Transmission de donnees corrige

March 24, 2021

1 Cryptographie

Un intérêt de la transmission de donnée et d'essayer d'assurer sa confidentialité. Aujourd'hui les méthodes de chiffrement RSA permettent d'assurer la fiabilité des transmissions même sur des réseaux publiques comme le réseaux internet.

Nous allons étudier des méthodes de chiffrement antérieure à cette technique.

1.1 Encodage d'un texte

Afin d'appliquer des méthodes de cryptographie à des textes, nous allons devoir effectuer des opérations sur les caractères (lettres, accents, ponctuation, ...) d'un texte. Afin de faciliter la manipulation de ces calculs numériques on met en place une correspondance entre les caractères et des nombres entiers. Pour cela on utilisera la norme Unicode qui est déjà en place à l'aide de deux fonctions ord() et chr()

[1]:
```
chr(65)
```

[1]:
```
'A'
```

[2]:
```
ord('z')
```

[2]:
```
122
```

Ecrire une fonction qui convertie un texte vers une liste de nombre entier représentant ses caractères dans la norme Unicode, ainsi que la fonction réciproque.

[3]:
```
def texte_vers_liste(texte):
    liste = []
    for lettre in texte :
        liste.append(ord(lettre))
    return liste
```

[4]:
```
def liste_vers_texte(liste):
    texte = ''
    for code in liste :
        texte+=chr(code)
    return texte
```

Tester vos fonctions pour le texte 'azerty'

[5]:
```
texte_vers_liste('azerty')
```

[5]:
```
[97, 122, 101, 114, 116, 121]
```

[6]:
```
liste_vers_texte([97, 122, 101, 114, 116, 121])
```

[6]:
```
'azerty'
```

On remarque que certains nombre entier seuleument sont utilisés pour les lettres de l'alphabet. Déterminer quels nombres sont utilisé pour les lettres minuscules, majuscules, et les accents.

[7]:
```
texte_vers_liste('AZ az')
```

[7]:
```
[65, 90, 32, 97, 122]
```

[8]:
```
accents =␣
 ↪[ord('à'),ord('â'),ord('é'),ord('è'),ord('ê'),ord('ë'),ord('î'),ord('ï'),ord('ô'),ord('ù'),
print(accents)
```

```
[224, 226, 233, 232, 234, 235, 238, 239, 244, 249, 251, 252, 231]
```

[9]:
```
a_accents = [ord('à'),ord('â')]
e_accents = [ord('é'),ord('è'),ord('ê'),ord('ë')]
i_accents = [ord('î'),ord('ï')]
o_accents = [ord('ô')]
u_accents = [ord('ù'),ord('û'),ord('ü')]
c_cedille = [ord('ç')]
```

Dans les algorithmes de chiffrement mis en place on se contentera de ne chiffrer que les lettres sans distinction entre majuscule, minuscule, accents et on gardera tels quels les autres caractères: ponctuation, apostrophes, ...

Il faudra alors faire une distinction de cas selon les valeurs des codes rencontrés.

2 Chiffrement de César

Une première stratégie de cryptographie consiste à décaler toutes les lettre d'un texte. Par exemple dans un texte remplacer tous les a par des c, les b par des d, les c par des e, ...

Lorsque le a est remplacé par un c, on dit que la clef du chiffrement est le 'c'.

Ecrire une fonction chiffre de César qui prend en argument une clef et un texte et retourne le texte chiffré.

[10]:
```
def chiffre_de_César(clef,texte):
    liste = texte_vers_liste(texte)
    decalage = ord(clef)-97
    liste_chiffre = []
    for code in liste :
        if code >= 65 and code <= 90:
            liste_chiffre.append((code+decalage-65)%26 + 65)
        elif code >= 97 and code <= 122:
            liste_chiffre.append((code+decalage-97)%26 + 97)
        elif code in a_accents:
            liste_chiffre.append((ord('a')+decalage-97)%26 + 97)
        elif code in e_accents:
            liste_chiffre.append((ord('e')+decalage-97)%26 + 97)
```

```
        elif code in i_accents:
            liste_chiffre.append((ord('i')+decalage-97)%26 + 97)
        elif code in o_accents:
            liste_chiffre.append((ord('o')+decalage-97)%26 + 97)
        elif code in u_accents:
            liste_chiffre.append((ord('u')+decalage-97)%26 + 97)
        else :
            liste_chiffre.append(code)
    return liste_vers_texte(liste_chiffre)
```

On pourra tester le chiffrement sur le texte “Toutes choses sont faites d’atomes, petites particules animées d’un mouvement incessant, qui s’attirent lorsqu’elles sont distantes les une des autres, mais se repoussent lorsqu’on les pousse à se serrer trop près.” avec la clef “c”.

[11]:
```
texte = "Toutes choses sont faites d'atomes, petites particules animées d'un␣
 ↪mouvement incessant, qui s'attirent lorsqu'elles sont distantes les une des␣
 ↪autres, mais se repoussent lorsqu'on les pousse à se serrer trop près."
chiffre_de_César('c',texte)
```

[11]:
```
"Vqwvgu ejqugu uqpv hckvgu f'cvqogu, rgvkvgu rctvkewngu cpkoggu f'wp oqwxgogpv
kpeguucpv, swk u'cvvktgpv nqtusw'gnngu uqpv fkuvcpvgu ngu wpg fgu cwvtgu, ocku
ug tgrqwuugpv nqtusw'qp ngu rqwuug c ug ugttgt vtqr rtgu."
```

Une fois le texte chiffré on remarque qu’il est illisible à moins d’avoir la clef. Ecrire une fonction decodage_de_Cesar() qui prend en argument la clef et le texte chiffré et retourne le texte en clair.

[12]:
```
def decodage_de_Cesar(clef,texte_chiffre):
    liste_chiffre = texte_vers_liste(texte_chiffre)
    decalage = ord(clef)-97
    liste = []
    for code in liste_chiffre :
        if code >= 65 and code <= 90:
            liste.append((code-decalage-65)%26 + 65)
        elif code >= 97 and code <= 122:
            liste.append((code-decalage-97)%26 + 97)
        else :
            liste.append(code)
    return liste_vers_texte(liste)
```

On pourra tester notre fonction sur le texte chiffré obtenu précédement avec la bonne clef et une mauvaise clef.

[13]:
```
texte_chiffre = chiffre_de_César('c',texte)
decodage_de_Cesar('c',texte_chiffre)
```

[13]:
```
"Toutes choses sont faites d'atomes, petites particules animees d'un mouvement
incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais
se repoussent lorsqu'on les pousse a se serrer trop pres."
```

[14]:
```
decodage_de_Cesar('v',texte_chiffre)
```

[14]:
```
"Avbalz jovzlz zvua mhpalz k'havtlz, wlapalz whyapjbslz huptllz k'bu tvbcltlua
pujlzzhua, xbp z'haapylua svyzxb'lsslz zvua kpzahualz slz bul klz hbaylz, thpz
```

```
zl ylwvbzzlua svyzxb'vu slz wvbzzl h zl zlyyly ayvw wylz."
```

La confidentialité du message est donc garantie par la confidentialité de la clef. Un moyen de décoder le message est de trouver une stratégie pour deviner la clef utilisée.

Une stratégie efficace pour retrouver la clef et d'effectuer une analyse en fréquence. L'idée est de remarquer qu'en langue française certaines lettres apparaissent plus souvent que d'autres. Par exemple en moyenne la lettre "e" est la lettre la plus utilisée en français. Bien sur ceci peut varier d'un texte à l'autre selon l'auteur on peut penser notamment au roman de Georges Perec: La Disparition.

Ecrire une fonction qui compte le nombre d'occurence de chaque lettre de l'alphabet (sans distinction de majuscule ou d'accent) dans un texte.

[15]:
```python
def analyse_en_frequence(texte):
    liste = texte_vers_liste(texte)
    occurences = 26*[0]
    for code in liste:
        if code >= 65 and code <= 90:
            occurences[code-65] += 1
        elif code >= 97 and code <= 122:
            occurences[code-97] += 1
        elif code in a_accents:
            occurences[ord('a')-97] += 1
        elif code in e_accents:
            occurences[ord('e')-97] += 1
        elif code in i_accents:
            occurences[ord('i')-97] += 1
        elif code in o_accents:
            occurences[ord('o')-97] += 1
        elif code in u_accents:
            occurences[ord('u')-97] += 1
    return occurences
```

Tester votre fonction sur le texte "Toutes choses sont faites d'atomes, petites particules animées d'un mouvement incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais se repoussent lorsqu'on les pousse à se serrer trop près." . Que remarquez-vous, commenter ?

[16]:
```python
analyse_en_frequence(texte)
```

[16]:
```
[10,
 0,
 3,
 4,
 29,
 1,
 0,
 1,
 9,
 0,
 0,
 7,
```

```
 5,
 12,
 12,
 6,
 3,
 11,
 31,
 19,
 11,
 1,
 0,
 0,
 0,
 0]
```

On remarque que c'est la lettre "s" ici qui a le plus d'occurence, en effet la phrase est au pluriel. Viens ensuite la lettre "e" comme on s'y attends.

Si l'algorithme de chiffrement utilisé est un algorithme de César à décalage, on observera alors simplement un décalage de la lettre la plus utilisée.

Ecrire un programme qui montre cet effet et mettre en oeuvre un protocole pour deviner la clef lors d'un chiffrement de César.

[17]:
```
analyse_en_frequence(texte_chiffre)
```

[17]:
```
[0,
 0,
 10,
 0,
 3,
 4,
 29,
 1,
 0,
 1,
 9,
 0,
 0,
 7,
 5,
 12,
 12,
 6,
 3,
 11,
 31,
 19,
 11,
 1,
 0,
```

```
0]
```

On remarque que les nombres d'occurences maximale ne sont pas "s" et "e" mais "u" et "g", on a donc bien un décalage de "u" - "s" = "g" - "e" = "c"

2.1 Chiffrement de Vigenère

Une stratégie pour renforcer la sécurité du chiffrement de César est d'utiliser un mot de plusieurs lettre comme clef, par exemple la clef n'est plus "c" mais "feynman".

L'utilisation d'une clef à plusieurs caractère consiste à effectuer un chiffrement de César avec la clef "f" sur la première lettre, puis la clef "e" sur la seconde, puis la clef "y" sur la troisième, et ainsi de suite jusqu'à la fin du mot "feynman" où on remcommence à la clef "f".

Ecrire une fonction chiffre de Vigenère qui prend en argument une clef et un texte et retourne le texte chiffré.

[18]:
```python
import numpy as np

def chiffre_de_Vigenère(clef,texte):
    liste = texte_vers_liste(texte)
    longueur_clef = len(clef)
    decalage = np.zeros(longueur_clef,dtype = int)
    decalage[:] = texte_vers_liste(clef)
    decalage -= 97
    liste_chiffre = []
    position = 0
    for code in liste :
        if code >= 65 and code <= 90:
            liste_chiffre.append((code+decalage[position]-65)%26 + 65)
            position += 1
            position %= longueur_clef
        elif code >= 97 and code <= 122:
            liste_chiffre.append((code+decalage[position]-97)%26 + 97)
            position += 1
            position %= longueur_clef
        elif code in a_accents:
            liste_chiffre.append((ord('a')+decalage[position]-97)%26 + 97)
            position += 1
            position %= longueur_clef
        elif code in e_accents:
            liste_chiffre.append((ord('e')+decalage[position]-97)%26 + 97)
            position += 1
            position %= longueur_clef
        elif code in i_accents:
            liste_chiffre.append((ord('i')+decalage[position]-97)%26 + 97)
            position += 1
            position %= longueur_clef
        elif code in o_accents:
            liste_chiffre.append((ord('o')+decalage[position]-97)%26 + 97)
            position += 1
```

```
            position %= longueur_clef
        elif code in u_accents:
            liste_chiffre.append((ord('u')+decalage[position]-97)%26 + 97)
            position += 1
            position %= longueur_clef
        else :
            liste_chiffre.append(code)
    return liste_vers_texte(liste_chiffre)
```

Puis tester votre fonction toujours sur le même texte avec la clef “feynman”.

[19]:
```
clef = 'feynman'
chiffre_de_Vigenère(clef,texte)
```

[19]:
```
"Yssgqs pmsqre sbsx dnutrx h'ygamrx, tcgutrx tyefipzpcf mnvricf p'ua rssiqmrsx
gaoefxelg, cuv x'ergurrsx jbdsdz'ijyqs ftrr qusgfrrre lrx ylr pef fyreqs, zfmq
fq rrussfeeay pmeeqh'tr jre pbzwqr m sr xipeqr gwsn cdef."
```

Ecrire ensuite une fonction de décodage qui à partir de la clef et du texte chiffré donne le texte initial, puis testez là.

[20]:
```
def decodage_de_Vigenère(clef,texte_chiffre):
    liste_chiffre = texte_vers_liste(texte_chiffre)
    longueur_clef = len(clef)
    decalage = np.zeros(longueur_clef,dtype = int)
    decalage[:] = texte_vers_liste(clef)
    decalage -= 97
    liste = []
    position = 0
    for code in liste_chiffre :
        if code >= 65 and code <= 90:
            liste.append((code-decalage[position]-65)%26 + 65)
            position += 1
            position %= longueur_clef
        elif code >= 97 and code <= 122:
            liste.append((code-decalage[position]-97)%26 + 97)
            position += 1
            position %= longueur_clef
        else :
            liste.append(code)
    return liste_vers_texte(liste)
```

[21]:
```
texte_chiffre = chiffre_de_Vigenère(clef,texte)
decodage_de_Vigenère(clef,texte_chiffre)
```

[21]:
```
"Toutes choses sont faites d'atomes, petites particules animees d'un mouvement
incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais
se repoussent lorsqu'on les pousse a se serrer trop pres."
```

L’enjeu est à nouveau de trouver une technique pour deviner la clef à partir du texte chiffré seuleument. On peut commencer par essayer d’utiliser une analyse en fréquence.

Suivez la même procédure que précédement que remarquez vous ?

Si on suit la procédure précédente on commence par compter les occurences

[22]:
```
analyse_en_frequence(texte_chiffre)
```

[22]:
```
[4,
 3,
 5,
 4,
 15,
 12,
 8,
 2,
 5,
 3,
 0,
 3,
 7,
 3,
 1,
 8,
 11,
 25,
 18,
 6,
 7,
 2,
 2,
 10,
 7,
 4]
```

On remarque que le “r” est la lettre qui apparaît le plus de fois, on peut donc supposer que la clef est “r” - “s” = -1 = “z”.

On essaye de décoder avec la clef “z”

[23]:
```
decodage_de_Cesar('z',texte_chiffre)
```

[23]:
```
"Ztthrt qntrsf tcty eovusy i'zhbnsy, udhvusy uzfgjqaqdg nowsjdg q'vb sttjrnsty
 hbpfgyfmh, dvw y'fshvssty kcetea'jkzrt guss rvthgsssf msy zms qfg gzsfrt, agnr
 gr ssvttgffbz qnffri'us ksf qcaxrs n ts yjqfrs hxto defg."
```

On remarque bien que c’est à nouveau incompréhensible. Il faut trouver une autre stratégie.

Une stratégie consiste à se ramener à la situation d’un chiffrement de César en commençeant par deviner la longueur de la clef. L’idée pour obtenir cette taille de clef et de remarquer que la langue française présente des répétitions de 3 lettres dans différents mots, par exemple les trois lettres “mes” dans mesquin, mesure, termes, … Ces successions de trois lettres sont appelés trigrammes. Et on peut aussi retrouver dans le texte chiffré des trigrammes qui se répètent. Si tel est le cas c’est qu’ils ont été codé par le même trigramme de la clef. La probabilité que deux trigrammes du texte différents encodés par deux trigrammes de la clef différent donne les même trigrammes est faibles.

Quelle relation a-t-on alors entre la distance de répétition entre les trigrammes dans le texte

chiffré et la longueur de la clef ?

Dans le texte chiffré les trigrammes se répètent quand on a une coincidence d'un même trigramme du texte et simultanément un même trigramme de la clef. Il y a donc forcément un nombre entier de longueur de clef entre deux trigrammes. On obtiens donc que la distance de répétition est un multiple de la longueur de la clef.

Comment peut-on alors retrouver la longueur de la clef, à partir du repérage de multiple répétitions dans le texte ?

Grâce aux distances de répétition des trigrammes on possède un ensemble de multiple de la longueur de la clef, on cherche alors un diviseur commun de ces distances. On prendra donc le PGCD de ces distances.

Une fois la longueur de la clef trouvée comment peut-on en déduire la clef ?

Une fois la longueur n de la clef trouvée, on pourra découper le texte en n sous texte chiffré chacun par une seule lettre, et effectuer une analyse en fréquence pour chaque sous texte.

Pour mettre en place l'algorithme de recherche de la longueur de la clef, on ne travaille plus sur les caractères individuellement mais sur les trigrammes.

Ecrire une fonction qui prend en argument un texte et qui retourne une liste dont chaque élément est un nombre entier représentant un trigramme.

[24]:
```
def texte_vers_trigramme(texte):
    liste = texte_vers_liste(texte)
    nombre = np.array([])

    for code in liste :
        if code >= 65 and code <= 90:
            nombre = np.append(nombre,code-65)
        elif code >= 97 and code <= 122:
            nombre = np.append(nombre,code-97)
        elif code in a_accents:
            nombre = np.append(nombre,ord('a')-97)
        elif code in e_accents:
            nombre = np.append(nombre,ord('e')-97)
        elif code in i_accents:
            nombre = np.append(nombre,ord('i')-97)
        elif code in o_accents:
            nombre = np.append(nombre,ord('o')-97)
        elif code in u_accents:
            nombre = np.append(nombre,ord('u')-97)

    taille = len(nombre)
    trigramme = np.zeros(taille-2)
    trigramme[0:taille-2] = nombre[0:taille-2]*(26**2)+nombre[1:
↪taille-1]*26+nombre[2:taille]
    return trigramme
```

Tester votre fonction sur le texte chiffré par la méthode de Vigenère.

[25]:
```
texte_chiffre = chiffre_de_Vigenère(clef,texte)
texte_vers_trigramme(texte_chiffre)
```

[25]:
```
array([16710., 12642., 12340.,  4490., 11299., 12570., 10470.,  8596.,
       12601., 11262., 11614.,  3173., 12212.,  1167., 12769., 15639.,
        2386.,  9327., 14031., 13309., 12097., 15754.,  5362., 16380.,
        4068.,   329.,  8577., 12109., 16044., 12902.,  1528.,  4595.,
       14031., 13309., 12109., 16066., 13472., 16333.,  2842.,  3603.,
        5823., 10805., 17292., 10197.,  1494.,  3705.,  8471.,  9351.,
       14646., 11702.,  5465.,  1497.,  3790., 10660., 13537.,   460.,
       11978., 12644., 12392.,  5836., 11145.,  8572., 11983., 12772.,
       15704.,  4070.,   368.,  9573.,  2857.,  3982., 15663.,  2996.,
        7594.,  4128.,  1893., 14089., 14798., 15669.,  3152., 11668.,
        4593., 13979., 11952., 11983., 12775., 15783.,  6113.,   772.,
        2499., 12271.,  2686., 17117.,  5666.,  6724., 16658., 11289.,
       12317.,  3891., 13303., 11950., 11928., 11354., 13994., 12329.,
        4203.,  3839., 11951., 11938., 11607.,  3007.,  7901., 12114.,
       16183., 16527.,  7893., 11886., 10249.,  2839.,  3534.,  4021.,
       16670., 11612.,  3138., 11309., 12823., 17042.,  3708.,  8533.,
       10962.,  3813., 11275., 11954., 12030., 14006., 12641., 12302.,
        3488.,  2808.,  2728.,   639., 16626., 10456.,  8220.,  2824.,
        3127., 11017.,  5243., 13295., 11743.,  6530., 11611.,  3095.,
       10191.,  1348., 17488., 15305., 11270., 11822.,  8597., 12633.,
       12098., 15771.,  5802., 10260.,  3137., 11264., 11670.,  4646.,
       15353., 12508.,  8843.,  1434.,  2137.])
```

Ecrire une fonction qui prend en argument la liste des trigrammes et retourne une liste de toutes les distances entre répétition de trigramme.

[26]:
```
def distance_repetition(trigramme):
    taille = len(trigramme)
    indices = np.argsort(trigramme)
    trigramme = np.sort(trigramme)
    distances = []
    for i in range(taille-1):
        if trigramme[i] == trigramme[i+1]:
            distances.append(np.abs(indices[i+1]-indices[i]))
    return distances
```

[27]:
```
trigramme = texte_vers_trigramme(texte_chiffre)
distance_repetition(trigramme)
```

[27]:
```
[21, 7, 14, 14]
```

Ecrire une fonction qui prend en argument la liste des distances entre les répétitions et en déduit la longueur de la clef.

[28]:
```
def longueur_clef(distances):
    p = distances[0]
    for element in distances:
        p = np.gcd(element,p)
    return p
```

Testez vos algorithmes de détermination de la longueur de clef sur le texte fournit et chiffré

par la clef “feynman”.

[29]:
```
distances = distance_repetition(trigramme)
longueur_clef(distances)
```

[29]:
```
7
```

2.2 Chiffrement de Vernam

Enfin pour éviter les attaques sur le chiffrement de Vigenère, une stratégie est de choisir une clef de même longueur que le texte à chiffrer et pour éviter toute répétition dans la clef de tirer au sort tous les caractères de la clef.

Mettre en place ce chiffrement et montrer que l’on ne peut pas deviner la clef à partir du message chiffré seuleument.

[30]:
```
import random as rd

def chiffre_de_Vernam(texte):
    liste = texte_vers_liste(texte)
    liste_chiffre = []
    decalage = []
    for code in liste :
        if code >= 65 and code <= 90:
            nombre = rd.randint(0,25)
            liste_chiffre.append((code+nombre-65)%26 + 65)
            decalage.append(nombre+97)
        elif code >= 97 and code <= 122:
            nombre = rd.randint(0,25)
            liste_chiffre.append((code+nombre-97)%26 + 97)
            decalage.append(nombre+97)
        else :
            liste_chiffre.append(code)
            decalage.append(code)
    return (liste_vers_texte(decalage),liste_vers_texte(liste_chiffre))
```

[31]:
```
chiffre_de_Vernam(texte)
```

[31]:
```
("cvluiv wluxfp dizh zgjgjd n'omdiks, zslgsow dlqafhptec jejiébx q'uo cisfkhlci
bchqkpdnp, yfz h'odhgisll jpccub'aaglm twgk igurspmvb tcv gac siu fhxpwt, knpl
cr regvtmygjn mluoko'uw zfs dmkxif à io tqimic uews qièn.",
 "Vjfnmn ysipjh vwma egrznv q'ofruok, oweolso slhtnjjeiu jrruéfp t'ob owmaotppb
jpjuchdai, ozh z'owaozwye udtukv'elrpe lktd lomkscfzt egn ang vmm fbqgal, wnxd
uv iivjneqkwg xzlgai'ij kjk saepaj à as luzdmt nvkh fzèf.")
```

[32]:
```
(clef,texte_chiffre) = chiffre_de_Vernam(texte)
trigramme = texte_vers_trigramme(texte_chiffre)
distance_repetition(trigramme)
```

[32]:
```
[35]
```

On remarque qu’il n’y a pas de répétition dans le message chiffré.

Les analyses en fréquences ne permettent pas de décoder ces messages. On peut le faire seuleument en connaissant la clef

[33]:
```
def decodage_de_Vernam(clef,texte_chiffre):
    liste_chiffre = texte_vers_liste(texte_chiffre)
    liste = []
    decalage = texte_vers_liste(clef)
    i=0
    for code in liste_chiffre :
        if code >= 65 and code <= 90:
            nombre = decalage[i]-97
            liste.append((code-nombre-65)%26 + 65)
            i+=1
        elif code >= 97 and code <= 122:
            nombre = decalage[i]-97
            liste.append((code-nombre-97)%26 + 97)
            i+=1
        else :
            liste.append(code)
            i+=1
    return liste_vers_texte(liste)
```

[34]:
```
decodage_de_Vernam(clef,texte_chiffre)
```

[34]:
```
"Toutes choses sont faites d'atomes, petites particules animées d'un mouvement
incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais
se repoussent lorsqu'on les pousse à se serrer trop près."
```

Quel problème voyez vous à l'utilisation d'un chiffrement de Vernam sur un réseau publique comme le réseau internet ?

Recherchez la solution apportée par le chiffrement RSA.

Pour utiliser un chiffrement de Vernam, il faut transmettre la clef au destinataire du message, mais pour les réseaux publiques la clef peut être tout aussi bien intercepté que le message lui même. Il n'y a pas de confidentialité de la clef.

Le chiffrement RSA est un chiffrement asymétrique, son principe simplifié est basée sur deux clefs une publique et une privée.

Le destinataire du message publie sur le réseau publique la clef publique qui sert uniquement à chiffrer le message. Tout le monde peut alors envoyé un message chiffré au destinataire, mais cette clef publique ne permet pas de déchiffrer les messages.

Seul le destinataire du message possède la clef privée qui permet de déchiffrer les messages et qui demande énormément de temps de calcul à retrouver à partir de la clef publique et du message chiffré. Dans le chiffrement RSA l'opération qui permettrait de retrouver la clef privée serait la factorisation en nombre premier du produit de deux très grands nombres premiers.

3 Transmissions fiables de données et codes correcteurs d'erreur

Lors de la transmission d'information à travers un réseau, des phénomènes physiques aléatoires peuvent modifier le message et générer des erreurs dans le message initialement transmis. Pour assurer le bon fonctionnement des transmissions d'information numérique, il faut être capable de détecter ces erreurs et si possible de les corriger automatiquement.

3.1 Transmission de Bit d'information et bit de parité

La transmission d'information numérique peut toujours se ramener à l'étude de la transmission de code binaire. En effet une information numérique est quantifié sur un nombre fini de niveaux de quantification que l'on peut compter en base 2.

On va dans cette partie étudier la transmission d'un texte :

[35]:
```
texte = "Toutes choses sont faites d'atomes, petites particules animées d'un␣
 ↪mouvement incessant, qui s'attirent lorsqu'elles sont distantes les une des␣
 ↪autres, mais se repoussent lorsqu'on les pousse à se serrer trop près."
print(texte)
```

```
Toutes choses sont faites d'atomes, petites particules animées d'un mouvement
incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais
se repoussent lorsqu'on les pousse à se serrer trop près.
```

Ce texte peut être codé comme une liste d'entier en base 10 grace à la correspondance établie par la norme Unicode. Et les fonctions ord() et chr() en python réalisent cette correspondance.

[36]:
```
print(chr(65))

print(ord('A'))
```

```
A
65
```

Enfin on peut transformer un nombre en base 10 en nombre binaire à l'aide de la fonction bin().

[37]:
```
bin(66)
```

[37]: `'0b1000010'`

On remarque que cette fonction prend en argument un entier en base 10 et ressort le code binaire correspondant sous le format d'une chaîne de caractère: la fonction type renvoie -> str.

[38]:
```
type(bin(66))
```

[38]: `str`

Cette chaîne de caractère est précédée du préfixe '0b' pour tout les codes binaires (il indique qu'il s'agit d'un code en base 2) puis des caractères '1' ou '0' selon la valeur du bit en débutant par le bit de poids le plus fort jusqu'au bit de poids le plus faible.

[39]:
```
print(bin(0))
print(bin(1))
print(bin(2))
```

```
0b0
0b1
0b10
```

Afin de faciliter la manipulation et la visualisation des effets des programmes sur le texte écrire une fonction qui prends en argument une chaine de caractère texte, et qui renvoie une liste

contenant les codes binaires de chaque caractère. Ces codes binaires de chaque caractère seront eux-même encodé sous forme d'une liste de nombre entier 1 ou 0 correspondant aux valeurs des bits du code binaire.

[40]:
```
def texte_vers_liste(texte):
    liste = []
    for lettre in texte :
        liste.append(ord(lettre))
    return liste

def texte_vers_binaire(texte):
    liste = texte_vers_liste(texte)
    binaire = []
    for nombre in liste:
        chaine = bin(nombre)
        code = []
        for digit in chaine[2:]:
            code.append(int(digit))
        binaire.append(code)
    return binaire
```

[41]:
```
texte_vers_binaire(texte)[0:2]
```

[41]:
```
[[1, 0, 1, 0, 1, 0, 0], [1, 1, 0, 1, 1, 1, 1]]
```

[42]:
```
def liste_vers_texte(liste):
    texte = ''
    for code in liste :
        texte+=chr(code)
    return texte

def binaire_vers_texte(binaire):
    liste = []
    for code in binaire :
        nombre = 0
        for digit in code:
            nombre = 2*nombre
            nombre += digit
        liste.append(nombre)
    texte = liste_vers_texte(liste)
    return texte
```

[43]:
```
binaire = texte_vers_binaire(texte)
binaire_vers_texte(binaire)
```

[43]:
```
"Toutes choses sont faites d'atomes, petites particules animées d'un mouvement
incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais
se repoussent lorsqu'on les pousse à se serrer trop près."
```

Pour s'assurer de la transmission d'un message sans erreur une stratégie consiste à rajouter une information redondante dans le message. En effet on peut vérifier après transmission que

cette répétition est toujours cohérente et donc qu'il n'y a pas eu de modification aléatoire d'une partie du message.

Une information redondante que l'on ajoute de manière usuelle à un code binaire est le bit de parité. Par définition ce bit de parité est : - égal à 0, si le code contient un nombre pair de 1 (donc si les bits sont de somme paire) - égal à 1, si le code contient un nombre impair de 1 (donc si les bits sont de somme impaire)

Calculer à la main le bit de parité pour des entiers 3, 16, 255

l'entier 3 en code binaire s'écrit '11' or 1+1 = 2 est pair donc son bit de parité est 0.

l'entier 16 en code binaire s'écrit '10000' or 1+0+0+0+0 = 1 est impair donc son bit de parité est 1.

l'entier 255 en code binaire s'écrit '11111111' or 1+1+1+1+1+1+1+1 = 8 est pair donc son bit de parité est 0.

Ecrire une fonction parite() qui prend en argument un code binaire sous forme de liste de nombre entier 1 ou 0 et retourne la valeur du bit de parité comme l'entier 0 ou 1.

[44]:
```
def parite(code):
    somme= sum(code)
    return somme % 2
```

[45]:
```
print(parite([1,1]))
print(parite([1,0,0,0,0]))
print(parite([1,1,1,1,1,1,1,1]))
```

```
0
1
0
```

3.2 Somme de contrôle (checksums)

Les sommes de contrôle sont des stratégies pour vérifier si le message est transmis sans altération du message ou si le message doit être renvoyé. Le protocole consiste avant émission de calculer la somme des bits présent dans le code binaire, puis de transmettre le code et sa somme en même temps. Après réception du message le destinataire recalcule de son côté la somme des bits du code est vérifie qu'il obtient bien un résultat en accord avec la somme transmise avec le message.

Expliquer pourquoi le bit de parité constitue une somme de contôle.

Ecrire une fonction controle_de_parite() qui prend en argument un code binaire et un bit de parité puis vérifie si le message à été altéré ou pas.

[46]:
```
def controle_de_parite(bit_de_parite,code):
    return bit_de_parite == parite(code)
```

[47]:
```
code = [1,1,1,1,1,1,1,1]
bit_de_parite = parite(code)
controle_de_parite(bit_de_parite,code)
```

[47]:
```
True
```

[48]:
```
code = [1,1,1,1,0,1,1,1]
bit_de_parite = (parite(code)+1)%2
controle_de_parite(bit_de_parite,code)
```

[48]:
```
False
```

Tester votre fonction à l'aide du texte fournit, du calcul du bit de parité, et d'un tirage aléatoire pour savoir si un bit est modifié ou pas.

[49]:
```
def simulation_erreur_aleatoire(texte):
    binaire = texte_vers_binaire(texte)
    for i in range(len(binaire)):
        code = binaire[i]
        if rd.randint(0,5) == 0:
            p = rd.randint(0,len(code)-1)
            code[p] = (code[p]+1)%2
        binaire[i] = code
    texte_avec_erreur = binaire_vers_texte(binaire)
    return texte_avec_erreur
```

[50]:
```
simulation_erreur_aleatoire(texte)
```

[50]:
```
"Toutes cHoses so.d faides d'`tomes, petites parti#unes\x00animées d%uj
louvement0inaascant, QuI s'attisent lorsqu'elles so.t!distantes |Es unE fes
eutreq, mais se repoursdnp lorwqu#on le{ po5sSe à se(sgrrez$trop près."
```

[51]:
```
def texte_vers_binaire_avec_parite(texte):
    binaire = texte_vers_binaire(texte)
    binaire_avec_parite = []
    for i in range(len(binaire)):
        code = binaire[i]
        bit_de_parite = parite(code)
        code_avec_parite = [bit_de_parite]+code
        binaire_avec_parite.append(code_avec_parite)
    return binaire_avec_parite
```

[52]:
```
texte_vers_binaire_avec_parite(texte)[0:2]
```

[52]:
```
[[1, 1, 0, 1, 0, 1, 0, 0], [0, 1, 1, 0, 1, 1, 1, 1]]
```

[53]:
```
def binaire_avec_parite_vers_texte(binaire_avec_parite):
    binaire = []
    for code in binaire_avec_parite:
        binaire.append(code[1:])
    texte = binaire_vers_texte(binaire)
    return texte
```

[54]:
```
binaire_avec_parite = texte_vers_binaire_avec_parite(texte)
binaire_avec_parite_vers_texte(binaire_avec_parite)
```

[54]:
```
"Toutes choses sont faites d'atomes, petites particules animées d'un mouvement
incessant, qui s'attirent lorsqu'elles sont distantes les une des autres, mais
se repoussent lorsqu'on les pousse à se serrer trop près."
```

[55]:
```
def simulation_controle_erreur_aleatoire(texte):
    binaire_avec_parite = texte_vers_binaire_avec_parite(texte)
```

```
    for i in range(len(binaire_avec_parite)): #génération d'erreur aléatoire
        code = binaire_avec_parite[i]
        if rd.randint(0,5) == 0:
            p = rd.randint(0,len(code)-1)
            code[p] = (code[p]+1)%2
        binaire_avec_parite[i] = code

    texte_avec_erreur = binaire_avec_parite_vers_texte(binaire_avec_parite)

    binaire_sans_erreur = []
    for i in range(len(binaire_avec_parite)):
        code = binaire_avec_parite[i][1:]
        bit_de_parite = binaire_avec_parite[i][0]
        if controle_de_parite(bit_de_parite,code) == True:
            binaire_sans_erreur.append(code)
        else :
            binaire_sans_erreur.append([1,0,0,0,0,0])

    texte_sans_erreur = binaire_vers_texte(binaire_sans_erreur)
    return (texte_avec_erreur, texte_sans_erreur)
```

[56]:
```
simulation_controle_erreur_aleatoire(texte)
```

[56]:
```
("Uoutes chOses so.t fa)|ms0d'!tOmer,(petites 0articules animéec d/un mouvement
incessant,\x00quk s'cttirent0lorsqu7elles sont diqtin|us les\x00une tes autres,
mais!se(vepous3ent lopsqu'oo |es!pousse à\x00se serreR Trop près.",
 " outes c  ses so t fa   s d' t me , petites  articules animée  d un mouvement
incessant, qu  s' ttirent lorsqu elle  sont di t n    les une  es autres, m is
se  epous ent lo s u'o   es pousse à s  serre   rop p ès.")
```

Quelle est la limite de cette technique de somme de contrôle ? Donner un exemple de mauvaise transmission qui n'est pas détectable par la technique du bit de parité. Puis montrer numériquement l'erreur qui survient.

Cette technique ne peut pas détecter d'erreur lorsque deux bits sont modifiés aléatoirement, car la parité est alors inchangée.

[57]:
```
def simulation_double_erreur_aleatoire(texte):
    binaire_avec_parite = texte_vers_binaire_avec_parite(texte)

    for i in range(len(binaire_avec_parite)): #génération d'erreur aléatoire␣
↪avec double erreurs
        code = binaire_avec_parite[i]
        if rd.randint(0,5) == 0:
            p = rd.randint(0,len(code)-1)
            code[p] = (code[p]+1)%2
        if rd.randint(0,5) == 0:
            p = rd.randint(0,len(code)-1)
            code[p] = (code[p]+1)%2
        binaire_avec_parite[i] = code
```

```
    texte_avec_erreur = binaire_avec_parite_vers_texte(binaire_avec_parite)

    binaire_sans_erreur = []
    for i in range(len(binaire_avec_parite)):
        code = binaire_avec_parite[i][1:]
        bit_de_parite = binaire_avec_parite[i][0]
        if controle_de_parite(bit_de_parite,code) == True:
            binaire_sans_erreur.append(code)
        else :
            binaire_sans_erreur.append([1,0,0,0,0,0])

    texte_sans_erreur = binaire_vers_texte(binaire_sans_erreur)
    return (texte_avec_erreur, texte_sans_erreur)
```

[58]:
```
simulation_double_erreur_aleatoire(texte)
```

[58]:
```
("Toute3 #hoses son$$fAitec d'au/mes,0pepitds pazticuler aninâeq d'uˆ eguvGment
)n#eryant,$quy q'atTireNt lopSqu'elles wonu distantes$`es(une!hes aetres, mci{
se repoussent(dobsqu'on dew4pmusse à cE serrez tRo0 pvÈs.",
 "Toute   hoses son$ f ite  d'a /mes, pe it s pa ticule  anin e  d'uˆ   uvGment
n e yant, quy  'at ire t l   qu'elles  on  distantes `es une hes a tres  m i  se
repousse    o squ'on  e 4p u se à    serre  t o  p  s.")
```

On remarque des erreurs ont quand même réussit à passer le contrôle du bit de parité.

Quelle stratégie permet de limiter l'apparition de cette erreur indétectable ?

Effectuer souvent les contrôle de parité lors de la chaîne de transmission pour rendre la probabilité de double erreur très faible.

3.3 Code de Hamming

Le code de Hamming est un autre protocole d'utilisation des bits de parité pour détecter les erreurs mais aussi pour les corriger automatiquement sans avoir besoin de retransmettre le message.

Un exemple de code de Hamming est le code dit (7,4), qui consiste à envoyer tout les 7 bits : 4 bits de données du message (d_1, d_2, d_3, d_4) et 3 bits de parité (p_1, p_2, p_3). Les bits de parité sont définis comme: - p_1 est le bit de parité de (d_1, d_2, d_4) - p_2 est le bit de parité de (d_1, d_3, d_4) - p_3 est le bit de parité de (d_2, d_3, d_4)

Ecrire une fonction encode_hamming() qui prend en argument une liste de 4 bit de donnée et retourne une liste des 7 bits de donnée et de parité.

[59]:
```
def encode_hamming(donnee):
    p1 = parite([donnee[0],donnee[1],donnee[3]])
    p2 = parite([donnee[0],donnee[2],donnee[3]])
    p3 = parite([donnee[1],donnee[2],donnee[3]])
    code = [p1, p2, donnee[0], p3]+donnee[1:]
    return code
```

[60]:
```
donnee = [1,0,1,1]
encode_hamming(donnee)
```

[60]:
```
[0, 1, 1, 0, 0, 1, 1]
```

Le contrôle du message après réception peut dans un premier temps consister à recalculer les 3 bits de parité et vérifier s'il n'y a pas eut d'altération. Mais la technique porposée par Hamming consiste à calculer 3 autres bits de parités. A partir du message $(m_1, m_2, m_3, m_4, m_5, m_6, m_7) = (p_1, p_2, d_1, p_3, d_2, d_3, d_4)$ on calcule : - c_1 le bit de parité de (m_4, m_5, m_6, m_7) - c_2 le bit de parité de (m_2, m_3, m_6, m_7) - c_3 le bit de parité de (m_1, m_3, m_5, m_7)

On peut montrer alors que si le message a bien été encodé et transmis sans altération alors les trois bits de contrôle doivent être égal à 0. Si ce n'est pas le cas alors il y a une erreur de transmission.

Verifier cette affirmation en calculant numériquement les bits de parité c en fonction d'un message m non altéré ou altéré.

[61]:
```
def bit_de_controle(message):
    c1 = parite([message[3],message[4],message[5],message[6]])
    c2 = parite([message[1],message[2],message[5],message[6]])
    c3 = parite([message[0],message[2],message[4],message[6]])
    return [c1, c2, c3]
```

[62]:
```
donnee = [1,0,1,1]
message = encode_hamming(donnee)
bit_de_controle(message)
```

[62]:
```
[0, 0, 0]
```

[63]:
```
donnee = [1,0,1,1]
message = encode_hamming(donnee)
p = rd.randint(0,6)
message[p] = (message[p]+1)%2
p+1,bit_de_controle(message)
```

[63]:
```
(5, [1, 0, 1])
```

Dans le cas d'une erreur unique on remarque non seuleument que les bits de contrôle c sont différents de $(0,0,0)$ mais qu'ils indiquent aussi en binaire la position du bit affecté par l'erreur. Par exemple si $c = (0,1,1)$ alors l'erreur porte sur le troisième bit du message. On peut alors corriger automatiquement l'erreur.

Ecrire une fonction decode_hamming() qui prend en argument une liste de 7 bits et retourne une liste de 4 bit de données décodés. En cas d'erreur, on renverra un message d'avertissement indiquant la position du bit affecté et on effectuera la correction.

[64]:
```
def decode_hamming(message):
    c = bit_de_controle(message)
    if c == [0,0,0] :
        return [message[2]]+message[4:]
    else :
        p = 4*c[0]+2*c[1]+c[2]
        message[p-1] = (message[p-1]+1)%2
        print('erreur corrigée sur le bit '+str(p))
        return [message[2]]+message[4:]
```

[65]:
```
donnee = [1,0,1,1]
message = encode_hamming(donnee)
decode_hamming(message)
```

[65]:
```
[1, 0, 1, 1]
```

[66]:
```
donnee = [1,0,1,1]
message = encode_hamming(donnee)
p = rd.randint(0,6)
message[p] = (message[p]+1)%2
p+1,decode_hamming(message)
```

```
erreur corrigée sur le bit 6
```

[66]:
```
(6, [1, 0, 1, 1])
```

Déterminer le codage de Hamming de la donnée 1011, puis la donnée décodée par l'algorithme dans l'hypothèse où les deux premiers bits du message codé ont été incorrectement transmis. Quel a été l'effet de la correction automatique sur les bits de donnée dans ce cas ?

[67]:
```
donnee = [1,0,1,1]
message = encode_hamming(donnee)
message[0] = (message[0]+1)%2
message[1] = (message[1]+1)%2
decode_hamming(message)
```

```
erreur corrigée sur le bit 3
```

[67]:
```
[0, 0, 1, 1]
```

La correction automatique a ajouté une erreur dans le code de départ.

Proposer une méthode simple de différencier une double erreur d'une erreur unique au moyen d'un bit de parité supplémentaire et expliquer comment cela permet d'éviter le problème mis en évidence à la question précédente. On ne cherchera pas à corriger la double erreur, mais juste à la détecter.

On peut rajouter un bit de parité sur les 7 bits du message: - s'il n'y a pas d'erreur on aura c = [0,0,0] et le test de parité sera validé - s'il y a une erreur sur les 7 bits du message on aura c qui donne la position de l'erreur et le controle de parité qui indique une erreur - s'il y a une erreur sur le bit de parité on aura c qui donne la position 0 et le controle de parité qui indique une erreur - s'il y a une double erreur on aura c qui donne une position non nule et le controle de parité qui n'indique pas d'erreur.

On peut donc toujours corriger les erreurs n'affectant que 1 bit avec le bit de parité compris, et on peut en plus détecter les doubles erreurs c'est le code de Hamming dit (8,4)