

# Computer Programming with MATLAB

## MM1CPM - Lecture 8

### Parallel programming, Functions - reusable code

Dr. Roderick MacKenzie

[roderick.mackenzie@nottingham.ac.uk](mailto:roderick.mackenzie@nottingham.ac.uk)

Autumn 2014



# Overview

---

- **Recap of last lecture on algorithms**
- Parallel computing
- Reusable code
  - Functions
  - A real world example of functions

# Recap: The bubble sort algorithm

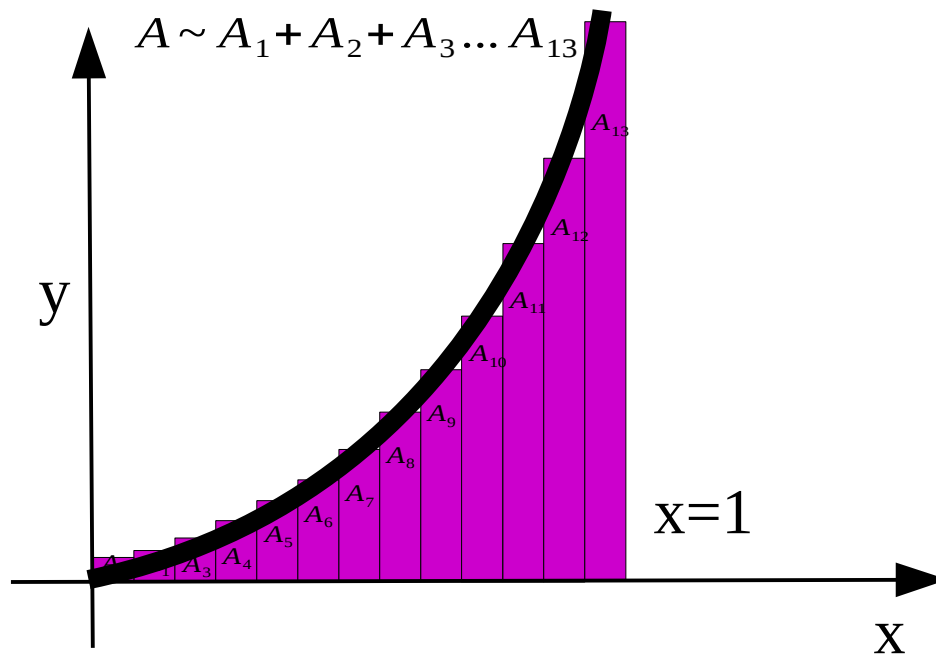
- Last time we learnt about sorting data using a bubble sort algorithm.



- Swap the numbers around, only if they are the wrong way around.

# Recap: Integration with a computer

- We learnt that we can calculate the area under a curve by dividing it into boxes.
- Then using a **for** loop to sum up the area under the curve.
- This is a very simple way to evaluate any integral.

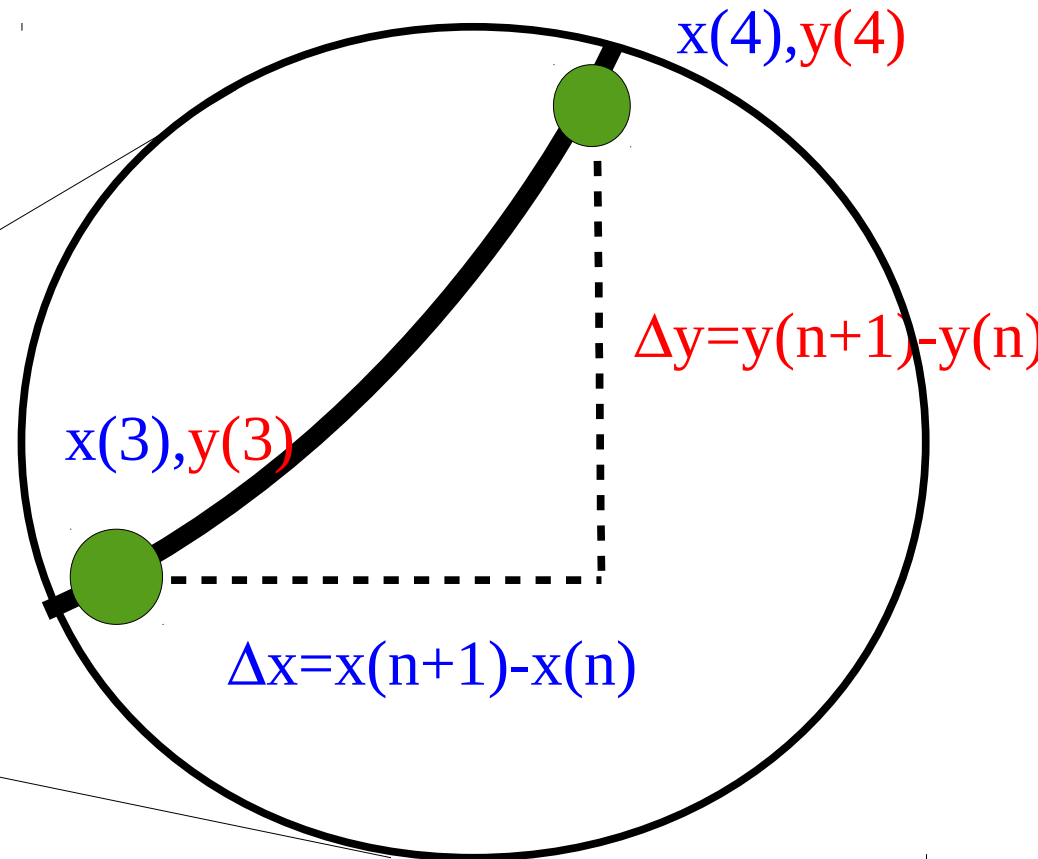
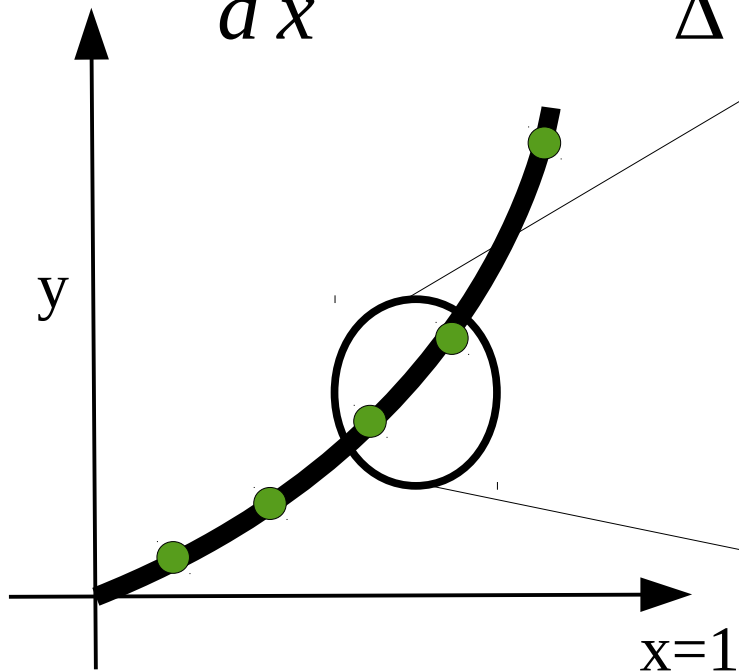


```
x=0
my_max=1.0
box_width=0.01
sum=0
while (x<my_max)
    x=x+box_width
    y=x*x
    box_area=y*box_width
    sum=sum+box_area
end
disp(sum)
```

# Recap: Numerical differentiation

- We also learnt an algorithm for calculating the derivative of any curve.
- We just find the x/y change between each point and calculate the slope.

$$\frac{d y}{d x} = \text{slope} = \frac{\Delta y}{\Delta x}$$



# Overview

---

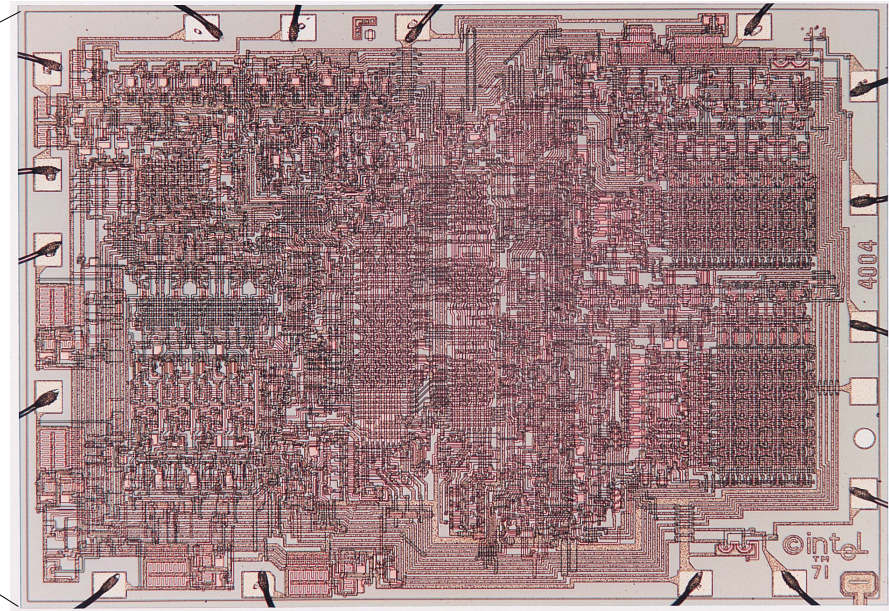
- Recap of last lecture on algorithms
- Parallel computing**
- Reusable code
  - Functions
  - A real world example of functions

# Non-parallel computing

Before I teach you about parallel computer I need to teach you about non parallel computing.



Processor

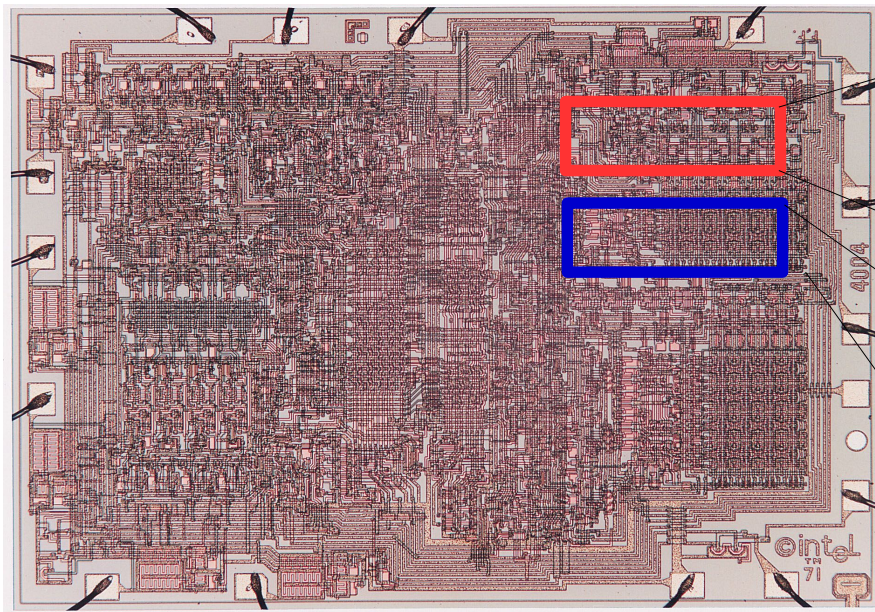


The electronics inside a processor

This is the chip that does all the maths and runs your program - understands (**if**, **while**, **for**, + , - , \*)

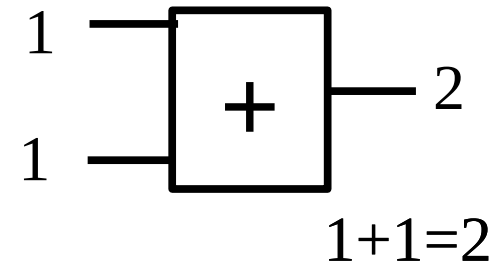
# Non parallel computing

Different parts of the computer chip do different things:

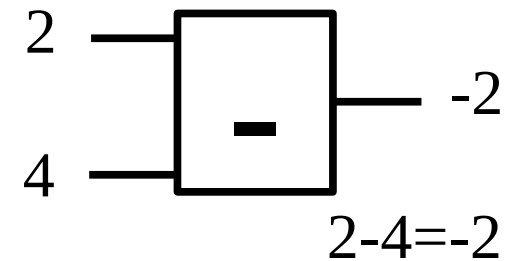


The electronics inside a processor

Adding unit



Subtracting unit



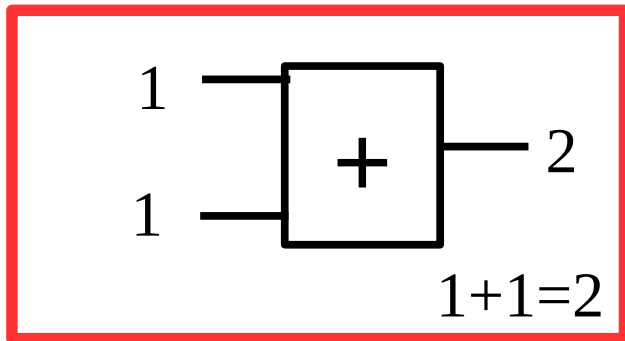
There are also multiplying units, dividing units, units for, if, while and for statements etc...



# Think about the following MATLAB program

Every time we tell the computer to add, we use the adding unit in the processor. Here is a program that uses the adding unit a lot.

Adding unit

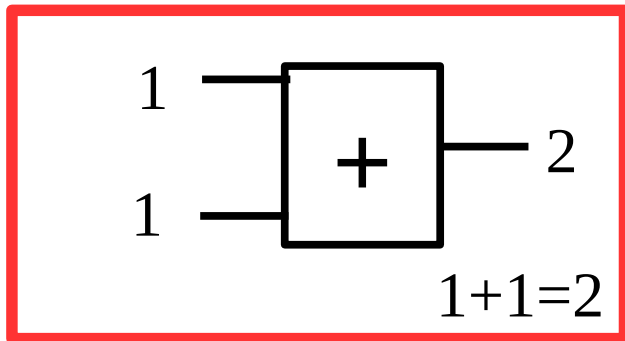


```
a=1+1  
b=1+2  
c=2+2  
d=4+4  
e=7+7  
f=1+2  
g=3+3  
h=1+1  
i=1+1  
j=1+2  
k=2+2  
l=4+4  
m=7+7  
n=1+2  
o=3+3
```

# Think about the following MATLAB program

Every time we tell the computer to add, we use the adding unit in the processor. Here is a program that uses the adding unit a lot.

Adding unit



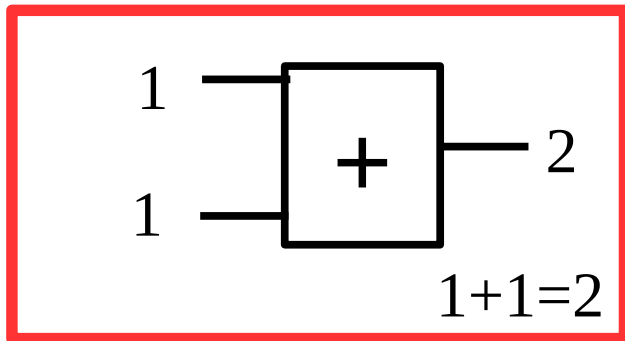
I need a volunteer

```
a=1+1  
b=1+2  
c=2+2  
d=4+4  
e=7+7  
f=1+2  
g=3+3  
h=1+1  
i=1+1  
j=1+2  
k=2+2  
l=4+4  
m=7+7  
n=1+2  
o=3+3
```

# Think about the following MATLAB program

Every time we tell the computer to add, we use the adding unit in the processor. Here is a program that uses the adding unit a lot.

Adding unit



I need a volunteer  
(chocolate on offer)

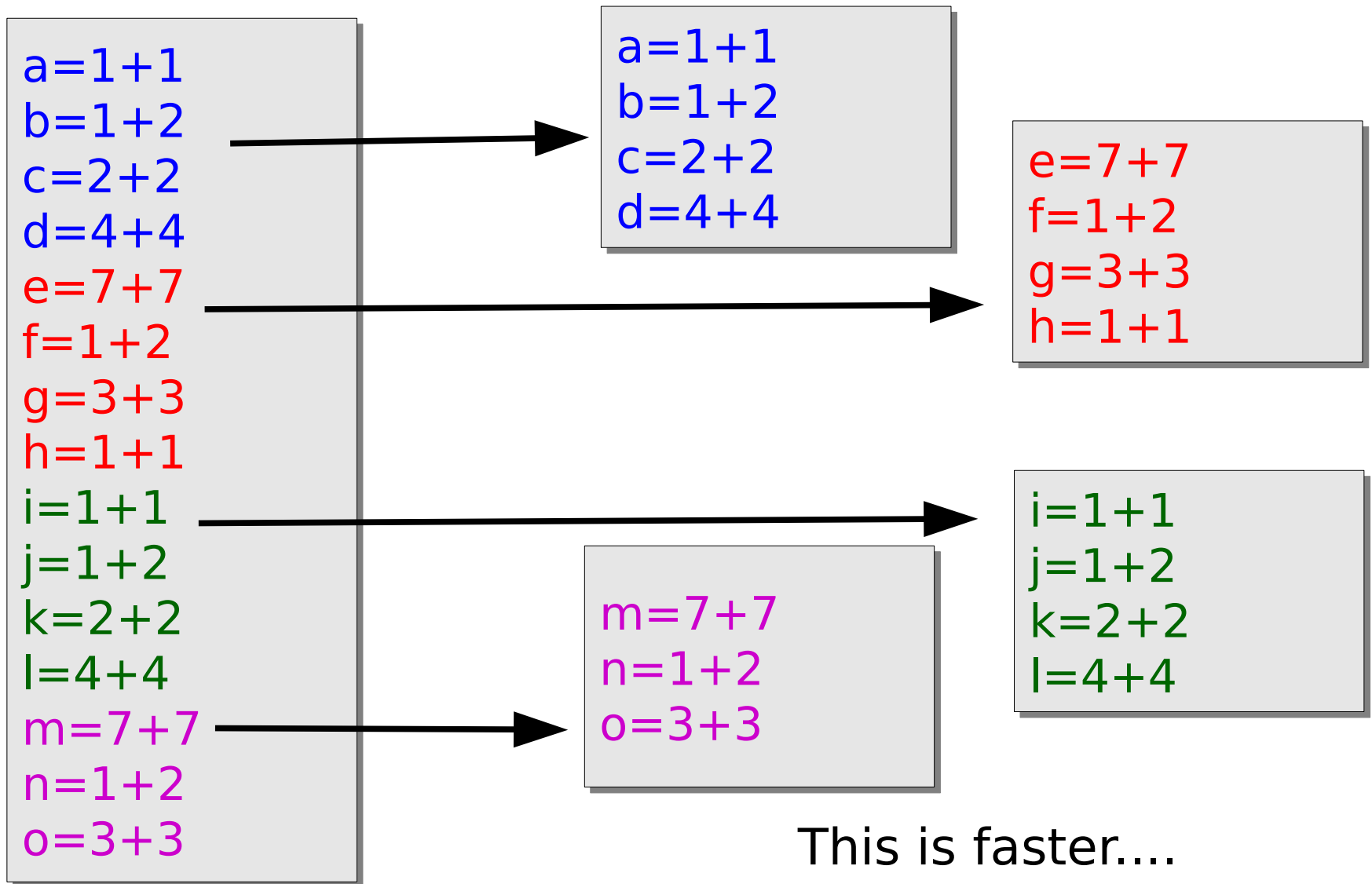
```
a=1+1  
b=1+2  
c=2+2  
d=4+4  
e=7+7  
f=1+2  
g=3+3  
h=1+1  
i=1+1  
j=1+2  
k=2+2  
l=4+4  
m=7+7  
n=1+2  
o=3+3
```

# Parallel computing – a different strategy

---

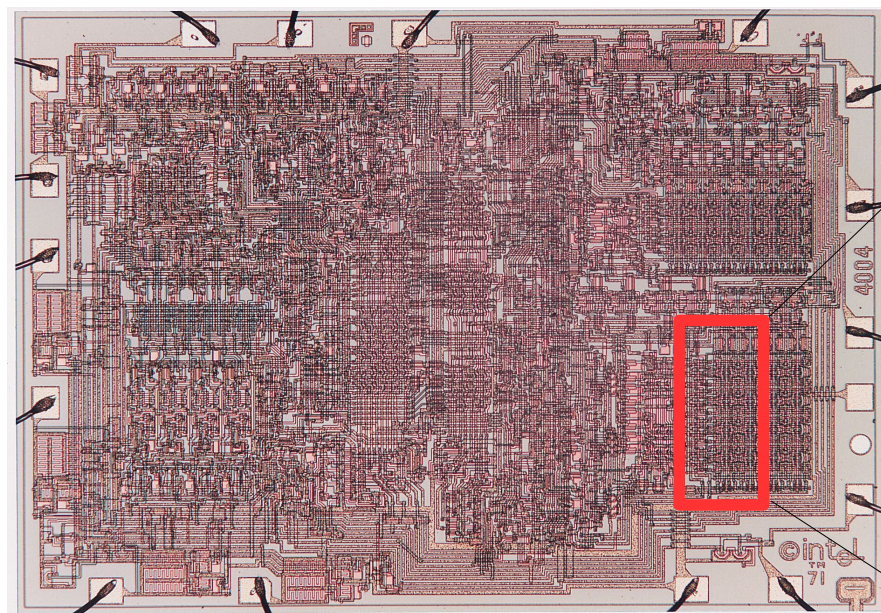
I need three more volunteers  
(more chocolate on offer)

# We are running our tasks in parallel



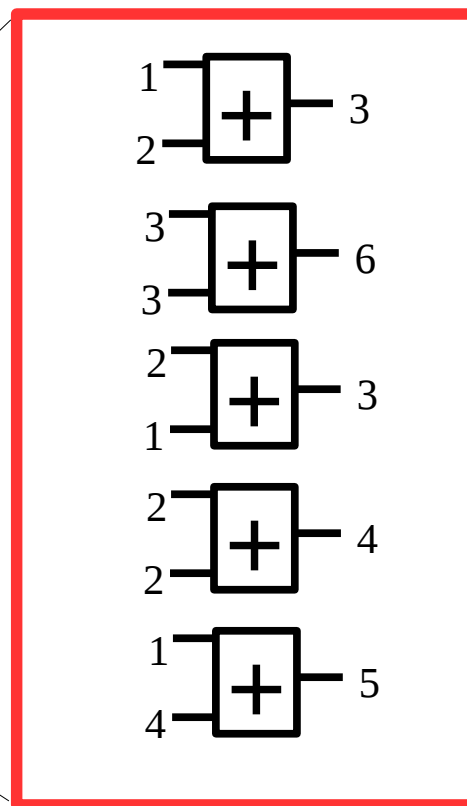
# Parallel computing

Modern computers can do something called a parallel add (multiply, subtract, divide etc..), they can do lots of operations a the same time



The electronics inside a processor

Parallel adding unit



# Parallel computing in MATLAB

Non-parallel code:

```
a=1+1
b=1+2
c=2+2
d=4+4
e=7+7
f=1+2
g=3+3
h=1+1
i=1+1
j=1+2
k=2+2
l=4+4
.....
```

Parallel code:

```
x=[1 1 2 4 7 1 3 1 1 1 2 4]
y=[1 2 2 4 7 2 3 1 1 2 2 4]
z=x+y
```

- When ever you do matrix operations in MATLAB the computer uses parallel hardware to do them.
- They are therefore faster.

# Parallel computing in MATLAB

Non-parallel code:

```
tic
a=rand(1000,1000)
b=rand(1000,1000)
z=zeros(1000,1000)

for x=1:100
    for y=1:100
        z(y,x)=b(y,x)+a(y,x);
    end
end
toc
```

Parallel code:

```
tic
a=rand(1000,1000);
b=rand(1000,1000);
z=zeros(1000,1000);
z=a+b;
toc
```



# Overview

---

- Recap of last lecture on algorithms
- Parallel computing
- Reusable code**
  - Functions
  - A real world example of functions

# Reusable code

- Often when using MATLAB you will spend a lot of time developing a program.
- For example our bubble sort algorithm:



```
numbers=[3 2 1 5 4]      %our array we want to sort
for ii=i:5               %pass five times over the data
  for i=1:4               %start of a for loop to count
    if (numbers(i)>numbers(i+1)) %if statement
      temp=numbers(i);    %do the swap
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
```

- This program will sort **one** list of numbers.

# Reusable code

- But what would you do if you wanted to **sort two** lists of numbers one after each other?
- Copy and paste the code twice right?

```
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
```

First copy of  
the code sorting  
[ 3 2 1 5 4]

First copy of  
the code sorting  
[ 1 4 3 5 2]

- What if you wanted to sort 6 lists of numbers

# Sort six lists of numbers?

- Make six copies of the code?

```
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
end
end
```

First copy of the code  
[1 1 1 5 4]

Second copy of the code  
[3 2 2 5 4]

Third copy of the code  
[8 2 1 5 9]

```
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
end
end
```

Forth copy of the code  
[3 6 1 2 4]

Fifth copy of the code  
[1 2 8 5 8]

Sixth copy of the code  
[8 2 9 8 4]

- And 1000 lists of numbers?
- This does not seem like a good plan...

# Functions

- To get around this problem we take our code that we want to use again and again, and put it in a special container called a function.

function

```
numbers=[3 2 1 5 4]
for ii=i:5
for i=1:4

if (numbers(i)>numbers(i+1))

temp=numbers(i);
numbers(i)=numbers(i+1);
numbers(i+1)=temp;

end
end
end
```



```
numbers=[3 2 1 5 4]
for ii=i:5
for i=1:4

if (numbers(i)>numbers(i+1))

temp=numbers(i);
numbers(i)=numbers(i+1);
numbers(i+1)=temp;

end
end
end
```

# Functions

- We then give the function a name

## my\_sorting\_function

```
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
```

# Functions

- When we want to sort a list of numbers we just **call** the function `my_sorting_function` from our program and it will sort a list of numbers for us.

```
%My program to sort ten lists of numbers
```

```
my_sorting_function([1 4 6 8 7 ])
```

```
my_sorting_function([1 7 10 4 7 ])
```

```
my_sorting_function([2 4 4 3 7 ])
```

```
my_sorting_function([3 7 6 4 6 ])
```

```
my_sorting_function([2 4 6 4 4 ])
```

```
my_sorting_function([7 6 5 7 7 ])
```

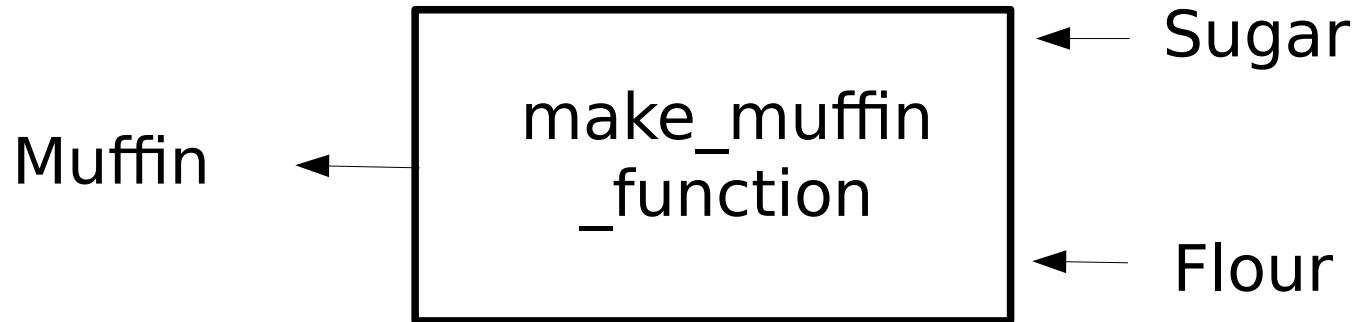
```
my_sorting_function([1 4 6 1 2 ])
```

`my_sorting_function`

```
numbers=[3 2 1 5 4]
for ii=i:5
  for i=1:4
    if (numbers(i)>numbers(i+1))
      temp=numbers(i);
      numbers(i)=numbers(i+1);
      numbers(i+1)=temp;
    end
  end
end
```

- The advantage is we only need one copy of the sorting code not 1000 copies.

# A real world example of a function

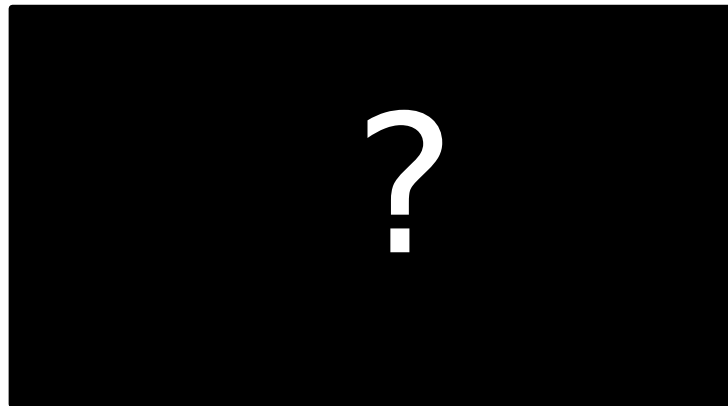
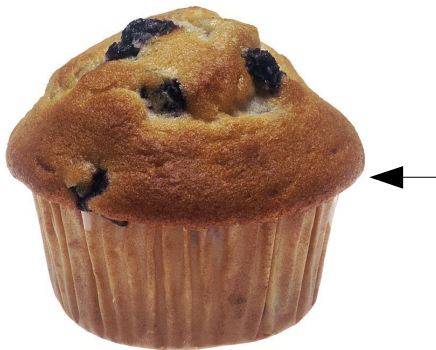




# Functions – as a black box

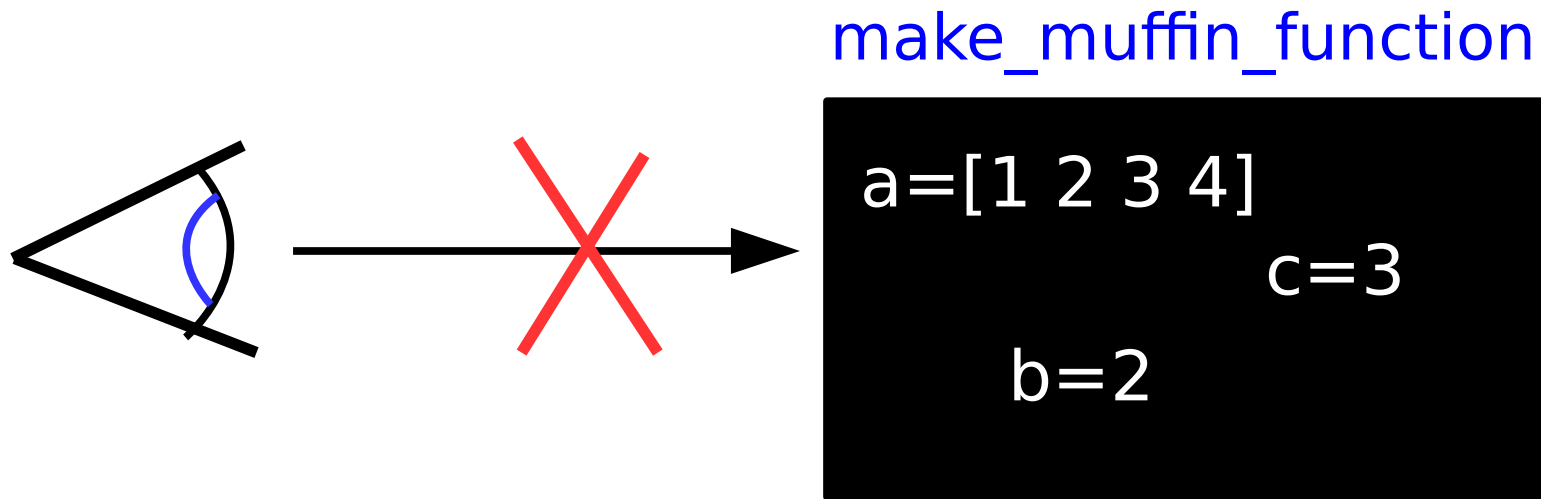
- You can treat other people's functions as a black box:
- Meaning that as long as you understand the **inputs** (sugar and flour ) and **outputs** (a muffin).
- You don't have to understand what goes on inside the function – advantages in big projects.

`make_muffin_function`



# Functions – as a black box

- Furthermore, the rest of your code can't see into a function, so any variables within it are invisible to the outside world.



- This makes functions a good way to divide your code up into chunks.
- Good when working on big projects with many people

# Functions

- You have already met and used simple functions other people have written.
- Think of the *sin* function, you don't know how it works but it does not stop you using it.



```
>y=sin(30)  
ans=0.5
```

# Functions

- You have also used many other pre-written functions, without realizing it: `cos()`, `exp()`, `tan()`, etc...

```
>cos(0)
ans=1.0
```

```
>exp(0)
ans=1.0
```

```
>tan(0)
ans=0.0
```

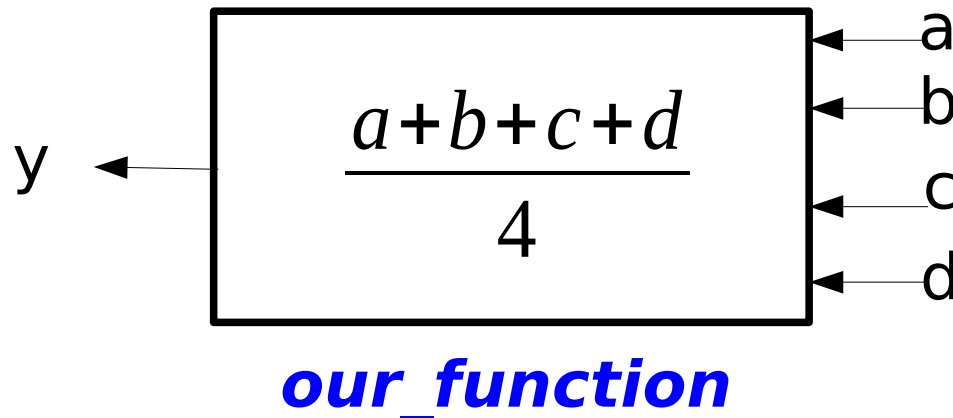
- In the rest of this lecture we will focus on how to write **your own functions.**

# Our first MATLAB function

Let's write a function called ***our\_function*** to average four numbers a,b,c,d

In mathematical form this would be:  $y = \frac{a+b+c+d}{4}$

Or in pictorial form:



## our\_function in MATLAB

We would like our function to return the average of four numbers when we type:

```
>our_function(1,2,3,4)  
ans=2.5
```

```
>our_function(2,3,4,5)  
ans=3.5
```

## Making our own function: `our_function`:

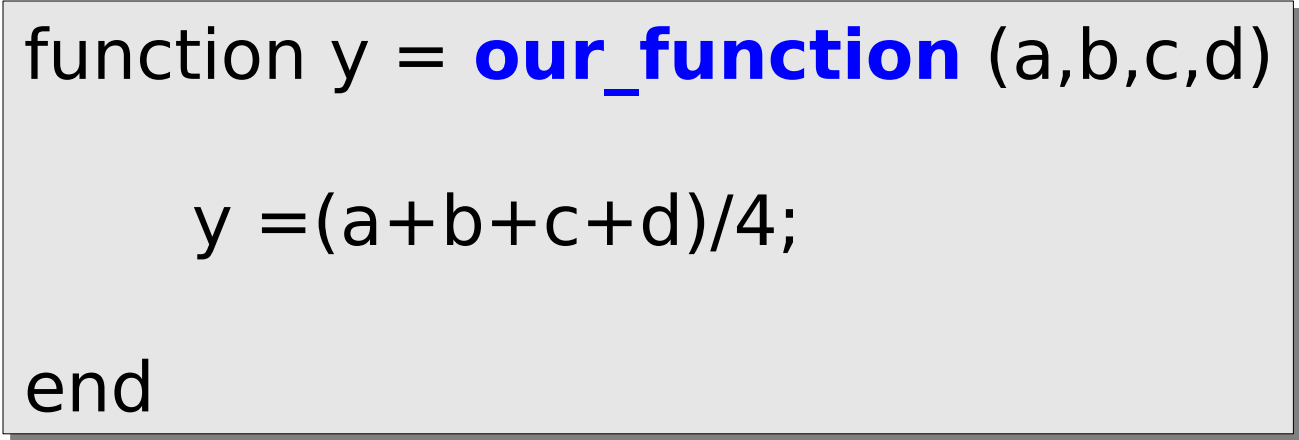
In MATLAB we can define a function in a new script file like this:

```
function y = our_function (a,b,c,d)
    y =(a+b+c+d)/4;
end
```

- All functions begin with the word **function** and ends with the word **end**.
- Any code that does something useful must go between these two lines.

## Making our own function: `my_function`:

The name of the function is defined here, we can call it **anything** we like.




```
function y = our_function (a,b,c,d)
    y =(a+b+c+d)/4;
end
```



## Making our own function: `my_function`:

The inputs to the function are defined here, in this case we have four inputs `a`, `b`, `c` and `d`.

```
function y = our_function (a,b,c,d)  
  
    y = (a+b+c+d)/4;  
  
end
```



# Getting information into a function

- Information is passed to your function using arguments (the variables in brackets)

```
>z =our_function(1,2,3,4)  
z=2.5
```

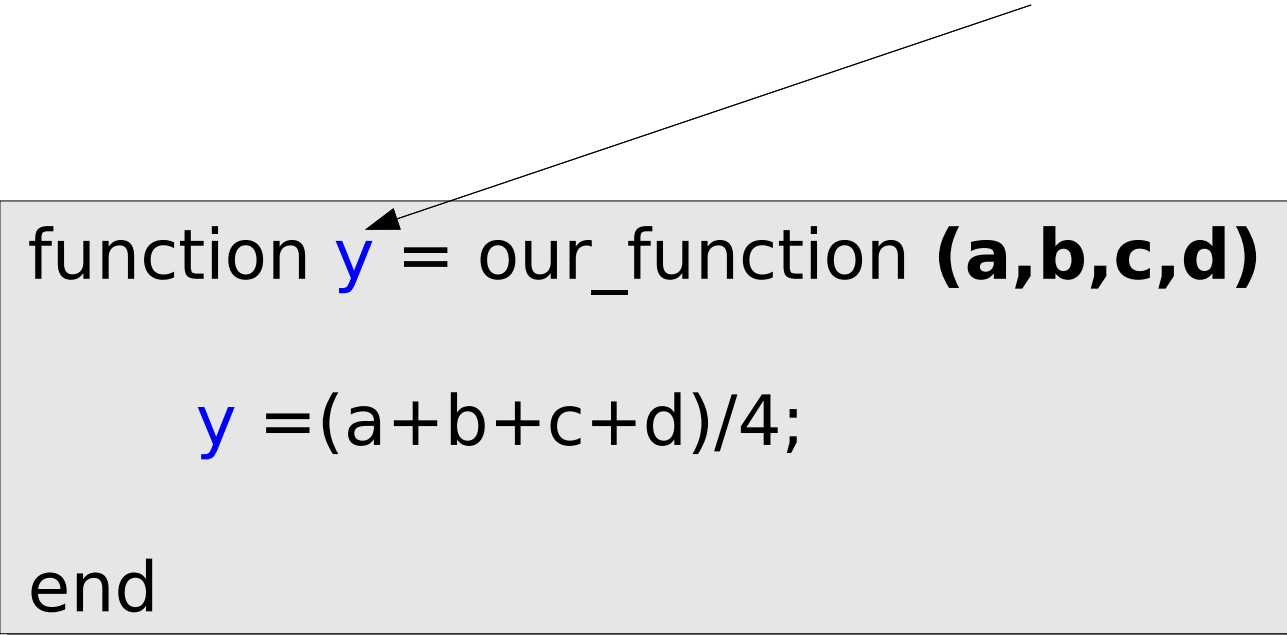


```
function y =our_function(a,b,c,d)  
  
y=(a+b+c+d)/4  
  
end
```

## Making our own function: `my_function`:

The variable which is used to return information to the rest of the program must be defined here.

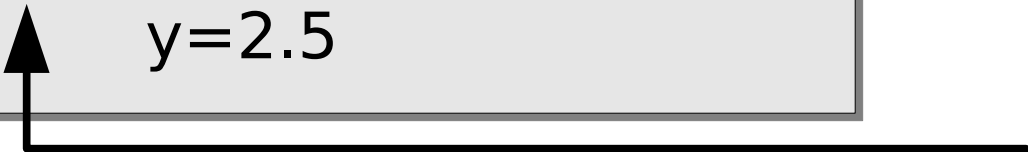
```
function y = our_function (a,b,c,d)  
  
    y = (a+b+c+d)/4;  
  
end
```



Any changes made to this variable by the function will be returned to the rest of the program. For example...

# Getting information out of a function

```
> y = our_function(1,2,3,4)  
    y = 2.5
```



```
function y = our_function(a,b,c,d)  
  
    y = (a+b+c+d)/4  
  
end
```

## Making our own function: `my_function`

- Finally, your function must be saved into its own script file.
- The name of the script file **must be the same as the name of the function**, otherwise MATLAB won't know where to look for it.

```
function y = our_function (a,b,c,d)

    y =(a+b+c+d)/4;

end
```

**NOW SAVE IN ITS OWN FILE CALLED `our_function.m`**

## Final thing about functions: The `help` command

- If you try typing 'help' then a MATLAB function name i.e.:

```
> help sin
sin    Sine of argument in radians.
       sin(X) is the sine of the elements of X.

       See also asin, sind.

       Reference page in Help browser
       doc sin
```

- It will print out help

# Defining a new function in MATLAB

You can make your own function print out help by adding comments above the definition of the function

```
% Evaluates the equation  
%(a+b+c+d)/4  
  
function y = our_function (a,b,c,d)  
    y =(a+b+c+d)/4;  
end
```

Then when you type:

```
>help our_function  
Evaluates the equation  
(a+b+c+d)/4
```

# Overview

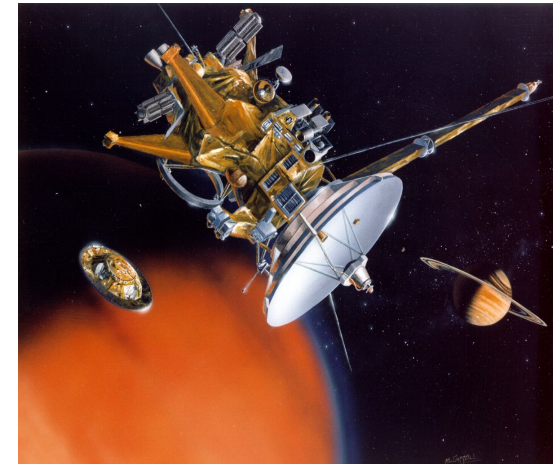
---

- Recap of last lecture on algorithms
- Parallel computing
- Reusable code
  - Functions
  - A real world example of functions**



# Case study: Navigating by the stars

- You are an engineer designing a navigation system for a space probe which will be traveling to Saturn.
- The space probe will use the position of the stars to navigate because there is no GPS signal far away from earth.



- The camera on the space probe takes this image.



# Navigating by the stars

- There are however thousands of stars along with some galaxies in this image, and they confuse the navigation system.
- It is your job to design a program (in a function) to remove everything but the brightest stars – this will improve the accuracy of the navigation system.

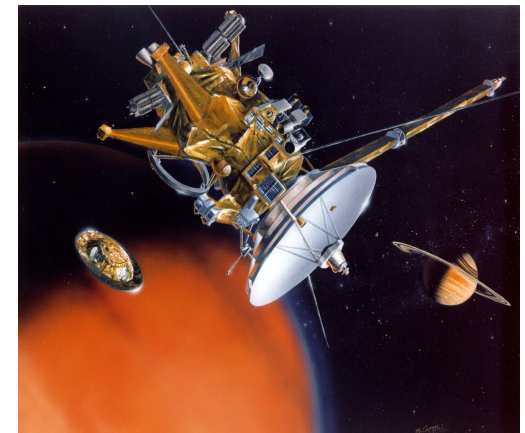


# Navigating by the stars

- The image of the stars is stored as jpg image **stars.jpg**
- The brightest points in the image have a value of 255, the darkest point in the image have a value of 0.
- Our function should remove all but the brightest pixels by setting any data points below the value of 240 to zero.



43



•Let's start...

# Loading images

To do this task we have to learn a new **function** which can load jpg images directly into 2D arrays.

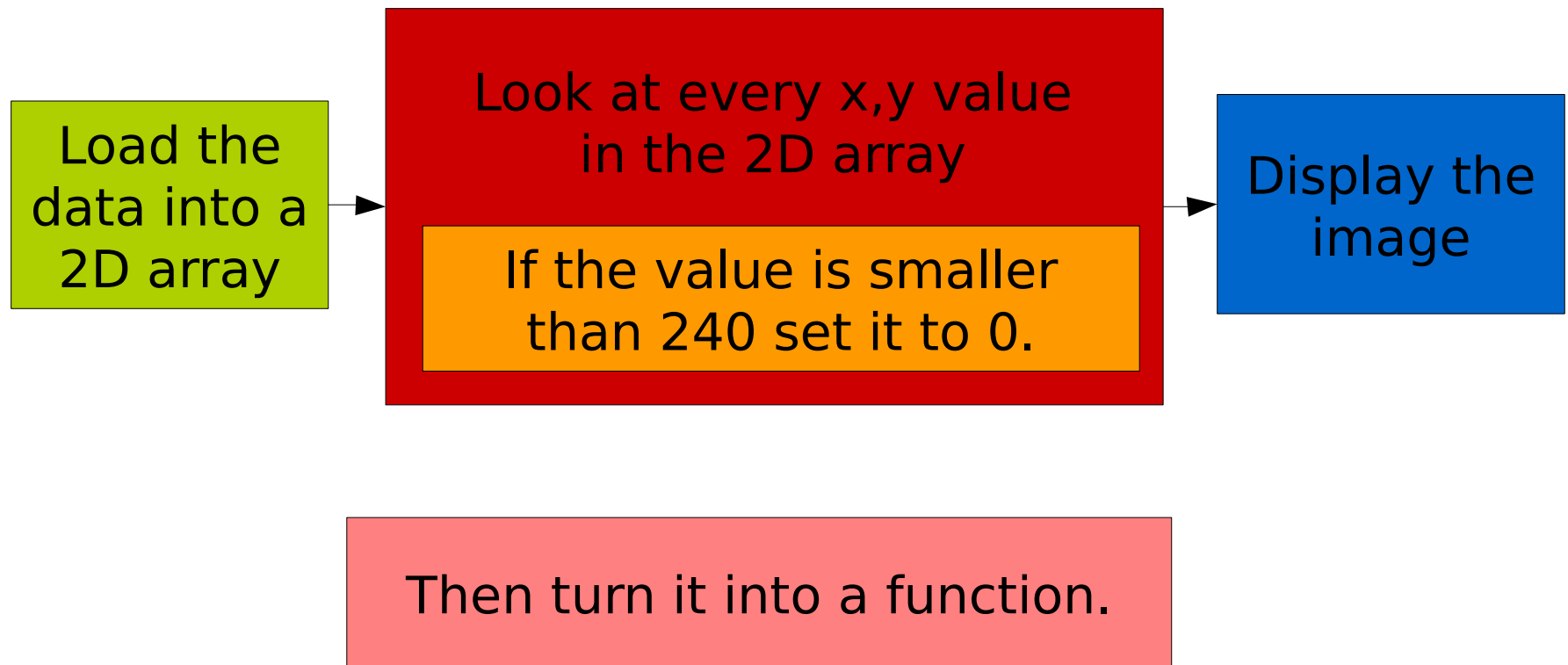
It's just like the **load** command you learnt about before but works on jpg images instead of .dat files.

```
stars = imread('stars.jpg');
```

This will give us a 2D array containing the data from the space probe's camera.

# Strategy for doing this task:

First write the program



Often it is good to break down computing tasks like this

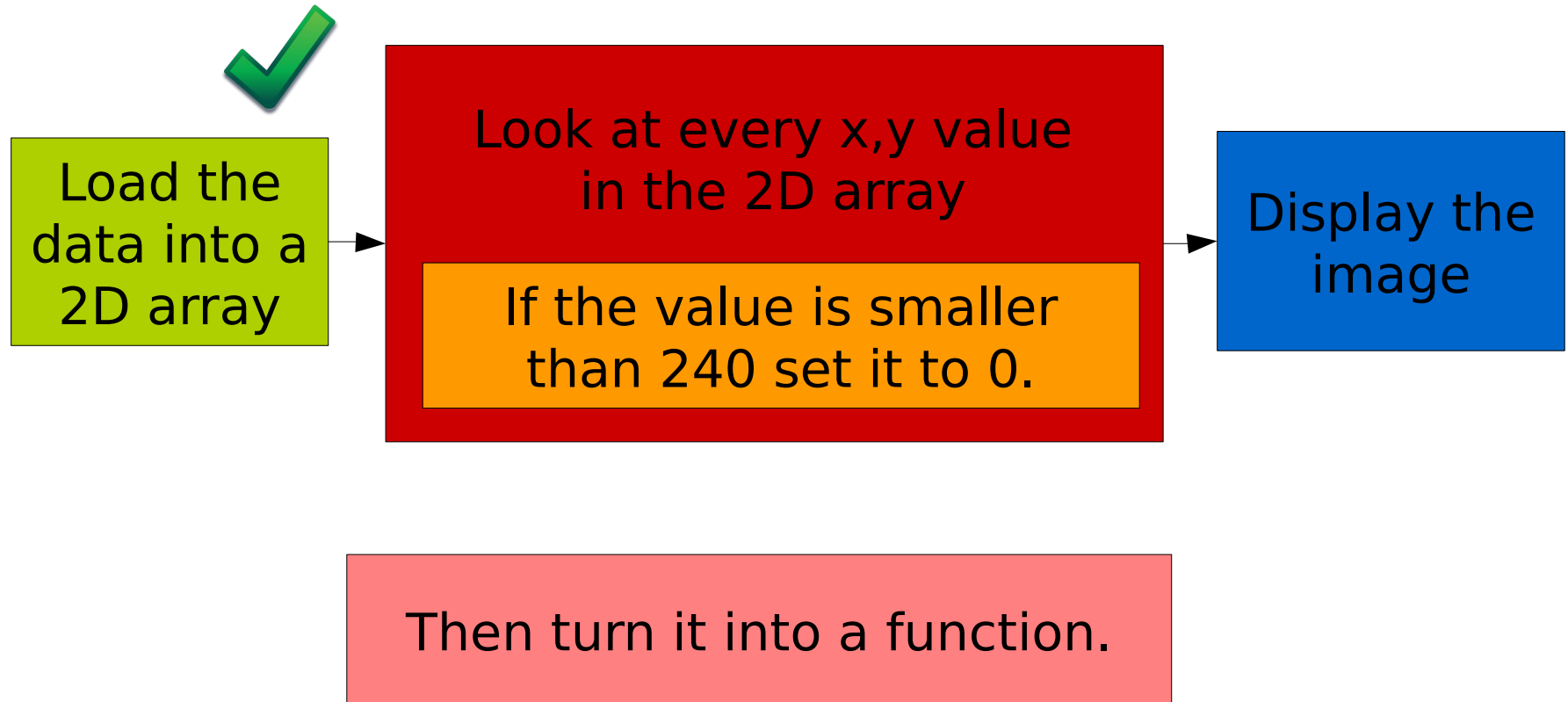
# Loading the image

- First load the image into a 2D array using `imread`

```
data = imread('stars.jpg');
```

# Strategy for doing this task:

First write the program



Often it is good to break down computing tasks like this

# Finding the size of the image

```
data = imread('stars.jpg');  
len=size(data)  
xlen=len(1)  
ylen=len(2)
```

- To check every value in the 2D array are going to need to index each value, using `data(x,y)`
- Therefore to know how big the 2D array is would seem a sensible thing to do.



# Finding the value of each data point

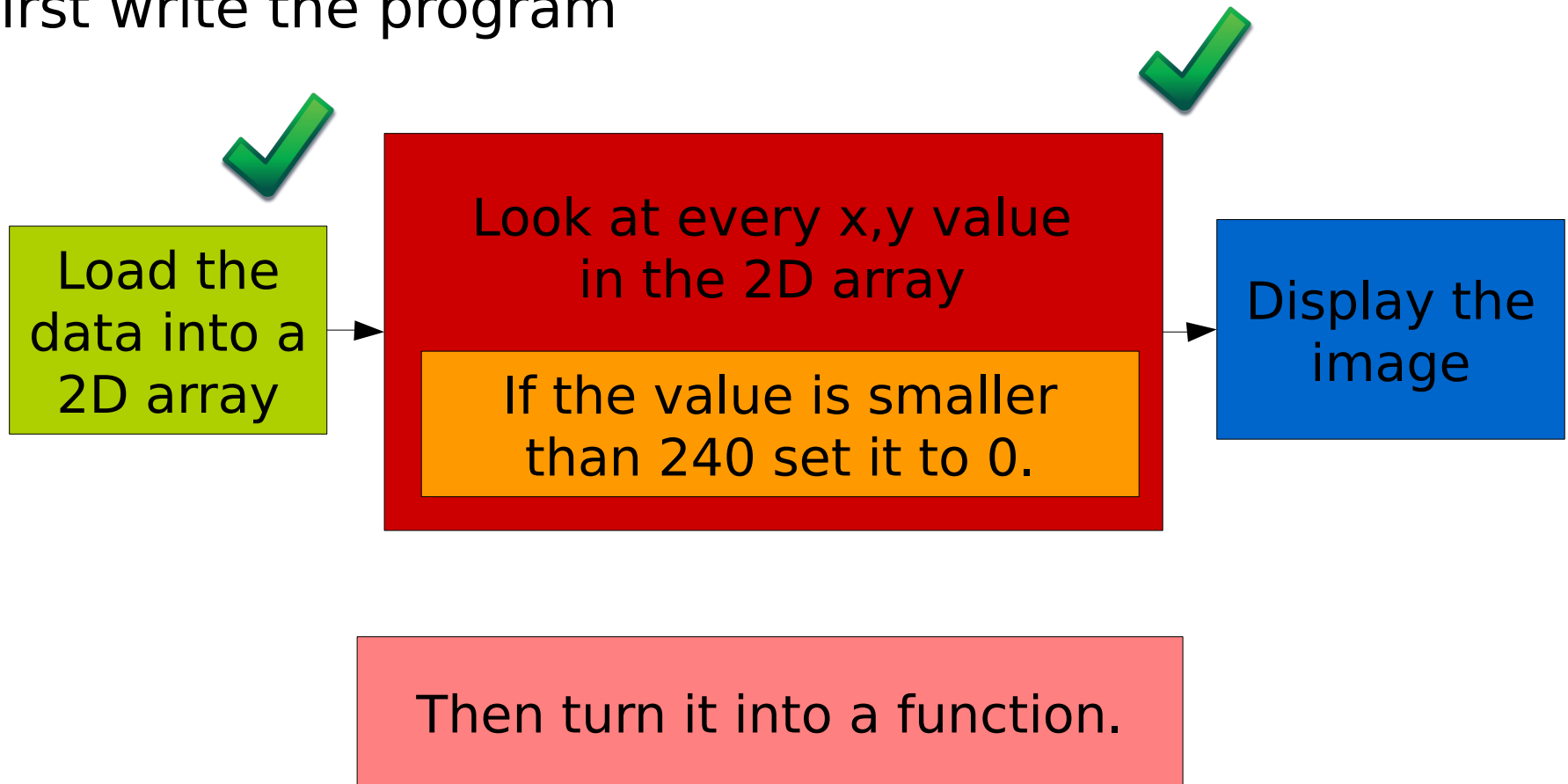
- To check the value of each pixel in the image we need two loops – one looping over x coordinates and one looping over y coordinates.

```
data = imread('stars.jpg');  
  
len=size(data)  
xlen=len(1)  
ylen=len(2)  
  
for x=1:xlen  
    for y=1:ylen  
        disp(data(y,x))  
    end  
end
```

- Instead of writing the `if` statement to check for the brightness of the pixel straight away, I am just displaying it's value.

# Strategy for doing this task:

First write the program



Often it is good to break down computing tasks like this

# Navigation by the stars

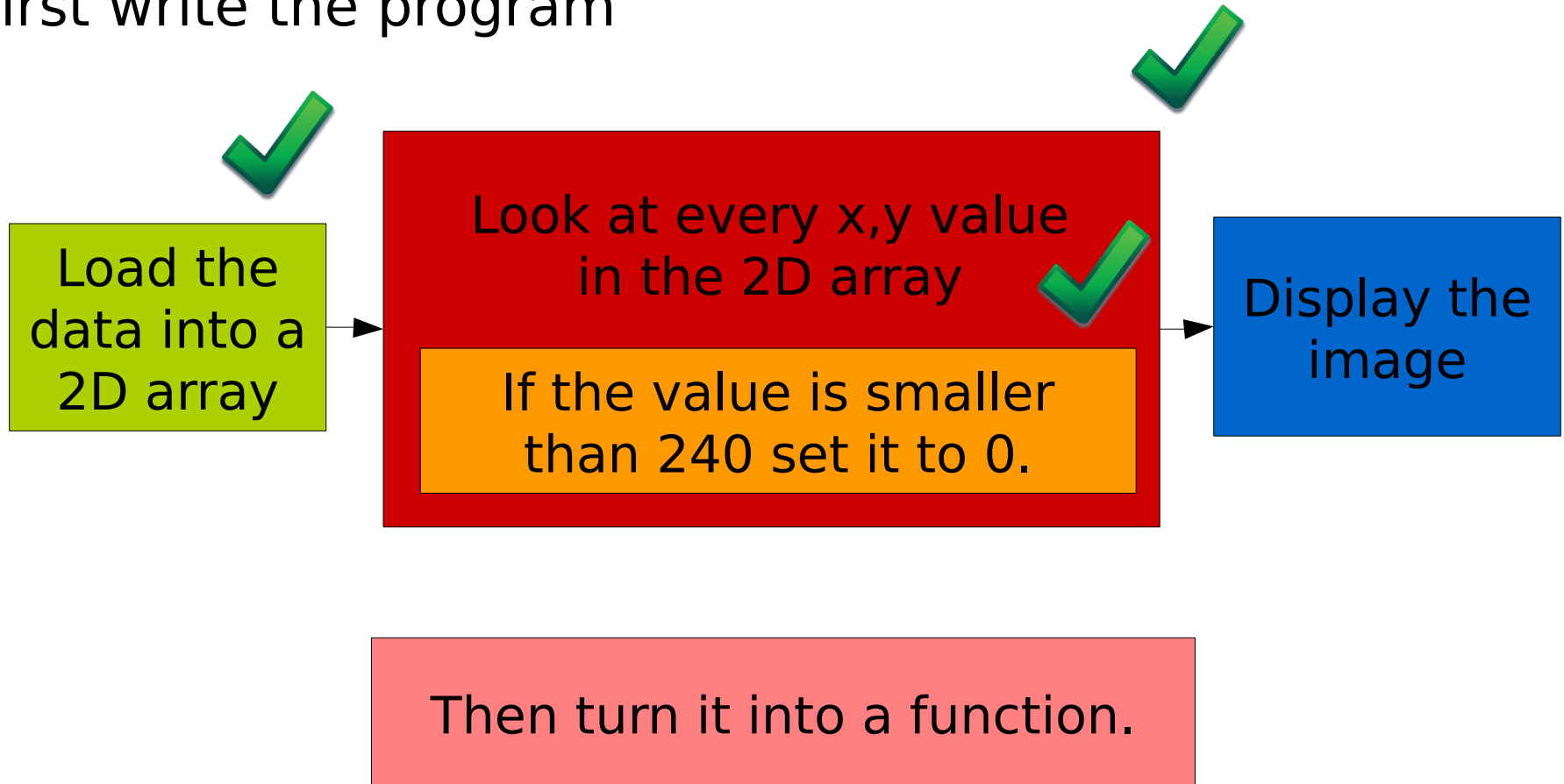
Now we know the loops are working, let's add the `if` statement. To set any dim data points to zero.

```
data=imread('stars.dat')
len=size(dat)
xlen=len(1)
ylen=len(2)

for x=1:xlen
    for y=1:ylen
        if (data(y,x)<240);
            data(y,x)=0;
        end
    end
end
```

# Strategy for doing this task:

First write the program



Often it is good to break down computing tasks like this

# Navigation by the stars

Now display the image:

```
data=imread('stars.dat')
len=size(dat)
xlen=len(1)
ylen=len(2)

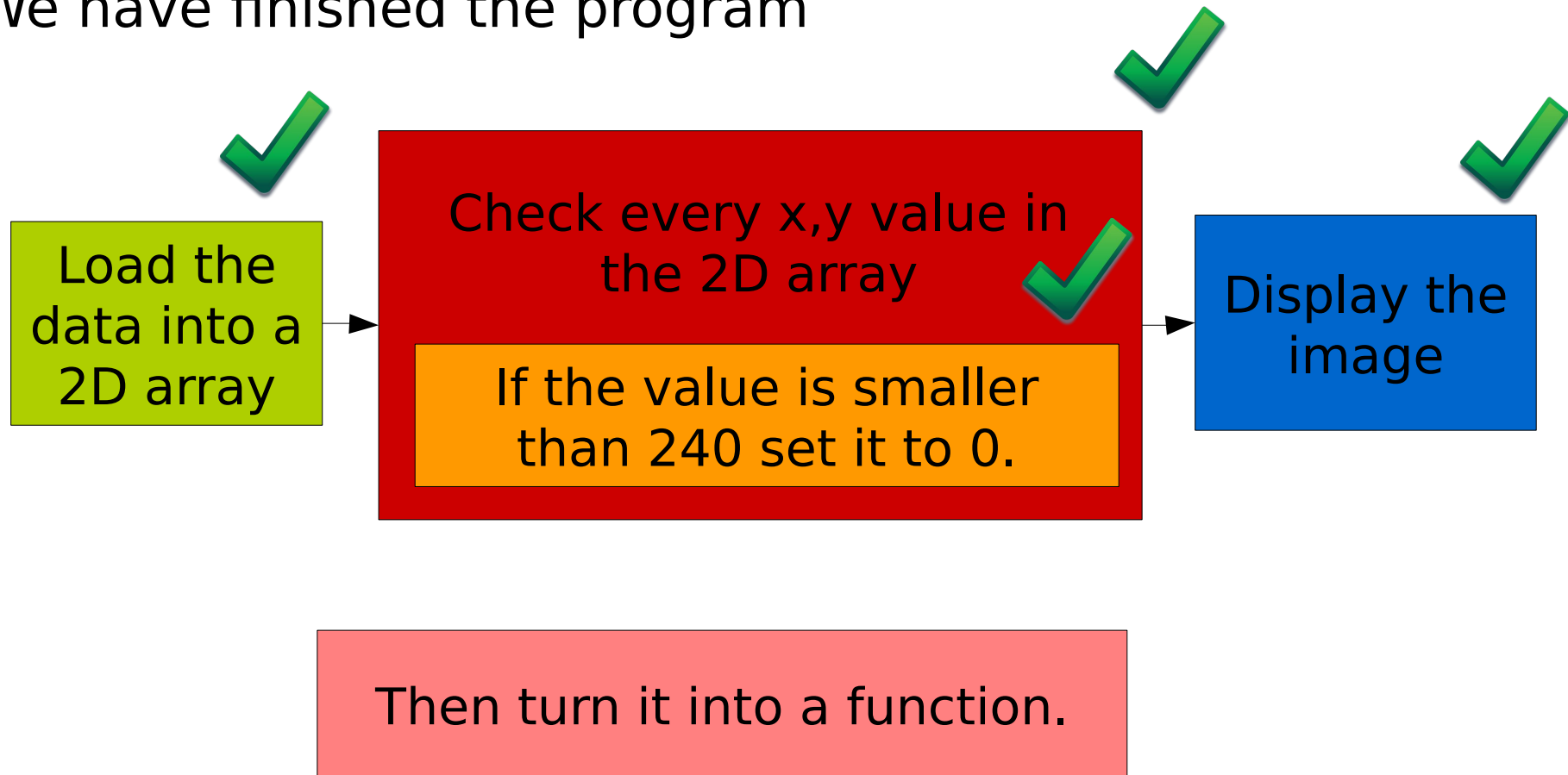
for x=1:xlen
    for y=1:ylen
        if (data(x,y)<240);
            data(x,y)=0;
        end
    end
end

imshow(data)
```

- `imshow` is a new command, it's just like `surf` but only gives a 2D image.
- Good for images.

# Strategy for doing this task:

We have finished the program



Let's turn it into a function....

# Turning a program into a function 'stars'

## Original program

```
data=imread('stars.dat')
len=size(data)
xlen=len(1)
ylen=len(2)

for x=1:xlen
    for y=1:ylen
        if (data(x,y)<240);
            data(x,y)=0;
        end
    end
end
```

```
imshow(data)
```

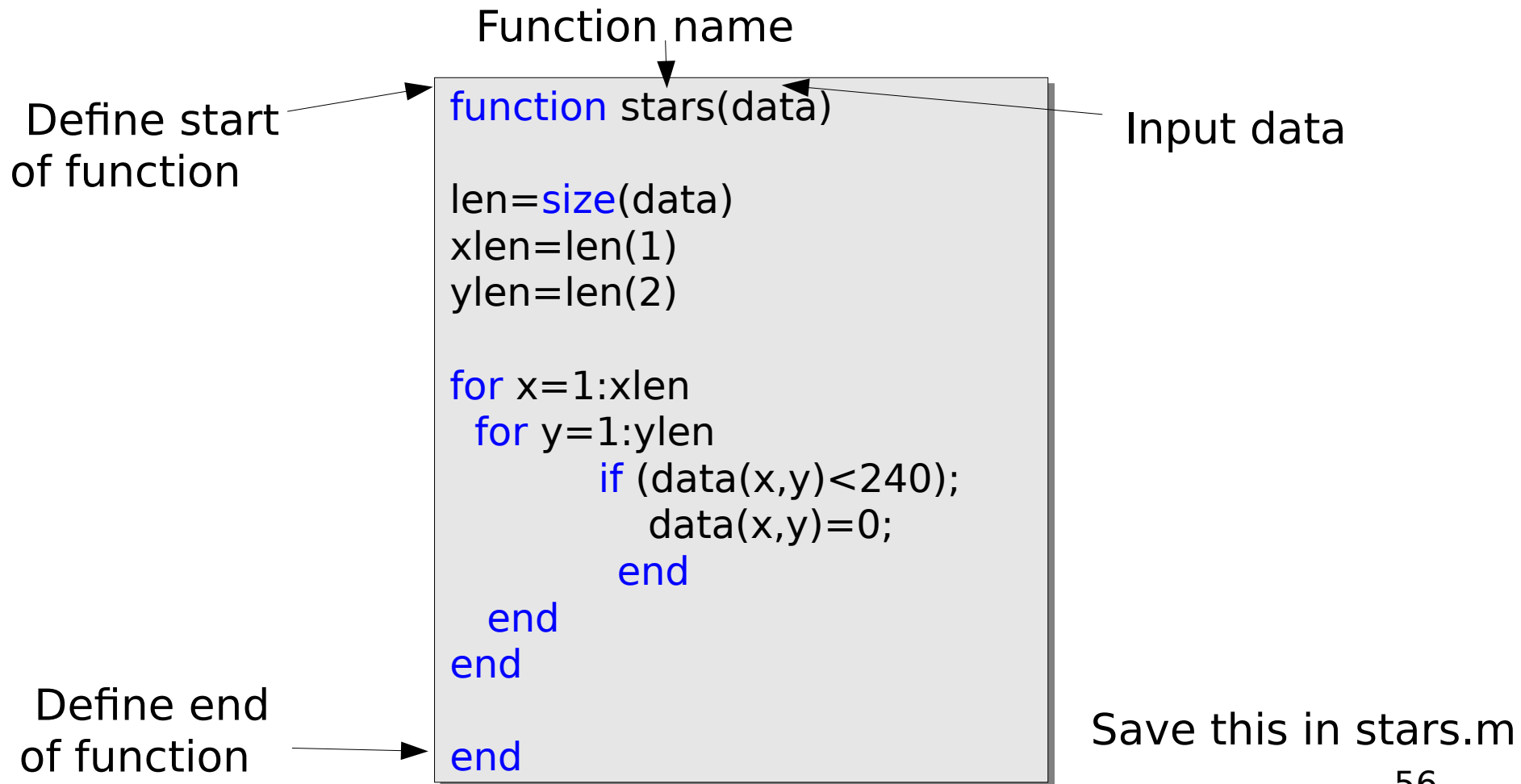
## New program calling our function

```
data=imread('stars.dat')
ret=stars(data)
imshow(ret)
```

## Now let's write the function

## Define the function 'stars'

Copy and paste the main body of our program between the words function and end in a new file.





# Getting data out of 'stars'

One last trick: We can't actually modify variables passed to functions so we need to make a copy of it and return it back to the main code.

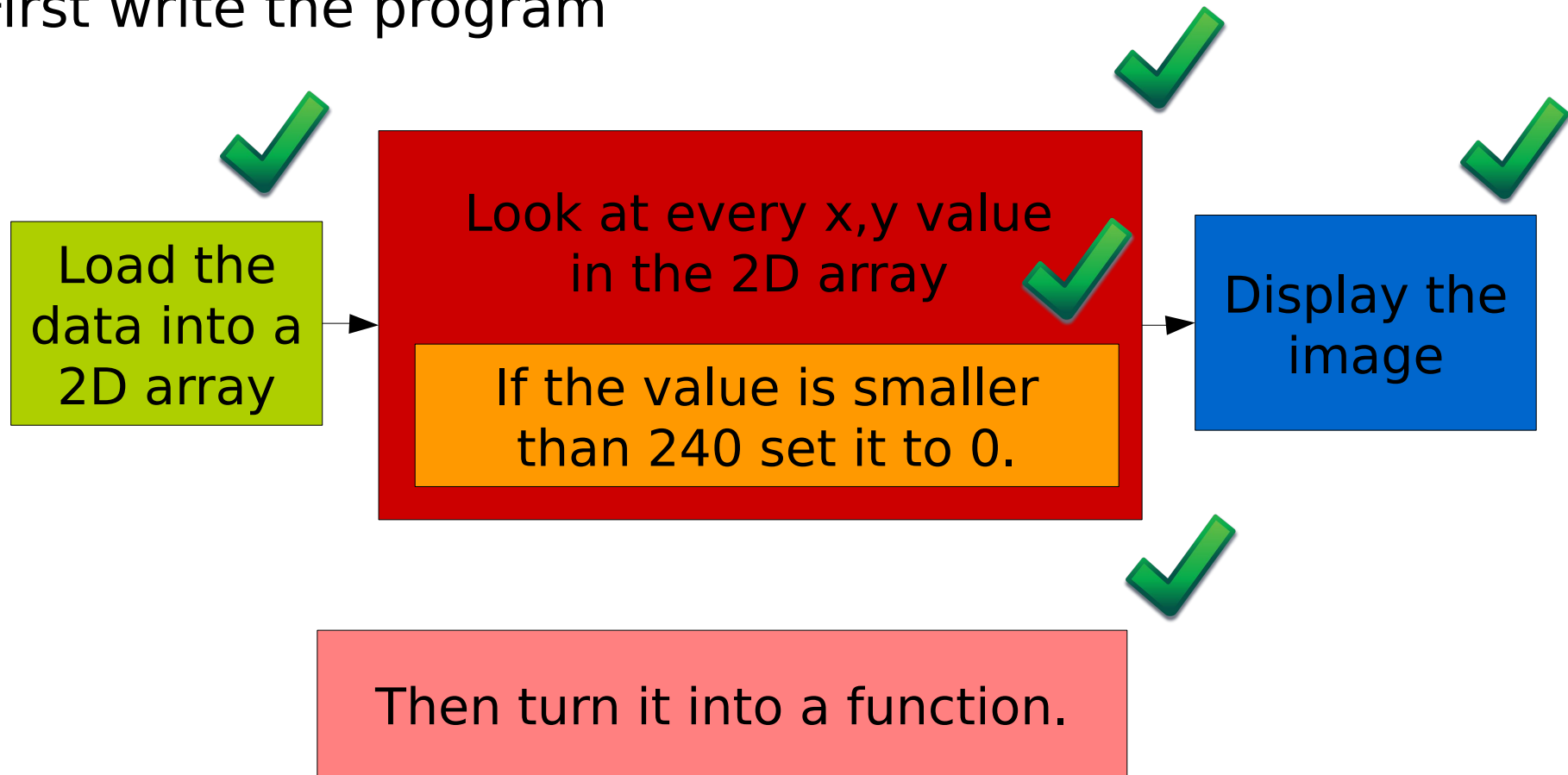
We work with the return variable so the modified data is given back to the user.

```
Return variable
↓
function out=stars(data)
out=data;
← len=size(data)
xlen=len(1)
ylen=len(2)

for x=1:xlen
  for y=1:ylen
    if (out(x,y)<240);
      → out(x,y)=0;
    end
  end
end
end
```

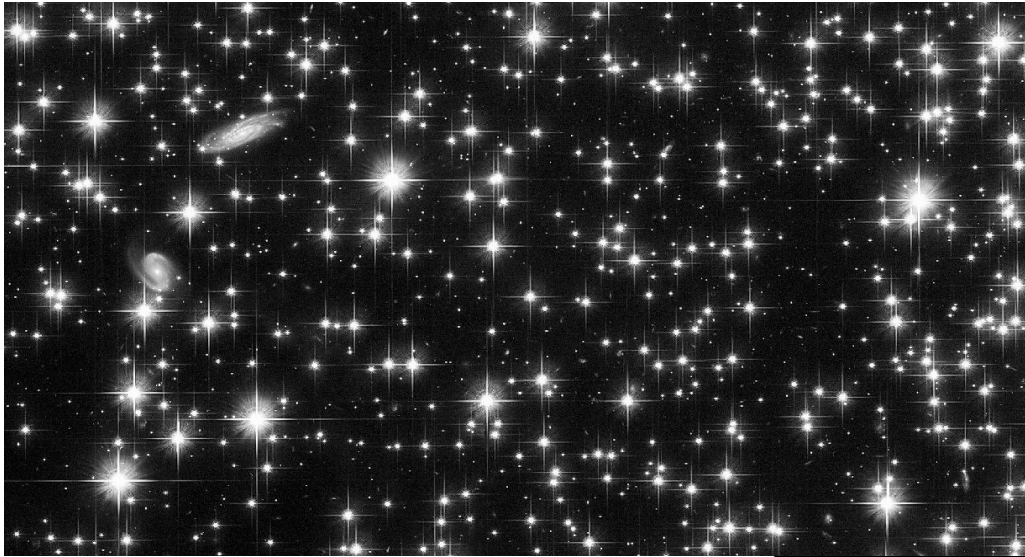
# Strategy for doing this task:

First write the program

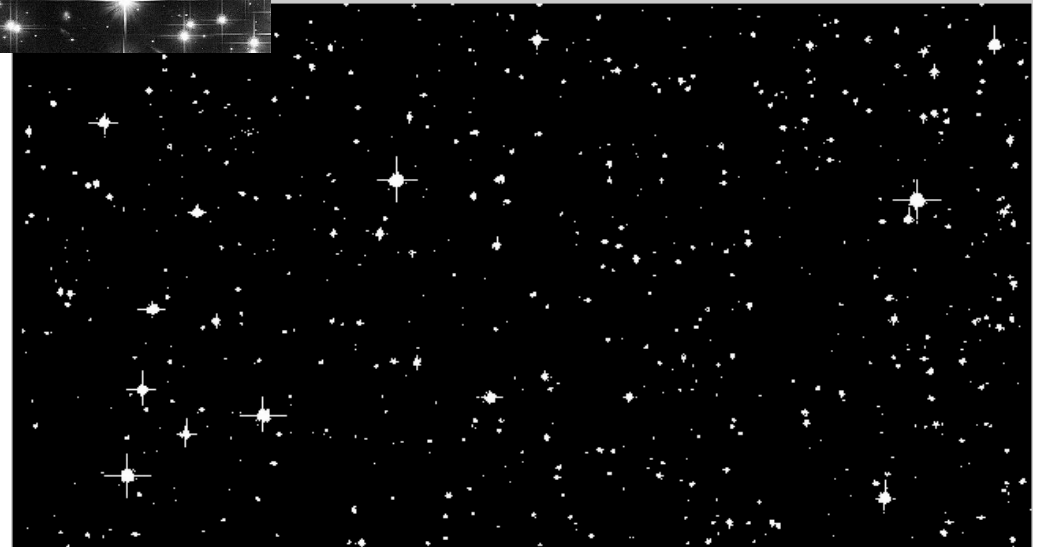


Often it is good to break down computing tasks like this

# The result:



Lots of the dim stars  
have been removed,  
and also the galaxies



# Summary

---

- Recap of last lecture on algorithms
- Parallel computing
- Reusable code
  - Functions
- A real world example of functions