Estelle Bayer, Robert Lord 5/31/2017

# Encoding QR Codes in Python

Created in 1994 by automobile manufacturers in Japan, QR codes are perhaps the most well-known 2d barcode in use today. Beyond numerous applications in shipping, ticketing, payment, and other industrial logistics, they're used extensively in advertisements on billboards and magazines for consumers to scan with their smartphone. A combination of Reed-Solomon error correction; standard finder, alignment, and timing patterns; and masking techniques allow the barcodes to scan correctly even if parts are obscured or destroyed, or when read at an angle or in poor light. QR codes of various sizes and error correction levels exist, ranging from the tiny Version 1 that stores as little as 7 bytes, to the gigantic Version 40 that can store almost 3 kilobytes when set to the lowest level of error correction. In this paper, we describe our implementation of a QR encoder written from scratch in Python, show how polynomial-based techniques translate into code, and demonstrate how Reed-Solomon error correction and masking techniques make QR scanning so robust.

## Encoding the Data

Not counting Micro QR codes (which are not covered by this paper) there are 40 different "versions" of QR codes, which represent the 40 different possible sizes, ranging from 21x21 squares for Version 1 to 177x177 for Version 40. The version used also has implications beyond just the size — it determines a couple constants like remainder bit counts throughout the algorithm. For each version, there are also four different error correction levels to choose from. L (low) can correct up to 7% of incorrect data, M (medium) recovers up to around 15%, Q (quartile) up to roughly 25%, and H (high) up to about 30%. In each case, increasing error correction without increasing the version reduces the amount of data that can be stored.

QR codes support several different data modes that allow messages using a small subset of unicode to encode data more compactly. For instance, if only digits are being encoded, you can pack two digits into a single byte, vs the 1 digit/byte required by standard unicode. QR has a numeric digit-only mode, an alphanumeric mode, a raw byte mode, a kanji mode for Japanese characters, and several other more esoteric modes. In this paper, we'll focus on the alphanumeric mode, which encodes decimal digits, uppercase-only letters, space characters, and the symbols \$, %, $*$, +, $-$, ., /, and :.

The message of a QR code is encoded in a specific format that conveys information about the mode, length, and content of the message over a binary alphabet. We begin the encoded message with the alphanumeric mode indicator 0010. In Python, we can set this directly by assigning the String value '0010' to the instance variable designated to represent the mode indicator. In future implementations, a method for determining mode based on user-supplied input could be designed.

Following the mode indicator, the length of the message is encoded as part of the data stream. The possible size of a message is restricted based on version and error-correction level. In order to calculate the appropriate version programmatically, we write a method in our encoder called, appropriately, getVersion(), which takes no arguments independent of the user-supplied message and desired error-correction level.

```
maxMessageLength = messageLength
    errorVersion = self.versionDictionary[self.errorCorrection]
    while maxMessageLength not in errorVersion:
        maxMessageLength += 1
    version = errorVersion[maxMessageLength]
```

```
        self.version = version
        return version
```

This method iteratively references the maximum possible message length for every version and error correction level, increments the actual message length variable, and repeats until the message length variable is equal to some error correction level and version, returning the resulting version number. In this way, we find the version whose size is nearest to the actual size of our message. Using the version information, we can determine the appropriate length for the size indicator, convert the actual message length to binary, and pad it on the left with 0s to obtain a length-encoding of the appropriate size.

The final step of encoding the data stream is encoding the message itself. In any given mode, the input message is sectioned into consecutive pairs of characters. The value of the first character is then multiplied by 45 and added to the second character. 45 as a multiplier is easily derived from the size of the alphabet in our message: there are 45 alphanumeric characters. Thus, we can think of every pair as a unique two-digit number in a base-45 system. We then convert this number to its 11-digit binary representation, padding on the left with 0s as necessary. (11 bits is also derived from the 45-character alphabet, since the maximum possible decimal value for a pair is $44 \times 45 + 44 = 2024$, which is representable in 11 binary bits.) This process, of determining successive 11-bit integers, is easily translated to Python code.

$a_i = (45x_{2i} + x_{2i+1})_2$, where $a_i$ is the $i$th 11-bit word.

The message encoding is simply the concatenation of $k$ such terms, where $k$ is half the length of the message, with an additional 6-bit word concatenated for odd-length messages.

```
for i in range [0, len(self.originalData)//2]
```

```
firstValue = 45 * alphanumericValueTable.index(self.originalData[i])
secondValue = alphanumericValueTable.index(self.originalData[i+1])
binaryValue = bin(firstValue+secondValue)
paddingBits = 11 - len(binaryValue)
for bit in paddingBits:
  binaryValue = str(0)+str(binaryValue)
messageEncoding = messageEncoding + str(binaryValue)
```

The final odd-length word can be implemented through a similar block of code, preceded by an if statement which checks for odd parity in the message length.

In order to finish the encoding of the message, we add an appropriate number of terminal bits such that the terminator has length of no more than four and the overall message length does not exceed the maximum allowance for the determined version and error-correction level.

Now we are ready to format the entire message encoding. First, we concatenate the already-determined mode indicator, the character count, and the encoded message. Thence, we append sufficient 0s to make the overall message length divisible by 8, in order to facilitate division into bytes, to aid in error correction. Further, if the message does not fill the allotted space for the version and error-correction level, the binary representations of 236 and 17 are added as alternating pad bytes. The message string, separated into 8-bit bytes, can now be used to generate error-correction codewords.

## Code Error Correction

The requisite number of error-correction codewords necessary is determined by version and error correction level. Error correction words will be determined through a process reminiscent of syndrome decoding for cyclic codes. Because the message is broken into data codewords of 8 bits, the size of the vector space containing any given data codeword is $2^8$. As a consequence, we generate error-correction codewords over $GF(2^8) = GF(256)$. When taken modulo an irreducible polynomial, every element of this field can be represented as $2^n$ for some $0 \leq n \leq 255$.

Every number $2^n$ for $0 \leq n \leq 255$ has a unique value in GF(256) when put into a binary representation in $Z_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$. $x^8 + x^4 + x^3 + x^2 + 1$ is a irreducible polynomial, which means that there are no elements in $Z_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$ that multiply together to $x^8 + x^4 + x^3 + x^2 + 1$. Since the polynomial is irreducible, we know that $Z_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$ is a field, and therefore there must exist at least one primitive element, an element $p$ in the field such that $\forall e \in F \exists n \in Z$ such that $e \equiv p^n$. For $Z_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$, one of these elements is 2, which can be quickly proven via brute force using the Ruby script further down in this section.

A quick side note — the selection of this polynomial space for QR codes is arbitrary — any irreducible polynomial in $Z_2[x]$ of degree 8 would be appropriate. Finding irreducible polynomials is a computationally difficult problem, however since our field is quite small at just 256 elements, we can still brute-force all possible solutions.

```
num = 1
puts (256...512).select { |space|
  num = 1
  255.times.map { |i|
    num = (num & 0b10000000 != 0) ? (num << 1) ^ space : num << 1
  }.inject(0, :+) == 32640
}.to_a.map{|i| i.to_s(2)}
```

Running the Ruby code above shows that any of the polynomials 100011101, 100101011, 100101101, 101001101, 101011111, 101100011, 101100101, 101101001, 101110001, 110000111, 110001101, 110101001, 111000011, 111001111, 111100111, 111110101 form a field with primitive element 2, and would have been a perfectly adequate selection, but the specification just decides to use 100011101.

Thus, any data codeword in our message can be represented as some $2^n$ value. This allows us to conveniently generate error-correction polynomials for QR codes. Error correction codewords are identically the remainder when dividing a special generator polynomial by the polynomial whose coefficients have the values of the data codewords. These generator polynomials are generated by $(x - \alpha^0) \dots (x^n - \alpha^{n-1})$, where $\alpha$ is the primitive element of the field- in this case, 2- and $n$ is the desired number of error correcting codewords. Generator polynomials for QR codes are known quantities, although they can be easily calculated with a recursive function in Python, presented here as pseudocode, in order to minimize extraneous field-arithmetic handling.

```
def findGenerator(numberErrorCorrection):
  n = numberErrorCorrection
  if (n == 1):
    return 2**0*x**2 + 2**25*x**1 + 2**1*x*0
  else:
    return (x**n-2**(n-1))findGenerator(n-1)
```

Once this generator polynomial is obtained, we can use it to calculate our error-correction codewords, which are the coefficients of the remainder when the generator polynomial is divided by the message polynomial. Mathematically, this is done through polynomial long division. Algorithmically, it can be achieved through successive iteration of the following steps:

1. Multiply the generator by the appropriate scalar in $GF(256)$ such that its leading coefficient equals that of the message polynomial.
2. Subtract the scalar multiple of the generator and the message polynomial. This is the equivalent of a bitwise XOR operation.

This is equivalent to the remainder of polynomial long division because the operation XOR performed bitwise over $GF(2)$ is equivalent to subtraction. Thus, instead of multiplying the message by an appropriate term and subtracting it from the generator polynomial, we can multiply the generator by the multiplicative inverse of the aforementioned term and bitwise XOR the result with the message, setting the resulting difference to be the new message and repeating. Once more in pseudocode- in order to forgo the somewhat messy algorithms necessary to convert string data to polynomial or array data, a Python implementation of this process could operate as follows:

```
def findErrorCorrection(numberErrorCodewords):
  generatorPolynomial = findGenerator(numberErrorCodewords)
  messagePolynomial = self.dataEncoding
  for i in range[0, len(messagePolynomial)]:
    leadingTerm = messagePolynomial[0]
    dividend = leadingterm*generatorPolynomial
    messagePolynomial = dividend^messagePolynomial
    return messagePolynomial
```

Once this process has been repeated as many times as there are data codewords, we are left with a remainder polynomial whose coefficients we take as our error correction codewords.

## Structuring the Final Message

All versions of QR codes except for version 1 contain multiple blocks of data codewords words with corresponding blocks of error-correcting codewords. In structuring the final bit stream, we practice a process called "interleaving," wherein data codewords are selected from successive blocks in order. In other words, if $a_{i,j}$ is the $i$th word of the $j$th block of data codewords, the final message begins with $a_{0,0}a_{0,1}\dots a_{0,n}a_{1,0}a_{1,1}\dots a_{1,n}\dots a_{m,n}$, where $n$ is the number of blocks and $m$ the number of words in a block. The error-correcting codewords are interleaved in the same manner. The final message, comprised of both data codewords and error-correcting codewords, is presented as a complete, uninterrupted binary string. Additional remainder bits may be specified by the version of the code, and are added as 0s to the end of the string. This message is ready for visual encoding.

## Layout and Masking

Layout of the QR code begins with the placement of the three finder patterns, the grid of alignment patterns, and two strips of timing patterns that run between the finder patterns. The finder patterns are the three distinctive "bullseyes" in the three corners that give the QR code its distinctive appearance. Space for reserved metadata about the QR code is also reserved in a square around the finder patterns. The data bits from the previous encoding phase are then placed, starting in the bottom right corner and weaving up and down in a 2-square column, skipping over the already placed alignment and timing patterns, until stopping at one of the corners on the left. Once the data has been placed, the grid must be "masked". This is a mathematical manipulation of the data bits that attempts to decrease four factors that contribute to errors in visual reading: long strips of a single color that make identifying timing of a region more difficult, 2x2 clumps of a single color that likewise make reading the size of the clump more difficult, patterns that look similar to the finder patterns that would completely screw up scanning software attempting to find the bounds of the code, and codes with significantly more of one color overall than the other. Masks are some simple operation such as flipping the bits of every third column or every other row. Each of the eight masks is tested one-by-one, and the mask that reduces the four factors the most is selected.

Finally, the format string is inserted. This is a two bit indicator of the error correction level followed by a three bit indicator of the mask pattern used. These five bits are then used to generate ten error-correcting bits using, again Reed-Solomon error correction. The fixed generator polynomial $x^10+x^8+x^5+x^4+x^2+x+1$ is used for this, since we know we have exactly 5 bits as input and 10 bits as error correction output. Encoded as a binary vector, this generator polynomial is 10100110111. We divide the format string by this number to get the error correcting bits — simply right-padding the format string with ten zeros and then repeatedly applying the standard binary polynomial long division technique results in the ten error correction bits. The final format string before being inserted into the squares surrounding the finder patterns is the original 5-bit format string, followed by the 10 error correction bits, all XORed with the mask 101010000010010.

## Conclusion

QR codes are a highly reliable 2d barcode system. Reed-Solomon error correcting codes, visual patterns and masking come together to produce codes that are readable and redundant. The generator polynomial encoding system used by Reed-Solomon translates well into operations on binary numbers. It's a calculation system that isn't just easy to implement efficiently with standard bitwise operations, but also possible to put on embedded devices with limited memory and computation resources. This makes sense for an barcode designed in 1994, when computing resources were far more limited, but it's impressive that the system has stayed relevant and effective after over two decades of use and many revolutions in computing.

## Sources

Eby, Carolyn. “QR Code Tutorial - Thonky.com.” Thonky. Accessed May 31, 2017. http://www.thonky.com/qr-code-tutorial/.

“ISO/IEC 18004:2015 - Information Technology – Automatic Identification and Data Capture Techniques – QR Code Bar Code Symbology Specification.” Accessed May 31, 2017. https://www.iso.org/standard/62021.html.

Shoup, Victor. “Searching for Primitive Roots in Finite Fields.” Mathematics of Computation 58, no. 197 (January 1992).