# Chapter 5. Binary, octal and hexadecimal numbers

A place to look for some of this material is the Wikipedia page
`http://en.wikipedia.org/wiki/Binary_numeral_system#Counting_in_binary`

Another place that is relevant is `http://accu.org/index.php/articles/1558` (which explains many more details thane we will discuss) and the website
`http://www.binaryconvert.com`

For ASCII (mentioned briefly below) see `http://en.wikipedia.org/wiki/ASCII`

However, I hope that the explanations given here are adequate.

The material here is probably new to you, but it should not require any prior knowledge. It really starts with primary school mathematics and is all more or less common sense.

The rationale for including this material in the course is that the ideas are used all through computing, at least once you look under the cover of a computer.

**5.1 Counting.** Normally we use *decimal* or base 10 when we count. What that means is that we count by tens, hundreds = tens of tens, thousands = tens of hundreds, etc.

We see that in the SI units we are familiar with in science (kilometres = $10^3$ metres, kilogrammes, centimetres = $10^{-3}$ metres). We can become so used to it that we don't think about it. When we write the number 5678, we learned in the primary school that the 8 means 8 units, the 7 is 7 tens, while the remaining digits are 6 hundreds = $6 \times 10^2$ and $5 \times 10^3$. So the number 5678 means

$$5 \times 10^3 + 6 \times 10^2 + 7 \times 10 + 8$$

Although base 10 is the most common, we do see some traces of other bases in every day life. For example, we normally buy eggs by dozens, and we can at least imagine shops buying eggs by the gross (meaning a dozen dozen or $12^2 = 144$). So we use base 12 to some extent.

We can see some evidence of base 60 in angles and in time. In time units, 60 seconds is a minute and 60 minutes ($= 60^2$ seconds) is an hour. Logically then we should have 60 hours in a day? Since we don't we stop using base 60. In degree measure of angles, we are familiar with 60 minutes in a degree.

**5.2 Binary.** In *binary* or *base 2* we count by pairs. So we start with zero, then a single unit, but once we get to two units of any size we say that is a pair or a single group of 2.

So, when we count in base 2, we find:

- 1 is still 1
- 2 becomes a single group of 2 (a single pair)
  Using positional notation as we do for decimal, we write this as $10$. To make sure we are clear which base we are using, we may write a subscript 2 as in $(10)_2$
- 3 is $(11)_2 =$ one batch of 2 plus 1 unit.
- 4 is $(100)_2 =$ one batch of $2^2 + 0$ batches of $2 + 0$ units

Using a more succinct format, we can explain how to count in binary as follows:

| **Decimal #** | **in binary** | Formula for the binary format |
|---|---|---|
| 1 | $(1)_2$ | $1$ |
| 2 | $(10)_2$ | $1\times 2+0$ |
| 3 | $(11)_2$ | $1\times 2+1$ |
| 4 | $(100)_2$ | $1\times 2^2+0\times 2+0$ |
| 5 | $(101)_2$ | $1\times 2^2+0\times 2+1$ |
| 6 | $(110)_2$ | $1\times 2^2+1\times 2+0$ |
| 7 | $(111)_2$ | $1\times 2^2+1\times 2+1$ |
| 8 | $(1000)_2$ | $1\times 2^3+0\times 2^2+0\times 2+0$ |

So we can figure out what number we mean when we write something in binary by adding up the formula. Mind you that can get tedious, but the principle is not complicated.

At least for small numbers, there is a way to find the binary digits for a given number (*i.e.*, given in base 10) by repeatedly dividing by 2. For very small numbers, we can more or less do it by eye. Say for the number twenty one, we can realise that it is more than $16 = 2^4$ and not as big as $32 = 2^5$. In fact

$$21 = 16 + 5 = 16 + 4 + 1 = 2^4 + 2^2 + 1 = (10101)_2$$

Or we can start at the other end and realise that since 21 is odd its binary digits must end in 1. $21/2 = 10+$ remainder 1.

To explain how this goes a bit more clearly, suppose we are starting with a positive integer number $n$ (recall that an integer is a whole number, no fractional part). We want to know it in binary and in order to dicuss what we are doing we will write down the units digit as $n_0$, the next digit from the right (multiples of 2) as $n_1$, etc. So we suppose the binary representation of the number $n$ is

$$n = (n_k n_{k-1}\cdots n_2 n_1 n_0)_2 = n_k 2^k + n_{k-1}2^{k-1} + \cdots + n_2 2^2 + n_1 2^1 + n_0$$

where the digits $n_0, n_1, \ldots, n_k$ are each either 0 or 1 and $k$ is big enough. (In fact we need $k$ to be so big that $2^k \le n < 2^{k+1}$.)

If we divide $n$ by 2 we get

$$\text{quotient} = \text{whole number part of } \frac{n}{2} = n_k 2^{k-1} + n_{k-1}2^{k-2} + \cdots + n_2 2^1 + n_1$$

and remainder $n_0$. The remainder is 1 if $n$ is odd and 0 if $n$ is even.

Now if we divide again by 2 we get remainder $n_1$ and new quotient

$$\text{quotient} = n_k 2^{k-2} + n_{k-1}2^{k-3} + \cdots + n_2$$

Look again at the case $n = 21$ as an example. We had $\frac{21}{2} = 10+$ remainder 1. So the last binary digit is $1 =$ that remainder. Now $10/2 = 5+$ no remainder. That makes the digit in the 2's place 0. $5/2 = 2+$ remainder 1. So if we repeatedly divide by 2 and keep track of the remainder *each time* (even when the remainder is zero) we discover the binary digits one at a time from the units place up.

One thing to notice about binary is that we only ever need two digits, 0 and 1. We never need the digit 2 because that always gets 'carried' or moved to the next place to the left.

**5.3 Octal.** In *octal* or *base 8* we count by 8's. Otherwise the idea is similar. We need 8 digits now: 0, 1, 2, 3, 4, 5, 6 and 7. So now zero is still 0 in octal, 1 is 1, 2 is 2, *etc.* 7 is still 7 in octal, but eight becomes $(10)_8$. In base 8 $(10)_8$ means $1 \times 8 + 0$.

Using a layout similar to the one used before we can explain how to count in octal as follows:

| **Decimal #** | **in octal** | Formula for the octal format |
|---:|---:|---:|
| 1 | $(1)_8$ | 1 |
| 2 | $(2)_8$ | 2 |
| 7 | $(7)_8$ | 7 |
| 8 | $(10)_8$ | $1 \times 8 = 0$ |
| 9 | $(11)_8$ | $1 \times 8 + 1$ |
| 10 | $(12)_8$ | $1 \times 8 + 2$ |
| 16 | $(20)_8$ | $2 \times 8 + 0$ |
| 17 | $(21)_8$ | $2 \times 8 + 1$ |

**5.4 Hex.** Now we have the idea, we can think of counting in other bases, such as base 6 or base 9, but these are not used in practice. What is used is base 16, also called *hexadecimal*.

We can go ahead as we did before, just counting in groups and batches of 16. However, we run into a problem with the notation caused by the fact that the (decimal) number 10, 11, 12 13, 14 and 15 are normally written using two adjacent symbols. If we write 11 in hexadecimal, should we mean ordinary eleven or $1 \times 16 + 1$?

To get around this difficulty we need new symbols for the numbers ten, eleven, ..., fifteen. What we do is use letters *a*, *b*, *c*, *d*, *e* and *f* (or sometimes the capital letters A, B, C, D, E and F).

Thus the number ten becomes a single digit number $(a)_{16}$ in hexadecimal. Eleven becomes $(b)_{16}$, and so on. But sixteen becomes $(10)_{16}$.

Using a layout similar to the one used before we can explain how to count in hex as follows:

| **Decimal #** | **in hex** | Formula for the hexadecimal format |
|---:|---:|---:|
| 1 | $(1)_{16}$ | 1 |
| 9 | $(9)_{16}$ | 9 |
| 10 | $(a)_{16}$ | 10 |
| 15 | $(f)_{16}$ | 15 |
| 16 | $(10)_{16}$ | $1 \times 16 = 0$ |
| 17 | $(11)_{16}$ | $1 \times 16 + 1$ |
| 26 | $(1a)_{16}$ | $1 \times 16 + 10$ |
| 32 | $(20)_{16}$ | $2 \times 16 + 0$ |
| 165 | $(a5)_{16}$ | $10 \times 16 + 5$ |
| 256 | $(100)_{16}$ | $1 \times 16^2 + 0 \times 16 + 0$ |

**5.5 Converting Octal or Hex to binary.** We did already discuss some conversions of integers between different bases.

There is a method based on repeated division and keeping track of remainders. We can use this to convert from decimal to octal, to hex, or to binary.

If we write out the formula corresponding to a number in binary, octal or hex, we can compute the number in decimal by evaluating the formula.

These methods involve quite a bit of work, especially if the number is large. However there is a very simple way to convert between octal and binary. It is based on the fact that $8 = 2^3$ is a power of 2 and so it is very easy to convert base 8 to base 2.

$$\begin{aligned}(541)_8 &= 5 \times 8^2 + 4 \times 8 + 1 \\ &= (1 \times 2^2 + 0 \times 2 + 1) \times 2^6 + (1 \times 2^2) \times 2^3 + 1 \\ &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^6 + 1 \times 2^5 + 1 \\ &= (101100001)_2\end{aligned}$$

If we look at how this works, it basically means that we can convert from octal to binary by converting each octal digit to binary separately *but* we must write each digit as a 3 digit binary number. Redoing the above example that way we have $5 = (101)_2$ (uses 3 digits anyhow), $4 = (100)_2$ (again uses 3 digits) and $1 = (1)_2 = (001)_2$ (here we have to force ourselves to use up 3 digits) and we can say

$$(541)_8 = (101\,100\,001)_2 = (101100001)_2$$

This method works with any number of octal digits and we never have to really convert anything but the 8 digits 0-7 to binary. It is also reversible. We can convert any binary number to octal very quickly if we just group the digits in 3's starting from the units. For example

$$(1111010100001011)_2 = (001\,111\,010\,100\,001\,011)_2 = (172413)_8$$

A similar method works for converting between binary and hex, except that now the rule is "4 binary digits for each hex digit". It all works because $16 = 2^4$. For example

$$(a539)_{16} = (1010\,0101\,0011\,1001)_2 = (1010010100111001)_2$$

Or going in reverse

$$(1111010100001011)_2 = (1111\,0101\,0000\,1011)_2 = (f50b)_{16}$$

We can use these ideas to convert octal to hex or *vice versa* by going via binary. We never actually have to convert any number bigger than 15.

If we wanted to convert a number such as 5071 to binary, it may be easier to find the octal representation (by repeatedly dividing by 8 and keeping track of all remainders) and then converting

to binary at the end via the "3 binary digits for one octal" rule.

$$\begin{aligned}
\frac{5071}{8} &= 633 + \text{ remainder } 7\\
\frac{633}{8} &= 79 + \text{ remainder } 1\\
\frac{79}{8} &= 9 + \text{ remainder } 7\\
\frac{9}{8} &= 1 + \text{ remainder } 1\\
\frac{1}{8} &= 0 + \text{ remainder } 1\\
(5071)_{10} &= (11717)_8\\
&= (001\,001\,111\,001\,111)_2\\
&= (001001111001111)_2\\
&= (1001111001111)_2
\end{aligned}$$

**5.6 Relation with computers.** Although computers are very sophisticated from the outside, with all kinds of flashy buttons, screens and so forth, the basic works are essentially many rows of on/off switches. Clearly a single on/off switch has only 2 possible settings of or states, but a row of 2 such switches has 4 possible states.

| on | on | | on | off | | off | on | | off | off |
|---|---|---|---|---|---|---|---|---|---|---|

A row of 3 switches has twice as many possible setting because the third switch can be either on or off for each of the 4 possibilities for the first two. So $2^3$ possibilities for 3 switches. In general $2^8 = 256$ possibilities for 8 switches, $2^n$ possible settings for a row of $n$ switches.

Computers generally work with groups of 32 switches (also called 32 *bits*, where a 'bit' is the official name for the position that can be either on or off) and sometimes now with groups of 64. With 32 bits we have a total of $2^{32}$ possible settings.

How big is $2^{32}$? We could work out with a calculator that it is $4294967296 = 4.294967296 \times 10^9$ but there is a fairly simple trick for finding out approximately how large a power of 2 is. It is based on the fact that

$$2^{10} = 1024 \cong 10^3$$

Thus

$$2^{32} = 2^2 \times 2^{30} = 4 \times (2^{10})^3 \cong 4 \times (10^3)^3 = 4 \times 10^9$$

You can see that the answer is only approximate, but the method is fairly painless (if you are able to manipulate exponents).

**5.7 Integer format storage.** Computers use binary to store everything, including numbers. For numbers, the system used for integers (that is whole numbers, with no fractional part) is simpler to explain than the common system for dealing with numbers that may have fractional parts.

In general modern computers will use 32 bits to store each integer. (Sometimes, they use 64 but we will concentrate on a 32 bit system.) How are the bits used? Take a simple example like 9. First write that in binary

$$9 = (1001)_2$$

and that only has 4 digits. We could seemingly manage by using only 4 bits (where we make them on, off, off, on) and it seems a waste to use 32 bits. However, if you think about it you can see that we would need to also record how many bits we needed each time. Generally it is simpler to decide to use 32 bits from the start. For this number 9 we can pad it out by putting zeros in front

$$9 = (1001)_2 = (00\ldots001001)_2$$

and then we end up filling our row of 32 bits like this:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 0 | 0 | … | 0 | 0 | 1 | 0 | 0 | 1 |
| Bit position: | 1 | 2 | … | 27 | 28 | 29 | 30 | 31 | 32 |

One practical aspect of this system is that it places a limit on the maximum size of the integers we can store.

Since we allocate 32 bits we have a total of $2^{32} \cong 4 \times 10^9$ different settings and so we have room for only that many different integers. So we could fit in all the integers from 0 to $2^{32} - 1$, but that is usually not such a good strategy because we may also want to allow room for negative integers. If we don't especially favour positive integers over negative ones, that leaves us with space to store the integers from about $-2^{31}$ to $2^{31}$. To be precise, that would be $2 \times 2^{31} + 1 = 2^{32} + 1$ numbers if we include zero and so we would have to leave out either $\pm 2^{31}$.

Notice that $2^{31} \cong 2 \times 10^9$ is not by any means a huge number. In a big company, there would be more Euros passing through the accounts than that in a year. In astronomy, the number of kilometres between stars would usually be bigger than that.

Computers are not actually limited to dealing with numbers less than $2 \times 10^9$, but they often are limited to dealing in this range for exact integer calculations. We will return to another method for dealing with numbers that have fractional parts and it allows for numbers with much larger magnitudes. However, this is done at the expense of less accuracy. When dealing with integers (that are within the range allowed) we can do exact calculations.

Returning to integers, we should explain about how to deal with negative integers. One way would be to allocate one bit to be a sign bit. So bit number 1 on could mean a minus sign. In this way we could store

$$-9 = -(1001)_2 = -(0\ldots001001)_2$$

by just turning on the first bit. However, if you ask your calculator to tell you $-9$ in binary, you will get a different answer. The reason is that computers generally do something more complicated with negative integers. This extra complication is not so important for us, but just briefly the idea is that the method used saves having to ever do subtraction. So $-1$ is actually stored as all ones:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| -1 | 1 | 1 | … | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | … | 0 | 0 | 0 | 0 | 0 | 1 |
| Bit position: | 1 | 2 | … | 27 | 28 | 29 | 30 | 31 | 32 |

If you add 1 to that in binary, you will have to carry all the time. Eventually you will get zeros in all 32 allowable places and you will have to carry the last 1 past the end. Since, only 32 places are allowed, this final carried 1 just disappears and we get 32 zeros, or 0.

In general, to store a negative number we take the binary form of 1 less than the number and then take what is called the 'ones complement' of that. So when $-9$ is stored we would look at the way we would store $+8$ and then flip all the 1's to 0's and the 0's to 1's.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 0 | 0 | ... | 0 | 0 | 1 | 0 | 0 | 0 |
| -9 | 1 | 1 | ... | 1 | 1 | 0 | 1 | 1 | 1 |
| Bit position: | 1 | 2 | ... | 27 | 28 | 29 | 30 | 31 | 32 |

Using this method of storing negative numbers, all subtractions can be performed as though they were additions (using the carrying rules and the possibility that a 1 will get lost past the 32nd position).

By the way, the numbering of the bits is not really fixed. We could number them from right to left instead, but this really depends on whether you are looking at the bits from the top or the bottom. In any case, you can't really see bits normally.

Simple computer programs that use integers will be limited to the range of integers from $-2^{31}$ up to $2^{31} - 1$ (which is the number that has 31 1's in binary). However, it is possible to write programs that will deal with a larger range of integers. You can arrange your program to use more than 32 bits to store each integer, for example to use several rows of 32 bits. However, the program will then generally have to be able to implement its own carrying rules and so forth for addition and subtraction of these bigger integers. So you will not simply be able to use the ordinary plus and times that you can use with regular integers.

**5.8 Why octal and Hex?.** We can now explain why computer people are fond of base 16 or hex. Octal looks easier to read (no need to worry about the new digits $a$ for ten, etc) but in computers we are frequently considering 32 bits at a time. Using the "3 binary for one octal" rule, this allows us to write out the 32 bits quickly, but it takes us eleven octal digits. The messy part is that we really don't quite use the eleventh octal digit fully. It can be at most $(11)_2 = 3$.

With hex, we have a "4 binary digits for one hex" rule and 32 binary digits or bits exactly uses up 8 hex digits.

Computers use binary for everything, not just numbers. For example, text is encoded in binary by numbering all the letters and symbols. The most well used method for doing this is called ASCII (an acronym that stands for 'American Standard Code for Information Interchange). and it uses 7 binary digits or bits for each letter. You may be surprised that there are $2^7 = 128$ letters, as there seem to be only 26 normally. But if you think a little you will see that there are 52 if you count upper and lower case separately. Moreover there are punctuation marks that you will see on your computer keyboard like

```
! , . ? / \ @ # ~ [ ] { } ( ) * & ^ % $
```

and we also need to keep track of the digits 0-9 as symbols (separately to keeping track of the numerical values). Finally the ASCII code allocates numbers or bit patterns to some 'invisible' things like space, tab, new line and in the end 127 is just barely enough. On a UNIX system, you can find out what the ASCII code is by typing the command

```
man ascii
```

at the command line prompt. Here is what you will see:

```
ASCII(7)     FreeBSD Miscellaneous Information Manual         ASCII(7)

NAME
 ascii - octal, hexadecimal and decimal ASCII character sets

DESCRIPTION
 The octal set:

 000 nul  001 soh  002 stx 003 etx  004 eot  005 enq  006 ack  007 bel
 010 bs   011 ht   012 nl  013 vt   014 np   015 cr   016 so   017 si
 020 dle  021 dc1  022 dc2 023 dc3  024 dc4  025 nak  026 syn  027 etb
 030 can  031 em   032 sub 033 esc  034 fs   035 gs   036 rs   037 us
 040 sp   041  !   042  "  043  #    044  $   045  %   046  &   047  ’
 050  (   051  )   052  *   053  +   054  ,   055  -   056  .   057  /
 060  0   061  1   062  2   063  3   064  4   065  5   066  6   067  7
 070  8   071  9   072  :   073  ;   074  <   075  =   076  >   077  ?
 100  @   101  A   102  B   103  C   104  D   105  E   106  F   107  G
 110  H   111  I   112  J   113  K   114  L   115  M   116  N   117  O
 120  P   121  Q   122  R   123  S   124  T   125  U   126  V   127  W
 130  X   131  Y   132  Z   133  [   134  \   135  ]   136  ^   137  _
 140  `   141  a   142  b   143  c   144  d   145  e   146  f   147  g
 150  h   151  i   152  j   153  k   154  l   155  m   156  n   157  o
 160  p   161  q   162  r   163  s   164  t   165  u   166  v   167  w
 170  x   171  y   172  z   173  {   174  |   175  }   176  ~   177 del

 The hexadecimal set:

 00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
 08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
 10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
 18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
 20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  ’
 28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
 30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
 38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
 40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
 48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
 50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
 58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
 60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
 68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
 70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
 78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del

 The decimal set:

   0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
   8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
  16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
  24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
  32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  ’
  40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
  48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
  56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
  64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
  72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
  80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
  88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
  96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
 104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
```

```
 112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
 120  x   121  y   122  z   123  {   124  |   125  }   126  ~   127 del

FILES
 /usr/share/misc/ascii

HISTORY
 An ascii manual page appeared in Version 7 AT&T UNIX.

BSD                      June 5, 1993
```

Another place to find this information is at `http://en.wikipedia.org/wiki/ASCII`

I certainly would not try to remember this, but you can see that the symbol 'A' (capital A) is given a code $(101)_8 = (41)_{16} = (65)_{10}$ and that the rest of the capital letters follow A in the usual order. This means that A uses the 7 bits 1000001 in ASCII but computers almost invariably allocate 8 bits to store each letter. They sometimes differ about what they do with the 'wasted' 8th bit, but the extra bit allows space for $2^8 = 256$ letters while ASCII only specifies 128 different codes. If you think about it you will see that there are no codes for accented letters like á or è (which you might need in Irish or French), no codes for the Greek or Russian letters, no codes for Arabic or Hindu. In fact 8 bits (or 256 total symbols) is nowhere near enough to cope with all the alphabets of the World. That is a reflection of the fact that ASCII goes back to the early days of computers when memory was relatively very scarce compared to now, and also when the computer industry was mostly American. The modern system (not yet universally used) is called UNICODE and it allocates 16 bits for each character. Even with $2^{16} = 65536$ possible codes, there is a difficulty accommodating all the worlds writing systems (including Chinese, Japanese, mathematical symbols, etc).

**5.9 Converting fractions to binary.** We now look at a way to convert fractions to binary. We already mentioned that in principle every real number, including fractions, can be represented as a "binary decimal" (or really using a "binary point"). We have already looked into whole numbers fairly extensively and what remains to do is to explain a practical way to find the binary digits for a fraction. You see if we start with (say $\frac{34}{5}$ we can say that is $6 + \frac{4}{5}$. We know $6 = (110)_2$ and if we could work out how to represent $\frac{4}{5}$ as 0. something in binary then we would have $\frac{34}{5} = 6 + \frac{4}{5} = (110.\text{ something})_2$.

To work out what 'something' should be we work backwards from the answer. We can't exactly do that without having the answer, but we can proceed as we do in algebra by writing down the unknown digits. Later we will work them out. Say the digits we want are $b_1, b_2, b_3, \ldots$ and so

$$\frac{4}{5} = (0.b_1b_2b_3b_4\cdots)_2$$

We don't know any of $b_1, b_2, b_3, \ldots$ yet but we know they should be base 2 digits and so each one is either 0 or 1. We can write the above equation as a formula and we have

$$\frac{4}{5} = \frac{b_1}{2} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \frac{b_4}{2^4} + \cdots$$

If we multiply both sides by 2, we get

$$\frac{8}{5} = b_1 + \frac{b_2}{2} + \frac{b_3}{2^2} + \frac{b_4}{2^3} + \cdots$$

In other words multiplying by 2 just moves the binary point and we have

$$\frac{8}{5} = (b_1.b_2b_3b_4\cdots)_2$$

Now if we take the whole number part of both sides we get 1 on the left and $b_1$ on the right. So we must have $\boxed{b_1 = 1}$. But if we take the fractional parts of both sides we have

$$\frac{3}{5} = (0.b_2b_3b_4\cdots)_2$$

We are now in a similar situation to where we began (but not with the same fraction) and we can repeat the trick we just did. Double both sides again

$$\frac{6}{5} = (b_2.b_3b_4b_5\cdots)_2$$

Take whole number parts of both sides: $\boxed{b_2 = 1}$. Take fractional parts of both sides.

$$\frac{1}{5} = (0.b_3b_4b_5\cdots)_2$$

We can repeat our trick as often as we want to uncover as many of the values $b_1$, $b_2$, $b_3$, etc as we have the patience to discover.

What we have is a method, in fact a repetitive method where we repeat similar instructions many times. We call a method like this an *algorithm*, and this kind of thing is quite easy to programme on a computer because one of the programming instructions in almost any computer language is REPEAT (meaning repeat a certain sequence of steps from where you left off the last time).

In this case we can go a few more times through the steps to see how we get on. Double both sides again.

$$\frac{2}{5} = (b_3.b_4b_5b_6\cdots)_2$$

Whole number parts. $\boxed{b_3 = 0}$. Fractional parts

$$\frac{2}{5} = (0.b_4b_5b_6\cdots)_2$$

Double both sides again.

$$\frac{4}{5} = (b_4.b_5b_6b_7\cdots)_2$$

Whole number parts. $\boxed{b_4 = 0}$. Fractional parts

$$\frac{4}{5} = (0.b_5b_6b_7\cdots)_2$$

This is getting monotonous, but you see the idea. You can get as many of the $b$'s as you like.

But, if you look more carefully, you will see that it has now reached repetition and not just monotony. We are back to the same fraction as we began with $\frac{4}{5}$. If we compare the last equation to the starting one

$$\frac{4}{5} = (0.b_1b_2b_3b_4\cdots)_2$$

we realise that everything will unfold again exactly as before. We must find $\boxed{b_5 = b_1 = 1}$, $\boxed{b_6 = b_2 = 1}$, $\boxed{b_7 = b_3 = 0}$, $\boxed{b_8 = b_4 = 0}$, $\boxed{b_9 = b_5 = b_1}$ and so we have a repeating pattern of digits 1100. So we can write the binary expansion of $\frac{4}{5}$ down fully as a repeating pattern

$$\frac{4}{5} = (0.\overline{1100})_2$$

and our original number as

$$\frac{34}{5} = (110.\overline{1100})_2$$

**5.10 Floating point format storage.** In order to cope with numbers that are allowed to have fractional parts, computers use a binary version of the usual 'decimal point'. Perhaps we should call it a 'binary point' as "decimal" refers to base 10.

Recall that what we mean by digits after the decimal point has to do with multiples of $1/10$, $1/100 = 1/10^2 = 10^{-2}$, etc. So the number 367.986 means

$$367.986 = 3 \times 10^2 + 6 \times 10 + 7 + \frac{9}{10} + \frac{8}{10^2} + \frac{6}{10^3}$$

We use the 'binary point' in the same way with powers of $1/2$. So

$$(101.1101)_2 = 1 \times 2^2 + 0 \times 2 + 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{0}{2^3} + \frac{1}{2^4}$$

As in the familiar decimal system, every number can be written as a 'binary decimal'. (Well that is a contradictory statement. Really we should say we can write every number in binary using a binary point.) As in decimal, there can sometimes be infinitely many digits after the point.

What we do next is use a binary version of scientific notation. (You will see the ordinary decimal scientific notation on your calculator at times, when big numbers are written with an E.) The usual decimal scientific notation is like this

$$54321.67 = 5.432167 \times 10^4$$

We refer to the $5.4321$ part (a number between 1 and 10 or between -1 and -10 for negative numbers) as the *mantissa*. The power (in this case the 4) is called the *exponent*. Another decimal example is

$$-0.005678 = -5.678 \times 10^{-3}$$

and here the mantissa is $-5.678$ while the exponent is $-3$.

This is all based on the fact that multiplying or dividing by powers of 10 simply moves the decimal point around. In binary, what happens is that multiplying or dividing by powers of 2 moves the 'binary point'.

$$\begin{aligned}(101)_2 &= 1\times 2^2+0\times 2+1\\ (10.1)_2 &= 1\times 2+0+\frac{1}{2}=(101)_2\times 2^{-1}\\ (1101.11)_2 &= (1.10111)_2\times 2^3\end{aligned}$$

This last is an example of a number in the binary version of scientific notation. The mantissa is $(1.110111)_2$ and we can always arrange (no matter what number we are dealing with) to have the mantissa between 1 and 2. In fact always less than 2, and so beginning with $1.$ something always. For negative numbers we would need a minus sign in front. The exponent in this last example is 3 (the power that goes on the 2).

What we do then is write every number in this binary version of scientific notation. That saves us from having to record where to put the binary point, because it is always in the same place. Or really, the exponent tells us how far to move the point from that standard place.

Computers then normally allocate a fixed number of bits for storing such numbers. The usual default is to allocate 32 bits in total (though 64 is quite common also). Within the 32 bits they have to store the mantissa and the exponent. Computers do everything in binary. The mantissa is already in binary, but we also need the exponent in binary. So in the last example the mantissa is $+(1.110111)_2$ while the exponent is $3=(11)_2$. Computers usually allocate 24 bits for storing the mantissa (including its possible sign) and the remaining 8 bits for the exponent.

In our little example, 24 bits is plenty for the mantissa and we would need to make it longer to fill up the 24 bits: $(1.110111000\dots)_2$ will be the same as $(1.110111)_2$. However, there are numbers that need more than 24 binary digits in the mantissa, and what we must then do is round off. In fact, we have to chop off the mantissa after 23 binary places (or more usually we will round up or down depending on whether the digit in the next place is 1 or 0).

The web site `http://accu.org/index.php/articles/1558` goes into quite a bit of detail about how this is done. What you get on `http://www.binaryconvert.com` (under floating point) tells you the outcome in examples but there are many refinements used in practice that are not evident from that and that also we won't discuss.

The method we have sketched is called *single precision floating point* storage.

Another common method, called *double precision*, uses 64 bits to store each number, 53 for the mantissa (including one for the sign) and 11 for the exponent.

**5.11 Limitations of floating point.** Apart from the information about how the computer uses binary to store these 'floating point numbers'[1] we can get an idea of the scope and accuracy that this system allows.

The largest possible number we can store has mantissa $(1.111\dots)_2$ (with the maximum[2] possible number of 1's) and exponent as large as possible. Now $(1.111\dots)_2$ (with 23 1's after

[1] the words indicate that the position of the binary point is movable, controlled by the exponent

[2] The maximum in single precision, where we use 23 bits for the mantissa excluding the sign bit, would be 23 1's but in fact the usual system does not store the 1 before the points as it is always there — so we can manage 24 1's total, 23 places after the point

the point) is just about 2 and the largest possible exponent is $2^7 - 1 = 127$. [We are allowed 8 bits for the exponent, which gives us room for $2^8 = 256$ different exponents. About half should be negative, and we need room for zero. So that means we can deal with exponents from $-128$ to $+127$.] Thus our largest floating point number is about

$$2 \times 2^{127} = 2^{128}$$

This is quite a big number and we can estimate it by using the $2^{10} \cong 10^3$ idea

$$2^{128} = 2^8 \times 2^{120} = 256 \times (2^{10})^{12} \cong 256 \times (10^3)^{12} = 256 \times 10^{36} = 2.56 \times 10^{38}$$

This is quite a bit larger than the limit of around $2 \times 10^9$ we had with integers. Indeed it is large enough for most ordinary purposes.

We can also find the smallest positive number that we can store[3]. It will have the smallest possible mantissa $(1.0)_2$ and the most negative possible exponent, $-128$. So it is

$$1 \times 2^{-128} = \frac{1}{2^{128}} = \frac{1}{2.56 \times 10^{38}} \cong \frac{4}{10} \times 10^{-38} = 4 \times 10^{-39}$$

or, getting the same result another way,

$$1 \times 2^{-128} = \frac{2^2}{2^{130}} = \frac{2^2}{(2^{10})^{13}} \cong \frac{4}{(10^3)^{13}} = 4 \times 10^{-39}$$

This is pretty tiny, tiny enough for many purposes.

If we use double precision (64 bits per number, requires twice as much computer memory per number) we get an exponent range from $-2^{10} = -1024$ to $2^{10} - 1 = 1023$. The largest possible number is

$$2^{1024} = 2^4 \times 2^{1020} = 16 \times (2^{10})^{102} \cong 16 \times (10^3)^{102} = 1.6 \times 10^{307}$$

and the smallest is the reciprocal of this.

In both single and double precision, we have the same range of sizes for negative numbers as we have for positive numbers.

So the limitation on size are not severe limitations, but the key consideration is the limit on the accuracy of the mantissa imposed by the 24 bit limit (or the 53 bit limit for double precision). We will return to this point later on, but the difficulty is essentially not with the smallest number we can store but with the next biggest number greater than 1 we can store. That number has exponent 0 and mantissa $(1.000\ldots01)_2$ where we put in as many zeros as we can fit before the final 1. Allowing 1 sign bit, we have 23 places in total and so we fit 22 zeros. That means the number is

$$1 + \frac{1}{2^{23}} = 1 + \frac{2^7}{2^{30}} = 1 + \frac{2^7}{(2^{10})^3} \cong 1 + \frac{127}{(10^3)^3} = 1 + \frac{1.3 \times 10^2}{10^9} = 1 + 1.2 \times 10^{-7}$$

[3]In this discussion some details are not fully in accordance with everything that goes on in practice, where there is a facility for somewhat smaller posoitive numbers with fewer digits of accuracy in the mantissa. These details are fairly well explained at `http://accu.org/index.php/articles/1558`

(An accurate calculation gives $1 + 1.19209290 \times 10^{-7}$ as the next number bigger than 1 that computers can store using the commonly used IEEE standard method.)

A consequence of this is that we cannot add a very small number to 1 and get an accurate answer, even though we can keep track of both the 1 and the very small number fine. For example the small number could be $2^{-24}$ or $2^{-75}$), but we would be forced to round $1+$ either of those numbers to 1.

We can get a similar problem with numbers of different magnitude than 1. If we look at the problem relative to the size of the correct result, we get a concept called *relative error* for a quantity.

**5.12 Relative Errors.** The idea is that an error should not be considered in the abstract. [An error of 1 milimetre may seem small, and it would be quite small if the total magnitude of the quantity concerned was 1 kilometre. Even if the total magnitude was 1 metre, 1 milimetre may well be not so significant, depending on the context. But if the measurement is for the diameter of a needle, then a 1 milimetre error could be huge.]

If we measure (or compute) a quantity where the 'true' or 'correct' answer is $x$ but we get a slightly different answer $\tilde{x}$ (maybe because of inaccuracies in an experiment or because we made some rounding errors in the calculation) then the *error* is the difference

$$\text{error} = x - \tilde{x} = (\text{true value}) - (\text{approximate value})$$

Normally we don't worry about the sign and only concentrate on the magnitude or absolute value of the error. In order to asses the significance of the error, we *have to compare it to the size* of the quantity $x$.

The *relative error* is a more significant thing:

$$\text{relative error} = \frac{\text{error}}{\text{true value}} = \frac{(\text{true value}) - (\text{approximate value})}{\text{true value}} = \frac{x - \tilde{x}}{x}$$

It expresses the error as a fraction of the size of the thing we are aiming at. 100 times this give the percentage error.

**5.13 Example.** Suppose we use $\frac{22}{7}$ as an approximation to $\pi$. Then the relative error is

$$\text{relative error} = \frac{(\text{true value}) - (\text{approximate value})}{\text{true value}} = \frac{\pi - \frac{22}{7}}{\pi} = 0.000402$$

or 0.04%.

**5.14 Remark.** Another way to look at the idea of a relative error will be to consider the number of significant figures in a quantity. What happens with single precision floating point numbers is that we have at most 23 significant binary digits. When translated into decimal, this means 6 or 7 significant digits. That means that a computer program that prints an answer $6543217.89$ should normally not be trusted completely in the units place. (The 7 may or may not be entirely right and the .89 are almost certainly of no consequence.) That is even in the best possible case. There

may also be more significant errors along the way in a calculation that could affect the answer more drastically.

If a computer works in double precision, then there is a chance of more significant digits. In double precision, the next number after 1 is $1 + 2^{-52} \cong 1 + 2.6 \times 10^{-16}$ and we can get about 15 accurate digits (if all goes well).[4]

**5.15 Linear approximation.** Here we just recall briefly the linear approximation formula. It applies to functions $f = f(x)$ (of a single variable $x$) with a derivative $f'(a)$ that is defined at least at one point $x = a$.

The graph $y = f(x)$ of such a function has a tangent line at the point on the graph with $x = a$ (which is the point with $y = f(a)$ and so coordinates $(a, f(a))$). The tangent line is the line with slope $f'(a)$ and going through the point $(a, f(a))$ on the graph.

We can write the equation of the tangent line as

$$y = f(a) + f'(a)(x - a)$$

The linear approximation formula says that the graph $y = f(x)$ will follow the graph of the tangent line as long as we stay close to the point where it is tangent, that is keep $x$ close to $a$. It says

$$\boxed{f(x) \cong f(a) + f'(a)(x - a) \qquad \text{for } x \text{ near } a}$$

The advantage is that linear functions are easy to manage, much more easy than general functions. The disadvantage is that it is an approximation.

**5.16 Condition Numbers.** We can use linear approximation to understand the following problem.

Say we measured $x$ but our answer was $\tilde{x}$ and then we compute with that to try to find $f(x)$ (some formula we use on our measurement). If there are no further approximation in the calculation we will end up with $f(\tilde{x})$ instead of $f(x)$. How good an approximation is $f(\tilde{x})$ to the correct value $f(x)$?

We assume that $\tilde{x}$ is close to $x$ and so linear approximation should be valid. We use the linear approximation formula

$$f(x) \cong f(a) + f'(a)(x - a) \qquad x \text{ near } a$$

with $x$ replaced by $\tilde{x}$

$$f(\tilde{x}) \cong f(a) + f'(a)(\tilde{x} - a) \qquad \tilde{x} \text{ near } a$$

and then $a$ replaced by $x$

$$f(\tilde{x}) \cong f(x) + f'(x)(\tilde{x} - x)$$

[4] A more precise calcualtion than this rough one gives the result $1 + 2.2204460492503131 \times 10^{-16}$.

So the final error

$$\text{(true value)} - \text{(approximate value)} = f(x) - f(\tilde{x}) \cong f'(x)(x - \tilde{x})$$

Notice that $x - \tilde{x}$ is the error in the initial measurement and so we see that the derivative $f'(x)$ is a magnifying factor for the error.

*But* we are saying above that relative errors are more significant things than actual errors. So we recast in terms of relative errors. The relative error in the end (for $f(x)$) is

$$\frac{f(x) - f(\tilde{x})}{f(x)} \cong \frac{f'(x)}{f(x)}(x - \tilde{x})$$

To be completely logical, we should work with the relative error at the start $\frac{x-\tilde{x}}{x}$ instead of the actual error $x - \tilde{x}$. We get

$$\frac{f(x) - f(\tilde{x})}{f(x)} \cong \frac{xf'(x)}{f(x)}\frac{x - \tilde{x}}{x}$$

or

$$\text{relative error in value for } f(x) = \frac{xf'(x)}{f(x)}(\text{relative error in value for } x)$$

Thus the relative error will be magnified or multiplied by the factor $\frac{xf'(x)}{f(x)}$ and this factor is called the *condition number*.

In summary

$$\boxed{\text{condition number} = \frac{xf'(x)}{f(x)}}$$

**5.17 Examples.** (i) Find the condition number for $f(x) = 4x^5$ at $x = 7$.

$$\frac{xf'(x)}{f(x)} = \frac{x(20x^4)}{4x^5} = \frac{20x^5}{4x^5} = 5.$$

So in this case it does happens not to depend on $x$.

(ii) Use the condition number to estimate the error in $1/x$ if we know $x = 4.12 \pm 0.05$.

If we take $f(x) = 1/x$ we can work out its condition number at $x = 4.12$:

$$\frac{xf'(x)}{f(x)} = \frac{x\left(\frac{-1}{x^2}\right)}{\frac{1}{x}} = \frac{\frac{-1}{x}}{\frac{1}{x}} = -1$$

This means it does not depend on 4.12 in fact.

Now our initial value $x = 4.12 \pm 0.05$ means we have a relative error of (at most)

$$\frac{\pm 0.05}{4.12} \cong \pm 0.012$$

The relative error in $f(4.12) = 1/(4.12)$ is then going to be about the same (because the condition number is $-1$ and this multiples the original relative error). So we have, using $\tilde{x} = 4.12$ as our best approximation to the real $x$,

$$f(\tilde{x}) = \frac{1}{\tilde{x}} = \frac{1}{4.12} = 0.242718$$

and this should have a relative error of about 0.012. The magnitude of the error is therefore (at worst) about $(0.012)(0.242718) = 0.0029$ or about 0.003. So we have the true value of

$$\frac{1}{x} = 0.242718 \pm 0.003 \text{ or, more realistically } 0.243 \pm 0.003$$

(no point in giving the 718 as they are not at all significant).

(iii) $f(x) = e^x$. Condition numbers at $x = 10/3$ is $xf'(x)/f(x) = xe^x/e^x = x = 10/3 = 3.333\dot{3}$. So if we use $\tilde{x} = 3.33$ instead of $x = 10/3$ we would have a relative error to begin with

$$\text{relative error in } x = \frac{\frac{10}{3} - 3.33}{\frac{10}{3}} = 0.001$$

(that is an error of 0.1%). If we now compute $e^{3.33}$ while we ought to have computed $e^{10/3}$ we will have a relative error about 10/3 times larger (the condition number is 10/3, or roughly 3). So we will end up with a 0.3% error. In other words, still quite small.

If instead we were working with $x = 100/3$ and we took $\tilde{x} = 33.3$, we would have the same initial relative error

$$\text{relative error in } x = \frac{\frac{100}{3} - 33.3}{\frac{100}{3}} = 0.001$$

but the condition number for $e^x$ is now $x \cong 33$. The error in using $e^{33.3}$ where we should have had $e^{100/3}$ will be a relative error about 33 times bigger than the initial one, or $33 \times 0.001 = 0.033$. This means a 3.3% error (not very large perhaps, but a lot larger than the very tiny 0.1% we began with.

In fact $e^{33.3} = 2.89739 \times 10^{14}$ and 3.3% of that is $9.56137 \times 10^{12}$.

(iv) For $f(x) = x^2 - 10x + 11$, find the condition number at $x = 10$.

Answer: $\frac{xf'(x)}{f(x)} \mid_{x=10} = \frac{100}{11} \cong 9$.