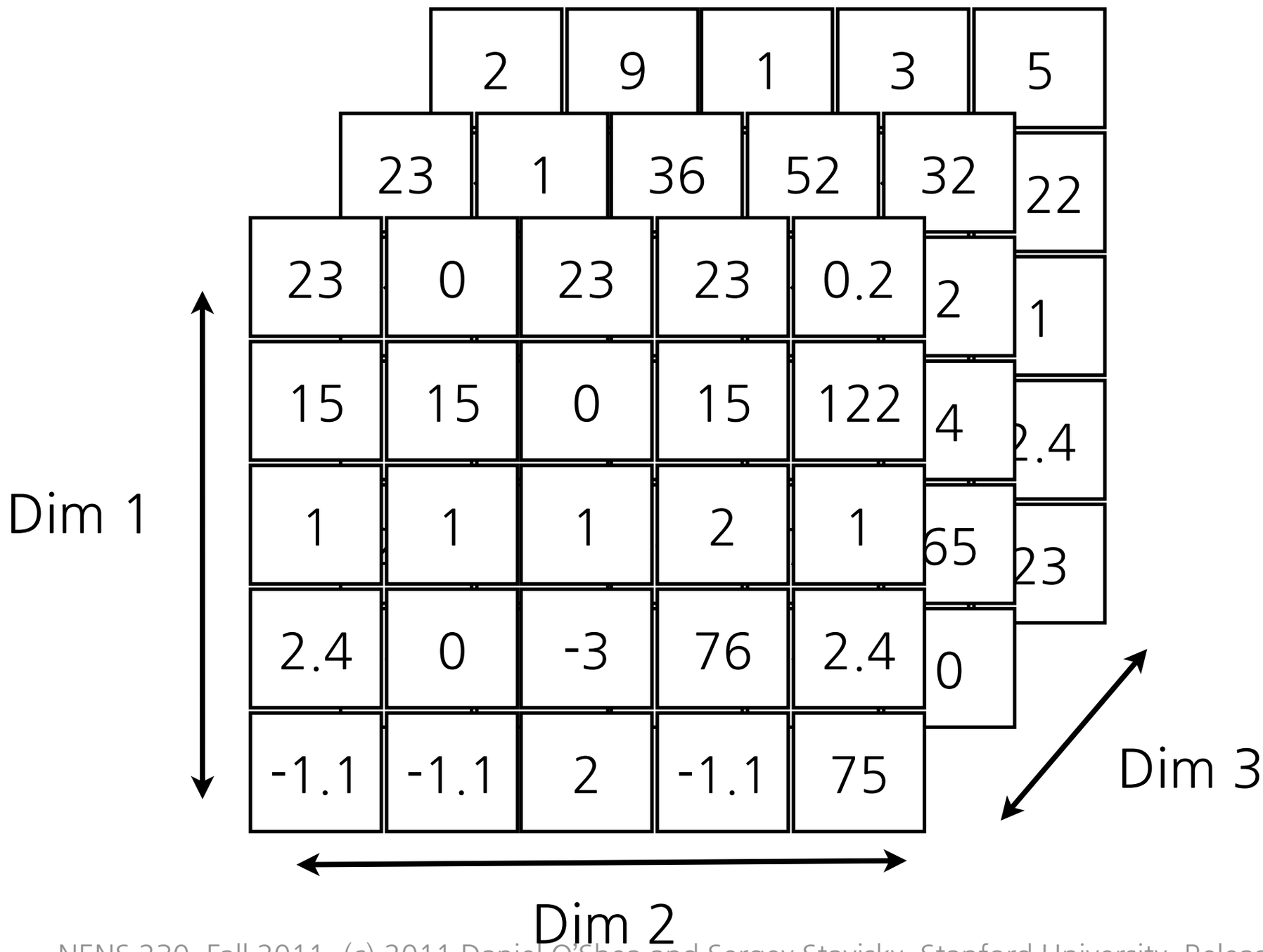
Dim 1

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

Data size examples

3-dimensional array: size is 5, 5, 3 or 5 rows, 5 columns, 3 pages



Useful functions for arrays

`size(array)`

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

`size(`

23	0
15	15
1	1

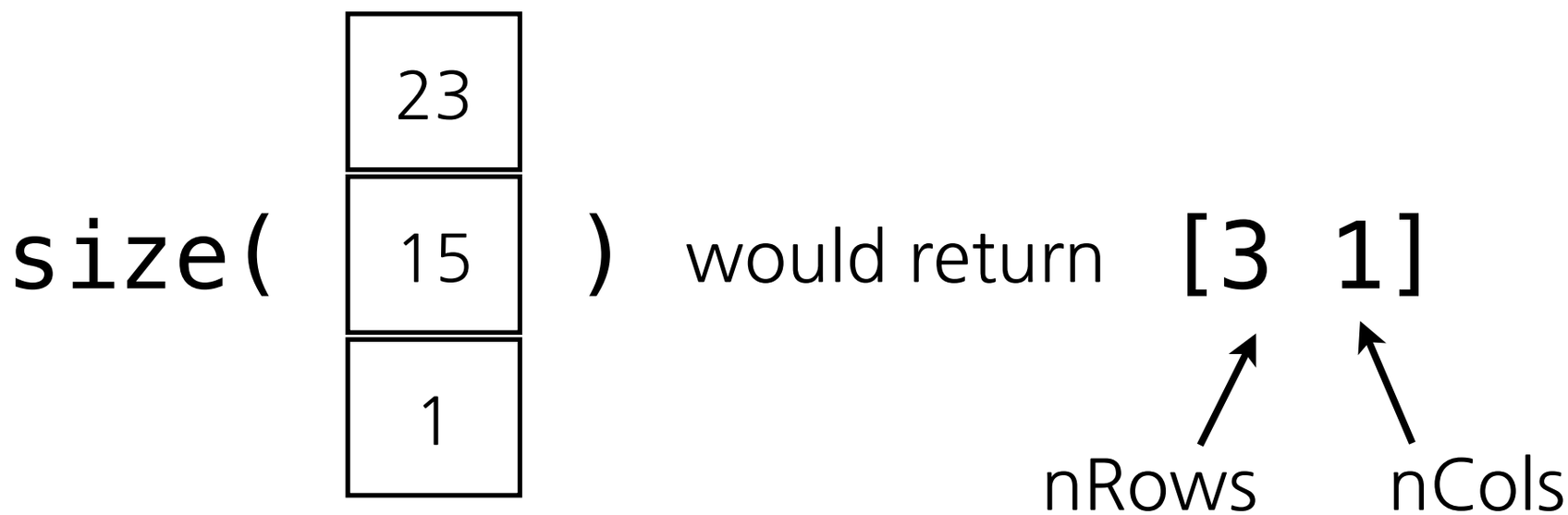
`)` would return `[3 2]`

nRows nCols

Useful functions for arrays

`size(array)`

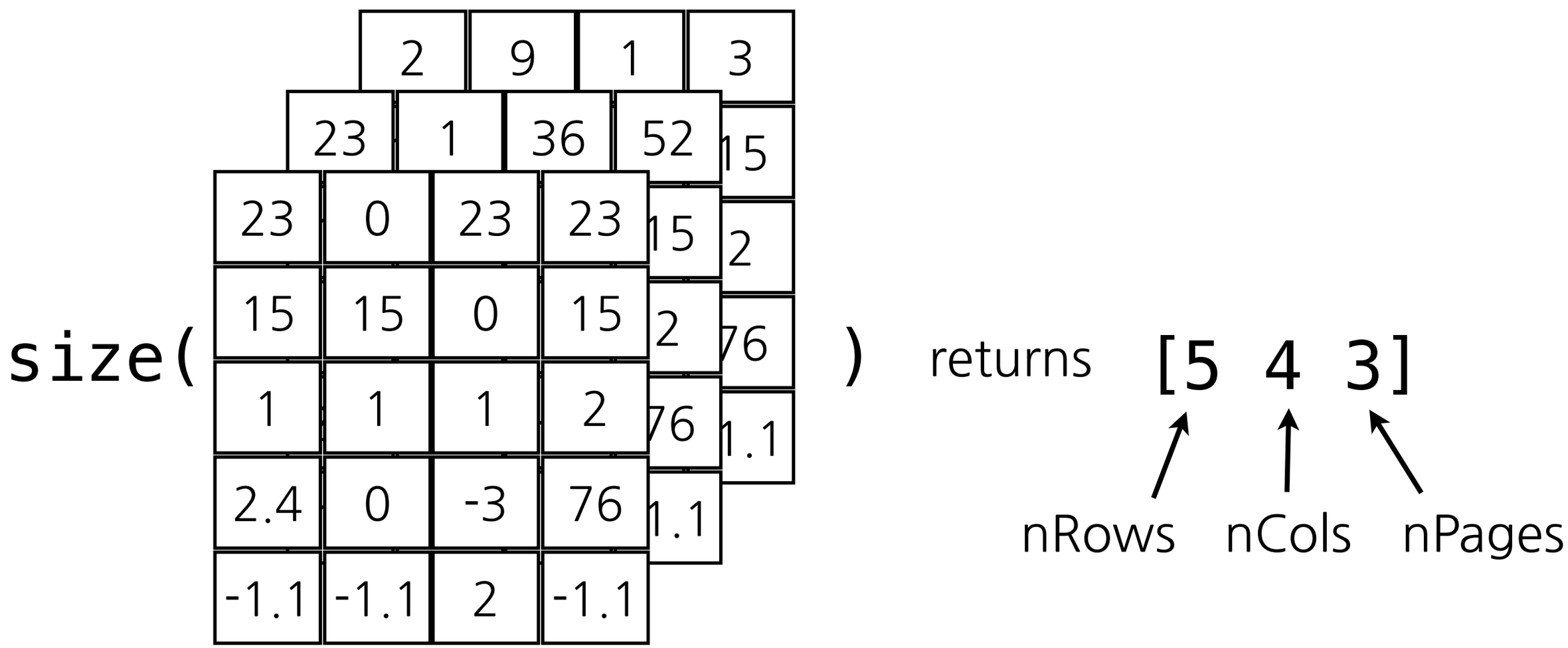
- returns the number of elements in each dimension of the array
- Example:



Useful functions for arrays

`size(array)`

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



Useful functions for arrays

```
zeros(nRows, nCols, ...)
ones(nRows, nCols, ...)
```

- Creates a matrix with nRows rows, nCols cols
- Include additional arguments to create a 3-dimensional, 4-dimensional, n-dimensional etc. array
- Include at least the first two arguments or you'll get something that's nRows x nRows
- Example:

`zeros(3, 2)` returns

0	0
0	0
0	0

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:

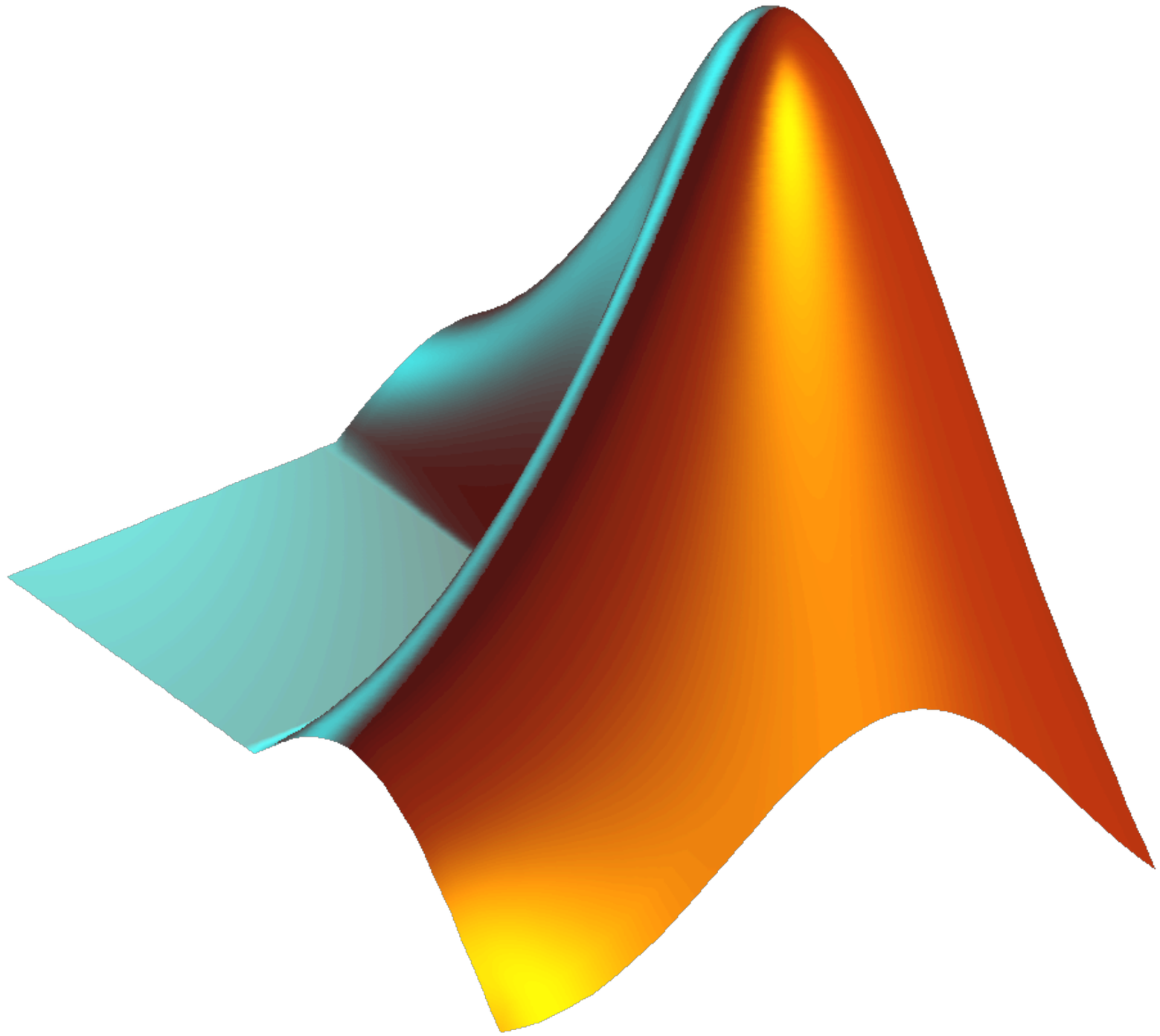
```
[1 2 3; 4 5 6; 7 8 9; 10 11 12]
```

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: Creating multidimensional arrays



Dimensions have meaning

What is each dimension named?

- Decide ahead of time what each dimension means
- Ask what is found in row i, column j, etc.
- Functions often describe the inputs and outputs by explaining the “shape” of the variables and what the dimensions mean

Examples:

- Image - row i, column j, page k is the intensity of a pixel in 3d image at x=i, y=j, slice number=k
- Voltage trace - row i, column j is the voltage on channel i, at time j

Axoclamp data

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

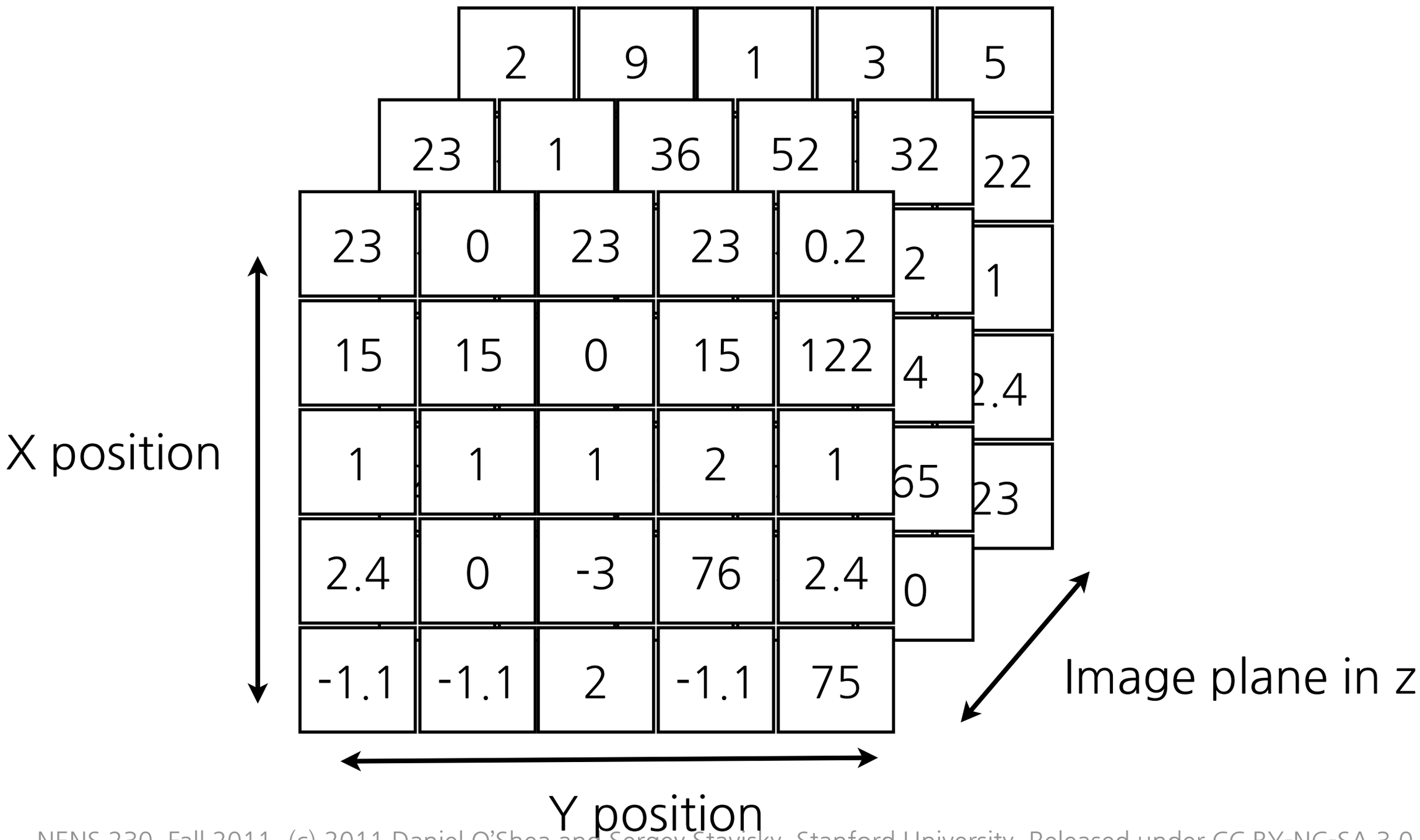
Trials

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

3d image example

3-dimensional array: size is 5, 5, 3 or 5 rows, 5 columns, 3 pages



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

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) ==
```

23

```
a(2) ==
```

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

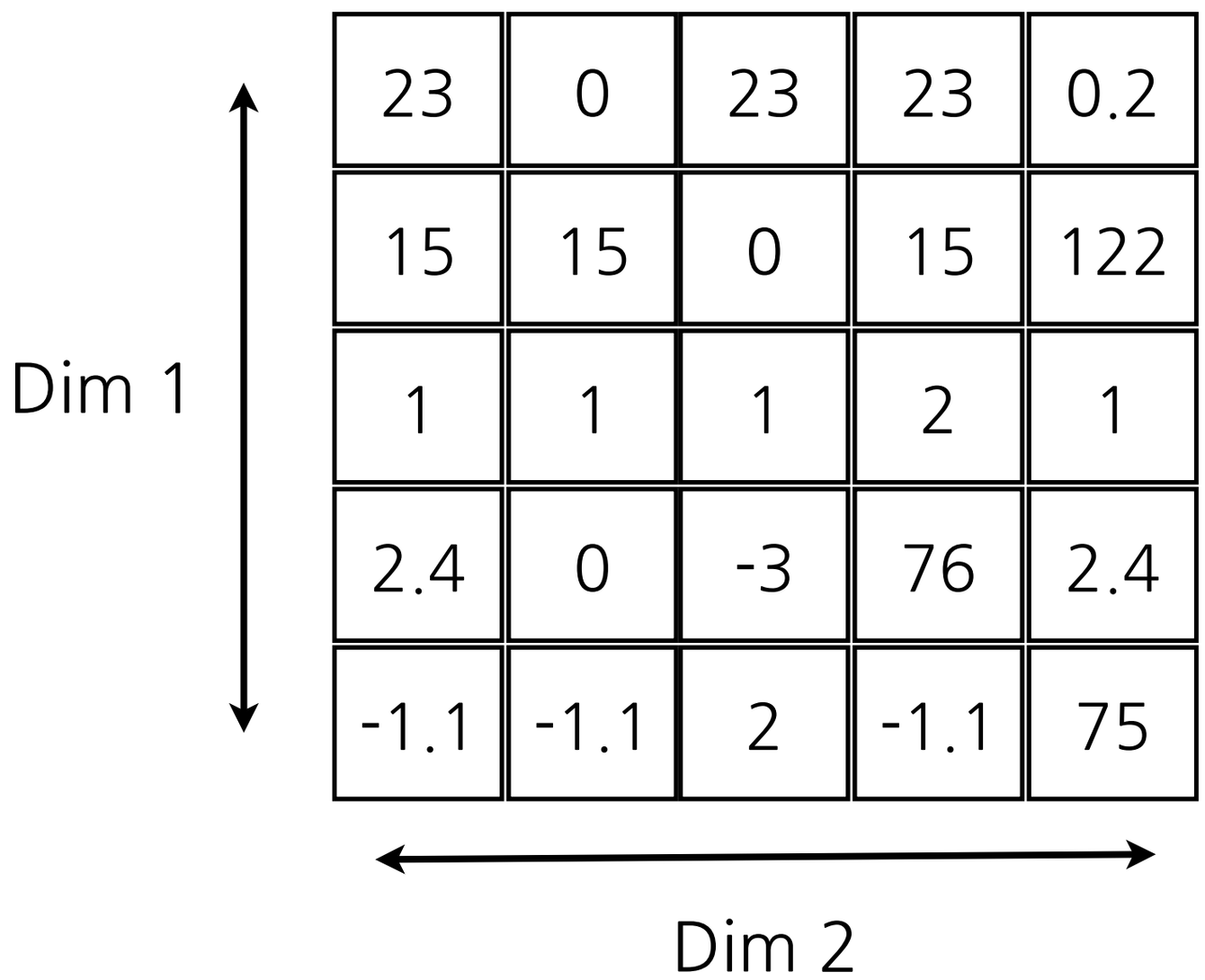
Within the parentheses, include indices for each dimension, separated by commas

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



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

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 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

	23	0	23	23	0.2	…	23	23	0.2
	15	15	0	15	122	…	0	15	122
Trials	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

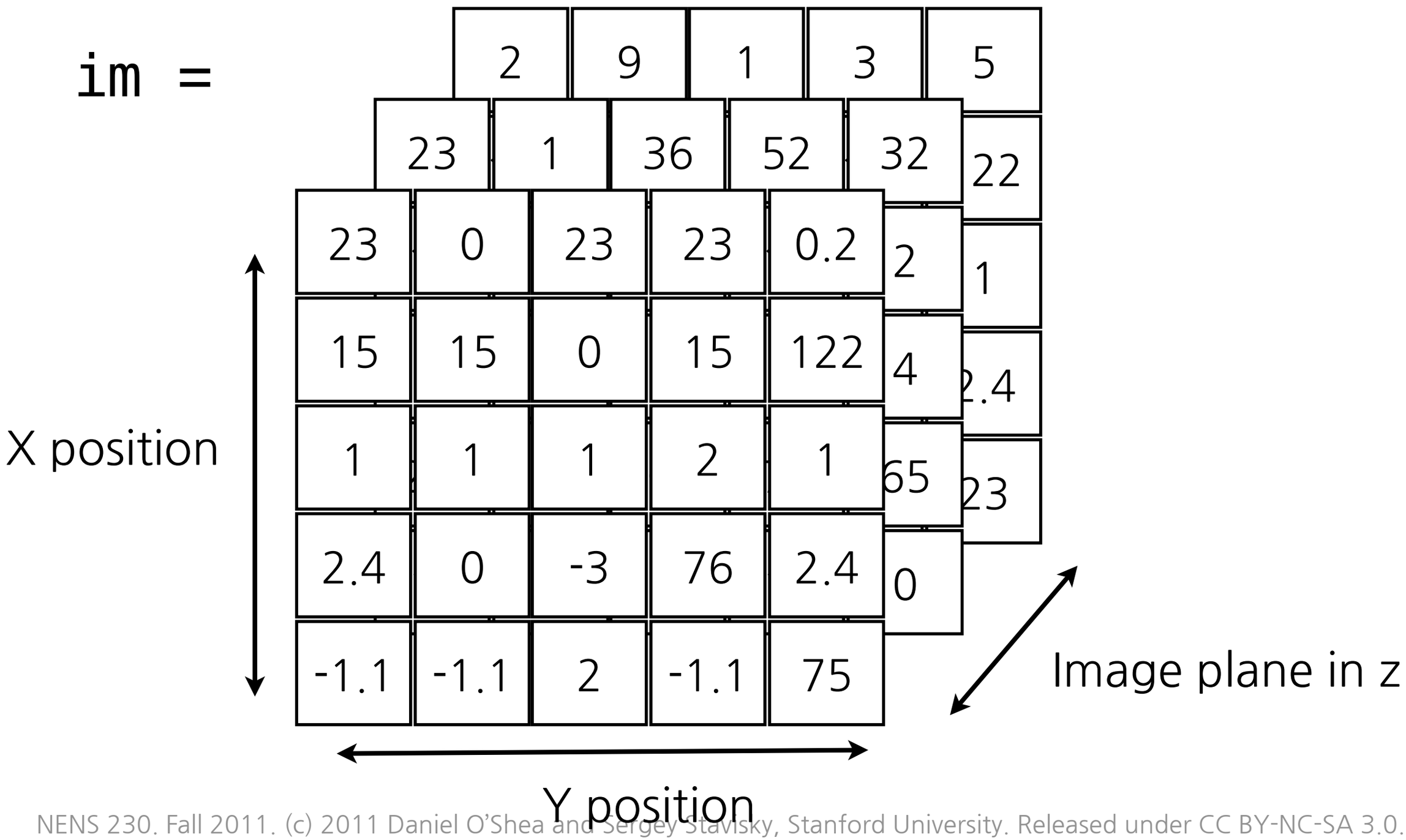
`data(1,:) ==`

23	0	23	23	0.2	…	23	23	0.2

(trial 1)

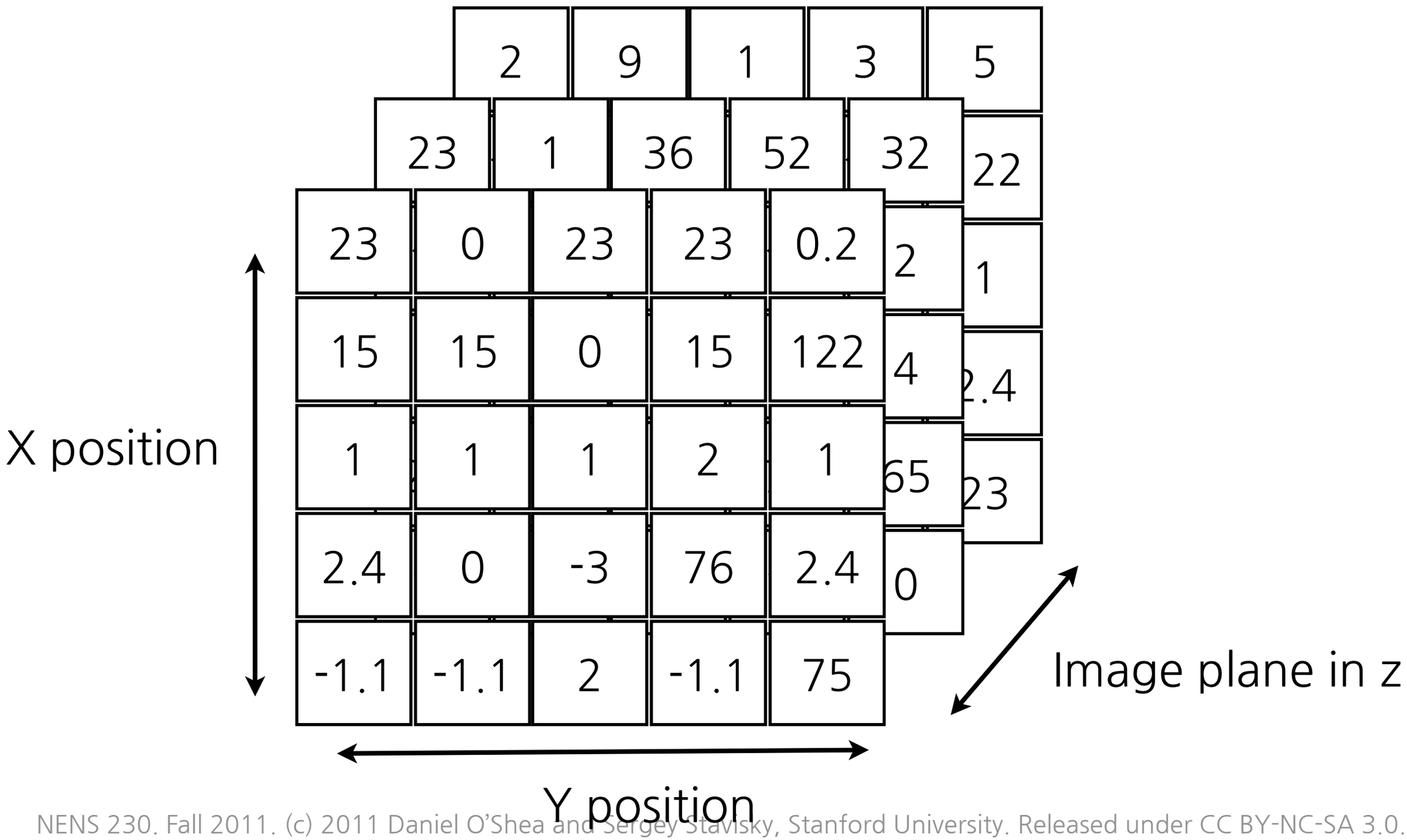
3d image example

3-dimensional image stack



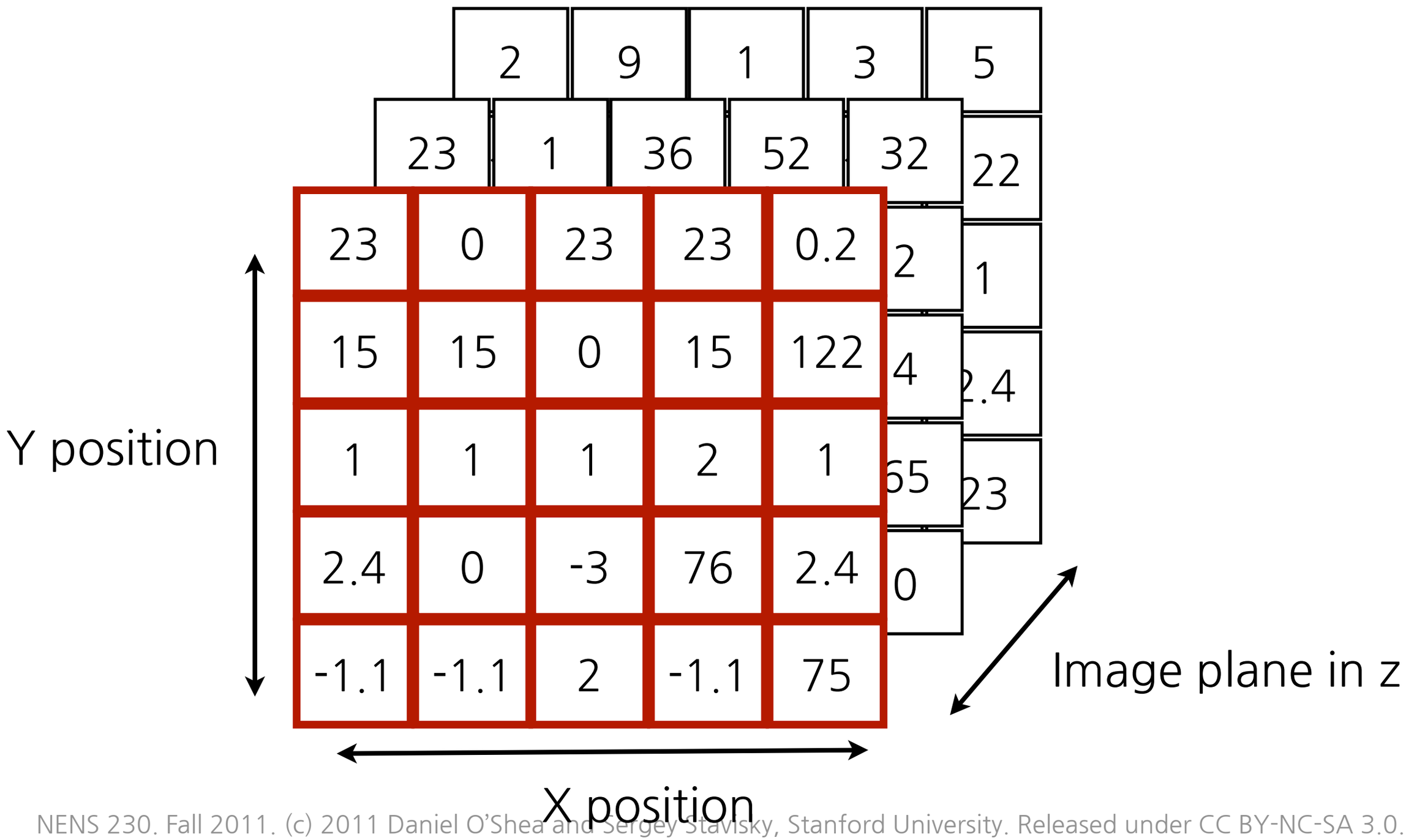
3d image example

How do we grab image 1 of the stack?



3d image example

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



3d image example

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

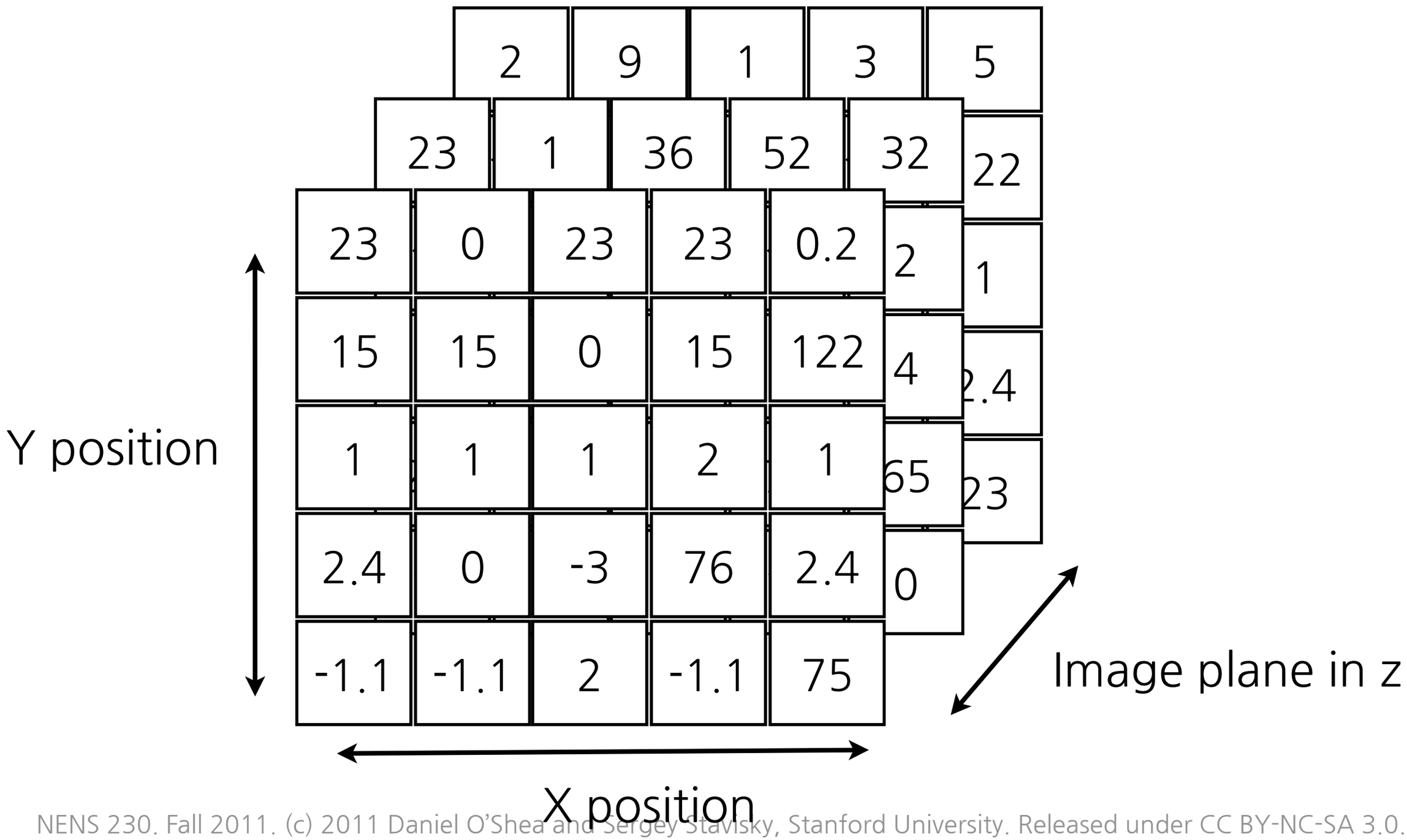
Y position

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

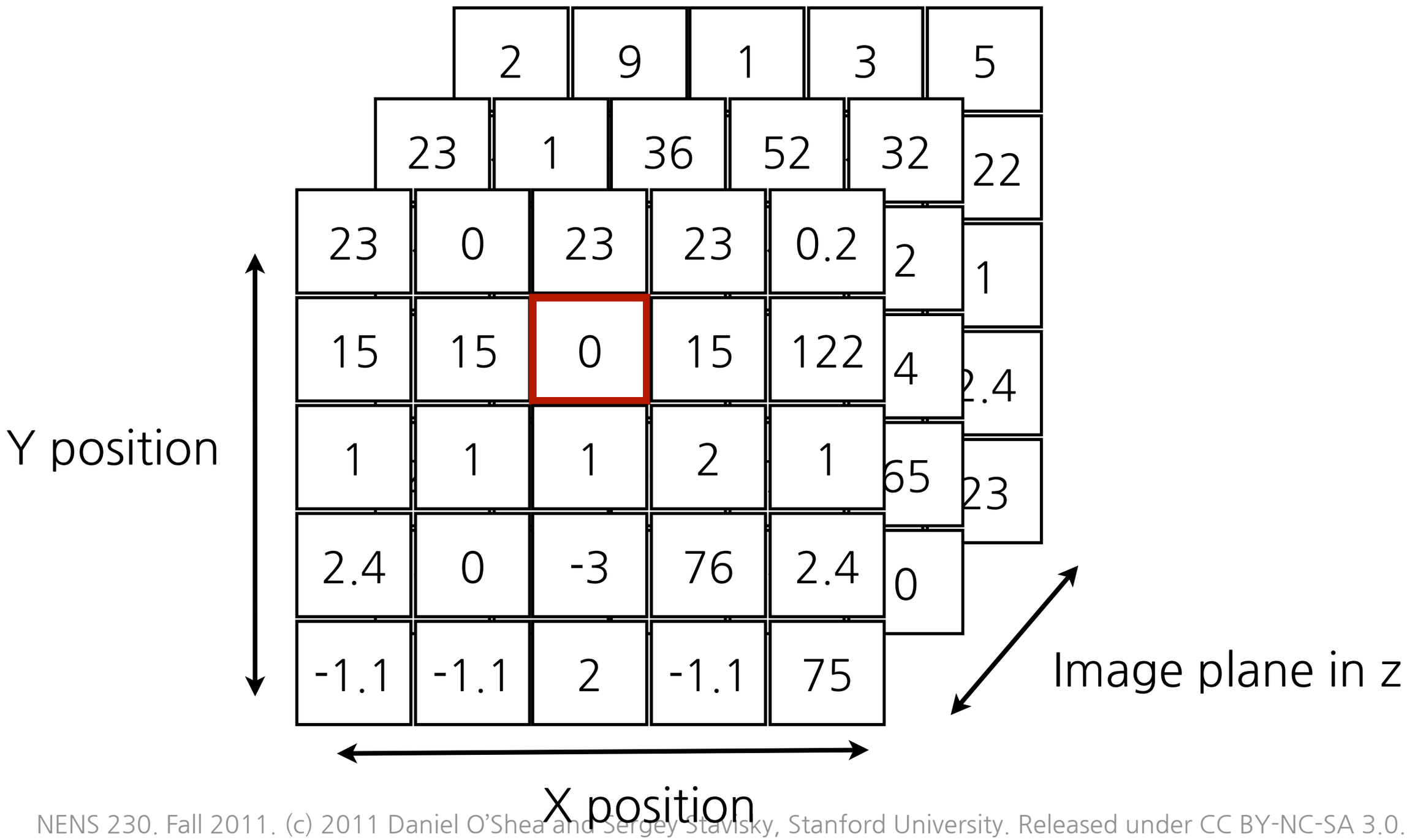
3d image example

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



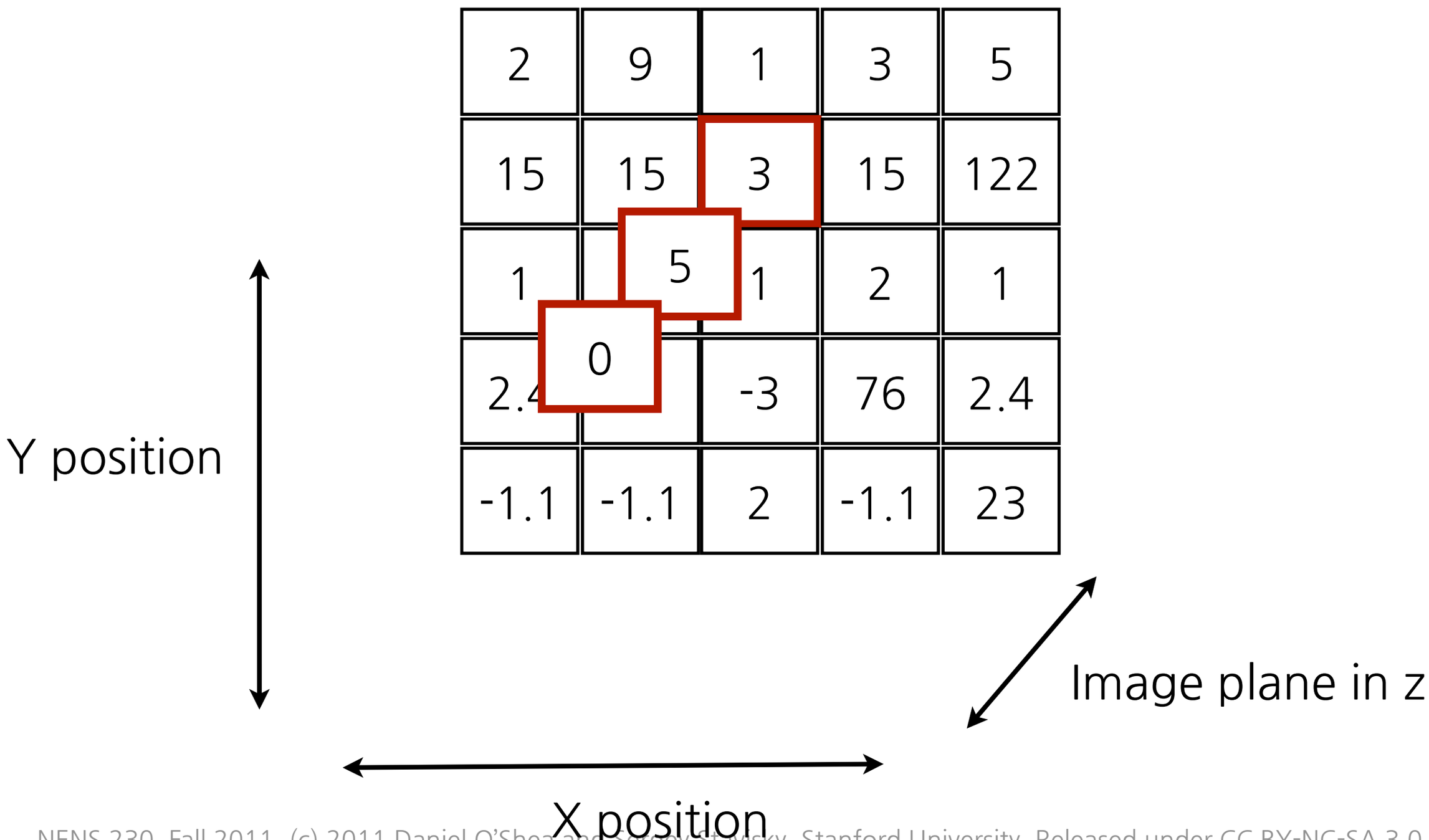
3d image example

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



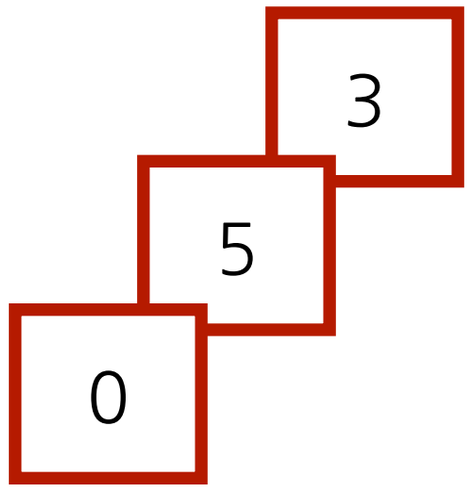
3d image example

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



3d image example

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



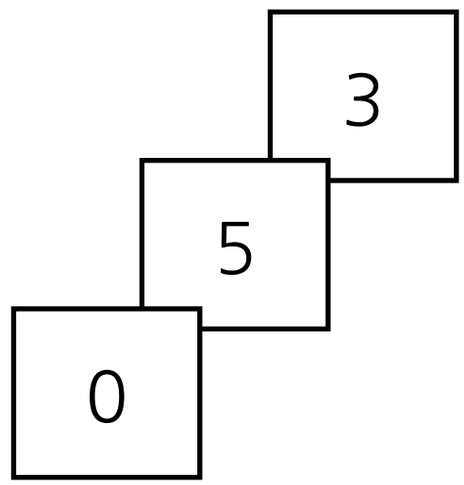

```
zProfile = im(2,3,:);

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

This is an unwieldy “shape” for this vector…

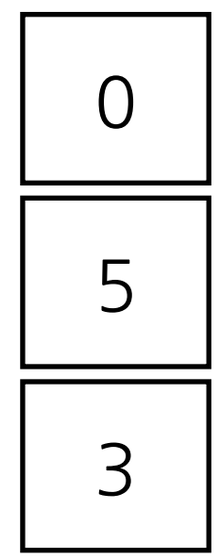
squeeze() function

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 ==
```



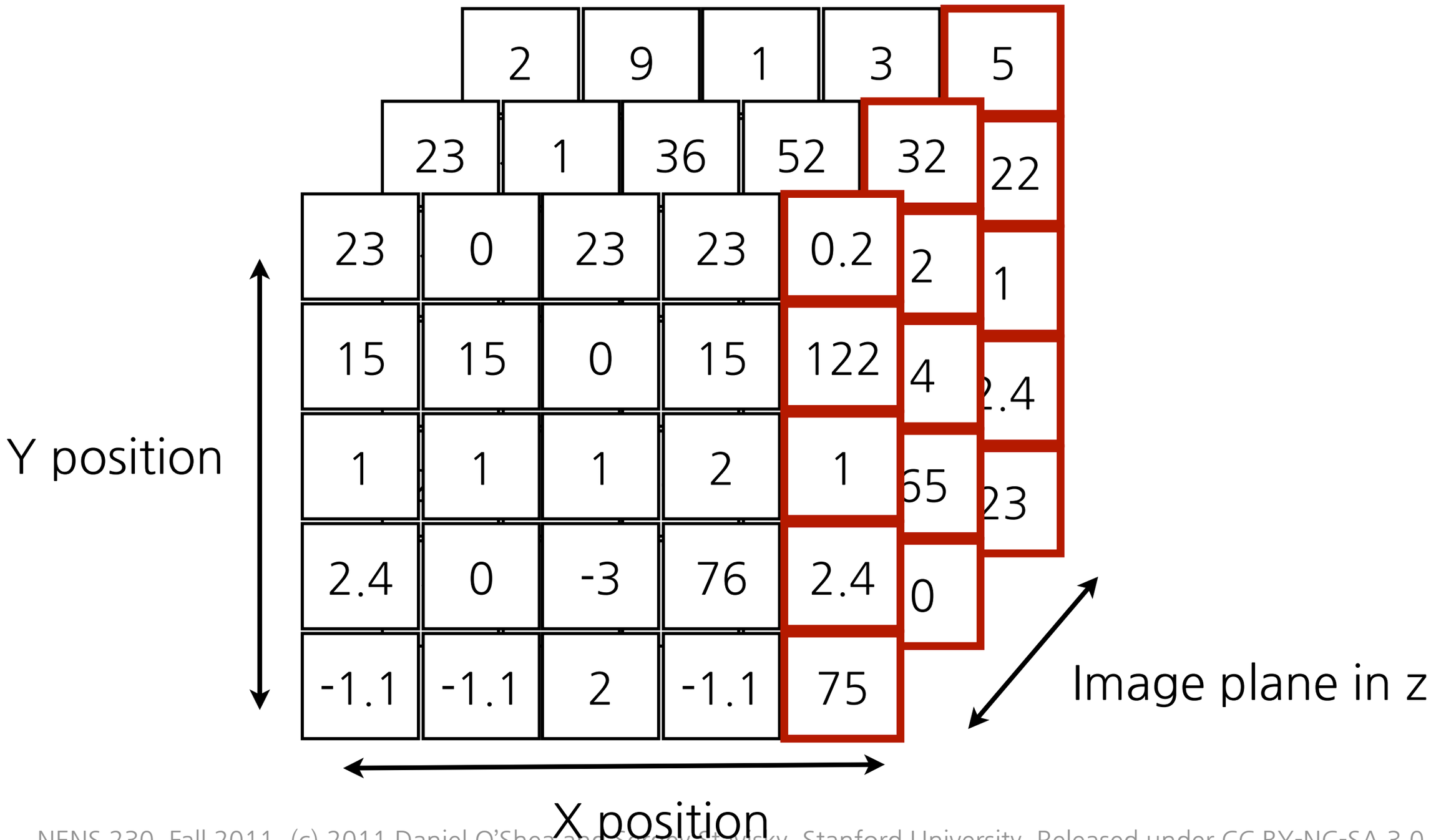

```
zProfile = squeeze(zProfile)
```




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

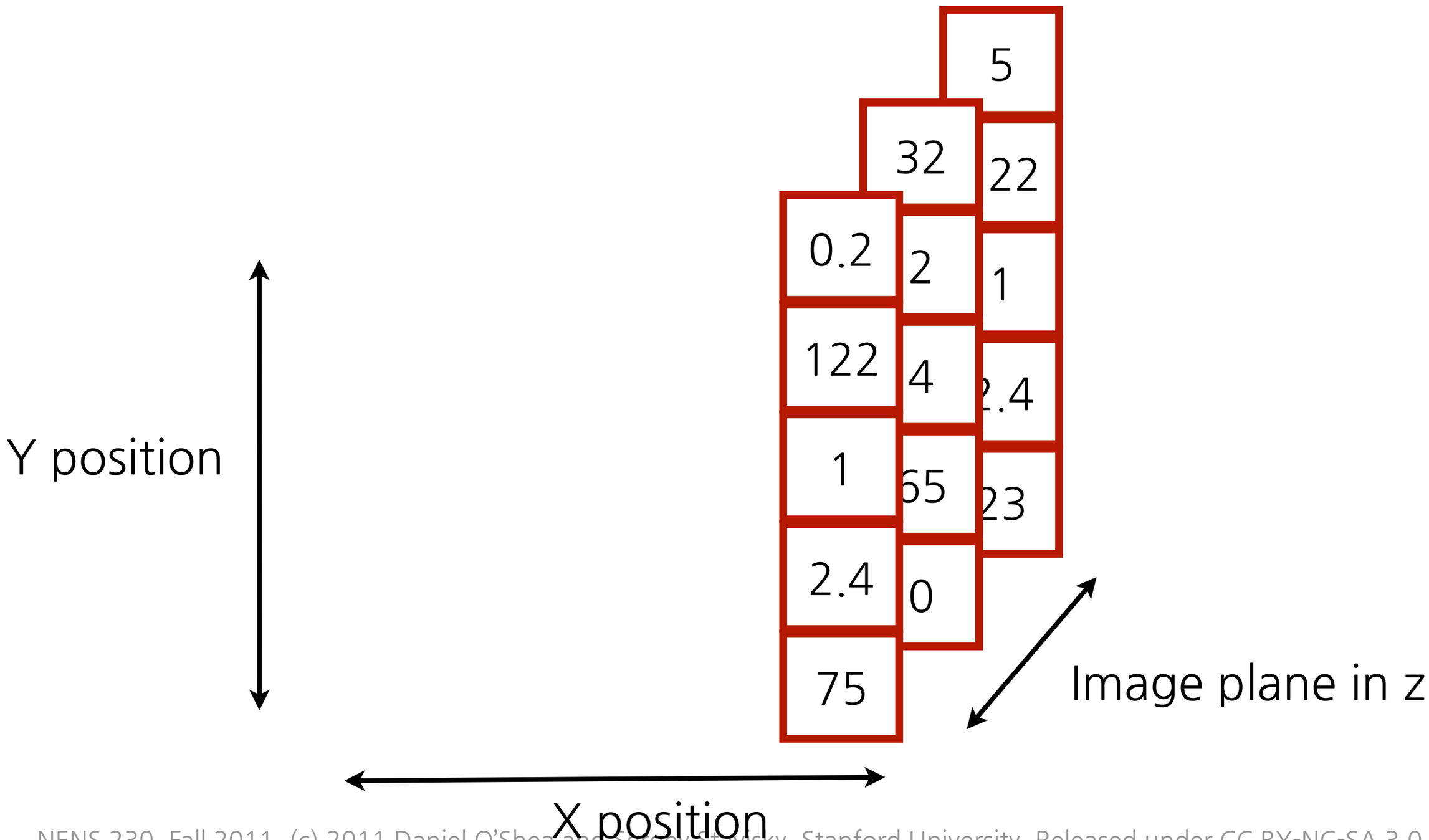
3d image example

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



3d image example

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



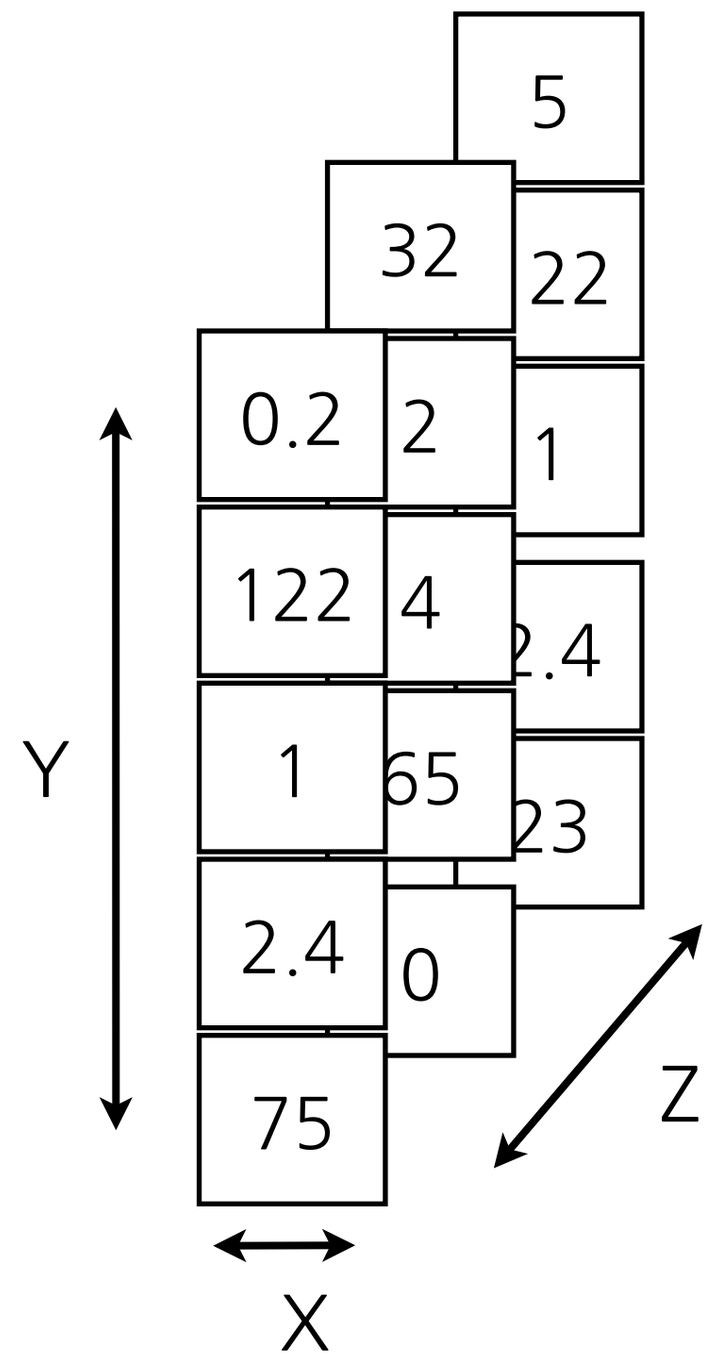
3d image example

What happens if we run `squeeze()`?

Looks at dimension 1 (Y):
- Not length 1, move on

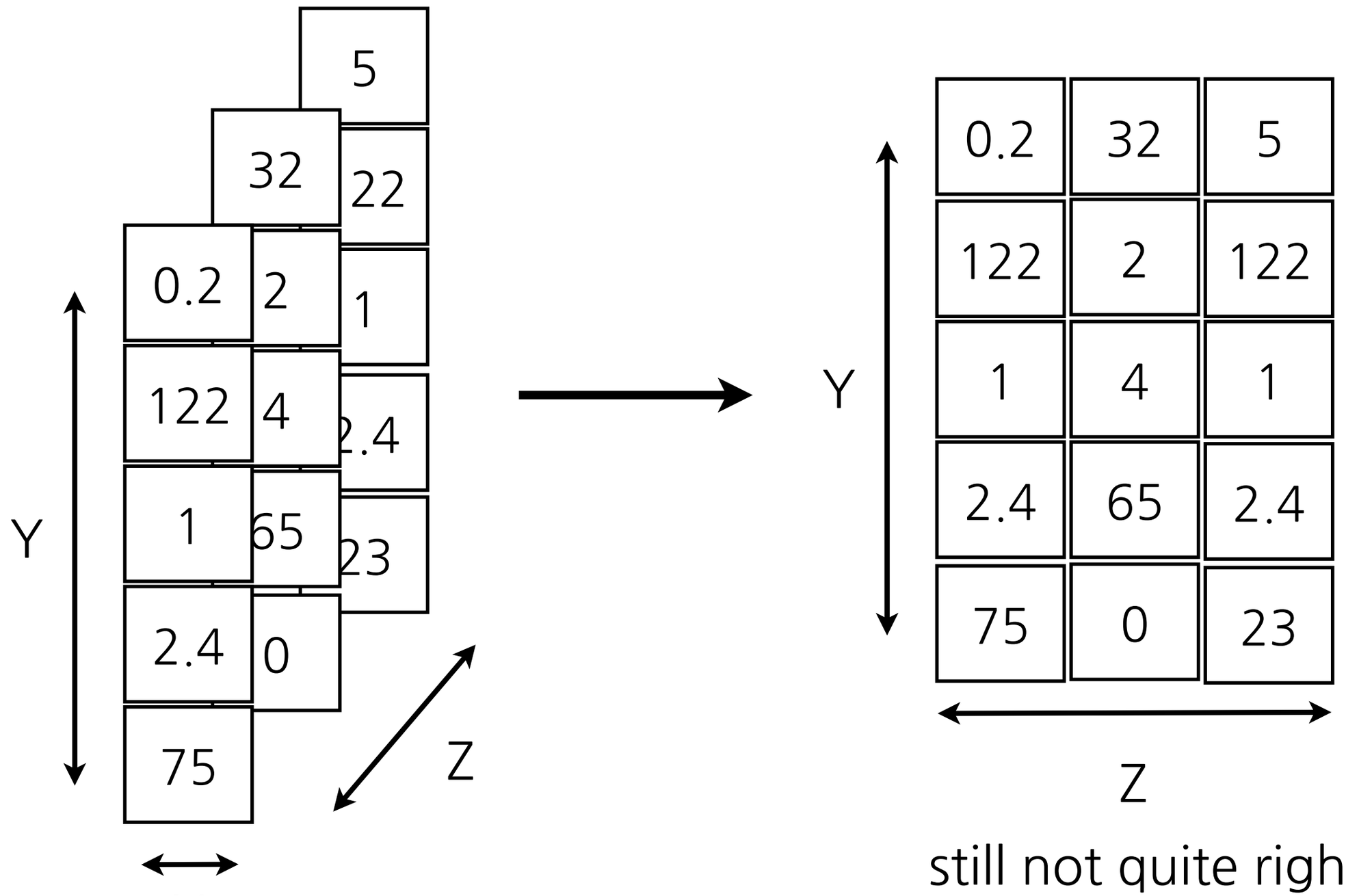
Looks at dimension 2 (X):
- length 1, get rid of this dimension!

Looks at what was dimension 3 (Z)
- Not length 1, move on



3d image example

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

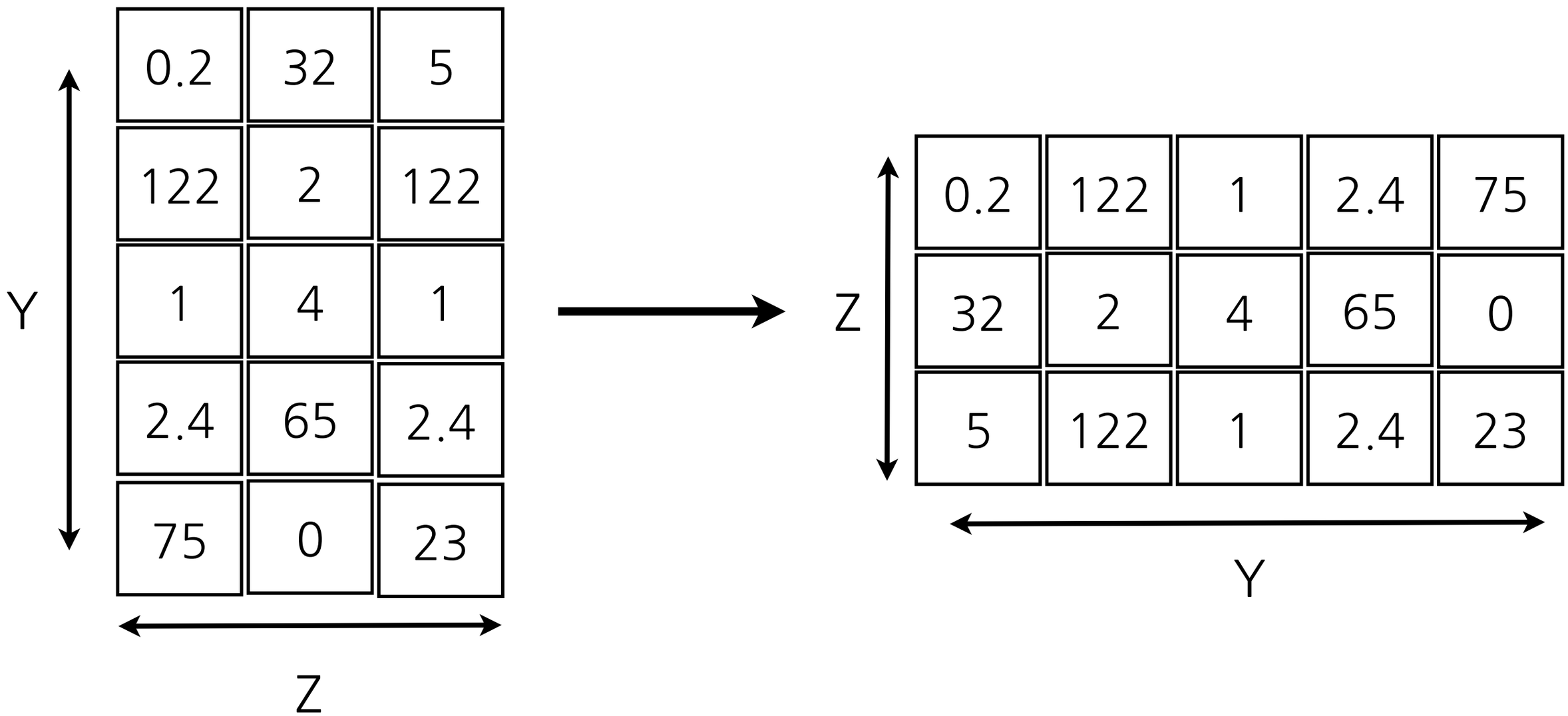


still not quite right...

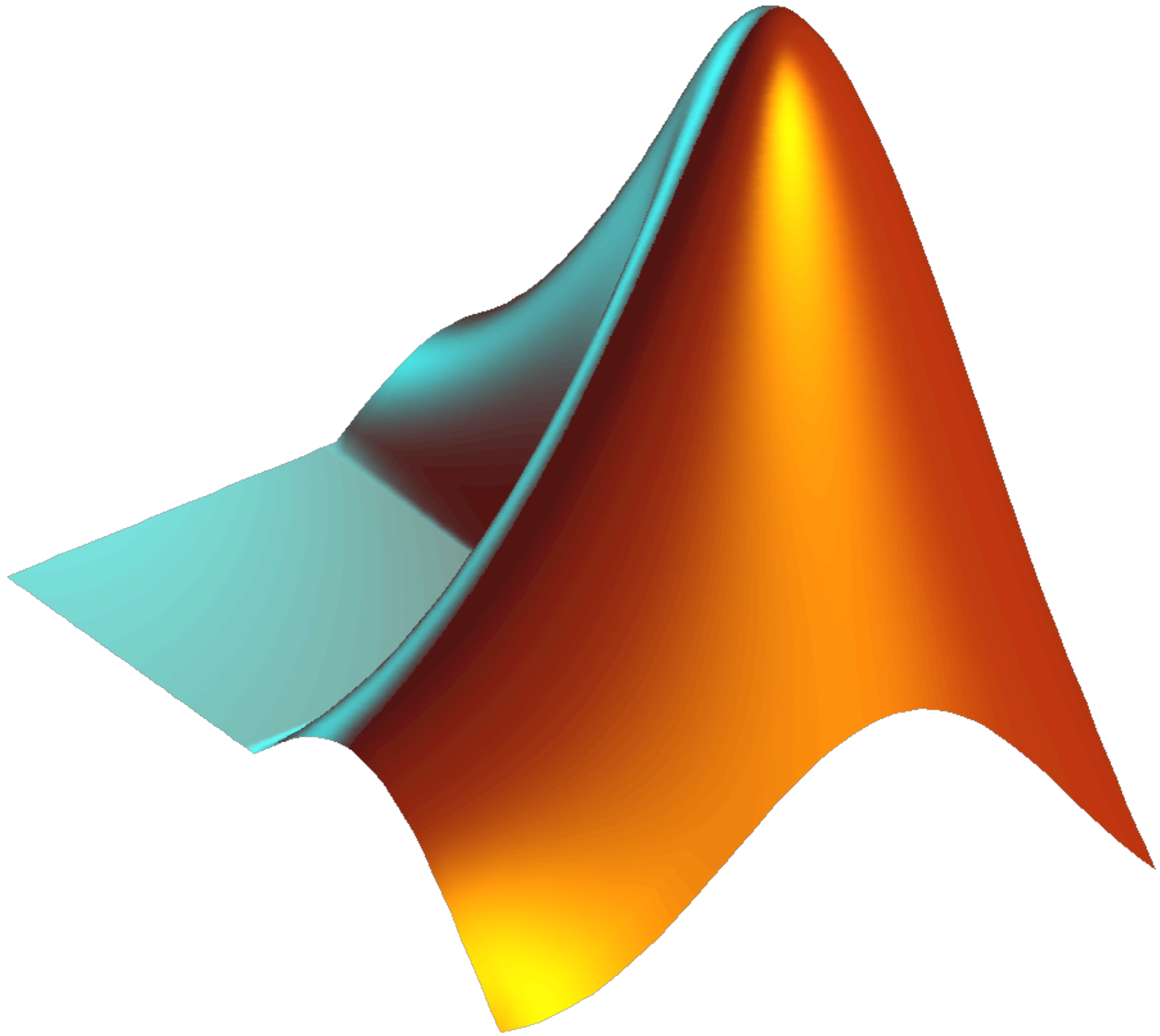
Transpose operation

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'
```



Demo: Multidimensional indexing



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

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`		`3 > 2` evaluates to `0`	
`2 < 2` evaluates to `0`		`1 > 2` evaluates to `0`	
`2 <= 2` evaluates to `1`		`2 >= 2` evaluates to `1`	
`2 == 2` evaluates to `1`		`3 ~= 2` evaluates to `1`	
`3 == 2` evaluates to `0`		`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

23	0
15	15
1	1
2.4	0
-1.1	-1.1

==

23	5
15	4
0	1
2.4	2
0	-1.1

evaluates to

1	0
1	0
0	1
1	0
0	1

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

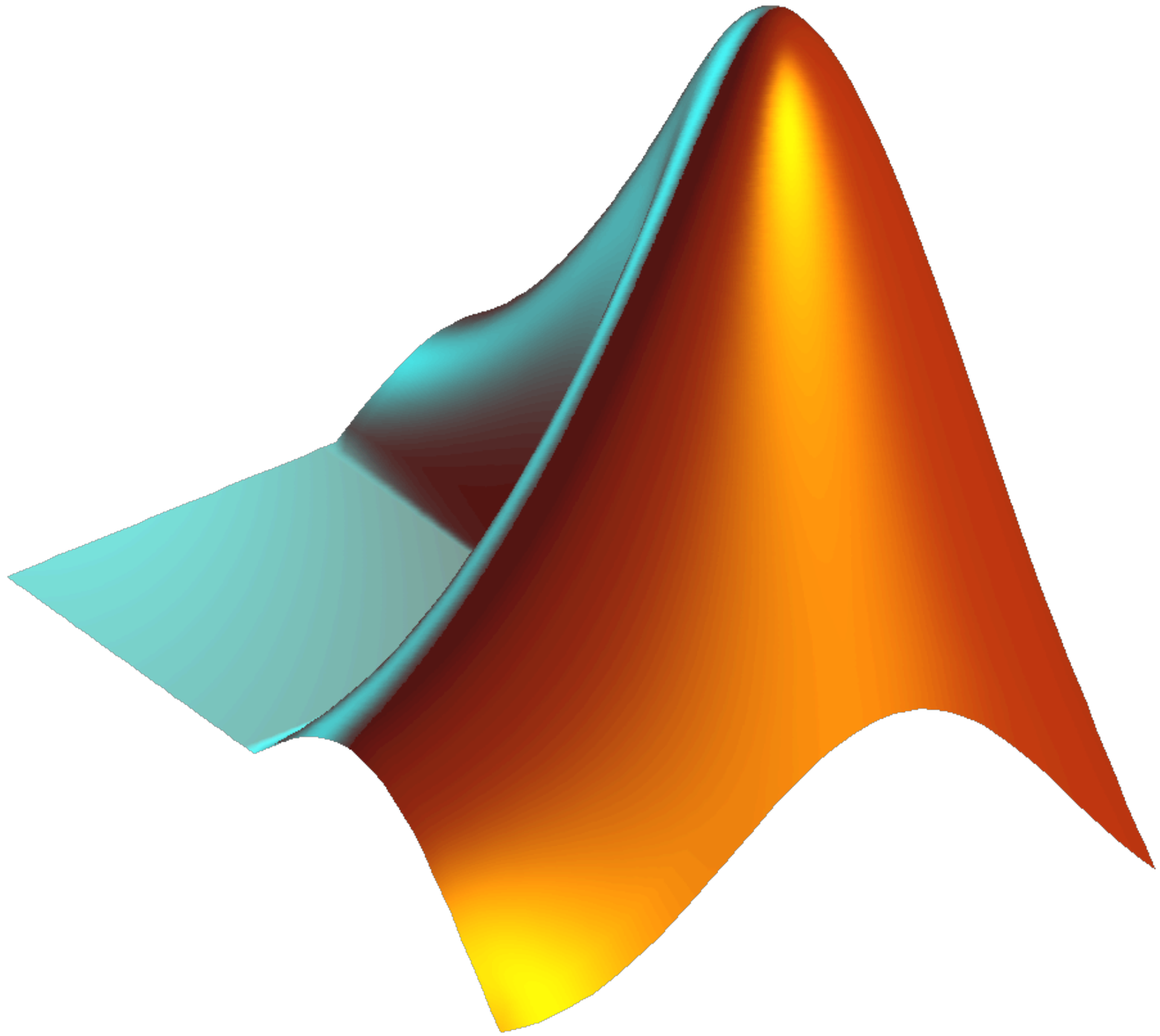
```
vals = [1 2 -1 1 -3];
```

`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

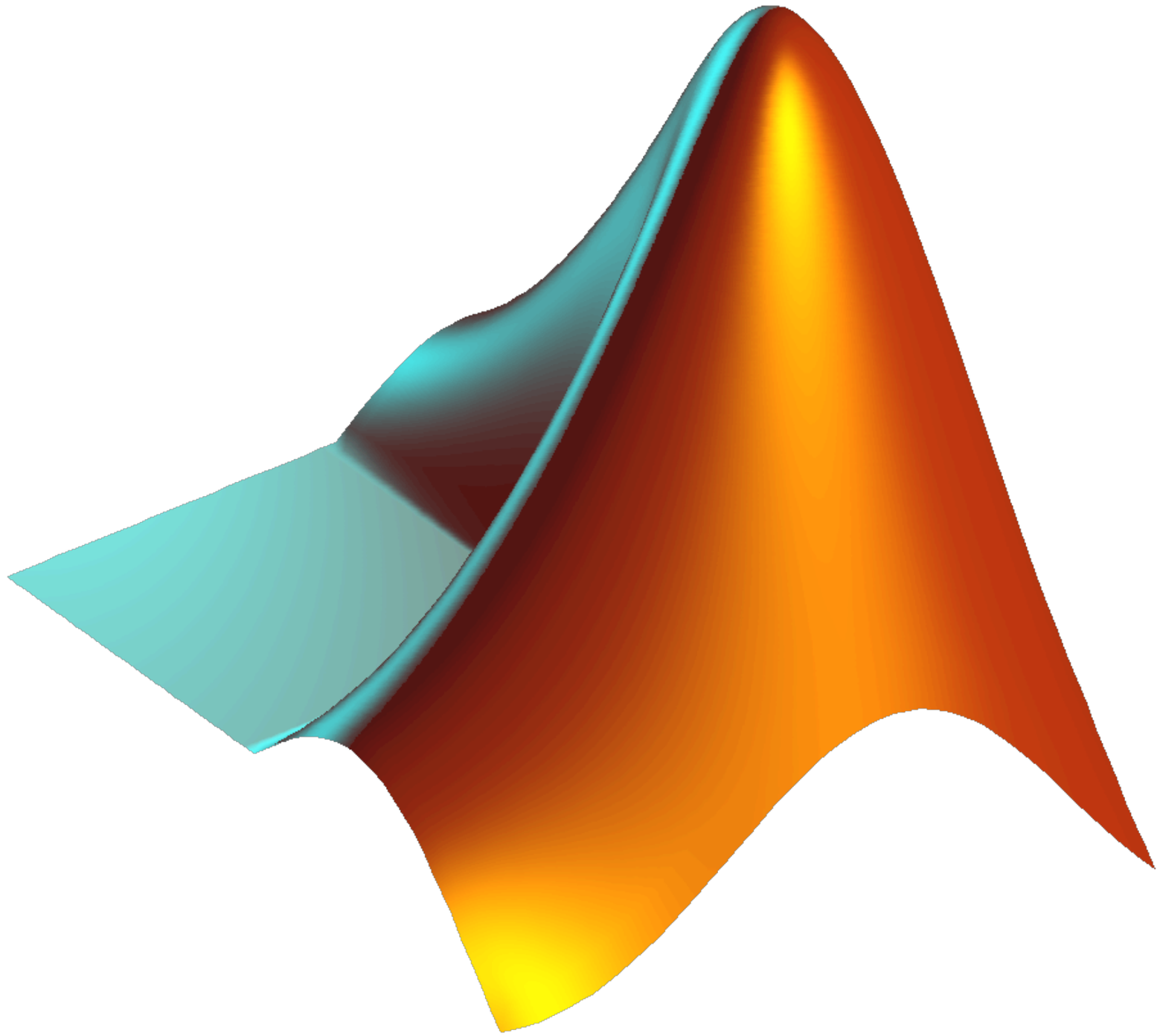
Can be used on any array, but with logical arrays, counts the number of elements that satisfy the conditions.

`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);
```

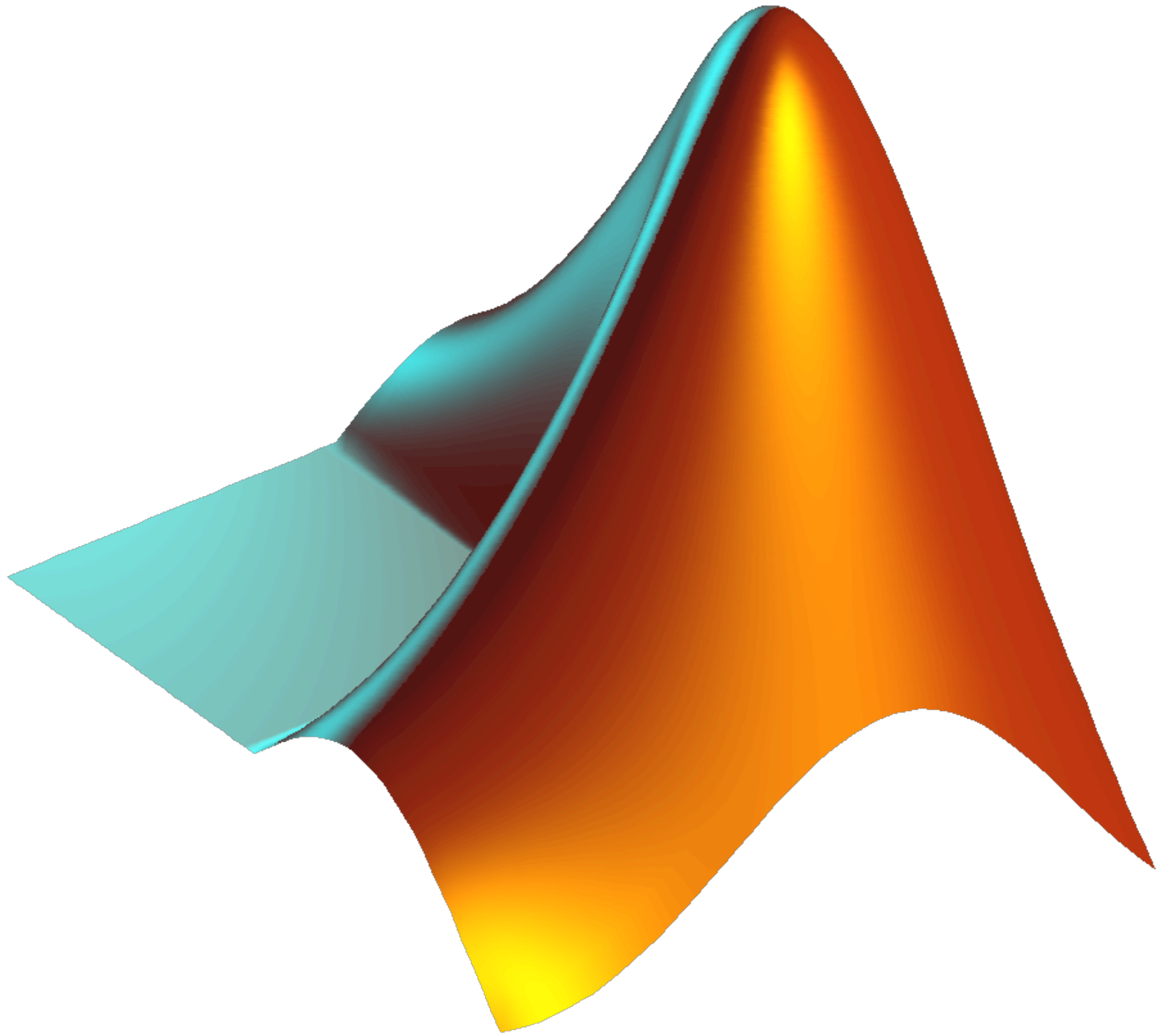
`i` evaluates to `[1; 2; 4]`

Rows on which the values are found

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

Columns in which the values are found

Demo: `find()` function



Other data structures

In addition to numeric arrays, there are:

- Character arrays or strings
- Cell arrays
- Structures

Fortunately, indexing works in pretty much the same way for all of them.

We'll get to these next week...

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

Axoclamp binary file

Format used by pCLAMP

Typically with .abf extension

`abfload()` utility exists to load into MATLAB

- Thank Forrest Collman for upgrading it to ABF v2.0

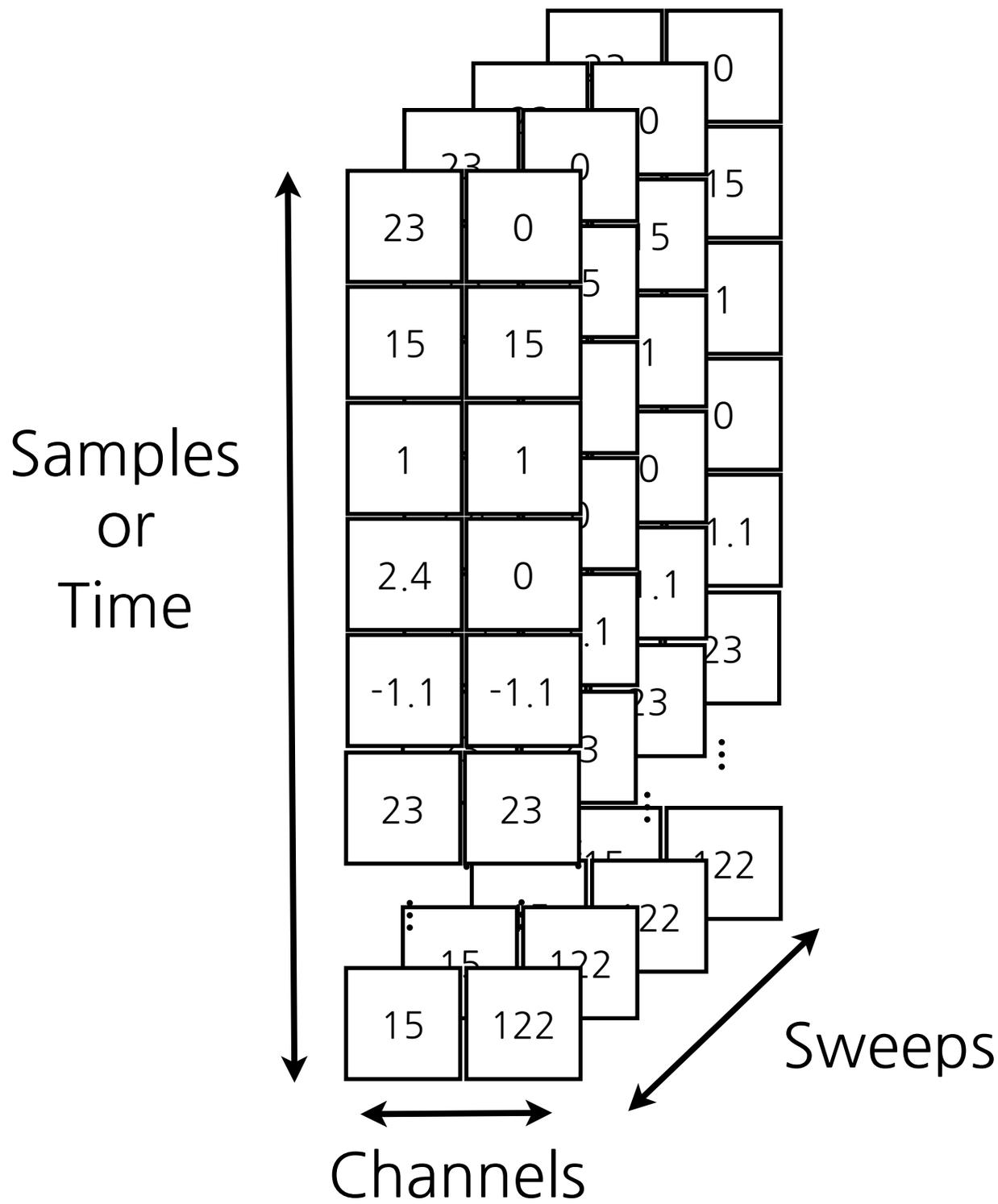
`abfload()` function

```
[data samplingIntervalUs info] = abfload(fname)
```

- `fname`: the abf file name, e.g. `'file.abf'`
- `data`: 3-dimensional array,
  size is nSamples x nChannels x nSweeps
- `samplingIntervalUs`: how frequently the data points are sampled. `1e6 / samplingIntervalUs` is the sampling frequency in Hz
- `info`: a struct (next week!) containing useful information about the Axoclamp configuration

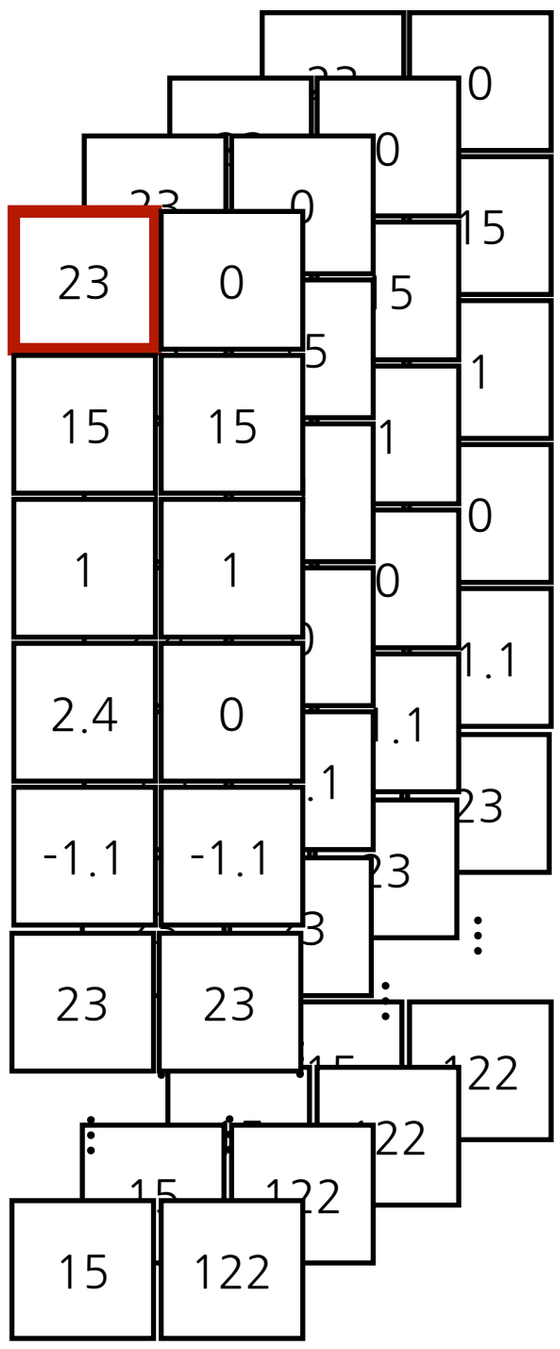
Axoclamp data

Axoclamp data as returned by abfload:
Samples (dim 1) by channels (dim 2) by sweeps (dim 3)



Axoclamp data

Axoclamp data as returned by abfload:
Samples (dim 1) by channels (dim 2) by sweeps (dim 3)



`data(1,1,1)`

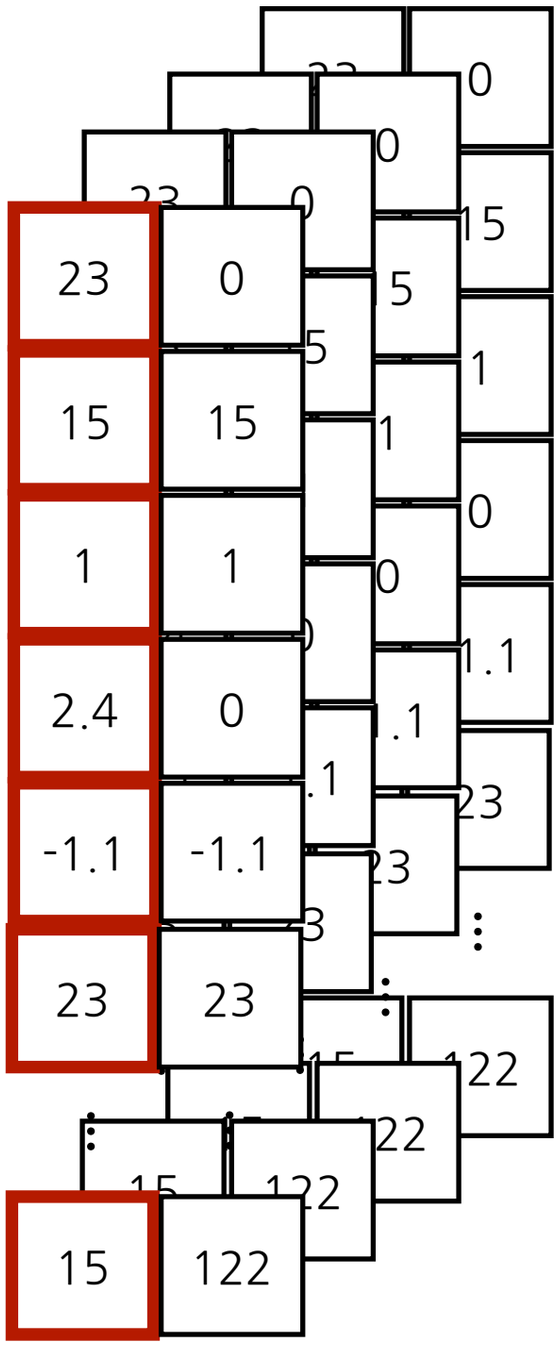
Sample 1

Channel 1

Sweep 1

Axoclamp data

Axoclamp data as returned by abfload:
Samples (dim 1) by channels (dim 2) by sweeps (dim 3)



`data(:,1,1)`

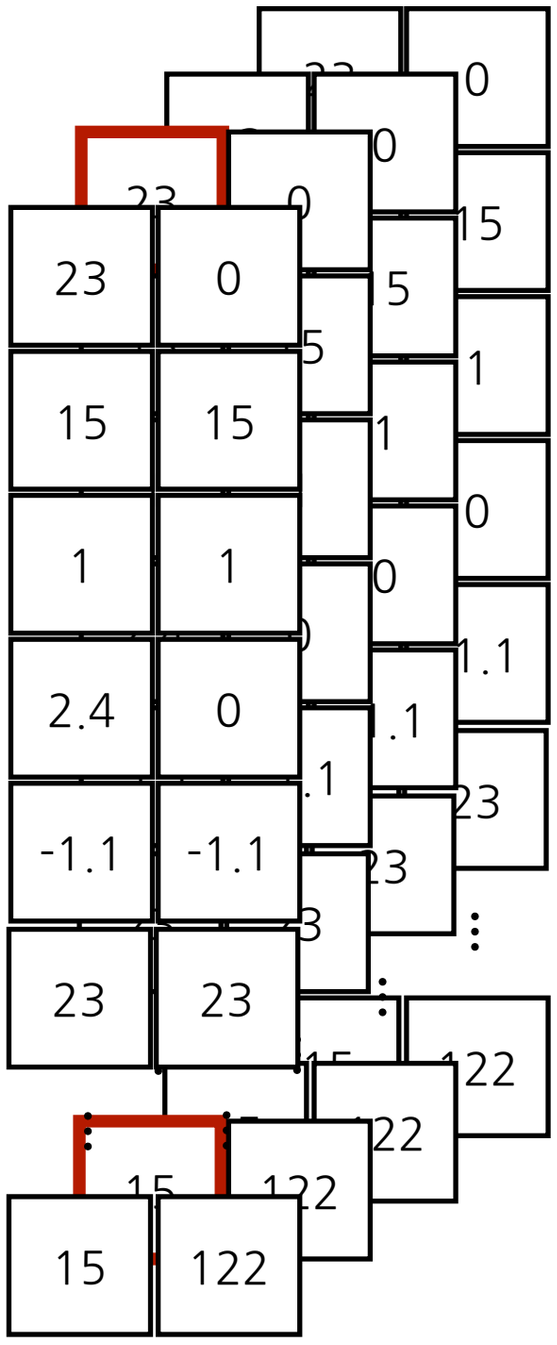
All samples

Channel 1

Sweep 1

Axoclamp data

Axoclamp data as returned by abfload:
Samples (dim 1) by channels (dim 2) by sweeps (dim 3)



`data(:,1,2)`

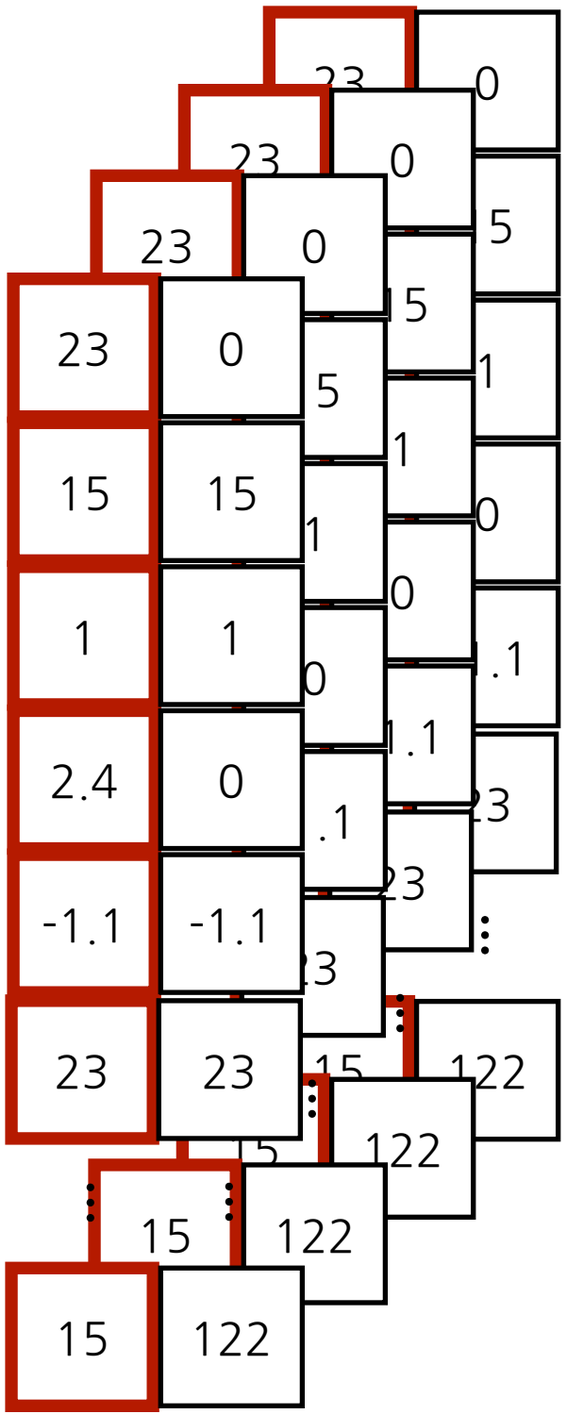
All samples

Channel 1

Sweep 2

Axoclamp data

Let's grab an entire channel's worth of data




```
chVm = data(:,1,:)
```

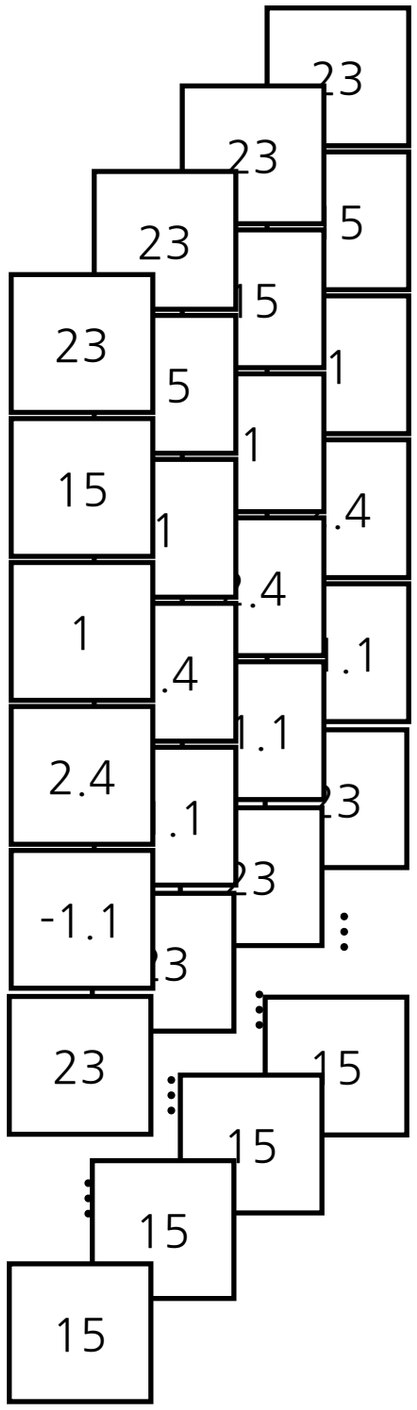
All samples

Channel 1

All sweeps

Axoclamp data

Let's grab an entire channel's worth of data




```
chVm = data(:,1,:);
```

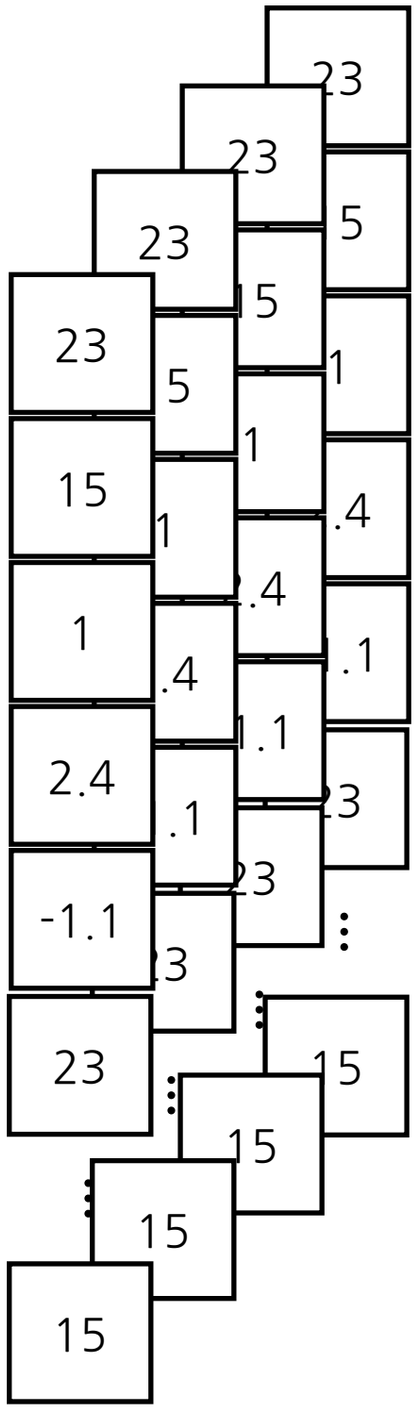
`size(chVm)` evaluates to

```
[nSamples 1 nSweeps]
```

This is an unwieldy shape for this data...dimension 2 is now unnecessary

Axoclamp data

Let's use `squeeze()` to make this a 2d array




```
chVm = squeeze(data(:,1,:));
```

What happens if we run `squeeze()`?

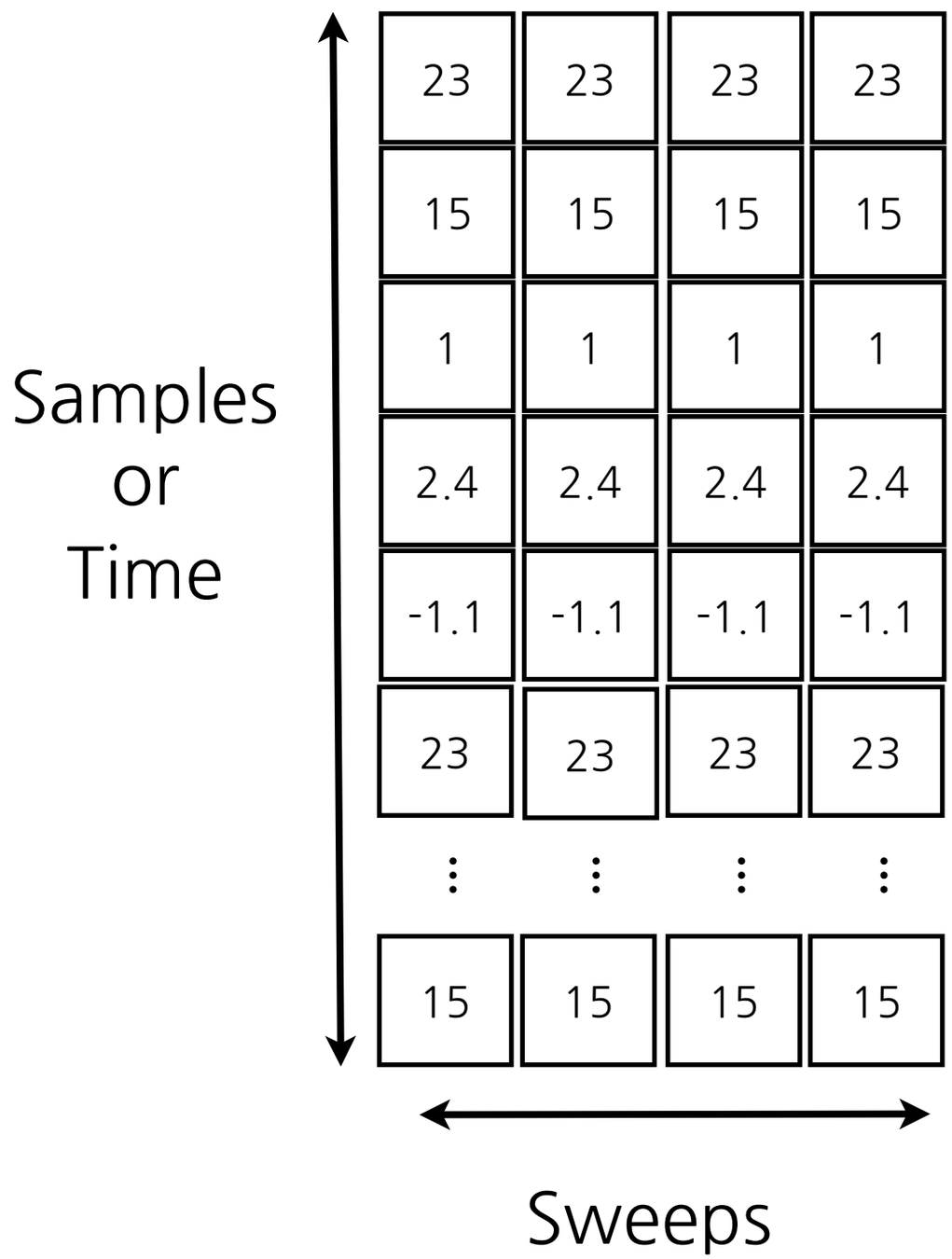
Looks at dimension 1 (samples):
- Not length 1, move on

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

Looks at what was dimension 3 (sweeps)
- Not length 1, move on

Axoclamp data

Let's use `squeeze()` to make this a 2d array



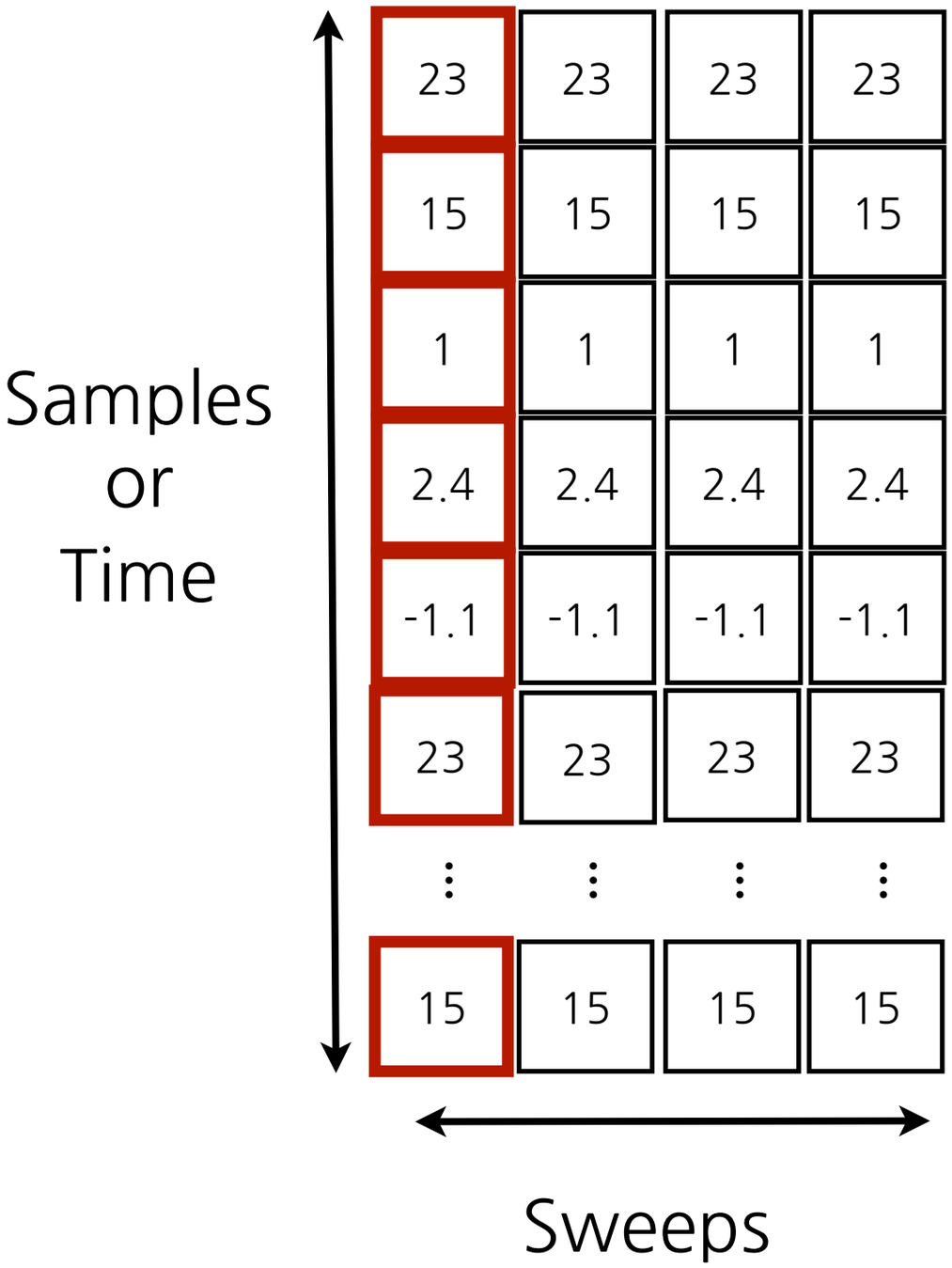

```
chVm = squeeze(data(:,1,:));
```

`size(chVm)` evaluates to

`[nSamples nSweeps]`

Axoclamp data

Now let's extract the trace for a given sweep




```
sweep1 = chVm(:,1)
```

`size(sweep1)` evaluates to

```
[nSamples 1]
```

Time vector

How do we plot this trace? We need a time vector that tells us what time each sample was taken at.

`sampleNumber = 1:nSamples` evaluates to

```
[1 2 3 4 5 ... nSamples]
```

```
tvecUs = (1:nSamples) * samplingIntervalUs;

tvecMs = (1:nSamples) * samplingIntervalUs / 1000;
```

`tvecMs(15)` is the time when sample 15 was taken, i.e. the time associated with `sweep1(15)`

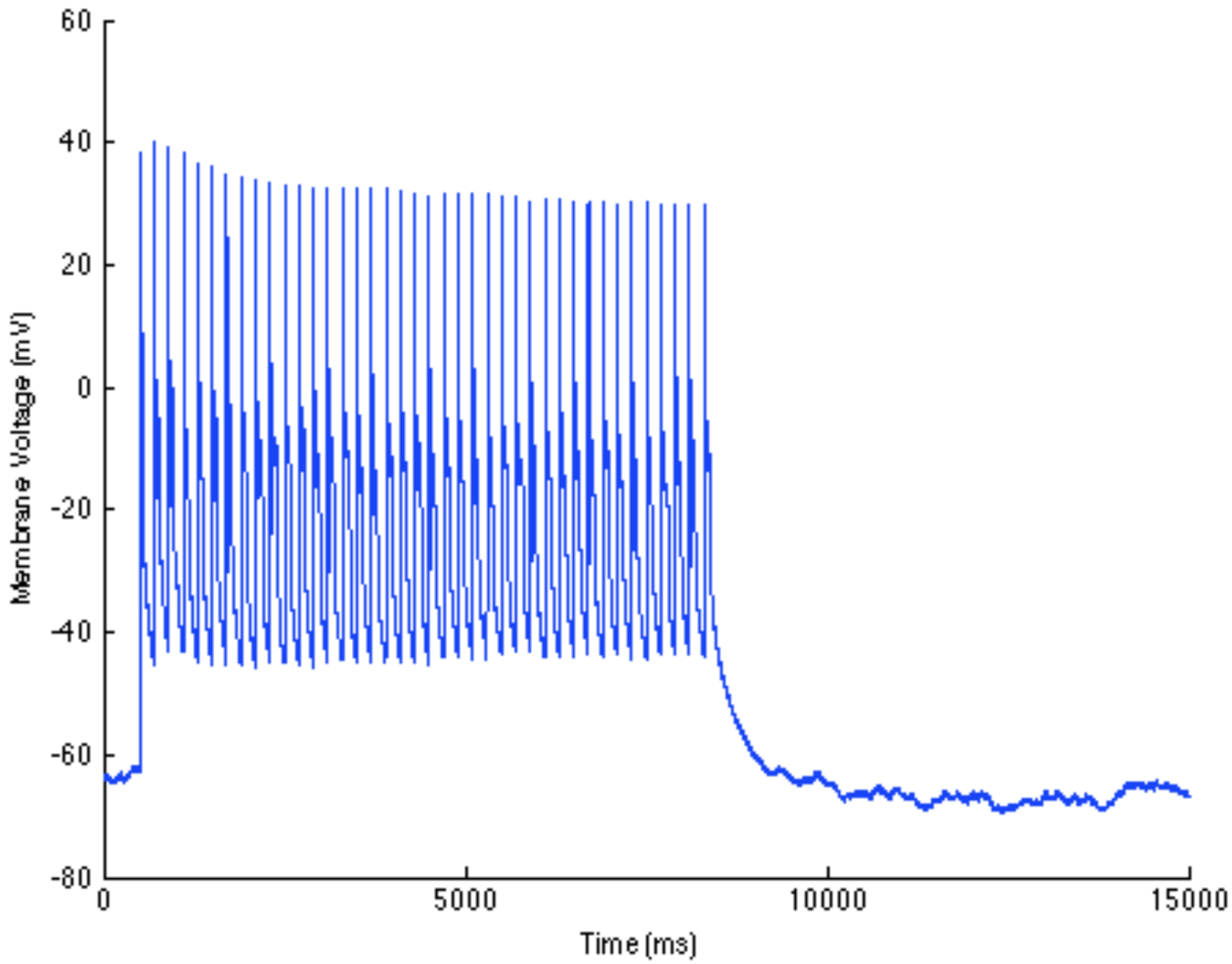
Plotting a sweep

Now that we have a time vector and the sweep, we can simply plot one against the other.

```
plot(tvecMs, sweep1);
xlabel('Time (ms)');
ylabel('Membrane Voltage (mV)');
```

Plotting a sweep

Now that we have a time vector and the sweep, we can simply plot one against the other.



Thresholding

How do we locate the spikes?

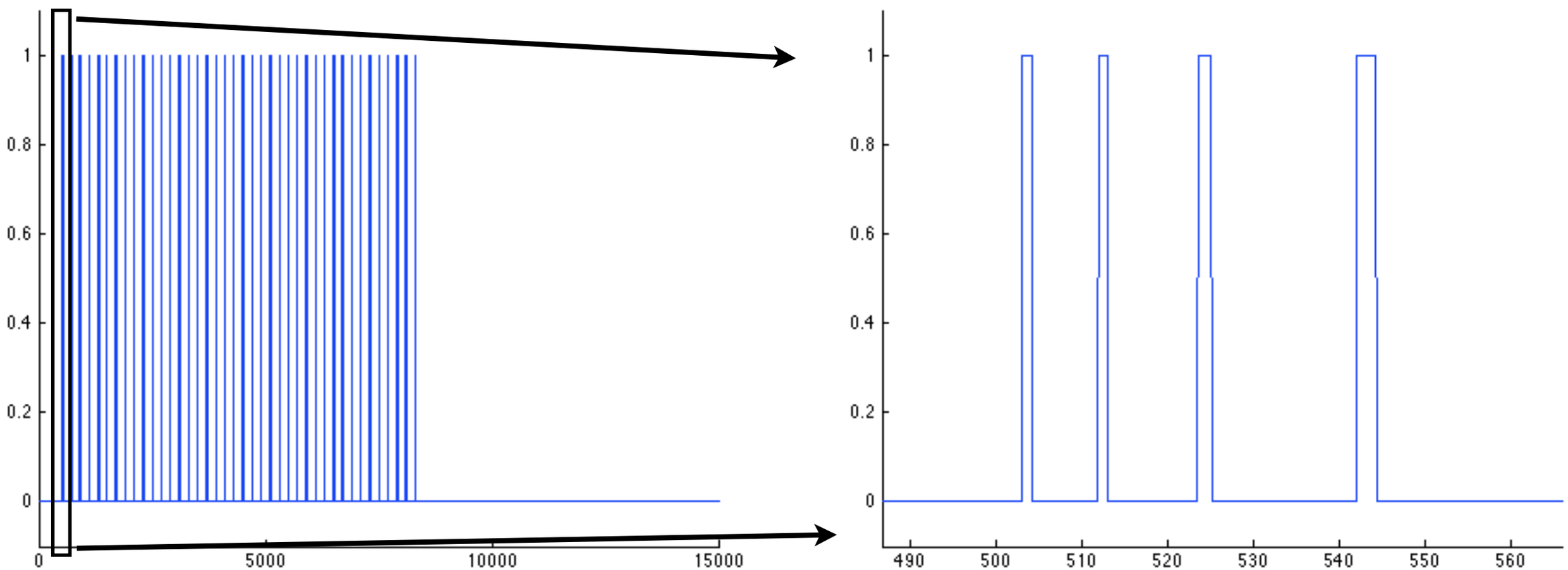
Set a threshold, create a logical array that indicates when the signal is above threshold.

```
vmThresh = 0;

vmAboveThresh = sweep1 >= vmThresh;

plot(tvecMs, vmAboveThresh);
```

Thresholding

How do we locate the spikes?

Set a threshold, create a logical array that indicates when the signal is above threshold.



Threshold crossings

Getting there, but we want to know when the signal first crosses threshold.

When this happens, the thresholded signal **`vmAboveThresh`** goes from being 0 (below thresh) to 1 (above thresh). When goes below threshold again, `vmAboveThresh` goes from being 1 to being 0.

So we want to locate where `vmAboveThresh` changes from 0 to 1.

`diff()` function

`diff()` returns a vector of the differences between successive elements of a vector, an approximation of the derivative. Shortens the vector by one.

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

23	15	1	2.4	-1.1

```
bar = diff(foo)
```

-8	-14	1.4	-3.5

diff() function

`diff()` returns a vector of the differences between successive elements of a vector, an approximation of the derivative. Shortens the vector by one

```
vmAboveThresh = [0 0 1 1 1 0 0]

vmCrossThresh = diff(vmAboveThresh)
```

evaluates to `[0 1 0 0 -1 0]`

Threshold crossings

Now find the locations at which these threshold crossings from below occur. Use the time vector to find when they occur.

```
vmCrossThreshFromBelow = vmCrossThresh == 1
```

evaluates to `[0 1 0 0 0 0]`

```
spikeInds = find(vmCrossThresh == 1)
```

evaluates to `[2]`

```
spikeTimesMs = tvecMs(spikeInds)
```

Annotating the signal

Let's plot the detected spike times on top of the membrane signal to allow us to visually check that everything's working correctly.

```
plot(tvecMs, sweep1);
hold on;
plot(spikeTimesMs, ...
 vmThresh*ones(size(spikeTimesMs)), ...
 'rx', 'MarkerSize', 8);
```

Annotating the signal

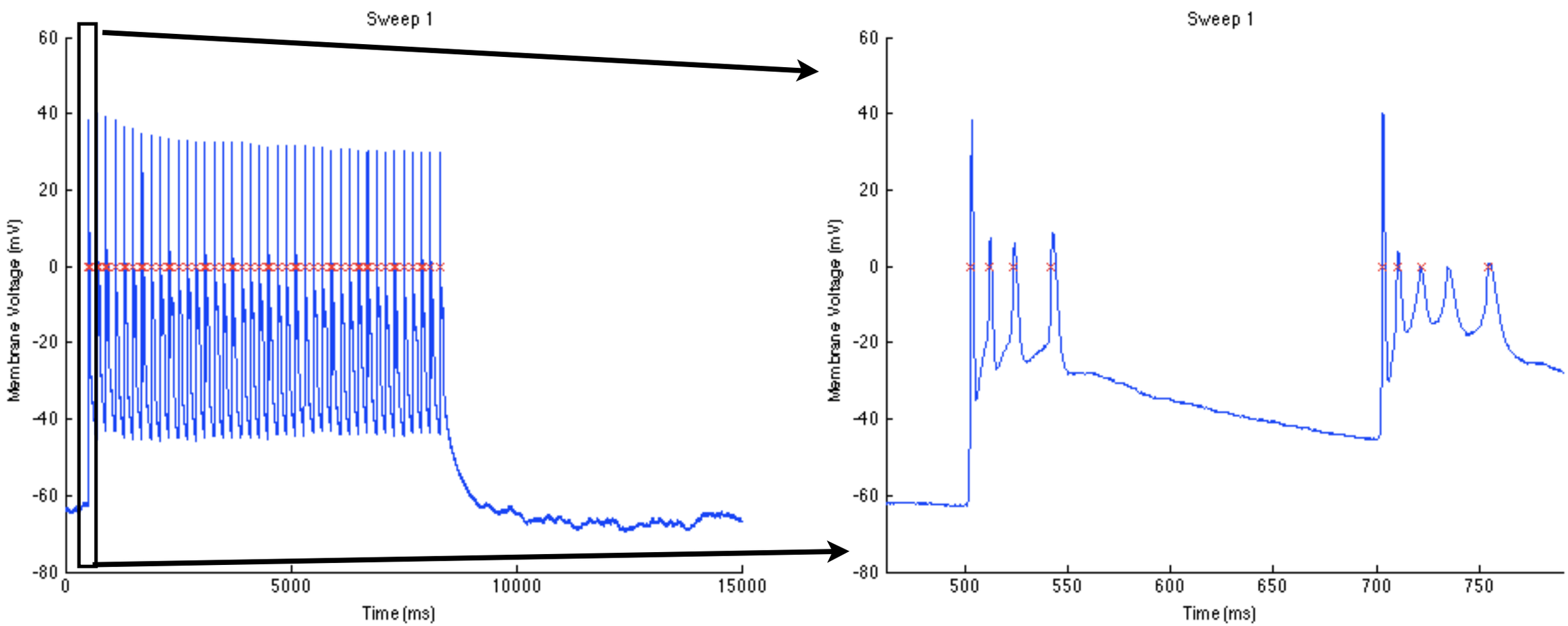
Let's plot the detected spike times on top of the membrane signal to allow us to visually check that everything's working correctly.

```
plot(tvecMs, sweep1);
hold on;
plot(spikeTimesMs, ...
 vmThresh*ones(size(spikeTimesMs)), ...
 'rx', 'MarkerSize', 8);
```

Create a vector the same size as spikeTimesMs filled with all values equal to vmThresh. These are the y coordinates to plot our spike detection markers at.

Annotating the signal

Let's plot the detected spike times on top of the membrane signal to allow us to visually check that everything's working correctly.



Very quickly: `for` loops

In this assignment, you'll want to count the number of spikes evoked in each sweep. This means you'll want to repeat everything nSweeps times. How do we do this?

We'll cover flow control next week, but for now, you'll use code that looks like this.

```
for iSweep = 1:nSweeps
    % code that runs nSweeps time
    % each time with a different value stored in iSweep
end
```

Very quickly: `for` loops

```
% preallocate a vector that stores the result for each sweep
nSpikesEvoked = zeros(nSweeps, 1);
for iSweep = 1:nSweeps
    % code that runs nSweeps time
    % each time with a different value stored in iSweep

    % grab this sweep's membrane voltage
    vmSweep = chVm(:, iSweep);

    % do some calculations, then store the results
    nSpikesEvoked(iSweep) = (something you've calculated)
end
```

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.

Function list

`size`

`ones`

`zeros`

Syntax for multidimensional indexing

Transpose operation (see `help ctranspose`)

Colon notation (see `help colon`)

`logical`

`class`

Conditional operators: `< <= >= > = ~=`

`nnz`

`find`

`squeeze`

`flipud`

`abfload`

`diff`

`for`