* Functions revisited
* Potpourri
* write efficient programs
* object oriented programming
* Applications
* project

'Mastering Programming with MATLAB'

Functions - Revisited

Recursion: ⟶ Some problems - easily solved. - Function call itself.

Part 1: $0! = 1$
Part 2: $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$

Recursive definition

* $0! = 1$
* $n! = n * (n-1)!$
  $= n * (n-1) * (n-2) \cdots 1$

(For any +ve integer n)

'Refers itself'

Recursive - adjective
Recursion - Noun.

$12 * 11!$
↳ dead end

$12 * 11 * \cdots * 0!$

$3! = 3 * 2!$
$= 3 * 2 * 1!$
$= 3 * 2 * 1 * 0!$
$= 6$

* 'n! → not refers to n but n-1 's factorial'
* n-1 closer to 0 than n
* (Ram painted Ram's painting) ⟶ whole Ram (Circular reason)
* 0! → end case
* Recursion - more like spiral (ends at a point)

* Recursion Complicated way to do a simple thing.

```
If n==0
    f = 1;
else
    call itself
end
```

'Recursive function call'

* 'Local variables - stored in stack' - when function call over - flushed (gone)

* Stack - top accessible

*  → Function call
(Bottom is accessible - only when top is removed)
(returns - off the stack)

* function called - new area allocated (Local variables) - Stack frame
function return -

* Stack frame - dish of variables

 → zonk (Active function) - Function activation - function is activated.
→ my_funct

* Instead of putting variables - in frames - it has pointers to variables
* 'Bottom can be only accessible only when top becomes inaccessible'
Inactive

---

* 1 function paused - another top dish active
* once returned - plate gone - function resumes

'one function active at a time'

---

* Restriction: It can't access the frame of the function that is called my_funct (Inactive disc) - frame

---

* Even though zonk & my_funct both have a variable name x
'there is no confusion' - different entirely.

---

when job done - kicked out - Inactive frame - (return removed)

↓

* when zonk gone, myfunc continues where is it left.

Note: matlab doesn't copies to workspace (Command prompt)

* MATLAB copies i/p values in the i/p arguments & copies o/p argument from o/p variable values - when return?

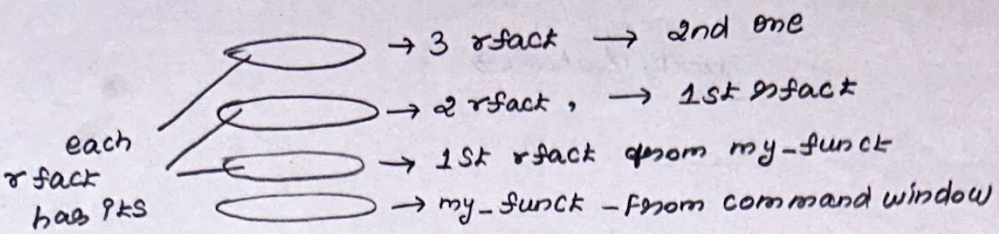
↓

Locally doing. → 'Books on Compilers'

'Stack limits an active functions access only its outer frame.

'No confusion - b/w two frames'

* Allocates new stack frame - even it calls itself.



* with 3 → 4 rfact frames preserved.

* 4th one active - all 3 paused.

* 4th one done - popped up - 3rd one will be active

```
function f = rfact(n)
    if n == 0
        f = 1;
    else
        f = n * rfact(n-1);
    end
end
```

∴ for -ve numbers / float → never be 0 → ∞ loop

* lot of frames - MATLAB warns ! (∞ loop)

* while (1) → doesn't create frames - so no worries about memory.

~isscalar(n) → non scale

n < 0 → negative

fix(n) ~= n → non integer

↓

call it error

* **Base case**
  n = 0

* f = n * rfact(n-1)

↓

Recursive case.

Atleast one base & recursive case will be there.

>> rfact(171)

Inf → out of MATLAB capability

>> rfact(69898)

Inf

>> rfact(69899)

out of memory.

↓

Not ∞ recursion - but too much empty.

Matlab stack is big but not enough to hold 69899 frames - 69899th one is active.

Handy features

★ Conditional breakpoint: Enters a condition that will be checked when ever code reaches that line

red - stop (break point)
Yellow - check for cross traffic - Stop down

Conditional breakpoint: stop only when the condition is true.

↓

eg: Conditional breakdown: n == 0

↓

When code runs

>> fact (4)

at that line of code

K >>

link to that line

'execution doesn't stop as till n == 0'

Look at stack

Function call stack: → In menu

fact

fact
fact
fact
fact
fact
Base → Command window.

Stack

★ 'we can see each frames variables - in workspace - by activating it'

green arrow → Next Codes starting point.
(Line - executed next)

white arrow → Next we will be there

Overcase

f = n * fact(n-1)

↓

recursive call

(Green arrow)

"use breakpoints - Analyze stack frame" - see local variables

what about f's ? → f is not assigned until the process is completed (or) 0! is reached

Then only assigned. ← f=1 when it 0!

"Not yet assigned" - So not there. (In workspace)

* Step in: execution stops at that function call: execute line by line that call

* Stepout: normal process (step: next call).

```
function f = fact(n)
    if ~isscalar(n) || n ~= fix(n) || n < 0
        error('non-negative integer scalar input expected');
    end
    f = 1;
    for i = 1:n                    → Non recursive version
        f = f * i;
    end
end
```

↓

∴ doesn't run out of stack space.

Recursive vs Iterative

* Every recursive implementation has iterative version
* Iterative: often than recursive (no frames)

"often easier to implement" - simplicity

Sierpinski triangle. - overall equilateral Δle within - within equilateral Δs.

https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle

"Subdivided into smaller & smaller Δs"

"Fractal" - Zoom in - see same pattern over & over

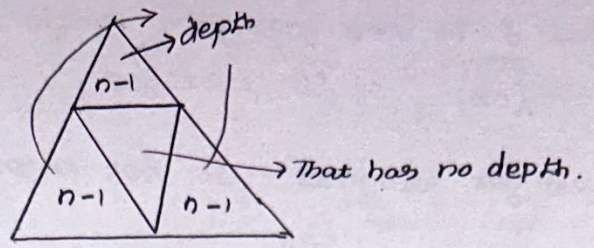
Base case:

= 1 equilateral → depth 0 △

Recursive case

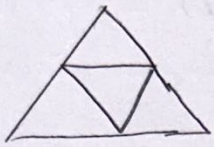
Sierpinski Δle of depth n = three Sierpinski Δs, each

depth n-1, arranged.



Zero depth

depth = 1



>> cls → clean figure.

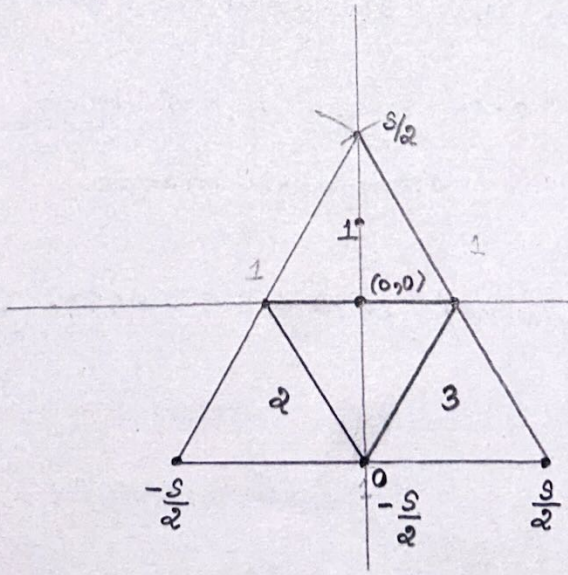
>> axis([x min, x max, y min, y max])

axis EQUAL - sets the aspect ratio so that equal tick mark increments on the x-, y- & z- axis are equal in size.

d - depth

S - size eg one side eg Δ^{le}

c - position of centers.



x points

① $\left[\frac{+\frac{S}{2}}{2}, 0, \frac{-\frac{S}{2}}{2}, \frac{\frac{S}{2}}{2}\right]$,

Centers

① $(0, \frac{S}{4})$,

② $(-\frac{S}{4}, -\frac{S}{4})$

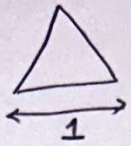
③ $(\frac{S}{4}, -\frac{S}{4})$

height of equilateral $\Delta^{le} = h = \frac{\sqrt{3}}{2} a$

Plot a equilateral Δ^{le}

>>plot ([1, 0, -1, 1], [0, 1, 0, 0])

why need s →?



,

'So we need a s'

"Huge job to realise using for loop" → 1 triangle, inside 3 ↓ inside 3

* no. of for loop need ↑ every depth ↑.
* Recursive: The choice comes handy.

Base case:

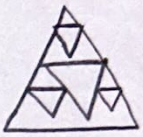
depth case: Plot

Recursive case:

go deeper.

Need

* center (for identify 3 Δ^{les} pos.)
* length of the graph (1 units)



1) depth 2

depth 1 1 1

0 0 0

3 Δ^{les}

6 Δ^{les}

Done.

Procedure - draw smallest Δ^{le}

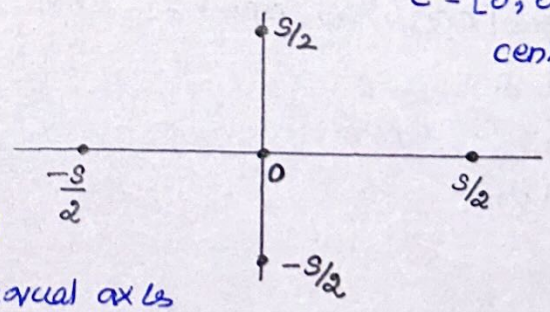
axis: we are always going to have axis → size = 1

"modifiable"

s = 1

c = [0; 0]

center

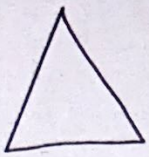


axis ([xmin, xmax, ymin, ymax], 'equal')

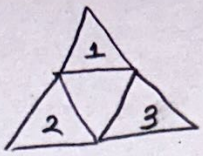
↳ equal axis

call function for plotting

Depth 0

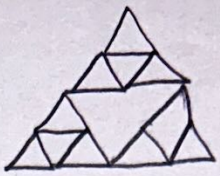


Depth 1



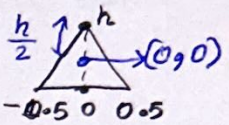
3 triangles

Depth 2



'9 triangles'

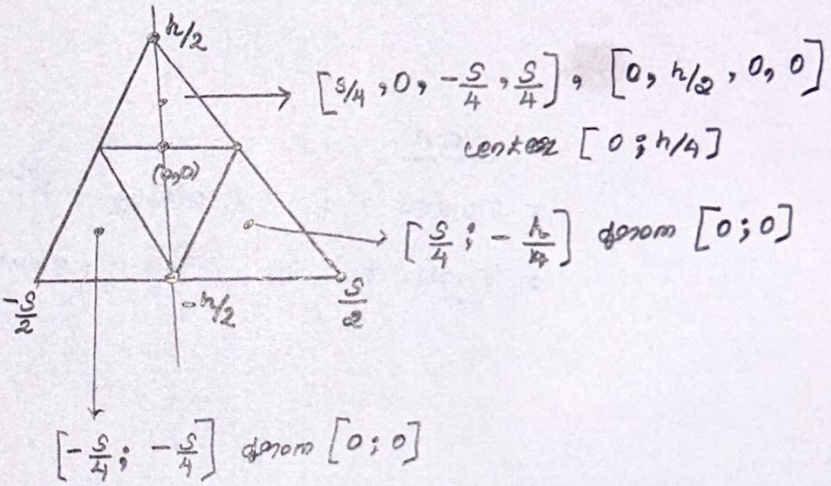
hold on & hold off → ∴ we are going to plot multiple Δ's.



$\frac{h}{2} = side * \frac{\sqrt{3}}{2} = s * \frac{\sqrt{3}}{4}$

0 → plot (cc(1) + [s 0 -s s]/2, c(2) + [-s s -s -s] * (sqrt(3)/4));

else : 3 Δles (each go to depth 0)




```
function siepin SHE (depth)
s = 1; c = [0; 0]; clf
axis ([c(1) - s/2, c(1) + s/2, c(2) - s/2, c(2) + s/2], 'equal');
hold on
siep (s, c, depth);
hold off
end

function siep (s, c, depth)
if depth == 0
    plot (cc(1) + [s, 0, -s, s]/2, c(2) + [-s, s, -s, -s] * (sqrt(3)/4));
else
    s = s/2; h = s * (sqrt(3)/2);
    siep (s, c + [s; -h]/2, depth - 1);
```

```
sieve(S, C + [-S; -h]/2, depth-1);
sieve(S, C + [0; h]/2, depth-1);
end
end
```

Secret: It contains only smallest Δles : while seeing : makes sense.

add

plot($\frac{1}{2}$ * [-1, 0, 1, -1], sqrt(3)/4 * [-1, 1, -1, -1], "r--");

↓

outline (add in sierpinski before sieve( )

↓

Debug: see line by line what happens

Animation : (2/3) seconds

pause(2/3)

depth : 3 : 4 calls → 1 – depth 4 gets 3 (from sierpinski)
2 – depth 3 gets 2
2 gets 1
1 gets 0 (plotted)

(each Δle is pointed in the 4th cycle)

Depth 4 : Grand total : 121 calls.

CPU time → from start to running the function.

```
t0 = cputime → upto running function
>> sierpinski(8);
>> cputime - t0 → function runtime.
```

$3^0 + 3 + 3^2 + \cdots + 3^8$ → sieve was called. (8 – depth

$3^0 + 3 + 3^2 + 3^3 + 3^4 = 121$ times

$\sum_{n=0}^{d} 3^n$ → no. of Δles drawn

↓

3.^[0:8]

depth = 8

$$\sum_{n=0}^{d} 3^n = (3^{d+1} - 1)/2$$

9841 Δ^{les} → depth = 8

deepest level: $3^8 = 6561$ – tiny Δ^{les}

9841 calls → 9841 frames are added to each level. → To stack

Analyze

depth 8

Factorial

100! → only after 100+ calls

① Sleep (7, .., .)
② Sleep (6, ..)
③ ⋮
④
⑤
⑥
⑦
⑧

"Total frames = depth + 1" = 9
↓
Initial call

After reaching depth 0 → closes the frame
Then return to next call of
⑧ Call — and sleep.

"maximum no. of active frames will always be depth + 1".

↳ No stack – overflow
"Max: depth + 1" → But takes time.
(As depth ↑)

Rfact & Sleep

* Rfact → 1 recursive case
* Sleep – executes 3 recursive case

"Level of recursion ↑ by 1 as case ↑ by 1"

"Sleep – Triples as Rfact"

Here: no. of calls ↑ linearly with the depth of recursion. → Rfact

10! → 10+ calls
100! → 100+ calls.

But in Sfern: the no. of calls increases recursively.
(exponential: Not by fixed amount - but by a fixed factor).

* Each Sfern will use all the 3 steps - exponentially ↑.

'Exponential growth - common in recursive cases'

```
>> sfernpinski(1); tic, for d = 1:9; to = cputime; sfernpinski(d);
                                    t(d) = cputime - to; end
```

↳ 'one' sample (throwaway call) → 'MATLAB' - does some setup on the first call on a sequence of calls of the same function.

sfernpinski(12) → 507.7800 (8 minutes +)

↳ 'Not plotted'

'Problem: with MATLAB: graphics system' - not designed to handle this many plot calls in one application.

12th depth → 531441 Δ'les → Too many.

display system - tried to process - plotting info - crashed.

Solution

depth 9 → 19seconds
depth 12 → 507 seconds
} CPU time ↑ by the factor of 3.

E → efficient

Solution

* don't plot in each recursive or base case
* create a list - then plot a single time.

$$\therefore \begin{bmatrix} 1 \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & \cdots \\ 2 & 4 & \end{bmatrix}$$

.. ↓ then plot.

'Don't plot directly'

```
Plot(C(1)+[5  0  -5  5]/2 , C(2)+[-5  5  -5  -5] * (sqrt(3)/4));
          ↓                        ↓
          x                        y
```

let $x \rightarrow$ 1st row

$y \rightarrow$ 2nd row.

---

2 by N array.

---

when everything - plotted at once: execution & memory → saved.

↓

we don't get to watch one by one Δles (we don't need)

'Still (after plotting at once): the recursion ↑ exponentially' → at a lower time than plotting recursively.

'writing iterative version is not worth the effort'

Play

```
>> t = cputime; sleepin(8); t = cputime - t;
```

7.01 seconds

>> 'plot' - time (vs) depth.

```
>> sleepin(1);
```
→ ∴ For better accuracy (As as best)

```
for n = 1:9
    t = CPUtime;
    Sleepin(n);
    t1(n) = CPUtime - t;
end
```

Plot(1:9, t1) → exponential graph.

Row 1 → Rows

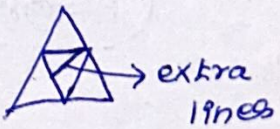
Row 2 → columns

1	2	3	4	5	7
2	3	4	5	6	8

Plot

↓

will be continuous with the above points



'No need'

↓

Use man.

c - [s/2, -s/2; s/2, -s/2] → error ∴ C = [0; 0]
2×1 2×2
Not possible

* But scalar subtraction is possible.

---

pts = sier(s, c, depth, [])
↳ current pts list (o/p: updated list)

pts = [pts, xpoints & ypoints] → append

c - [s, -s; s, -s]/2 → -0.5 0.5
-0.5 0.5
2×2
(not able to do)

```
function sierpinski(depth)
if (depth<0 || ~isscalar(depth) || fix(depth)~=depth)
    error('depth must be a +ve scalar integer');
end
s=1; c=[0;0]
axis([c(1)-[s,-s]/2, c(2)-[s,-s]/2], 'equal');
pts=sier(s,c,depth,[]);
plot(pts(1,:), pts(2,:));
end
```

scale: 1 unit; (0,0) centre

```
function pts=sier(s,c,depth,pts)
if (depth==0)
    pts=[pts, c+[[s,0,-s,s,nan]/2; [-s,s,-s,-s,nan]*sqrt(3)/4]];
else
    s=s/2; h=s*(sqrt(3)/2);
    pts=sier(s, c+[s;-h]/2, depth-1, pts);
    pts=sier(s, c+[-s;-h]/2, depth-1, pts);
    pts=sier(s, c+[0;h]/2, depth-1, pts);
end
end
```

→ Add a point to pts (4 points) ↓ A △le + ran value (continuation)

3 sub △les

Recursion part-3

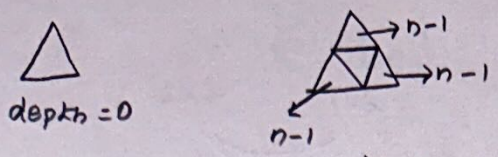
Factorial

Base case: 0!

Rec case: n*(n-1)!

Base case = depth 0 - 1 equi △le

Re case:
3 sierpinski △les,
each @ depth (n-1)



Greatest Common divisor

Euclid (≈300 BCE) – Algorithm

"Two numbers have may be more than one common divisor" – what's the greatest one.

35, 14
\ /
7 (Remainder = 0)

GCD(35, 14) = 7.
GCD(12, 4) = 4
GCD(6, 0) = 6

Algorithm:

Start from 1 to min(a, b) → largest one = GCD. [Common]

efficient algorithm – based on observation.

* $GCD(a, b) = GCD(b, rem(a, b))$ for $a \geq b$

(remainder of $a \div b$)

The GCD of a, b is the same as the GCD of b, remainder $a \div b$, $(a \geq b)$

'a & b' has a common divisor d.

$a = Pd$, $b = Qd$

$\therefore gcd(a, b) = gcd(rem(a, b), b)$

$gcd(a, b) = gcd(a - b, b)$

$a \geq b$

$\therefore a = Pd$, $b = Qd$

$a - b = (P - Q)d - Qd$

$\therefore$ ↓

d is the CD of R.HS

d is the CD of L.H.S

$\therefore$ gcd is equal.

$$b = 3 \overline{)\, 7 = a}\ \ q = x$$
$$\underline{6}$$
$$\underline{1}$$

$a = bx + r$

$r = a - bx$

↓

quotient.

$\therefore a = Pd$, $b = Qd$

r also has a CD 'd'

$gcd(a,b) = gcd(b, r_1)$

$gcd(b, r_1) = gcd(r_1, r_2)$

base case: $gcd(a,0) = a$

$a \geq b$

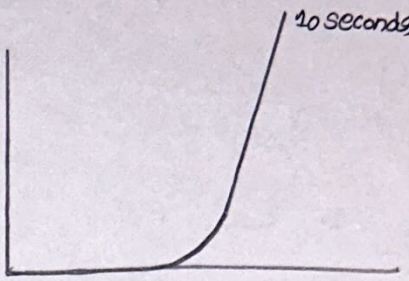
$$11 \overline{)100} \quad 9, \ 99, \ 1$$

'r < b'

gcd (a,b) → gcd (2,0) = 1
8,2

8,2 → 2,0 → 2

35,14 → 14,7 → 7,0 → 7



efficient?
↓
Sleepinski

```
function d=mygcd(x,y)
a=max([x,y])
b=min([x,y])
if b==0
    d=0
else
    d=mygcd(b,mem(a,b))
```

Iterative

```
while b~=0
    r=mem(a,b);
    a=b;
    b=r;
end
d=a;
```

---

while b~=0

12345 → rem(12345, 10) = 5

rem(1234, 10) = 4

fix(1234/10) = 123

fix(123/10) = 12.

1. solution

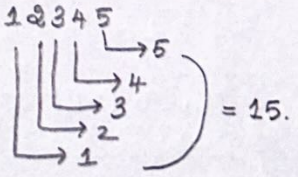
12345 → 15

```
function output=digit_sum(n,output)
if nargin==1
    output=0;
end
if n==0
    return;
else
    output=output+rem(n,10);
    n=fix(n/10);
    output=digit_sum(n,output);
end
end
```

other sum

less than ten → 1 digit → Base case

my requirement



① 5, 4, 3, 2, 1

5 + (4 + (3 + (2 + 1)))

1 roll 2 3 4 5

roll

```
function output = digit_sum(n)
if n==0
    output =0;
else
    output = digit_sum(fix(n/10)) +
                                rem(n,10)
end
end
```

1. ___ + 5
2. ___ + 4
3. ___ + 3
4. ___ + 2
5. 0 + 1
6. 1 + 2 = 3
7. 3 + 3 = 6
8. 6 + 4 = 10
9. 10 + 5 = 15

```
function output = digit_sum(n)
if n< 10
    output =n;
else
    output = digit_sum(fix(n/10)) +rem(n,10)
end
end
```

1. ___ +5
2. ___ +4
3. ___ +3
4. ___ +2
5. 1 (< 10)
6. 1 + 2 = 3
7. 3 + 3 = 6
8. 6 + 4 = 10
9. 10 + 5 = 15

maximum

recursive_max.m

[1 2 3 4 5]

len = length(v) - 1

max = v(len+1)

① 5 > 4
② 5 > 3
③ 5 > 2
④ 5 > 1

calls.

⑤ ∴ len = 0 (return)

recursive_max_1.m

[1 6]

v(1) Compare with v(2:end)

Compare.

[1 2 3 4 5 6] → max

1 4 6 6 6 6

make length 1, make that element max compare with prev element.

```
function big= biggest(a,b)
big=b;
if a>b
   big=a;
end
end
```

```
function maxi = recursive_max(v)
if length(v)==1
      maxi=v(1)
else
    maxi = biggest(v(1), recursive_max(
                                v(2,:)));
end
end
```

1. 1, [2 3 4 5]
2. 1, (2, [3 4 5])
3. 1, (2, (3, [4, 5]))
4. 1, (2, (3, (4, [5]))) → maxi
5. 1, (2, (3, (4, 5)))
6. 1, (2, (3, 5))
7. 1, (2, 5)
8. 1, 5
9. 5

[1 2 3 4 5] ⟶ [5 4 3 2 1]

1. ([1,2,3,4,5]), v(1)
2. ([2,3,4,5], 2), 1
3. ([3,4,5], 3), 2, 1
4. ([4,5], 4), 3, 2, 1
5. 5 4 3 2 1

```
function w = reversal(v)
if length(v)==1
      w = v(1)
else
      w = [reversal(2:end), v(1)]
end
end
```

[5]
↓
[5, 4]
↓
[[5,4], 3]
↓
[[5,4,3], 2]
↓
[[5,4,3,2], 1]
↓
[5 4 3 2 1]

Fibonacci

w = fibon(4) = 1, 1, 2, 3

fibon(5) → calls fibon(4), 3,
fibon(4) → calls fibon(3)
) unnecessary "Tackle"

n = 3	n = 4	n = 6
[1, 1, 2]	[1, 1, 2, 3]	[1, 1, 2, 3, 5, 8]

```
function w = fibon(n)
if n < 3
    w = ones(1,n);
    return;
else
    w = fibon(n-1);
    w = [w, sum(w(end:-1:end-1))];
end
end.
```

[1 1 2 3 5]

n = 2

w = [1 1]

w = [1 1 2]

n = 4

w = [1 1 2 3] ← sum

n = 5

w = [1 1 2 3 5]

Logic n=3

w = fibon(n-1) → [1, 1] (fibon(2))

w = [[1,1], sum(1,1)] = [1, 1, 2]

n=4

w = fibon(n-1) → [1, 1, 2]

[1, 1, 2, 3].

Palindrome

✓ racecaar → racecaar

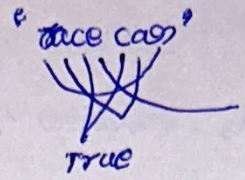
✓ live on time emit no evil → live on time emit on live

✗ live on time, emit no evil → live on time , emit on live
↓
comma (no)

✗ Live //

↓

caps.



$a = a \;\&\&\; a = a \;\&\&\; c == c \;\&\&$ (length = 1 → True)

True

True.

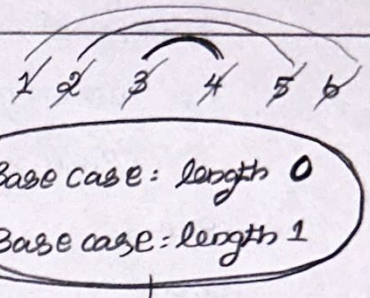
[3 4]

>> 2:end-1

↓

1x0 empty array → length 0

Base case



↓

True.

```
function palindrome (str)
if length (str) <= 1
        p = true;
else
        p = (str(1) == str(end) && palindrome (str(2:end-1)));
end
end
```

'caamaace' – No

'racecar' – Yes

1 thing not in position – No

) AND gate.

---

variable number of arguments

>> fprintf ('%d times %d equal %d \n', a, b, a*b)

2 times 3 equals 6 → Take any no. of arguments.

↓

No limit

unlimited no. of arguments

Actual arguments – from user.

```
function index = find_first (v, e)
  if nargin == 0
      error ('At least one argument is required')
  else if nargin == 1
      e = 0;
  end
  index = 0 ——————————————→ forbidden index in MATLAB
  indices = find (v == e);
  if ~isempty (indices)
      index = indices (1);
  end
end.
```

find ( ) ↴

returns index & value.

```
>> rng(0); w = randi ([-3  3], 1, 12)
```

↓ −3 to 3 ; 1 × Column.

```
>> find(w == 9)

S =

  1×0 empty array

        ↓

      empty
```

```
>> w = [1 1 2 3 3 4 5 6 5 6];
>> find (w == 1)
1   2
>> find (w == 3)
4   5
>> find (w == 6)
8   10
```

```
>> find_first (W)
        ↓
  e = 0 (default)
```

```
>> find_first (w, 10)

0
```

'find_first.m'

Infinite i/p function sum (a, b)

formal arguments

'we can't type 1000's of arguments (formal)'

There has to be other way!

'varargin' - variable i/p argument - cell vector.

* varargin - variable i/p argument list - special argument
* cell vector. (∞ arguments - i/p)

```
function print_all(varargin)
    for ii = 1:nargin ----------> 'no. of arguments' - i/p
        fprintf('Here is i/p argument number %d\n', ii, varargin{ii})
    end                                                    cell vector.
end
```

```
» print_all(3)
Here is an input argument number 1:3

» print_all(7,-3)
"                "          1:7
"                "          2:-3
```

---

varargin → cell argument → {}

() → provide a cell argument - which can't be provided.

---

* varargin doesn't have to be the only i/p argument.
* the signature can include any no. of arguments.
* But only one varargin alone → varargin is a cache which stores all remaining actual i/p s which are beyond the ones explicitly listed listed as formal arguments.

---

sprintf → put it in a string & returns as o/p.

[1 2 3 4 5 6 7] (∞)

print_all(1:7)

∴ nargin = 1

Input no 1: 1
" 2: 3
" 4: 5
" 6: 7

list elements.

'our printf' → takes only numerical values.

(simple).

'don't featured' - v.5 → Some precision / anything [our version]

```
function out = print_num(format, varargin)
out = '';
argindex = 1
skip = false;
for i = 1:length(format)
    if skip
        skip = false;
    else
        if format(i) ~= '%'
            out(end+1) = format(i)
        else
            if i+1 > length(format)
                break;
            end
            if format(i+1) == '%'
                out(end+1) = '%'
            else
                if argindex >= nargin
                    error('not enough i/p arguments');
                end
                out = [out numstr(varargin{argindex}, format(i:i+1))];
                argindex = argindex + 1;
            end
            skip = true
        end
    end
end
end
```

(the first three positive integers are %.d %.d %.d , 1,2,3)

If 3 format spec, 4 i/p arg → No problem

3 format spec, 2 i/p arg → error.

3 format specifiers

3 i/p argument

op → 'the first three positive integers are 1, 2 and 3'

sp → ('the first three positive integers are %d, %d, and %d', 1, 2, 3)

%d → Substitute next argument

%% → put a % (skip next loop)

* If argument doesn't exist → error
* format done → break (leave other arguments)

'skip next loop: when you see % symbol'

* No % → print (save in /PST) as it is.

---

① skip on continue

↳ ② % on not

↳ ③ → exceed on not (length)

→ % on not (if yes: print (next) skip next.

↳ ④ if not → take arg & sub then skip next loop.

Note: 'abc' + 'd' ≠ 'abcd'

↓

converted to double then will be added to 'd''s double

---

num2str(1, '%d') → '1'

'print_all1.m'

```
function varargout = inf_out(v)
for i = 1:length(v)
    varargout{i} = v(i);
end
end
```

* varargin — varargin → ∞ i/p
* varargout — varargout. → ∞ o/p

o/p

```
[a,b,c] = inf_out([1,2,3,4])
a = 1
b = 2
c = 3
```

[a,b] = inp-out (1)

'error - not enough i/p args'.

Assignment

Name-value-pairs

* 'Name value pairs' - Frequently used in programming
* name (character vectors)
* value (Any data type)

name_value_pairs.m

* name_value_pairs (variable-no. of i/p)
* come in pairs - First (name) & value
* even no. of i/p arguments
* o/p → cell array: Two columns
  - ↙ name
  - ↘ value
* If no i/p or odd i/p or non-char i/p name
  - ↓
  - { } → empty cell array.

'Data entry'

Voting center - voters → name, ID number - before voting.

* voters (record info)
* Take 2 i/p s (To current data base)

database =

1x3 Struck array with fields:

Name

ID

» name /string/char array

* Id / Integer/double.

'return - name (String), ID - double'

'Illegal call' → No change.

Struck with fields

```
>> a.name(1) = "Hari";
>> a.name(2) = "Gopal";
>> a.Id(1) = 9600;
>> a.Id(2) = 9601;
>> a
```

struck with fields

name : ["hari", "Gopal"]

Id : [1234 9600]

Struck array with fields.

'array - with all - struck elements'

```
>> a(1).name = "Hari";
>> a(2).name = "Gopal";
>> a
```

1x2 struck array with fields:

name

```
>> a(1)
```

name : 'hari' ⟶ struck.

★ even one I/P → not legal → return the database as it is

★ Think: we are going to add a new array struck element.

★ So if wrong we need to

Remove element from array.

```
>> a = [1 2 3 4 5];
>> a(1) = [ ]
```

a =

2 3 4 5

```
>> a(1).name = "Hari";
>> a(1).Id = 9600;
>> a(2).name = "Gopal";
>> a(2).Id = 9601;
>> a
```

1x2 struck array

```
>> a(1) = [ ];
>> a
```

struck with fields.

```
>> a.name
```

↓

a(1), a(2), ... all name will be these

'Take as a list' - check

```
>> ismember([a.names], 'hari')
```

0 0 0 1 0 0 0

↓

member → sum > 0

No ⟶ sum = 0

```
>> sum(ismember([a.names], 'hari'))
```

'Voters.m'

'Voters_mod.m'

my case: take all the legal names & id

Here : If any one of the name / id - Illegal - Don't update the database.

↳ 'once updated' - then remove - handjob'

```
count = length(database);
tmp = count;
```

⋮

As updated count ↑ (increment)

Any detection: illegal names

↓

database (count : end) = [ ] ⟶ the updates are gone (original one)

Functions

* handy to call a function through a variable
* Variable - identifies a function : function handle. (useful)

eg: Some functions take others functions as argument.

```
>> integral (@cos, 0, pi/4)
```

↳ function handle

↙

'uses numerical appn to estimate the ∫ of any function' supplied as an i/p arg

eg: callback functions - UI actions

'which fun should be called' - which button.

↓

function handle.

Adv cases:

* create a one-line function without a name (called appropriately - an anonymous function) ⟶ function handle.
* function handle allows function inside one M file to call a sub function inside other M file.

Uses of function handles:

* function as i/p arguments

>> integral (@cos, 0, pi/4)

ans = 0.7071

* call back functions – UI actions
* Anonymous functions
* Accessing sub functions & nested functions from outside

Nested functions: similar to sub-functions, defined inside other functions.

---

function handle

```
>> trig = @sin
>> x = trig (pi/2)
x =
    1
```

```
>> plot (trig (0 : 0.01 : 2 * pi));
```

we can assign one function handle to other

```
>> a = @sin;
>> b = a
```

function_handle with value :

@sin

> function handle copied itself

```
>> x (pi/2)
ans =
    1
```

---

pi → builtin function (Takes no i/p arg & 1 o/p arg)

```
>> mypi = @pi
>> x = mypi
x =
    function_handle with value
    @pi
```

→ Instead of value copied handle itself.

value

```
>> x()
x =
    3.1416
```

array

cell

```
xpt = [@sin @cos @plot]
```
↓
'Non-scalar'

```
xpt = [@sin @cos]
```
↓
'Non-scalar'

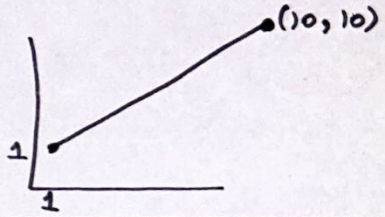
```
xpt = [@sin]
```
↓
acceptable

'use-cells'

```
>> a = {@sin, @cos, @plot};
>> a{1}(1)
0.8415
```
→ sin

```
>> a{2}(1)
0.5403
```

```
>> a{3}(1:10)
```




```
>> xpt{3}(xpt{1}(-pi:0.01:pi))
```

sin o/p

As plot's i/p

```
>> xpt{3}(xpt{1}(0:0.01:pi))
```

↓

writing code: hard to understand (dark-net purposes)

* fprintf → doesn't accept function handling.

Solution: fplot

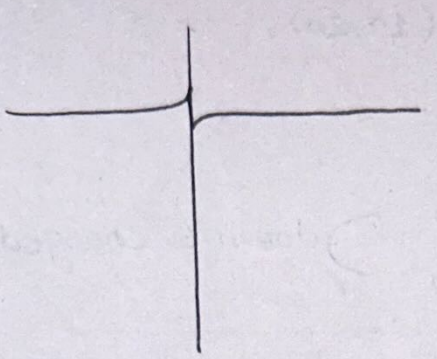
```
fplot(@sin, [0, 2*pi])
fplot(0:0.01:2*pi, sin(0:0.01:2*pi))
fplot(@tan, [0, pi])
```
↓ function handle

→ arguments (interval)

$\tan \frac{\pi}{2} = \infty$

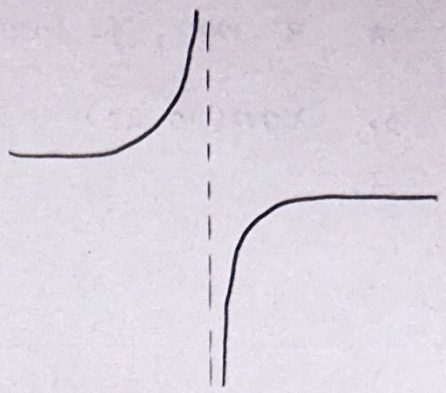
fplot (@tan, [0, pi])
↳ finteval.

plot(0:0.01: pi) tan(0:0.01:pi))



'No matter - how much resolution (point finteval) → still not better

fplot (@tan [0, pi])



'Better'

'fplot handles discontinuities very well comparing plot'

Polynomial expressions

* Creating a subfunction is awkward (overkill - for a single expression)
↳ 'meaningful name'

Anonymous function (as a single expression & returns a fun handle)

Awe: Anonymous function can't be inside a control structures (if, switch / loop) - Anywhere - in fun, script / command window.

```
» poly = @(x) 2*x.^3 - x.^2 + 2*x - 12
```

↓ Anonymous function

↓ x may be a list

```
>> poly(1)
-9
>> poly(0:5)
-12   -9   4   39   108   223
>> plot(-10:10, poly(-10:10))
>> fplot(poly, [-10, 10]) → 'Better'
```

multiple i/ps

```
>> xfn = @(x,y) x+y;
>> xfn(1,2)
ans =
     3
```

* `xfn = @(x,y) x+y;`

Scope: Local. (Inside anonymous function only)

* `x=1000; y=2000;`

`>> xfn(10,20)`

`30`

`>> xfn(10,20)`

`30`

`>> x,y`

`x=1000` `y=2000`) doesn't changed.

All the other outside variables are accessible

```
>> c = 10;
>> xfn = @(x) c * x
>> xfn(3)
30
```

∴ c is not an i/p argument

search for outer workspace for c

* Looks like c as a global preference → No.

```
>> c = 11;
>> xfn(3)
30
```

→ No change

∴ c is not a global preference.

Anonymous function

Fresh start!

```
>> a = @(x) c + x
```

error: no c is defined

* Even c is remove

```
>> clear c;
>> xfn(3)
30
```

→ works! → Still No effect on our function!

```
>> c = 1;
>> a = @(x) x + c;
>> a(1)
2
>> c = 12;
>> a(123)
124
```

→ c got fixed as 1 (No further change).

"The value 10 is fixed in the anonymous function!"

"First time"