How to Read a QR Code

(An algorithm perspective)

Steven Mitchell, Ph.D.

steve@componica.com

Me and My Motivations

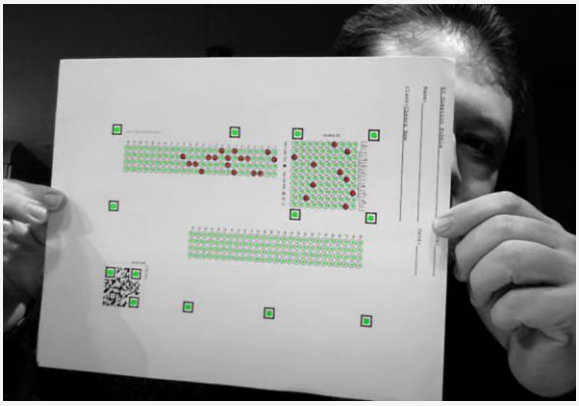
Serial entrepreneur focused on Computer Vision and AI startups.

In 2010 I co-founded a startup called Lightning Grader to create and scan student assessments in real-time using Computer Vision.

In 2015 it was acquired by Illuminate Education and used by millions of students.

The platform used QR Codes to both identify documents and also provide the initial alignment of documents.

This required me to implement many parts of a QR Code decoder myself, and in this talk I'll explain three problems I was pondering while creating a decoder.... Finding then, aligning them, and error correction.



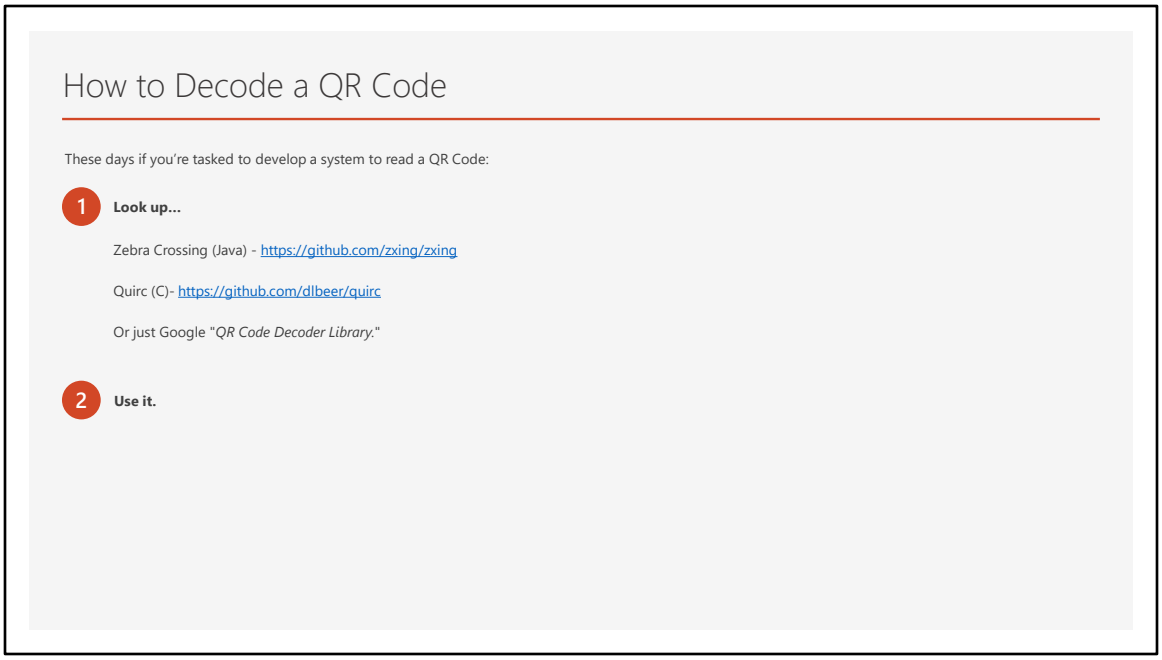
https://www.youtube.com/watch?v=2Xi8HjZe6Eg

- Hello, my name is Steve Mitchell and I'm a serial entrepreneur focused on CV and AI startups, and this is a very short 10-minute talk on how QR codes are decoded.
- Motivation: I once co-founded a startup called Lightning Grader that scanned and graded student assessments in real-time using Computer Vision which was acquired by Illuminate Education almost a decade ago. QR Codes were the initial first step both to identifying the document, but also providing the initial alignment of the document, and back then, I had to implement many parts of a QR decoder myself.
- As for the rest of the stuff like how to accurately align and grade documents real-time running on school-issued 2012 Chromebook… well that's left for a different TED talk.

This talk covers three problems I was pondering while creating a decoder.

1. How do you find a QR Code in an image?
2. Aligning a QR Code.
3. How do you correct for errors?

Reading the data is straight forward so I won't cover that part because this is meant to be a short talk.



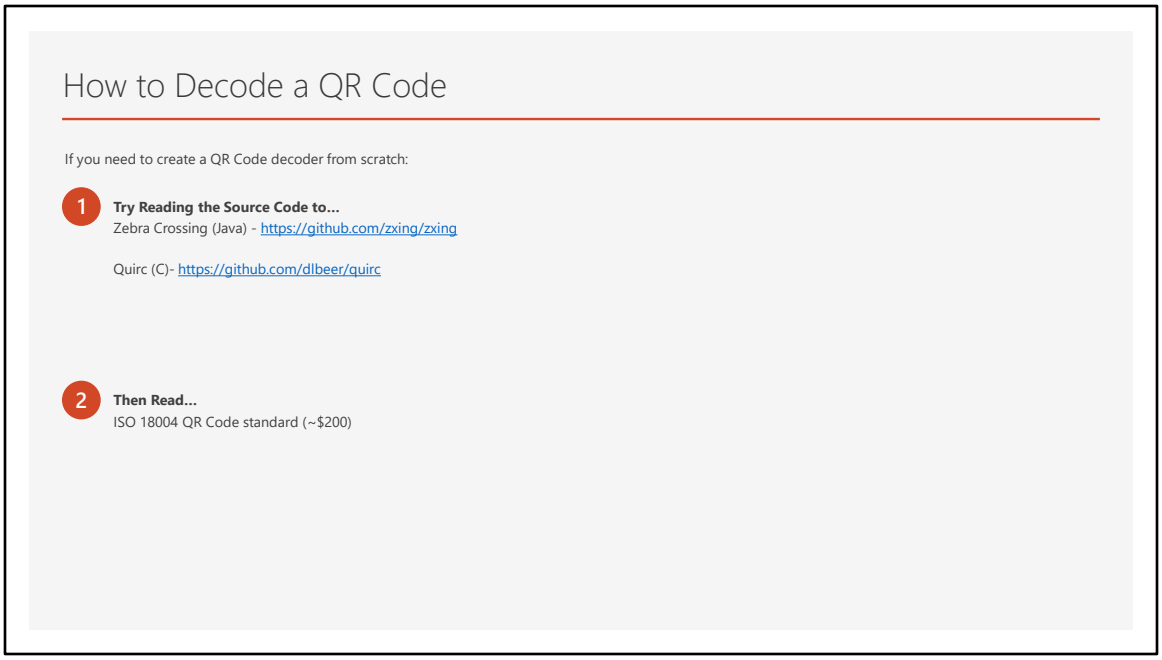
Today, if someone asked me to implement a QR code decoder, I'd tell them to go to GitHub and consider these libraries:

- Zebra Crossing (Java) - https://github.com/zxing/zxing
- Quirc (C)- https://github.com/dlbeer/quirc
- Or just Google "QR Code Decoder Library."

And that's it. The End. Thank you.

The End.

And that’s it. The End. Thank you.



But if you need to implement a QR code reader from scratch well…

- Read the source code to Zebra Crossing and Quirc.
- Buy a copy of ISO 18004 QR Code standard (~$200).

How to Read of QR Code

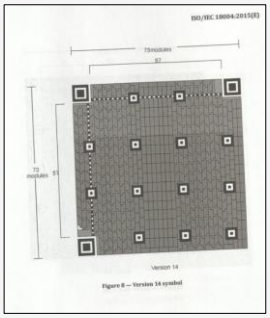
If you need to create a QR Code decoder from scratch:

1. **Try Reading the Source Code to…**
   Zebra Crossing (Java) - https://github.com/zxing/zxing

   Quirc (C)- https://github.com/dlbeer/quirc

2. **Then Read…**
   ISO 18004 QR Code standard (~$200)

   Do not Google "ISO 18004 FileType:pdf" or you will accidentally find a free copy of it!



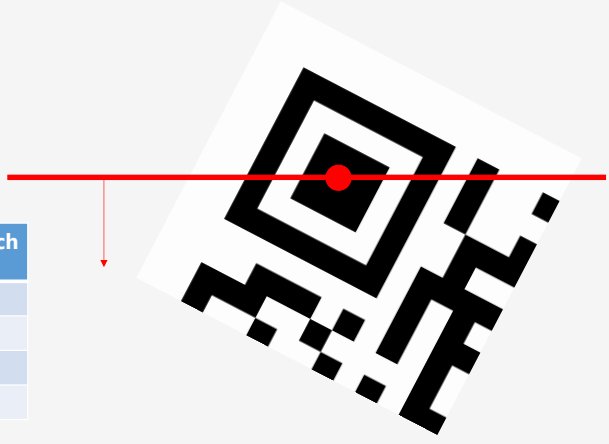
But if you need to implement a QR code reader from scratch well…

- Read the source code to Zebra Crossing and Quirc.
- Buy a copy of ISO 18004 QR Code standard (~$200).
  - Do not Google "ISO 18004 filetype:pdf" or you will accidentally find a free copy of it.

Step 1: Locating the Finder Patterns

Find the finder patterns…

Black	White	Black	White	Black	Match
0.76	0.99	2.15	0.88	2.23	
1.06	0.93	3.05	0.91	1.06	Yes
0.85	3.78	1.33	0.62	0.42	
1.59	1.12	1.07	1.89	1.34	

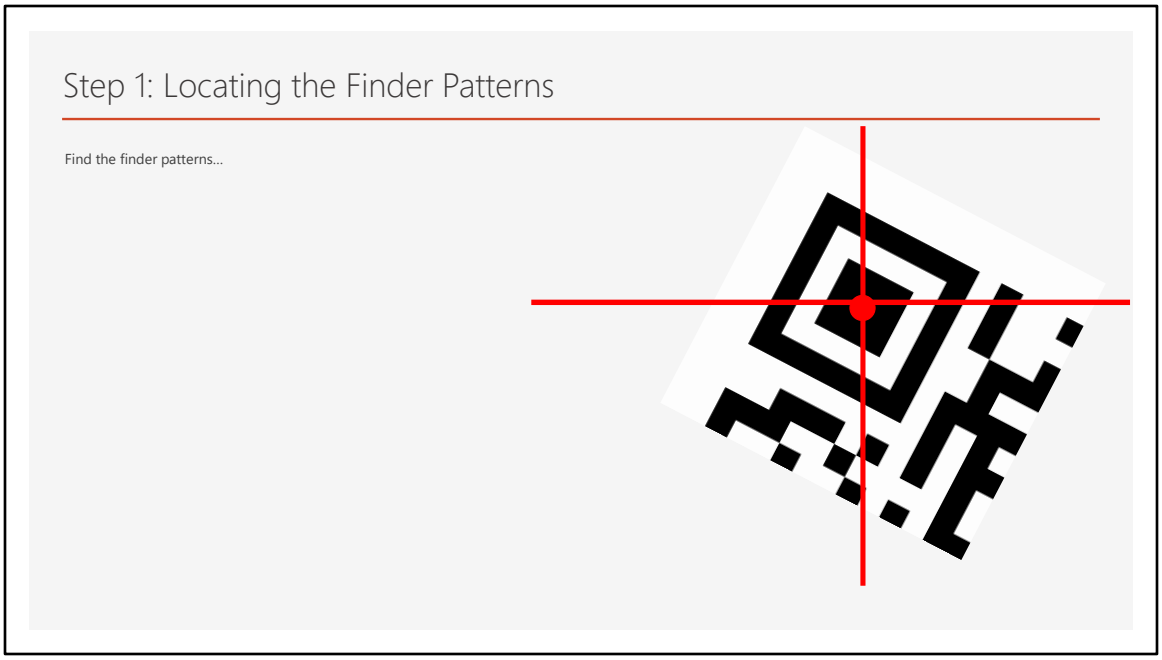


Step 1: Finding the finder patterns – Seems like Magic, but like all magic tricks, once you understand how they work, it's surprisingly simple!

For each row of your binary image: scan, count, and remember the number of consecutive black, white, black, white, black pixels.

If you see a ratio of 1, 1, 3, 1, 1 within a +/- error, that could be a marker pad. Above are actual snippets of pixels from the original-sized image on the right. I've manually counted the pixels and computed the ratios in Excel and as you can see, the second entry fits the 11311 ratio and is indeed a line is over a finder pattern.

Since you know the starting and ending pixel locations of the 11311 pattern, you can compute its exact center and size estimate by computing the mid-point of the start and ending of the pattern.



Once a potential finder pattern is found, scan vertically and look if it's in the center of another 1, 1, 3, 1, 1 ratio pattern. Compute the center of the vertical pattern, that's the best guess center of the finder pattern.

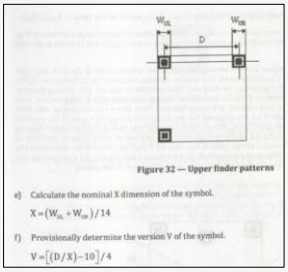
Step 1: Locating the Finder Patterns

Find the finder patterns...

You can reduce false positives by adding additional criteria like checking that the horizontal and vertical ratios are square-like and checking if the center square is somewhat square-like as well.

Step 2: Locating the QR Code

1. Look at triplets of finder patterns and guess the version number, and shape.
2. Try to decode the version info sections.
3. If decoding is successful, you'd found a QR code. Error correction codes will minimize false positives.
4. Else try a different set of finder patterns and repeat. Computers are fast enough to brute force possible combinations.



Step 2: So right now, you have a set of x, y coordinates of finder patterns and their relative pixel size, basically a bunch of dots. You still don't know where a QR Code is located in the image.

According to the ISO document, it suggests the following to figure out which finder dots are part of a QR code:

1. Look for sets of two/three finder patterns and guess the shape and layout based on pixel distances (shown in center image).
2. Check if they're part of a QR Code by decoding these green version info blocks. If the decoding isn't garbage, you probably found a QR Code. Because the version info block includes BCH error correction code, the code acts like a checksum so you'll know with high probability if the green rectangles are legit or random noise.
3. If they're garbage, repeat going back to step 1 with a different set of finder patterns.

Additional Notes:

- Computers are fast enough to brute force all possible combinations quickly.
- You have hints like:
    - The two furthest points are probably the hypotenuse of the right-angle triangle. That helps with guessing which of the three points are which.
    - At least two finder patterns must be within an estimated distance from your current finder pattern. If not, reject it because you need three points for a QR code.

Step 3: Align the QR Code to a Square Bit-Plane

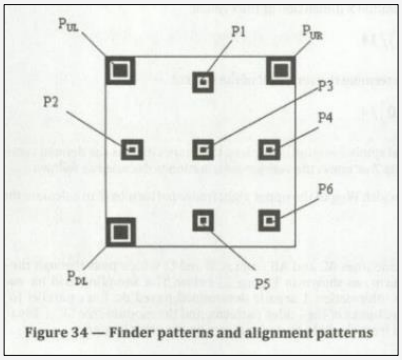
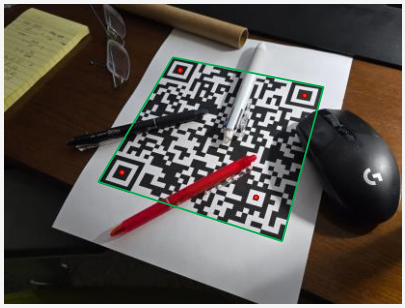


Figure 34 — Finder patterns and alignment patterns

Step 3: To align an image of a CR Code to a bit-plane for decoding, we use a perspective transform. This transform require 4 known points on an image, and so QR standard defines small alignment patterns of ratios 11111 to help dewarp the image. As the QR code gets larger, you need more alignment patterns due to errors caused by slight warping of the original image, lens distortion, and the high tolerances required to read such small pixels. Please try to avoid using large QR codes as it often very problematic to dewarp and decode even with error correction codes. I'm looking at you What's App, Samsung's Smart Switch app, and a couple metro ticketing systems.

Step 3: Align the QR Code to a Square Bit-Plane

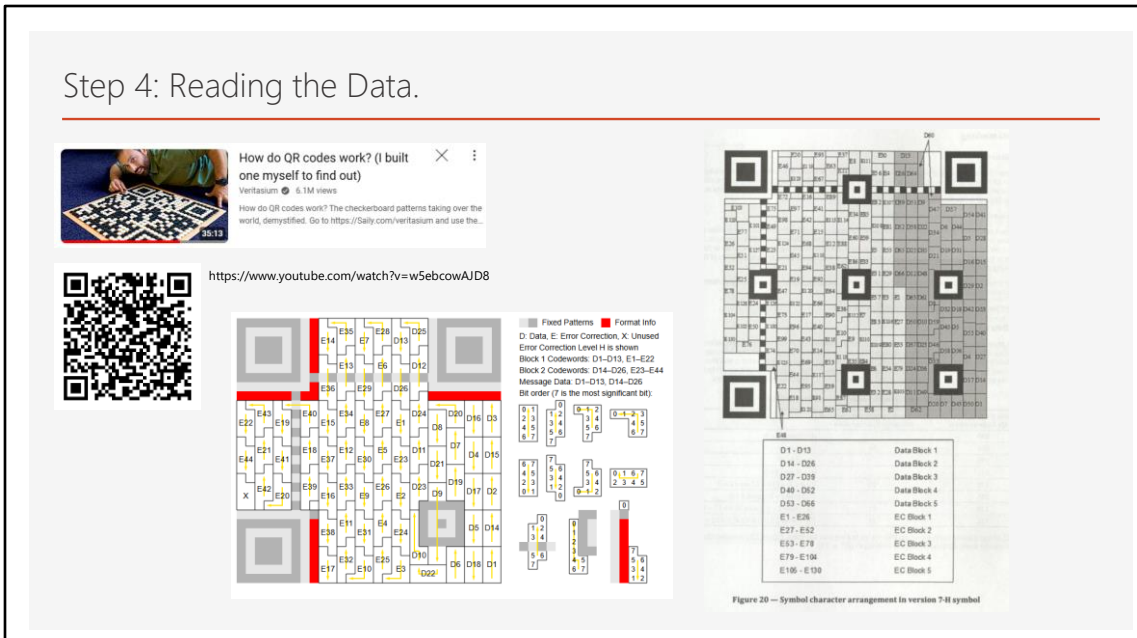


```python
import cv2
import numpy as np

paper_image = cv2.imread('./qr_code.jpg')
pts1 = np.float32([[678,212],[1287,331],[379,714],[1118,951]])
pts2 = np.float32([[68,68],[955,68],[68,955],[955,955]])

M = cv2.getPerspectiveTransform(pts1,pts2)
dewarped_image = cv2.warpPerspective(paper_image,M,(1024,1024))
cv2.imwrite('./dewarped_qr_code.png', dewarped_image)
```

In this example I have three finder patterns and an alignment pattern. I computed a perspective transform to dewarp this image to a square bit-plane for decoding. Here I've manually noted the pixel locations of the four corners and using this Python script to align it to the original QR Code using a perspective transform for comparison.

Despite the three pens occluding the bit-plane, a decoder can reliably read this data because of error correction.

You might notice some bending near the top of my dewarped image. Such bending can be caused by things like lens distortion or paper/image curl. Again, avoid large QR codes.



Step 4: Not going to cover reading the data because it's just following the standards and I don't much time. The Youtuber Veritasium (of course he posted his video a few days after I agreed to do this talk) and other have created a nice explanation on this. Basically, it's about reading 8-bit blocks of your message Tetris-ed around the alignment sections with masking to make the dots look random.

Additional explanation:

Not going to cover reading the data because it's just following the standards and I don't much time. You read these format and version information bits to determine the masking, size, and shape of the QR code. This would have taken additional slides to cover and explain.

- After aligning to the correct size, you demask the bit plane. The purpose of masking / demasking is to make the data look more random, avoid long segments of all black or all white, and also to prevent accidentally creating false alignment patterns. Both of these improve detection and decoding.

- Reading bytes involves decoding 4- and 8-bit blocks in a zig-zag pattern through the data region. There are rules to how you Tetris your way around the QR Code to place data, but the ISO standard doesn't seem to explicitly state the exactly locations expecting you to figure that part out. I believe it's because of the use of 4-bit blocks used to note the data type (numbers, alphanumeric, raw bytes, Kanji) and these 4-bit blocks can appear anywhere based on the length of your message.
- Once your data payload is complete, the rest of the space is filled with error correction code to protect the data.

Step 5: Reed Solomon Error Correction

These are some great resources for understanding Reed-Solomon:

- https://tomverbeure.github.io/2022/08/07/Reed-Solomon.html
- https://en.m.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders

Suppose I want to encode the message [1, -5, 3, 2] and protect it with two more symbols.

- Represent it as a polynomial with two extra terms for error correction symbols.
  $p(x) = x^5 - 5x^4 + 3x^3 + 2x^2 + ax + b$, a and b are to be solved.
- Create the following equation:
  $1x^5 - 5x^4 + 3x^3 + 2x^2 + ax + b = q(x)(x-1)(x-2)$.
  The $(x-1)(x-2)$ are called the generator
- Solve for a and b by dividing the generator by the polynomial without a and b terms:

$\frac{(1x^5 - 5x^4 + 3x^3 + 2x^2)}{(x-1)(x-2)} \rightarrow q(x) = x^3 - 2x^2 - 5x - 9$ with remainder $(18 - 17x)$

- The remainder is subtracted from p(x) and it defines a and b:

$$1x^5 - 5x^4 + 3x^3 + 2x^2 + 17x - 18 = (x^3 - 2x^2 - 5x - 9)(x-1)(x-2)$$
$$1x^5 - 5x^4 + 3x^3 + 2x^2 + 17x - 18 = q(x)(x-1)(x-2)$$

Transmits the message embedded in this modified polynomial:

[1, -5, 3, 2, 17, -18] The 17, -18 protects the message. If any digit is modified, I'll know and I can fix it.

Step 5: Reed-Solomon Error correction is what makes QR codes reliable and allow you to embed logos in the center of them. The two links are great resources for understanding Reed Solomon. The following example here is taken directly from the tutorial in the first link (I suspect Veritasium also referenced this example using he own values.)

Messages are represented as coefficients of polynomials with an additional number of unknown coefficients to be solved for (in this example, a & b). What I'm trying to do is modify the p(x) polynomial representing my data into the equation of form q(x)(x-1)(x-2)… (x-N) by solving for a & b. This is done by computing the remainder of p(x) / (x-1)(x-2)… (x-N) using polynomial long division and subtracting it from p(x). This subtracted remainder (17x -18) is the error correction data used to protect the original message.

I don't care what q(x) is once I've subtracted the remainder from p(x). It was defined

so I could modify the message polynomial, p(x), to have the form q(x)(x-1)(x-2) if factored out.

Step 5: Reed Solomon Error Correction

Checking a message

- Convert the transmitted message [1, -5, 3, 2, 17, -18] as a polynomial.

$$f(x) = x^5 - 5x^4 + 3x^3 + 2x^2 + 17x - 18$$

- Check if f(1) and f(2) are zeros. If so, the message is correct.

Correcting the message. (This algorithm only handles one error).

- Suppose the message was modified: [1, 7, 3, 2, 17, -18]

$$f(x) = x^5 + 7x^4 + 3x^3 + 2x^2 + 17x - 18$$
$$f(1) = 12, f(2) = 192$$

- Solve the polynomial for each term where x = 1 and x = 2.

$$\boldsymbol{k}x^5 + 7x^4 + 3x^3 + 2x^2 + 17x - 18 = 0$$

s = -11 for f(1) and -5 for f(2)

$$x^5 + \boldsymbol{k}x^4 + 3x^3 + 2x^2 + 17x - 18 = 0$$

s = -5 for f(1) and -5 for f(2) ← These two values match.

$$x^5 + 7x^4 + \boldsymbol{k}x^3 + 2x^2 + 17x - 18 = 0$$

s = -9 for f(1) and -21 for f(2)

$$x^5 + 7x^4 + 3x^3 + \boldsymbol{k}x^2 + 17x - 18 = 0$$

s = -10 for f(1) and -21 for s(2)

- The recovered message is
  [1, -5, 3, 2, 17, -18]

There are different ways of decoding and correcting a message with this scheme. The follow example is the simplest but only handles one error.

Checking a message:
[Walk thru the example on the slide.] Why does f(1) and f(2) equal zero if the message is correct? Because we modified the message polynomial to take the form q(x)(x-1)(x-2) when we subtracted the remainer (18-17x) on both sides, so if x=1 then q(x)*0*(-1) = 0 and if x = 2 then q(x)*1*0 = 0. If any of them are not zero it means the original message has been modified.

Correcting a message:
[Walk thru the example on the slide.] Why does it work? You know the equation must equal to zero if 'x' is either 1 or 2, and you're solving for k. It's a very easy to work out on paper since you're solve one single variable equation, and not solving multiple equation simultaneously. We also know the missing value k is at the same position for both equations therefore, k must be the same whether x is 1 or 2 if k is the missing value. That's how you simultaneously determine which value was scrambled and

what the original value.

There's actually a handful of methods for correcting a modified message. This example works if there's only one modified value, but it's easy to understand. Commonly used methods involve analyzing f(1), f(2) .. F(n) etc., which are known as syndromes [Insert multiple images of Disney's '*The Incredibles*' villain here for humor] These methods typically are split into steps, first step identifies which values were modified and the second step corrects those values. How they work is explained in the second URL in the previous slide.

Step 5: Reed Solomon Error Correction – Galois Fields

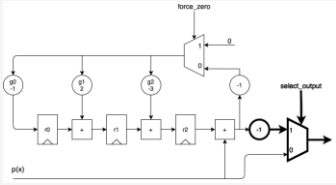
There's a key issue with the error correction example I just showed you:
- QR codes use bytes, not arbitrary large positive and negative numbers.
- Reed-Solomon coding was developed in the 1960s, a time when computers were slow.

The solution is to use the same algorithms but apply a different kind of numbers and algebra, known as Galois Fields, specifically GF(2^8), which operates over 256 values (0 to 255).

In Galois Fields, GF(2^8):
- Only numbers from 0 to 255 are valid, which happens to fit into bytes.
- Addition and subtraction are done with XOR operations, while multiplication, division, and exponentiation are handled with small lookup tables or shift registers and feedback logic. Cheap to create in hardware and solvable using 70s era computers.

This approach allows all the polynomial calculations to be implemented in digital circuitry, which enabled the Voyager space probes to communicate in the 1970s and made scratched CDs playable in the 1980s. Rad!

```
                                  12 da df
01 0f 36 78 40 ) 12 34 56 00 00 00 00
               ^ 12 ee 2b 23 f4
                  da 7d 23 f4 00
                ^ da a2 85 79 84
                     df a6 8d 84 00
                   ^ df 91 6b fc d9
                        37 e6 78 d9
```



[Walk through the slide.]

The main point here is the math in the prior slides is doable with only bytes instead of positive and negative potentially large integers if I use a different algebra system known as a Galois (gal-Wah) Field. Use the same algorithm, but use 0 to 255 as my numbers and redefining +, -, *, /.

Because a negative value is the same as it's positive counterpart in Galois math, computing an error correction code is simply computing this synthetic division problem with XORs being subtraction operation. The remainder 37 e6 78 d9 is the code that's appended to 12 34 56 creating 12 34 56 37 e6 78 d9. This is trivial to implement in digital logic circuits.

Who was Évariste Galois?

Galois Fields are critical for making Reed-Solomon a practical and useful algorithm. Never heard of Galois before, and I ended up in a Wikipedia rabbit hole learning the following…

Évariste Galois was a young mathematical genius with a deep interest in polynomials but unknown . He was also politically active during a turbulent time in France. At the age of 20, he was challenged to a duel, possibly due to his political beliefs or over a woman. Knowing he was likely to die, he spent the night before the duel writing letters to his friends and family, outlining his mathematical ideas and attaching three manuscripts with instructions to pass them on to mathematicians Gauss and Jacobi after his death.

The next morning, he was shot in the abdomen and died the following day. His last words were, “Don’t weep, Alfred (younger brother)! I need all my courage to die at twenty!”

About a decade later, his manuscripts were discovered. He had posthumously

founded a new branch of mathematics, now known as Group Theory - a field that is foundational to particle physics, chemistry, cryptography, and information theory.

If you were to remember anything from this talk: Remember, this guy died so you could add “googly bits” to your QR codes… and also satellite communication, playing scratched CDs/DVDs, have WiFi, read hard drives and flash drives, etc.

While researching how Reed-Solomon worked, I found myself often wondering what the world might have been like if Galois had lived?

The End.

The End. Thank you.