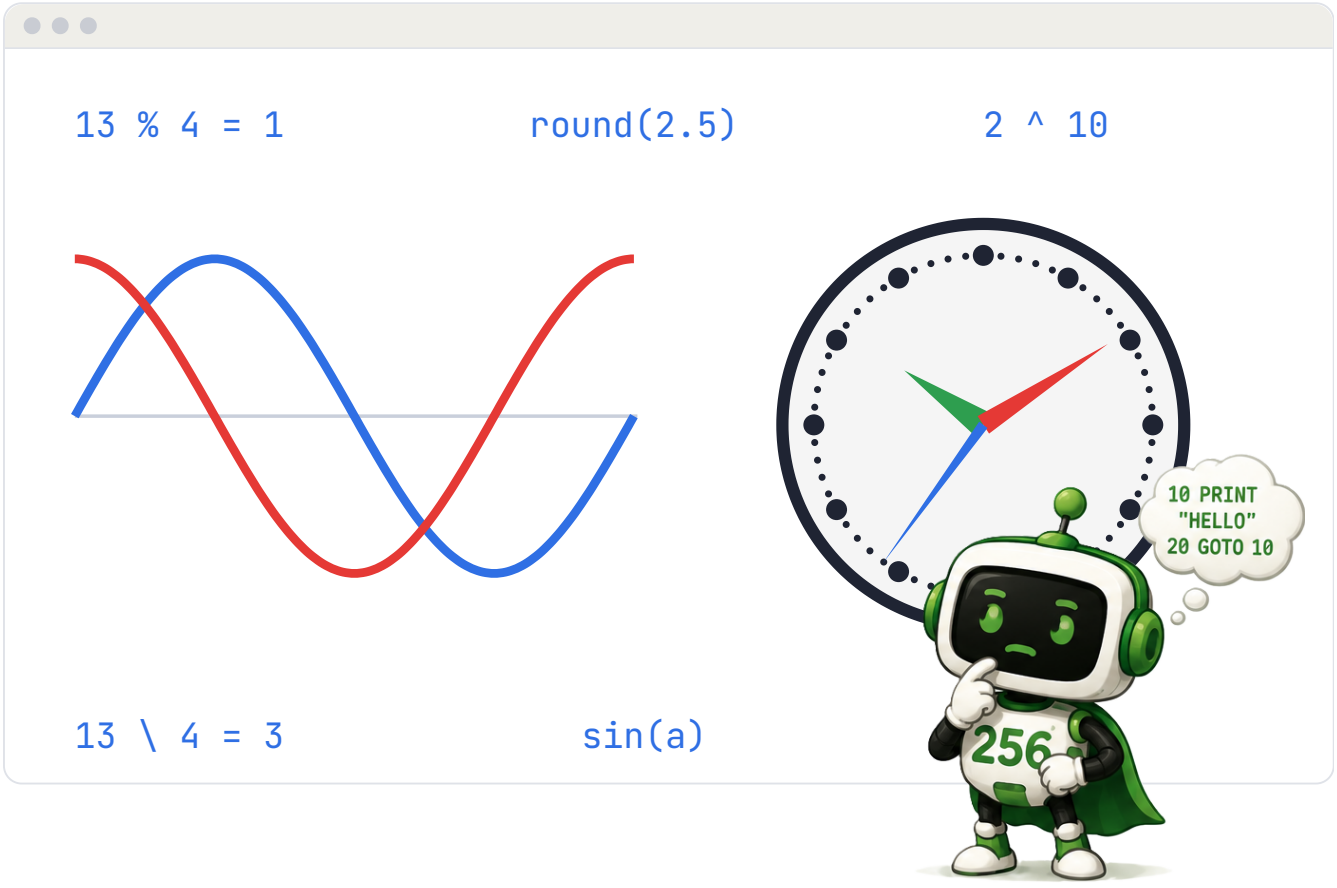
LEARN TO CODE · NUMBERS

Chapter 16

Mathematics — More Fun with Numbers

You have added, subtracted, multiplied and divided since Chapter 1. This chapter adds the rest of BASIC-256's number toolbox: three new operators (the remainder, whole-number division and powers), the ways to turn a decimal number into a whole one, square roots and logarithms, and the trigonometric functions that turn angles into positions. At the end, you use all of them in a clock with hands.



© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

Three new operators

The remainder, whole-number division and powers.

Besides `+`, `-`, `*` and `/`, BASIC-256 has three more operators for numbers. Most programming languages have the same three.

OPERATOR	NAME	EXAMPLE	WHAT IT GIVES
`% or mod`	Modulo	`13 % 4` is 1	the remainder of a whole-number division
`\`	Integer division	`13 \ 4` is 3	how many whole times one number goes into another
`^`	Power	`2 ^ 3` is 8.0	a number multiplied by itself a number of times

The modulo operator: the remainder

When you divide 13 by 4 with whole numbers, 4 goes into 13 three times, and 1 is left over: the **remainder**. The modulo operator `%` gives that remainder: `13 % 4` is 1. When the remainder is 0, the second number divides the first exactly. Program 109 uses that to tell which of the numbers 2, 3, 5 and 7 divide into yours.

modulo.kbs BASIC-256

```
# modulo.kbs - which numbers divide into yours?

inputinteger "A whole number: ", n
if n % 2 = 0 then print "Divisible by 2."
if n % 3 = 0 then print "Divisible by 3."
if n % 5 = 0 then print "Divisible by 5."
if n % 7 = 0 then print "Divisible by 7."
end
```

TEXT OUTPUT

```
A whole number: 42
Divisible by 2.
Divisible by 3.
Divisible by 7.
```

Program 109 · The Modulo Operator



NEW CONCEPT

```
a % b
a mod b
```

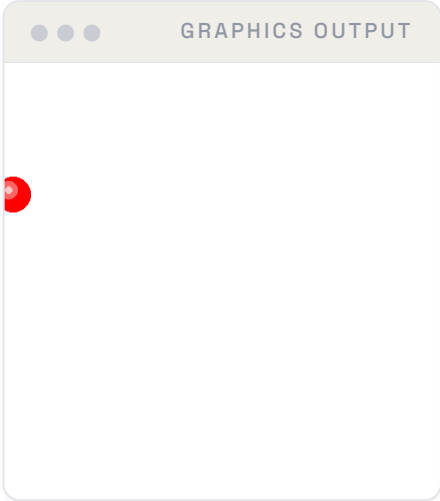
The **modulo** operator divides *a* by *b* as whole numbers and gives the remainder. You may also write it as the word **mod**: `13 mod 4` is 1. Numbers with decimals are first cut down to whole numbers, as with **int**: `7.9 % 2` is `7 % 2`, which is 1. The remainder has the sign of the first number: `-7 % 3` is −1.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

% also means percent. A `%` straight after a number with nothing behind it means percent: `50%` is 0.5. That can catch you out with a negative number: `7 % -3` is read as "7 percent, minus 3", which is −2.93. Put a negative number in brackets, `7 % (-3)`, or use `mod`, and you get the remainder, 1.

Programmers use the modulo operator a lot, for two things. The first is to test whether one number divides into another, as in Program 109. The second is to keep a number inside a range: `x % 500` is always between 0 and 499, however big *x* gets. Program 110 is the moving ball of Program 81 in Chapter 12, rewritten with `%`. When the ball goes off one side of the screen, it comes back on the other side.

Moving right, `x = (x + r) % w` adds 20 pixels and then takes the remainder of a division by the width, 500. As long as *x* is below 500, that changes nothing; 510 becomes 10. Moving left, *x* − *r* could become negative, and the remainder of a negative number is negative too. Adding the width first, `(x - r + w) % w`, keeps it positive: −10 becomes 490. The place of the ball is its center, so near an edge, part of the ball is cut off.



After K, K from x = 470: the ball came back on the left.

moveballmod.kbs — BASIC-256

```
# moveballmod.kbs - Program 81 again: the ball
# wraps around the edges with the modulo operator

print "Use i for up, j for left, k for right, m for down,"
print "and q to quit."

fastgraphics
w = graphwidth
h = graphheight
r = 20      # size of the ball (radius)

# start in the middle
x = w / 2
y = h / 2
call drawball(x, y, r)

while true
   k = key
   if k = asc("I") then
      # y - r may be below 0: adding h keeps it positive
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
      y = (y - r + h) % h
      call drawball(x, y, r)
   end if
   if k = asc("J") then
      x = (x - r + w) % w
      call drawball(x, y, r)
   end if
   if k = asc("K") then
      x = (x + r) % w
      call drawball(x, y, r)
   end if
   if k = asc("M") then
      y = (y + r) % h
      call drawball(x, y, r)
   end if
   if k = asc("Q") then exit while
end while
print "All done."
end

subroutine drawball(bx, by, br)
   clg white
   color red
   circle bx, by, br
   # a shine: smaller, lighter circles up and to the left
   color rgb(255, 100, 100)
   circle bx - br / 4, by - br / 4, br / 2
   color rgb(255, 200, 200)
   circle bx - br / 4, by - br / 4, br * 0.2
   refresh
end subroutine
```

Program 110 · Move Ball — Use Modulo to Keep on Screen

The integer division operator

The integer division operator `\` gives the other half of a long division: how many whole times the divisor goes into the dividend. 43 divided by 6 is 7, with 1 left over. Program 111 uses `\` and `%` together, and checks the answer.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

integerdivision.kbs BASIC-256

```
# integerdivision.kbs - check your long division

inputinteger "Dividend (the number to divide): ", dividend
inputinteger "Divisor (the number to divide by): ", divisor
quotient = dividend \ divisor
remainder = dividend % divisor
s = dividend + " / " + divisor + " is " + quotient
print s + " remainder " + remainder
s = "Check: " + quotient + " * " + divisor + " + " + remainder
print s + " = " + (quotient * divisor + remainder)
```

Program 111 · Check Your Long Division

TEXT OUTPUT

```
Dividend (the number to divide): 43
Divisor (the number to divide by): 6
43 / 6 is 7 remainder 1
Check: 7 * 6 + 1 = 43
```



NEW CONCEPT

```
a \ b
```

Integer division divides *a* by *b* and gives the whole number of times *b* goes into *a*; the part after the decimal point is dropped. Decimal numbers are cut down to whole numbers first, as with **%**. Together, `a \ b` and `a % b` always fit: (*a* \ *b*) × *b* + *a* % *b* is *a* again. Dividing by 0, with `\`, `%` or `/`, stops the program with the error *Division by zero*.

The power operator

The power operator `^` raises a number to a power: `2 ^ 3` is 2 × 2 × 2. Program 112 prints the powers of two, which you meet everywhere in computers: a byte has 2^8 = 256 values.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

power.kbs BASIC-256

```
# power.kbs - the powers of two

for t = 0 to 16
   print "2 ^ " + t + " = " + (2 ^ t)
next t
```

Program 112 · The Powers of Two

TEXT OUTPUT

```
2 ^ 0 = 1.0
2 ^ 1 = 2.0
2 ^ 2 = 4.0
2 ^ 3 = 8.0
2 ^ 4 = 16.0
2 ^ 5 = 32.0
2 ^ 6 = 64.0
2 ^ 7 = 128.0
2 ^ 8 = 256.0
2 ^ 9 = 512.0
2 ^ 10 = 1024.0
2 ^ 11 = 2048.0
2 ^ 12 = 4096.0
2 ^ 13 = 8192.0
2 ^ 14 = 16384.0
2 ^ 15 = 32768.0
2 ^ 16 = 65536.0
```



NEW CONCEPT

```
a ^ b
```

The **power** operator raises *a* to the power *b*: the mathematical a^b is written `a ^ b`. Its answer is always a decimal number, even for whole numbers, which is why Program 112 prints 1024.0; use `int(2 ^ 10)` if you need a whole number. The power may be a fraction or negative: `2 ^ 0.5` is the square root of 2, and `2 ^ -1` is 0.5.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

From decimals to whole numbers

`int`, `ceil`, `floor` and `round` each deal with the part after the point in their own way.

You have used **int** since Chapter 7: it simply throws the part after the decimal point away. For a positive number that makes it a little smaller, but for a negative number it makes it a little *bigger*: `int(-4.56)` is −4. Three more functions give you a choice. **ceil** (from "ceiling") always goes up to the next whole number, **floor** always goes down, and **round** goes to the nearest one.

VALUE	INT	CEIL	FLOOR	ROUND
-4.56	−4	−4	−5	−5.0
-0.56	0	0	−1	−1.0
0.56	0	1	0	1.0
2.5	2	3	2	3.0
5.56	5	6	5	6.0

intceilfloor.kbs BASIC-256

```
# intceilfloor.kbs - the difference between int,
# ceil, floor and round

for t = 1 to 8
   n = rand * 100 - 50
   s = n + "  int=" + int(n) + "  ceil=" + ceil(n)
   s += "  floor=" + floor(n) + "  round=" + round(n)
   print s
next t
```

Program 113 · Difference Between Int, Ceil, Floor and Round

TEXT OUTPUT

```
31.4527921971  int=31  ceil=32  floor=31  round=31.0
-39.9422407499  int=-39  ceil=-39  floor=-40  round=-40.0
36.9984091044  int=36  ceil=37  floor=36  round=37.0
-5.42281124672  int=-5  ceil=-5  floor=-6  round=-5.0
3.91075079662  int=3  ceil=4  floor=3  round=4.0
-46.5999813672  int=-46  ceil=-46  floor=-47  round=-47.0
-0.17026386134  int=0  ceil=0  floor=-1  round=0.0
28.5695201601  int=28  ceil=29  floor=28  round=29.0
```

Every run picks other random numbers.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
int(expression)
ceil(expression)
floor(expression)
round(expression)
round(expression, digits)
```

int drops the part after the point. It also turns a string into a whole number: `int("42")` is 42; a string that isn't a number gives 0.
ceil gives the nearest whole number that is not smaller, **floor** the nearest one that is not bigger. Both give whole numbers.
round gives the nearest whole number; a number exactly halfway goes away from zero, so 2.5 becomes 3 and −2.5 becomes −3. Its answer is a decimal number (3.0); wrap it in **int** for a whole one. With a second number, it rounds to that many digits after the point: `round(3.14159, 2)` is 3.14.



ADD 0.5 AND DROP THE REST?

You may have learned to round by adding 0.5 and dropping the decimals. With **int** that goes wrong for negative numbers: `int(-2.7 + 0.5)` is −2, not −3. `floor(b + 0.5)` works for negative numbers too, but rounds −2.5 to −2. **round** does it the way most people expect, so use it.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

More functions for decimal numbers

Absolute values, square roots and logarithms.

Most programs don't need these functions, but when you do need one, it saves a lot of work.



NEW CONCEPT

FUNCTION	GIVES
`abs(x)`	the **absolute value**: *x* without its minus sign. `abs(-3)` is 3.
`sqr(x)` or `sqrt(x)`	the **square root**: the number that, multiplied by itself, gives *x*. `sqr(16)` is 4.0. The root of a negative number is an error.
`exp(x)`	*e* (about 2.718) to the power *x*.
`log(x)`	the natural **logarithm** (base *e*): the opposite of **exp**.
`log10(x)`	the base 10 logarithm: the power of 10 that gives *x*. `log10(1000)` is 3.0.

Program 114 puts a few of them to work. `int(log10(n)) + 1` is the number of digits of a whole number *n*: `log10` of a number from 100 to 999 is from 2 to just under 3. Then the program works out the distance between two points with the rule of Pythagoras: the square root of $dx^2 + dy^2$, where *dx* and *dy* are how far apart the points are across and down.

numberfacts.kbs BASIC-256

```
# numberfacts.kbs - square roots, logarithms and distances

numbers = {7, 42, 365, 2026, 1000000}
for i = 0 to numbers[?] - 1
   n = numbers[i]
   digits = int(log10(n)) + 1
   print n + ": " + digits + " digit(s), square root " + sqr(n)
next i

# how far apart are two points? (Pythagoras)
x1 = 100 : y1 = 400
x2 = 400 : y2 = 0
dx = abs(x2 - x1)
dy = abs(y2 - y1)
distance = sqr(dx ^ 2 + dy ^ 2)
print "The two points are " + distance + " pixels apart."
```

Program 114 · Square Roots, Logarithms and Distances

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
7: 1 digit(s), square root 2.64575131106
42: 2 digit(s), square root 6.48074069841
365: 3 digit(s), square root 19.1049731745
2026: 4 digit(s), square root 45.0111097397
1000000: 7 digit(s), square root 1000.0
The two points are 500.0 pixels apart.
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 4

Advanced: trigonometry

The trigonometric functions turn angles into lengths and positions, and back.

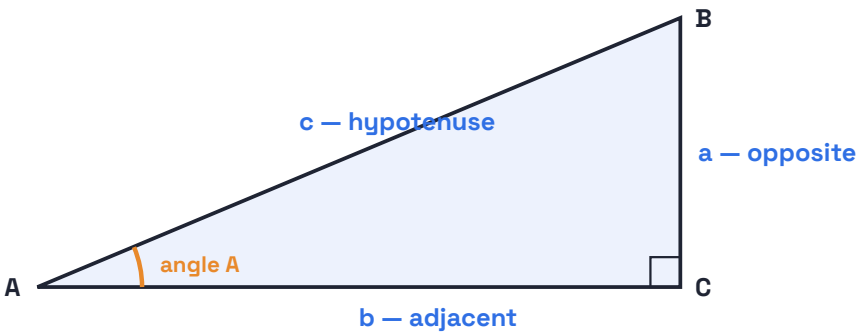
Trigonometry is the mathematics of triangles and angles. BASIC-256 measures angles in **radians**, as you saw in Chapter 9: a whole turn is 2π radians, which is 360°. If you think in degrees, **radians** and **degrees** convert for you.



NEW CONCEPT

FUNCTION	GIVES
`sin(a)`	the sine of the angle *a*
`cos(a)`	the cosine of the angle *a*
`tan(a)`	the tangent of the angle *a*
`asin(x)`	the inverse sine: the angle whose sine is *x*
`acos(x)`	the inverse cosine: the angle whose cosine is *x*
`atan(x)`	the inverse tangent: the angle whose tangent is *x*
`degrees(a)`	the angle *a* in degrees: *a* / 2π × 360
`radians(d)`	the angle *d* in radians: *d* / 360 × 2π

The first three are easiest to understand in a triangle with one right angle. The side across from the right angle, the longest, is the **hypotenuse**. Seen from angle A, side *a* is the **opposite** side and side *b* is the **adjacent** side, the one next to it.



A right triangle, seen from angle A.

Sine and cosine

The **sine** of angle A is the opposite side divided by the hypotenuse, and the **cosine** is the adjacent side divided by the hypotenuse:

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

$$\sin A = \frac{a}{c} \qquad \cos A = \frac{b}{c}$$

Both go from −1 to 1 and repeat themselves every 2π radians: a wave. Program 115 draws both waves across the screen, from 0 on the left to 2π on the right.

sinewave.kbs BASIC-256

```
# sinewave.kbs - draw the sine and cosine waves

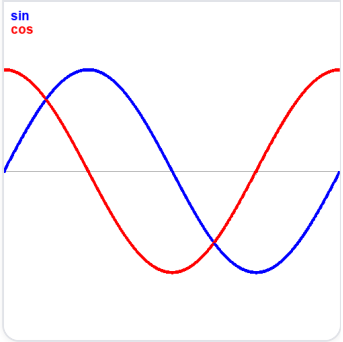
clg white
w = graphwidth
middle = graphheight / 2  # y = 0 is in the middle
amp = 150                 # 1 is 150 pixels up

# the axes: a whole turn (2 pi) fills the width
color grey
line 0, middle, w, middle
for q = 1 to 3
   x = q * w / 4
   line x, middle - 5, x, middle + 5
next q

# plot both waves, one point for each x
for x = 0 to w - 1
   a = x / w * 2 * pi       # the angle for this x
   color blue
   circle x, middle - sin(a) * amp, 2
   color red
   circle x, middle - cos(a) * amp, 2
next x

font "Arial", 14, 700
color blue
text 10, 10, "sin"
color red
text 10, 30, "cos"
```

GRAPHICS OUTPUT



sin starts at 0, cos at 1.

Program 115 · The Sine and Cosine Waves

Because the hypotenuse is the longest side, the sine and cosine can never be more than 1. That makes them perfect for circles: a point at angle *a* on a circle with radius *r* around *cx*, *cy* is `cx + sin(a) * r` across and `cy - cos(a) * r` down, with angle 0 at the top, going clockwise like a clock. The Big Program uses exactly that.

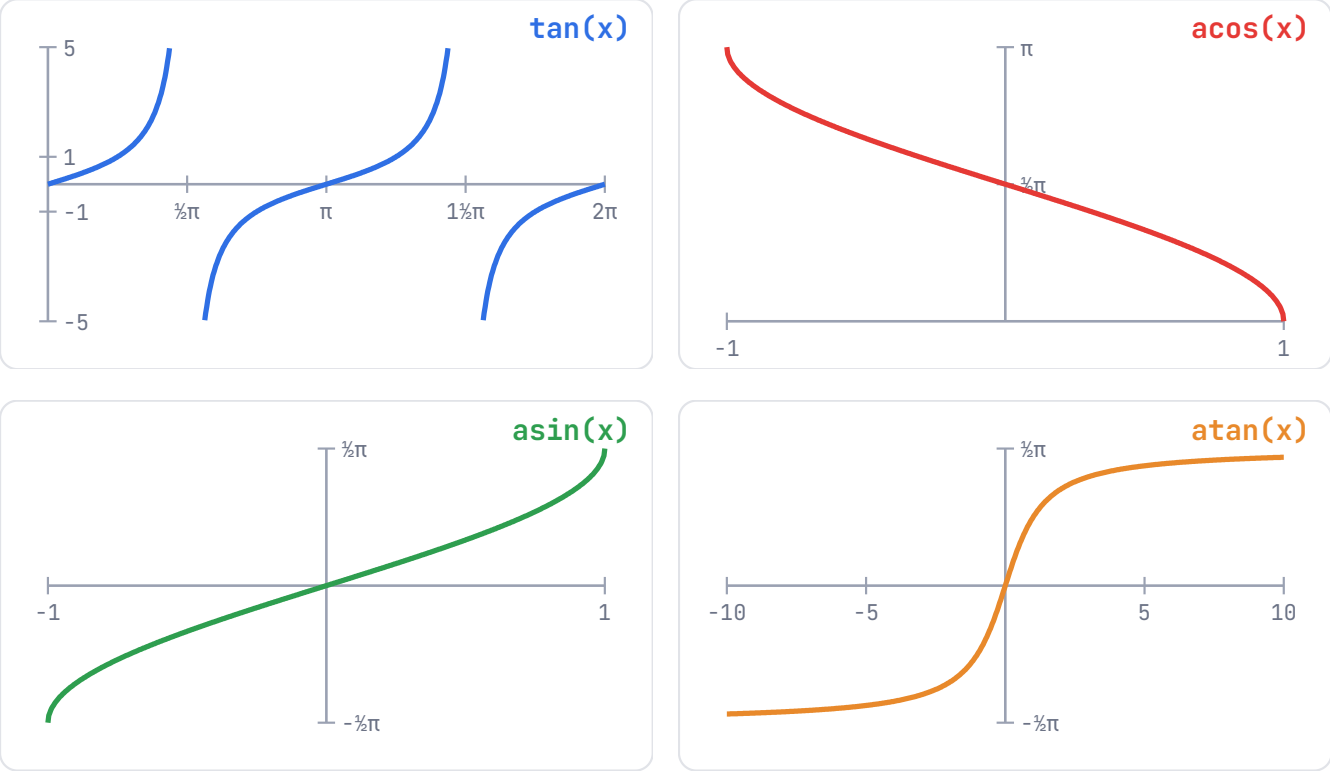
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

Tangent and the inverse functions

The **tangent** is the opposite side divided by the adjacent side:

$$\tan A = \frac{a}{b}$$

It repeats every π radians and can be any number at all. As the angle gets close to ½π (90°), the adjacent side gets very short, so the tangent grows without end. At exactly ½π there is no tangent at all; `tan(pi / 2)` gives a huge number instead, 1.6×10^{16}, because a computer can't store π exactly. The **inverse** functions work the other way round: from a sine, cosine or tangent back to the angle. `asin(1)` is ½π, `acos(-1)` is π and `atan(1)` is ¼π.



The tangent, and the three inverse functions. Each graph shows the values of *x* across and the answer of the function up.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

A Clock with Hands

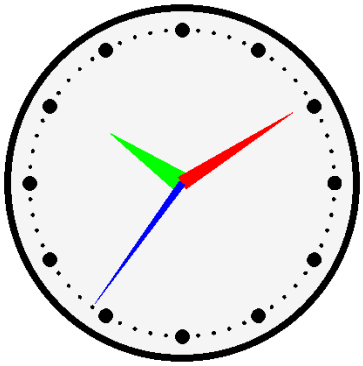
In Chapter 10, Program 68 showed the time in digits. This clock has a face with 60 marks and three hands, and the mathematics of this chapter puts every mark and every hand in its place.

The marks. Mark *m* sits at the angle 2π × *m* / 60, and **sin** and **cos** turn that angle into a place on a circle of radius 175, as explained in Section 4. `m % 5 = 0` picks out every fifth mark, the hours, and draws it bigger.

The hands. Each hand gets its place on the face as a number from 0 to 60, like the marks. For the minute hand that is the minutes plus a part of a minute. The hour hand moves 5 marks an hour: `hour % 12` turns 13 to 23 into 1 to 11, times 5, plus a fifth of a mark for every 12 minutes.

Drawing a hand. The **stamp** is a thin triangle pointing right. **stamp** turns it clockwise by its angle (Chapter 9), and the extra −π/2 (a quarter turn back) makes place 0 point straight up.

GRAPHICS OUTPUT



The clock at 10:09:36.

handyclock.kbs BASIC-256

```
# handyclock.kbs - a clock with hands

fastgraphics
cx = graphwidth / 2      # the center of the clock
cy = graphheight / 2

while true
   clg white
   # the face
   color black, rgb(245, 245, 245)
   penwidth 8
   circle cx, cy, 200
   # 60 marks around the face; every fifth one is bigger
   penwidth 1
   color black
   for m = 0 to 59
      a = 2 * pi * m / 60
      if m % 5 = 0 then
         pip = 8
      else
         pip = 2
      end if
      circle cx + sin(a) * 175, cy - cos(a) * 175, pip
   next m
   # the hands: their place on the face, from 0 to 60
   h = hour % 12 * 5 + minute / 12
   call drawhand(cx, cy, h, 100, 12, green)
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
    m = minute + second / 60
    call drawhand(cx, cy, m, 150, 8, red)
    call drawhand(cx, cy, second, 170, 4, blue)
    refresh
    pause 1
end while
end

subroutine drawhand(x, y, f, l, w, handcolor)
    # x, y: the center of the clock
    # f: the place on the face, from 0 to 60
    # l, w: the length and width of the hand
    color handcolor
    stamp x, y, 1, f / 60 * 2 * pi - pi / 2, {0, -w, l, 0, 0, w}
end subroutine
```

Program 116 · Big Program — Clock with Hands



TRY IT

Add the numbers 1 to 12 around the face with **text**, at radius 140. Work out the place of each number with **sin** and **cos**, the same way as the marks.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

t	j	v	d	e	g	r	e	e	s	a	f	d	f
n	l	i	e	c	o	v	h	r	k	b	a	l	e
e	s	m	a	t	a	n	h	e	o	s	d	e	u
g	y	h	o	o	n	e	n	d	s	f	j	m	w
n	y	t	l	p	v	n	f	n	a	l	a	b	c
a	h	i	u	p	f	i	w	i	s	o	c	j	e
t	h	r	d	o	p	s	e	a	i	o	e	d	l
s	f	a	o	s	s	o	y	m	n	r	n	u	c
i	d	g	m	i	o	c	o	e	u	b	t	b	v
n	j	o	i	t	c	p	d	r	m	i	u	j	k
e	k	l	n	e	a	g	r	e	w	o	p	v	v
o	w	e	s	u	n	e	t	o	p	y	h	w	d
m	r	a	d	i	a	n	s	e	i	t	h	p	y
w	y	h	j	w	r	o	u	n	d	m	n	a	o

Find: abs, acos, adjacent, asin, atan, ceil, cosine, degrees, floor, hypotenuse, logarithm, modulo, opposite, power, radians, remainder, round, sine, tangent



Problems

Try these — they only use what this chapter and the earlier ones covered.

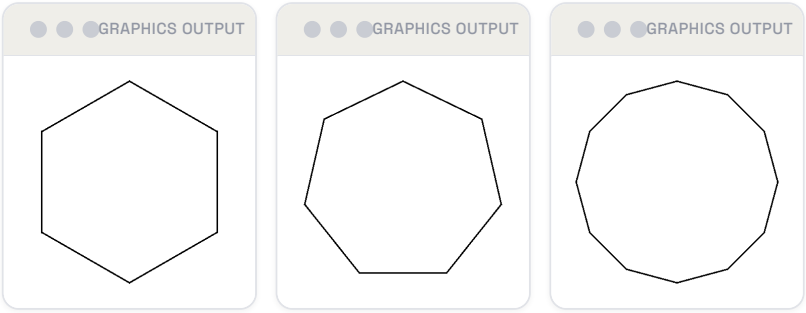
1 Ask the user for a decimal number with **inputfloat**. Print it as a whole number and the nearest fraction over 1000. For 3.14159, print *3 and 142/1000*.

2 Take your program from Problem 1 and make the fraction as simple as possible: as long as some number divides both the top and the bottom exactly (test it with `%`), divide both by it. 142/1000 becomes 71/500.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

3 Write a program that draws a regular polygon with any number of sides, 3 or more. Put its center in the middle of the graphics output and its corners 200 pixels from the center. Hint: this loop plots a circle with a radius of 200 pixels around 250, 250 — you only need some of its points.

```
for a = 0 to 2 * pi step 0.01
   plot 250 + 200 * sin(a), 250 - 200 * cos(a)
next a
```



6 sides 7 sides 12 sides

4 A **prime number** can only be divided exactly by 1 and by itself: 2, 3, 5, 7, 11 … Write a function that returns true when a number is prime, by trying every divisor from 2 up to its square root with `%`. Use it to print all the prime numbers below 100.

Chapter 16 · Mathematics — More Fun with Numbers. Modernised layout of the BASIC-256 beginner's guide, updated for BASIC-256 2.3. Programs are numbered 109–116, continuing from Chapter 15; the array sections the original printed at the start of this chapter are now in Chapter 15. Corrected for 2.3: `^` always gives a decimal number (1024.0); `mod` is the word form of `%`, and `7 % -3` is read as a percentage unless the negative number is in brackets; `round` exists and replaces the `floor(b + 0.5)` advice; the remainder takes the sign of the first number. Also corrected: the tangent is the opposite side over the adjacent side, and it is the adjacent side that gets short near ½π; integer division gives how many times the divisor goes into the dividend (the old box had it the other way round); the cosine graph was called Illustration 24; the Big Program was described as a long division program but is a clock; its hour hand added second / 3600, which is now left out (minute / 12 is enough); "floor(0)", "next lowers" and "faction" are fixed. New: Program 114 (square roots, logarithms, Pythagoras), Program 115 (the sine and cosine waves, drawn by BASIC-256), the round column in Program 113, the `sqr` and `exp` functions, a percent warning, and Problem 4. The clock, the moving ball and the polygons are drawn for the 500 × 500 graphics output. Every program was run in BASIC-256 2.3; sample outputs of programs that ask for input were made with copies that print the typed answers (shown in orange), and the clock picture with a copy that draws a fixed time. The graphs of tan, asin, acos and atan are calculated, not screenshots. Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence. Mascot: BitBot.