Mobile Systems

Todd Davies

May 9, 2015

Introduction

Now that the mobile telephone has evolved into a powerful computer, the mobile dimension of computing is a vital part of Computer Science. This unit will give insights into many issues of mobile systems, including wireless communication networks, the processing of speech, music and other real-time signals, the control of bit-errors and maximising battery life. The techniques and software which underlie commonplace applications of mobile computing systems, including smart-phones, tablets, laptop computers, MP3 players and GPS satellite navigation, will be addressed.

Aims

Computing is becoming increasingly mobile. This unit will give insights into the issues of mobile systems, covering mobile communications, real-time signals such as speech, video and music, codecs, and maximising battery life.

- Commonplace examples of mobile computing systems: - mobile phones; - MP3 players; - laptop computers; - PDAs; - GPS satellite navigation.
- Real-time signals
- Analogue and digital signals; - time and frequency domain representations; - sampling, aliasing, quantization; - companding; - real-time computation.
- Coding, decoding and compression
- GSM speech coding; - MP3 music, JPEG image and MPEG video coding & decoding; - error correcting codes; - communications coding schemes.
- Mobile communication
- Transmitting real-time information over wireless networks; - principles of cellular and ad-hoc networks; - Coding of multimedia signals - to increase the capacity of radio channels; - to minimise the effect of transmission errors.
- Maximising battery life
- May be addressed at many levels including: - chip design; - signal coding and processing; - medium access control; - transmit power control.

Contents

1 Course intro

A smartphone is a mobile phone running a mobile operating system, with advanced capabilities with regard to computing power and connectivity. They are much more advanced than traditional feature phones, and have lots of features, including cameras, multimedia functionality, GPS, touch screens etc.

There are three main mobile operating systems in use today:

Android
Founded in 2003 by Andy Rubin and backed by Google. It's mostly free and open source, and holds a very strong position in the market.

iOS
Introduced in 2007 by Apple, this is a closed source operating system. The first iOS phones were very groundbreaking in terms of their technology, and were the first to feature touch screens, which are now ubiquitous in the market.

Blackberry, Symbian, Palm OS etc could be listed here too.

Windows Phone
Version seven was released in 2010, previous versions were terrible (imho).

1.1 What's in a smartphone?

Modern smartphones are jam packed with technology, containing cameras, multimedia, GPS, high resolution touch screens, motion sensors, bluetooth, RFID, NFC and even more stuff besides. They let you talk over multiple different networks (2G, 3G, 4G etc), and access data using the same methods.

Of course, the millions of apps available for them is also a massive attraction.

Smartphones Operating Systems need to be very advanced in order to step up to the tasks required of them by users. They need to be multitasking, to run lots of different apps, and interface with the typical hardware a normal computer might use (DMA, standard (ish) input devices etc). However, they also need to have real-time elements in, since they must be able to drive sound, IO, radio communications, digital signal processing etc

2 Signals in mobile systems

Signals such as speech and music arrive at the device as physical, analogue quantities that vary in a continuous manner over time. If we plot a graph of voltage over time, then we get a waveform for that signal. We can convert analogue signals to *discrete time signals* by sampling them at set intervals (discrete points in time). This produces a list of numbers from $-\inf$ to $\inf$.

2.1 Generating waves

In order to create a wave with a period of T seconds, you can use the formula:

$$y = sin(\frac{2\pi x}{T})$$

Of course, since $f = \frac{1}{T}$, the frequency of the wave is the reciprocal of the time period.

2.2 Sampling waves

We can work out the frequency of a wave, by finding how many complete cycles it undergoes in one thousand samples, and multiplying that by the sample rate. For example, if a wave has ten

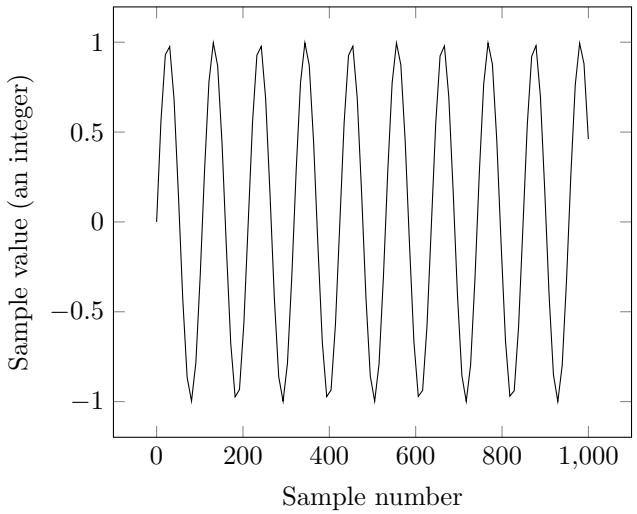

Figure 1: 1000 samples of an analogue sin wave of frequency $300Hz$ sampled at $30,000$Hz.

cycles from 1000 samples at $30,000Hz$, then its frequency will by $\frac{10}{1000} \times 30,000 = 300Hz$. This is visible in Figure 2.2.

Not all waves are so easy to analyse, rarely will a wave be at just one frequency, instead, they are usually 'noisy' and will be composed of many waves added together. Many waves (that aren't just noise) will have a discernible frequency that you can extract. The 'extra' waves on top of this frequency aren't necessarily bad, they might add harmonics, or texture to a sound.

2.2.1 The Sampling Theorem

Also known as *Nyquist criterion*.

The Sampling Theorem states that if a signal has all of its spectral energy below BHz, and is sampled at FHz, where $F \geq 2B$, then it can be reconstructed *exactly* from the samples and nothing is lost.

In other words, if you are sampling a signal at *less than half* of the maximum frequency of the signal, then you won't be able to fully reconstruct the signal from the samples, and you will get distortion.

Since music is usually sampled at 44.1kHz, we can accurately sample music with frequencies up to around 22kHz. Speech rarely has frequencies above 3.4kHz, so the sample rate for it can be much lower, as we will see later on in the course.

2.2.2 Aliasing

If we sample at less than twice the frequency of the wave, then we will get distortion, known as aliasing. The first lab on this course relates to aliasing. Even if we don't want frequencies higher than half our sample rate, we need to filter them out anyway, since otherwise, we will get aliasing when we sample them.

To be precise, if a wave of frequency f is sampled at a frequency of below $2f$ (lets call this F, then the sampled output will be a wave of frequency $F - f$.

For example, if we sample a 6kHz wave at a frequency of 10kHz, when we will get a wave of frequency $10 - 6 = 4$kHz.

This is bad, because if you have harmonics in a musical note that are higher than half the sampling frequency, then these harmonics will be out of tune post sampling, and will go down when it's supposed to go up.

It may be hard to work out exactly why aliasing occurs, but Figure 2, showing a 6kHz wave sampled at 8kHz makes it easier to see:

2.3 Decibels

Sound can be measured in decibels, which is a logarithmic ratio. The formula is as follows:

$$\text{dB} = 10 \times log_{10}\left(\frac{s_1}{s_2}\right)$$

If s_1 is twice as loud as s_2, then it will be $10 \times \log_{10}(2) = 3$dB louder than s_2.

The following table shows the power ratio ($\frac{s_1}{s_2}$) against the equivalent decibels:

Power ratio	Decibels
$\frac{1}{2}$	-3
1	0
2	3
4	6
10	10
100	20
1000	30
10^5	50
10^{10}	100

3 Encoding and storing signals

There are a variety of different ways to encode sound. The two main types of coders for talking over the phone are *waveform* coders, and *parametric coders* (sometimes called vocoders).

Waveform coders
Waveform coders 'operate on' the sound waves directly, and aim to change the wave so that it is transmitted in an optimal way. They are simple to understand and implement, but can't achieve *really* low bit rates. They aim to preserve the exact shape of the wave.

Parametric coders
Parametric coders try and model human speech by exploiting how we produce sound with our vocal chords and mouth shape. They don't try and preserve the exact wave shape, but instead try and describe *perceptually significant features* as sets of parameters. This is more complicated and harder to implement, but achieves lower bit rates.

Techniques that fall into both types of coder will be discussed in this section.

Dynamic range is the ratio between the loudest sound we can hear, and the most quiet sound we can hear.

Humans can hear frequencies between 20Hz and 20kHz, and have a dynamic range of 120dB. We can represent six decibels per bit of information using uniform quantisation (see section 3.2), so we need about $\frac{120}{6}$ bits to properly represent the sample.

3.1 Bitrates in different settings

Sometimes, 20 bits per sample is too much, so different bit rates are used in different applications:

CD's
Unfortunately, 20 bits per sample was too much for CD's (you wouldn't be able to fit enough songs on each CD), so instead, 16 bits item per sample was adopted as the standard. In order to 'lose' these four bits, we make the quiet bits of the songs louder (since we probably wouldn't be able to hear them anyway).

As a consequence of this, a song on a CD plays at a rate of $16 \times 44100 \times 2 = 1411$kbit/s.

Landlines

Landlines are band-limited to a range of frequencies from 50Hz to 3.4kHz. Obviously, this is much, much less than the dynamic range that humans can hear, but we only usually talk in frequencies covered by the range provided. This means that the sound will lose its 'naturalness' but not intelligibility (supposedly).

In practice, sometimes, vowel sounds like 'f' and 's' can be mixed up.

Band limiting is when a sound is Fourier transformed, and frequencies above or below a certain frequency are stripped.

The sample rate for telephone quality speech is 8kHz, with 8bit*s*. This gives a bitrate of 64kbit/s. In order to use just 8bit per sample, we need to use non-uniform quantisation (like in the (second?) lab), such as μ-law quantisation.

Mobile Telephones

Mobile phones use a bit rate that changes based on factors such as the signal quality, but has a minimum value of 4.75kbit/s, and a maximum value of 12.2kbit/s. AMR encoding (see Section **??**) is used to achieve such low bitrates.

3.2 Quantisation

When we've sampled a wave, if we leave all of our readings as floating point numbers, then we could be using a lot of space to store each sample. We could slightly reduce the quality of our sound by *quantising* it. This is when you map the continuous values onto a range of integers, where the range of integers spans a power of two (so you can use as few bits as possible).

You can see that the third wave in Figure 3 has been quantised into five integers, since there are five discrete values in the waveform $(0, 1, 2, 3, 4)$ (requiring four bits), and there will be an extra bit to indicate the sign. If whoever made the diagram was trying to be as efficient as possible, they might have had either one less value (so that the numbers could be represented in three bits), or made full use of the four bits for the value, and used values from $0 - 7$.

Quantisation produces noise (known as quantisation noise), since errors are introduced in the process. If there are many bits per sample (e.g. 16), then this error will be small, but if you only used say, five bits, then the error would be noticeable. As a consequence of this, we have to work a trade-off between storage capacity or bandwidth and quality.

There are two types of quantisation, uniform and non-uniform:

Uniform quantisation

A simple type of quantisation, each binary number represents a voltage, where incrementing the binary representation will correspond to an increase in the voltage of ΔV, called the *quantisation step-size*.

It might be hard to decide on a value for Δ, because different sounds will span different ranges of amplitude as shown in Figure 4.

One solution to this, is to adjust the value of Δ on a sample-to-sample basis. This uses up extra bandwidth though, so is not a preferred solution.

Non-uniform quantisation

Non-uniform quantisation is when the quantisation step-size is not the same between different quantised values, as evidenced in Figure 5.

Non-uniform quantisation is implemented like so:

1. Apply accurate uniform quantisation.
2. Apply a 'compranding' formula, such as μ-law.
3. Uniformly quantise again using fewer bits.
4. Transmit.
5. Reverse the process of steps three and two with an 'expander'.

A 'comprander' will increase the value of small samples, and decrease the value of large samples, and an 'expander' just does the reverse. Even though we only apply two uniform quantisations, the comprander and expander mean the overall effect is one of non-uniform quantisation.

Companding is just a coding technique similar to compression. It doesn't increase the quality of the sound or anything (though it might do if the alternative was to use the same number of bits with uniform quantisation.)

3.2.1 Quantisation error

Quantisation Error is produced when we quantise a wave, and is random in the range of $\pm\frac{\Delta}{2}$. Since the error is random, the error is heard as white noise, and is spread evenly across all frequencies. The Mean Square Value (MSV) of the noise, is $\frac{\Delta^2}{12}$.

We can calculate the Signal to Quantisation Noise Ratio (SQNR), which is how loud the signal is in comparison to the noise (hence it is given in decibels - see Section 2.3). To do this, we use the formula:

$$\text{SQNR} = 10\log_{10}\left(\frac{\text{MSV of signal power}}{\text{MSV of noise}}\right)$$

We can use this formula to calculate the dB/bit for sending telephone data (speech, text etc):

$$\text{SQNR} = 10\log_{10}\left(\frac{A^2/2}{\Delta^2/12}\right)$$

The maximum amplitude of a sine wave that has been quantised with a step size of Δ is $\frac{2^{bits}}{2}\Delta$

$$= 10\log_{10}\left(\frac{\left(\frac{2^{bits}}{2}\Delta\right)^2/2}{\Delta^2/12}\right)$$

$$= 10\log_{10}\left(\frac{\left(2^{2\times bits-2}\Delta^2\right)/2}{\Delta^2/12}\right)$$

$$= 10\log_{10}\left(\frac{\left(\frac{2^{2\times bits}}{4}\Delta^2\right)/2}{\Delta^2/12}\right)$$

$$= 10\log_{10}\left(\frac{\frac{2^{2\times bits}}{8}\Delta^2}{\Delta^2/12}\right)$$

$$= 10\log_{10}\left(\frac{\frac{2^{2\times bits}}{8}}{1/12}\right)$$

$$= 10\log_{10}(1.5\times 2^{2\times bits})$$

$$= 10\log_{10}(1.5) + 10\log_{10}(2^{2\times bits})$$

$$\approx 1.8 + 10\log_{10}(2^{2\times bits})$$

$$\approx 1.8 + (6\times bits)$$

This only strictly applies for uniformly quantised sine waves, but also holds fairly well for speech and music.

If an 8 bit uniform quantisation scheme is designed for loud talkers (so they will use all eight bits), then it will have a SQNR of $(6\times 8)+1.8 = 49.8$. If another talker is much quieter, and talks 30dB quieter, then the samples will be encoded using only 3 bits:

$$\begin{aligned} 6\times bits - 1.8 &= 49.8 - 30\\ 6\times bits &= 48 - 30\\ 6\times bits &= 18\\ bits &= 3 \end{aligned}$$

Since the quantisation noise for three bits is much lower (19.8dB) for the quietly talking person, they will probably be able to hear the noise over the phone.

Remember, since the SQNR is calculated by $\frac{\text{MSV of signal power}}{\text{MSV of noise}}$ it has an inversely proportional relationship with the amount of noise. I.e. will go up as the noise decreases.

For the loudest CD quality (16 bit) music, the maximum SQNR value is 97.8dB, whereas for the most quiet sounds, it is -22.8dB (which means the noise is louder than the actual sound). Obviously this isn't ideal, so we need to apply DRC (Dynamic Range Compression) improve the balance.

3.3 Differential encoding

A sample of audio isn't just random data, and since the data represents a wave, consecutive values are likely to be relatively close together. If that is the case, then maybe we could encode the data as the difference between one sample and the next. If the differences are easier to transmit than the raw data, then we could save resources such as bandwidth. Such an encoder is shown in Figure 6.

PCM stands for Pulse Code Modulation, and is basically the same as what we mean by quantisation.

This is known as Adaptive Differential PCM, and can be made to work at bitrates of $16-32$kbit/s. However, for use in mobile telephones, we need a bit rate that is at least four times lower than this...

3.4 Linear Predictive Speech Coding

Voiced speech is normal speech, unvoiced speech is whispering. If normal speech is sampled, then some of it will be classed as unvoiced, since only vowel sounds actually require the use of vocal chords (try it!).

LPC is a type of parametric encoder. Working on the principle that voiced speech is created by the vocal chords opening and closing at frequencies that change over time, the coder can send the fundamental frequency of the noise as well as parameters (called coefficients here) modelling the shape of the mouth over the network to recreate the speech at the receiver.

The sender derives the parameters at a rate of 50hertz, which include how loud the speech is, the coefficients modelling the vocal tract, and the fundamental frequency of the sound. Figure 7 shows how these parameters are used to reconstruct the speech at the receiver.

3.4.1 LPC-10

LPC-10 was a coder once used by the military that had a bit rate of 2.4kbit/s. Each 20ms frame has the following components:

37 bits	Ten filter coefficients
1 bit	Voiced/unvoiced decision
8 bits	Gain (the amplitude)
8 bits	Fundamental frequency

MIPS is Millions of Instructions Per Second

This made for a total of 56bit per frame. Although it's easy to understand and implement, and requires little processing power (according to Wikipedia[1], it requires 20MIPS and 2kB of RAM).

3.4.2 Codebook Excited LPC

Codebook Excited LPC (CELP) is a way of further reducing the bandwidth used by LPC. The principle is that there are a number of pre-configured parameters that the receiver and sender have stored in a 'codebook', and the sender just sends the index of the one that is most similar to the sound that it is trying to produce.

It finds the most similar sample using 'analysis by synthesis', which involves trying all the entries in the codebook one by one, and sending whichever is the closest to the one we want.

CELP is used in commercial mobile telephones, and is utilised by the **Adaptive MultiRate** (AMR) coder for rates at 12kbit/s or lower.

[1] `http://en.wikipedia.org/wiki/FS-1015`

3.5 Comfort noise

When a two way conversation is going on, and neither party are speaking, nothing will be played through the speakers to either party (since we don't transmit quiet sounds since that wastes power). To mitigate this, the phones will generate and insert 'comfort noise' which is psudorandom background noise that sounds like whatever is going on at the other end of the phoneline. Each phone will determine if its owner is silent, and transmit the characteristics of the background noise using very few bits if they are so that the other phone can recreate a similar background noise for the other person.

3.6 Error correction, detection & prevention

Mobile systems are affected by errors in the transmission of data, often in the form of noise affecting radio reception. There are many ways of avoiding/mitigating errors, the most important ones looked at in this course are; Forward Error Correction (FEC) and error detection and retransmission (Automatic ReQuest).

3.6.1 Forward Error Correction

We can build redundancy into the transmission by appending check bits or some other form of coding that will bloat the size of the transmission, but result in less errors.

We can use block coding or convolutional coding to implement this. **Block coding** is when you process the data in blocks (well duh), which requires you to encode the whole block at the transmitter and decode it again at the receiver. You can't use a block unless it's been fully decoded. **Convolutional coding** on the other hand can yield usable bits soon as just a few of the transmitted bits arrive at the receiver. Convolutional coders treat data as a stream, whereas block coders deal with data in chunks.

Here are some examples of the above:

Repeating bits:
: We could simply send bits a few times, instead of just once and take the mode of them at the receiver. If we sent three bits for every bit, then if there was one bit error, then it would be fine since we could take the majority of the received bits, and the one error bit would be ignored.

  The number of repeats must be an odd number, otherwise we could end up with a half and half split.

Parity
: As a simple (and not very effective) way of detecting bit errors, we could append a parity bit to the end of blocks. The parity bit would be `0` if the block had an even number of ones, and `1` if it had an odd number of ones. If we have an n bit number, we can calculate the parity using $n-1$ `XOR` operations; $b_0 \oplus b_1 \oplus \cdots \oplus b_n$.

  When the receiver gets the bits, the parity of all of them (including the parity bit) must be computed. If the parity is `1`, then some bit error definitely occurred, whereas if it is `0`, then the data *may be* correct (but there could be two or more errors as well).

  Parity checking is a block code, since you need to wait for the whole block to arrive before checking the parity.

Hamming distance
: The Hamming distance between two binary numbers is the number of bits that are different. This is obtained by `XOR`ing the two numbers together and counting the number of ones. This is useful if we want to know how many bit errors it takes to convert one number to another.

You have a set list of code words that you can use, each with a minimum hamming distance from it to any other. Then, you can detect and even errors by comparing your received code to the allowed ones.

Figure 3.6.1 shows an implementation.

Hamming codes are block codes, since you need the whole block to calculate the distance between the received block and the ones in the codebook. It is common practice to introduce check bits onto code words to make the distance larger.

The notation for Hamming Codes is $(m + r, m)$, where m is the number of message bits, and r is the number of check bits. The r bits are appended to the end of the code words, and are derived from the value of the m bits. For example, with a $(7, 4)$ hamming code, we could make:

- $r_0 = m_0 \oplus m_1 \oplus m_2$
- $r_1 = m_0 \oplus m_1 \oplus m_3$
- $r_2 = m_0 \oplus m_2 \oplus m_3$

This would give a *syndrome table* (i.e. correction table) of:

r_0	r_1	r_2	Correction to
0	0	0	None
0	0	1	r_2
0	1	0	r_1
0	1	1	m_1
1	0	0	r_0
1	0	1	m_2
1	1	0	m_3
1	1	1	m_0

Interleaving:
Since bit errors in radio links often occur in bursts (e.g. caused by a car turning on), we could transform one dimensional blocks of data into two dimensional matrices, and transfer them column by column. This way, if there is a 'bursty' bit error, then it will affect multiple rows, but the bit error correction (such as) the repeating method, or hamming codes could detect the error since only one or two bits may be wrong, not all of them.

3.6.2 CRC checks

A Cyclic Redundancy Check is a block code for detecting bit errors. If we find the value of the number we're transmitting in decimal and divide it by a number such as seven, we can append the remainder onto the message. Then at the receiver, we can divide the received bits again and if we get a different remainder, then we know that a bit error has occurred.

If the bit errors happened to add or subtract the number that you were dividing by, then the remainder would be the same and you would not be able to detect the errors. If we make the number large, then this is unlikely.

Real CRC checks use *Galois Field Arithmetic*[2] to do the maths, since binary numbers can be expressed as polynomials:

Summing:
To calculate the sum of two binary numbers, we can just XOR their bits. Subtracting is the same: N

$$1001 \oplus 0111 = 1110$$

Long division:

[2] `http://en.wikipedia.org/wiki/Finite_field_arithmetic`

There are three CRC generators used in practice:

- CRC-8-ATM: $x^8 + x^2 + x + 1$
- CRC-16-IBM: $x^{16} + x^{15} + x^2 + 1$
- CRC-32-IEEE: **Quite long...**

A generator of order r can detect all error bursts of length $\leq r$.

3.6.3 Convolutional Coders

If we are processing streams of data, then we could have a 'rolling parity check'. When we encode, the previous n bits are `XOR`'d together to produce another stream of bits. We can then interleave the new parity stream with the normal stream.

At the decoder, the bits are valid if each of the parity bits (i.e. not the normal ones) is the `XOR` of the previous three normal bits. If the bit sequence is invalid, then we can select the valid sequence with the minimum Hamming distance to the received sequence and therefore have error correction.

With sequences of around 8 bits, that is fine, since there would be 256 valid sequences of 16 bits long each, however, for longer sequences (e.g. 1024 bits), then this would not be feasible.

Convolutional coders that encode bit streams are called *rolling parity* coders.

A soft decision decoder is where if we're unsure as to what the value of a bit is, then we could use previously known probabilities to determine what we could pick.

3.6.4 Miscellaneous FEC stuff

Well thought out use of FEC techniques in mobile systems increases the energy efficiency and effectiveness of the system. Transmitting at higher power is one way of reducing errors in the system, but 'loud' signals also cause lots of interference over a wider range and, of course, uses lots of battery. FEC lets us reduce the transmission power, yet overcome the bit errors that come as a result, which allows us to re-use different frequencies and reduce power consumption.

We could increase the bit rate by not encoding in binary, but using any 2^n number of states. If we could send the values $\{0, 1, 2, 3\}$, then we could send two bits per pulse, effectively doubling our bandwidth. This is limited by the noise in the channel.

The **Shannon-Hartley Law** says that the channel capacity C is equal to:

$$C = B \log_2(1 + \frac{S}{N})\text{bit/s}$$

Where B is the bandwidth in Hz, and $\frac{S}{N}$ is the signal to noise power ratio.

4 Mobile networks

Since their inception, the desired use of cellular networks has changed from being solely about transmitting speech, to transmitting textual data, and now arbitrary packets. This has required them to evolve from 'circuit switched networks', where (at a fundamental level) a physical connection must be made between wires to facilitate data transfer, to 'packed switched networks'.

This evolution is really interesting, since circuit switching originated from landline phones, but packet switching was from digital networking such as Ethernet. As the technology has progressed, cellular data has moved towards the Internet's way of doing things.

However, mobile phones have moved on from being single feature devices, merely letting you call other users. Now they are *smart*phones, and this requires them to use a plethora of different technologies to be able to fulfil user's demand; GPS, bluetooth, radio etc to name a few.

4.1 Cellular networks

One reason why mobile phones can work so well is that they take advantage of *spatial multiplexing*. This divides an area into small areas called cells, each having a frequency band. The cells are different sizes based on their expected usage rates (cells will be more dense in areas with more demand). Frequency bands are re-used when a cell with the same frequency band is far enough away, and because of this, mobile phones must take care not to transmit their signals too loud and pollute the spectrum for nearby cells.

Cells are typically $0.1 - 35$km in diameter. Obviously small cells are better since they will handle more users per unit area (assuming cell size is independent of user capacity), but users will need to transmit at a lower power (so the neighbouring cells using the same frequency aren't affected). This is actually a good thing sometimes, since it uses less battery, and we can use FEC to mask bit errors.

Breaking the spectrum up into frequencies, and mapping that onto physical cells is a nice concept - and it works very very well in reality, but it does have some drawbacks; for example, what happens when a user moves into a different cell while on the phone? Thankfully, the system supports transparent and seamless handovers as the user moves from cell to cell.

4.1.1 The evolution of cellular technologies

Cellular networks have gone through four iterations to date, with a fifth one coming in the future:

0G:
The zero'th generation of phones were merely radio telephones. They didn't use cells at all.

1G (1983):
This is when cellular mobile was created, all signals were analogue.

2G (1991):
Data is introduced, though it is very slow. In 1998, GPRS was introduced, which brought speeds from $56 - 114$kbit/s and in 2003, EDGE was introduced which has speeds of up to 384kbit/s. These are called 2.5 and 2.75G respectively.

3G (2001):
This introduced better speech and faster data. 3G has multiple revisions; 3.5G (2007) is HSPDA with speeds of $1.8 - 7.2$Mbit/s download and 384kbit/s upload, 3.75G (2010) is HSPA+ which has speeds of 56Mbit/s download and 22Mbit/s upload, and finally 3.9G which is still being created and launched, and is related to technologies such as WiMax and LTE.

4G (2011):
The specification for 4G is 1Gbit/s download and 100Mbit/s upload. However, the technology doesn't meet that yet (even now!). Henceforth, companies marked 3.9G technologies as 4G since they 'aspire' to meet the target speeds.

5G:
Due to be launched around 2021, though we've still not met the 4G speeds yet...

This isn't the whole story though. It wasn't just the marketing names and the speeds that changed, they also used completely different multiplexing techniques to let multiple users share the spectrum:

All of these use spatial multiplexing with cells except 0G.

0/1G:
Frequency Division Multiplexed Access (FDMA) was used, which is where each transmitter is given a different carrier frequency.

2G (in europe - GSM):
Time Division Multiplexed Access (TDMA) was used so that each transmitter is given a regular time slot to send data in.

3G/2G (america only):
Code Division Multiplexed Access (CDMA) - each transmitter uses the same frequency band, but different codes are used to send data.

4G:
Orthognal Frequency Division Multiplexed Access (OFDMA) is used by 4G, which utalises

multi-input/multi-output (MIMO) antennae to transmit on multiple frequencies at once. Data is sent in packets.

4.1.2 Multiplexing in detail

TDMA:

CDMA:

OFDMA:

Beam Division Multiple Access (BDMA):

5 Frequency Domain Processing

Instead of processing waves, and parts of waves, how about we turn them into frequencies and process those instead? Frequencies are a lot easier to deal with in many ways since they are easier to understand and smaller to store.

5.0.3 Sinusoids

A sinusoid is a sine wave delayed by D seconds, and is given by the formula:

$$y = M \times cos(2\pi F(t - D))$$

Figure 5.0.2 shows two sinusoids; one where $D = 30°$ and another where $D = 0$.

Figure 5.0.2 is a simple sine wave, but any wave (such as the one in Figure 5.0.2) that is periodic can be found to have a fundamental frequency of $\frac{1}{T}$, where T is the period of the wave. A recording of a voice, or musical instrument may be found to have a similar waveform to that of Figure 5.0.2 as long as we 'zoom in' to a segment short enough that any gradual change in frequency is negligible. This is called **psudo-periodicity**.

5.0.4 Fourier Series

Any periodic wave of frequency F can be written as:

$$\begin{aligned} x(t) = A_0/2 &+ A_1 cos(2\pi Ft) + B_1 sin(2\pi Ft) \\ &+ A_2 cos(2\pi 2Ft) + B_2 sin(2\pi 2Ft) \\ &+ A_3 cos(2\pi 3Ft) + B_3 sin(2\pi 3Ft) \\ &+ \dots \end{aligned}$$

This is sometimes called its 'cos and sin' form, but we could re-write it as:

$$x(t) = \frac{M_0}{2} + \sum_{k=1}^{\infty} M_k \cos(2\pi(kF)t + \phi_k)$$

Here, each time the summation iterates, we are adding the next harmonic of the sound. The fundamental frequency F is the first harmonic when $k = 1$.

Because the harmonics get higher and higher, we can't sample the whole series because otherwise the frequency of the harmonics will start to become higher than half the sampling frequency and we will get aliasing (see Section 2). In that case, we only go up to $k = \frac{N}{2} - 1$, where $F(\frac{N}{2} - 1) \leq \frac{Fs}{2}$. If we took Fs samples every second, then the formula would be:

$$x(n) = \frac{M_0}{2} + \sum_{k=1}^{\frac{N}{2}-1} M_k \cos\left(2\pi(kF)\frac{n}{Fs} + \phi_k\right)$$

$$\text{Where } n = 0, 1, \ldots, \text{size}(x)$$

The more harmonics we compute, the closer we can get to reconstructing the wave from them afterwards.

5.0.5 Discrete Fourier Transform (iDFT)

The DFT converts samples of length n into $n/2 - 1$ samples and a 'dc term'. They are arranged in a length n array, but the second half is the same as the first half but reversed. The 0^{th} element of the array is the DC term, but the i^{th} element is the i^{th} harmonic.

The algorithm to convert from the time domain into the frequency domain is *N-point DFT*. To do the opposite, we can use the *N-point inverse DFT* algorithm.

DFT is used to:

- Perform spectral analysis on a signal to find out what components are present.
- Convert into the frequency domain before applying signal processing (and converting back again afterwards). For example, you could filter out noise from a sine wave, since you could remove all frequencies with an amplitude below a certain level.

5.1 Wav files

Signals are stored in many different ways. One 'easy' format, is the `.wav` format. It is just a list of binary numbers, each representing a single value of the wave at a discrete time.

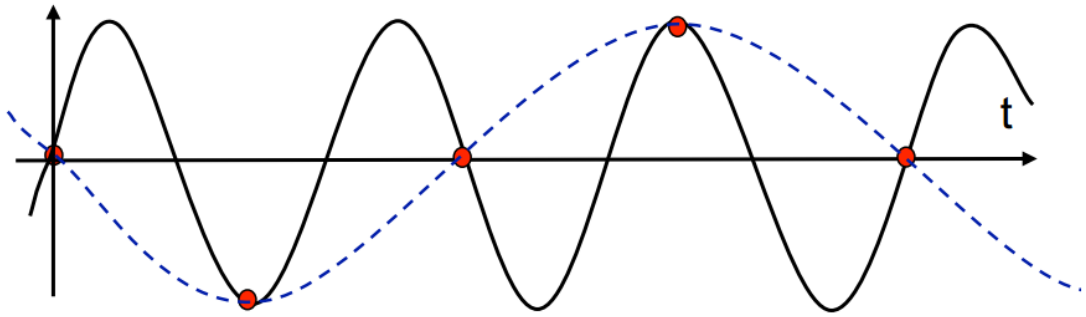

Figure 2: A 6kHz wave sampled at 8kHz has a post-sampling frequency of 2kHz

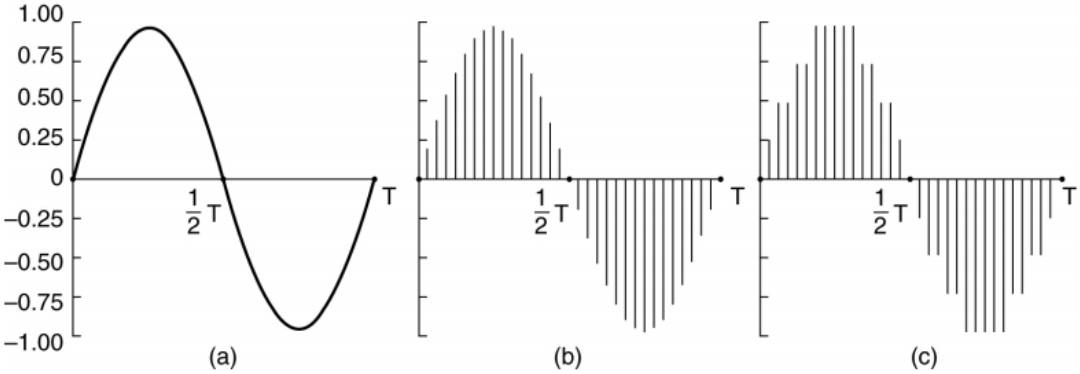

Figure 3: An analogue wave (a), after its sampled (b), and once it's been quantised (c)

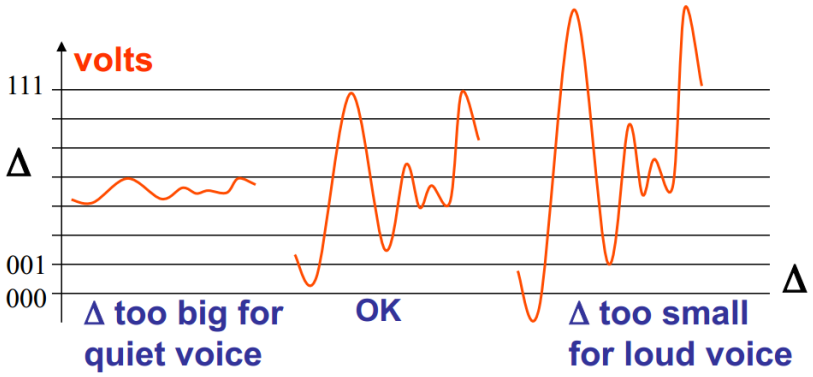

Figure 4: This waveform shows how a a bad choice for Δ can adversely affect the quality of the sound, when it is undergoing uniform quantisation.

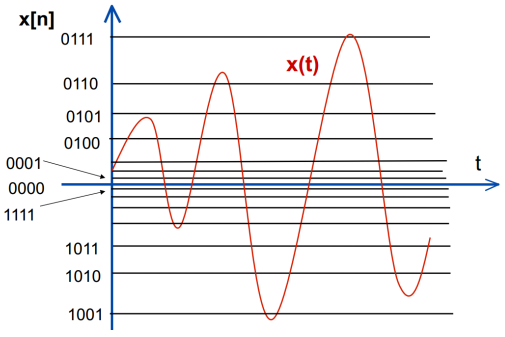

Figure 5: An example of non-uniform quantisation.

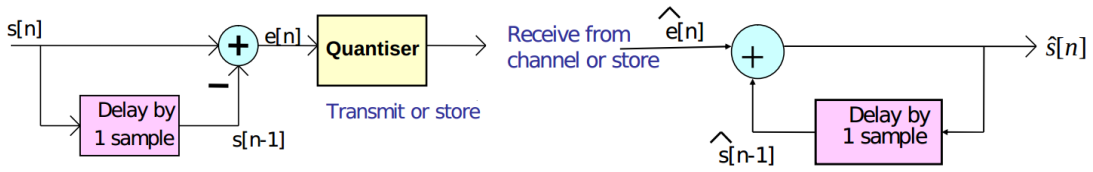

Figure 6: An example of how a differential encoder might both encode and decode audio samples.

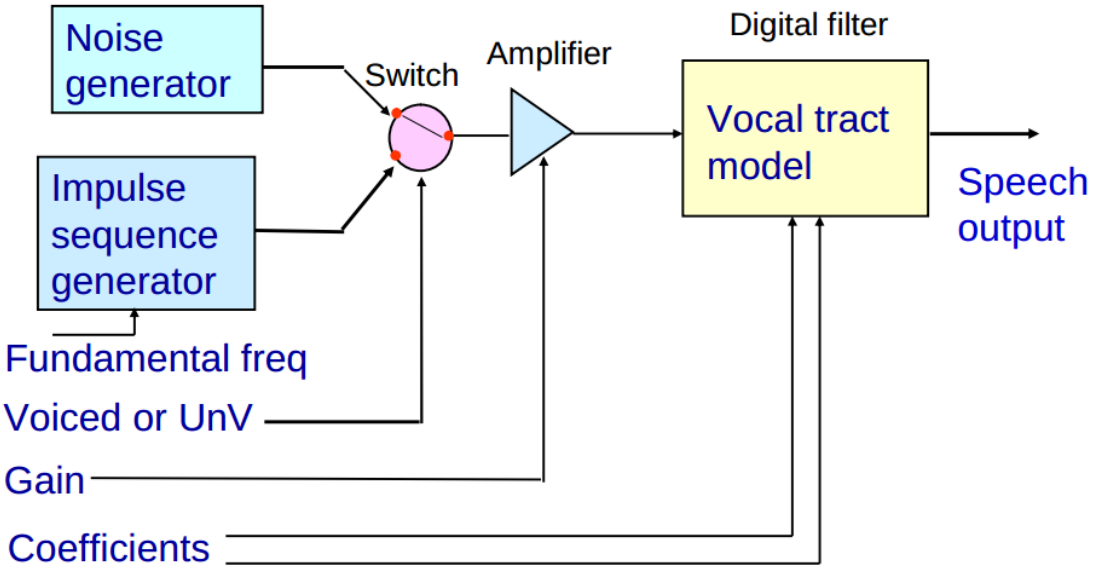

Figure 7: An implementation of a Linear Predictive Speech Decoder

```
public class HammingCode {

  private final HammingNumber[] numbers;
  int correctionDistance = Integer.MAX_VALUE;

  public HammingCode(HammingNumber[] numbers) {
    this.numbers = numbers;
    for(int i = 0; i < numbers.length - 1; i++) {
      for(int j = i + 1; j < numbers.length; j++) {
        int distance = numbers[i].hammingDistance(numbers[j]);
        if(distance < correctionDistance) {
          correctionDistance = distance;
        }
      }
    }
  }

  public HammingNumber correct(HammingNumber input) {
    for(HammingNumber number : numbers) {
      int distance = input.hammingDistance(number);
      if(distance == 0 || distance < correctionDistance) {
        return number;
      }
    }
    return null;
  }

  public static class HammingNumber {

    private final boolean[] number;

    public HammingNumber(boolean[] number) {
      this.number = number;
    }

    public int hammingDistance(HammingNumber other) {
      int distance = 0;
      for(int i = 0; i < number.length; i++) {
        if(number[i] != other.number[i]) distance++;
      }
      return distance;
    }

    public String toString() {
      StringBuilder sb = new StringBuilder();
      for(boolean b : number) sb.append(b ? "1" : "0");
      return sb.toString();
    }
  }
}
```

Figure 8: A Java implementation of HammingCodes

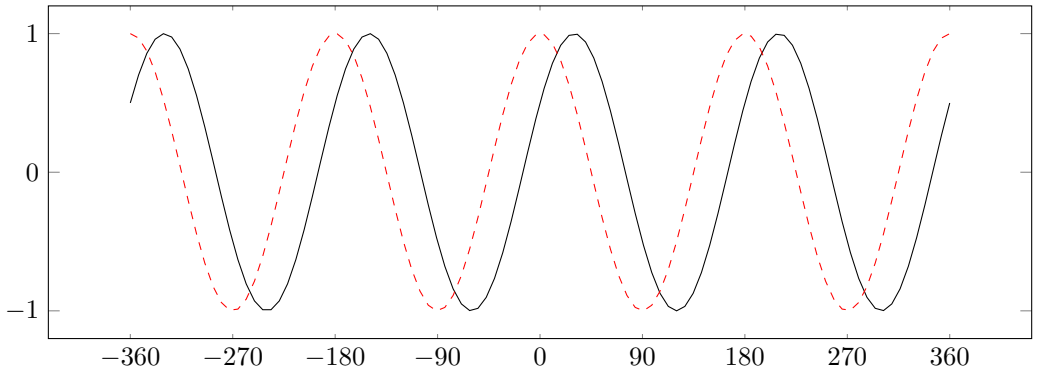

Figure 9: 1000 samples of a sinusoid where $D = 30°$ (black) and where $D = 0$ (dotted red).

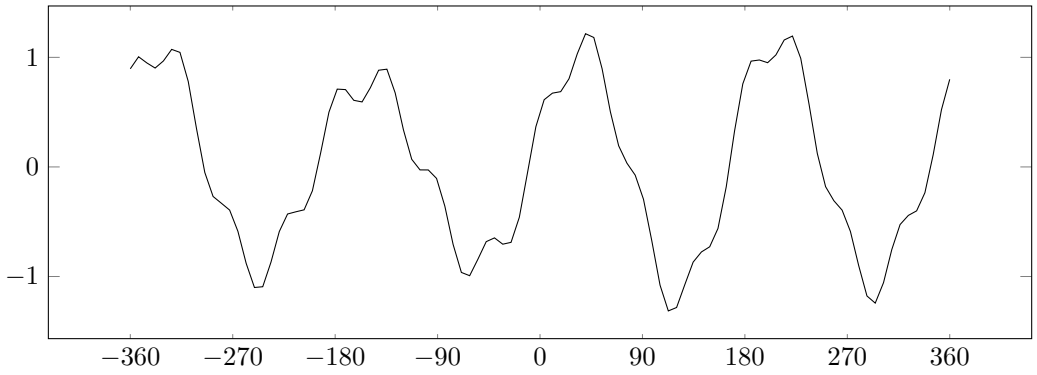

Figure 10: A complex wave with a frequency of $\frac{1}{180} = 0.00\overline{5}$Hz