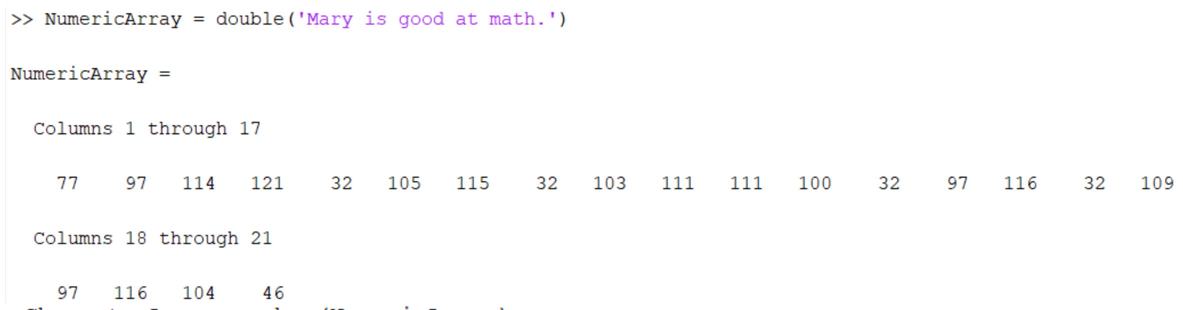
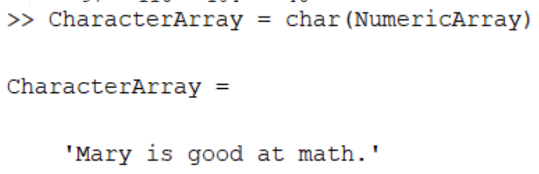
EE3731C Project Report　　　Ricky Sanjaya　　A0161326M

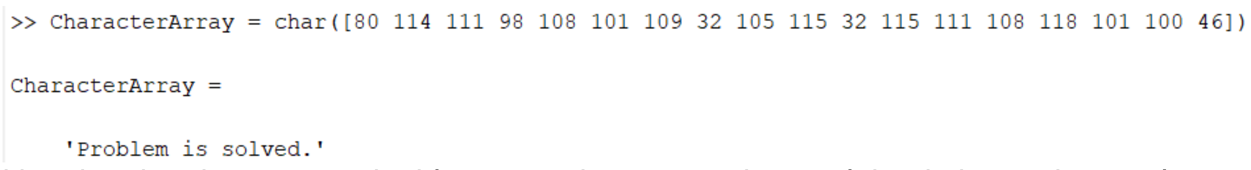
Q1) Mapping between Character and Double Arrays
In this section, we will create functions that would convert characters into numbers. The conversion is A->1, B->2, C->3, …. Z->26. Any other characters are mapped as 27. Furthermore, both uppercase and lowercase letters are mapped to the same number (e.g. A and a are both mapped to 1).
Results
a) double(input) converts the input to double-precision real numbers. char(input) converts integers to characters following the ASCII table (e.g. A is 65, B is 66, etc).



```
>> NumericArray = double('Mary is good at math.')

NumericArray =

  Columns 1 through 17

    77    97   114   121    32   105   115    32   103   111   111   100    32    97   116    32   109

  Columns 18 through 21

    97   116   104    46
```

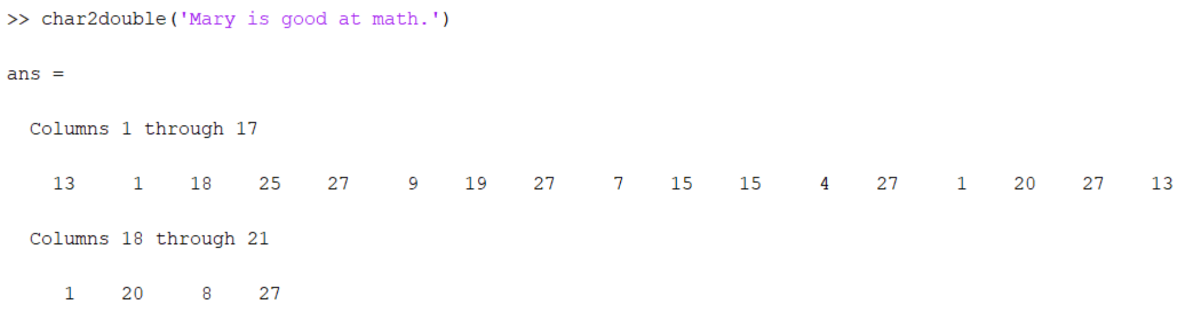


```
>> CharacterArray = char(NumericArray)

CharacterArray =

    'Mary is good at math.'
```



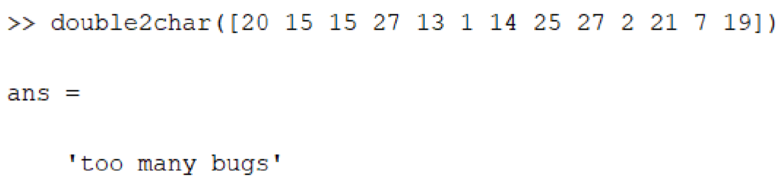
```
>> CharacterArray = char([80 114 111 98 108 101 109 32 105 115 32 115 111 108 118 101 100 46])

CharacterArray =

    'Problem is solved.'
```

b) Using the character to double conversion convention explained above, the results are



```
>> char2double('Mary is good at math.')

ans =

  Columns 1 through 17

    13     1    18    25    27     9    19    27     7    15    15     4    27     1    20    27    13

  Columns 18 through 21

     1    20     8    27
```

c) Using the double to character conversion explained above, the results are



```
>> double2char([20 15 15 27 13 1 14 25 27 2 21 7 19])

ans =

    'too many bugs'
```

Q2) In this section, we will perform encryption and decryption by using the concept of indexing an array by another array. An example of this is: Suppose we have an array A = [2 8 8 10]. Indexing A with another array, e.g. A([1 3]) will return us elements at the position specified by the elements in the indexing array (in our example, A([1 3]) = [2 8]). Inspecting the decryption and encryption keys, we can see that the keys (which correspond to the substitution cipher) are in the form of a row vector of numbers. The number at the ith position refers to the ‘value’ of the mapping of the ith alphabet, and the value refers to the integer representation of the corresponding alphabet following the convention in part 1 (i.e. 1 refers to A, 2 refers to B, etc). E.g. the frank_decrypt_key is as follows:

```
frank_decrypt_key =

  Columns 1 through 17

     5    13    26    27    22    25    23    12     2    21    19    11     6    14    17     7    10

  Columns 18 through 27

    16    24    20     9     8    15     4     3    18     1
```

This means that the first letter, i.e. letter a/A, is mapped to the fifth letter which is E, the letter B is mapped to the letter M, etc. This means that the substitute (decryption/encryption) for alphabet A lies in the first element of the array, for B it lies in the second element, and so on. Thus, for a letter (e.g. X), to grab the integer value corresponding to the substitute letter, we can simply index the array with the integer value of this letter following the convention in part 1 (e.g. to obtain the integer value of the substitute letter for A, we index the decrypt/encrypt key array with integer value of A which is 1, i.e. frank_decrypt_key(1)).

With this logic, decryption and encryption can thus be performed as follows:

- First convert the string to be decrypted/encrypted into numeric array using the function char2double from part 1.
- Then, index the decrypt/encrypt key array with the resulting numeric array. The result would be a numeric array corresponding to the substituted letter.
- Finally, convert this numeric array back to character, with the function double2char from part 1. The resulting array would be the encrypted/decrypted string.

With this workflow, several encryption and decryptions are done, and the results are reported below:

a) Encrypting ‘Mary is good at math.’ with frank_encrypt_key results in

```
>> test_double = char2double('Mary is good at math.');
test_encrypted_double = frank_encrypt_key(test_double);
test_encrypted = double2char(test_encrypted_double)

test_encrypted =

    'b zfdukdpwwxd tdb tvd'
```

b) Decrypting ‘gvwdukdnagdvaza’ with frank_decrypt_key results in

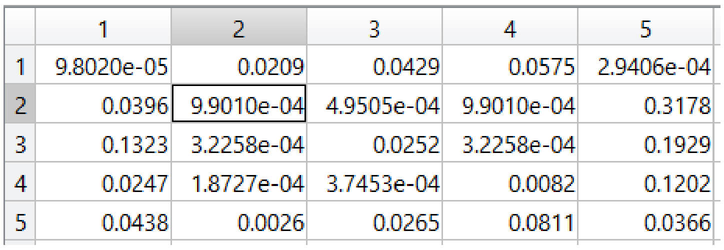
```
>> test_double = char2double('gvwdukdnagdvaza');
test_decrypted_double = frank_decrypt_key(test_double);
test_decrypted = double2char(test_decrypted_double)

test_decrypted =

    'who is new here'
```

Q3a) In this section, we examine the probability of letters occurring after some letters. This allows us the compute the probability of occurrence of a certain text, which will be used as posterior probabilities in the subsequent sections. To store the estimation of probability of occurrences, we create a 27 by 27 matrix pr_trans, where the rows represent the preceding letter, and the column represents the next letter.

The entry at any row (e.g. row i column j) represents the probability of the jth alphabet occurring after the ith alphabet. To compute these probabilities, we are given a training text of 174963 characters. Then for any row and column, e.g. row i column j, the probability is estimated as

$$\text{pr\_trans}(i,j) = \frac{1 + \#\text{times } j\text{-th alphabet appears after } i\text{-th alphabet in input text}}{27 + \#\text{times } i\text{-th alphabet appears in input text except at the last position}}$$

From compute_transition_probability.m, a snippet of the computed transition matrix is as follows:



	1	2	3	4	5
1	9.8020e-05	0.0209	0.0429	0.0575	2.9406e-04
2	0.0396	9.9010e-04	4.9505e-04	9.9010e-04	0.3178
3	0.1323	3.2258e-04	0.0252	3.2258e-04	0.1929
4	0.0247	1.8727e-04	3.7453e-04	0.0082	0.1202
5	0.0438	0.0026	0.0265	0.0811	0.0366

A simple sanity check on the computed transition matrix is applied, by inspecting the sum over each row (by definition, it must sum to 1). And as seen below, this is indeed true (result is technically a column vector but is reshaped to a row vector for ease of display).

```
Columns 1 through 10

  1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000

Columns 11 through 20

  1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000

Columns 21 through 27

  1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
```

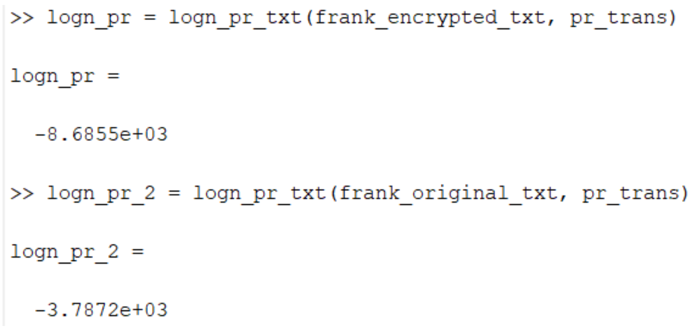
Furthermore, we can inspect some of the transition probabilities, such as
>> res1 = pr_trans(1,1)

res2 = pr_trans(2,3)
res1 = 9.8020e-05
res2 = 4.9505e-04
And maximum of the matrix as well as the corresponding alphabet transition as follows

```
M = max(max(pr_trans));
ind = find(pr_trans==M);
[rm,cm] = ind2sub(size(pr_trans),ind);

fprintf('Highest Probability is from letter %s to %s with transition probability of %f',double2char(rm),double2char(cm),M);
```

Highest Probability is from letter q to u with transition probability of 0.792000

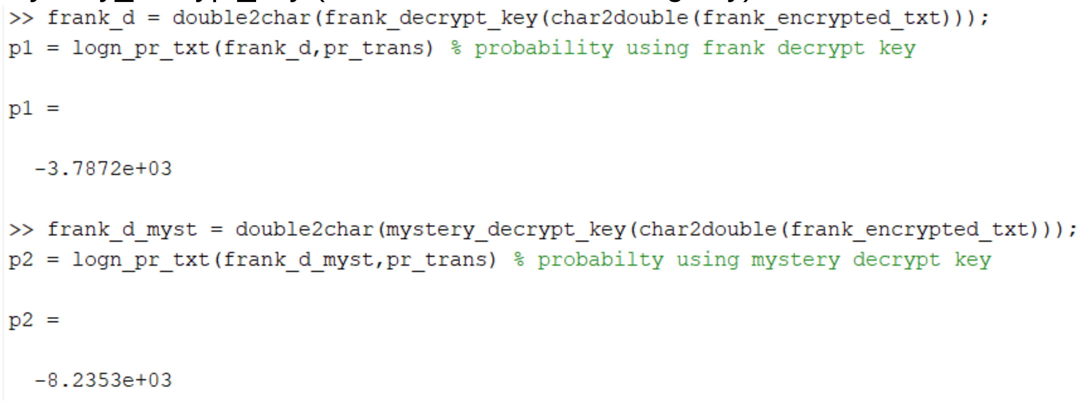
3b) As mentioned previously, we can then use this transition matrix to compute the probability of occurrence for a text (a string of characters). This is done by going through the string from start to 1 minus the end, inspecting a pair of letters at a time, and computing the probability of transition between these 2 letters (e.g. if text is 'Signal', then compute probability of transitions of s->i, i->g, g->n, n->a, a->l). Then the probability of the text can be computed by treating the text as a Markov Chain, where the current alphabet only depends on the first preceding alphabet. In this way, the probability of a text is simply the multiplication of each probability of transition contained in the text (in the example above, it would be pr_trans(19,9) x pr_trans(9,7) x …. x pr_trans(1,12)). Since such multiplication will lead to a very small value, we will use log of base e (natural logarithm) instead to represent the probabilities. This is done in the logn_pr_txt.m function. Some computations to check the correctness of the function are done below as per the project document's instructions.



```
>> logn_pr = logn_pr_txt(frank_encrypted_txt, pr_trans)

logn_pr =

  -8.6855e+03

>> logn_pr_2 = logn_pr_txt(frank_original_txt, pr_trans)

logn_pr_2 =

  -3.7872e+03
```

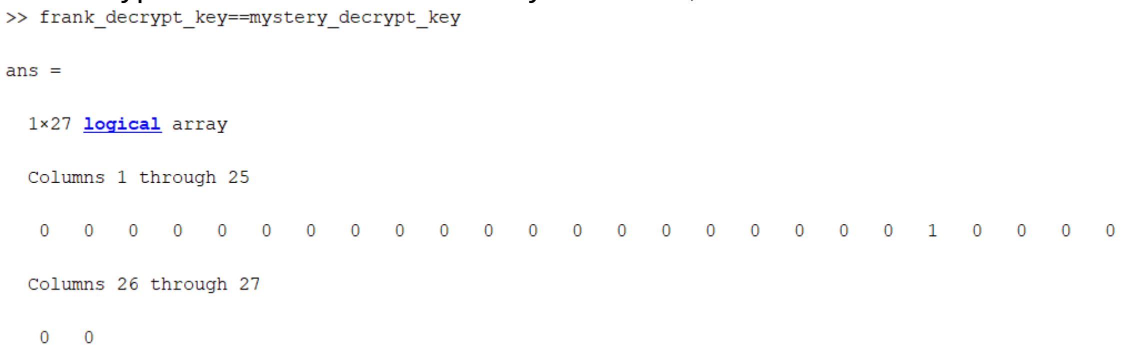
Here, we can observe that the decrypted text has a higher log probability than the encrypted one, and that the difference is 4.8985e+03. This means that the probability of occurrence of the decrypted text is e^(4.8985e+03) = 2.457e+2127 times higher than the probability of occurrence of the encrypted text. This suggests that our pr_trans matrix correctly reflects the desired phenomenon that correct/actual English text would have a significantly higher probability of occurring than scrambled text like an encrypted one.

Q3c) Here we will explore the idea of decryption using 2 different keys. By our modelling assumption, the probability p(encrypted_message|decrypt_key) is treated as the probability of the decrypted message, decrypted using decrypt_key, i.e. p(decrypted_message with decrypt key). This is found by first decrypting the message as per part 2, and then computing the probability of occurrence of the decrypted text using pr_trans. With this modelling assumption, we can thus expect that if we decrypted a message using the wrong key, then the probability of occurrence of the decrypted message would be significantly higher with the correct key vs that with the wrong key. To confirm this hypothesis, we decrypted the frank_encrypted_txt using the actual frank_decrypt_key (the correct key) and mystery_decrypt_key (assumed to be the wrong key). The results are as follows



```
>> frank_d = double2char(frank_decrypt_key(char2double(frank_encrypted_txt)));
p1 = logn_pr_txt(frank_d,pr_trans) % probability using frank decrypt key

p1 =

  -3.7872e+03

>> frank_d_myst = double2char(mystery_decrypt_key(char2double(frank_encrypted_txt)));
p2 = logn_pr_txt(frank_d_myst,pr_trans) % probabilty using mystery decrypt key

p2 =

  -8.2353e+03
```

Here, p1= logn(p(**frank_encrypted_txt | frank_decrypt_key**)) and p2 = logn(p(**frank_encrypted_txt | mystery_decrypt_key**)). Observe that p1 is larger than p2, and their difference is 4.4483e+03. This means that the decrypted text with frank decrypt key is e^(4.4483e+03) = 7.46615e+1931 more likely to occur than the decrypted text with the mystery decrypt key. This indicates that using a wrong key to decrypt a text will result in a text that has a much lower probability of occurring than the decrypted text with the correct key. However, it should be noted that

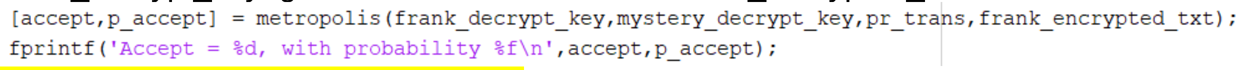


```
>> frank_decrypt_key==mystery_decrypt_key

ans =

  1×27 logical array

  Columns 1 through 25

   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0

  Columns 26 through 27

   0   0
```

The mystery decrypt key's mapping differs with the frank decrypt key for all alphabets except for letter u. As we will see in part 4, if the 2 keys do not differ much, then the probability of the resulting decrypted text might not differ too much.

Q4a) In this section, we will apply all the functions and ideas developed before into the Metropolis Algorithm. In applying Metropolis Algorithm for our case, the goal is to sample decryption keys from its posterior distribution (given the observation which is the encrypted message), i.e. sampling from p(decrypt_key | encrypted_message). Hence the switching rules in Metropolis algorithm will be based on this probability (more specifically, the ratio of this probability for 2 decryption keys). By Bayes rule, this probability is proportional to p(encrypted_message | decrypt_key) under the assumption that the prior probability of decryption keys are uniform (i.e. p(decrypt_key) = constant). Since the switching depends only on ratios, the constant terms will cancel, hence the switching can be equivalently governed by p(encrypted_message | decrypt_key). By the modelling assumption in part 3, this is equal to p(decrypted message using the given key). This idea is then used to implement the metropolis.m function, which takes in a current key and a new key as well as the transition probabilities from part 3, and returns the probability of accepting the new key, as well as using a random number generator to decide whether to accept or not. To ensure correct functionality of this switching, some test runs are done as per the project instructions

i) Probability of accepting mystery_decrypt_key with current key as frank_decrypt_key, given the observation is the frank_encrypted_txt.

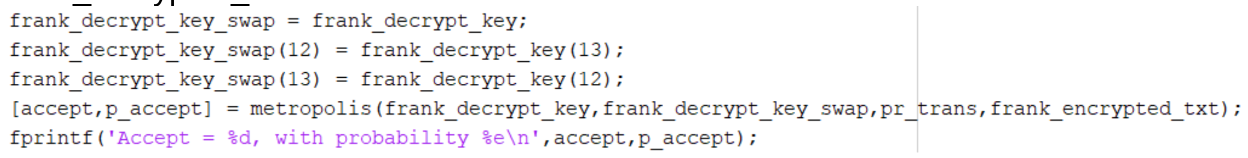


```
[accept,p_accept] = metropolis(frank_decrypt_key,mystery_decrypt_key,pr_trans,frank_encrypted_txt);
fprintf('Accept = %d, with probability %f\n',accept,p_accept);
```

Accept = 0, with probability 0.000000

This is because, the probability of accepting the new key is the ratio between the p(decrypted text) using the new key, over that using the old key. As seen in part 3c), the latter probability is 7.46615e+1931 times of the former probability. Hence the probability of accepting, which is the reciprocal of this value, is near 0, i.e.1.34e-1932.

ii) Probability of accepting a new key by swapping 12th and 13th element of frank_decrypt_key with current key as frank_decrypt_key, given the observation is frank_encrypted_txt.



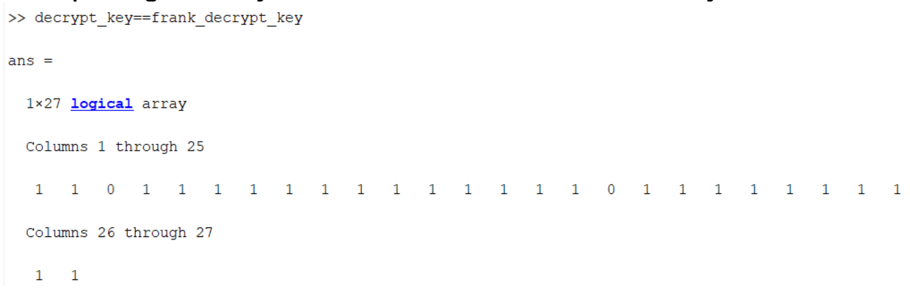
```
frank_decrypt_key_swap = frank_decrypt_key;
frank_decrypt_key_swap(12) = frank_decrypt_key(13);
frank_decrypt_key_swap(13) = frank_decrypt_key(12);
[accept,p_accept] = metropolis(frank_decrypt_key,frank_decrypt_key_swap,pr_trans,frank_encrypted_txt);
fprintf('Accept = %d, with probability %e\n',accept,p_accept);
```

Accept = 0, with probability 1.524835e-36

As seen here, when the new key is not so significantly different from the old key, the probability of accepting increases drastically (as opposed to case i)). However, the probability of accepting this new key is still very small, because the swapped mapping has a huge impact on the decrypted text (i.e. the alphabets whose mappings are swapped occur a lot of times in the encrypted text, which thus affects the probability greatly).

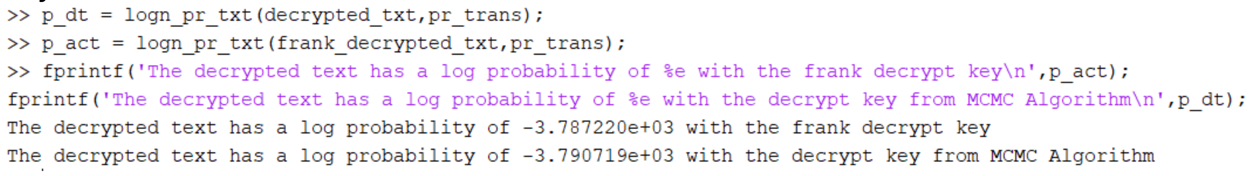
Q4b) In this section, we will apply the Metropolis Algorithm, extending the switching idea of part 4a) into 15000 iterations. This would thus result in a single sample drawn from the decrypt_key posterior distribution, which we would then treat as the decryption key. This is implemented in the mcmc.m file, and the result of applying this to frank_encrypt_text is as follows:
decrypted text:
'i cannot describe to you my sensations on the near prospect of my undertaking it is impossible to communicate to you a conception of the trembling sensation half pleasurable and half fearful with which i am preparing to depart i am going to unexplored regions to the land of mist and snow but i shall kill no albatross therefore do not be alarmed for my safety or if i should come back to you as worn and woeful as the   ancient mariner you will smile at my allusion   but i will disclose a secret i have of ten attributed my attachment to my passionate enthusiasm for the dangerous mysteries of ocean to that production of the most imaginative of modern poets there is something at work in my soul which i do not understand i am practically industrious painstaking a workman to execute with perseverance and labour but besides this there is a love for the marvellous a belief in the marvellous intertwined in all my prozects which hurries me out of the common pathways of men even to the wild sea and unvisited regions i am about to explore   but to return to dearer considerations shall i meet you again after having traversed immense seas and returned by the most southern cape of africa or america i dare not expect such success yet i cannot bear to look on the reverse of the picture continue for the present to write to me by every opportunity i may receive your letters on some occasions when i need them most to support my spirits i love you very tenderly remember me with affection should you never hear from me again   '
log probability = -3.790719e+03.

Comparing the key from mcmc.m vs the actual key,



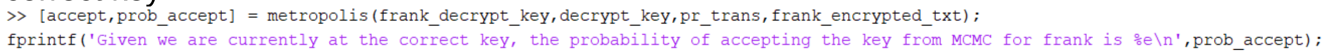
```
>> decrypt_key==frank_decrypt_key

ans =

  1×27 logical array

  Columns 1 through 25

   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1

  Columns 26 through 27

   1   1
```

As seen above, the decrypted key differs from the actual key by 2 alphabet substitutes; the substitute for the alphabet c and q. The actual key maps c->z and q->j, whereas the decrypted key maps c->j and q->z. These differences arise in the word ‘prozects’ highlighted in the decrypted text above. The actual word should have been ‘projects’, but the letter that was supposed to be decrypted to j (which is q), is decrypted to z instead, hence the error. On further inspection, there is only one occurrence of the letter q in the encrypted text and no occurrence of the letter c, hence there is only 1 error.
Inspecting the log probability of the decrypted text using the actual key vs the wrong key obtained from mcmc.m



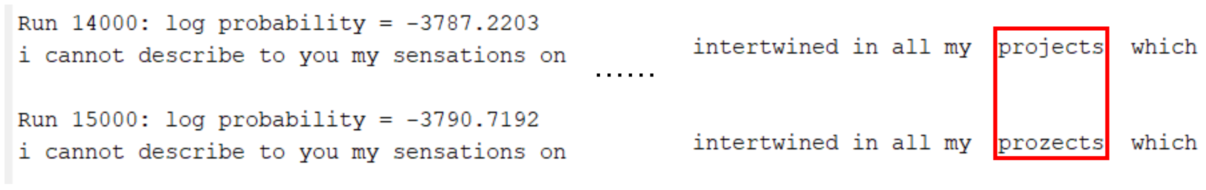
```
>> p_dt = logn_pr_txt(decrypted_txt,pr_trans);
>> p_act = logn_pr_txt(frank_decrypted_txt,pr_trans);
>> fprintf('The decrypted text has a log probability of %e with the frank decrypt key\n',p_act);
fprintf('The decrypted text has a log probability of %e with the decrypt key from MCMC Algorithm\n',p_dt);
The decrypted text has a log probability of -3.787220e+03 with the frank decrypt key
The decrypted text has a log probability of -3.790719e+03 with the decrypt key from MCMC Algorithm
```

and also inspecting the probability of switching to the wrong key given we are at the correct key



```
>> [accept,prob_accept] = metropolis(frank_decrypt_key,decrypt_key,pr_trans,frank_encrypted_txt);
fprintf('Given we are currently at the correct key, the probability of accepting the key from MCMC for frank is %e\n',prob_accept);
```

Given we are currently at the correct key, the probability of accepting the key from MCMC for frank is 3.022919e-02
As seen above, the log probability of the decrypted text using the correct key vs using the wrong key differs very slightly. This thus results in the probability of switching from the correct key to the wrong key to be significantly larger than the cases in part 4a i) and ii). This means that when the algorithm is running, it is very probable that we arrived at the correct key but switched to the wrong one because the random number generated is less than 3.022919e-02. This is shown below



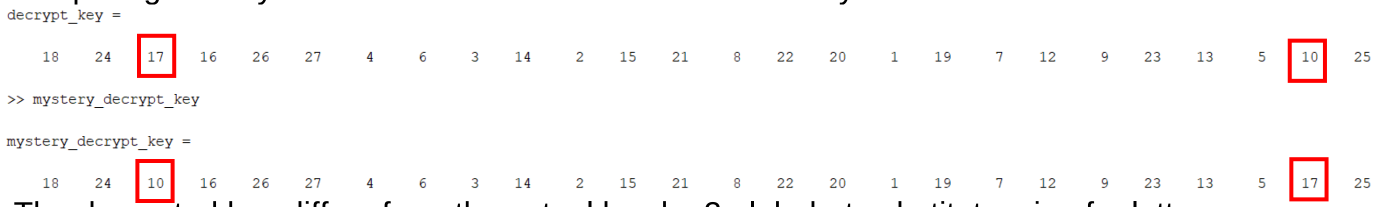
```
Run 14000: log probability = -3787.2203
i cannot describe to you my sensations on   ......   intertwined in all my projects which

Run 15000: log probability = -3790.7192
i cannot describe to you my sensations on            intertwined in all my prozects which
```

As seen above, at run 14000, we already obtained the correctly decrypted text (the log probability is the same as log probability of decrypted text using frank_decrypt_key), but in run 15000, we obtained the wrong key (log probability is the same as the log probability of decrypting the text with the wrong key). Therefore, the algorithm may not give the correct answer (in this case, it will definitely not give the correct answer due to the seed in the code).

Q4c) Here, we will repeat the decryption process in 4b), but on the mystery text. The resulting decrypted text is
well three or four months run along and it was well into the winter now i had been to school most all the time and could spell and read and write qust a little and could say the multiplication table up to six times seven is thirty five and i don t reckon i could ever get any further than that if i was to live forever i don t take no stock in mathematics anyway at first i hated the school but by and by i got so i could stand it whenever i got uncommon tired i played hookey and the hiding i got next day done me good and cheered me up so the longer i went to school the easier it got to be i was getting sort of used to the widow s ways too and they warn t so raspy on me living in a house and sleeping in a bed pulled on me pretty tight mostly but before the cold weather i used to slide out and sleep in the woods

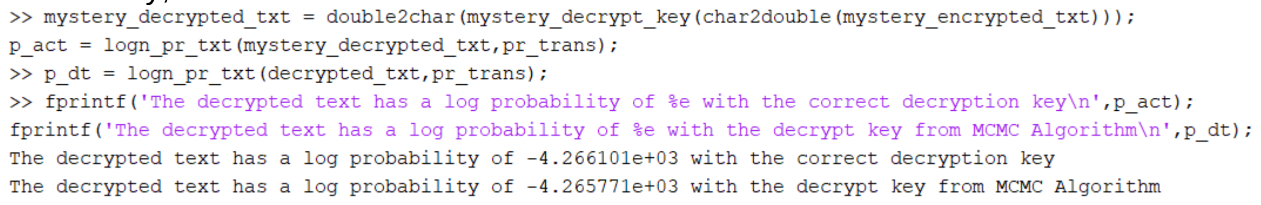
sometimes and so that was a rest to me   i liked the old ways best but i was getting so i liked the new ones too a little bit the widow said i was coming along slow but sure and doing very satisfactory   she said she warn t ashamed of me   one morning i happened to turn over the salt cellar at breakfast   i reached for some of it as juick as i could to throw over my left shoulder and keep off the bad luck but miss watson was in ahead of me   and crossed me off she says   take your hands away huckleberry what a mess you are always making   the widow put in a good word for me but that warn t going to keep off the bad luck i knowed that well enough   i started out after breakfast feeling worried and shaky and wondering where it was going to fall on me and what it was going to be   there is ways to keep off some kinds of bad luck but this wasn t one of them kind so i never tried to do anything but qust poked along low spirited and on the watch out
log probability = -4.265771e+03

Comparing the key obtained from mcmc.m vs the actual key

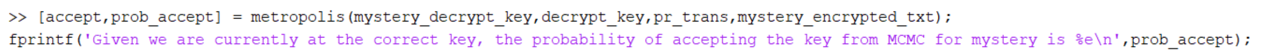


```
decrypt_key =

   18    24    17    16    26    27     4     6     3    14     2    15    21     8    22    20     1    19     7    12     9    23    13     5    10    25

>> mystery_decrypt_key

mystery_decrypt_key =

   18    24    10    16    26    27     4     6     3    14     2    15    21     8    22    20     1    19     7    12     9    23    13     5    17    25
```

The decrypted key differs from the actual key by 2 alphabet substitutes, i.e. for letter c and y. The MCMC key maps c->q and y->j, whereas the actual decryption key maps c->j and y->q. This gives rise to errors in words highlighted in yellow above, i.e. ‘qust’ and ‘juick’. This is because the letter that is supposed to be mapped to j, i.e. c, is erroneously mapped to q, and letter y which is supposed to be mapped to q, is mapped to j instead. Hence, the word just becomes qust, and quick becomes juick. Inspecting the log probability of the decrypted text using the wrong vs the correct key, as well as the probability of accepting the wrong key given, we are at the correct key,

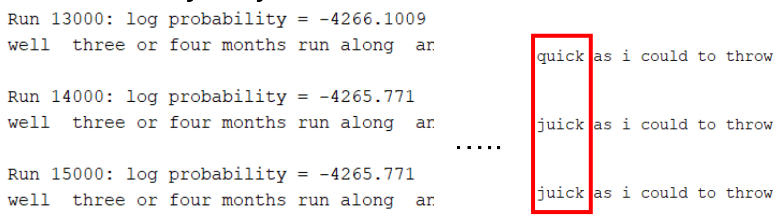


```
>> mystery_decrypted_txt = double2char(mystery_decrypt_key(char2double(mystery_encrypted_txt)));
p_act = logn_pr_txt(mystery_decrypted_txt,pr_trans);
>> p_dt = logn_pr_txt(decrypted_txt,pr_trans);
>> fprintf('The decrypted text has a log probability of %e with the correct decryption key\n',p_act);
fprintf('The decrypted text has a log probability of %e with the decrypt key from MCMC Algorithm\n',p_dt);
The decrypted text has a log probability of -4.266101e+03 with the correct decryption key
The decrypted text has a log probability of -4.265771e+03 with the decrypt key from MCMC Algorithm
```



```
>> [accept,prob_accept] = metropolis(mystery_decrypt_key,decrypt_key,pr_trans,mystery_encrypted_txt);
fprintf('Given we are currently at the correct key, the probability of accepting the key from MCMC for mystery is %e\n',prob_accept);
```

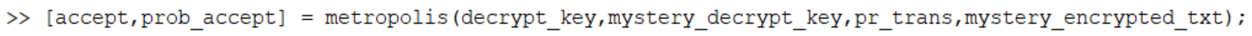
Given we are currently at the correct key, the probability of accepting the key from MCMC for mystery is 1



```
Run 13000: log probability = -4266.1009
well  three or four months run along  ar        quick as i could to throw

Run 14000: log probability = -4265.771
well  three or four months run along  ar  .....  juick as i could to throw

Run 15000: log probability = -4265.771
well  three or four months run along  ar        juick as i could to throw
```

As seen above, the log probability computed using pr_trans for the decrypted text using the wrong key is **higher** than the that using the correct key. Hence given we are at the correct key, the probability of accepting the new wrong key is 1. This is seen below

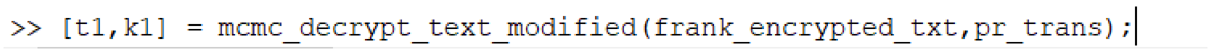
As seen above, at run 13000, we already obtained the correct key (observe the log probability of the decrypted text being equal to that of the text decrypted using the correct key), but in run 14000 (and all the way to the end), the algorithm switches to the wrong key and stayed there. We can also inspect that given we are at the wrong key, the probability of switching to the correct key is



```
>> [accept,prob_accept] = metropolis(decrypt_key,mystery_decrypt_key,pr_trans,mystery_encrypted_txt);
```

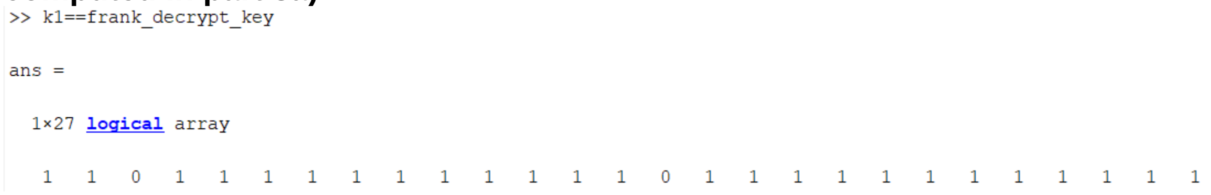
Given we are currently at the wrong key, the probability of accepting the correct key for mystery is 7.190071e-01.
This shows that the algorithm will more likely output the wrong key as opposed to the correct one, and therefore the algorithm did not give the correct answer
This finding also highlights an important problem; that the estimated pr_trans matrix could be erroneous due to lack of training data. We will attempt to fix this problem in part 5.

Q5) In this section, a method will be proposed to improve the Metropolis Algorithm, such that we get the correct decryption key. There will be 2 improvements proposed. The first one is based on the findings in section 4i). In this section, we found that given we are at the correct key, the probability of switching to the wrong key is non-zero (i.e. it is 3.022919e-02 which is quite significant compared to cases like part 3). Thus, there would be instances where after 15000 iterations, we would end up with the wrong key. Hence the proposed improvement is to do MAP estimation instead. We would reduce the number of iterations to 3000 for the metropolis algorithm, but we run these 3000 iterations 50 times to obtain 50 samples. Then, we count the frequency of each samples, and take the sample that has the highest frequency, effectively obtaining the MAP estimate of the posterior distribution p(decrypt_key | encrypted_message). Although technically the minimum number of iterations needed within each Metropolis Algorithm run is 27 (because by the mcmc.m method of swapping 2 positions, at most 27 iterations is needed to obtain any key from all key possibilities, i.e. by swapping each position from left to right to match the desired key), we need a much larger number of iteration to ensure that each Metropolis Algorithm run converges to the stationary distribution which corresponds to the posterior probability of the decryption keys. 3000 iterations are chosen so that the code’s running time is reasonable. Finally, the random number generator seed is disabled so that every run of the Metropolis Algorithm won’t result in the same sample and also to show that the code is robust enough, that is we can repeat the procedure multiple times and even with different frequencies and different samples, the MAP estimate must still be the key that gives the correct decryption. Running the procedure on frank case, the result is as follows:



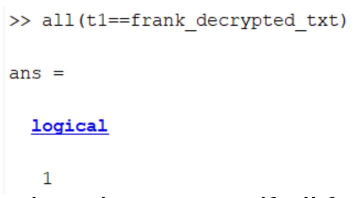
```
>> [t1,k1] = mcmc_decrypt_text_modified(frank_encrypted_txt,pr_trans);
```

***Note: To perform decryption on frank, run the above line with the pr_trans computed in part 3a)**



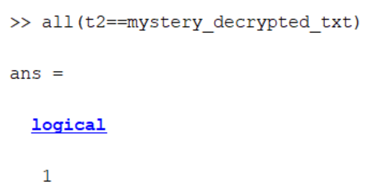
```
>> k1==frank_decrypt_key

ans =

  1×27 logical array

   1   1   0   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1
```

It seems that the key obtained still differs from the actual frank decrypt key. However, comparing the decrypted text t1 with the correct decrypted text, we see that the t1 is the same as the correct decrypted text (note that all function checks an



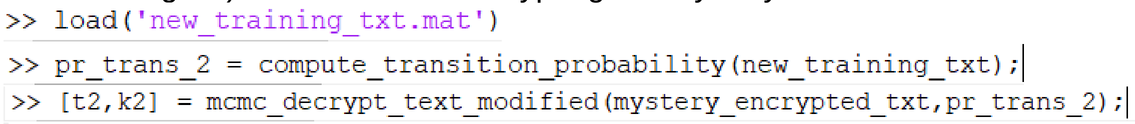
```
>> all(t1==frank_decrypted_txt)

ans =

  logical

   1
```

array and outputs 1 if **ALL** elements of the array is 1. Doing t1==frank_decrypted_txt will output a logical array whose length is equal to the text length, with 1's at the position where the letter in t1 at that position is same as letter in the frank_decrypted_txt at that position and 0 otherwise. Hence if all function returns a 1, the whole t1 is identical to frank_decrypted_txt). Hence the key from the MAP estimate still gives the correct decryption even when it is not the 'correct key'.
k1 = 5 13 17 27 22 25 23 12 2 21 19 11 6 14 26 7 10 16 24 20 9 8 15 4 3 18 1
This is because, on further inspection, notice that the wrong mappings are for letters c and o. However, if we inspect the encrypted text, there are no c's nor o's. Hence, even if the mapping for these 2 letters are wrong, the key will still correctly decrypt the text. Thus, MAP have successfully decrypted the frank case.
For the mystery text case, there is another improvement to be done. As explained in part 4c), the probability of accepting the wrong key given we are at the correct key is 1, which highlights that pr_trans might have been estimated wrongly. Inspecting the training text, we can see that it is written in modern English, as seen in this sentence from the text: 'he had many disappointments and hardships but through his kindness to an old man his own relatives and right name were revealed to him'. However, from the decrypted mystery text, notice that it is written in old English, as seen from the parts highlighted in green in part 4c). This difference in writing style could be a possible cause for erroneously estimating pr_trans. Hence, the second improvement proposed is to obtain another training text, which is written in old English, and use this new training text to compute pr_trans. Then using the newly computed pr_trans, we perform the above MAP estimate to get the decryption key for this mystery text. A

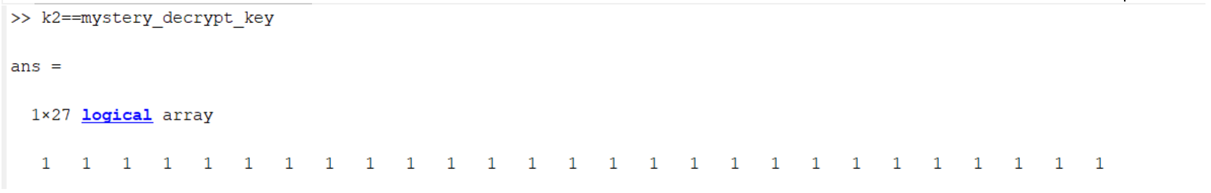


```
>> all(t2==mystery_decrypted_txt)

ans =

  logical

   1
```

sentence in the new training text is as follows: 'i made fast to a willow then i took a bite to eat and by and by laid down in the canoe to smoke a pipe and lay out a plan'. Note that this newly computed pr_trans will not work in decrypting frank, again because of the different English style (notice from part 4b that frank text is also written in modern English). The results of decrypting the mystery text is as follows:



```
>> load('new_training_txt.mat')
>> pr_trans_2 = compute_transition_probability(new_training_txt);
>> [t2,k2] = mcmc_decrypt_text_modified(mystery_encrypted_txt,pr_trans_2);
```



```
>> k2==mystery_decrypt_key

ans =

  1×27 logical array

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

In this case, the key obtained is identical to the actual decryption key for the mystery text. As a final check, the decrypted text is compared with the correct decrypted text and it is identical. Hence, we have successfully decrypted the mystery text. Note that these types of improvement can be applied to a wide variety of text, by first applying some training text to sufficiently decrypt the text. Then after getting a sense of the style of the text, find another training text with the same style (e.g. if text is a mail, might want to consider message texts) and reapply the decryption.
Note 2: For demonstration on how to use this decryption to decrypt the mystery text, run the 3 lines of code shown above.