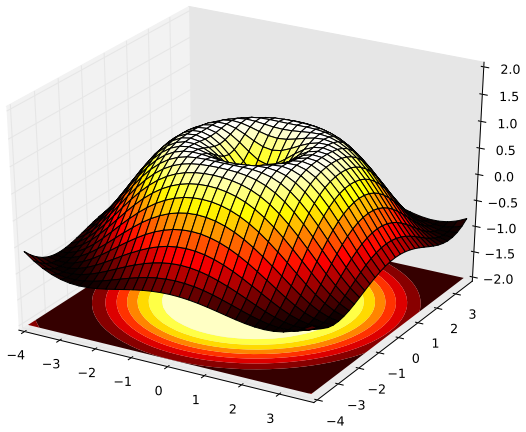
Informatique

Saint Joseph La Salle LORIENT

Marc Beutier
obelix56@free.fr
http://obelix56.free.fr

cpge TSI

Établissement St Joseph - LaSalle Lorient



book_20-06_python_eleve_annee2 30 juin 2020

Python & Scilab

Version Élève

Ce fascicule s'adresse avant tout à vous, étudiants de première année de CPGE. J'espère qu'il pourra vous servir de base pour l'année et pourquoi pas pour les années suivantes ?
Il sera mis à jour sur

http://obelix56.free.fr

Vous trouverez également sur ce site les dossiers *eleve* dans lequels se trouvent tous les programmes écrits ici en *python* (*.py*) et *scilab* (*.sci* et *.sce*).
Les programmes proposés durant les activités et ceux demandés en exercices ne sont pas disponibles, pour le bon déroulement des séances ...
La progression de l'année sera calquée sur ce fascicule et comme nous ne trouvons le temps de tout faire, vous pourrez combler vos lacunes le soir, avant de vous endormir ...

Le fascicule existe rassemble le programme des deux années en algorithmique, sur *Scilab* , sur *Python* et sur le traitement d'images.

Je ne peux remercier toutes les personnes qui m'ont aidé à rédiger ce fascicule. Entre les collègues, les auteurs d'ouvrages de référence, les tutoriels, FAQ et autres forums, je tiens particulièrement à remercier Arnaud Coatanhay et Christophe Osswald, enseignants - chercheurs à l'ENSTA Bretagne, pour leur disponibilité et la qualité de leur formation en *Python* et Philippe Roux, Jean-Paul Chehab, Jérôme Briot et Michael Baudin pour le partage de leurs ouvrages en format LaTeXconcernant *Scilab*.
"Une introduction à Python 3" écrit par Bob Cordeau et Laurent Pointal a aussi certainement inspiré mon travail. Leurs sources LaTeXet *Python* sont disponibles sur leur site. On peut trouver cela ici :

http://perso.limsi.fr/pointal/python:courspython3

Pour ce qui est des forums, celui que j'utilise régulièrement est :

http://www.developpez.net/forums/f96/autres-langages/python-zope/

En cas de réclamation, veuillez me contacter à l'adresse indiquée sur la première page.

Table des matières

Partie 1

Python année 2

Table des matières

1

Traitement d'image

Objectifs

Les capacités évaluées dans cette partie de la formation sont :

- programmer un algorithme dans un langage de programmation moderne et général,
- modifier un algorithme existant pour obtenir un résultat différent,
- concevoir un algorithme répondant à un problème précisément posé,
- expliquer le fonctionnement d'un algorithme,
- comprendre le fonctionnement d'un algorithme récursif et l'utilisation de la mémoire lors de son exécution,
- comprendre les avantages et défauts respectifs des approches récursive et itérative,
- s'interroger sur l'efficacité algorithmique temporelle d'un algorithme,
- distinguer par leurs complexités deux algorithmes résolvant un même problème.

1.1 Généralités

- Un *bit* (abréviation de *binary digit*, signifiant chiffre binaire en français) correspond à la plus petite unité élémentaire de stockage d'informations en informatique, soit un chiffre binaire 0 ou 1. Le symbole de bit est b.
- Un *byte* signifie *octet* en français. Un *octet* (*byte*) est constitué de 8 bits. Le symbole de byte est B.
- Le symbole de *octet* est o.

Remarque

Il ne faut donc pas confondre les unités de stockage :

1 B	= 1 o	= 8 b
1 kB	= 1 ko	= 8 kb
1 MB	= 1 Mo	= 8 Mb

Une image matricielle est aussi appelée image ponctuelle.

Un "bitmap" est un rectangle de "pixels" (*pixel* pour l'abréviation de *picture element*), donc une matrice.

Une image en "Noir et Blanc" contient des pixels qui sont codés sur 1 bit : 0 ou 1 (ou encore 0 ou 255).

Une image en niveaux de gris contient des pixels qui sont eux-mêmes codés sur 1 octet = 8 bits, c'est-à-dire de 0 à 255 : 256 = 2^8. Cette image en niveaux de gris est donc une liste de nombres compris entre 0 et 255. Le nombre d'éléments de cette liste est égal au nombre de pixels, 64 par exemple pour une image de 8 x 8.

Une image en couleurs contient des pixels contenant, chacun, une liste de 3 nombres compris entre 0 et 255, chaque liste correspondant à 3 couleurs : rouge, vert et bleu pour le mode "RGB" (red, green, blue). Une image en couleurs est donc une liste de listes de 3 termes.

Chaque pixel est alors codé sur 3 x 8 = 24 bits = 3 octets. Cela correspond à un choix de :
$\left(2^8\right)^3 = 2^{24} = 16777216$ couleurs, soit environ 16,7 millions de couleurs.

Définition

La définition d'une image est le nombre de pixels total, c'est-à-dire sa "dimension informatique" (le nombre de colonnes de l'image multiplié par son nombre de lignes). Ainsi, une image possédant 3000 pixels en largeur et 2000 en hauteur aura une définition de 3000 pixels par 2000, notée 3000 x 2000 = 6 Mpx. Puisqu'un pixel est codé 3 octets, cette image pèse 18 Mo (1 $Mo = 10^6$ octets).

Résolution

La résolution d'une image composée de points est définie par la densité des points par unité de surface. La résolution permet de définir la finesse de l'image. Plus la résolution est grande, plus la finesse de l'image est grande.
Sur les écrans on parle de "pixel" alors que sur le papier, on parle de "points" où "dots".
Ainsi, la résolution dans le domaine de l'écran est *ppi* (pixels per inch : ppp en français, soit pixels par pouce) alors que la résolution sur le papier est *dpi* (dots per inch : points par pouce) : 1 pouce (*inch* en anglais) = 2,54 cm.

On a la relation : $\text{résolution} = \dfrac{\text{définition}}{\text{dimension}}$

Il existe beaucoup de possibilités pour réaliser du traitement d'image sous *Python*. Par exemple, sous *MatPlotLib*, certaines fonctionnalités permettent de traiter les images. Il existe aussi des bibliothèques dédiées au traitement d'images de niveau professionnel avec lesquelles on peut travailler sous *Python* (par exemple OpenCV [1]).

Dans un premier temps, nous passerons en revue quelques opérations réalisables sous *PIL* ainsi que *Scipy* et *Matplotlib*. Ensuite, nous verrons comment traiter les images en parcourant les pixels, écrits sous forme de matrices.

L'avantage principal de la bibliothèque *PIL* est sa simplicité de mise œuvre.

On pourra importer le module *Image* de la bibliothèque *PIL* par la commande suivante :

```
from PIL import Image
```

⚠ Il faudra éviter les conflits avec d'autres bibliothèques (*tkinter* par exemple). En cas de message d'insultes de la part de *Python*, il faudra penser à ce type d'erreur possible.

1.2 Le module *PIL*

1.2.1 Installation du module

1. http://docs.opencv.org/trunk/doc/py_tutorials/py_tutorials.html

1.2.1.1 Sous *Ubuntu*

En principe, les commandes en console *linux* doivent fonctionner :

```
sudo apt-get install python3-setuptools

sudo easy_install3 pip

sudo apt-get install python3-dev

 wget https://github.com/python-imaging/Pillow/archive/master.zip

sudo unzip master.zip

ls


ls*

cd Pillow-master/

sudo python3 setup.py build

sudo python3 setup.py install

exit

sudo apt-get install libtiff4-dev libjpeg8-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.5-dev tk8.5-dev python-tk

cd Pillow-master/

sudo python3 setup.py build

cd ..

sudo apt-get install python3-pip

sudo pip install -I pillow
```

1.2.1.2 Sous *Windows*

En principe, le téléchargement et l'installation du fichier téléchargeable ici :

https://pypi.python.org/pypi/Pillow/2.5.3

doit suffire.

Il faut aller au bas de la page et choisir la version qui vous convient parmi les fichiers suivants :

- Pillow-2.5.3-py3.2-win-amd64.egg (md5)
- Pillow-2.5.3-py3.2-win32.egg (md5)
- Pillow-2.5.3-py3.3-win-amd64.egg (md5)
- Pillow-2.5.3-py3.3-win32.egg (md5)
- Pillow-2.5.3-py3.4-win-amd64.egg (md5)
- Pillow-2.5.3-py3.4-win32.egg (md5)

Vous trouverez la version pour *Python 3.3* dans le dossier d'installation.

1.2.2 Ouverture d'un fichier image

La méthode `open(chemin)` permet d'ouvrir une image. Les formats supportés par la bibliothèque *PIL* sont :

- "BMP" : extension *.bmp* ou *.dib*,
- "DCX" : extension *.dcx*,
- "EPS" : extension *.eps* ou *.ps*,
- "GIF" : extension *.gif*,
- "IM" : extension *.im*,
- "JPEG" : extension *.jpg*, *.jpe* ou *.jpeg*,
- "PCD" : extension *.pcd*,
- "PCX" : extension *.pcx*,
- "PDF" : extension *.pdf*,
- "PNG" : extension *.png*,
- "PPM" : extension *.pbm*, *.pgm* ou *.ppm*,
- "PSD" : extension *.psd*,
- "TIFF" : extension *.tif* ou *.tiff*,
- "XBM" : extension *.xbm*,
- "XPM" : extension *.xpm*.

La méthode `getdata()` retourne un objet-séquence contenant les valeurs des pixels de l'image. Cependant il n'est lisible que par *PIL*. La fonction `list()` permet ensuite de récupérer cette séquence sous un format lisible par l'utilisateur. On récupère alors une liste de tuples à 3 composantes si l'image est couleur, une liste simple sinon.

```
mon_image = Image.open("tux20.jpg")
donnees = list(mon_image.getdata())
```

Exemple avec le programme suivant :

data1.py

```
1 from PIL import Image
2
3 mon_image = Image.open("test.png")
4 donnees = list(mon_image.getdata())
5 print(donnees)
```

qui donne :

```
[(109, 148, 114), (109, 148, 114), (109, 148, 114), (167, 57, 51),
(109, 148, 114), (109, 148, 114), (109, 148, 114), (109, 148, 114),
(239, 226, 21), (109, 148, 114), (109, 148, 114), (109, 148, 114)]
```

1.2.3 Visualisation d'une image

Il faut utiliser la méthode `show()` :

```
mon_image.show()
```

1.2.4 Taille d'une image

taille

La taille d'une image correspond à son nombre de colonnes (largeur) et son nombre de lignes (hauteur).

La fonction `size` permet d'accéder à ce tuple (largeur, hauteur) :

```
largeur,hauteur = mon_image.size
```

On obtient ici :

```
4 3
```

Remarque

`largeur` correspond à `mon_image.size[0]` et `hauteur` à `mon_image.size[1]`

1.2.5 Bounding box

Bounding box

Une *bounding box* est un rectangle contenu dans l'image. Elle est définie par un tuple de 4 termes (x_0, y_0, x_1, y_1). (x_0, y_0) sont les coordonnées du point qui est en haut à gauche alors que(x_1, y_1) correspondent à celles du coin en bas à droite.

Cela peut servir par exemple à extraire ou copier une partie de l'image.

1.2.6 Autres propriétés de l'image

On peut accéder à d'autres propriétés de l'image à l'aide des commandes `format`, `mode`, `palette` et `info`. Ainsi, les instructions suivantes :

```
print(mon_image.format)
print(mon_image.mode)
print(mon_image.palette)
print(mon_image.info)
```

donnent :

```
JPEG
RGB
None
{'jfif_density': (35, 35), 'jfif_version': (1, 1), 'jfif': 257, 'jfif_unit': 2}
```

Le mode définit la façon dont sont définies les couleurs. Chaque mode est représenté par une chaîne.
Le nombre de bandes est un ensemble de valeurs qui définit un pixel (1 bande pour le noir et blanc, 3 bandes pour le mode "RGB", ...). Les principaux modes sont :

Mode	bandes	Description
"1"	1	Noir et blanc (monochrome) : 1 bit par pixel
"L"	1	Niveaux de gris : 1 byte de 8 bits par pixel
"RGB"	3	Mode couleurs RGB : 3 bytes de 8 bits par pixel
"RGBA"	4	Mode couleurs RGB avec une bande de transparence A : 4 bytes par pixel. Le canal A varie de 0 pour transparent à 255 pour opaque
"CMYK"	4	Mode Cyan - Magenta - Yellow - Black : 4 bytes par pixel
"YCbCr"	3	Format de couleur spécial : [2]3 bytes de 8 bit par pixel
"I"	1	pixels 32 bit entier
"F"	1	pixels 32 bit flottant

TABLE 1.1 – Modes d'une image

1.2.7 Création d'une image à partir d'une liste de valeurs de pixels

La méthode `new(mode, taille)` permet de créer une nouvelle image où *mode* ("L" ou "RGB", par exemple) définit si l'image sera en niveaux de gris ou en couleurs et *taille* est un tuple (*nombre de colonnes*, *nombre de lignes*), donc (*largeur*, *hauteur*).
La méthode `putdata(données)` remplit l'image avec la séquence de valeurs *données*.
Dans l'exemple suivant, on reconstitue une image identique à l'originale :

```
ma_nouvelle_image=Image.new('RGB',(largeur,hauteur))
ma_nouvelle_image.putdata(donnees)
ma_nouvelle_image.show()
```

1.2.8 Décomposition d'une image couleur en ses 3 composantes *RGB*

La méthode `split()` renvoie une séquence des 3 composantes RGB de l'image :

```
## Récupération des différentes composantes de l'image
r,g,b = ma_nouvelle_image.split()
## Sauvegarde des différentes composantes de l'image
r.save('red.jpg')
g.save('green.jpg')
b.save('blue.jpg')
## Visualisation de la composante rouge
Image.open('red.jpg').show()
```

(a) rouge (b) vert (c) bleu

FIGURE 1.1 – Niveaux de rouge, vert et bleu de l'image originale

2. http://fr.wikipedia.org/wiki/YCbCr

1.2.9 Composition d'une image à partir des ses 3 composantes *RGB*

La méthode `merge(mode,bands)` crée une image à partir des images *bands* selon le mode précisé :

```
## décomposition de l'image
ma_composition=r,g,b
 ## Recomposition de l'image
mon_image_recomposee = Image.merge('RGB',ma_composition)
## Sauvegarde de l'image décomposée puis recomposée
mon_image_recomposee.save("tux20c.jpg")
## Visualisation de l'image recomposée
Image.open('tux20c.jpg').show()
```

Les différentes instructions qui précèdent sont intégrées au programme *ouverture.py* : celui-ci est imprimé page 25.

1.2.10 Conversion de format

Avec *PIL*, il est possible de convertir une image sous un autre format, par exemple : "png", "jpeg", "bmp", "eps", ... avec la méthode `save()`.

```
## Conversion de format
r.save('red.eps')
g.save('green.eps')
b.save('blue.eps')
```

1.2.11 Permutation des bandes

Une fois que l'on a accès aux bandes "RGB", il est facile de les permuter. Exemple avec le programme suivant :

permute_bandes.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  mon_image = Image.open("billes.jpg")
6  mon_image.save("billes.eps")
7
8  def permute_bandes1(im):
9      """permutation circulaire des composantes RGB"""
10     rouge,vert,bleu = im.split()
11     return Image.merge("RGB",(bleu,rouge,vert))
12
13 def permute_bandes2(im):
14     """permutation circulaire des composantes RGB"""
15     rouge,vert,bleu = im.split()
16     return Image.merge("RGB",(vert,bleu,rouge))
17
18 nouvelle_image1 = permute_bandes1(mon_image)
19 nouvelle_image1.save('billes_perm1.eps')
20 nouvelle_image1.show()
21
22 nouvelle_image2 = permute_bandes2(mon_image)
23 nouvelle_image2.save('billes_perm2.eps')
24 nouvelle_image2.show()
```

Le résultat :

(a) billes (b) billes_perm1 (c) billes_perm2

FIGURE 1.2 – Permutation des bandes

1.2.12 Conversion de mode

Avec *PIL*, il est également possible de convertir une image dans un autre mode avec la méthode `convert()`, par exemple : "L" pour niveaux de gris, "1" pour monochrome, ...

```
## Conversion de mode
image_gris = mon_image.convert('L')
image_gris.save('tux20_gris.eps')

image_NetB = mon_image.convert('1')
image_NetB.save('tux20_NetB.bmp')
```

On obtient :

(a) couleurs RGB (b) niveaux de gris (c) monochrome

FIGURE 1.3 – Niveaux de gris et monochrome

1.2.13 Image miniature

Pour créer une miniature à partir de l'image originale, on peut utiliser la méthode `resize(taille,resample)`. La *taille* est un tuple (largeur, hauteur) et *resample* peut être précisé pour choisir le mode d'interpolation. Par défaut, c'est `NEAREST` qui est choisi. Les 4 filtres applicables sont :

- NEAREST,
- BILINEAR,
- BICUBIC,
- ANTIALIAS.

```
## Miniature
retaillee=mon_image.resize((128,128),resample=Image.BILINEAR)
retaillee.show()
```

1.2.14 Opérations sur un répertoire

Pour réaliser par exemple l'opération précédente sur tout un répertoire, on peut utiliser le programme :

operation_rep.py

```
1  # -*- coding: utf-8 -*-
2  from PIL import Image
3  import glob, os
4
5  size = (128, 128)
6
7  for fichier in glob.glob("*.png"):
8      nom_fichier, ext = os.path.splitext(fichier)
9      mon_image = Image.open(fichier)
10     mon_image.thumbnail(size, Image.ANTIALIAS)
11     mon_image.save(nom_fichier + "_small.png", "PNG")
```

Bien sûr, l'importation de *glob* et de *os* peut servir à bien d'autres utilisations : ces modules permettent d'explorer et manipuler les répertoires, sous-répertoires ainsi que les fichiers.

1.2.15 Transformations isométriques

On peut réaliser une rotation à l'aide de la fonction `rotate()` et réaliser d'autres transformations à l'aide de la fonction *transpose(méthode)*. Ces opérations sont visibles dans le programme suivant :

rotation.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  mon_image = Image.open('tux20.jpg')
6  sortie = mon_image.rotate(45)
7  sortie.save('tourne.jpg')
8  sortie.save('tux20_rot45.eps')
9  sortie.show()
10
11 sortie = mon_image.transpose(Image.FLIP_LEFT_RIGHT)
12 sortie.save('tux20_flip_left_right.eps')
13 sortie.show()
14
15 sortie = mon_image.transpose(Image.FLIP_TOP_BOTTOM)
16 sortie.save('tux20_flip_top_bottom.eps')
17 sortie.show()
18
19 sortie = mon_image.transpose(Image.ROTATE_270)
20 sortie.save('tux20_rotate_270.eps')
21 sortie.show()
```

On obtient les images suivantes :

(a) rotate(45) (b) FLIP_LEFT_RIGHT (c) FLIP_TOP_BOTTOM (d) ROTATE_270

FIGURE 1.4 – rotate et transpose

Les opérations possibles avec `transpose()` sont :

- sortie = *mon_image.transpose(Image.FLIP_LEFT_RIGHT)*
- sortie = *mon_image.transpose(Image.FLIP_TOP_BOTTOM)*
- sortie = *mon_image.transpose(Image.ROTATE_90)*
- sortie = *mon_image.transpose(Image.ROTATE_180)*
- sortie = *mon_image.transpose(Image.ROTATE_270)*

1.2.16 Filtrage

Le filtrage d'une image se réalise à l'aide de la fonction `filter()` de la sous-bibliothèque *ImageFilter* qui propose de nombreux filtres (filtres min, max, médian, flou, ...). La liste des filtres est indiquée ci-dessous :

- ImageFilter.BLUR,
- ImageFilter.CONTOUR,
- ImageFilter.DETAIL,
- ImageFilter.EDGE_ENHANCE,
- ImageFilter.EDGE_ENHANCE_MORE,
- ImageFilter.EMBOSS,
- ImageFilter.FIND_EDGES,
- ImageFilter.SMOOTH,
- ImageFilter.SMOOTH_MORE,
- ImageFilter.SHARPEN.

Des exemples sont présentés dans le programme suivant dont un utilisant `ImageChops` :

filtres.py

```
1  #-*- coding: utf-8 -*-
2
3  from PIL import Image
4  from PIL import ImageChops
5  from PIL import ImageFilter
6
7  Im = Image.open('billes.jpg')
8  # print(Im.format, Im.size, Im.mode)
9
10 # binarisation
11 ImBW = Im.convert('1')
12 ImBW.save('billes_BW.bmp')
13 ImBW.show()
14
15 # niveaux de gris
16 ImL = Im.convert('L')
17 ImL.save('billes_gris.eps')
18 ImL.show()
19
20 # inversion d'image (négatif)
21 ImInv = ImageChops.invert(Im)
22 ImInv.save('billes_Inv.eps')
23 ImInv.show()
24
25 # BLUR filter
26 ImBlur = Im.filter(ImageFilter.BLUR)
```

```
27  ImBlur.save('billes_BLUR.eps')
28  ImBlur.show()
29
30  # CONTOUR filter
31  ImContour = Im.filter(ImageFilter.CONTOUR)
32  ImContour.save('billes_CONTOUR.eps')
33  ImContour.show()
34
35  # DETAIL filter
36  ImDetail = Im.filter(ImageFilter.DETAIL)
37  ImDetail.save('billes_DETAIL.eps')
38  ImDetail.show()
39
40  # EDGE_ENHANCE filter
41  ImEH = Im.filter(ImageFilter.EDGE_ENHANCE)
42  ImEH.save('billes_EDGE_ENHANCE.eps')
43  ImEH.show()
44
45  # EDGE_ENHANCE_MORE filter
46  ImEHM = Im.filter(ImageFilter.EDGE_ENHANCE_MORE)
47  ImEHM.save('billes_EHM.eps')
48  ImEHM.show()
49
50  # EMBOSS filter
51  ImEmb = Im.filter(ImageFilter.EMBOSS)
52  ImEmb.save('billes_EMBOSS.eps')
53  ImEmb.show()
54
55  # FIND_EDGES filter
56  ImFed = Im.filter(ImageFilter.FIND_EDGES)
57  ImFed.save('billes_EDGES.eps')
58  ImFed.show()
59
60  # SMOOTH filter
61  ImSm = Im.filter(ImageFilter.SMOOTH)
62  ImSm.save('billes_SMOOTH.eps')
63  ImSm.show()
64
65  # SMOOTH_MORE filter
66  ImSmm = Im.filter(ImageFilter.SMOOTH_MORE)
67  ImSmm.save('billes_SMOOTH_MORE.eps')
68  ImSmm.show()
69
70  # SHARPEN filter
71  ImSh = Im.filter(ImageFilter.SHARPEN)
72  ImSh.save('billes_SHARPEN.eps')
73  ImSh.show()
74
75  # MinFilter filter
76  ImMin = Im.filter(ImageFilter.MinFilter)
77  ImMin.save('billes_MinFilter.eps')
78  ImMin.show()
79
80  # MedianFilter filter
81  ImMed = Im.filter(ImageFilter.MedianFilter)
82  ImMed.save('billes_MedianFilter.eps')
83  ImMed.show()
84
85  # MaxFilter filter
86  ImMax = Im.filter(ImageFilter.MaxFilter)
```

```
87  ImMax.save('billes_MaxFilter.eps')
88  ImMax.show()
```


et le résultat :

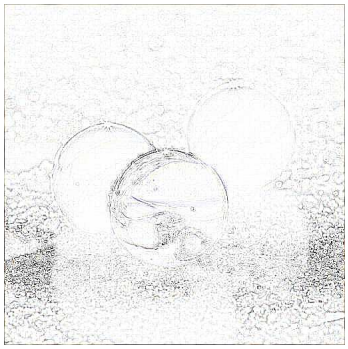
(a) original

(b) convert('1')

(c) convert('L')

(d) Invert

(e) BLUR

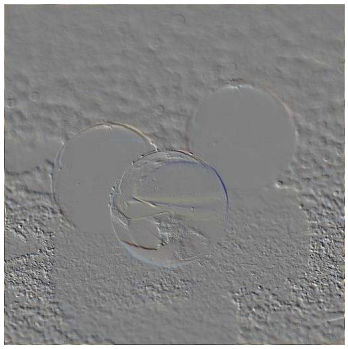


(f) CONTOUR

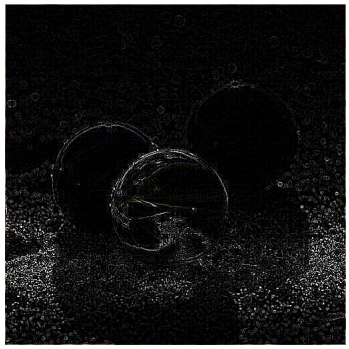
(g) DETAIL

(h) EDGE_ENHANCE

(i) EHM



(j) EMBOSS



(k) FIND_EDGES

(l) SMOOTH

(m) SMOOTH_MORE (n) SHARPEN (o) MinFilter

(p) MedianFilter (q) MaxFilter

FIGURE 1.5 – ImageFilter

Remarque

`ImageChops` permet de comparer deux images, de les soustraire (`difference()`), de les additionner (`add()`), de les copier (`duplicate`), de les surimprimer, de les comparer, ...

Vous pouvez tester si vous voulez, le programme suivant :

addition-image.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4  from PIL import ImageChops
5
6  tour = Image.open('tourne.jpg')
7  tux20 = Image.open('tux20.jpg')
8
9  res = ImageChops.add(tour,tux20,1,0)
10 res.save('addition.eps')
11 res.show()
```

qui donne :

FIGURE 1.6 – Addition

1.2.17 Autres Exemples de manipulations

1.2.17.1 Concaténation d'images

Pour concaténer une image avec son symétrique, on peut regarder le programme ci-dessous :

miroir.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4  mon_image = Image.open("tux20.jpg")
5
6  largeur,hauteur=mon_image.size
7  box = (0, 0, largeur, hauteur)
8  src = mon_image.crop(box)
9  sortie = mon_image.resize((largeur*2,hauteur))
10 sortie.paste(src,(0,0,largeur,hauteur))
11 src=mon_image.transpose(Image.FLIP_LEFT_RIGHT)
12 sortie.paste(src,(largeur,0,2*largeur,hauteur))
13 sortie.save('tux20_miroir.eps')
14 sortie.show()
```

FIGURE 1.7 – Image et symétrique

1.2.17.2 Dessiner sur une image

Le module de dessin de *PIL*, *ImageDraw*, contient différentes méthodes. On peut en citer quelques unes :

- `arc (bbox, début, fin, couleur de trait)` (par défaut, la couleur est `white`) qui trace un arc de cercle,
- `ellipse (bbox, couleur intérieure, couleur de trait)` pour représenter une ellipse,
- `line (points extrêmes, couleur de trait)`,
- `point ((x, y), couleur)` pour représenter un point,
- `polygon (points extrêmes, couleur intérieure, couleur de trait)` pour tracer un polygone,,
- `text ((x, y), message, couleur de trait, police)` pour écrire un texte de coordonnées (x, y) au coin haut à gauche.

Certaines nécessitent une *bounding box* (cf. 1.2.5 page 13).

Quelques fonctions sont illustrées dans le programme suivant :

dessin.py

```
1  # -*- coding: utf-8 -*-
2
```

```
3  from PIL import Image
4  from PIL import ImageDraw
5
6  mon_image = Image.new("RGB", (400,200), "lightgrey")
7  draw = ImageDraw.Draw(mon_image)
8  largeur,hauteur=mon_image.size
9  draw.ellipse((largeur/4,hauteur/4) + (3*largeur/4,3*hauteur/4), fill=
                                          "blue")
10 draw.line((0, 0) + mon_image.size, width=8,fill="red")
11 offsetx=100
12 draw.rectangle(((80+offsetx,100),(130+offsetx,200)),fill="gray")
13 draw.ellipse(((largeur/3),(hauteur/3),(largeur*2/3),(hauteur*2/3)),
                                          fill="white")
14 del draw
15 mon_image.save("dessin.eps", "EPS")
16 mon_image.show()
```

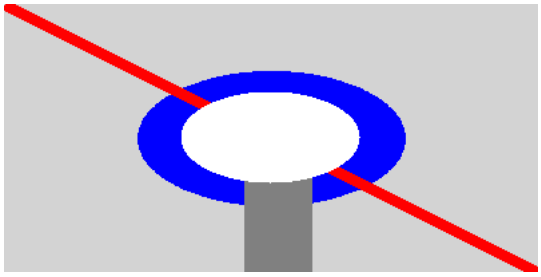


FIGURE 1.8 – dessin

1.2.17.3 Image multiple

pil_multi.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4  from PIL import ImageDraw
5
6  mon_image = Image.open("tux20.jpg")
7  nouvelle_image = Image.new("RGB", (530,350), "white")
8  draw = ImageDraw.Draw(mon_image)
9  largeur,hauteur=mon_image.size
10 nouvelle_image.paste(mon_image, (0,0,largeur,hauteur))
11 nouvelle_image.paste(mon_image, (300,0,300+largeur,hauteur))
12 nouvelle_image.paste(mon_image, (200,200,200+largeur,200+hauteur))
13 del draw
14 nouvelle_image.save("pil_multi.eps")
15 nouvelle_image.show()
```

Le résultat :

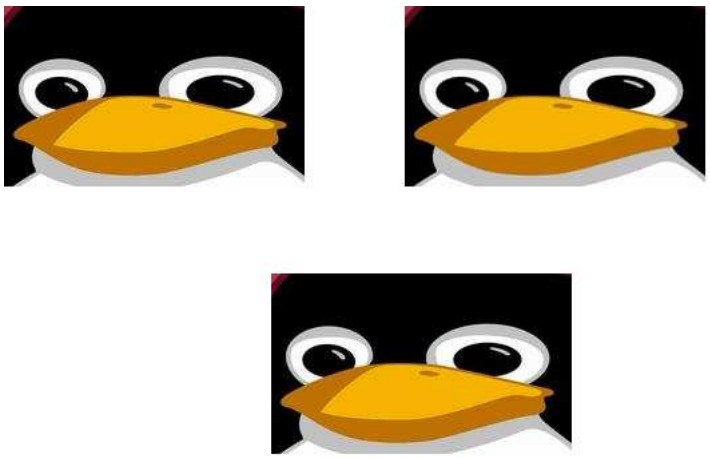


FIGURE 1.9 – pil_multi.eps

1.2.17.4 Méthodes d'un objet image

Certaines méthodes ont déjà été abordées, d'autres non. Voici quelques méthodes applicables sur un objet Image :

méthode	description
`save(fichier, format=None)`	Sauvegarde l'image dans un fichier. Si fichier est un nom de fichier, le format est choisi automatiquement
`convert(mode)`	Renvoie une nouvelle image dans un mode différent
`copy()`	Renvoie une copie de l'image
`crop(bbox)`	Renvoie une nouvelle image constituée par la *bounding box* : cf. 1.2.5 page 13 (spécifiée en argument) correspondante de l'image d'origine
`filter(nom)`	Renvoie une copie de l'image, filtrée avec le filtre d'amélioration du nom de *nom* : cf 1.2.16 page 18
`getbands()`	Renvoie une chaîne de caractères, une par bande, représentant le mode de l'image
`getbox()`	Renvoie la plus petite bounding box qui enferme la partie non nulle de l'image
`getdata()`	Renvoie le contenu d'une image dans une suite contenant les valeurs des pixels. La suite est "étalée" : les valeurs de la seconde ligne suivent immédiatement celles de la première, ...
`getextrema()`	Dans le cas des images simple-bande, renvoie un couple (`min`, *max*), où `min` est la plus petite valeur, et `max` la plus grande valeur, des pixels de l'image. Dans le cas des images multi-bandes, renvoie le tuple des couples (`min`, `max`) pour chaque bande
`getpixel((x,y))`	Retourne la (ou les) valeur(s) du pixel de coordonnées (x, y).
`histogram(masque=None)`	Pour les images simples-bandes, renvoie une liste de valeurs [c_0, c_1, ...], où c_i est le nombre de pixels ayant la valeur i. Pour les multi-bandes, les différentes listes de valeurs (pour chaque bande) sont concaténées (masque facultatif)
`offset(dx,dy)`	dy est facultatif
`paste(mon_image,bbox, mask = None)`	Remplace les pixels par ceux de *mon_image* (masque facultatif)
`point(fonction)`	Permet de modifier une image pixel par pixel, grâce à une `fonction` de transformation.

méthode	description
`point(tableau)`	Permet de transformer les pixels suivant un `tableau`. Le `tableau` doit être une suite de $256\,n$ valeurs, où n est le nombre de bandes de l'image. Chaque pixel de la bande b de l'image est remplacé par la valeur de `tableau[p+256*b]`, où p est l'ancienne valeur du pixel dans la bande considérée.
`putalpha(bande)`	Ajoute une bande alpha (transparence) à une image de mode "RGBA"
`putpixel((x,y), couleur)`	Remplace le pixel $(x,\ y)$ de l'image. Pour les images multi-bandes, la valeur est un tuple
`resize(taille,resample)`	Renvoie une nouvelle image ayant la taille spécifiée, en procédant linéairement (*resample* facultatif)
`rotate(`θ`)`	Renvoie une image qui a tournée d'un angle θ, en degrés, par rapport au centre de l'image considérée. Le sens de la rotation est trigonométrique
`show()`	Affiche l'image
`split()`	Renvoie un tuple contenant chaque bande de l'image originale, vue comme une image de mode "L". Par exemple, appliquer cette méthode à une image "RGB" produit un triplet d'images, la première pour la bande rouge, la seconde pour la verte et la dernière pour la bleue
`thumbnail(taille,filter=None)`	Remplace l'image originale par une nouvelle image ayant la taille *taille*. Le filtre optionnel fonctionne comme pour la méthode `resize()`
`transform(`x_s`, `y_s`, Image.EXTENT, (`x_0`, `y_0`, `x_1`, `y_1`))`	Renvoie une transformation de l'image : le point originellement en (x_0, y_0) se retrouvera en $(0, 0)$, et le point (x_1, y_1) en (x_s, y_s).
`transpose(méthode)`	Renvoie une copie, culbutée ou retournée, de l'image originale

TABLE 1.3 – Méthodes d'un objet image

Voici le programme *ouverture.py* pour vous repérer éventuellement :

ouverture.py

```
1  # -*- coding: utf-8 -*-
2
3  ## Import du module PIL
4  from PIL import Image
5
6  ## Ouverture de l'image
7  mon_image = Image.open("tux20.jpg")
8  ## sauvegarde de l'image en eps
9  mon_image.save("tux20.eps")
10 ## sauvegarde de l'image en png
11 mon_image.save("tux20.png")
12 ## Visualisation de l'image
13 mon_image.show()
14
15 ## Matrice image
16 donnees = list(mon_image.getdata())
17 print(donnees)
18
19 ## Taille de l'image
20 largeur,hauteur = mon_image.size
21 print(largeur,hauteur)
22
23 ## Propriétés
```

```
24 print(mon_image.format)
25 print(mon_image.mode)
26 print(mon_image.palette)
27 print(mon_image.info)
28
29 ma_nouvelle_image=Image.new('RGB',(largeur,hauteur))
30 ma_nouvelle_image.putdata(donnees)
31 ma_nouvelle_image.show()
32
33 ## Récupération des différentes composantes de l'image
34 rouge,vert,bleu = ma_nouvelle_image.split()
35 ## Sauvegarde des différentes composantes de l'image
36 rouge.save('red.jpg')
37 vert.save('green.jpg')
38 bleu.save('blue.jpg')
39 ## Visualisation de la composante rouge
40 Image.open('red.jpg').show()
41
42 ## décomposition de l'image
43 ma_composition=rouge,vert,bleu
44  ## Recomposition de l'image
45 mon_image_recomposee = Image.merge('RGB',ma_composition)
46 ## Sauvegarde de l'image décomposée puis recomposée
47 mon_image_recomposee.save("tux20c.jpg")
48 ## Visualisation de l'image recomposée
49 Image.open('tux20c.jpg').show()
50
51 ## Transformations
52
53 ## Conversion de format
54 rouge.save('red.eps')
55 vert.save('green.eps')
56 bleu.save('blue.eps')
57
58 ## Conversion de mode
59 image_gris = mon_image.convert('L')
60 image_gris.save('tux20_gris.png')
61 image_gris.save('tux20_gris.eps')
62
63 image_NetB = mon_image.convert('1')
64 image_NetB.save('tux20_NetB.bmp')
65
66 ## Miniature
67 retaillee=mon_image.resize((128,128),resample=Image.BILINEAR)
68 retaillee.show()
```

1.3 *Scipy* et les images

Le module *Scipy* aidé de *matplotlib* permet également d'effectuer des traitements sur les images.

Les manipulations sont un peu compliquées et assez inintéressantes du point de vue de la programmation. On n'y consacrera donc pas de temps.

Vous pourrez trouver des ressources ici :
http://perso.telecom-paristech.fr/~gramfort/liesse_python/3-Scipy.pdf

1.4 Manipulation des fichiers images

Dans cette partie, il s'agira, pour modifier une image, de parcourir sa matrice.

1.4.1 Introduction

Voici le programme *image-array.py* :

image-array.py

```
1  #-*- coding: utf-8 -*-
2
3  from PIL import Image
4  import numpy as np
5
6  image1 = Image.open('tux20_NetB.bmp')
7  largeur, hauteur = image1.size
8  print(largeur,hauteur,'\n')
9
10 donnees = list(image1.getdata())
11 print(donnees[0:30],'\n')
12
13 tableau = np.array(donnees)
14 print(tableau,'\n')
15 print(np.shape(tableau),'\n')
16
17 matrice = np.reshape(tableau,(hauteur,largeur))
18 # attention matrice = lignes * colonnes
19 # alors que donnees = largeur * hauteur
20 print(matrice,'\n')
21 print(np.shape(matrice),'\n')
22
23 matrice_etalee = list(matrice.flat)
24 print(matrice_etalee[0:30],'\n')
25
26 image2 = Image.new('L',(matrice.shape[1],matrice.shape[0]))
27 image2.putdata(matrice_etalee)
28 image2.save('tux20_reconstruit.eps')
29 image2.show()
```

ainsi que le résultat en console :

```
225 135

[0, 255, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

[  0 255   0 ..., 255 255 255]

(30375,)

[[  0 255   0 ...,   0   0   0]
 [  0   0 255 ...,   0   0   0]
 [255   0   0 ...,   0   0   0]
 ...,
 [  0   0   0 ..., 255 255 255]
 [  0   0   0 ..., 255 255 255]
 [  0   0   0 ..., 255 255 255]]

(135, 225)

[0, 255, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

⇒ **Activité 1.1**

Commentez chaque ligne du programme en fonction du résultat donné. Il est très important de bien comprendre cela pour la suite car même si on pourra éviter de passer par les matrices (type `array`), l'idée sera la même.

Par la suite, il sera parfois plus facile de parcourir les matrices images et de travailler pixel par pixel plutôt que d'utiliser la méthode `putdata()`.
On pourra utiliser une des deux méthodes suivantes :

creer-image.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  largeur,hauteur = 200,50
6  im1=Image.new('L',(largeur, hauteur))
7  pix1=im1.load()
8  for ligne in range(hauteur):
9      for colonne in range(largeur):
10         pix1[colonne,ligne]=100
11 im1.show()
12
13 im2=Image.new('L',(largeur, hauteur)) # en niveaux de gris
14 im3=Image.new('RGB',(largeur, hauteur)) # en couleurs
15 for ligne in range(hauteur):
16     for colonne in range(largeur):
17         im2.putpixel((colonne,ligne),150)
18         im3.putpixel((colonne,ligne),150)
19 im2.show()
20 im3.show()
```

1.4.2 Réflexions et rotations

Nous avons vu qu'il existait des fonctions prédéfinies dans *PIL* pour faire tourner des images. Nous allons ici le faire en agissant sur les matrices.

Appliquons un effet de miroir horizontal à l'image originale.

Rappelons que le coin supérieur gauche d'une image est repéré par ses coordonnées $(0, 0)$ alors que son coin inférieur droit l'est par $(\text{hauteur} - 1, \text{largeur} - 1)$.

Si l'on souhaite effectuer un effet de réflexion par un miroir horizontal, il faut envoyer le coin supérieur gauche dans le coin inférieur gauche, c'est-à-dire le pixel de coordonnées $(0, 0)$ en $(\text{hauteur} - 1, 0)$. Plus généralement, il faudra envoyer le pixel de coordonnées $(\text{ligne}, \text{colonne})$ en $(\text{hauteur} - 1 - \text{ligne}, \text{colonne})$ (ou l'inverse). Essayons avec le programme ci-dessous :

miroir_horiz.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  # extension de sauvegarde
6  ext='.eps'
7
8  # fonction miroir horizontal
9  def miroir1(image1):
10     # taille de l'image
11     largeur,hauteur = image1.size
12     # création d'une nouvelle image de même taille
13     image2 = Image.new('RGB',(largeur,hauteur))
14     # lecture des données de l'image originale
15     pixels1 = image1.load()
16     # lecture des données de l'image créée
17     pixels2 = image2.load()
18     # on parcourt les pixels (ligne et colonne)
19     for ligne in range(hauteur):
20         for colonne in range(largeur):
21             # on envoie les pixels de l'image1 dans l'image2
22             # en les changeant de place
23             pixels2[colonne,ligne] = pixels1[colonne,hauteur-1-ligne]
24     return image2
25
26 # ouverture de l'image
27 mon_image1 = Image.open("tux20.jpg")
28 # on applique la fonction la photo précédente
29 mirror = miroir1(mon_image1)
30 # on sauve l'image2 créée
31 mirror.save('tux20_mir1'+ext)
32 # on la visualise
33 mirror.show()
```

Le résultat :

(a) tux20 (b) tux20_mir1

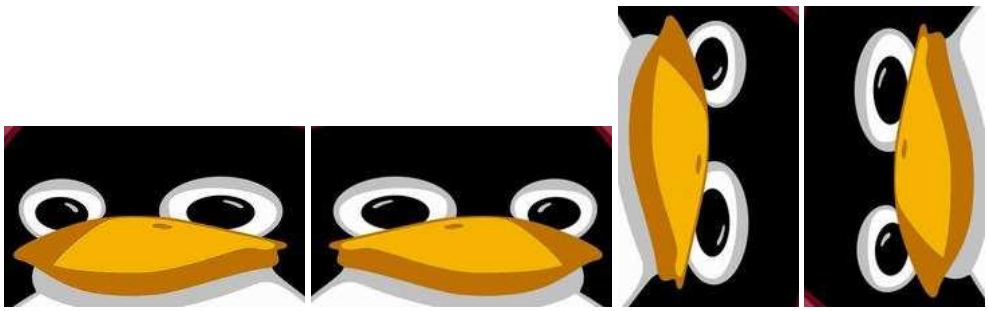
FIGURE 1.10 – Miroir horizontal

 Il est très important que la fonction "retourne" une image : en effet, si on souhaite faire subir plusieurs traitements à la suite à l'image de départ, cela ne pourra être possible si la fonction se termine par `show()` ou `save()`. Il est ici de même des fonctions vues auparavant qui devaient "retourner" un résultat et non une instruction `print()`.

⇒ **Activité 1.2**

Écrivez les 3 autres fonctions réalisant la réflexion par un miroir vertical ainsi que les rotations dans les sens trigonométrique et horaire.

Le résultat :



(a) tux20 (b) tux20_mir2 (c) tux20_rot1 (d) tux20_rot2

FIGURE 1.11 – Rotations

1.4.3 Conversion d'une image en niveaux de gris

Soit le programme suivant :

niveaugris1.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  # ouverture du fichier image source
6  mon_image = Image.open("tux20.jpg")
7
8  # largeur = largeur de l'image (nombre de colonnes)
9  # hauteur = hauteur de l'image (nombre de lignes)
10 largeur,hauteur = mon_image.size
11
12 # création de l'image destination
13 nouvelle_image = Image.new("RGB",(largeur,hauteur))
14
15 # on balaie toutes les lignes de l'image source, de 0 à hauteur-1
16 for ligne in range(hauteur):
17     # pour chaque ligne on balaie toutes les colonnes, de 0 à largeur
                                        -1
18     for colonne in range(largeur):
19         # on stocke le pixel (x,y) dans pix
20         pix = mon_image.getpixel((colonne,ligne))
21         # on calcule le niveau de gris (gris moyen ici)
22         gris = int((pix[0]+pix[1]+pix[2])/3)
23         # pix[0] est la composante rouge
24         # pix[1] la composante verte,
25         # pix[2] la composante bleue
26         # et int() pour avoir un entier compris entre 0 et 255
27
28         # on affecte à chaque couleur la valeur du gris moyen
29         rouge = gris
30         vert = gris
31         bleu = gris
32         # on écrit le pixel modifié sur l'image destination
33         nouvelle_image.putpixel((colonne,ligne),(rouge,vert,bleu))
34
35 # on stocke l'image résultante
36 nouvelle_image.save("tux20_gris_matrice.png")
37 nouvelle_image.save("tux20_gris_matrice.eps")
38
39 # on montre l'image
```

```
40  nouvelle_image.show()
41
42  mon_image = Image.open("tux20_gris_matrice.png")
43  data = list(mon_image.getdata())
44  print(data[-5:-1])
```

La méthode utilisée dans le programme précédent permet de conserver 3 composantes (RGB). On peut le voir en observant le résultat des dernières lignes de code :

```
[(255, 255, 255), (255, 255, 255), (252, 252, 252), (250, 250, 250)]
```

⇒ **Activité 1.3**

En vous aidant du programme précédent, écrivez-en un autre (*niveaugris2.py*) qui permet de créer réellement une image en niveaux de gris, c'est-à-dire avec une seule bande. Vous pourrez le vérifier en observant le résultat des dernières lignes de code :

```
[255, 255, 252, 250]
```

Le résultat :

(a) tux20_gris_matrice (b) tux20_gris_matrice_bis

FIGURE 1.12 – Niveaux de gris

1.4.4 Négatif en niveaux de gris

Partant de l'image "RGB", on peut la convertir en niveaux de gris puis l'inverser.

⇒ **Activité 1.4**

Écrivez, à l'aide d'une fonction *complement* que vous créerez, le programme *negatif.py* correspondant. Vous pourrez pour la suite, utiliser la méthode `convert('L')` afin d'être sûr d'avoir une image en niveaux de gris, ceci de manière plus rapide que par le programme précédent, par exemple, comme ceci :

```
image1 = Image.open('tux20.png').convert('L')
```

Le résultat :

(a) tux20 (b) niveaux de gris (c) tux20_inv

FIGURE 1.13 – Inversion des couleurs

1.4.5 Éclaircissement et assombrissement

1.4.5.1 Éclaircissement

Pour éclaircir une image en niveaux de gris, il faut augmenter la valeur des pixels (comprise entre 0 et 255). La plus grande valeur représente le blanc, et la plus petite le noir.

Une solution peut consister à ajouter un certain nombre à la valeur de chaque pixel : c'est ce qui est proposé dans le programme ci-dessous :

eclairciss1.py

```
1 # -*- coding: utf-8 -*-
2
3 from PIL import Image
```

```
4
5  # extension de sauvegarde
6  ext='.eps'
7  nom = 'billes'
8
9  def plus(x,k):
10     if x <= 255-k:
11         return (int(x+k))
12     else:
13         return 255
14
15 def eclair1(image1,k):
16     # taille de l'image
17     largeur,hauteur = image1.size
18     # création d'une nouvelle image de même taille
19     image2 = Image.new('L',(largeur,hauteur))
20     # lecture des données de l'image originale
21     pixels1 = image1.load()
22     # lecture des données de la nouvelle image
23     pixels2 = image2.load()
24     # on parcourt les pixels (ligne et colonne)
25     for ligne in range(hauteur):
26         for colonne in range(largeur):
27             pixels2[colonne,ligne] = plus(pixels1[colonne,ligne],k)
28     return image2
29
30 # ouverture de l'image
31 mon_image1 = Image.open("billes.jpg").convert('L')
32 # on applique la fonction la photo précédente
33 resultat = eclair1(mon_image1,60)
34 # on sauve l'image créée
35 resultat.save(nom+'_eclair1'+ext)
36 # on la visualise
37 resultat.show()
```

Une autre possibilité est de ramener les valeurs de pixels entre 0 et 1, et de leur appliquer une fonction croissante et concave entre 0 et 1 telle que $f(0) = 0$ et $f(1) = 1$. La fonction croissante et concave $f(x) = \sqrt{x}$ convient. Il faut ensuite ramener la valeur des pixels dans l'intervalle $[0, 255]$.

⇒ **Activité 1.5**

Complétez le programme *eclairciss2.py* suivant pour qu'il fasse l'opération précédemment décrite.

Le programme :

eclairciss2.py

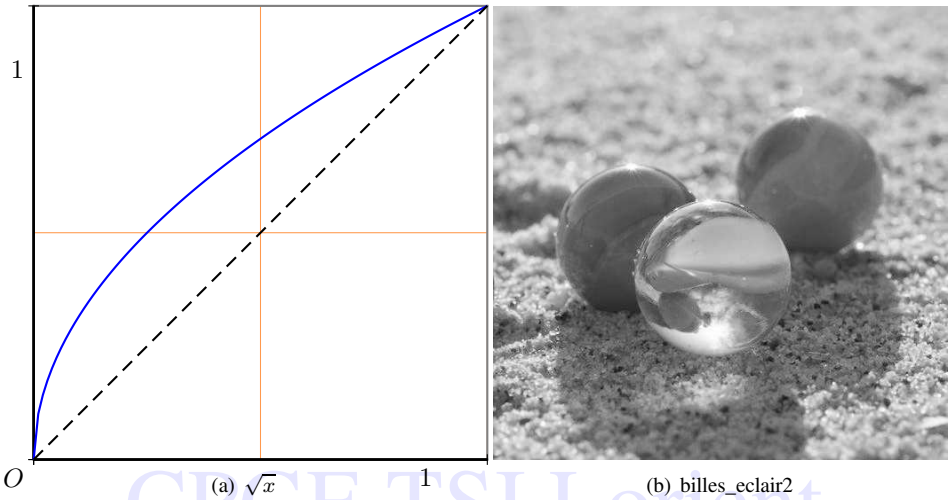
```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4  import numpy as np
5
6  # extension de sauvegarde
7  ext='.eps'
8  nom = 'billes'
9
10 def racine(x):
11
12
13
```

```
14  def eclair2(image1):
15      # taille de l'image
16      largeur,hauteur = image1.size
17      # création d'une nouvelle image de même taille
18      image2 = Image.new('L',(largeur,hauteur))
19      # lecture des données de l'image originale
20      pixels1 = image1.load()
21      # lecture des données de la nouvelle image
22      pixels2 = image2.load()
23      # on parcourt les pixels (ligne et colonne)
24      for ligne in range(hauteur):
25          for colonne in range(largeur):
26
27
28
29     return image2
30
31  # ouverture de l'image
32  mon_image1 = Image.open("billes.jpg").convert('L')
33  # on applique la fonction la photo précédente
34  resultat = eclair2(mon_image1)
35  # on sauve l'image créée
36  resultat.save(nom+'_eclair2'+ext)
37  # on la visualise
38  resultat.show()
```

Les résultats :

(a) billes_gris (b) billes_eclair1

FIGURE 1.14 – Éclaircissement 1



(a) $\sqrt{x}$ (b) billes_eclair2

FIGURE 1.15 – Éclaircissement 2

1.4.5.2 Assombrissement

Pour assombrir une image en niveaux de gris, il faut au contraire diminuer la valeur des pixels. Une fonction convient bien : c'est $f(x) = x^2$, qui est croissante et convexe.

⇒ **Activité 1.6**

Complétez les programmes *assombriss1.py* et *assombriss2.py* sur le même modèle que l'éclaircissement.

On obtient :

assombriss1.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  # extension de sauvegarde
6  ext='.eps'
7  nom = 'billes'
8
9  def moins(x,k):
10     if
11         return
12     else:
13         return
14
15 def assomb1(image1,k):
16     # taille de l'image
17     largeur,hauteur = image1.size
18     # création d'une nouvelle image de même taille
19     image2 = Image.new('L',(largeur,hauteur))
20     # lecture des données de l'image originale
21     pixels1 = image1.load()
22     # lecture des données de la nouvelle image
23     pixels2 = image2.load()
24     # on parcourt les pixels (ligne et colonne)
25     for ligne in range(hauteur):
26         for colonne in range(largeur):
27             pixels2[colonne,ligne] = moins(pixels1[colonne,ligne],k)
28     return image2
29
30 # ouverture de l'image
31 mon_image = Image.open("billes.jpg").convert('L')
32 # on applique la fonction la photo précédente
33 res = assomb1(mon_image,60)
34 # on sauve l'image créée
35 res.save(nom+'_assomb1'+ext)
36 # on la visualise
37 res.show()
```

et :

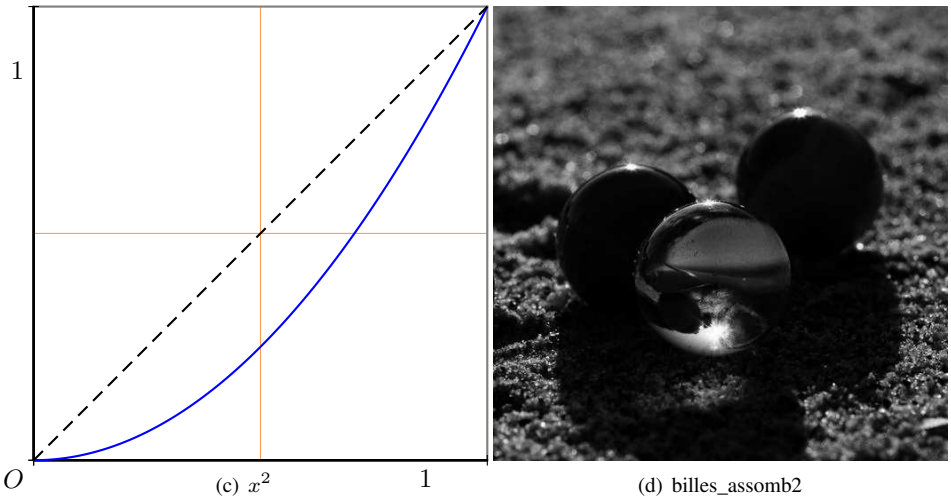
assombriss2.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
```

```
4  import numpy as np
5
6  # extension de sauvegarde
7  ext='.eps'
8  nom = 'billes'
9
10 def carre(x):
11
12
13 def assomb2(image1):
14     # taille de l'image
15     largeur,hauteur = image1.size
16     # création d'une nouvelle image de même taille
17     image2 = Image.new('L',(largeur,hauteur))
18     # lecture des données de l'image originale
19     pixels1 = image1.load()
20     # lecture des données de la nouvelle image
21     pixels2 = image2.load()
22     # on parcourt les pixels (ligne et colonne)
23     for ligne in range(hauteur):
24         for colonne in range(largeur):
25             # on envoie les pixels de l'image1 dans l'image2
26         # en leur appliquant la fonction carré
27         # définie par carre
28         pixels2[colonne,ligne] =
29     return image2
30
31 # ouverture de l'image
32 mon_image1 = Image.open("billes.jpg").convert('L')
33 # on applique la fonction la photo précédente
34 res = assomb2(mon_image1,)
35 # on sauve l'image créée
36 res.save(nom+'_assomb2'+ext)
37 # on la visualise
38 res.show()
```

Les résultats :

(a) billes_gris (b) billes_assomb1



(c) x^2 (d) billes_assomb2

FIGURE 1.16 – Assombrissement

1.4.6 Contraste d'une image

Principe

Pour contraster une image, on applique une fonction telle que :

- $f(0) = 0$
- $f(1) = 1$
- $f(0{,}5) = 0{,}5$

Ainsi, les noirs seront plus noirs et les blancs seront plus blancs.

Bien sûr, ceci est applicable sur des pixels de niveau de gris compris entre 0 et 1. Il faudra donc faire la conversion correspondante.

2 fonctions sont particulièrement adaptées :

- $f(x) = 3x^2 - 2x^3$
- $g(x) = x^3 \left(6x^2 - 15x + 10\right)$

⇒ **Activité 1.7**

Complétez le programme *contraste.py* pour qu'il réalise ces 2 opérations.

Le programme :

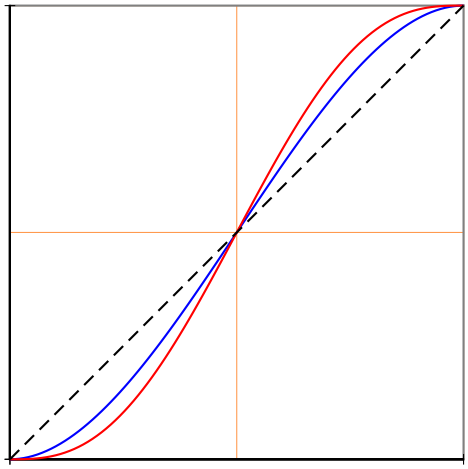
contraste.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4  import numpy as np
5
6  # extension de sauvegarde
7  ext1='.jpg'
8  ext2='.eps'
9  nom = 'billes'
10
11 def f(x):
12
13
14 def g(x):
15
16
17 def cont1(image1):
18     # taille de l'image
19     largeur,hauteur = image1.size
20     # création d'une nouvelle image de même taille
21     image2 = Image.new('L',(largeur,hauteur))
22     # lecture des données de l'image originale
23     pixels1 = image1.load()
24     # lecture des données des nouvelles images
25     pixels2 = image2.load()
26     # on parcourt les pixels (ligne et colonne)
27     for ligne in range(hauteur):
28         for colonne in range(largeur):
29             pixels2[colonne,ligne] =
30     return image2
31
32 # ouverture de l'image
```

```
33  mon_image1 = Image.open("billes.jpg").convert('L')
34  # on applique la fonction cont1 à la photo précédente
35  resultat = cont1(mon_image1)
36  # on sauve les images créées
37  resultat.save(nom+'_contr1'+ext1)
38  resultat.save(nom+'_contr1'+ext2)
39  # on visualise
40  Image.open('billes_contr1.jpg').show()
41
42  def cont2(image1):
43      # taille de l'image
44      largeur,hauteur = image1.size
45      # création d'une nouvelle image de même taille
46      image2 = Image.new('L',(largeur,hauteur))
47      # lecture des données de l'image originale
48      pixels1 = image1.load()
49      # lecture des données des nouvelles images
50      pixels2 = image2.load()
51      # on parcourt les pixels (ligne et colonne)
52      for ligne in range(hauteur):
53          for colonne in range(largeur):
54              pixels2[colonne,ligne] =
55      return image2
56
57  # ouverture de l'image
58  mon_image1 = Image.open("billes.jpg").convert('L')
59  # on applique la fonction cont2 à la photo précédente
60  resultat = cont2(mon_image1)
61  # on sauve les images créées
62  resultat.save(nom+'_contr2'+ext1)
63  resultat.save(nom+'_contr2'+ext2)
64  # on visualise
65  Image.open('billes_contr2.jpg').show()
```

Les résultats :

(a) billes_gris



(b) $f(x)$, $g(x)$

(c) billes_contr1 (d) billes_contr2

FIGURE 1.17 – Contraste

Remarque

On peut aussi composer les fonctions précédentes pour améliorer le contraste d'une image. Exemple avec le fichier *contraste_comp.py* suivant :

contraste_comp.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  # extension de sauvegarde
6  ext1='.png'
7  ext2='.eps'
8  nom = 'billes'
9
10 def f(x):
11
12
13 def g(x):
14
15
16 # composition des fonctions : f o g
17 def cont3(image1):
18     # taille de l'image
19     largeur,hauteur = image1.size
20     # création d'une nouvelle image de même taille
21     image2 = Image.new('L',(largeur,hauteur))
22     # lecture des données de l'image originale
23     pixels1 = image1.load()
24     # lecture des données des nouvelles images
25     pixels2 = image2.load()
26     # on parcourt les pixels (ligne et colonne)
27     for ligne in range(hauteur):
28         for colonne in range(largeur):
29             pixels2[colonne,ligne] =\
30                 int(f(g(pixels1[colonne,ligne]/255))*255)
31     return image2
32
33 # ouverture de l'image
34 mon_image1 = Image.open(nom+".jpg").convert('L')
35 # on applique la fonction cont3 à la photo précédente
36 resultat = cont3(mon_image1)
```

```
37  # on sauve les images créées
38  resultat.save(nom+'_contr3'+ext2)
39  resultat.save(nom+'_contr3'+ext1)
40  Image.open(nom+'_contr3.png').show()
41
42  # ou bien plus simplement :
43  # resultat_bis = cont1(cont2(mon_image1))
```

1.4.7 Transparence d'une image

1.4.8 La transparence

Les formats *.gif* et *.png* offrent la possibilité de rendre le fond de l'image transparent. Ceci permet de poser l'image sur des fonds de couleur, sans que l'on voit par exemple le blanc de l'image qui constitue son propre fond.

Le mode "RGBA" d'une image permet de gérer cette transparence. C'est le dernier canal (alpha) qui traduit celle-ci : 255 pour opaque, 0 pour transparent.

⇒ **Activité 1.8**

Complétez le programme *transpar.py* suivant afin que la transparence passe de 255 à 100 :

Le programme :

transpar.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  mon_image = Image.open("billes.jpg")
6  largeur, hauteur = mon_image.size
7
8  # conversion de l'image RGB en RGBA (A pour alpha, la transparence)
9  mon_image_bis = mon_image.convert("RGBA")
10 nouvelle_image = Image.new("RGBA",(largeur,hauteur))
11
12 for ligne in range(hauteur):
13     for colonne in range(largeur):
14         pixel = mon_image_bis.getpixel((colonne,ligne))
15         rouge =
16         vert =
17         bleu =
18         alpha =
19         nouvelle_image.putpixel(                                    )
20
21 nouvelle_image.save("billes_transp.png","PNG")
22 nouvelle_image.show()
```

1.4.9 Seuillage d'une image

Segmentation

La segmentation consiste à découper une image en régions distinctes. Cette segmentation peut se réaliser par deux méthodes :

- par contours
- par homogénéité, par exemple en séparant les zones de même couleur

Le seuillage d'image est la méthode la plus simple de segmentation d'image. À partir d'une image en niveau de gris, le seuillage d'image peut être utilisé pour créer une image comportant uniquement deux valeurs, noir ou blanc : l'image est alors en noir et blanc (monochrome).

Principe

Le seuillage d'image remplace un à un les pixels d'une image à l'aide d'une valeur seuil fixée (par exemple 122). Ainsi, si un pixel à une valeur supérieure au seuil (par exemple 150), il prendra la valeur 255 (blanc), et si sa valeur est inférieure (par exemple 100), il prendra la valeur 0 (noir).

1.4.9.1 Seuillage à 1 seuil d'une image en niveaux de gris

Le programme *seuillage.py* réalise un seuillage à un seuil, donc une binarisation :

seuillage.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  def seuil(x):
6      if x<122:
7          return 0
8      else:
9          return 255
10
11 # seuillage d'une image en niveaux de gris
12 mon_image = Image.open('tux20.jpg').convert('L')
13 pixel = mon_image.load()
14 for colonne in range(mon_image.size[0]):
15     for ligne in range(mon_image.size[1]):
16         pixel[colonne,ligne] = seuil(pixel[colonne,ligne])
17
18 mon_image.save('tux20_seuil.eps')
19 mon_image.show()
```

Le résultat :

FIGURE 1.18 – tux20_seuil

1.4.9.2 Seuillage à 2 seuils d'une image en niveaux de gris

On peut réaliser l'opération avec 2 seuils : par exemple, tous les pixels de valeur inférieure à 100 sont mis à 0 et tous ceux de valeur supérieure à 150 sont mis à 255. On obtient alors du blanc, du noir et une teinte de gris.

⇒ **Activité 1.9**

Modifiez le programme précédent et l'enregistrer sous le nom de *seuillage_2seuils.py* de façon à ce qu'il permette de réaliser cette opération.

Le programme :

Le résultat :

FIGURE 1.19 – tux20_seuil2

1.4.9.3 Image en couleurs

On peut étendre le principe avec une image en couleurs. On peut regarder le programme *seuillageRGB.py* :

seuillageRGB.py

```
1 # -*- coding: utf-8 -*-
2
3 from PIL import Image
```

```
4
5  def seuil(x):
6      if x<180:
7          return 0
8      else:
9          return 255
10
11 fichier='tux20'
12 # Essai avec billes :
13 # fichier='billes'
14
15 ext='.jpg'
16 fich_ext=fichier+ext
17
18 # Seuillage d'une image en couleurs
19 mon_image = Image.open(fich_ext)
20 composition = mon_image.split()
21 largeur,hauteur = mon_image.size
22
23 rouge = list(composition[0].getdata())
24 vert = list(composition[1].getdata())
25 bleu = list(composition[2].getdata())
26
27 # On recrée l'image rouge
28 for i in range(len(rouge)):
29     rouge[i] = seuil(rouge[i])
30 nouveau_rouge = Image.new("L",(largeur,hauteur))
31 nouveau_rouge.putdata(rouge)
32
33 # On recrée l'image verte
34 for i in range(len(vert)):
35     vert[i] = seuil(vert[i])
36 nouveau_vert = Image.new("L",(largeur,hauteur))
37 nouveau_vert.putdata(vert)
38
39 # On recrée l'image bleue
40 for i in range(len(bleu)):
41     bleu[i] = seuil(bleu[i])
42 nouveau_bleu = Image.new("L",(largeur,hauteur))
43 nouveau_bleu.putdata(bleu)
44
45 # Composition de l'image
46 ma_composition=nouveau_rouge,nouveau_vert,nouveau_bleu
47
48 # Recomposition de l'image
49 mon_image_recomposee = Image.merge('RGB',ma_composition)
50
51 # Sauvegarde de l'image décomposée puis recomposée
52 mon_image_recomposee.save(fichier+"_seuilRGB.eps")
53 mon_image_recomposee.save(fichier+"_seuilRGB.png")
54
55 # Visualisation de l'image recomposée
56 mon_image_recomposee.show()
```

Le résultat :

FIGURE 1.20 – tux20_seuilRGB

Remarque

Si vous voulez tester l'image *billes.jpg*, il vous suffit de commenter la ligne 12 et de décommenter la ligne 14.

1.4.10 Filtrage

Nous nous limiterons dans cette partie à filtrer des images en niveaux de gris mais ceci est réalisable bien sûr sur chacune des composantes d'une image en couleurs.

1.4.10.1 Filtrage du contour

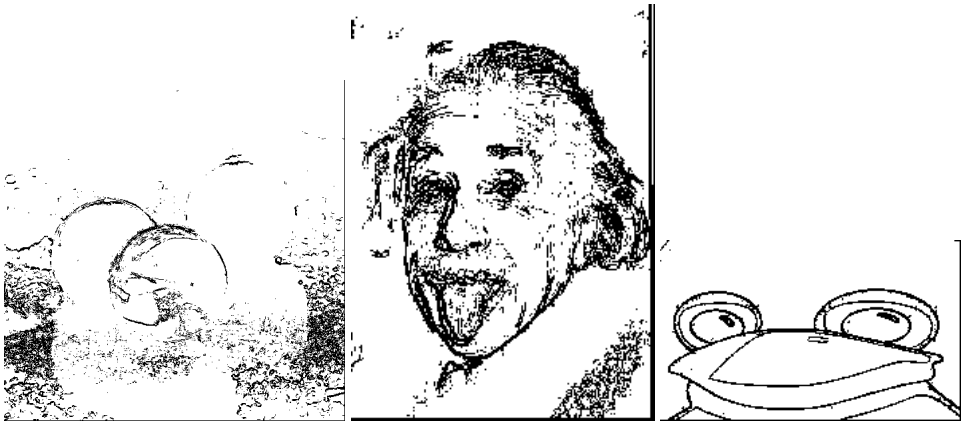
Filtrer le contour d'une image se fait à partir de l'image en niveaux de gris : il vaut mieux donc la convertir tout d'abord. Pour filtrer les contours, on peut noircir les pixels dont la différence de valeurs avec leurs voisins excède un certain seuil (souvent inférieur à 25). C'est ce que réalise le programme *contour1.py* :

contour1.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  ext= '.eps'
6
7  mon_image1 = Image.open("billes.jpg")
8  mon_image2 = Image.open('einstein1.png')
9  mon_image3 = Image.open('tux20_gris.png')
10
11 def filtre_contour(image1,seuil):
12     largeur,hauteur = image1.size
13     image2 = Image.new('L',(largeur,hauteur))
14     image1 = image1.convert('L')
15     pixels1 = image1.load()
16     pixels2 = image2.load()
17     for ligne in range(hauteur-1):
18         for colonne in range(largeur-1):
19             if abs(pixels1[colonne,ligne]-pixels1[colonne,ligne+1])\
20                   > seuil or\
21                   abs(pixels1[colonne,ligne]-pixels1[colonne+1,ligne]
                                                  ) > seuil:
22                 pixels2[colonne,ligne] = 0
23             else:
24                 pixels2[colonne,ligne] = 255
25     return image2
26
27 res1 = filtre_contour(mon_image1,20)
28 res1.save('billes_cont'+ext)
29 res1.show()
30 res2 = filtre_contour(mon_image2,20)
```

```
31  res2.save('einstein1_cont'+ext)
32  res2.show()
33  res3 = filtre_contour(mon_image3,20)
34  res3.save('tux20_gris_cont'+ext)
35  res3.show()
```

Le résultat :



(a) billes_cont (b) einstein11_cont (c) tux20_contour

FIGURE 1.21 – Filtrage du contour1

Le filtrage peut se faire à l'aide d'un filtre à matrice de convolution. Le principe est assez simple : on "fait passer" un noyau (en pratique, une matrice 2 x 2, 3 x 3, ...) sur la matrice de l'image à traiter. Exemple de masque 3 x 3 :

a	b	c
d	e	f
g	h	i

Soit la matrice image ci-contre. Supposons qu'elle soit en niveaux de gris.

53	47	72	98	118	136	165
58	37	39	84	130	130	167
63	34	28	80	101	139	180
69	23	29	73	112	140	174
64	21	20	67	127	145	160
61	18	12	79	104	113	121
12	15	19	66	102	108	119

Principe

L'application du filtre va consister à remplacer la case centrale colorée en rouge de valeur 80 par la valeur s_t donnée par la somme des 9 produits :
$s_t = 39\,a + 84\,b + 130\,c + 28\,d + 80\,e + 101\,f + 29\,g + 73\,h + 112\,i$

Le problèmes suivants vont être à considérer :

1. Sur un pixel appartenant à la bordure de l'image, il manque des voisins,
2. Le résultat peut sortir de l'intervalle $[0, 255]$ (c'est même probable !)

Les solutions possibles sont les suivantes :

1. (a) On ignore les pixels des bords et on parcourt donc la matrice image à traiter de la deuxième ligne à l'avant-dernière ligne et de la deuxième colonne à l'avant-dernière colonne.
   (b) On crée une bordure (par exemple noire ou blanche), par exemple de 1 pixel de chaque côté et on reprend la méthode précédente.
2. (a) Chaque valeur de pixel est divisée par la somme s_c des cœfficients de la matrice *noyau* : on parle alors de normalisation. Si cette somme s_c est nulle, on divise par 1. Dans ce dernier cas, on ajoute 128 pour éviter les valeurs négatives.
   (b) Chaque valeur de pixel est écrêtée par 0 ou 255.
   (c) L'ensemble de la matrice résultat est ramenée dans l'intervalle $[0, 255]$ à l'aide d'une règle de 3.

On peut ainsi écrire les programmes *contour2.py* :

contour2.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  ext1='.jpg'
6  ext2='.eps'
7
8  # détecteurs de contours
9  noyau1 =\
10 [-1,-1,-1,\
11 -1 , 8,-1,\
12 -1 ,-1,-1]
13
14 def filtre_contour1(image1,noyau):
15     largeur,hauteur = image1.size
16
17     # conversion en niveaux de gris, mode RGB
18     image1 = image1.convert("L")
19     image2 = Image.new("L",(largeur,hauteur))
20     # filtrage de l'image en niveaux de gris
21     for y in range(1,hauteur-1):
22         for x in range(1,largeur-1):
23             liste_pix=[]
24             liste_pix.append(image1.getpixel((x-1,y-1)))
25             liste_pix.append(image1.getpixel((x,y-1)))
26             liste_pix.append(image1.getpixel((x+1,y-1)))
27             liste_pix.append(image1.getpixel((x-1,y)))
28             liste_pix.append(image1.getpixel((x,y)))
29             liste_pix.append(image1.getpixel((x+1,y)))
30             liste_pix.append(image1.getpixel((x-1,y+1)))
31             liste_pix.append(image1.getpixel((x,y+1)))
32             liste_pix.append(image1.getpixel((x+1,y+1)))
33             pixel=0
34             for i in range(9):
35                 pixel=int(pixel+noyau[i]*liste_pix[i])
36
37             pixel = pixel + 128
38             image2.putpixel((x,y),pixel)
39     return image2
40
41 nom_image = "billes"
42 mon_image = Image.open(nom_image+ext1)
43 filtre1 = filtre_contour1(mon_image,noyau1)
44 filtre1.save(nom_image+'_contours11'+ext2)
45 filtre1.show()
```

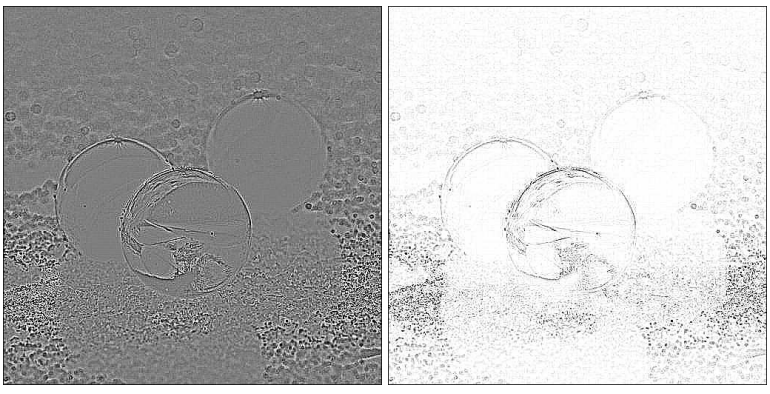
et *filtre_contour3* :

filtre_contour3

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  ext1='.jpg'
6  ext2='.eps'
7
8  # détecteurs de contours
```

```
noyau1 =\
[-1,-1,-1,\
-1 , 8,-1,\
-1 ,-1,-1]

def filtre_contour2(image1,noyau):
    largeur,hauteur = image1.size

    # conversion en niveaux de gris, mode RGB
    image1 = image1.convert("L")
    image2 = Image.new("L",(largeur,hauteur))
    # filtrage de l'image en niveaux de gris
    for y in range(1,hauteur-1):
        for x in range(1,largeur-1):
            liste_pix=[]
            liste_pix.append(image1.getpixel((x-1,y-1)))
            liste_pix.append(image1.getpixel((x,y-1)))
            liste_pix.append(image1.getpixel((x+1,y-1)))
            liste_pix.append(image1.getpixel((x-1,y)))
            liste_pix.append(image1.getpixel((x,y)))
            liste_pix.append(image1.getpixel((x+1,y)))
            liste_pix.append(image1.getpixel((x-1,y+1)))
            liste_pix.append(image1.getpixel((x,y+1)))
            liste_pix.append(image1.getpixel((x+1,y+1)))
            pixel=0
            for i in range(9):
                pixel=int(pixel+noyau[i]*liste_pix[i])

            if pixel < 0:
                pixel =0
            if pixel > 255:
                pixel = 255
            pixel = pixel + 128
            image2.putpixel((x,y),pixel)
    return image2

nom_image = "billes"
mon_image = Image.open(nom_image+ext1)
filtre1 = filtre_contour2(mon_image,noyau1)
filtre1.save(nom_image+'_contours21'+ext2)
filtre1.show()
```

qui donnent :



(a) billes_contours11 (b) billes_contours21

FIGURE 1.22 – Filtrage du contour

1.4.10.2 Embossage

En utilisant la méthode précédente, ce filtrage sert à mettre en relief l'image. Le noyau est une matrice 3 x 3 et le décalage est de 128. Voici le noyau utilisé :

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

Le programme :

embossage.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  ext1='.jpg'
6  ext2='.eps'
7
8  noyau0 =\
9  [-2 , -1, 0,\
10 -1  , 0, 1,\
11 0 , 1, 2]
12
13 def embossage(image1,noyau):
14     image2 = Image.new("RGB",image1.size)
15     image3 = Image.new("RGB",image1.size)
16
17     # conversion en niveaux de gris, mode RGB
18     for ligne in range(image1.size[1]):
19          for colonne in range(image1.size[0]):
20              pix = image1.getpixel((colonne,ligne))
21              rouge = int((pix[0]+pix[1]+pix[2])/3)
22              vert = rouge
23              bleu = rouge
24              image2.putpixel((colonne,ligne),(rouge,vert,bleu))
25
26     # filtrage de l'image en niveaux de gris
27     for ligne in range(1,image1.size[1]-1):
28         for colonne in range(1,image1.size[0]-1):
29             liste_pix=[]
30             liste_pix.append(image2.getpixel((colonne-1,ligne-1)))
31             liste_pix.append(image2.getpixel((colonne,ligne-1)))
32             liste_pix.append(image2.getpixel((colonne+1,ligne-1)))
33             liste_pix.append(image2.getpixel((colonne-1,ligne)))
34             liste_pix.append(image2.getpixel((colonne,ligne)))
35             liste_pix.append(image2.getpixel((colonne+1,ligne)))
36             liste_pix.append(image2.getpixel((colonne-1,ligne+1)))
37             liste_pix.append(image2.getpixel((colonne,ligne+1)))
38             liste_pix.append(image2.getpixel((colonne+1,ligne+1)))
39             rouge=0
40             for i in range(9):
41                 rouge=int(rouge+noyau[i]*liste_pix[i][0])
42
43             rouge = rouge+128
44             vert = rouge
45             bleu = rouge
46             image3.putpixel((colonne,ligne),(rouge,vert,bleu))
```

```
47
48      return image3
49
50  nom_image = "billes"
51  mon_image = Image.open(nom_image+ext1)
52  emboss = embossage(mon_image,noyau0)
53  emboss.save(nom_image+'_relief'+ext2)
54  emboss.save(nom_image+'_relief'+ext1)
55  emboss.show()
```

Le résultat :

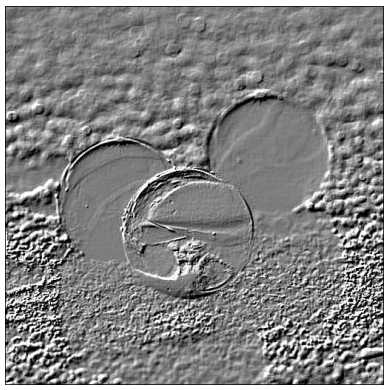


FIGURE 1.23 – billes_relief

On peut par exemple s'amuser à embosser l'image `quimper-cathedrale.jpg`.

1.4.10.3 Flou gaussien

Ce filtrage sert à atténuer les changements brusques d'intensité. On utilise la méthode précédente. La matrice est une matrice 5 x 5 et la normalisation est égale à 273. Voici le noyau utilisé :

$$\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Le programme :

flou-gaussien.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  ext1='.jpg'
6  ext2='.eps'
7
8  noyau0 =\
9  [1,  4,   7,   4,  1,\
10 4,  16,  26,  16,  4,\
11 7,  26,  41,  26,  7,\
12 4,  16,  26,  16,  4,\
```

```
13  1,   4,   7,   4,  1 ]
14
15  def flou_gaussien(image1,noyau):
16      image2 = Image.new("RGB",image1.size)
17      image3 = Image.new("RGB",image1.size)
18
19      # conversion en niveaux de gris, mode RGB
20      for ligne in range(image1.size[1]):
21           for colonne in range(image1.size[0]):
22               pix = image1.getpixel((colonne,ligne))
23               rouge = int((pix[0]+pix[1]+pix[2])/3)
24               vert = rouge
25               bleu = rouge
26               image2.putpixel((colonne,ligne),(rouge,vert,bleu))
27
28      # filtrage de l'image en niveaux de gris
29      for ligne in range(2,image1.size[1]-2):
30          for colonne in range(2,image1.size[0]-2):
31              liste_pix=[]
32              for x in range(colonne-2,colonne+3):
33                  for y in range(ligne-2,ligne+3):
34                      liste_pix.append(image2.getpixel((x,y)))
35              rouge=0
36              for i in range(25):
37                  rouge=rouge+noyau[i]*liste_pix[i][0]
38
39              rouge = int(rouge/273)
40              vert = int(rouge)
41              bleu = int(rouge)
42              image3.putpixel((colonne,ligne),(rouge,vert,bleu))
43
44      return image3
45
46  nom_image = "billes"
47  mon_image = Image.open(nom_image+ext1)
48  emboss = flou_gaussien(mon_image,noyau0)
49  emboss.save(nom_image+'_gauss'+ext2)
50  emboss.save(nom_image+'_gauss'+ext1)
51  emboss.show()
```

Le résultat :

FIGURE 1.24 – billes_gauss

1.4.10.4 Autres filtres

Il existe une multitude de filtres (passe-bas, passe-haut, réhausseur de contraste, ...). Si vous souhaitez utiliser des matrices de convolutions, en voici quelques-unes :

- $\begin{bmatrix} -\frac{1}{6} & -\frac{2}{3} & -\frac{1}{6} \\ -\frac{2}{3} & \frac{13}{3} & -\frac{2}{3} \\ -\frac{1}{6} & -\frac{2}{3} & -\frac{1}{6} \end{bmatrix}$
- $\begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$
- $\begin{bmatrix} 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 \\ 4 & 4 & 0 & 2 & 2 \\ 4 & 4 & 3 & 3 & 3 \\ 4 & 4 & 3 & 3 & 3 \end{bmatrix}$
- $\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$
- filtre binomial (normalisation à 256)

  $\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$
- filtre pyramidal (normalisation à 81)

  $\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$
- filtre pyramidal (normalisation à 25)

  $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 2 & 2 & 0 \\ 1 & 2 & 5 & 2 & 1 \\ 0 & 2 & 2 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$
- $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
- $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
- $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$
- $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$
- $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$
- $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & 1 \end{bmatrix}$
- $\begin{bmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix}$
- $\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
- ...

Il est également possible de combiner ces différents traitements.

1.4.11 Histogrammes

Un premier programme que vous devez absolument comprendre :

histo10-ss-csv.py

```
1  #-*- coding: utf-8 -*-
2
3  from PIL import Image
4  import matplotlib.pyplot as plt
5
6  mon_image = Image.open("billes.jpg")
7  hist = [] # ou hist = list()
8
9  largeur,hauteur = mon_image.size
10
11 for i in range(256):
12     hist.append(0)
13 for ligne in range(hauteur):
14     for colonne in range(largeur):
15         p = mon_image.getpixel((colonne,ligne))
16         n_gris = int((p[0]+p[1]+p[2])/3)
17         hist[n_gris] = hist[n_gris]+1
18
19 n_abscisses = list(range(256))
20 plt.xlim(0,255)
21 plt.plot(n_abscisses,hist)
22 plt.grid(True)
23 plt.show()
```

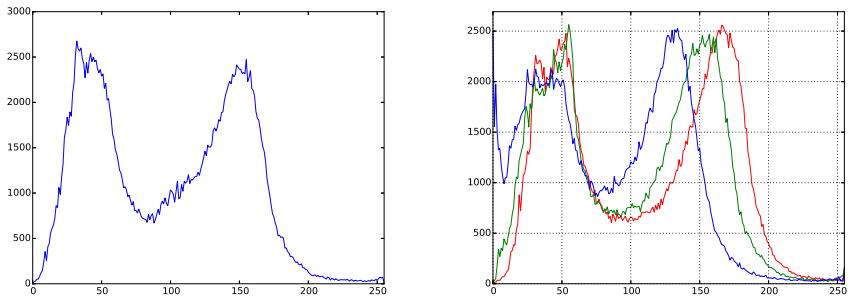
Le même avec écriture dans un fichier `.csv` :

histo10.py

```
1  #-*- coding: utf-8 -*-
2
3  from PIL import Image
4  import csv
5  import numpy as np
6  import matplotlib.pyplot as plt
7
8  mon_image = Image.open("billes.jpg")
9  hist = [] # ou hist = list()
10
11 with open("billes_histo_niv_gris.csv","w") as sortie:
12     largeur,hauteur = mon_image.size
13     mon_fichier = csv.writer(sortie,delimiter=';')
14     mon_fichier.writerow(["niveau_gris","nb_pixels"]) # titres
15
16     for i in range(256):
17         hist.append(0)
18     for ligne in range(hauteur):
19         for colonne in range(largeur):
20             p = mon_image.getpixel((colonne,ligne))
21             n_gris = int((p[0]+p[1]+p[2])/3)
22             hist[n_gris] = hist[n_gris]+1
23
24     for i in range(256):
25         mon_fichier.writerow([i,hist[i]])
26
27 with open("billes_histo_niv_gris.csv", "r") as fich:
```

```
28      fichier = csv.reader(fich,delimiter=';')
29      abscisses = []
30      ordonnees = []
31      for ligne in fichier:
32          if ligne==[]:
33              pass
34          else:
35              try:
36                  abscisses.append(float("".join(ligne[0].split(":"))))
37                  ordonnees.append(float("".join(ligne[1].split(":"))))
38              except ValueError: # à cause des titres
39                  pass
40
41  plt.xlim(0,255)
42  plt.plot(abscisses,ordonnees)
43  plt.savefig("histo10.eps")
44  plt.grid(True)
45  plt.show()
```

Les résultats :



(a) histogramme personnel (b) histogramme couleurs + image

FIGURE 1.25 – Histogrammes

⇒ **Activité 1. 10**

Comme vous pouvez le voir ; l'histogramme de droite porte sur les 3 composantes : à vous de le faire dans le programme *histo10-ss-csv-coul.py*.

Le programme :

1.5 Mini-projets sur les images

1.5.1 Conversion d'images

L'objectif de ce projet est d'écrire un programme de conversion d'image.

Celui-ci devra proposer un menu à l'utilisateur. Ce dernier aura le choix dans un premier temps entre :

- binarisation (conversion en noir et blanc),
- conversion en niveaux de gris,
- conversion en négatif de l'image en niveaux de gris,
- conversion en sépia.
- conversion en négatif de l'image en couleurs.

Cette liste n'est pas exhaustive et pourra bien sûr être complétée avec d'autres transformations (vues dans le cours, dans les activités, les exercices, ...).

Pour réaliser le script, on sera limité dans l'utilisation du module *PIL*. On ne pourra utiliser dans celui-ci que les instructions suivantes :

```
mon_image = Image.open(nom_fichier)
mon_image.size
image.getpixel((x,y))
image.putpixel((x,y),(r,v,b))
```


Il faudra donc parcourir les matrices des images.

Le menu pourra s'inspirer du programme *menu_vide.py* :

menu_vide.py

```
# -*- coding: utf-8 -*-

from PIL import Image

def.........

if __name__ == "__main__":
    nb_transfos = ...........
    choix=-1
    while choix !=0:
        print ("Quelle opération désirez-vous faire ?")
        print ("0 - Pour sortir")
        print ("1 ................")
        ............................
        ............................
        while True:
            try:
                choix = int(input())
            except ValueError:
                print("Vous n'avez pas entré un nombre entre 0 et %s !"\
                %nb_transfos)
                continue
            if choix>=0 and choix<=nb_transfos:
                break
        if choix == 0:
            break
        while True:
            try:
                nom_fichier = input("Entrez le nom de votre fichier image : ")
                mon_image = Image.open(nom_fichier)
                largeur,hauteur = mon_image.size
            except Exception:
                print("Le fichier", nom_fichier,"est introuvable")
                continue
            break
        if choix == 1:
            .............
        mon_image.show()
```

Remarque

- Pour convertir une image en sepia, **il faut d'abord passer en niveaux de gris**, mais en mode "RGB". Ensuite, on applique les règles suivantes :
  - Si le rouge est inférieur ou égal à 62, on multiplie le rouge par 1,1 et le bleu par 0,9,
  - Si le rouge est compris entre 63 et 191, on multiplie le rouge par 1,15 et le bleu par 0,85,
  - Si le rouge est supérieur à 192, on le remplace par le minimum entre 255 et le rouge multiplié par 1,08. Quant au bleu, on le multiplie par 0,93. La fonction `min()` existe sous *Python*.
- Comme déjà vu, créer le négatif d'une image consiste à inverser toutes les couleurs : le 0 devient 255 et vice-versa.

⇒ **Activité 1.11**

- Écrivez la fonction permettant de réaliser la conversion en sepia.
- Complétez le menu fourni pour que le programme fasse les conversions demandées.

Vous enregistrerez le programme sous le nom *menu_total.py*.

Le programme :

1.5.2 Stéganographie

La stéganographie consiste à cacher un message ou une image dans une autre image.

1.5.2.1 Message caché dans une image

On va ici cacher une chaîne de caractères dans la photo *einstein1.png*.

La chaîne à crypter est par exemple : "Les sciences, c'est de la balle !" Si vous voulez changer, c'est possible mais plus il est long, plus l'image devra être grande.

Pour cela on va créer une image avec Python qui contiendra le message. Elle s'appellera *einstein11.png*.

Principe

- On va travailler sur la composante rouge des images.
  On rappelle que chaque pixel est défini par un triplet de trois nombres entiers compris entre 0 et 255, le premier donnant la composante rouge, le deuxième la verte et le troisième la bleue.
- Dans le tableau ci-dessous, on donne à la deuxième ligne la composante rouge des 16 premiers pixels.
  Dans une première étape, à la ligne 3 du tableau, les valeurs sont réduites au plus grand nombre pair inférieur ou égal à la valeur.
- On va ajouter ensuite à ces nombres pairs des 0 et des 1 et ce sont ces 0 et ces 1, regroupés par 8, qui vont transmettre les informations cachées.
- La chaîne contient 33 caractères. Dans les huit premiers pixels, on entre l'information 00100001 qui est la représentation binaire de 33.
  On va donc coder l'image sur $8 + 8 * 33 = 272$ pixels.
- Dans les huit pixels suivants, on rentre l'information 01001100 qui est la représentation binaire de 76 et qui est le code *ascii* de L (l majuscule).
  On a donc récupéré le premier caractère de la chaîne.
- La quatrième ligne contient donc la longueur de la chaîne et le premier caractère

pixel	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
einstein1	61	59	56	53	52	54	56	57	62	62	63	63	64	65	65	66
réduction	60	58	56	52	52	54	56	56	62	62	62	62	64	64	64	66
chaîne	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	0
einstein11	60	58	57	52	52	54	56	57	62	63	62	62	65	65	64	66

TABLE 1.4 – message caché

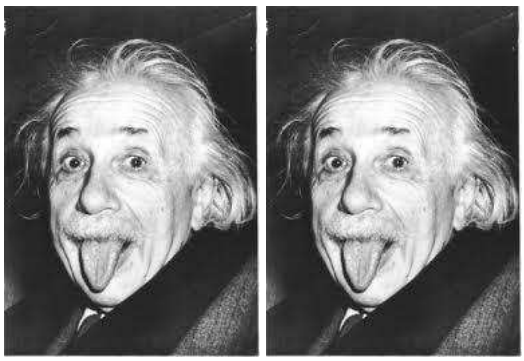
Le programme *message.py* réalise cette série d'opérations (la fonction `lireTexte(nomFichier)` permet de lire un texte enregistré dans fichier *nom.Fichier.txt* s'il est dans le même dossier) :

message.py

```
1  # -*- coding: utf-8 -*-
2
3  def lireTexte(nomFichier) :
4      try:
5          fichier = open(nomFichier, 'r',encoding="utf-8")
6      except IOError:
7          print (nomFichier + " : ce fichier n'existe pas")
8          return None
9      texte = ''
10     while True:
11         ligne = fichier.readline()
12         if ligne == '':
13             break
```

```
14          for mot in ligne:
15              texte = texte + mot
16      fichier.close()
17      return texte
18
19  from PIL import Image
20
21  mon_image = Image.open("einstein1.png")
22  mon_image.show()
23  # pour le prof, sauve au format .eps
24  mon_image.save("einstein1.eps")
25  # on éclate l'image en trois (rouge vert bleu)
26  red,green,blue=mon_image.split()
27  # on transforme le rouge de l'image en liste
28  rouge=list(red.getdata())
29
30  # message à insérer
31  mon_message = lireTexte('message.txt')
32
33  # on note la longueur de la chaine et on la transforme en binaire
34  # on enlève le "0b" devant avec [2:] et on ajuste à 8 bits
35  longueur=len(mon_message)
36  longueur_bin=bin(len(mon_message))[2:].rjust(8,"0")
37
38  # la longueur de la chaîne vaut ici 33 qui en base 2 donne 100001
39
40  # on transforme la chaine en une liste de 0 et de 1
41  long=[bin(ord(caractere))[2:].rjust(8,"0") for caractere in
                                          mon_message]
42
43  # transformation de la liste en chaine
44  chaine=''.join(long)
45
46  # on code la longueur de la liste dans les 8 premiers pixels rouges
47  # après pairisation
48  for j in range(8):
49      rouge[j]=2*(rouge[j]//2)+int(longueur_bin[j])
50  # on code la chaine dans les pixels suivants après pairisation
51  for i in range(8*longueur):
52      rouge[i+8]=2*(rouge[i+8]//2)+int(chaine[i])
53
54  # on recrée l'image rouge
55  nouveau_rouge = Image.new("L",(largeur,hauteur))
56  nouveau_rouge.putdata(rouge)
57  # fusion des trois nouvelles images
58  nouvelle_image = Image.merge('RGB',(nouveau_rouge,green,blue))
59  nouvelle_image.save("einstein1-code.png")
60  nouvelle_image.show()
61  nouvelle_image.save("einstein1-code.eps")
```

Le résultat :



(a) einstein1 (b) einstein1-code

FIGURE 1.26 – message caché

⇒ **Activité 1. 12**

Écrivez le programme en *Python* de façon à découvrir le message caché dans l'image *einstein1-code.png*.
Vous le nommerez par exemple *decodage.py*.
Vous aurez besoin ici de la fonction `str()` qui sert à convertir une donnée en chaîne.
L'instruction `int(ch,2)` permet de convertir un binaire `ch` sous forme de chaîne en nombre décimal.
L'instruction chr(nombre) permet de traduire l'entier `nombre` sous forme d'entier en caractère codé en *ascii*.

```
>>> str(5)
'5
>>> int("01101001",2)
105
>>> chr(105)
"i"
```

1.5.2.2 Image cachée dans une autre

Pour cacher une image dans une autre, nous prendrons 2 images de même dimension et au format *.tif* de façon à ne pas perdre d'information en route.

Principe

Dans l'écriture d'un nombre, qu'il soit décimal ou binaire, les termes de gauche possèdent un poids plus important. Par exemple, pour $57 = 5 \text{ x } 10^1 + 7 \text{ x } 10^0$, le 5 représente 50, plus proche de la valeur réelle du nombre initial. Il en est de même pour sa version binaire 111001 où le 1 de gauche représente $1 \text{ x } 2^5$.
Les couleurs étant codées sur 8 bits, on ne va retenir que les 4 premiers bits, de poids le plus important (les bits "forts"). Exemple avec le programme suivant :

```
16            image2.putpixel((colonne,ligne),(rouge,vert,bleu))
17
18  image2.save('billes_allege.tif')
19  image2.save('billes_allege.eps')
```

On peut ensuite comparer les 2 images :

```
21  image2.sh
```


(a) billes.tif


(b) billes_allege.tif

FIGURE 1.27 – stegano1

Considérons le dernier pixel de l'*image10* codé en binaire : 10111001. Ses 4 bits forts sont 1011.
De même pour l'*image11* : De 100001, nous ne retiendrons que 0010.
On va ensuite concaténer ces 8 bits, ce qui donnera 10110010.
De cette façon, en affichant la nouvelle image (*image2*), l'*image11* sera peu visible car cachée dans les bits de poids faibles.
Vous trouverez plus bas le programme *stegano2.py* qui permet de réaliser cette "insertion".
Dans les lignes 15 à 17, on fait un *ET* logique "(&)" avec le nombre 240, soit 11110000 en binaire. Ainsi, les 4 derniers bits sont mis à zéro. (Pour mettre à zéro les 2 derniers bits, on utiliserait 252).
Dans les lignes 20 à 22, on utilise "» n" qui permet de décaler de n bits vers la droite (pour décaler à gauche, on utiliserait "« n").
Dans les lignes 24 à 26, l'opérateur "|" (*OU* logique) permet d'insérer les pixels de l'*image11* dans l'*image10* pour obtenir finalement l'*image2*.

stegano2.py

```
#-*- coding: utf-8 -*-

from PIL import Image
image_billes = Image.open('billes.tif')
image_paysage = Image.open('paysage.tif')
largeur, hauteur = image_billes.size

image_finale = Image.new("RGB",(largeur, hauteur))

# Insertion de image_paysage dans image_billes
for ligne in range(hauteur):
    for colonne in range(largeur):
        pixel1 = image_billes.getpixel((colonne,ligne))
        rouge1 = pixel1[0] & 240 # opérateur "and"
        vert1 = pixel1[1] & 240  # on met à zéro les
        bleu1 = pixel1[2] & 240  # 4 derniers bits

        pixel2 = image_paysage.getpixel((colonne,ligne))
        rouge2 = pixel2[0] >> 4 # décalage de 4 bits
        vert2 = pixel2[1] >> 4  # vers la droite
        bleu2 = pixel2[2] >> 4

        rouge = rouge1 | rouge2 # opérateur "or"
        vert = vert1 | vert2    # qui permet de concaténer
        bleu = bleu1 | bleu2    # les 2 images

        image_finale.putpixel((colonne,ligne),(rouge,vert,bleu))
print(bin(pixel1[0])) # 0b10111001 bits forts de l'image10 : 1011
print(bin(pixel2[0])) # 0b100001   bits forts de l'image11: 0010
print(bin(rouge1))    # 0b10110000 image_billes
print(bin(rouge2))    # 0b10 image_paysage
print(bin(rouge))      # 0b10110010 bits de l'image_finale

image_finale.save('billes_incrust.tif')
image_billes.show()
image_paysage.show()
image_finale.show()
```

⇒ **Activité 1. 13**

Écrivez un programme, appelé par exemple *stegano3.py*, qui permet de réaliser l'opération inverse, c'est-à-dire de récupérer l'image cachée.

Le programme :

1.5.3 Traitement des images avec *numpy*

Dans cette partie, on importe le module *numpy* sous l'alias *np* et on récupère la matrice de l'objet image avec la fonction `np.array()`. Les *array* de *numpy* sont des tableaux multidimensionnels comme les listes de *Python* mais qui ne peuvent contenir que des objets du même type (*int*, *float*, *bool*, *complex*) contrairement aux listes. Enfin on aplatit cette matrice 2×2 avec la méthode `flatten()`.
Un exemple pour commencer avec le programme *image-numpy-1.py* :

image-numpy-1.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Thu Jan 24 21:10:23 2019
5
6  @author: marco5
7  """
8
9  from PIL import Image
10 import numpy as np
11
12 mon_image = Image.open("billes.jpg")
13 data = mon_image.getdata()
14 print("data[0] : ", data[0])
15 print("len(data) : ",len(data))
16 image_np = np.array(mon_image)
17 largeur, hauteur, profondeur = image_np.shape
18 print("image_np.shape : ",image_np.shape)
19 print("largeur,hauteur : ",largeur,hauteur)
20 print(image_np[0][0],image_np[0][1],image_np[0][2])
```

```
21  data_plat = image_np.flatten()
22  print("data_plat : ",data_plat)
23  print("data_plat.shape : ",data_plat.shape)
24  print("data_plat[0] : ",data_plat[0])
25  mat1 = np.array([[12,12,15,28],[4,15,18,56],[14,5,11,26]])
26  print("mat1.shape : ",mat1.shape)
27  mat2 = np.zeros((3,6))
28  print("mat2 : ",mat2)
29  print("mat2.shape : ",mat2.shape)
30  img1=np.zeros((5,2,1),dtype=np.uint8)
31  img2=np.zeros((5,2,4),dtype=np.uint8)
32  print("img1 : ",img1)
33  print("img2 : ",img2)
```

Le résultat en console :

```
data[0] :  (176, 160, 135)
len(data) :  262144
image_np.shape :  (512, 512, 3)
largeur,hauteur :  512 512
[176 160 135] [175 159 134] [173 157 132]
data_plat :  [176 160 135 ... 146 127  95]
data_plat.shape :  (786432,)
data_plat[0] :  176
mat1.shape :  (3, 4)
mat2 :  [[0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0.]]
mat2.shape :  (3, 6)
img1 :  [[[0]
[0]]

[[0]
[0]]

[[0]
[0]]

[[0]
[0]]

[[0]
[0]]]
img2 :  [[[0 0 0 0]
[0 0 0 0]]

[[0 0 0 0]
[0 0 0 0]]

[[0 0 0 0]
[0 0 0 0]]

[[0 0 0 0]
[0 0 0 0]]

[[0 0 0 0]
[0 0 0 0]]]
```

On pourra également se reporter à la partie qui traite de ce module *numpy* dans le document distribué en première année.

On rappelle que la matrice d'une image couleur est semblable à la matrice d'une image en niveaux de gris, sauf que chaque pixel est représenté par un triplet de nombres entre 0 et 255 au lieu d'un nombre. Cette propriété est illustrée à la fin du programme précédent.
Par exemple, `image[i,j][k]` est un nombre entre 0 et 255 qui représente l'intensité du canal k ($k = 0$ pour le rouge, $k = 1$ pour le vert et $k = 2$ pour le bleu).
Pour créer une image couleur remplie de 0, il faut initialiser les triplets de couleur. On la déclarera avec :
`image = np.zeros([nb_lignes,nb_colonnes,3],dtype=np.uint8)`

Ainsi, pour convertir une image en niveaux de gris avec 3 canaux identiques (profondeur = 3), on pourra utiliser la fonction suivante :

```
def niveaux_de_gris (image):
  hauteur,largeur,profondeur = image.shape
  new_image=np.zeros((hauteur,largeur,profondeur),dtype=np.uint8)
  for i in range (hauteur) :
    for j in range (largeur) :
      new_image[i,j] = int(image[i,j][0]/3+image[i,j][1]/3+image[i,j][2]/3)
      # pb de dépassement de uint8 si division après la somme
  return new_image
```

On pourra visualiser l'image originelle avec :

```
import numpy as np
from PIL import Image

nom_image = "billes.jpg"
image = Image.open(nom_image)
mon_image = np.array(image)
image.show()
```

puis l'image obtenue avec :

```
ndg = niveaux_de_gris(mon_image)
nouvelle_image = Image.fromarray(ndg)
nouvelle_image.show()
```

Quelques opérations maintenant que vous devez réaliser en gardant pour chaque image les 3 canaux RGB et en travaillant bien entendu avec le module *numpy*...

1.5.3.1 Conversion en niveaux de gris avec la valeur du rouge

Écrire la fonction `image_rouge_gris` qui partant d'une image en couleurs renvoie une image en niveau de gris dont chaque pixel contient l'intensité en rouge de l'image précédente.
Par exemple si un pixel de l'image originelle a comme valeur $(203, 15, 23)$, la valeur du même pixel de l'image en niveaux de gris sera 203.

1.5.3.2 Conversion en rouge

Écrire la fonction `image_rouge` qui partant d'une image en couleurs renvoie une image en couleur dont chaque pixel ne contient que l'intensité en rouge de l'image originelle.
Par exemple si un pixel de l'image originelle vaut $(203, 15, 23)$, la valeur du même pixel de l'image en couleurs sera $(203, 0, 0)$.

1.5.3.3 Conversion en noir et blanc

Écrire la fonction `noir_et_blanc` qui partant d'une image en couleurs renvoie une image en noir (0) et blanc (255).
Par exemple si un pixel de l'image originelle vaut $(203, 15, 23)$, la valeur du même pixel de l'image en couleurs sera $(0, 0, 0)$. Si un pixel de l'image originelle vaut $(203, 180, 102)$, la valeur du même pixel de l'image en couleurs sera $(255, 255, 255)$.

1.5.3.4 Rotation trigonométrique

Écrire la fonction `rotation` qui effectue une rotation de $90°$ de l'image dans le sens trigonométrique.

1.5.3.5 Dilatation

Écrire la fonction dilatation qui prend en argument une image en noir et blanc et qui rend noir un pixel si au moins un de ses voisins est noir et le rend blanc sinon.

1.5.3.6 Érosion

Écrire la fonction `erosion` qui prend en argument une image en noir et blanc et qui rend noir un pixel si tous ses voisins sont noirs et le rend blanc sinon.

1.5.3.7 Détection de contours

Écrire la fonction `contour` qui prend en argument une image en niveaux de gris sur 3 canaux et détecte les contours par convolution. On prendra comme noyau la matrice suivante :

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Le programme :

1.6 Exercices

1.6.1 Palettes de couleurs

► Exercice 1. 1 : Palette de gris

Créez une palette en niveaux de gris, par exemple une bande de hauteur 100 pixels et de largeur 2560 pixels, allant du noir à gauche au blanc à droite.

Le résultat :

FIGURE 1.28 – palette_gris

► Exercice 1. 2 : Palette de couleurs

Créez une palette en couleurs, par exemple une bande de hauteur 256 pixels et de largeur 256 pixels. plusieurs possibilités existent : à vous de choisir.

Un résultat possible :

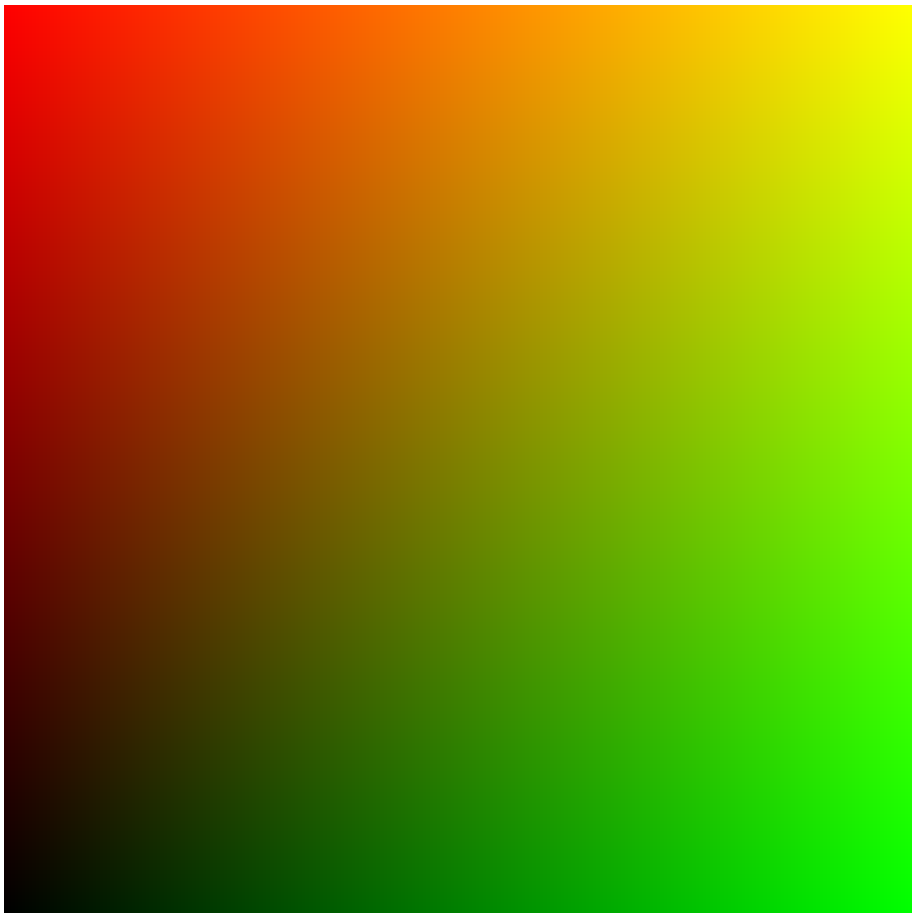


FIGURE 1.29 – palette-coul

1.6.2 Mosaïque de couleurs

► Exercice 1.3 : Mosaïque de couleurs

Découpez une image en 9 rectangles et gardez un seul canal (ou 2) pour chaque rectangle.

Un résultat possible :

FIGURE 1.30 – mosaique

1.6.3 Fusion d'images

► Exercice 1.4 : Fusion d'images

Ouvrez 2 images et si elles sont de tailles différentes, prenez la plus petite. Pour les fusionner, on peut prendre le maximum de chaque canal.

Un résultat possible :

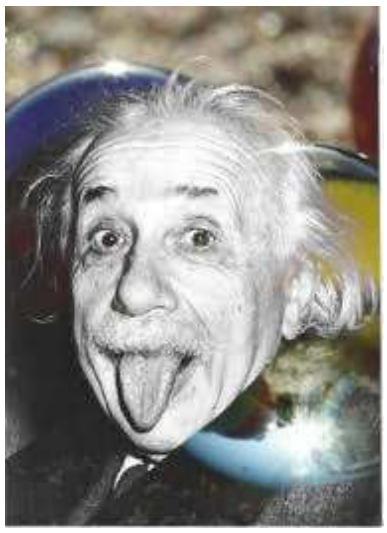


FIGURE 1.31 – fusion

1.6.4 Ajout d'une bordure

Principe

Pour créer une bordure, il faut créer une image plus grande, faire le cadre (par exemple blanc, donc avec des valeurs prises à 255), puis copier l'image initiale au milieu.

► Exercice 1.5 : Cadre

Écrivez une fonction permettant de créer un cadre de largeur n pixels de la couleur de votre choix sur une image. Vous pourrez compléter le programme *bordure-eleve.py*.

bordure-eleve.py

```
1  # -*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  ext1='.jpg'
6  ext2='.eps'
7
8  def bordure(image,nb_bord=20,rouge=100,vert=150,bleu=250):
9
10     largeur,hauteur = image.size
11
12     new_largeur=.................
13     new_hauteur=.................
14
15     nouvelle_image = Image.new("RGB",\
16     (new_largeur,new_hauteur))
17
18     for colonne in range(new_largeur):
19         for ligne in range(new_hauteur):
20             nouvelle_image.putpixel((colonne,ligne),\
21             (rouge,vert,bleu))
22
23     for ligne in range(hauteur):
24         for colonne in range(largeur):
25             pix = image.getpixel((colonne,ligne))
26             nouvelle_image.putpixel(.................)
27
28     return(nouvelle_image)
29
30 im = "billes"
31 mon_image = Image.open(im+ext1)
32 ma_nouvelle_image = bordure(mon_image)
33 ma_nouvelle_image.save(im+'_bord'+ext2)
34 ma_nouvelle_image.show()
```

Un résultat possible :

FIGURE 1.32 – billes_bord

1.6.5 Pixelisation

Dans la suite, on traitera des images en niveaux de gris ou on les convertira, pour plus de facilité.

Principe

Pour pixeliser une image, on peut faire la moyenne par carrés de 4 pixels par exemple. Cela nécessite un nombre pair de pixels en largeur et en hauteur.

► Exercice 1.6 : Pixelisation

Écrivez la fonction correspondante, en supposant que l'image de départ contient effectivement un nombre pair de pixels en largeur et en hauteur. Vous pourrez compléter le programme *pixelisation-eleve.py*.

pixelisation-eleve.py

```
1  #-*- coding: utf-8 -*-
2
3  from PIL import Image
4
5  # Si l'image contient un nombre pair de pixels
6  def pixelisation(image1):
7      image_gris = image1.convert('L')
8      largeur,hauteur = image_gris.size
9      pixel_gris = image_gris.load()
10     image2 = Image.new("L",(largeur,hauteur))
11     nouveau_pixel = image2.load()
12     for colonne in range (.............):
13         for ligne in range (.............):
14             moyenne = ........................
15             nouveau_pixel[colonne,ligne] = \
16             nouveau_pixel[colonne+1,ligne] = \
17             nouveau_pixel[colonne,ligne+1] = \
18             nouveau_pixel[colonne+1,ligne+1] = moyenne
19     return image2
20
21 nom_fichier = 'billes'
22 ext1 = '.jpg'
23 ext2 = '.eps'
24 mon_image=nom_fichier+ext1
25
26 image_originale = Image.open(mon_image)
27 image_pix = pixelisation(image_originale)
28 image_pix.save(nom_fichier+'_pixel'+ext1)
```

```
29 image_pix.save(nom_fichier+'_pixel'+ext2)
30 image_pix.show()
```

Le résultat :

FIGURE 1.33 – billes_pixel

1.6.6 Bruitage

Principe

Le bruit d'image est la présence d'informations parasites ajoutées de façon aléatoire à l'image.

► Exercice 1.7 : Bruitage

Écrivez une fonction qui ajoute du bruit sur une image.

Vous pourrez utiliser le module *random* ainsi que *randint(a,b)* pour générer des entiers aléatoires compris entre a et b.

Pour ajouter du bruit, vous prendrez $[a,b] = [0,20]$ par exemple et vous créerez une **fonction**.

Le résultat :

FIGURE 1.34 – billes_bruit

1.6.7 Floutage

Principe

Le floutage peut se réaliser avec la méthode de filtrage vue page 48 avec la matrice :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

► Exercice 1.8 : Floutage

Utilisez la méthode vue page 48 pour effectuer du floutage sur une image de votre choix. Vous créerez une **fonction**.

⚠ Il ne faudra pas oublier de ramener les valeurs dans le domaine visible puisque $s_c = 9 \ldots$

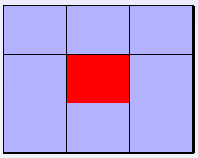
Ce que l'on obtient :

FIGURE 1.35 – billes_flou

1.6.8 Dilatation

Principe

La morphologie mathématique repose sur l'utilisation d'un élément structurant. Ce dernier est composé d'un pixel central (ici en rouge) et d'un ensemble de pixels (ici en bleu). Il se déplace sur la matrice *image_initiale* qui est en noir (bit de valeur 0) et blanc (bit de valeur 1) et lorsque le pixel central de l'élément structurant (en rouge) est sur un pixel *pix[x,y]* de l'image, un test est réalisé sur les pixels voisins (ici 8) pour déterminer la valeur du pixel *nouveau_pix* de la matrice *image_finale*.
On se donne ici un élément structurant simple :



L'ébauche de l'algorithme pourrait être comme suit :

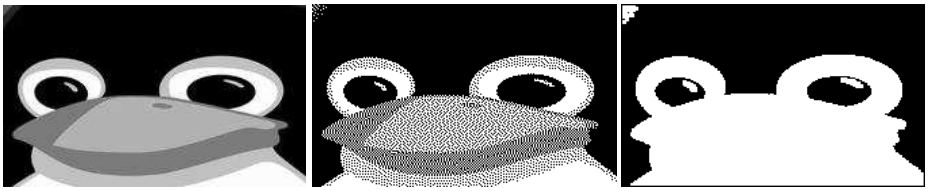
- Parcourir les pixels de l'*image_initiale*
- Pour chaque pixel *pix[x,y]* :
  - Centrer l'élément structurant sur ce pixel,
  - Considérer les 8 voisins du pixel *pix[x,y]*,
  - Si l'un de ces pixels est blanc, mettre *nouveau_pix* en blanc (bit de valeur 1).

Évidemment, les bords de l'image vont poser problème. L'astuce consiste à créer une bordure de largeur 1 pixel de couleur noire autour de l'*image_initiale*.

► Exercice 1.9 : Dilatation

Proposez un programme permettant d'effectuer cette opération de dilatation, par exemple sur l'image *tux20_NetB.bmp*.

Les images obtenues :



(a) tux_20_gris (b) tux_20_NetB (c) tux_20_dilat

FIGURE 1.36 – dilatation

1.6.9 Érosion

Principe

On utilise la même technique que précédemment avec le même élément structurant.
L'ébauche de l'algorithme est très semblable :

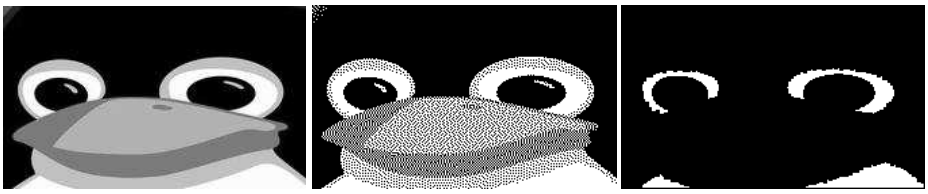
- Parcourir les pixels de l'*image_initiale*
- Pour chaque pixel *pix[x,y]* :
  - Centrer l'élément structurant sur ce pixel,
  - Considérer les 8 voisins du pixel *pix[x,y]*,
  - Si l'un de ces pixels est noir, mettre *nouveau_pix* en noir (bit de valeur 0).

Là encore, les bords de l'image vont poser problème. L'astuce consiste ici à créer une bordure de largeur 1 pixel de couleur blanche autour de l'*image_initiale*.

► Exercice 1. 10 : Érosion

Proposez un programme permettant d'effectuer cette opération d'érosion, par exemple sur l'image *tux20_NetB.bmp*.

Les images obtenues :



(a) tux_20_gris (b) tux_20_NetB (c) tux_20_erod

FIGURE 1.37 – érosion

1.6.10 Dilatation et érosion en niveaux de gris

Principe

La démarche est la même sauf que le pixel central de l'élément structurant prend la valeur maximale des pixels environnants pour la dilatation et la valeur minimale pour l'érosion.

► Exercice 1. 11 : Dilatation et érosion en gris

Modifiez les programmes précédents de façon à pouvoir dilater et éroder une image en niveaux de gris, par exemple l'image *tux20_gris.png*.

Les résultats :

(a) tux_20_gris (b) tux_20_dilat_bis (c) tux_20_erod_bis

FIGURE 1.38 – dilatation et érosion en niveaux de gris

1.7 Solutions des exercices

2

Cryptologie

2.1 Introduction

La cryptologie est la science des messages secrets.Elle se décompose en deux parties :

- la cryptographie, qui est l'art de crypter un message, c'est-à-dire de transformer un message intelligible en un autre incompréhensible pour le commun des mortels,
- la cryptanalyse, qui est l'art d'analyser un message crypté afin de le décrypter.

Nous allons étudier dans ce chapitre plusieurs façons de crypter un message et évidemment de le décrypter, voire de casser son cryptage.
Évidemment, il y a de multiples méthodes de cryptage, monoalphabétiques ou polyalphabétiques. Nous nous restreindrons ici à quelques méthodes monoalphabétiques.
Auparavant, nous formaterons le texte à crypter.
Nous allons également analyser les fréquences de lettres d'un texte.
Tous les programmes et fonctions seront ici écrites en *Python*.

Nous aurons notamment besoin dans cette étude des fonctions `ord()`, `chr()`, % et // :

```
>>> ord('A')
65
>>> ord('a')
97
>>> chr(66)
'B'
>>> for i in range(65,91):
...     print(chr(i),end=',')
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
>>> for i in range(97,123):
...     print(chr(i),end=',')
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,
>>> 25%3
1
>>> 25//3
8
```

2.2 Formatage du texte

2.2.1 Gestion des accents, cédilles, ...

On peut effectuer le remplacement des lettres accentuées par exemple, dans une chaîne :

```
>>> ma_chaine = "arrêté grâce à Loïc
>>> ma_chaine.replace("ê","e").replace("ï","i")
'arreté grâce à Loic'
```

On peut également écrire une fonction pour remplacer toutes les lettres accentuées. On les place tout d'abord dans un dictionnaire :

```
dictionnaire = {"é" : 'e', "ê" : 'e', "è" : "e", "î" : "i", "ï" : "i", "ö" : "o", "ô"
: "o", "à" : "a", "â" : "a", "ù" : "u", "ç" : "c", "æ" : "ae", "œ" : "oe"}
```

Puis, pour chaque caractère accentué (ce sont les clés) de la chaîne `texte`, on le remplace par sa valeur :

crypt-01.py{11}{14}

```
1 def multi_replace(texte, dico):
2     for clef in dico:
3         texte = texte.replace(clef, dico[clef])
4     return texte
```

2.2.2 Élimination de la ponctuation et des espaces

Les lettres majuscules sont codées en ASCII de 65 à 90 inclus.

⇒ **Activité 2. 14**

Écrivez une fonction `epurage(chaine)` prenant un argument `chaine` qui permet de passer le message en majuscules (ou bien en minuscules si vous préférez) puis d'éliminer la ponctuation et des espaces dans un texte.

Le programme :

2.2.3 Séparation du texte en blocs

Souvent, on découpe les messages en cryptographie : les textes sont séparés en blocs de 4 ou 5 caractères.

⇒ **Activité 2. 15**

Écrivez une fonction `separe_bloc(chaine, bloc=4)` prenant en argument `chaine` ainsi qu'un argument optionnel `bloc` qui permet de découper le texte en blocs de n caractères (4 par défaut) en insérant un espace entre chaque bloc.

Le programme :

2.2.4 Opérations sur un fichier texte

Nous pouvons faire la même chose avec un fichier texte avec une extension `.txt`.
Pour le lire, on peut utiliser la fonction suivante :

```
def lecture(fichier):
  with open(fichier,'r') as source:
    resultat = source.read()
  return resultat # [:20] # si texte très long

message2 = lecture("poeme1.txt")
```

⇒ **Activité 2. 16**

Rassemblez ce qui précède dans un seul programme et le tester avec le fichier *poeme1.txt*.
Nous obtenons alors le programme *crypt-01.py* :

Le programme :

2.3 Outils pour le cryptologue

2.3.1 Analyse de fréquence

Principe

L'analyse de fréquence d'un texte permet, lorsque nous avons remplacé une lettre d'un texte par une autre, de reconnaître les lettres les plus fréquentes du texte, le "e" en français, par exemple.

Dans la langue française, en faisant abstraction des accents et suivant le type de langage utilisé, nous pouvons prendre les pourcentages suivants :

Lettre	A	B	C	D	E	F	G	H	I	J	K	L	M
Fréquence	8.40	1.06	3.03	4.18	17.26	1.12	1.27	0.92	7.34	0.31	0.05	6.01	2.96
Lettre	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Fréquence	7.13	5.26	3.01	0.99	6.55	8.08	7.07	5.74	1.32	0.04	0.45	0.30	0.12

Nous pourrons pour plus de précisions et d'autres langues, nous reporter au tableau 2.2 en fin de chapitre.

2.3.1.1 Fréquence d'une lettre de l'alphabet

Voici un algorithme qui permet de calculer la fréquence de la lettre `lettre` dans le texte `texte` :

Fréquence d'une lettre

```
1   VARIABLES
2   nbcar, cpt, i : entier
3   freq : réel
4   lettre : chaine
5   texte : chaine
6   DEBUT_ALGORITHME
7      INITIALISATION
8      nbcar ← longueur(texte)
9      cpt ← 0
10      POUR i ALLANT_DE 1 A nbcar
11         DEBUT_POUR
12         SI texte[i] = lettre ALORS
13            DEBUT_SI
14            cpt ← cpt + 1
15            FIN_SI
16         FIN_POUR
17      freq ← cpt/nbcar
18  FIN_ALGORITHME
```

2.3.1.2 Fréquence de l'ensemble des lettres de l'alphabet

Nous prendrons par exemple :

```
alphabet_min="abcdefghijklmnopqrstuvwxyz"
alphabet_maj="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

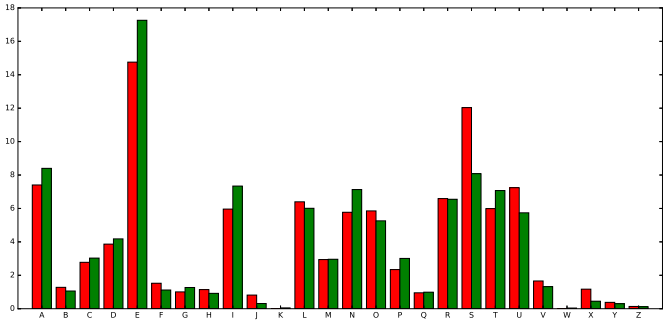
⇒ **Activité 2. 17**

Modifiez l'algorithme précédent de façon à calculer la fréquence de chacune des lettres et déduisez-en la fonction `freq_lettre(texte)` en *python* correspondante.

2.3.1.3 Illustration graphique

⇒ **Activité 2. 18**

Représentez, sous forme d'histogramme et sur un même graphe, les fréquences des lettres d'un texte et les fréquences de l'alphabet en général dans la langue française.
Nous obtenons par exemple :



Le programme :

2.3.2 Outils de cryptanalyse

2.3.2.1 Indice de coïncidence

Pour casser le code de Vigenère, nous aurons besoin de calculer ce que l'on appelle l'indice de coïncidence d'un texte. Son expression est la suivante :

$$I_C = \frac{\sum_{\lambda=A}^{\lambda=Z} f_\lambda\ (f_\lambda - 1)}{n\ (n-1)} = \frac{\sum_{\lambda=A}^{\lambda=Z} n_\lambda\ (n_\lambda - 1)}{n\ (n-1)}$$

où :

- n est le nombre total de lettres du message,
- n_λ (respectivement f_λ) est le nombre (respectivement la fréquence) de lettres λ dans le message.

En français, l'indice de coïncidence vaut environ $0{,}074$ (si en remplaçant les caractères accentués par les mêmes non accentués), en anglais, il vaut environ $0{,}065$ alors que pour un texte dont les lettres seraient distribuées de façon aléatoire, il faudrait $0{,}0385$.

Remarque

Le livre de Georges Perec "La disparition" ne contient pas la lettre "e"...

⇒ **Activité 2. 19**

En modifiant la fonction précédente `freq_lettre(texte)`, nous pouvons écrire une nouvelle fonction `indice_coinc(texte)` qui calcule l'indice de coïncidence d'un texte. C'est votre mission.

Le programme :

2.3.2.2 Indice de coïncidence mutuelle

Pour casser le code de *César*, nous aurons besoin de calculer ce que l'on appelle l'indice de coïncidence mutuelle de deux chaînes.
Soient deux chaînes `ch1` et `ch2` de longueurs respectives ℓ_1 et ℓ_2. L'indice de coïncidence mutuelle est la probabilité qu'un caractère aléatoire de `ch1` soit égal à un caractère aléatoire de `ch2`.
Son expression est alors la suivante :

$$I_{CM} = \frac{\sum_{i=0}^{25} f_{i_1} f_{i_2}}{\ell_1 \ell_2}$$

où :

- les f_{i_1} sont les fréquences des 26 caractères dans `ch1`,
- les f_{i_2} sont les fréquences des 26 caractères dans `ch2`.

On pourra prendre $\ell_2 = 1$.

⇒ **Activité 2. 20**

Écrivez une fonction `ind_coinc_mutuel(texte)` qui calcule cet indice en prenant pour une des chaînes l'alphabet avec comme fréquences :

```
freq_francais = [8.40, 1.06, 3.03, 4.18, 17.26, 1.12, 1.27, 0.92, 7.34, 0.31, 0.05,\
6.01, 2.96, 7.13, 5.26, 3.01, 0.99, 6.55, 8.08, 7.07, 5.74, 1.32, 0.04, 0.45, 0.30,
    0.12]
```

Vous pourrez utiliser la fonction `indice_coinc(texte)` précédemment définie. Vous pourrez également utiliser la fonction `zip` dont le comportement est illustré ci-dessous :

```
>>> list_1 = [1,2,3]
>>> list_2 = [3,4,5]
>>> sum(a*b for a,b in zip(list_1,list_2))
26
>>> sum(a+b for a,b in zip(list_1,list_2))
18
>>> sum(b/a for a,b in zip(list_1,list_2))
6.666666666666667
>>> sum(b-a for a,b in zip(list_1,list_2))
6
```

Le programme :

2.3.3 Outils mathématiques

Certains outils mathématiques sont indispensables en cryptologie [1].
Il est notamment souvent question de congruence *modulo n*, de *pgcd*, ...

2.3.3.1 Les nombres premiers

Nombres premiers entre eux

On dit que deux nombres entiers relatifs non nuls a et b sont premiers entre eux lorsqu'ils n'admettent pas de diviseur commun autre que le nombre 1.

1. Certaines parties mathématiques parmi les suivantes sont largement inspirées de la partie *Cryptographie* de Arnaud Bodin et François Recher.

Par conséquent :
On dit que deux nombres entiers relatifs non nuls a et b sont premiers entre eux lorsque leur pgcdest égal à 1.

Égalité de Bézout

Soit a et b deux entiers relatifs non nuls et D leur *PGDC*. Il existe deux entiers relatifs u et v tels que :

$$a\,u + b\,v = D$$

au ‘ bv “ D

Théorème de Bézout

Soit a et b deux entiers relatifs non nuls.

a et b sont premiers entre eux $\iff$ il existe deux entiers relatifs u et v tels que $a\,u + b\,v = 1$

2.3.3.2 Modulo

Soit $n \geqslant 2$ un entier fixé.

Modulo

On dit que a **est congru à** b **modulo** n, si n divise $b - a$. On note alors :

$$a \equiv b\ (n) \equiv b\ [n] \equiv b \pmod{n}$$

Pour nous $n = 26$. Ce qui fait que $28 \equiv 2 \pmod{26}$, car $28 - 2$ est bien divisible par 26. De même $85 = 3 \times 26 + 7$ donc $85 \equiv 7 \pmod{26}$.

On note $\mathbb{Z}/26\mathbb{Z}$ l’ensemble de tous les éléments de $\mathbb{Z}$ modulo 26. Cet ensemble peut par exemple être représenté par les 26 éléments $\{0, 1, 2, \ldots, 25\}$. En effet, puisqu’on compte modulo 26 :

$$0, 1, 2, \ldots, 25, \quad \text{puis} \quad 26 \equiv 0, 27 \equiv 1, 28 \equiv 2, \ldots, \quad 52 \equiv 0, 53 \equiv 1, \ldots$$

et de même $-1 \equiv 25$, $-2 \equiv 24$, …

Plus généralement $\mathbb{Z}/n\mathbb{Z}$ contient n éléments. Pour un entier $a \in \mathbb{Z}$ quelconque, son **représentant** dans $\{0,1,2,\ldots,n-1\}$ s’obtient comme le reste k de la division euclidienne de a par n : $a = bn + k$. De sorte que $a \equiv k \pmod{n}$ et $0 \leq k < n$.

De façon naturelle l’addition et la multiplication d’entiers se transposent dans $\mathbb{Z}/n\mathbb{Z}$.

Pour $a,b \in \mathbb{Z}/n\mathbb{Z}$, on associe $a + b \in \mathbb{Z}/n\mathbb{Z}$.

Par exemple dans $\mathbb{Z}/26\mathbb{Z}$, $15 + 13$ égale 2. En effet $15 + 13 = 28 \equiv 2 \pmod{26}$. Autre exemple : que vaut $133 + 64$? $133 + 64 = 197 = 7 \times 26 + 15 \equiv 15 \pmod{26}$. Mais on pourrait procéder différemment : tout d’abord $133 = 5 \times 26 + 3 \equiv 3 \pmod{26}$ et $64 = 2 \times 26 + 12 \equiv 12 \pmod{26}$. Et maintenant sans calculs : $133 + 64 \equiv 3 + 12 \equiv 15 \pmod{26}$.

On fait de même pour la multiplication : pour $a,b \in \mathbb{Z}/n\mathbb{Z}$, on associe $a \times b \in \mathbb{Z}/n\mathbb{Z}$.

Par exemple 3×12 donne 10 modulo 26, car $3 \times 12 = 36 = 1 \times 26 + 10 \equiv 10 \pmod{26}$. De même : $3 \times 27 = 81 = 3 \times 26 + 3 \equiv 3 \pmod{26}$. Une autre façon de voir la même opération est d’écrire d’abord $27 = 1 \pmod{26}$ puis $3 \times 27 \equiv 3 \times 1 \equiv 3 \pmod{26}$.

2.3.3.3 Le petit théorème de Fermat amélioré

Voici le petit théorème de Fermat :

Petit théorème de Fermat

Si p est un nombre premier et $a \in \mathbb{Z}$ alors $a^p \equiv a \pmod{p}$

et sa variante :

Corollaire

Si p ne divise pas a alors : $a^{p-1} \equiv 1 \pmod{p}$

Nous allons voir une version améliorée de ce théorème dans le cas qui nous intéresse :

Petit théorème de Fermat amélioré

Soient p et q deux nombres premiers distincts et soit $n = pq$. Pour tout $a \in \mathbb{Z}$ tel que $\text{pgcd}(a, n) = 1$ alors : $a^{(p-1)(q-1)} \equiv 1 \pmod{n}$

On note $\varphi(n) = (p-1)(q-1)$, la **fonction d'Euler**. L'hypothèse $\text{pgcd}(a, n) = 1$ équivaut ici à ce que a ne soit divisible ni par p, ni par q. Par exemple pour $p = 5$, $q = 7$, $n = 35$ et $\varphi(n) = 4 \cdot 6 = 24$. Alors pour $a = 1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18, \ldots$ on a bien $a^{24} \equiv 1 \pmod{35}$.

Démonstration

Notons $c = a^{(p-1)(q-1)}$. Calculons c modulo p :

$$c \equiv a^{(p-1)(q-1)} \equiv (a^{(p-1)})^{q-1} \equiv 1^{q-1} \equiv 1 \pmod{p}$$

où l'on appliquer le petit théorème de Fermat : $a^{p-1} \equiv 1 \pmod{p}$, car p ne divise pas a.
Calculons ce même c mais cette fois modulo q :

$$c \equiv a^{(p-1)(q-1)} \equiv (a^{(q-1)})^{p-1} \equiv 1^{p-1} \equiv 1 \pmod{q}$$

où l'on appliquer le petit théorème de Fermat : $a^{q-1} \equiv 1 \pmod{q}$, car q ne divise pas a.
Conclusion partielle : $c \equiv 1 \pmod{p}$ et $c \equiv 1 \pmod{q}$.

Nous allons en déduire que $c \equiv 1 \pmod{pq}$.
Comme $c \equiv 1 \pmod{p}$ alors il existe $\alpha \in \mathbb{Z}$ tel que $c = 1 + \alpha p$; comme $c \equiv 1 \pmod{q}$ alors il existe $\beta \in \mathbb{Z}$ tel que $c = 1 + \beta q$. Donc $c - 1 = \alpha p = \beta q$. De l'égalité $\alpha p = \beta q$, on tire que $p | \beta q$.
Comme p et q sont premiers entre eux (car ce sont des nombres premiers distincts) alors par le lemme de Gauss on en déduit que $p | \beta$. Il existe donc $\beta' \in \mathbb{Z}$ tel que $\beta = \beta' p$.
Ainsi $c = 1 + \beta q = 1 + \beta' pq$. Ce qui fait que $c \equiv 1 \pmod{pq}$, c'est exactement dire $a^{(p-1)(q-1)} \equiv 1 \pmod{n}$.

2.3.3.4 L'algorithme d'Euclide étendu

Nous avons déjà étudié l'algorithme d'Euclide qui repose sur le principe que $\text{pgcd}(a, b) = \text{pgcd}(b, a \pmod{b})$.

Voici sa mise en œuvre informatique.

euclide.py{5}{8}

```
1  def euclide(a,b):
2      while b !=0 :
3          a , b = b , a % b
4      return a
```

On profite que *Python* assure les affectations simultanées, ce qui pour nous, correspond aux suites

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & \equiv & a_i \pmod{b_i} \end{cases}$$

initialisée par $a_0 = a$, $b_0 = b$.

Nous avons vu aussi comment « remonter » l'algorithme d'Euclide à la main pour obtenir les cœfficients de Bézout u, v tels que $a\,u + b\,v = \text{pgcd(a, b)}$. Cependant il nous faut une méthode plus automatique pour obtenir ces cœfficients, c'est l'**algorithme d'Euclide étendu**.

On définit deux suites (x_i), (y_i) qui vont aboutir aux cœfficients de Bézout.

L'initialisation est :

$$x_0 = 1 \qquad x_1 = 0 \qquad y_0 = 0 \qquad y_1 = 1$$

et la formule de récurrence pour $i \geq 1$:

$$x_{i+1} = x_{i-1} - q_i x_i \qquad y_{i+1} = y_{i-1} - q_i y_i$$

où q_i est le quotient de la division euclidienne de a_i par b_i.

euclide.py{14}{22}

```
 6  def euclide_etendu(a,b):
 7      x = 1 ; xx = 0
 8      y = 0 ; yy = 1
 9      while b !=0 :
10          q = a // b
11          a , b = b , a % b
12          xx , x = x - q*xx , xx
13          yy , y = y - q*yy , yy
14      return (a,x,y)
```

Cet algorithme renvoie d'abord le *pgcd*, puis les cœfficients u,v tels que $a\,u + b\,v = \text{pgcd(a, b)}$.

2.3.3.5 Inverse modulo n

Soit $a \in \mathbb{Z}$, on dit que $x \in \mathbb{Z}$ est un **inverse de** a **modulo** n si $a\,x \equiv 1 \pmod{n}$.

Trouver un inverse de a modulo n est donc un cas particulier de l'équation $ax \equiv b \pmod{n}$.

- a admet un inverse modulo n si et seulement si a et n sont premiers entre eux.
- Si $a\,u + n\,v = 1$ alors u est un inverse de a modulo n.

En d'autres termes, trouver un inverse de a modulo n revient à calculer les cœfficients de Bézout associés à la paire (a, n).

Démonstration

La preuve est essentiellement une reformulation du théorème de Bézout :

$$\begin{aligned} \text{pgcd(a, n)} = 1 &\iff \exists\, u, v \in \mathbb{Z} \qquad a\,u + n\,v = 1 \\ &\iff \exists\, u \in \mathbb{Z} \qquad au \equiv 1 \pmod{n} \end{aligned}$$

Voici le code :

euclide.py{32}{37}

```
16  def inverse(a,n):
17      c,u,v = euclide_etendu(a,n)
```

```
18      if c != 1 :           # Si pgcd différent de 1 renvoie 0
19          return 0
20      else :
21          return u % n      # Renvoie l'inverse
```


2.3.3.6 Exponentiation rapide

Cette partie a déjà été traitée en cours dans le chapitre sur la complexité.

1. Méthode naïve

   (a) Algorithmique :

   Calculer a^n nécessite a priori $n-1$ multiplications selon l'algorithme suivant :

   Exponentiation naïve

   ```
   VARIABLES
   a : float
   n : int
   ENTRÉES
   un réel a et une puissance n
   INITIALISATION
   Resultat ← a
   DEBUT_ALGORITHME
      POUR k ALLANT_DE 2 A n
          DEBUT_POUR
          Resultat ← Resultat × a
          FIN_POUR
       AFFICHER Resultat
    FIN_ALGORITHME
   ```

   (b) Avec *Python* :

   puiss-naive.py{3}{7}

   ```
   def puissance_naive(a,n):
       res=a
       for k in range(1,n):
           res*=a
       return res
   ```

2. Algorithme de HÖRNER

   (a) Conventions :

   Dans la suite, nous confondrons polynôme et fonction polynôme.

   (b) Principe :

   Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications.

   On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1}) x + a_{n-2}) x + a_{n-3}) x + \cdots) x + a_0 \end{aligned}$$

   Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum 2 fois le degré de P opérations (voire moins avec les zéros).

(c) L'algorithme :

On peut essayer de faire mieux.

Voyons une première méthode qui utilise l'algorithme de HÖRNER étudié précédemment et la décomposition en base 2 de l'exposant.

Par exemple, $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. On peut donc associer à cette écriture un polynôme P tel que 11 soit égal à $P(2)$:

$$P(X) = 1 \times X^3 + 0 \times X^2 + 1 \times X^1 + 1 \times X^0 = ((X+0)X+1)X+1$$

Donc

$$\begin{aligned} a^{P(2)} &= a^{1+2(1+2(2\times 1+0))} \\ &= a \times a^{2(1+2(2\times 1+0))} \\ &= a \times \left(a^{1+2(2\times 1+0)}\right)^2 = a^1 \times \left(a^1 \times \left(a^0 \times \left(a^1\right)^2\right)^2\right)^2 \\ &= \left(\left(\left(a^1\right)^2 a^0\right)^2 a^1\right)^2 a^1 \end{aligned}$$

On en déduit l'algorithme :

Exponentiation « à la HÖRNER »

```
1   VARIABLES
2   a : float
3   n : int
4   ENTRÉES
5   un réel a et un entier n
6   INITIALISATION
7   C ← décomposition de n en base 2
8   res ← a
9   DEBUT_ALGORITHME
10      POUR j ALLANT_DE 1 A taille de C − 1
11          DEBUT_POUR
12          res ← res × res
13          SI le j-ème chiffre de la décomposition en base 2 est 1 ALORS
14              DEBUT_SI
15              res ← a × res
16              FIN_SI
17          AFFICHER res
18          FIN_POUR
19   FIN_ALGORITHME
```

Pour calculer a^{11} nous n'avons donc plus à effectuer que 6 multiplications ou plutôt 5 si on ne compte pas la multiplication par 1.

(d) Avec Python :

puiss-horner.py{3}{19}

```
1  def base2(n):
2      r=n%2
3      q=n//2
4      R=[r]
5      while q>0 :
6          r=q%2
7          q=q//2
8          R=[r]+R
9      return R
10
11 def puissance_horner(a,n):
12     C=base2(n)
13     res=a
14     for j in range(1,len(C)):
15         res*=res
16         if C[j]==1: res*=a
17     return(res)
```

3. Variante sans utiliser l'écriture en base 2

(a) Algorithme :

Voyons comment procéder sur notre exemple habituel :

$a^{11} = a \times a^{10} = a \times \left(a^5\right)^2 = a \times \left(a \times a^4\right)^2 = a \times \left(a \times \left(a^2\right)^2\right)^2$

Ainsi, on réduit l'exposant k selon sa parité :

- si k est pair, on écrit $a^k = \left(a^{\frac{k}{2}}\right)^2$;
- sinon, $a^k = a \times \left(a^{\frac{k-1}{2}}\right)^2$

Exponentiation rapide sans utiliser la base 2

```
1   VARIABLES
2   a : float
3   n : int
4   ENTRÉES
5   un réel a et un entier n
6   INITIALISATION
7   res ← 1
8   puissance ← n
9   temp ← a
10   DEBUT_ALGORITHME
11      TANT_QUE puissance non nulle FAIRE
12         DEBUT_TANT_QUE
13         SI puissance est impaire ALORS
14            DEBUT_SI
15            res ← temp × res
16            puissance ← puissance − 1
17            FIN_SI
18         puissance ← puissance/2
19         temp ← temp × temp
20         FIN_TANT_QUE
21   FIN_ALGORITHME
```

(b) Avec *Python* :

puiss-rapid.py{3}{13}

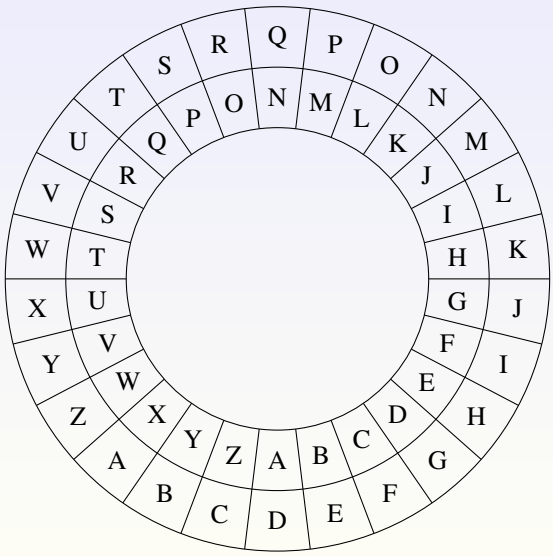
```
1  def expo_rapid(a,n):
2      res=1
3      Puissance=n
4      temp=a
5      while Puissance>0:
6          if Puissance%2==1 :
7              res*=temp
8              Puissance+=-1
9          Puissance=Puissance/2
10         temp*=temp
11     return(res)
```

2.4 Chiffrement de *César*

Jules César était un général, homme politique et écrivain romain, né à Rome le 12 (ou 13) juillet 100 avant *J.-C.* et mort le 15 mars 44 avant *J.-C.*.

Principe

Jules César, pour ses communications importantes, cryptait ses messages. Le chiffrement de *César* est un simple décalage de lettres. Par exemple, avec un décalage de 3, A devient D, B est remplacé par E, C par F, ...
On peut utiliser , pour crypter et décrypter, deux alphabets sur deux anneaux concentriques, qui peuvent tourner l'un par rapport à l'autre :



2.4.1 Cryptage

⇒ **Activité 2. 21**

Écrivez une fonction `cesar_chiffre_lettre(lettre, dec)` qui code une lettre `lettre` avec le décalage `dec`.
Déduisez-en ensuite une fonction `cesar_chiffre_texte(texte, k)` (qui fait appel à la première fonction), qui décale tout un texte.

Le programme :

2.4.2 Décryptage

⇒ **Activité 2. 22**

Écrivez une fonction `cesar_dechiffre_lettre(x, dec)` qui code une lettre `lettre` avec le décalage `dec`.

Le programme :

2.4.3 Attaque du chiffrement de *César*

2.4.3.1 Attaque par force brute

Nous pouvons effectuer une attaque par force brute du code de *César* car il n'y a que 26 possibilités de chiffrement.

⇒ **Activité 2. 23**

Écrivez une procédure `cesar_dechiffre_texte(texte)` qui déchiffre un texte avec toutes les combinaisons possibles.

Le programme :

2.4.3.2 Attaque par analyse fréquentielle

Nous pouvons également, si le texte est suffisamment long, effectuer une analyse fréquentielle, puis, en calculant l'indice de coïncidence mutuelle, en déduire la clé (valeur du décalage de *César*).

⇒ **Activité 2. 24**

Écrivez une fonction `recherche_clef(texte)` qui retourne la valeur du décalage de *César* d'un texte crypté par calcul de l'indice de coïncidence mutuel. En effet, il est maximal pour le bon décalage. Il suffit donc de le calculer pour chaque décalage et de retenir la bonne valeur.

Le programme :

Vous pouvez alors décrypter le message codé `message2_code` en utilisant :

```
cle = recherche_clef(message2_code)
cesar_chiffre_texte(message2_code,-cle)}
```

2.5 Chiffrement affine

Comme le code de *César* est très simple, il est relativement facile à décrypter. C'est la raison pour laquelle on complique un peu les choses avec le codage dit affine. Le principe est le suivant :

Codage affine

- On remplace les lettres par des nombres allant de 0 (pour le A) à 25 (pour le Z).
- On choisit f une fonction affine, donc telle que $f(x) = a\,x + b$, avec a et b entiers compris entre 1 et 25.

Chaque lettre A, B, ... est codée par son rang entre 0 et 25.
On choisit un couple de nombres a et b (on peut se limiter à l'intervalle $[0, 25]$ car on retrouve ensuite les mêmes résultats.
En notant x le rang d'une lettre $x \in [0, 25]$ et $r(x)$ le reste de la division euclidienne de $y = a\,x + b$ par 26, $r(x)$ est la lettre correspondante codée.

Exemple 1

Choisissons par exemple f telle que $f(x) = 3\,x + 2$.

- Pour crypter la lettre C (donc le nombre 2), par exemple, on calcule $f(2) = 3 \times 2 + 2 = 8$, ce qui correspond à la lettre I.
- Pour crypter la lettre U (donc le nombre 20), par exemple, on calcule $f(20) = 3 \times 20 + 2 = 62$. Comme le résultat est supérieur à 25, on doit faire la division euclidienne de 62 par 26, ce qui donne $62 = 2 \times 26 + 10$ et la lettre correspond au reste, ici 10. On trouve alors la lettre K.

Exemple 2

$a = 17$, $b = 2$.
La lettre A, de rang 0 est codée par le nombre $0 \times 17 + 2 = 2$, qui correspond à la lettre codée C. La lettre K, de rang 10 est codée par le nombre $10 \times 17 + 2 = 172 = 16 + 6 \times 26 \equiv 16 \pmod{26}$, qui correspond à la lettre codée Q.

⇒ **Activité 2.25**

1. Expliquez pourquoi le couple $(2, 0)$ ne convient pas pour effectuer un chiffrement affine.
2. Le couple $(3, 0)$ convient-il ?

Ainsi, si a n'est pas premier avec 26, deux lettres différentes de rang x et x' peuvent être chiffrées par la même lettre.
Dans ce cas, le déchiffrement est impossible.

Par contre, si a est premier avec 26, donc si $pgcd(a, 26) = 1$, deux lettres différentes de rang x et x' ne peuvent posséder le même codage. Le chiffrement est une bijection et le déchiffrement est possible.

En conséquence, **a doit être premier avec 26**.
Il y a ainsi pour le chiffrement affine 311 clés utilisables.

⇒ **Activité 2. 26**

Retrouver le nombre de clés utilisables pour le chiffrement affine.

2.5.1 Fonctions préliminaires

⇒ **Activité 2. 27**

Écrivez une fonction `code(texte)` qui, à partir d'une chaîne `texte`, retourne une liste dans laquelle chaque lettre du texte est remplacée par son numéro dans l'alphabet.

Le programme :

⇒ **Activité 2. 28**

Écrivez une fonction `encode(liste)` qui, à partir d'une liste `liste`, retourne une chaîne dans laquelle chaque numéro est remplacé par la lettre majuscule correspondante dans l'alphabet.

Le programme :

2.5.2 Cryptage

Nous allons maintenant crypter un message avec une fonction du type $y = f(x) = a\,x + b$.

⇒ **Activité 2. 29**

Écrivez une fonction `code_affine(texte, couple)` qui code un texte par fonction affine avec le couple `couple = (a, b)`. Nous pourrons accéder aux éléments a et b du tuple *couple* par :

```
a, b = couple[0], couple[1]
```

Le programme :

2.5.3 Décryptage

Pour déchiffrer un message, il faut procéder de la même façon. On commence par transcrire le message en nombres. Pour chaque nombre, on doit inverser la relation $y = a\,x + b$ (ici, on connaît y et on doit retrouver x). On a envie de poser $x = \frac{y}{a} - \frac{b}{a}$. C'est presque cela, sauf que l'on fait de l'arithmétique modulo 26. Ce qui remplace $\frac{1}{a}$, c'est l'inverse de a modulo 26, autrement dit un entier a' tel que, lorsqu'on fait le produit $a\,a'$, on trouve un entier de la forme $1 + 26\,k$. On sait qu'un tel entier existe si a et 26 sont premiers entre eux. Par exemple, pour $a = 3$, on peut choisir $a' = 9$ car $9 \times 3 = 1 + 26$.

Cette valeur de a' déterminée, on a alors $x = a'\,y - a'\,b$, qu'on retranscrit en une lettre comme pour l'algorithme de chiffrement.

Déchiffrement affine

Pour décoder le message, on peut appliquer le même principe avec un chiffrement affine dont la clé est le couple (a', b') tels que :

$$\begin{cases} a\,a' \equiv 1\ [26]\ :\ a' \text{ est l'inverse de } a \text{ modulo } 26 \\ b' \text{ est le reste de la division euclidienne de } a'\ (26 - b)\ \text{ par } 26 \end{cases}$$

Exemple 1

Reprenons le couple $(a, b) = (17, 2)$.
L'inverse de $a = 17$ modulo 26 est $a' = 23$.
De plus :

$$a'\,(26 - b) \equiv 6 \pmod{26}$$

qui donne $b' = 6$.
Ainsi, la lettre, codée S, correspondant au rang 18 correspond à la lettre originelle donnée par :

$$23 \times 18 + 6 \equiv 4 \pmod{26}$$

C'est la lettre de rang 4, donc E.

Exemple 2

Un message a été codé par un chiffrement affine. On note f la fonction de codage : $f(x)$ est le reste de la division euclidienne de $a\,x + b$ par 26. À l'aide d'une analyse fréquentielle, on a remarqué que la lettre E est codée par E et que la lettre T est codée par Z.

- E est codé par E donc $a \times 4 + b \equiv 4\ [26]$.
  T est codé par Z donc $a \times 19 + b \equiv 25\ [26]$.
  a et b vérifient donc le système :

$$\begin{cases} 4\,a + b \equiv 4\ [26] & (1) \\ 19\,a + b \equiv 25\ [26] & (2) \end{cases}$$

- En effectuant la soustraction (2) − (1) membre à membre, on en déduit :

$$15\,a \equiv 21\ [26]$$

- Pour déterminer l'inverse de 15 modulo 26, on écrit la relation de Bézout :

$$15 \times 7 - 26 \times 4 = 1$$

  Donc $15 \times 7 \equiv 1\ [26]$.
  De $15\,a \equiv 21\ [26]$, on tire :
  $7 \times 15\,a \equiv 7 \times 21\ [26]$, soit :
  $a \equiv 17\ [26]$.
  Donc $a = 17$ puisque $0 \leqslant a \leqslant 25$.
  On obtient alors :

$$\begin{cases} 68 + b \equiv 4\ [26] \\ 323 + b \equiv 25\ [26] \end{cases} \Longrightarrow b \equiv 14\ [26]$$

  Finalement, $(a, b) = (17, 14)$.
- Pour déterminer la fonction de décodage, on cherche tout d'abord l'inverse de 17 modulo 26.
  On trouve $17 \times (-3) + 26 \times 2 = 1$.
  On a $f(x) \equiv 17\,x + 14\ [26]$, donc :
  $-3\,f(x) \equiv x - 42\ [26]$, soit :
  $x \equiv -3\,f(x) + 42\ [26] \equiv 23\,f(x) + 16$.
  Pour décoder, il suffit de coder le message crypté à l'aide du codage affine $23\,x + 16$.

Exemple 3

On considère la phrase *LACLEESTSOUSLEPAILLASSON* que nous codons à l'aide du codage affine $y = 7\,x + 10 \pmod{26}$.
Nous obtenons alors *JKYJMMGNGEUGJMLKOJJKGGEX*.

- Pour trouver a' (inverse de a modulo 26), une des deux clés du décodage, on va utiliser l'algorithme de Bézout :

  $$7\,a' \equiv 1 \pmod{26}$$
  $$7\,a' = 1 + 26\,k$$

  Soit :

  $$-26\,k + 7\,a' = 1$$

  En posant $u = -k$ et $v = a'$, on obtient :

  $$26\,u + 7\,v = 1$$

- En utilisant l'algorithme de Bézout, on obtient :

  $$v = a' = -11 \equiv 15 \pmod{26}$$

- On obtient b' grâce à la relation :

  $$b' = -a'\,b = 11 \times 10 = 110 \equiv 6 \pmod{26}$$

- Ainsi, la transformation affine de décodage est :

  $$y = 15\,x + 6 \pmod{26}$$

⇒ **Activité 2. 30**

Le déchiffrement affine peut alors s'effectuer en créant une fonction `decode_affine(texte,couple)`. Celle-ci fait appel à la fonction `inverse(a,n)` définie dans les outils mathématiques ainsi qu'aux fonctions `code(texte)` et `decode(liste)`. À vous de jouer !

Le programme :

2.5.4 Casser un chiffrement affine

Supposons que nous ayons le message suivant codé au moyen d'une transformation affine. :
DIADSSOXAIATSSOEGODSNICDDIOOEVEGOEGODIBSVSXFS.
En français, la lettre la plus fréquente est le E suivi du S puis du A.
Avec un peu de chance, cet ordre fréquentiel va être suivi, à quelque chose près, par les lettres du texte à décoder : ceci sera d'autant plus vrai que le texte sera long et, également d'autant plus que le texte à décoder appartiendra à une famille analogue aux textes ayant servi à établir la table des fréquences. Ici, dans le court message que nous avons à décrypter, la lettre la plus fréquente est le S (18,75 %), suivi du O (14,6 %) :

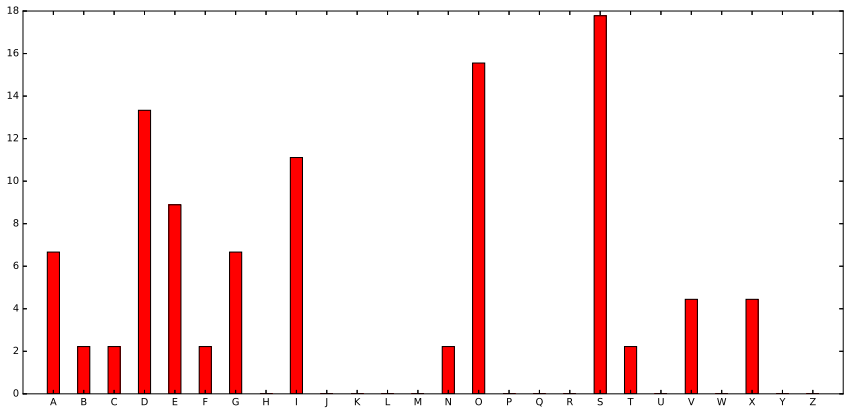


FIGURE 2.1 – affine1.eps

Faisons alors l'hypothèse que le S correspond au E et le O au S :
Passons aux équivalents numériques :

$$\begin{cases} S & \to 18 & \rightsquigarrow 4 & \to & \text{E} \\ O & \to 14 & \rightsquigarrow 18 & \to & \text{S} \end{cases}$$

Les paramètres de décryptage a' et b' doivent alors vérifier le système de deux équations :

$$\begin{cases} 18\,a' + b' & \equiv 4\ [26] & S \rightsquigarrow E & (3) \\ 14\,a' + b' & \equiv 18\ [26] & O \rightsquigarrow S & (4) \end{cases} \Longrightarrow 4\,a' \equiv -14\ [26] \equiv 12\ [26]$$

Les règles élémentaires de calcul sur des congruences de même module nous permettent d'opérer quasiment comme pour la résolution de systèmes d'équations classiques : nous ne pouvons faire de divisions, mais nous pouvons faire des multiplications.
On en déduit, avec (3) − (4) :

$$4\,a' \equiv -14 \pmod{26} \equiv 12 \pmod{26}$$

ou encore :

$$2\,a' \equiv -7 \pmod{13} \equiv 6 \pmod{13}$$

- Si a' vérifie cette équation, il existe $k \in \mathbb{Z}$ tel que :

$$2\,a' = -7 + 13\,k$$

- Multiplions par 7 qui est l'inverse de 2 modulo 13 :

$$\begin{aligned} 2\,a' \times 7 &= -7 \times 7 + 13 \times 7\,k \\ 14\,a' &\equiv -49 \pmod{13} \\ a' &\equiv -10 \pmod{13} \\ a' &\equiv 3 \pmod{13} \end{aligned}$$

Donc $a' \in \{3, 16, 29, \ldots\}$.
Modulo 26, cela nous donne $a' \equiv 3 \pmod{26}$ ou $a' \equiv 16 \pmod{26}$ (29 n'est pas dans l'intervalle).
a' doit être premier avec 26 pour être admissible, les deux options restent possibles.
De (3) ou (4), on déduit :
$b' = 4 - 18 \times 3 \equiv -50 \pmod{26} \equiv 2 \pmod{26}$ ou bien $b' = 18 - 16 \times 14 \equiv -206 \pmod{26} \equiv 2 \pmod{26}$.
Nous retrouvons ainsi la transformation affine de décodage avec :

```
print(code_affine(ch3_code,(3,2)))
```

et nous obtenons :
LACLEESTCACHEESOUSLEPAILLASSONOUSOUSLAFENETRE.
Nous vérifions l'hypothèse que le S correspond au E et le O au S grâce à l'histogramme du message originel :

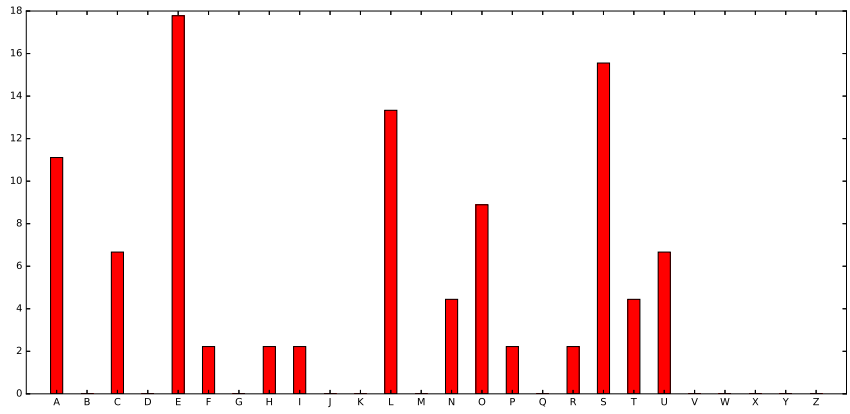


FIGURE 2.2 – affine2.eps

Cet exemple montre qu'une analyse statistique des fréquences de lettres permet facilement de briser un cryptogramme quand on sait que celui-ci est le fruit d'une transformation affine monoalphabétique.

Nous pourrions effectuer ces tâches avec *Python* mais une méthode de chiffrement plus intéressante nous attend.

2.6 Chiffrement de Vigenère

Même si l'on connaissait depuis fort longtemps les faiblesses de la cryptographie par substitution, et malgré les essais notamment d'*Alberti*, de *Porta* et de *Trithème*, il n'y eut pas entre *César* et le $XVI^{\text{è}}$ siècle de véritable nouveau procédé cryptographique, à la fois sûr (pour les moyens de l'époque) et facile à utiliser ! *Blaise de Vigenère* (1523-1596), fut l'initiateur d'une nouvelle façon de chiffrer les messages qui mit en échec les cryptanalystes trois siècles durant. *Vigenère* était un personnage à multiples facettes, tantôt alchimiste, écrivain, historien, il était aussi diplomate au service des ducs de Nevers et des rois de France. C'est en 1549 que *Vigenère* fut envoyé à Rome pour une affaire d'Etat. Durant son voyage, il se passionna pour les écrits de *Léone Batista Alberti*, *Jean Trithème* et *Giovanni Battista della Porta*, des cryptographes contemporains à son époque. Ses intérêts pour la cryptographie à cette époque étaient purement professionnels. Vers 1560, *Vigenère* se jugeant à l'abri du besoin, se consacra uniquement à la reprise des travaux d'*Alberti*, de *Trithème* et de *Porta*. C'est en 1586 qu'il publie son Traité des chiffres ou Secrètes manières d'écrire, qui explique son nouveau chiffre.
L'idée de *Vigenère* est d'utiliser un chiffre de *César*, mais où le décalage utilisé change de lettres en lettres.

Principe

Choisissons un mot "clef" qui va servir à coder le message, par exemple la clef *PREPATSI* pour coder le message *LA CRYPTOGRAPHIE NOUS EMBALLE*.

Lettre originelle	L	A	C	R	Y	P	T	O	G	R	A	P	H
Nombre originel	11	0	2	17	24	15	19	14	6	17	0	15	7
Clef	P	R	E	P	A	T	S	I	P	R	E	P	A
Décalage	15	17	4	15	0	19	18	8	15	17	4	15	0
Nombre codé	0	17	6	6	24	8	11	22	21	8	4	4	7
Lettre codée	A	R	G	G	Y	I	L	W	V	I	E	E	H

Lettre originelle	I	E	N	O	U	S	E	M	B	A	L	L	E
Nombre originel	8	4	13	14	20	18	4	12	1	0	11	11	4
Clef	T	S	I	P	R	E	P	A	T	S	I	P	R
Décalage	19	18	8	15	17	4	15	0	19	18	8	15	17
Nombre codé	1	22	21	3	11	22	19	12	20	18	19	0	21
Lettre codée	B	W	V	D	L	W	T	M	U	S	T	A	V

La clef est répétée autant de fois qu'il le faut en dessous du message.
Appliquons-lui ensuite un décalage "de *César*" en fonction de la lettre de la clef.
La première lettre L doit être codée avec P (p dans le carré de Vigenère). Appliquons-lui un décalage de 15, ce qui donne A.
La deuxième lettre A doit être codée avec R (r dans le carré de Vigenère). Appliquons-lui un décalage de 17, ce qui donne R.
La troisième lettre C doit être codée avec E (e dans le carré de Vigenère). Appliquons-lui un décalage de 4, ce qui donne G.
Le message codé est donc : *ARGG YILW VIEE HBWV DLWT MUST AV*.
On constate qu'une même lettre peut être codée de plusieurs façons, ce qui rend inefficace l'analyse des fréquences pour décrypter!
Pour coder-décoder, nous pouvons utiliser un carré de Vigenère (donné ci-dessous). Les trois premiers caractères y sont mis en valeur.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
b	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
c	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
d	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
e	E	F	G[3]	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
f	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
g	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
h	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
i	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
j	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
k	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
ℓ	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
m	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
n	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
o	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
p	P	Q	R	S	T	U	V	W	X	Y	Z	A[1]	B	C	D	E	F	G	H	I	J	K	L	M	N	O
q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
r	R[2]	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
s	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
t	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
u	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
v	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
w	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
x	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

TABLE 2.1 – Carré de Vigenère

2.6.1 Cryptage

Le cryptage par la méthode de Vigenère est très simple.

⇒ **Activité 2.31**
Écrivez une fonction `codage_vig1(message, clef)` qui chiffre une chaîne `message` à l'aide de la clé `clef` par la méthode de Vigenère.

Le programme :

1. Première lettre du message
2. Deuxième lettre
3. Troisième lettre

2.6.2 Décryptage

2.6.2.1 Decryptage avec la clé

Le décryptage n'est pas non plus très compliqué si la clé est connue.

⇒ **Activité 2. 32**

Écrivez une fonction `decodage_vig1(message_code, clef)` qui déchiffre une chaîne `message_code` à l'aide de la clé `clef` par la méthode de Vigenère.

Le programme :

2.6.2.2 Détermination de la clé

Nous voyons bien que l'histogramme n'a plus rien à voir avec celui d'une substitution simple : il est beaucoup plus "plat" :

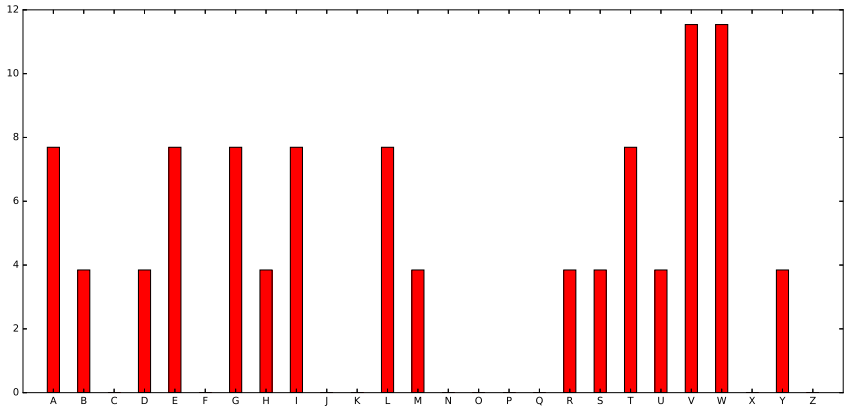


FIGURE 2.3 – vigen1.eps

Ce chiffre, qui a résisté trois siècles aux cryptanalystes, est pourtant relativement facile à casser, grâce à une méthode mise au point indépendamment par Babagge et Kasiski. Une autre méthode complètement différente a été encore mise au point plus tard par le commandant Bazeries.
Comment déchiffrer Vigenère sans connaître la clé ?
Les techniques les plus courantes utilisent des méthodes statistiques qui permettent de retrouver la longueur de la clé, puis une analyse des fréquences permet de retrouver la clé.

Test de Kasiski

Le test de Kasiski consiste à repérer des répétitions de lettres dans le texte chiffré.
ABC apparaît trois fois dans le message *ABCXYZABCKLMNOPQRSABC*.
Le fait qu'il existe une répétition signifie soit qu'une même suite de lettres du texte clair a été chiffrée avec une même partie de la clef, soit que différentes suites de lettres du texte en clair se chiffrent de la même façon. Cette seconde possibilité n'a qu'une faible probabilité d'arriver.
En analysant les écarts entre deux redondances de séquences identiques, nous pouvons déterminer un multiple de la longueur de la clé. En analysant pour chaque écart, les diviseurs possibles, nous pouvons déduire avec une grande probabilité la longueur de la clé.
Les positions de *ABC* sont 0, 6 et 18, les écarts sont donc de 6, 12 et 18, les diviseurs les plus courants de ces nombres sont 2, 3 et 6, la clé a donc une forte probabilité d'être de longueur 2, 3 ou 6.

Test de l'indice de coïncidence

Le test de l'indice de coïncidence consiste à prendre une lettre sur n dans le message, et de mesurer l'indice de coïncidence. Plus il est élevé, plus la probabilité que n soit la longueur de la clé est grande.
En effet, prendre une lettre sur n, lorsque n est la longueur de la clé, revient à prendre une série de lettres chiffrées toujours chiffrées avec le même décalage, l'indice de coïncidence est donc égal à celui du texte clair.

⇒ **Activité 2. 33**

Écrivez une fonction `texte_decale(texte, dec)` (quasiment identique à `cesar_chiffre_texte(texte, k)`) qui décale une chaîne `texte` avec un décalage de `dec`, entier relatif.

Le programme :

⇒ **Activité 2. 34**

Écrivez une fonction `indice_corr_max(texte)` qui détermine l'indice de coïncidence maximum pour la langue française.

Le programme :

Voici par exemple ci-dessous le contenu du fichier `poeme2_code.txt` chiffré par la méthode de Vigenère.

EVMC	DKWL	PSSG	DNFM	RRKT	AOWK	JEIE	OKLM	DLZT	RMWX	TZRS
RXWV	HLMI	EJMM	AHYT	CAGA	TUIY	OEAY	JVPF	UXUP	DJIS	ELAU
ECIF	UXDY	JVGW	OLWL	TSIP	UJMM	AHYT	CAGA	TUYI	IEWX	DLVA
OBKM	PLTA	AVWZ	TEWJ	IMWT	PKSX	LXUW	CKVT	UGSZ	QIIS	AGKC
CAEG	DBFL	PEWJ	NUGQ	HFYS	AGKC	CVJD	RXLA	TTER	HXJL	TIVX
EKWT	PIFG	ELSV	HIMT	NWAZ	TJEC	SUGC	VVVE	AKXW	XJPD	ILWI
JRVG	IOWD	XKIB	ABKQ	AGIJ	TTMA	HZFX	EGEM	IKVT	DXDW	CXYT
STFV	TVWP	VTFB	SVWT	DXUQ	SVVC	EISA	HVHT	CHMZ	PXIG	AMLM
CUVT	AMLM	CUVT	SBDT	TWEJ	TIWV	SRRI	DXKI	CEIT	SESD	XKIH
SXGC	ARPT	NMWC	GUIA	AKJQ	KVIS	EEGQ	HVEJ	NTQI	CKEJ	CNFZ
PGTD	RMSD	TTPP	RXMA	HZXT	DNLI	QCIP	UJMI	CUPD	ILWI	JRVG
IOWA	XCEG	RBNM	DSWT	ROWZ	AVTA	ULHZ	DWSC	DLAT	TEGT	AMLM
CUVT	QNWT	DZWT	ANWV	IIIS	AGKT	PTEV	EXLY	JRRS	IEWA	IVRI
RXXM	GDIG	DHMK	TDIC	TESX	DIXT	AOWK	AVTX	NVWI	JGYX	SXXN
PTIG	UGSC	CCIH	BTJZ	TRYM	EMSG	PEXH	OBFL	TEII	ONUP	TIEJ
CNFM	SVWE	LNEM	HUIA	OBKM	PLJP	IKWM	CJYX	TXDM	EFVI	RTAB
SVPP	RUJM	TEGW	OBKQ	HJEC	TESX	ALWQ	EEDM	SVWT	SUJI	CTLT
SIGC	GCSX	SXSC	EVMC	DKWI	JJWX	LXNM	GKJT	UBDT	PXIT	TESN
GRMR	HXMZ	SLZT	NMDI	EFYH	SBWZ	TUYH	OEWQ	AVXA	EUJC	XKHT
SUWB	TJHT	LAWZ	QVHP	NLDI	RYEA	ENJL	TCII	EXLX	JZWP	TMWV
SIIF	UXDW	XJIP	ULWL	TTMS	ETUP	PEXT	RLAT	DZWT	ANFM	RYEC
TXHI	HTIH	TFSC	KRMH	SBYV	TDEX	SLAT	RYEC	TXUM	HKFD	NLAO
CVWX	GGWY	JVZD	ULHW	JMIO	SBYV	TIEA	OKKD	DLWP	RKSK	WVDI
ONLL	DLGT	MXFB	JEIS	ELHT	JDIH	DXDW	XJIP	UXLD	DLWT	CKAD
TQZD	TKWV	DDHP	NLMV	RFMC	DNLI	QCIP	UCSK	FLIH	PKWD	TIX

Pour déchiffrer un message codé par la méthode de Vigenère, il est nécessaire de trouver la longueur de la clé utilisée pour le codage.

Pour cela, nous recherchons des motifs dans le texte codé, c'est-à-dire des suites de caractères identiques apparaissant plusieurs fois dans le message. La longueur de la clé a de bonnes chances d'être un diviseur du nombre de lettres de la chaîne située entre le début du premier motif et celui du second motif. Par exemple, il y a 56 caractères entre les deux occurrences du motif `JMM AHYT CAGA TU` dans le message codé, $54 \times 4 = 216$ caractères entre les deux occurrences `PD ILWI JRVG IOW` et 24 caractères entre les deux occurrences `UGC`. Il est probable que la clé soit de longueur égale à $pgcd\ (56, 216, 24)$, soit 8.

Il y a 8 caractères entre les deux occurrences du motif `AMLM CUVT` dans le message codé, ce qui semble confirmer la longueur de la clé.

Il faut donc chercher dans le texte crypté toutes les séquences de 3 ou 4 caractères (ou plus) qui se répètent deux fois. La distance entre ces séquences est probablement (mais ce n'est pas sûr) un multiple de la taille de la clé.

Vous pourrez construire une liste des distances entre deux chaînes répétées. Ensuite à l'aide d'une boucle, vous balaierez les longueur de clé (entre 3 et 10 par exemple) en testant si la distance est un multiple de la longueur de la clé. Vous choisirez la longueur qui a le meilleur score.

⇒ **Activité 2. 35**

Écrivez une fonction `longueur_clef(texte)` qui détermine la longueur de la clé d'une chaîne `texte`. Vous pourrez utiliser la méthode `find()` dont le comportement est illustré ci-dessous :

```
>>> chain = "abcjkhgsjhgabcjjdjkabc"
>>> chain.find("abc")
0
>>> chain.find("jkh")
3
```

Le programme :

Lorsque la longueur n de la clé est connue, il faut trouver le mot clé. On commence par découper le texte T en n sous textes : T_0, T_1, ..., T_{n-1}.

$$T_i = T[i], T[i+n], T[i+2n], \ldots$$

La particularité des textes T_i est qu'ils sont codés avec le même alphabet (qui correspond a un décalage de k_i de l'alphabet). Trouvez la clé est équivalent à trouver les décalages k_0, k_1, ..., k_{n-1}.

- Une première méthode (simple) consiste a calculer pour chacun des textes la lettre la plus fréquente (on suppose alors que c'est une lettre E et nous en déduisons le décalage correspondant). Un défaut de cette méthode est que si le texte est trop court ou si la clé est trop longue, on ne peut pas assurer que cela soit toujours la lettre E qui soit la lettre la plus fréquente.

- Nous allons utiliser une deuxième méthode :
  Pour cela nous allons calculer successivement :

  $$d_1 = k_1 - k_0, d_2 = k_2 - k_0, \ldots, d_{n-1} = k_{n-1} - k_0$$

  Attention, ce point est un peu délicat à comprendre : Pour calculer d_i, nous allons boucler sur tous les décalages possible, de 0 à 25. La partie d'algorithme peut alors être :

Décalage

```
DEBUT_ALGORITHME
   indice ← 0
   decalage ← 0
   POUR d ALLANT_DE 0 A 25
      DEBUT_POUR
      t ← concaténation de T_0 avec T_i auquel on a appliqué le décalage d
      Calcul de l'indice de coïncidence du texte t
      SI I_C(t) > indice ALORS
         DEBUT_SI
           indice ← I_C(t)
           decalage ← d
           FIN_SI
        FIN_POUR
     di ← decalage
FIN_ALGORITHME
```

En effet quand nous trouvons le bon décalage alors cela veut dire que les deux textes T_0 et T_i sont codés avec exactement le même alphabet et correspondent donc a du français : l'indice de coïncidence est alors maximal.
Il ne reste plus maintenant qu'à fixer l'origine pour cela nous concaténons tous les textes T_i, chacun avec son décalage d_i et nous cherchons la lettre la plus fréquente qui va correspondre alors à la lettre E, ce qui nous permet de fixer le décalage global.

⇒ **Activité 2.36**

Écrivez une fonction `recherche_clef(texte, long_clef)` qui retourne la clé d'une chaîne `texte` grâce à la longueur `long_clef`.
Vous pourrez utiliser un dictionnaire ainsi que la méthode `keys()` dont le comportement est illustré dans le programme ci-dessous :

dico-01.py

```
# -*- coding: utf-8 -*-

dico = {}
dico["a"] = 0
dico["b"] = 1
dico["c"] = 2
dico["d"] = 3
dico["début"] = 6
print("dico 1 : ", dico)
dico["a"] = dico["a"] + 11
print("dico 2 : ",dico)
maxi = 0
for car in dico.keys():
    if dico[car] > maxi:
        print("début : ", maxi)
        maxi = dico[car]
        print("fin : ", maxi)
print("finalement : ", maxi)
```

dont le résultat est :

```
dico 1 :  {'b': 1, 'début': 6, 'c': 2, 'd': 3, 'a': 0}
dico 2 :  {'b': 1, 'début': 6, 'c': 2, 'd': 3, 'a': 11}
début :  0
fin :  1
début :  1
fin :  6
début :  6
fin :  11
finalement :  11
```

La fonction `recherche_clef(texte, long_clef)` :

2.6.2.3 Decryptage sans la clé

⇒ **Activité 2. 37**

Il reste ensuite à casser le message grâce à la procédure `casse_texte(texte)`, à écrire bien sûr... Celle-ci fait appel bien entendu à `longueur_clef(texte)` et `recherche_clef(texte, long_clef)`.

La procédure `casse_texte(texte)` :

2.7 Chiffrement RSA

Pour crypter un message, nous commençons par le transformer en un – ou plusieurs – nombres. Les processus de chiffrement et déchiffrement font appel à plusieurs notions :

- On choisit deux **nombres premiers** p et q que l'on garde secrets et on pose $n = p \times q$. Le principe étant que même connaissant n il est très difficile de retrouver p et q (qui sont des nombres ayant des centaines de chiffres).
- La clé secrète et la clé publique se calculent à l'aide de l'**algorithme d'Euclide** et des **cœfficients de Bézout**.
- Les calculs de cryptage se feront **modulo** n.
- Le déchiffrement fonctionne grâce à une variante du **petit théorème de Fermat**.

Dans cette section, c'est Célestin qui veut envoyer un message secret à Sidonie. La processus se décompose ainsi :

1. Sidonie prépare une clé publique et une clé privée,
2. Célestin utilise la clé publique de Sidonie pour crypter son message,
3. Sidonie reçoit le message crypté et le déchiffre grâce à sa clé privée.

2.7.1 Calcul de la clé publique et de la clé privée

2.7.1.1 Choix de deux nombres premiers

Sidonie effectue, une fois pour toute, les opérations suivantes (en secret) :

- elle choisit deux nombres premiers distincts p et q (dans la pratique ce sont de très grand nombres, jusqu'à des centaines de chiffres),
- Elle calcule $n = p \times q$,
- Elle calcule $\varphi(n) = (p-1) \times (q-1)$.

Exemple 1

- $p = 5$ et $q = 17$
- $n = p \times q = 85$
- $\varphi(n) = (p-1) \times (q-1) = 64$

Vous noterez que le calcul de $\varphi(n)$ n'est possible que si la décomposition de n sous la forme $p \times q$ est connue. D'où le caractère secret de $\varphi(n)$ même si n est connu de tous.

Exemple 2

- $p = 101$ et $q = 103$
- $n = p \times q = 10\,403$
- $\varphi(n) = (p-1) \times (q-1) = 10\,200$

2.7.1.2 Choix d'un exposant et calcul de son inverse

Sidonie continue :

- elle choisit un exposant e tel que $\text{pgcd}(e, \varphi(n)) = 1$,
- elle calcule l'inverse d de e module $\varphi(n)$: $d \times e \equiv 1 \pmod{\varphi(n)}$. Ce calcul se fait par l'algorithme d'Euclide étendu.

Exemple 1

- Sidonie choisit par exemple $e = 5$ et on a bien $\text{pgcd}(e, \varphi(n)) = \text{pgcd}(5, 64) = 1$,
- Sidonie applique l'algorithme d'Euclide étendu pour calculer les cœfficients de Bézout correspondant à $\text{pgcd}(e, \varphi(n)) = 1$. Elle trouve $5 \times 13 + 64 \times (-1) = 1$. Donc $5 \times 13 \equiv 1 \pmod{64}$ et l'inverse de e modulo $\varphi(n)$ est $d = 13$.

Exemple 2

- Sidonie choisit par exemple $e = 7$ et on a bien $\text{pgcd}(e, \varphi(n)) = \text{pgcd}(7, 10\,200) = 1$,
- L'algorithme d'Euclide étendu pour $\text{pgcd}(e, \varphi(n)) = 1$ donne $7 \times (-1457) + 10 \times 200 \times 1 = 1$. Mais $-1457 \equiv 8743 \pmod{\varphi(n)}$, donc pour $d = 8743$, on a $d \times e \equiv 1 \pmod{\varphi(n)}$.

2.7.1.3 Clé publique

La **clé publique** de Sidonie est constituée des deux nombres : n et e.

Et comme son nom l'indique, Sidonie communique sa clé publique au monde entier.

Exemple 1

$n = 85$ et $e = 5$.

Exemple 2

$n = 10\,403$ et $e = 7$.

2.7.1.4 Clé privée

Sidonie garde pour elle sa **clé privée** : d

Sidonie détruit en secret p, q et $\varphi(n)$ qui ne sont plus utiles. Elle conserve secrètement sa clé privée.

Exemple 1

$d = 13$

Exemple 2

$d = 8743$

2.7.2 Chiffrement du message

Célestin veut envoyer un message secret à Sidonie. Il se débrouille pour que son message soit un entier (quitte à découper son texte en bloc et à transformer chaque bloc en un entier).

2.7.2.1 Message

Le message est un entier m, tel que $0 \leq m < n$.

Exemple 1

Célestin veut envoyer le message $m = 10$.

Exemple 2

Célestin veut envoyer le message $m = 1234$.

2.7.2.2 Message chiffré

Célestin récupère la clé publique de Sidonieé : n et e avec laquelle il calcule, à l'aide de l'algorithme d'exponentiation rapide, le message chiffré :

$$x \equiv m^e \pmod{n}$$

Il transmet ce message x à Sidonie.

Exemple 1

$m = 10$, $n = 85$ et $e = 5$ donc

$$x \equiv m^e \pmod{n} \equiv 10^5 \pmod{85}$$

On peut ici faire les calculs à la main :

$$10^2 \equiv 100 \equiv 15 \pmod{85}$$
$$10^4 \equiv (10^2)^2 \equiv 15^2 \equiv 225 \equiv 55 \pmod{85}$$
$$x \equiv 10^5 \equiv 10^4 \times 10 \equiv 55 \times 10 \equiv 550 \equiv 40 \pmod{85}$$

Le message chiffré est donc $x = 40$.

Exemple 2

$m = 1234$, $n = 10\,403$ et $e = 7$ donc :

$$x \equiv m^e \pmod{n} \equiv 1234^7 \pmod{10\,403}$$

On utilise l'ordinateur pour obtenir que $x = 10\,378$.

2.7.3 Déchiffrement du message

Sidonie reçoit le message x chiffré par Célestin, elle le décrypte à l'aide de sa clé privée d, par l'opération : $m \equiv x^d \pmod{n}$ qui utilise également l'algorithme d'exponentiation rapide.

Nous allons prouver dans le lemme 2.7.5, que par cette opération Sidonie retrouve bien le message original m de Célestin.

Exemple 1

$c = 40$, $d = 13$, $n = 85$ donc

$$x^d \equiv (40)^{13} \pmod{85}$$

Calculons à la main $40^{13} \equiv \pmod{85}$ on note que $13 = 8 + 4 + 1$, donc $40^{13} = 40^8 \times 40^4 \times 40$.

$$40^2 \equiv 1600 \equiv 70 \pmod{85}$$
$$40^4 \equiv (40^2)^2 \equiv 70^2 \equiv 4900 \equiv 55 \pmod{85}$$
$$40^8 \equiv (40^4)^2 \equiv 55^2 \equiv 3025 \equiv 50 \pmod{85}$$

Donc

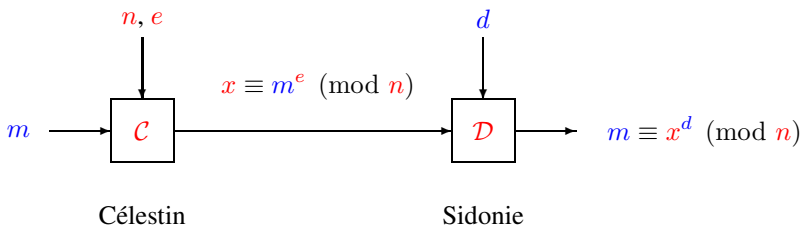
$$x^d \equiv 40^{13} \equiv 40^8 \times 40^4 \times 40 \equiv 50 \times 55 \times 40 \equiv 10 \pmod{85}$$

qui est bien le message m de Célestin.

Exemple 2

$c = 10\,378$, $d = 8743$, $n = 10\,403$. On calcule par ordinateur $x^d \equiv (10\,378)^{8743} \pmod{10\,403}$ qui vaut exactement le message original de Célestin $m = 1234$.

2.7.4 Schéma



Clés de Sidonie :

- publique : n, e
- privéeé : d

2.7.5 Lemme de déchiffrement

Le principe de déchiffrement repose sur le petit théorème de Fermat amélioré.

Lemme

Soit d l'inverse de e modulo $\varphi(n)$. Si $x \equiv m^e \pmod{n}$ alors $m \equiv x^d \pmod{n}$.

Ce lemme prouve bien que le message original m de Célestin, chiffré par clé publique de Sidonie (e,n) en le message x, peut-être retrouvé par Sidonie à l'aide de sa clé secrète d.

Démonstration

- Que d soit l'inverse de e modulo $\varphi(n)$ signifie $d\cdot e \equiv 1 \pmod{\varphi(n)}$. Autrement dit, il existe $k\in\mathbb{Z}$ tel que $d\cdot e = 1 + k\,\varphi(n)$.
- On rappelle que par le petit théorème de Fermat généralisé : lorsque m et n sont premiers entre eux
$$m^{\varphi(n)} \equiv m^{(p-1)(q-1)} \equiv 1 \pmod{n}$$
- **Premier cas** $\mathrm{pgcd(m,n)} = 1$.
Notons $c \equiv m^e \pmod{n}$ et calculons x^d :
$$x^d \equiv (m^e)^d \equiv m^{e\,d} \equiv m^{1+k\,\varphi(n)} \equiv m\,m^{k\,\varphi(n)} \equiv m\,(m^{\varphi(n)})^k \equiv m\,(1)^k \equiv m \pmod{n}$$
- **Deuxième cas** $\mathrm{pgcd(m,n)} \neq 1$.
Comme n est le produit des deux nombres premiers p et q et que m est strictement plus petit que n alors si m et n ne sont pas premiers entre eux cela implique que p divise m ou bien q divise m (mais pas les deux en même temps). Faisons l'hypothèse $\mathrm{pgcd(m,n)} = \mathrm{p}$ et $\mathrm{pgcd(m,q)} = 1$, le cas $\mathrm{pgcd(m,n)} = \mathrm{q}$ et $\mathrm{pgcd(m,p)} = 1$ se traiterait de la même manière.
Étudions $(m^e)^d$ à la fois modulo p et modulo q à l'image de ce que nous avons fait dans la preuve du théorème de Fermat amélioré.
  - modulo p : $m \equiv 0 \pmod{p}$ et $(m^e)^d \equiv 0 \pmod{p}$ donc $(m^e)^d \equiv m \pmod{p}$,
  - modulo q : $(m^e)^d \equiv m\times(m^{\varphi(n)})^k \equiv m\times(m^{q-1})^{(p-1)k} \equiv m \pmod{q}$.

  Comme p et q sont deux nombres premiers distincts, ils sont premiers entre eux et on peut écrire comme dans la preuve du petit théorème de Fermat amélioré que
$$(m^e)^d \equiv m \pmod{n}$$

2.7.6 Algorithmes

La mise en œuvre est maintenant très simple. Sidonie choisit deux nombres premiers p et q et un exposant e.

Voici le calcul de la clé secrète :

rsa-tex1.py

```
1  def cle_privee(p,q,e) :
2      n = p * q
3      phi = (p-1)*(q-1)
4      c,d,dd = euclide_etendu(e,phi)          # Pgcd et coeff de Bézout
5      return(d % phi)                         # Bon représentant
```

Le chiffrement d'un message m est possible par tout le monde, connaissant la clé publique (n,e).

rsa-tex2.py

```
1  def codage_rsa(m,n,e):
2      return pow(m,e,n)
```

Seule Sidonie peut déchiffrer le message crypté x, à l'aide de sa clé privée d.

rsa-tex3.py

```
1 def decodage_rsa(x,n,d):
2     return pow(x,d,n)
```

Lettre	Français	Anglais	Allemand	Espagnol	Portugais	Italien
a	7.636	8.167	6.516	11.525	14.634	11.745
b	0.901	1.492	1.886	2.215	1.043	0.927
c	3.260	2.782	2.732	4.019	3.882	4.501
d	3.669	4.253	5.076	5.010	4.992	3.736
e	14.715	12.702	16.396	12.181	12.570	11.792
f	1.066	2.228	1.656	0.692	1.023	1.153
g	0.866	2.015	3.009	1.768	1.303	1.644
h	0.737	6.094	4.577	0.703	0.781	0.636
i	7.529	6.966	6.550	6.247	6.186	10.143
j	0.613	0.153	0.268	0.493	0.397	0.011
k	0.049	0.772	1.417	0.011	0.015	0.009
l	5.456	4.025	3.437	4.967	2.779	6.510
m	2.968	2.406	2.534	3.157	4.738	2.512
n	7.095	6.749	9.776	6.712	4.446	6.883
o	5.796	7.507	2.594	8.683	9.735	9.832
p	2.521	1.929	0.670	2.510	2.523	3.056
q	1.362	0.095	0.018	0.877	1.204	0.505
r	6.693	5.987	7.000	6.871	6.530	6.367
s	7.948	6.327	7.270	7.977	6.805	4.981
t	7.244	9.056	6.154	4.632	4.336	5.623
u	6.311	2.758	4.166	2.927	3.639	3.011
v	1.838	0.978	0.846	1.138	1.575	2.097
w	0.074	2.360	1.921	0.017	0.037	0.033
x	0.427	0.150	0.034	0.215	0.253	0.003
y	0.128	1.974	0.039	1.008	0.006	0.020
z	0.326	0.074	1.134	0.467	0.470	1.181
à	0.486	0	0	0	0.072	0.635
â	0.051	0	0	0	0.562	0
á	0	0	0	0.502	0.118	0
ä	0	0	0.578	0	0	0
ã	0	0	0	0	0.733	0
œ	0.018	0	0	0	0	0
ç	0.085	0	0	0	0.530	0
è	0.271	0	0	0	0	0.263
é	1.504	0	0	0.433	0.337	0
ê	0.218	0	0	0	0.450	0
ë	0.008	0	0	0	0	0
î	0.045	0	0	0	0	0
ï	0.005	0	0	0	0	0
í	0	0	0	0.725	0.132	0
ì	0	0	0	0	0	0.030
ñ	0	0	0	0.31	0	0
ß	0	0	0.307	0	0	0
ô	0.023	0	0	0	0.635	0
ò	0	0	0	0	0	0.002
ó	0	0	0	0.827	0.296	0
ö	0	0	0.443	0	0	0
ú	0	0	0	0.168	0.207	0
ù	0.058	0	0	0	0	0.166
ü	0	0	0.995	0.012	0.026	0

TABLE 2.2 – Fréquences en pourcentages (https://en.wikipedia.org/wiki/Letter_frequency)

3

Bases de données

Ce chapitre constitue un rappel de première année et ne peut bien évidemment remplacer le cours...
Il est très largement inspiré du cours disponible ici : http://michel.stainer.pagesperso-orange.fr/

3.1 Introduction

Les données constituant une base de données sont organisées en tables. Une table contient des enregistrements composés de différents champs appelés également attributs. Le type associé au champ définit le domaine dans lequel le champ pourra prendre ses valeurs : par exemple entier, flottant, chaîne, date, ...
Les chaînes s'écrivent entre 'simples' ou "doubles" quotes (apostrophes ou guillemets).
Chaque table est désignée de manière unique par un identificateur (son nom) au sein de la base de données, de même que chaque champ au sein d'une table.
On peut représenter une table par un tableau dont les colonnes correspondent aux champs et les lignes aux enregistrements.
Une clé primaire d'une table est un champ (ou un ensemble de champs) qui identifie de manière unique chaque enregistrement dans la table. Chaque table devrait avoir une clé primaire.
Une clé étrangère dans une table est un champ (ou un ensemble de champs) qui fait référence à un champ (ou un ensemble de champs) d'une autre table (généralement la clé primaire).
Les requêtes SQL permettent d'interroger la base de données.
Pour ce faire, on peut utiliser des logiciels comme *SQLiteStudio* ou *Sqliteman*.

3.2 Création d'une base de données à partir de fichiers .csv

La base de données utilisée plus loin a été créée à partir de fichiers `.csv` à l'aide des commandes suivantes qui permettent d'importer des tables (dans le terminal Linux) : ici on importe dans la base `bdd_geo.db` les trois tables `communes`, `departements` et `regions` à partir des fichiers `villes_france_simple.csv`, `depts.csv` et `regs.csv`.

```
ordi@ordi-All-Series:~:$ cd repertoire
ordi@ordi-All-Series:~:repertoire$ sqlite3 bdd_geo.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import villes_france_simple.csv communes
sqlite> .import depts.csv departements
sqlite> .import regs.csv regions
sqlite> .exit
ordi@ordi-All-Series:/repertoire$
```

3.3 Connexion à base via *Python*

On peut se connecter via *Python* à une base de données via le module *sqlite3*.
L'exécution d'une requête renvoie alors un itérateur de type `Cursor`, que l'on peut utiliser directement à l'aide d'une boucle `for`, ce qui permet de parcourir les lignes du résultat de la requête.
Ces lignes se présentent sous forme de tuples. On peut aussi convertir le résultat en "liste de lignes" grâce à la méthode `fetchall()`. On peut enfin obtenir une ligne après l'autre avec la méthode `fetchone()` C'est ceete dernière méthode que l'on utilisera lorsque la requête n'attend qu'un seul résultat.

⚠ Ces trois procédés détruisent les lignes utilisées : si on souhaite les utiliser plusieurs fois, il est nécessaire de les stocker dans une liste par une instruction du type `result = cursor.fetchall()`.

```
import sqlite3

db_loc = sqlite3.connect('bdd_geo.db')
cursor = db_loc.cursor()

cursor.execute('''SELECT ... FROM ...''')
exemple1 = cursor.fetchone() # première ligne de la recherche
print(exemple1)

cursor.execute('''SELECT ... FROM ...''')
result = cursor.fetchall() # pour sélectionner tous les élements
print(result)
for el in result:
    print(el)
```

3.4 Requêtes

3.4.1 Définitions

1. **Valeur** `NULL`
   Un champ qui n'est pas renseigné, donc vide, contient la valeur `NULL`. Cette valeur est différente de zéro et représente l'absence de valeur.
2. **Expressions**
   Les expressions valides mettent en jeu des noms de champs, le mot clé "*" (qui signifie "tous les champs"), des constantes, des fonctions et des opérateurs classiques. Il existe des fonctions logiques, mathématiques, de manipulation de chaînes, de dates et des fonctions d'agrégation. Les fonctions d'agrégation permettent de calculer un résultat numérique (un entier ou un flottant) à partir d'un ensemble de valeurs. Les fonctions d'agrégation sont au nombre de cinq : `COUNT`, `SUM`, `MIN`, `MAX` et `AVG`.

3.4.2 Interrogation d'une base

Dans les descriptions qui suivent, les expressions entre crochets sont facultatives.
Lorsque plusieurs opérateurs sont utilisables à un endroit donné, ils sont séparés par des barres verticales (il faut en mettre un seul dans la requête).
La syntaxe classique est la suivante :

```
SELECT ... FROM ... [WHERE ...] [GROUP BY ... [HAVING ...]] [ORDER BY ...] [LIMIT
    ... [OFFSET ...]]
```

1. `SELECT ...`
   La commande

```
SELECT [DISTINCT] expr1 [[AS] alias1], expr2 [[AS] alias2], ...
```

réalise une projection.
`expr1`, `expr2`, ... indiquent quelles expressions devront être renvoyées, par exemple des noms de champs.
S'il y a une ambiguïté (cas de deux tables contenant des champs de même nom), il est nécessaire de préfixer le nom du champ par celui de la table suivi d'un point.
Le mot clé `AS` permet le renommage : ainsi, `alias1`, `alias2`, ... sont des alias qui fournissent un identificateur abrégé pour chaque champ concerné, pour la suite de la requête. En outre, un tel alias est utilisé comme titre de la colonne correspondante dans le résultat de la requête (le mot clé `AS` est facultatif mais il améliore grandement la lisibilité).
Le mot clé `DISTINCT` permet de supprimer les doublons.

2. `FROM ...`
   La commande

```
FROM table1 [[AS] alias1], table2 [[AS] alias2], ...
```

   donne la liste des tables participant à l'interrogation. `alias1`, `alias2`, ... sont des alias attribués aux tables pour le temps de la requête. Quand une table se voit attribuer un alias, elle n'est alors plus reconnue sous son nom d'origine dans la requête.

3. `WHERE ...`
   La commande `WHERE` permet d'effectuer une restriction dans la recherche effectuée dans une table ou un produit cartésien de tables.

   (a) Prédicats simples
   Un prédicat simple est la comparaison de plusieurs expressions au moyen d'un opérateur logique `opérateur_logique` :

```
WHERE expr1 opérateur\_logique expr2
```

   pour les opérateurs classiques.

```
WHERE expr1 [NOT] LIKE expr2
```

   où `expr2` est un masque, une chaîne de caractères contenant au moins un joker : _ pour un unique caractère ou % pour un nombre quelconque de caractères. Un masque comme '%7%' permet de filtrer les chaînes, mais aussi les nombres contenant un 7.

```
WHERE expr1 IS [NOT] NULL
```

   La valeur spéciale NULL est adaptée pour un champ vide.

```
WHERE expr1 [NOT] IN (expr2,expr3, ...
```

   teste l'appartenance à la liste d'expressions.

```
WHERE [NOT] EXISTS (expr1)
```

   teste si la sous-requête `expr1` renvoie au moins un résultat.

   (a) Prédicats composés
   Les opérateurs logiques `AND` et `OR` permettent de combiner plusieurs prédicats. `AND` est prioritaire par rapport à `OR` mais l'utilisation de parenthèses permet de modifier l'ordre d'évaluation.

(b) Sous-requêtes
Le critère de recherche employé dans une clause `WHERE` (l'expression à droite d'un opérateur de comparaison) peut être le résultat d'un `SELECT`.
Dans le cas des opérateurs classiques, la sous—interrogation ne doit ramener qu'une ligne et une colonne. Elle peut ramener plusieurs lignes à la suite des opérateurs `[NOT] IN`, `[NOT] EXISTS`, ou encore à la suite des opérateurs classiques `opérateur_logique` moyennant l'ajout d'un mot clé (`ALL` ou `ANY`).

```
WHERE expr1 opérateur_logique ALL (SELECT ...)
WHERE expr1 opérateur_logique ANY (SELECT ...)
```

- `ALL` : la comparaison est vraie si elle est vraie pour tous les éléments de l'ensemble (donc vraie si l'ensemble des enregistrements est vide).
- `ANY` : la comparaison est vraie si elle est vraie pour au moins un élément de l'ensemble (donc fausse si l'ensemble des enregistrements est vide). Vous pourrez que noter que `= ANY (...)` équivaut à `IN (...)`.

4. `GROUP BY ... [HAVING ...]`
La commande

```
GROUP BY expr1, expr2, ...
```

permet de subdiviser la table en groupes, chaque groupe étant l'ensemble des enregistrements ayant une valeur commune pour les expressions spécifiées. Les champs de la clause `SELECT` doivent alors être des fonctions d'agrégation ou des expressions figurant dans la clause `GROUP BY`. Comme son nom l'indique, une fonction d'agrégation s'applique à chaque groupe d'enregistrements créé par la clause `GROUP BY`.
`HAVING prédicat` sert à sélectionner une partie seulement des groupes formés par `GROUP BY`. Le prédicat ne peut porter que sur des fonctions d'agrégation ou des expressions figurant dans la clause `GROUP BY`.

5. `ORDER BY ...`
La commande

```
ORDER BY expr1 [ASC | DESC], expr2 [ASC | DESC], ...
```

classe les enregistrements retournés selon les valeurs de `expr1` puis `expr2`, ...
L'ordre par défaut est croissant, la clause `ASC` est sous-entendue et par conséquent facultative. La clause `DESC` implique un tri par ordre décroissant. Pour préciser lors d'un tri sur quelle expression va porter le tri, il est possible de donner sa position dans la liste des expressions de la clause `SELECT` ou encore l'alias qu'on lui a attribué.

6. `LIMIT ...`
La commande

```
LIMIT nb [OFFSET dec]
```

limite à `nb` le nombre d'enregistrements retournés. Cette clause est souvent utilisée après un tri.
L'option `OFFSET dec` permet d'occulter les `dec` premiers résultats. Ainsi par exemple, `LIMIT 3 OFFSET 8` retournera les enregistrements compris entre 8 et 11.

3.4.3 Jointures

Avec une base de données complexe, on a souvent besoin de rassembler des données réparties dans plusieurs tables, typiquement concaténer les enregistrements provenant de deux tables, `table1` et `table2`, et vérifiant `table1.champ1 = table2.champ2` (l'un des deux `champ1` ou `champ2` est souvent une clé primaire dans sa table).
Pour ce faire, une solution (coûteuse) consiste à former le produit cartésien des deux tables, puis à sélectionner parmi le (trop) grand nombre d'enregistrements obtenus :

```
SELECT expr1, expr2, ... FROM table1, table2 WHERE table1.champ1 = table2.champ2
```

Il faut préférer à ce type de requête l'utilisation d'une jointure (partie du produit cartésien formée des enregistrements vérifiant la condition). Une telle jointure est crée par les logiciels de gestion de bases de données de façon beaucoup plus efficace que la sélection dans le produit cartésien. La syntaxe est alors la suivante :

```
SELECT expr1, expr2, ... FROM table1 JOIN table2 ON table1.champ1 = table2.champ2
```

Remarque

Le résultat de `JOIN ... ON ...` étant une nouvelle table, on peut enchaîner plusieurs clauses `JOIN`.

3.4.4 Les opérateurs ensemblistes

Les opérateurs ensemblistes `opérateur` tels que `UNION`, `INTERSECT`, `MINUS` utilisés ainsi :

```
requête1 opérateur requête2 ...
```

permettent de réaliser des opérations ensemblistes sur les résultats de plusieurs interrogations.
Les champs renvoyés par les requêtes impliquées doivent être identiques. Les opérations sont évaluées de gauche à droite mais l'utilisation de parenthèses permet de modifier cet ordre.

3.5 Contenu de la base

Sur le site http://sql.sh/736-base-donnees-villes-francaises, on peut trouver une table des communes de France. Dans le fichier *bdd_geo.db* fourni, seuls certains attributs ont été conservés. Vous y trouverez 3 tables. La première, intitulée *communes*, contient les attributs suivants :

- **id** : identifiant,
- **num_departement** : numéro du département,
- **nom** : nom de la commune,
- **canton** : numéro de canton de la commune,
- **code_postal** : code postal de la commune,
- **population_2010** : nombre d'habitants lors du recensement de 2010,
- **population_1999** : nombre d'habitants lors du recensement de 1999,
- **densite_2010** : densité en 2010,
- **surface** : surface de la commune en km^2,
- **longitude_deg** : longitude du centre de la commune en degrés,
- **latitude_deg** : latitude du centre de la commune en degrés,
- **z_min** : hauteur minimale de la commune par rapport au niveau de la mer,
- **z_max** : hauteur maximale de la commune par rapport au niveau de la mer.

La deuxième, intitulée *departements*, contient les attributs suivants :

- **num_dept** : numéro du département,
- **num_region** : numéro de la région
- **nom_dept** : nom du département

La troisième, intitulée *regions*, contient les attributs suivants :

- **num_region** : numéro de la région
- **nom_region** : nom de la région

3.6 Exercice

Sauf indication contraire, on travaille sur la population en 2010.

3.6.1 Requêtes sans jointure

1. Obtenez le nombre d'habitants en France (en supposant que chaque habitant est rattaché à exactement une commune) ?
2. Combien y-a-t-il de communes en France ? de communes ayant des noms différents ?
3. Combien de communes dans l'Ain (numéro de département 1) ?
4. Obtenez la liste des 5 plus petites communes de France en terme de surface.
5. Obtenez la liste des 6 communes de France les plus peuplées.
6. Quelles sont les 7 communes les plus densément peuplées de l'Ain ?
7. Obtenez la commune la plus au Nord ainsi que celle la plus au Sud en France métropolitaine.
8. Obtenez la liste des numéros des départements, et pour chaque département correspondant, le nombre de communes par ordre décroissant (on se limitera aux 9 premiers départements).
9. Obtenez la liste des numéros des départements, et pour chaque département correspondant, la population totale. Ordonnez par population totale décroissante en se limitant aux 11 premiers départements.
10. Combien de communes françaises contiennent la lettre "z" ou "Z" dans leur nom ?
11. Combien de communes françaises contiennent les six voyelles "a", "e", "i", "o", "u", "y" (en majuscule ou minuscule) dans leurs noms ?
12. Combien de communes dans l'Ille-et-Vilaine (35) contiennent les six voyelles "a", "e", "i", "o", "u", "y" (en majuscule ou minuscule) dans leurs noms ?
13. Quelles sont les 6 communes de France dont l'écart entre l'altitude la plus haute, et l'altitude la plus basse, est le plus élevé ?
14. Quelles sont les 5 communes du Morbihan (56) où la plus population a le plus augmenté entre 1999 et 2010 ? Répondez en augmentation absolue d'abord, puis en augmentation relative.

3.6.2 Jointures à deux tables

15. Obtenez la liste des départements (les noms), et pour chaque département correspondant, le nombre de communes. Vous ordonnerez par nombre de communes décroissant en vous limitant aux 12 premiers obtenus.
16. Obtenez la liste des départements des régions "Grand Est" et "Occitanie".
17. Quelle est la population en 2010 de chaque département (on souhaite les noms des départements en toutes lettres) ? Les départements seront classés par population décroissante en se limitant aux 6 premiers.
18. Quelle est la densité de population en 1999 de chaque département (on veut les noms des départements en toutes lettres) ? On se limitera aux 8 premiers départements.
19. Obtenez la liste des régions de France, ainsi que le nombre de départements composant la région, rangée par nombre de départements décroissant.
20. Obtenez la liste des départements (les noms) de plus de 1500000 habitants.
21. Donner la liste des départements (les noms) où il y a une ville de plus de 200000 habitants. On se limitera aux 7 premiers.
22. Obtenez les noms de commune de la région "Bretagne" (3) dont le nom contient les six voyelles "a", "e", "i", "o", "u", "y" (en majuscules ou minuscules) ?
23. Donnez toutes les communes ayant le même nombre d'habitants qu'une autre en vous limitant aux 10 premières.

3.6.3 Jointures à trois tables

24. Obtenez la liste des régions de France, et pour chaque région, la population totale, ainsi que le nom et le nombre d'habitants de la commune la plus peuplée.
25. Obtenez la liste des régions (avec le nom en français), avec la population totale de chaque région.
26. Obtenez la densité de population de chaque région en 1999.
27. Quelles sont les 6 communes de France dont l'écart entre l'altitude la plus haute, et l'altitude la plus basse, est le plus élevé ? Cette fois, précisez le nom du département et le nom de la région pour chaque commune.
28. Obtenez le nombre de communes en Normandie.

3.6.4 Utilisation de sous-requêtes

29. Obtenez (sans doublons) la liste des régions contenant des départements dont le nom commence par "f" ou "F" (vous devriez y trouver la Bretagne).
30. Obtenez (sans doublons) la liste des régions ne contenant aucune commune dont le nom contient les six voyelles.
31. Trouvez les communes ayant un nom identique à celui d'une autre commune (on se limitera à 10).

3.6.5 Un peu de dessin

32. Représenter les communes de métropole (avec un numéro de département inférieur ou égal à 95) par des points de couleur fonction de l'altitude du point le plus haut dans la commune (points placés en fonction des longitude et latitude).
33. Représenter par des points de couleur l'attractivité des cantons français, représentée par l'augmentation relative du nom d'habitants entre 1999 et 2010 (points placés en fonction des longitude et latitude).
34. Représenter par des points les 3000 plus grosses communes de métropole (points placés en fonction des longitude et latitude).

Partie 2

Listes

Table des matières

Figures et tables

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des programmes

Index

Index

Index Python

G

H

I

J

L

M

N

O

P

R

S

T

V

X

Commandes Python

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T