IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Calum S. Skene, Peter J. Schmid
GTA email: css11@ic.ac.uk
Arjun Gill, CID 01053495

February 15, 2018

REED-SOLOMON DECODING

You have been given a new version of *ReedSolomon.py* for this coursework with more functions to complete and the solutions to the previous coursework included. You can substitute the solution code with the code you wrote in the previous coursework if you wish. You should submit

- 1 pdf report - using the LaTeX template provided on BB.
- 1 completed python script - *ReedSolomon.py*.
- The *.csv* files *recipe.csv*, *recipeOrder.csv*.
- The audio files *audioCorrupted.wav* and *audio.wav*.
- The python file *extraCode.py* containing any extra python code used.

Do not modify any of the functions that are already provided. Specifically, do not change the inputs and outputs specified. If we cannot run your codes from our main script you will lose marks.

Program in Python 3.

A python box in the question with 'Your code here' means that to score full marks for the question you must show the python code used. This code should also be in the file *extraCode.py*

1 Part A: Reed Solomon decoding (**8 pts**)

We assume there are at most t errors. This means the error polynomial must be of the form

$$e(X) = E_{j_1}X^{j_1} + E_{j_2}X^{j_2} + ...E_{j_T}X^{j_T}. \tag{1.1}$$

The equation that will let us obtain the error polynomial is called the key equation and is given as

$$\frac{\Omega(X)}{\Lambda(X)} = S(X) \bmod X^{2t}, \tag{1.2}$$

where

$$\Lambda(X) = \prod_{j=1}^{k}(1 - X_j X),$$
$$\Omega(X) = \sum_{j=1}^{k}\left[E_{j_i}X_j \prod_{p=1,p\neq j}^{k}(1 - X_p X)\right]$$

The roots of $\Lambda(X)$ are therefore inverse of the error locations $X_j = \alpha^j$. The provided function *xEuclid* takes the polynomials $S(X)$ and x^{2t} and returns the polynomials $\Omega(X)$ and $\Lambda(X)$, denoted by v and a respectively in the python code.

1. Complete the function *locateError* so that for a provided polynomial $\Lambda(X)$ it returns the error locations $\{j_1, j_2, ..j_t\}$.

```python
def locateError(Lam, GF, iGF):
    # error locator from Lambda (Chien)

    # we go through every power of alpha to determine the
      roots
    eval_vals = np.array([evalPoly(Lam, a, GF, iGF) for a
      in GF])
    error_locs = np.where(eval_vals == 0)[0]

    # now take inverses of these
    error_locs = -error_locs % len(GF)
    return error_locs
```

Once the error locations are found the error amplitudes E_{j_i} can be found using Forney's algorithm.

$$E_{j_i} = -\frac{\Omega(X_{j_i}^{-1})}{\Lambda'(X_{j_i}^{-1})}, \tag{1.3}$$

where the term

$$\Lambda'(X) = \sum_{i=1}^{v} i \cdot \lambda_i X^{i-1}, \tag{1.4}$$

is the formal derivative of the polynomial degree v polynomial $\Lambda(X) = \lambda_0 + \lambda_1 X + ...\lambda_v X^v$. Note that the operation $i \cdot \lambda_i$ is not multiplication as defined in the field GF(2^p), but is instead the usual addition i.e. λ_i added to itself $i-1$ times.

2. Complete the function *determineError* using Forney's algorithm so that for provided polynomials $\Lambda(X)$ and $\Omega(X)$ and error locations it returns the error amplitudes $\{E_{j_1}, E_{j_2}, ..E_{j_t}\}$.

```
def determineError(Lam, Om, loc, GF, iGF):
    # takes the polynomials Lam, Om, a list of the error
       locations loc
    # GF and iGF and returns a list of the corresponding
       error locations ev.
    ev = np.zeros(loc.shape).astype(int)
    p = len(GF)

    # compute derivative of Lambda
    dLam = Lam[:]
    for i in range(0, len(Lam), 2):
        dLam[len(Lam) - 1 - i] = 0
    dLam = trim(dLam[:len(dLam)-1])[0]

    for i, j in zip(range(len(loc)), loc):
        ev[i] = gf_div(evalPoly(Om, GF[-j % p], GF, iGF),
           evalPoly(dLam, GF[-j % p], GF, iGF), GF, iGF)
        ev[i] = gf_mult(ev[i], GF[j], GF, iGF)
    return ev
```

Now that we have a library of functions we are able to check a given message for errors, and find the locations and amplitudes of the errors. The errors can then easily be fixed by adding the determined error polynomial to the corrupted message.

3. Complete the function *RS_decode* so that for a given RS encoded message R containing *npar* parity bits the correct message containing zero errors is outputted. You may find it useful to test your function on the examples contained in *message1.csv, message2.csv* and *message3.csv* from Coursework 3.

```
def RS_decode(R, npar, GF, iGF):
    # Takes the encoded polynomial R, number of parity
       bits npar
    # GF and iGF and returns the decoded polynomial with
       no errors CC
    x2t = np.zeros(npar + 1).astype(int)
    x2t[0] = 1
    syndromePoly = compSyndromePoly(R, npar, GF, iGF)
    if syndromePoly.max() > 0:
        lam, om = xEuclid(x2t, syndromePoly, GF, iGF)
        lam = trim(lam)[0]
        om = trim(om)[0]
        error_locs = locateError(lam, GF, iGF)
        error = determineError(lam, om, error_locs, GF,
            iGF)
        CC = R.copy()
        CC[len(CC) - error_locs - 1] = CC[len(CC) -
            error_locs - 1] ^ error
    else:
        CC = R.copy()
    return CC
```

Testing this using the three examples from the previous coursework:

```
    p, t = 8, 5
    R = np.loadtxt("message1.csv", dtype='int', delimiter=
       ",")
    C = RS_decode(R, 2 * t, *table)
    print(C)
    np.savetxt("message1_decoded.csv", C.reshape(1, len(C)
       ), fmt="%d", delimiter=",")

>>> [ 73  32 119 114 111 116 101  32 116 104 105 115  32
   99 111 117 114 115 101 119 111 114 107  32 116 111  32
   116 104 101  32 109 117 115 105  99 32 111 102  32 109
   121  32 102  97 118 111 117 114 105 116 101  32  98 97
   110 100  44  32  82 117 115 104  46 169  33 254  17  21
    102  38  62 138 198]
```

```
    p, t = 8, 8
    R = np.loadtxt("message2.csv", dtype='int', delimiter=
       ",")
    C = RS_decode(R, 2 * t, *table)
    print(C)
    np.savetxt("message2_decoded.csv", C.reshape(1, len(C)
       ), fmt="%d", delimiter=",")

>>> [ 80 114 111 102 46  32  83  99 104 109 105 100 32 114
    101 97 108 108 121  32 101 110 106 111 121 115  32  98
    101 101 114  32 102 114 111 109 32  71 101 114 109  97
    110 121  46 229 246  99 250  98 226 103 1 37 110 157
   220 128  87 238  10]
```

```
    p, t = 8, 10
    R = np.loadtxt("message3.csv", dtype='int', delimiter=
       ",")
    C = RS_decode(R, 2 * t, *table)
    print(C)
    np.savetxt("message3_decoded.csv", C.reshape(1, len(C)
       ), fmt="%d", delimiter=",")

>>> [ 78 111 119  32 116 104  97 116  32 116 104 101  32
   99 111 117 114 115 101 119 111 114 107  32 105 115  32
   110 101  97 114 108 121  32 100 111 110 101  32 121 111
    117  32  99  97 110  32 114 101 108  97 120  46   0 42
    110  18 234 116 136 183  79  51 174  73  86  74 255
   40  60 141 159 199]
```

2 Part B: Recipe (**4 pts**)

Someone has sent you a recipe and has RS encoded it. Each line of the recipe is RS(255,223) encoded and the resulting polynomials are given as the lines of *recipeC.csv*. The integers in the message represent the corresponding ASCII character. Further to this the order that the lines come in has been RS(63,33) encrypted and this polynomial is given in *recipeOrderC.csv*.

1. Decode this message, correctly fixing the errors.Write the corrected polynomials to files named *recipe.csv* and *recipeOrder.csv*. Print the corrected recipe in the correct order.

```
    p, t = 8, 16
    table = gf_multTable(p)

    # recipe decoding
    recipe_length = len(np.genfromtxt("recipeC.csv", dtype
       ='int', delimiter=",", usecols=0))
    recipe_R = [0]*recipe_length
    recipe_CC = recipe_R.copy()
    recipe_M = recipe_R.copy()
    recipe_ordered = recipe_R.copy()
    for i in range(recipe_length):
        recipe_R[i] = np.genfromtxt("recipeC.csv", dtype='
           int', delimiter=",", skip_header=i, max_rows=1)
        recipe_CC[i] = RS_decode(recipe_R[i], 2*t, *table)
        recipe_M[i] = recipe_CC[i][:len(recipe_CC[i]) - 2
           * t]
```

```
p, t = 15, 15
table = gf_multTable(p)

# recipe order decoding
order_R = np.genfromtxt("recipeOrderC.csv", dtype='int
   ', delimiter=",", max_rows=1)
order_CC = RS_decode(order_R, 2*t, *table)
order_M = order_CC[:len(order_CC) - 2 * t]

# reorder recipe
for i, j in zip(range(recipe_length), order_M):
    recipe_ordered[i] = recipe_M[j]

with open('recipe.csv', 'w', newline='\n') as f:
    writer = csv.writer(f)
    writer.writerows(recipe_M)

with open('recipeOrder.csv', 'w', newline='\n') as f:
    writer = csv.writer(f)
    writer.writerows([order_M])

with open('recipe_ordered.csv', 'w', newline='\n') as
   f:
    writer = csv.writer(f)
    writer.writerows(recipe_ordered)

recipe_file = open('recipe_text.txt', 'w')
for i in range(0, recipe_length):
    recipe_file.write(''.join(chr(j) for j in
       recipe_ordered[i]) + '\n')
recipe_file.close()
```

The decoded recipe was saved to a .txt file, which reads:

Ingredients

125g butter, softened
100g light brown soft sugar
125g caster sugar
1 egg, lightly beaten
1 tsp vanilla extract
225g self-raising flour
0.5 tsp salt

200g chocolate chips

Method

Preheat the oven to 180°C, gas mark 4
Cream butter and sugars, once creamed, combine in the egg and vanilla.
Sift in the flour and salt, then the chocolate chips.
Roll into walnut size balls, for a more homemade look, or roll into a long, thick sausage shape and slice to make neater looking cookies.
Place on ungreased baking paper. Bake for just 7 minutes, till the cookies are just setting - the cookies will be really doughy and delicious.
Take out of the oven and leave to harden for a minute before transferring to a wire cooling rack. These are great warm, and they also store well, if they dont all get eaten straight away!

Source: https://www.bbcgoodfood.com/recipes/1580654/millies-cookies-recipe

3 Part C: Audio (**4 PTS**)

Some audio has been encoded using RS(32767,30000). The function *writeAudio* has been provided to you. It takes the audio in the form of a list *audio* and writes the audio to a *.wav* file with name *fileName*. The encoded polynomial is given in the file *audio.csv*.

1. Write the corrupted audio to a *.wav* file called *audioCorrupted.wav*.

```
p = 15
table = gf_multTable(p)

audio_R = np.genfromtxt("audio.csv", dtype='int',
   delimiter=",", max_rows=1)
writeAudio(audio_R[:len(audio_R) - 2767], '
   audioCorrupted.wav')
```

2. Decode the audio and write the correct audio to a file named *audio.wav*.

```
audio_CC = RS_decode(audio_R, 2767, *table)
audio_M = audio_CC[:len(audio_CC) - 2767]

writeAudio(audio_M, 'audio.wav')
```

The .wav file is a quote from Stanley Kubrick's 2001: A Space Odyssey.

4 Part D: Secret Question (**4 pts**)

The question for part D has been encoded using RS(511,411). The integers in the message represent the corresponding ASCII character. The encoded polynomial is contained in *partD.csv*.

1. Decode the question and answer it.

```
# secret question decoding
p = 9
table = gf_multTable(p)

message_R = np.genfromtxt("partD.csv", dtype='int',
   delimiter=",", max_rows=1)
message_CC = RS_decode(message_R, 100, *table)
message_M = message_CC[:len(message_CC) - 100]

file = open('partD.txt', 'w')
file.write(''.join(chr(j) for j in message_M) + '\n')
file.close()
```

The text for part D is saved under partD.txt. It reads:

Starting with RS(255,223), the message whose encoded polynomial is contained in date.csv, has been recursively encoded 10 times. Each time the Galois field used is increased to the next power of 2. The number of parity bits is always kept the same. Decode the message and print the date.

```
# secret question
npar = 32
date_R = np.genfromtxt("date.csv", dtype='int',
   delimiter=",", max_rows=1)
# iterate from 17 down to 8
for p in range(17, 7, -1):
    print(p)
    table = gf_multTable(p)
    date_CC = RS_decode(date_R, 32, *table)
    date_R = date_CC[:len(date_CC) - npar]

file = open('date.txt', 'w')
file.write(''.join(chr(j) for j in date_R) + '\n')
file.close()
```

The date is 28:06:42:12, which is from the film *Donnie Darko*.