Logic, loops and control flow

NENS 230

7 Sep 2014

Benjamin Naecker

Outline for today

Outline for today

- Review of relational operators

Outline for today

- Review of relational operators
- Logic and branching

Outline for today

- Review of relational operators
- Logic and branching
- Loops

Outline for today

- Review of relational operators
- Logic and branching
- Loops
- Advanced control flow

Outline for today

- Review of relational operators
- Logic and branching
- Loops
- Advanced control flow

Relational operators

Relational operators

Test if relationship is true or false

Relational operators

Test if relationship is true or false

- Less than `2 < 3` `(true)`

Relational operators

Test if relationship is true or false

• Less than	`2 < 3`	`(true)`
• Greater than	`2 > 3`	`(false)`

Relational operators

Test if relationship is true or false

• Less than	`2 < 3`	`(true)`
• Greater than	`2 > 3`	`(false)`
• Less than or equal to	`1 <= 1`	`(true)`

Relational operators

Test if relationship is true or false

• Less than	`2 < 3`	`(true)`
• Greater than	`2 > 3`	`(false)`
• Less than or equal to	`1 <= 1`	`(true)`
• Greater than or equal to	`4 >= 5`	`(false)`

Relational operators

Test if relationship is true or false

Less than	`2 < 3`	`(true)`
Greater than	`2 > 3`	`(false)`
Less than or equal to	`1 <= 1`	`(true)`
Greater than or equal to	`4 >= 5`	`(false)`
Exactly equal	`8 == 2^3`	`(true)`

Relational operators

Test if relationship is true or false

• Less than	`2 < 3`	`(true)`
• Greater than	`2 > 3`	`(false)`
• Less than or equal to	`1 <= 1`	`(true)`
• Greater than or equal to	`4 >= 5`	`(false)`
• Exactly equal	`8 == 2^3`	`(true)`
• Not equal	`pi ~= 3.14`	`(true)`

Relational operators

Test if relationship is true or false

• Less than	`2 < 3`	`(true)`
• Greater than	`2 > 3`	`(false)`
• Less than or equal to	`1 <= 1`	`(true)`
• Greater than or equal to	`4 >= 5`	`(false)`
• Exactly equal	`8 == 2^3`	`(true)`
• Not equal	`pi ~= 3.14`	`(true)`

Output is datatype called `logical`

Remember...

Remember...

“=” means assignment

Remember...

“=” means assignment

```
>> x = 5

ans =
     5
```

Remember...

“=” means assignment

“==” tests equality

```
>> x = 5

ans =
     5
```

Remember...

“=” means assignment

```
>> x = 5

ans =
     5
```

“==” tests equality

```
>> x = 5;
>> x == 5

ans =
     1
```

Remember...

“=” means assignment

```
>> x = 5

ans =
     5
```

“==” tests equality

```
>> x = 5;
>> x == 5

ans =
     1
```

One of the most common programming mistakes ever

Remember...

“=” means assignment

```
>> x = 5

ans =
     5
```

“==” tests equality

```
>> x = 5;
>> x == 5

ans =
     1
```

One of the most common programming mistakes ever

You will make it. Many times.

Relational operators are *vectorized*

Operate on all members of an array simultaneously

Relational operators are *vectorized*

Operate on all members of an array simultaneously

```
>> x = 1:10
```

Relational operators are *vectorized*

Operate on all members of an array simultaneously

```
>> x = 1:10

ans =
     1  2  3  4  5  6  7  8  9  10
```

Relational operators are *vectorized*

Operate on all members of an array simultaneously

```
>> x = 1:10

ans =
     1  2  3  4  5  6  7  8  9  10

>> x > 5
```

Relational operators are *vectorized*

Operate on all members of an array simultaneously

```
>> x = 1:10

ans =
     1  2  3  4  5  6  7  8  9  10

>> x > 5

ans =
     0  0  0  0  0  1  1  1  1  1
```

Logical indexing

Logical indexing

- Allows selection of any elements meeting criteria

Logical indexing

- Allows selection of any elements meeting criteria
- Criteria usually defined by relational operators

Logical indexing

- Allows selection of any elements meeting criteria
- Criteria usually defined by relational operators

```
>> x = 1:10;
```

Logical indexing

- Allows selection of any elements meeting criteria
- Criteria usually defined by relational operators

```
>> x = 1:10;
>> idx = x > 5;
```

Logical indexing

- Allows selection of any elements meeting criteria
- Criteria usually defined by relational operators

```
>> x = 1:10;
>> idx = x > 5;
>> x(idx)
```

Logical indexing

- Allows selection of any elements meeting criteria
- Criteria usually defined by relational operators

```
>> x = 1:10;
>> idx = x > 5;
>> x(idx)

ans =
     6  7  8  9  10
```

Logical indexing

- Allows selection of any elements meeting criteria
- Criteria usually defined by relational operators

```
>> x = 1:10;
>> idx = x > 5;
>> x(idx)

ans =
     6     7     8     9    10
```

Usually combine in one expression: `x(x > 5)`

Assignment by logical indexing

Assignment by logical indexing

```
>> x = -2:2;
```

Assignment by logical indexing

```
>> x = -2:2;
>> x(x < 0) = 0 % Truncate values at 0
```

Assignment by logical indexing

```
>> x = -2:2;
>> x(x < 0) = 0 % Truncate values at 0

ans =
     0  0  0  1  2
```

Assignment by logical indexing

```
>> x = -2:2;
>> x(x < 0) = 0 % Truncate values at 0

ans =
     0   0   0   1   2


>> x(x < 0) = [] % Remove negatives
```

Assignment by logical indexing

```
>> x = -2:2;
>> x(x < 0) = 0 % Truncate values at 0

ans =
     0   0   0   1   2


>> x(x < 0) = [] % Remove negatives

ans =
     0   1   2
```

Logical indexing and `find`

`find` returns indices of nonzero elements

Logical indexing and `find`

`find` returns indices of nonzero elements

```
>> x = 1:10;
```

Logical indexing and `find`

`find` returns indices of nonzero elements

```
>> x = 1:10;
>> x > 5
```

Logical indexing and `find`

`find` returns indices of nonzero elements

```
>> x = 1:10;
>> x > 5

ans =
     0   0   0   0   0   1   1   1   1   1
```

Logical indexing and `find`

`find` returns indices of nonzero elements

```
>> x = 1:10;
>> x > 5

ans =
     0   0   0   0   0   1   1   1   1   1

>> find(x > 5)
```

Logical indexing and `find`

`find` returns indices of nonzero elements

```
>> x = 1:10;
>> x > 5

ans =
     0  0  0  0  0  1  1  1  1  1

>> find(x > 5)

ans =
     6  7  8  9  10
```

A comment on the `find` function

A comment on the `find` function

```
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
```

A comment on the `find` function

```
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
>> ltime = avg_logical_time(data);
```

A comment on the `find` function

```
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
>> ltime = avg_logical_time(data);
>> ftime = avg_find_time(data);
```

A comment on the `find` function

```
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
>> ltime = avg_logical_time(data);
>> ftime = avg_find_time(data);
>> ftime / ltime
```

A comment on the `find` function

```
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
>> ltime = avg_logical_time(data);
>> ftime = avg_find_time(data);
>> ftime / ltime

ans =
     7.5426
```

A comment on the `find` function

```
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
>> ltime = avg_logical_time(data);
>> ftime = avg_find_time(data);
>> ftime / ltime

ans =
     7.5426
```

Logical indexing is faster by an order of magnitude

When to use `find`

When to use `find`

- Need the actual indices of a condition

When to use `find`

- Need the actual indices of a condition
- Need to know only first place condition is satisfied

When to use `find`

- Need the actual indices of a condition
- Need to know only first place condition is satisfied
- Better for arrays with many zeros (sparse)

When to use `find`

- Need the actual indices of a condition
- Need to know only first place condition is satisfied
- Better for arrays with many zeros (sparse)

Everything else should use logical indexing

Other useful logical functions

Other useful logical functions

- `is____`

Other useful logical functions

- `is____`
- `isempty`

Other useful logical functions

- `is____`
- `isempty`
- `isnan, isinf`

Other useful logical functions

- `is____`
- `isempty`
- `isnan, isinf`
- `isfinite  % true if not +/-Inf, NaN`

Other useful logical functions

- is____
- isempty
- isnan, isinf
- isfinite % true if not +/-Inf, NaN
- isnumeric, ischar, iscell, isstruct

Our code thus far

Our code thus far

```
do_thing1(data);
```

Our code thus far

```
do_thing1(data);
do_thing2(data);
```

Our code thus far

```
do_thing1(data);
do_thing2(data);
do_thing3(data);
```

Our code thus far

```
do_thing1(data);
do_thing2(data);
do_thing3(data);
.
.
.
do_thing73(data);
```

Our code thus far

```
do_thing1(data);
do_thing2(data);
do_thing3(data);
.
.
.
do_thing73(data);
```

All code executed unconditionally

Our code thus far

```
do_thing1(data);
do_thing2(data);
do_thing3(data);
.
.
.
do_thing73(data);
```

All code executed unconditionally

But what if we only want to `do_thing2()` *if* `data` *meets some criterion?*

Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow

Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow

Branching

Branching

- Execute blocks of code only if certain conditions are met

Branching

- Execute blocks of code only if certain conditions are met
- Each possible path is a “branch”

Branching

- Execute blocks of code only if certain conditions are met
- Each possible path is a “branch”

```
if condition1 % keyword to start block
```

Branching

- Execute blocks of code only if certain conditions are met
- Each possible path is a “branch”

```
if condition1 % keyword to start block
   do_something();
```

Branching

- Execute blocks of code only if certain conditions are met
- Each possible path is a “branch”

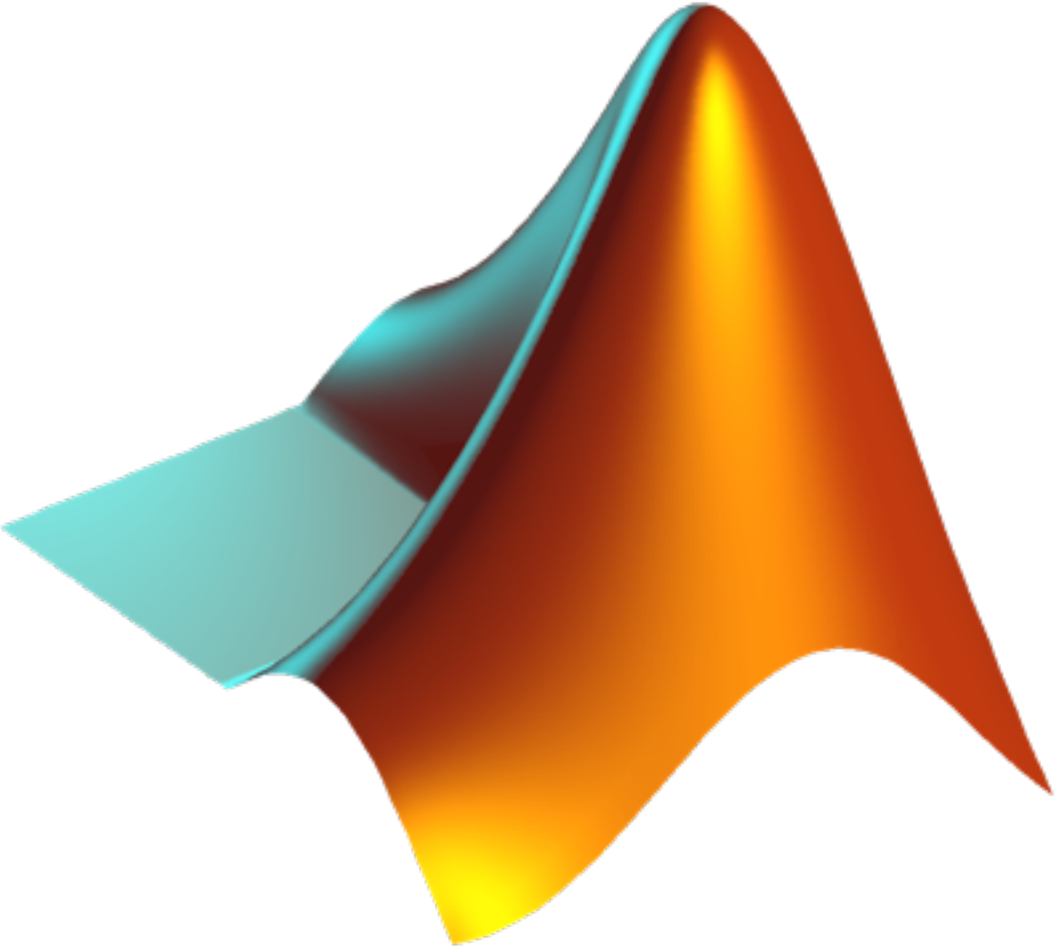
```
if condition1 % keyword to start block
  do_something();
end           % keyword to end block
```

Branching: `if`/`else`

Branching: if/else

```
if p_value <= significance_threshold
   % jump here if condition is true
   keep_data();
else
   % jump here if condition is false
   % keep_data is NOT executed!
   ignore_data();
end
```

Branching: `if/else`



Branching: multiple conditions

Branching: multiple conditions

```
if condition1
  do_thing1();
end
```

Branching: multiple conditions

```
if condition1
  do_thing1();
end
if condition2
  do_thing2();
end
```

Branching: multiple conditions

```
if condition1
  do_thing1();
end
if condition2
  do_thing2();
end
if condition3
  do_thing3();
end
```

Branching: `if`/`elseif`

Branching: `if`/`elseif`

Test multiple conditions in a single block

Branching: `if/elseif`

Test multiple conditions in a single block

```
if condition1
```

Branching: `if/elseif`

Test multiple conditions in a single block

```
if condition1
  do_thing1();
```

Branching: if/elseif

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
```

Branching: if/elseif

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
  do_thing2();
```

Branching: `if/elseif`

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
  do_thing2();
elseif condition3
```

Branching: `if/elseif`

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
  do_thing2();
elseif condition3
  do_thing3();
```

Branching: if/elseif

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
  do_thing2();
elseif condition3
  do_thing3();
...
else
```

Branching: if/elseif

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
  do_thing2();
elseif condition3
  do_thing3();
...
else
  do_last_thing();
end
```

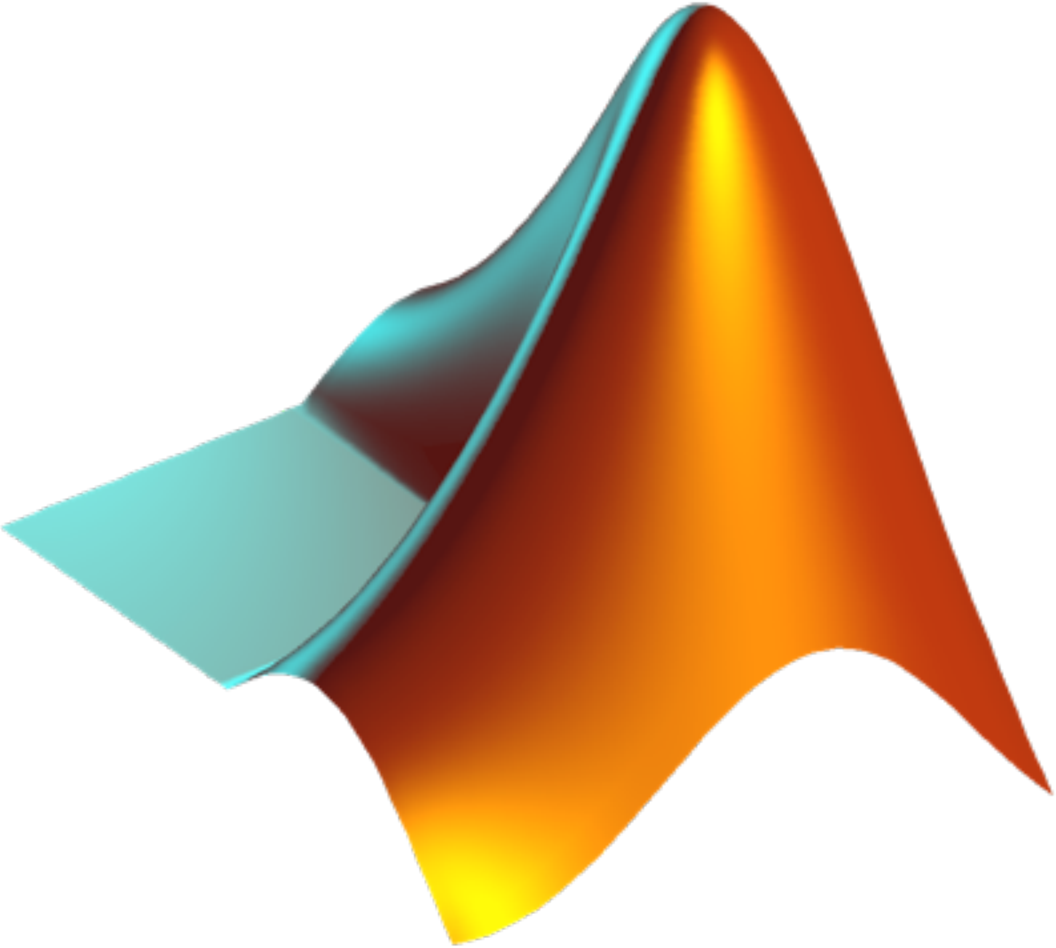
Branching: `if/elseif`

Test multiple conditions in a single block

```
if condition1
  do_thing1();
elseif condition2
  do_thing2();
elseif condition3
  do_thing3();
...
else
  do_last_thing();
end
```

One and *only* one of these functions will be executed.

Branching: `if`/`elseif`



Branching with many conditions

Branching with many conditions

- Used `if`/`elseif` blocks to test many conditions

Branching with many conditions

- Used `if/elseif` blocks to test many conditions
- Becomes cumbersome and hard to read with a large number of conditions

Branching: `switch`/`case`

Branching: `switch/case`

```
switch my_var
```

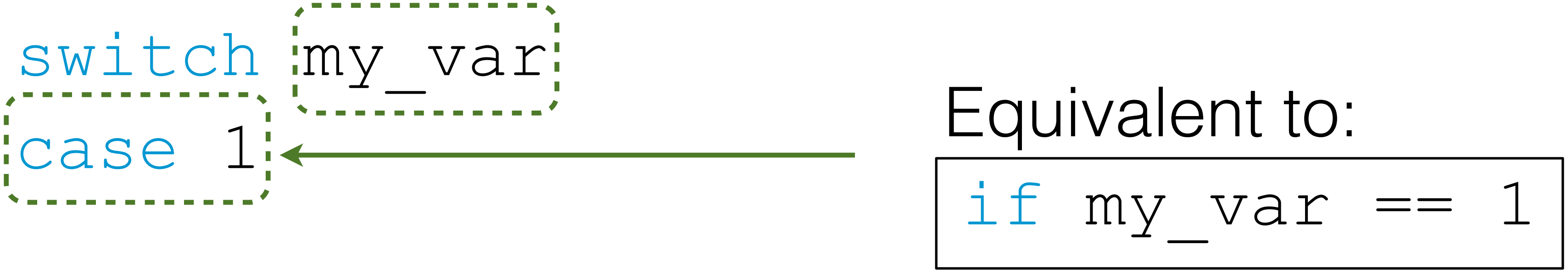
Branching: switch/case

```
switch my_var
```

Branching: switch/case

```
switch my_var
case 1
```

Branching: `switch/case`



```
switch my_var
case 1
```

Equivalent to:

```
if my_var == 1
```

Branching: switch/case

```
switch my_var
case 1
   do_thing1();
```

Equivalent to:

```
if my_var == 1
```

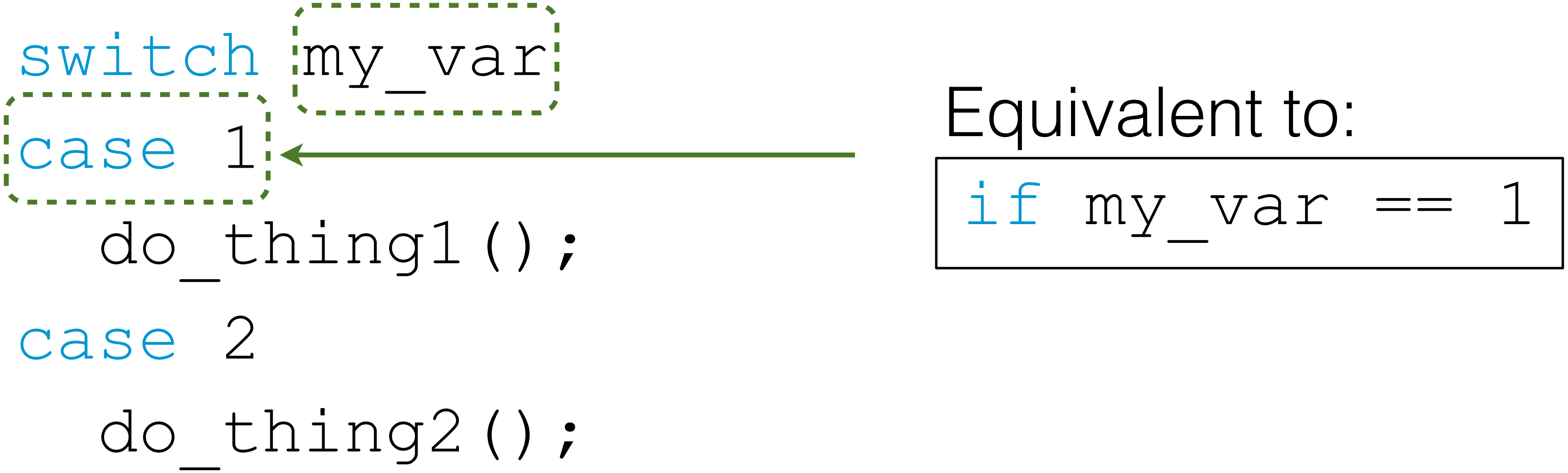
Branching: switch/case

```
switch my_var
case 1
    do_thing1();
case 2
```

Equivalent to:

```
if my_var == 1
```

Branching: switch/case



```
switch my_var
case 1
   do_thing1();
case 2
   do_thing2();
```

Equivalent to:

```
if my_var == 1
```

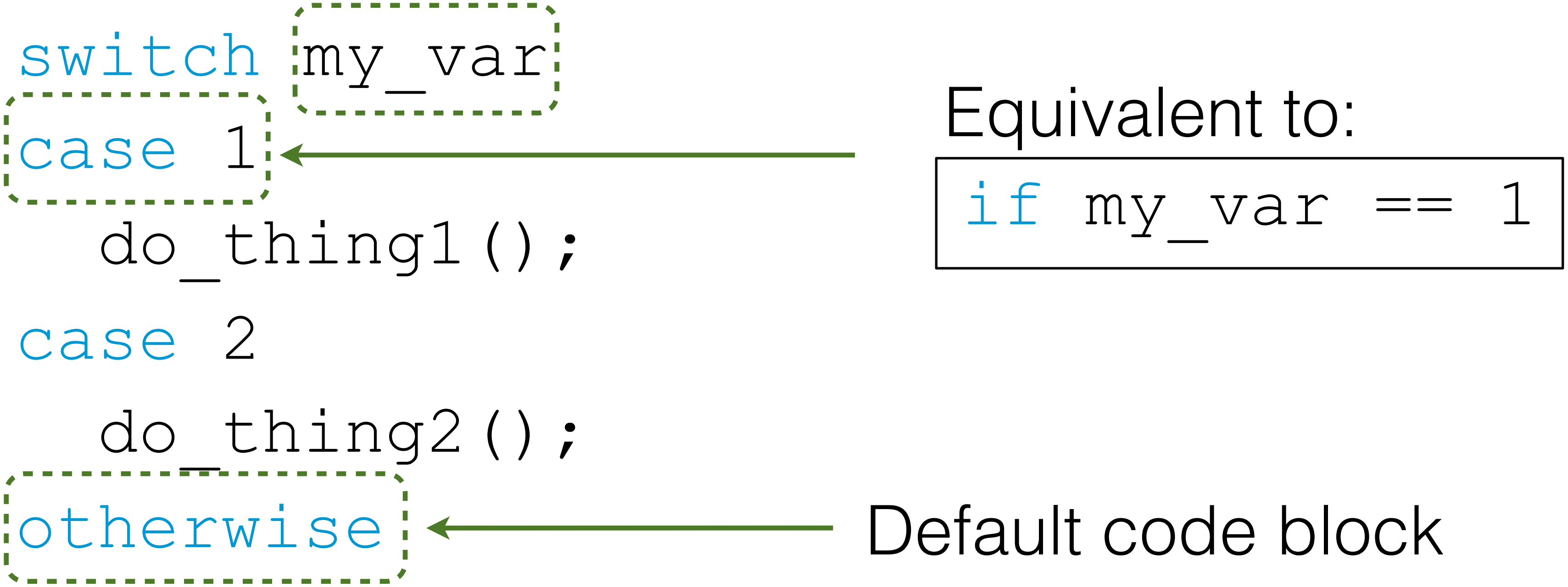
Branching: switch/case

```
switch my_var
case 1
   do_thing1();
case 2
   do_thing2();
otherwise
```

Equivalent to:

```
if my_var == 1
```

Branching: switch/case



```
switch my_var
case 1
   do_thing1();
case 2
   do_thing2();
otherwise
```

Equivalent to:

```
if my_var == 1
```

Default code block

Branching: switch/case

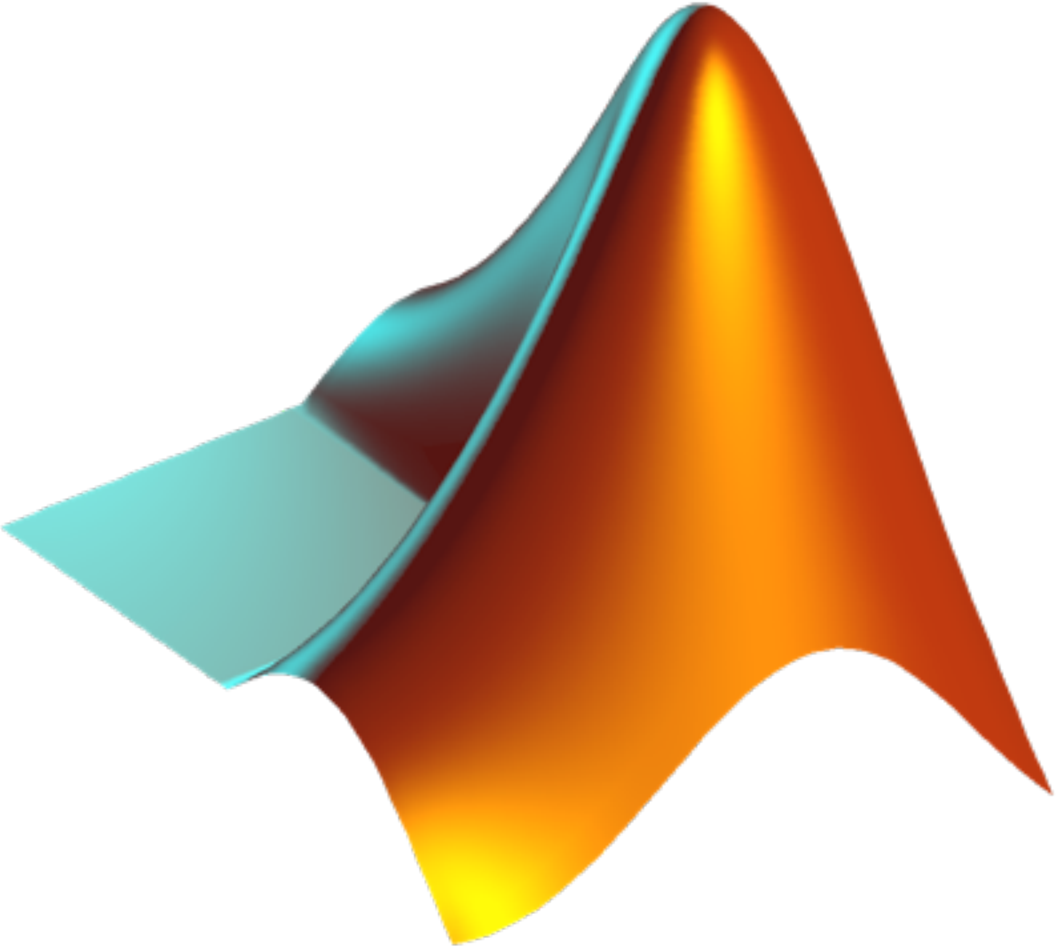
```
switch my_var
case 1
   do_thing1();
case 2
   do_thing2();
otherwise
   do_default_thing();
end
```

Equivalent to:

```
if my_var == 1
```

Default code block

Branching: `switch`/`case`



Combining logical values

Combining logical values

Use “`&`” or “`|`” to combine *arrays* of logicals

Combining logical values

Use “`&`” or “`|`” to combine *arrays* of logicals

```
>> val1 = [0, 1, 1, 0];
```

Combining logical values

Use “`&`” or “`|`” to combine *arrays* of logicals

```
>> val1 = [0, 1, 1, 0];
>> val2 = [1, 1, 0, 0];
```

Combining logical values

Use “`&`” or “`|`” to combine *arrays* of logicals

```
>> val1 = [0, 1, 1, 0];
>> val2 = [1, 1, 0, 0];
>> result = val1 & val2
```

Combining logical values

Use “`&`” or “`|`” to combine *arrays* of logicals

```
>> val1 = [0, 1, 1, 0];
>> val2 = [1, 1, 0, 0];
>> result = val1 & val2

result =
     0  1  0  0
```

Combining logical values

Use “`&`” or “`|`” to combine *arrays* of logicals

```
>> val1 = [0, 1, 1, 0];
>> val2 = [1, 1, 0, 0];
>> result = val1 & val2

result =
     0   1   0   0
```

Compares two arrays of same size, element-wise

Combining logical values

Combining logical values

Use “`&&`” or “`||`” to combine *scalar* logicals

Combining logical values

Use “`&&`” or “`||`” to combine *scalar* logicals

```
>> 0 && 1
```

Combining logical values

Use “`&&`” or “`||`” to combine *scalar* logicals

```
>> 0 && 1


ans =
       0
```

Combining logical values

Use “`&&`” or “`||`” to combine *scalar* logicals

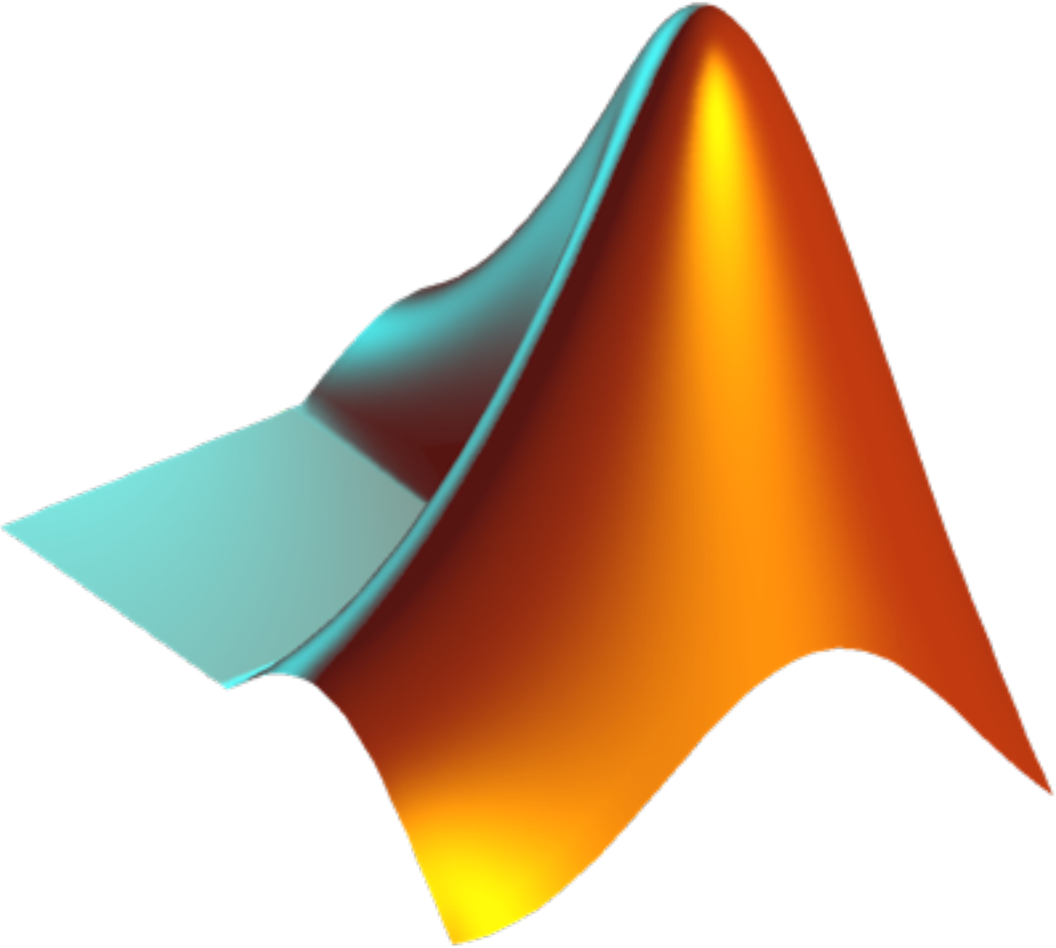
```
>> 0 && 1


ans =
       0


>> 0 || 1
```

Combining logical values

Use “&&” or “||” to combine *scalar* logicals

```
>> 0 && 1


ans =

     0


>> 0 || 1


ans =

     1
```

Combining logical values



What is truth?

What is truth?

Value	`(if w) == ?`

What is truth?

Value

`(if w) == ?`

```
w = -1;
```

What is truth?

Value	`(if w) == ?`
`w = -1;`	`true`

What is truth?

Value	(if w) == ?
`w = -1;` →	`true`
`x = 0.5;`	

What is truth?

Value	`(if w) == ?`
`w = -1;`	`true`
`x = 0.5;`	`true`

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = ‘a string’;`	

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`

What is truth?

Value	(if w) == ?
w = -1;	true
x = 0.5;	true
y = 'a string';	true
z = Inf;	

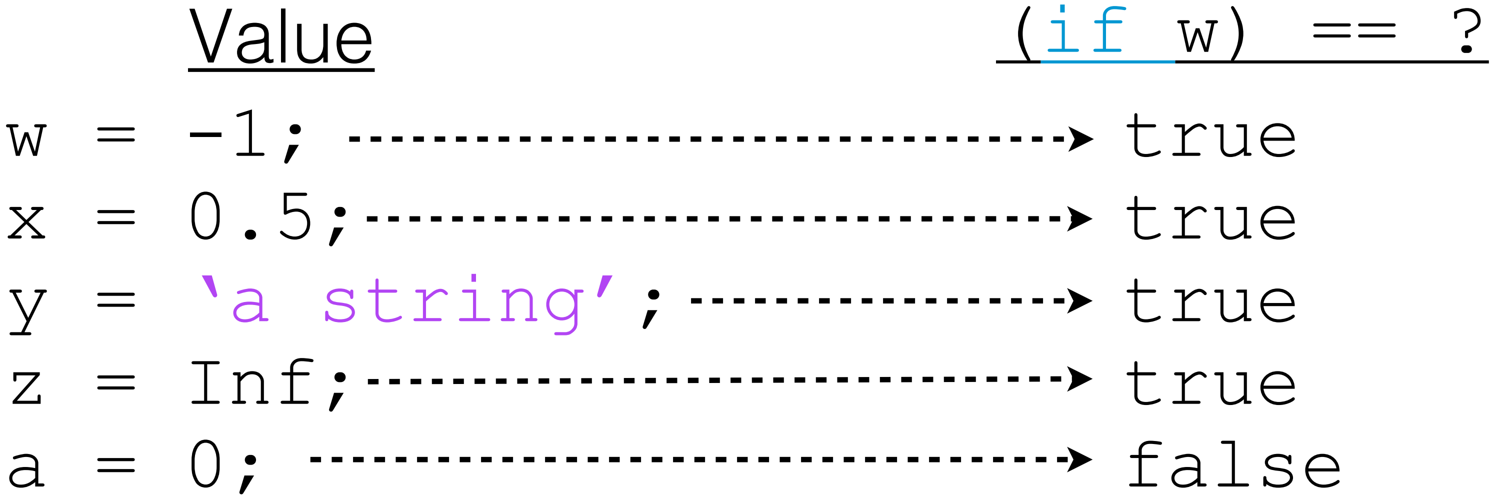
What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`
`z = Inf;`	`true`

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`
`z = Inf;`	`true`
`a = 0;`	

What is truth?



	Value	(if w) == ?
w =	-1;	true
x =	0.5;	true
y =	'a string';	true
z =	Inf;	true
a =	0;	false

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`
`z = Inf;`	`true`
`a = 0;`	`false`
`b = [0, 1];`	

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`
`z = Inf;`	`true`
`a = 0;`	`false`
`b = [0, 1];`	`?`

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`
`z = Inf;`	`true`
`a = 0;`	`false`
`b = [0, 1];`	`?`

Any scalar value except 0 evaluates to true

What is truth?

```
      Value                          (if w) == ?
w = -1;             →  true
x = 0.5;            →  true
y = 'a string';     →  true
z = Inf;            →  true
a = 0;              →  false
b = [0, 1];         →  ?
```

Any scalar value except 0 evaluates to true

Arrays evaluate to true if *all* elements are true

What is truth?

Value	(if w) == ?
`w = -1;`	`true`
`x = 0.5;`	`true`
`y = 'a string';`	`true`
`z = Inf;`	`true`
`a = 0;`	`false`
`b = [0, 1];`	`?`

Any scalar value except 0 evaluates to true

Arrays evaluate to true if *all* elements are true

Generally do direct tests:

```
(if w == -1)
```

Our code thus far

Our code thus far

```
>> data1 = load_data(file1);
```

Our code thus far

```
>> data1 = load_data(file1);
>> process_data(data1);
```

Our code thus far

```
>> data1 = load_data(file1);
>> process_data(data1);
>> data2 = load_data(file2);
```

Our code thus far

```
>> data1 = load_data(file1);
>> process_data(data1);
>> data2 = load_data(file2);
>> process_data(data2)
.
.
.
>> process_data(data87)
```

Our code thus far

```
>> data1 = load_data(file1);
>> process_data(data1);
>> data2 = load_data(file2);
>> process_data(data2)
.
.
.
>> process_data(data87)
>> publish_results();
```

Our code thus far

```
>> data1 = load_data(file1);
>> process_data(data1);
>> data2 = load_data(file2);
>> process_data(data2)
.
.
.
>> process_data(data87)
>> publish_results();
```

Loops to the rescue

Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow

Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow

Loops

Allow repeated execution of a block of code

Loops

Allow repeated execution of a block of code

For loop:

Loops

Allow repeated execution of a block of code

For loop:

```
for i = N:M
  do_something;
end
```

Loops

Allow repeated execution of a block of code

For loop:

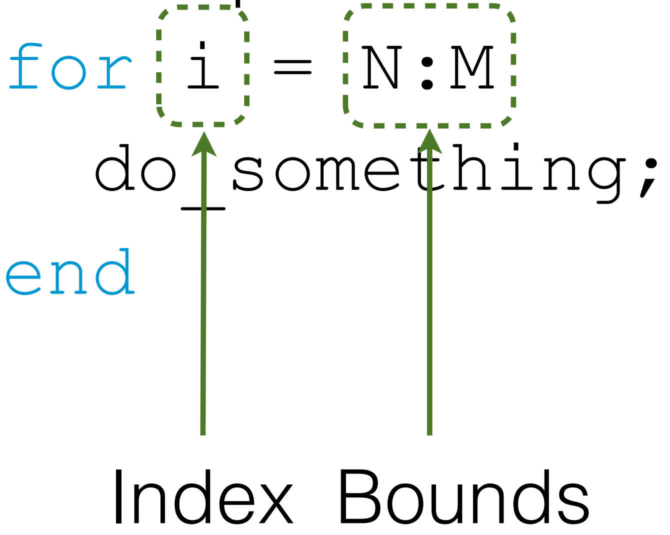
```
for i = N:M
   do_something;
end
```

Index

Loops

Allow repeated execution of a block of code

For loop:



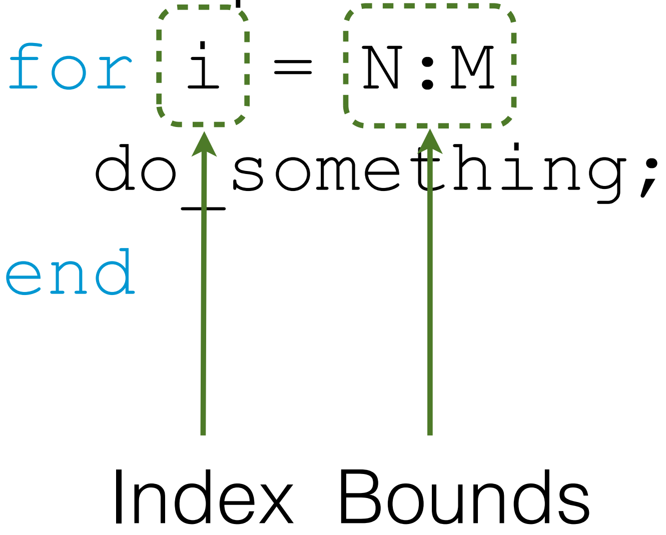
```
for i = N:M
   do_something;
end
```

Index Bounds

Loops

Allow repeated execution of a block of code

For loop:



```
for i = N:M
   do_something;
end
```

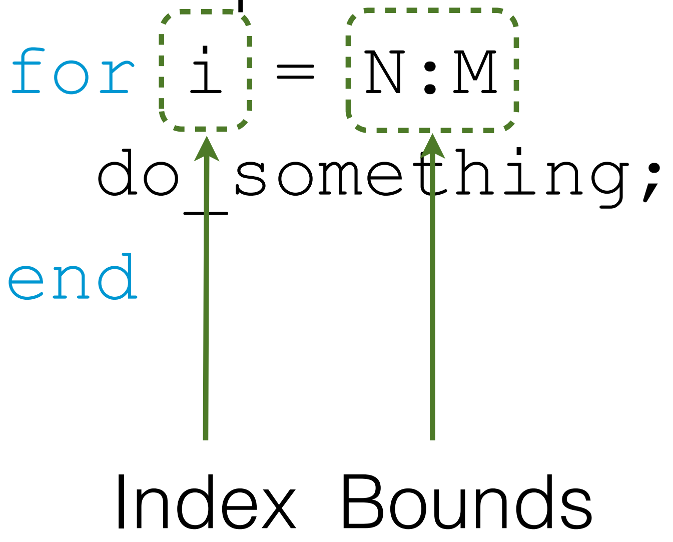
Index Bounds

While loop:

Loops

Allow repeated execution of a block of code

For loop:



```
for i = N:M
  do_something;
end
```

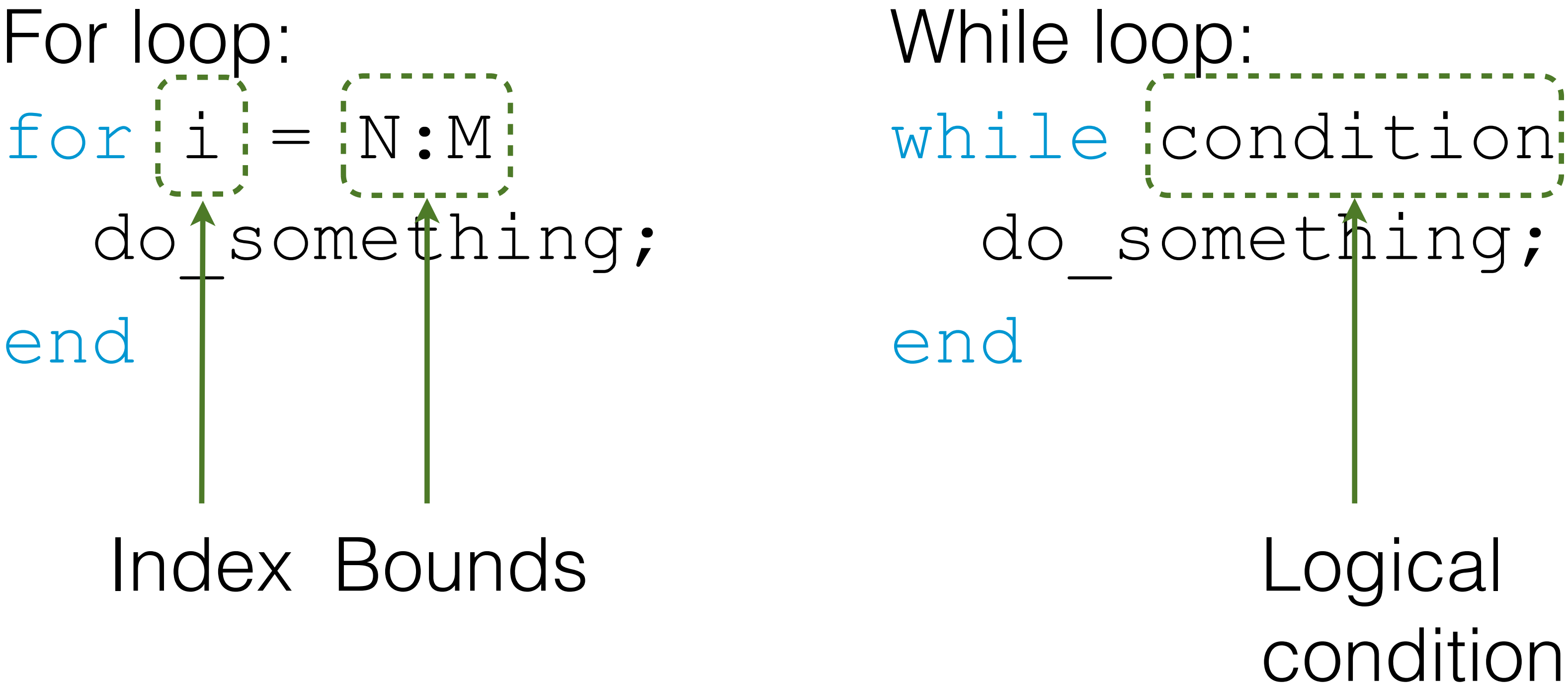
Index Bounds

While loop:

```
while condition
  do_something;
end
```

Loops

Allow repeated execution of a block of code



For loop:

```
for i = N:M
  do_something;
end
```

Index Bounds

While loop:

```
while condition
  do_something;
end
```

Logical condition

Loops

Loops

```
>> for i = 1:5
```

Loops

```
>> for i = 1:5
fprintf(‘“i” is now %d\n’, i);
```

Loops

```
>> for i = 1:5
fprintf(‘“i” is now %d\n’, i);
end
“i” is now 1
“i” is now 2
“i” is now 3
“i” is now 4
“i” is now 5
>>
```

Loops

Loops

```
>> i = 1;
```

Loops

```
>> i = 1;
>> while i <= 5
```

Loops

```
>> i = 1;
>> while i <= 5
fprintf(‘“i” is now %d\n’, i);
```

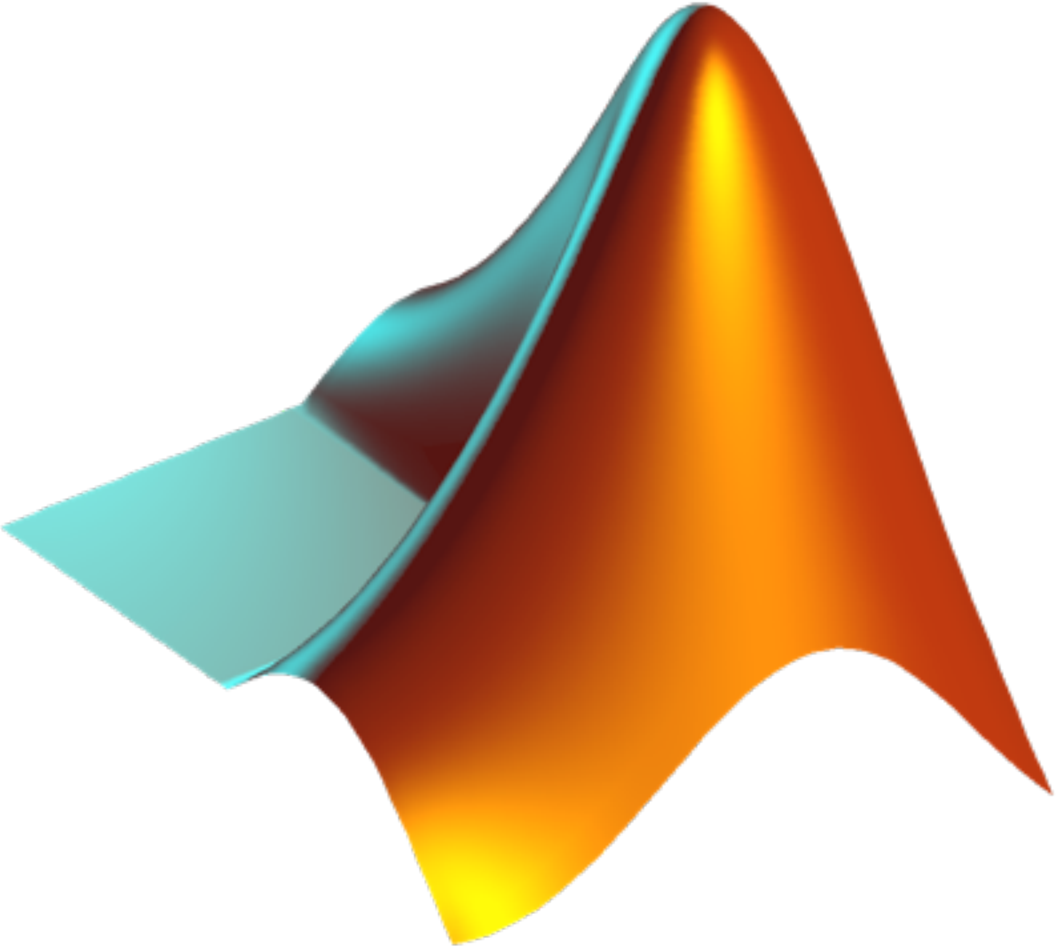
Loops

```
>> i = 1;
>> while i <= 5
fprintf('“i” is now %d\n’, i);
i = i + 1;
```

Loops

```
>> i = 1;
>> while i <= 5
fprintf(‘“i” is now %d\n’, i);
i = i + 1;
end
“i” is now 1
“i” is now 2
“i” is now 3
“i” is now 4
“i” is now 5
>>
```

Loops: factorials and plotting



Loops: index variables

Difficult to understand code's purpose
from loop variable names

Loops: index variables

```
for i = 1:n
 for j = 1:m
  for k = 1:b
   get_grade(i, j, k);
  end
 end
end
```

Difficult to understand code’s purpose
from loop variable names

Loops: index variables

Better

Loops: index variables

```
for si = 1:num_students
  for ai = 1:num_assignments
    for pi = 1:num_problems
       get_grade(si, ai, pi);
    end
  end
end
```

Better

Loops: index variables

Best!

Loops: index variables

```
for student = 1:num_students
  for assgn = 1:num_assignments
    for problem = 1:num_problems
       get_grade(student, assgn, ...
                 problem);
    end
  end
end
```

Best!

Which loop type?

We can use a `for` loop wherever can use a `while` loop, and vice versa.

`for`

- Use to repeat code a predetermined number of times
- Automatic tracking of index variable

`while`

- Use to repeat code as long as condition is true
- Automatic tracking of condition’s truth value

Which loop type?

Which loop type?

```
start = 10;
fact = 1;
for ni = start:-1:2
  fact = fact * ni;
end
```

Which loop type?

```
start = 10;
fact = 1;
for ni = start:-1:2
  fact = fact * ni;
end
```

```
start = 10;
ni = start;
fact = 1;
while ni > 1
  fact = fact * number;
  ni = ni - 1;
end
```

Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow

Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow

Control flow

Control flow

Methods for fine-grained control of loops

Control flow

Methods for fine-grained control of loops

1. Combining loops and branching

Control flow

Methods for fine-grained control of loops

1. Combining loops and branching
2. `continue`: Skip to next loop iteration

Control flow

Methods for fine-grained control of loops

1. Combining loops and branching
2. `continue`: Skip to next loop iteration
3. `break`: Exit loop altogether

Control flow: loops and branching

Control flow: loops and branching

```
for i = 1:N
  % check condition on each
  % loop iteration
  if condition
    do_this();
  else
    do_that();
  end
end
```

Control flow: loops and branching

Control flow: loops and branching

```
for i = 1:100
  if is_even(i)
    fprintf(‘%d is even\n’, i);
  else
    fprintf(‘%d is odd\n’, i);
  end
end
```

Control flow: `continue`

Skip to next loop iteration

Control flow: continue

Skip to next loop iteration

```
for i = 1:num_datasets
  if (~meets_criteria(i))
    continue;  % skips the rest of
               % the loop, but still
               % increments i
  else
    process_data(i);
  end
end
```

Control flow: `break`

Exit loop altogether

Control flow: break

Exit loop altogether

```
for i = 1:num_datasets
  if (meets_criteria(i))
    break; % abort the loop as soon
           % as we find valid data
  end
end
% break moves us here
process_data(i);
```

Control flow: `break`

Continuous loops

Control flow: break

Continuous loops

```
while 1 % loop forever
  new_data = get_more_data();
  if isempty(new_data)
    break; % no more data, exit loop
  end
  process_data(new_data);
end
```

Problem set 3

Problem set 3

Förster resonance energy transfer (FRET)

Problem set 3

Förster resonance energy transfer (FRET)

1. Two nearby light-sensitive molecules, chromophores

Problem set 3

Förster resonance energy transfer (FRET)

1. Two nearby light-sensitive molecules, **chromophores**
2. Excited chromophore may **donate** energy to neighbor

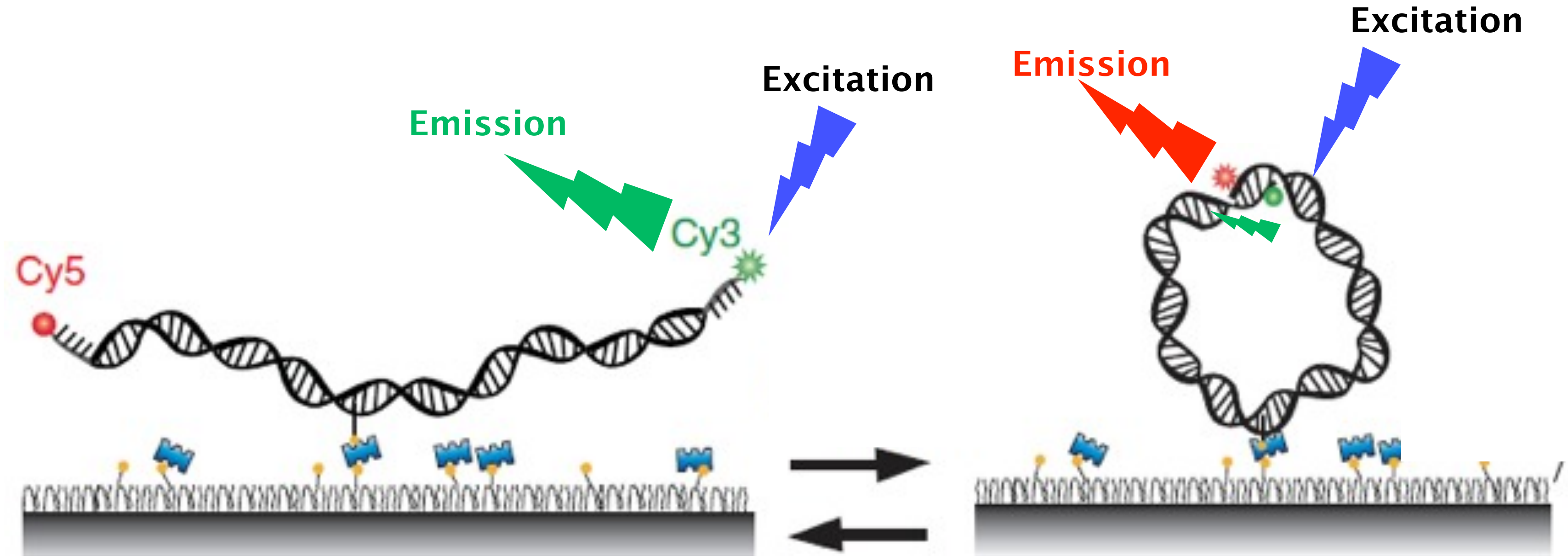
Problem set 3

Förster resonance energy transfer (FRET)

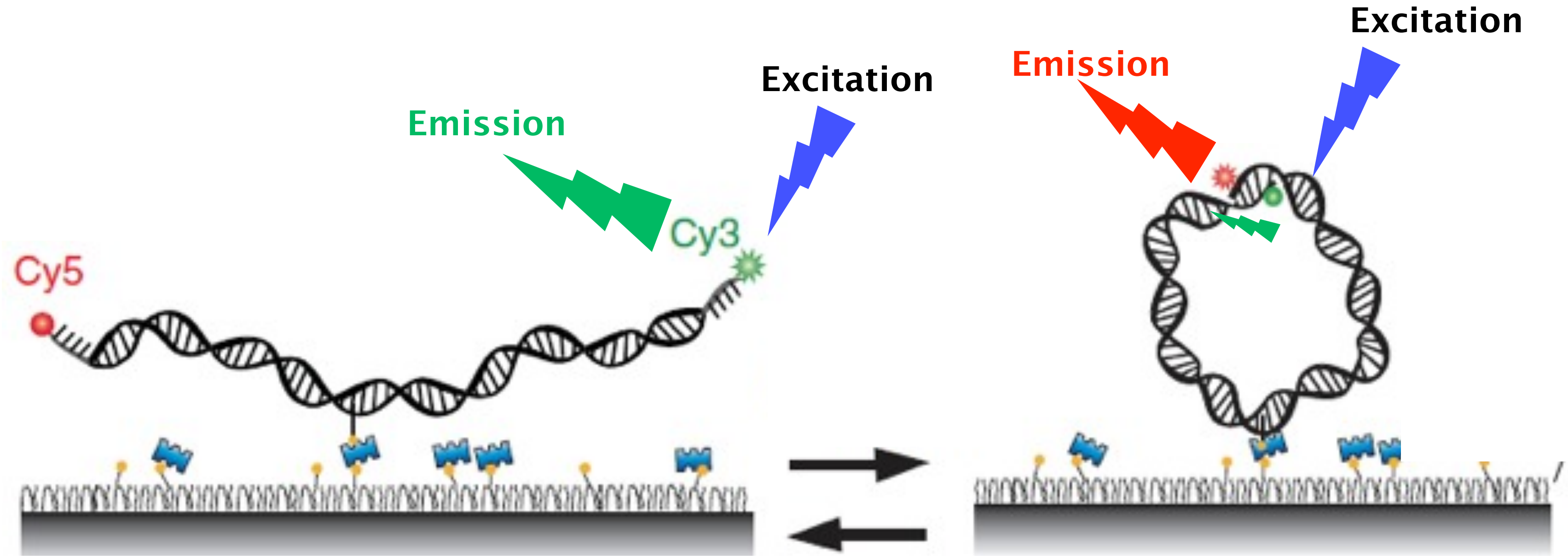
1. Two nearby light-sensitive molecules, **chromophores**
2. Excited chromophore may **donate** energy to neighbor
3. Energy transfer proportional to distance

Problem set 3

Problem set 3

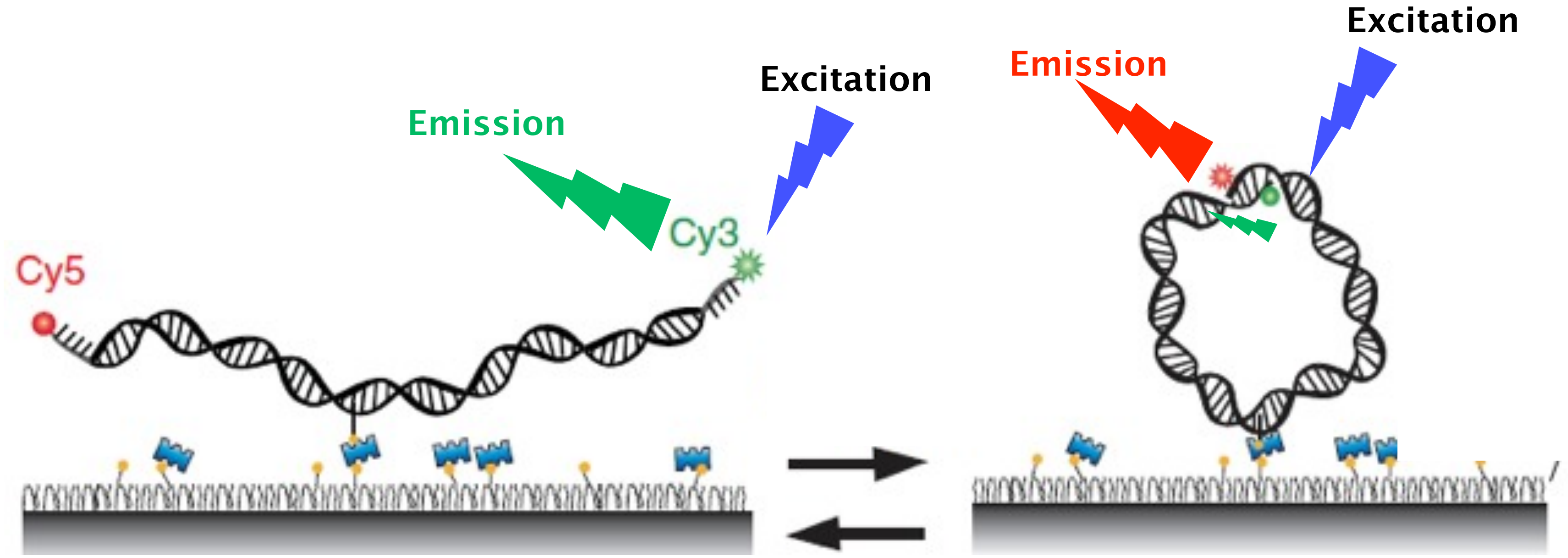


Problem set 3



Modified from: Vafabakhsh R and Ha T. (2012) Extreme Bendability of DNA Less than 100 Base Pairs Long Revealed by Single-Molecule Cyclization. Science, 337: 1097 (2012)

Problem set 3



Modified from: Vafabakhsh R and Ha T. (2012) Extreme Bendability of DNA Less than 100 Base Pairs Long Revealed by Single-Molecule Cyclization. Science, 337: 1097 (2012)

$$\frac{F_A}{F_A + F_D}$$

Review

Review

- Branching

Review

- Branching
  - `if`: execute code if condition true

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops
  - `for`: execute block defined number of times

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops
  - `for`: execute block defined number of times
  - `while`: execute block as long as condition true

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops
  - `for`: execute block defined number of times
  - `while`: execute block as long as condition true
- Control flow

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops
  - `for`: execute block defined number of times
  - `while`: execute block as long as condition true
- Control flow
  - combine loops and branching

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops
  - `for`: execute block defined number of times
  - `while`: execute block as long as condition true
- Control flow
  - combine loops and branching
  - `continue`: skip this loop iteration, start next

Review

- Branching
  - `if`: execute code if condition true
  - `else`: execute code if condition false
  - `elseif` & `switch/case`: test multiple statements
- Loops
  - `for`: execute block defined number of times
  - `while`: execute block as long as condition true
- Control flow
  - combine loops and branching
  - `continue`: skip this loop iteration, start next
  - `break`: exit loop altogether