

Travaux pratiques de mathématiques appliquées

Calcul matriciel – Opérations bit à bit

Manipulation d'images – Stéganographie

Durée prévue : 5 H

1 Introduction

1.1 Objectifs

Les objectifs de ce laboratoire sont les suivants :

- modéliser la notion de matrice de nombres entiers au moyen d'une classe Java ;
- apprendre à utiliser les opérations logiques bit à bit et les opérations de décalage de bits ;
- manipuler des images représentées sous la forme de matrices de points ;
- apprendre à cacher une image dans une autre (stéganographie).

Ce laboratoire utilise et met en pratique les notions expliquées dans les chapitres 6 et 10 du cours théorique. ***L'étudiant est supposé avoir révisé la matière de ces chapitres avant le début du laboratoire !***

Dans un premier temps, vous devrez compléter l'ébauche de la classe « Matrix » qui vous est fournie, afin de l'enrichir d'un certain nombre de constructeurs, méthodes de test et opérations élémentaires sur des matrices. Vous utiliserez la classe ainsi complétée pour réaliser les autres exercices.

Afin de valider votre compréhension des opérations logiques bit à bit et des opérations de décalage de bits, l'exercice 2 vous propose de décoder un message secret.

Dans les exercices 3 et 4, nous montrerons qu'une image bitmap peut être représentée par une matrice. Vous utiliserez des opérations de calcul matriciel pour manipuler et transformer des images.

1.2 Evaluation

Les séances de travaux pratiques ont un caractère essentiellement formatif. Les exercices proposés dans cet énoncé ne seront pas notés. Néanmoins, il est fondamental que vous tentiez d'en résoudre un maximum par vous-même, ceci afin d'acquérir une bonne compréhension et une bonne maîtrise de la matière du cours.

Si vous éprouvez des difficultés, n'hésitez pas à solliciter l'aide de votre responsable de laboratoire. Vous pouvez également collaborer avec vos condisciples pour élaborer les solutions. Assurez-vous cependant que vous avez compris les notions mises en pratique. Ne vous contentez pas de recopier une solution sans la comprendre : vous devez être capable de la recréer par vous-même.

1.3 Consignes générales

Les exercices proposés dans cet énoncé doivent être résolus au moyen du langage Java et de l'environnement de développement Eclipse, en complétant le projet de départ « 2425_B1MATH_MAT.zip » mis à votre disposition sur HELMo Learn.

Le projet de départ contient toutes les classes citées dans cet énoncé avec, dans celles-ci, les prototypes des méthodes obligatoires que vous devrez compléter pour la réalisation des différentes tâches. Si nécessaire, vous pouvez enrichir le projet avec vos propres packages, classes et méthodes.

Chaque méthode est précédée d'une Javadoc. Lisez attentivement les informations qui y sont données en complément de l'énoncé.

Le projet de départ contient un plan de test pour la plupart des classes à compléter. Ce plan de test valide les éléments essentiels de votre solution, mais il n'a pas la prétention d'être exhaustif !

2 La classe « Matrix » (2h15)

2.1 Introduction

Le projet de départ disponible sur HELMo Learn contient l'ébauche d'une classe « Matrix » qui modélise une matrice de nombres entiers. Elle permet de créer des instances de telles matrices et de réaliser des opérations sur celles-ci.

La classe « Matrix » du package « matrix » contient un certain nombre de méthodes déjà complétées. Ces méthodes sont listées ci-dessous. Consultez le code ou générez la documentation Javadoc du projet pour obtenir plus d'informations au sujet des différentes méthodes fournies.

Constructeurs

- `public Matrix(int numRows, int numCols)`
- `public Matrix(int[][] data)`

Accesseurs et mutateurs

- `public int getNumRows()`
- `public int getNumCols()`
- `public int get(int row, int col)`
- `public void set(int row, int col, int value)`

Méthodes de test

- `public boolean equals(Matrix mat)`
- `public boolean hasSameSize(Matrix mat)`
- `public boolean isDiagonal()`

Opérations sur les matrices ou matrice/scalaire

- `public Matrix add(Matrix m) // Addition de 2 matrices`
- `public Matrix add(int v) // Addition d'un nombre à une matrice`

Méthodes utilitaires

- `public void print()`
- `public void print(int numDigits)`
- `public void printAscii()`
- `public String toAscii()`

2.2 Méthodes additionnelles

Dans ce premier exercice, vous devez compléter l'ébauche de la classe « `Matrix` » qui vous est fournie, afin de l'enrichir d'un certain nombre de constructeurs, méthodes de test et opérations élémentaires. Vous utiliserez la classe ainsi complétée pour réaliser les exercices suivants.

Exercice 1 – Compléter la classe « `Matrix` » (1h30)

- Complétez les méthodes suivantes dans la classe « `Matrix.java` ».
Consultez le code ou la documentation Javadoc du projet pour obtenir la description des différentes méthodes. Inspirez-vous des méthodes existantes.
- `public Matrix(int numRows, int numCols, int value)`
- `public Matrix(Matrix mat)`
- `public boolean isNull()`
- `public boolean isSquare()`
- `public boolean isSymmetric()`
- `public Matrix and(Matrix mat)`
- `public Matrix or(Matrix mat)`
- `public Matrix div(int v)`
- `public Matrix and(int v)`
- `public Matrix or(int v)`
- `public Matrix not()`
- `public Matrix shiftLeftLogical(int n)`
- `public Matrix shiftRightLogical(int n)`
- Vérifiez que les paramètres fournis lors de l'appel d'une méthode sont valides. Par exemple, pour pouvoir additionner ou multiplier deux matrices, elles doivent avoir des tailles compatibles. À défaut, la méthode doit lever une exception au moyen de l'instruction « `throw new IllegalArgumentException(<message>)` ». Les méthodes prédéfinies illustrent ce comportement.
- **Consignes à respecter !!!**
 1. Ne modifiez pas les signatures des méthodes.
 2. Commentez votre code à bon escient.
 3. Ne réinventez pas la roue ! Réutilisez autant que possible les méthodes qui existent déjà dans la classe plutôt que de dupliquer du code.
- Testez que vos méthodes fonctionnent correctement au moyen des plans de test JUnit.

2.3 Les opérateurs logiques bit à bit et les opérations de décalage

Nous allons à présent utiliser les méthodes de la classe « Matrix » et valider que vous parvenez à utiliser à bon escient les opérateurs logiques bit à bit et les opérations de décalage de bits.

Pour ce faire, nous vous demandons de décoder un message secret qui a été chiffré selon le principe décrit ci-après.

- Chaque caractère du message en clair est codé par un entier sur 32 bits que l'on peut décomposer en 4 octets identifiés par A, B, C et D :

A	B	C	D
---	---	---	---

- Les octets A, B, C et D ont été mélangés comme suit :

D	C	B	A
---	---	---	---

- On a ensuite effectué le complément logique (NOT) des 4 bits de poids faibles de chaque octet (c'est-à-dire les bits 0-3, 8-11, 16-19 et 24-27).

31	24	23	16	15	8	7	0
D		C		B		A	
	Not		Not		Not		Not

Exemple

Départ	A	B	C	D
	00000101	00111101	11001010	11111011
Mélange	D	C	B	A
	11111011	11001010	00111101	00000101
Complément	11110100	11000101	00110010	00001010

Exercice 2 – Message secret (45 min)

- Dans la classe « Decode.java » du package « bitwise », complétez la méthode « decode() » de manière à décoder le message secret.

```
public static Matrix decode(Matrix m)
```

- Le message est fourni sous la forme d'une matrice d'entiers, dont chaque élément a subi la transformation décrite ci-dessus. À vous de décoder le message en réalisant la transformation inverse au moyen d'opérations logiques bit à bit et de décalages de bits sur la matrice d'entrée.
- Remarque: les méthodes de la classe Matrix ne modifient jamais la matrice courante (this) ou une matrice passée en paramètre, mais renvoient une nouvelle instance qui contient le résultat de l'opération demandée.
- Vérifier le bon fonctionnement de votre programme en exécutant le plan de test JUnit et la méthode main().

Attention, le décodage doit être réalisé exclusivement avec des opérations de calcul matriciel. Il est interdit d'utiliser des boucles ou de faire appel à un constructeur (new).

3 Matrices, images, couleurs RGB (45 min)

3.1 Représentation d'une image par une matrice

Une image bitmap en couleur n'est rien d'autre qu'une zone rectangulaire composée de points de couleur. Il est donc possible de représenter une image par une matrice M :

- l'élément $m(i, j)$ de la matrice correspond au point de coordonnées (i, j) de l'image ;
- la valeur de l'élément indique la couleur du point.

3.2 Le système de couleurs RGB

Une manière courante de représenter la couleur d'un point est d'utiliser le système RGB (Rouge / Vert / Bleu) sur 24 bits. L'intensité de chaque composante est représentée par un nombre sur 8 bits qui peut donc varier de 0 à 255 : 0 signifie que la composante est absente et 255 correspond à son intensité maximale.

Les 3 composantes sur 8 bits sont concaténées de manière à former un entier sur 24 bits. Ce nombre représente le code RGB de la couleur choisie. Le code RGB est souvent représenté par un triplet avec les valeurs des 3 composantes, ou de manière plus compacte par un nombre en hexadécimal dans lequel chaque composante correspond à 2 chiffres hexa. Voici quelques exemples de couleurs :

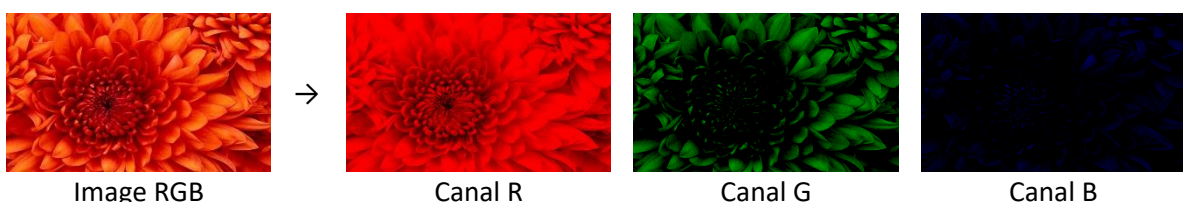
- | | | | |
|---------|-------------------------|---------------------|-------------------------|
| • Noir | (0, 0, 0) | 0x000000 | |
| • Rouge | (1, 0, 0) à (255, 0, 0) | 0x010000 à 0xFF0000 | rouge foncé à rouge vif |
| • Vert | (0, 1, 0) à (0, 255, 0) | 0x000100 à 0x00FF00 | vert foncé à vert vif |
| • Bleu | (0, 0, 1) à (0, 0, 255) | 0x000001 à 0x0000FF | bleu foncé à bleu vif |
| • Blanc | (255, 255, 255) | 0xFFFFFFFF | |

Lorsque le code RGB est exprimé sur 32bits, l'octet de poids fort est soit mis à zéro, soit il représente une valeur de transparence (canal alpha).

Nous avons expliqué au cours théorique (chapitre 10) comment extraire l'intensité d'une composante RGB, ou encore comment en modifier la valeur, au moyen des opérations logiques et de décalage bit à bit.

3.3 Canaux R, G et B

On appelle « canal rouge » d'une image, l'image obtenue en conservant la composante R de l'image de départ et en annulant les composantes G et B. Exemple : la couleur 0xA1B2C3 devient 0xA10000. On obtient alors une image composée exclusivement de nuances de rouge. Il en va de même pour le canal vert (0x00B200) et le canal bleu (0x0000C3).



Attention, il ne faut pas confondre le canal rouge (0xA10000 = 10.551.296) et la composante rouge (0xA1 = 161). Idem pour le canal vert et la composante verte (0xB200 ≠ 0xB2).

3.4 Cinquante nuances de gris et plus...

Les nuances de gris correspondent aux couleurs RGB pour lesquelles les 3 composantes ont des intensités identiques : 0x000000, 0x010101, 0x020202, ..., 0xFFFFFF. Il y a donc 256 nuances de gris si on inclut le noir (0x000000) et le blanc (0xFFFFFF).

Il existe différentes techniques pour convertir une couleur RGB vers une nuance de gris. La méthode la plus simple consiste à effectuer la moyenne arithmétique des 3 composantes R, G et B. Cette valeur moyenne est ensuite utilisée pour définir l'intensité des 3 composantes.

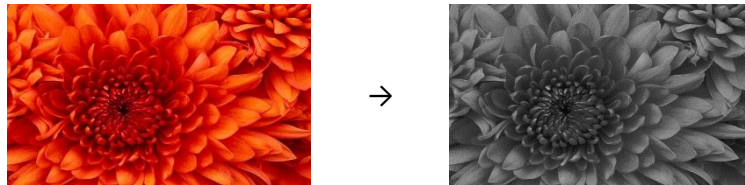
Exemple

Soit la couleur 0x6432E6, c'est-à-dire rgb(100,50,230), on calcule la moyenne des 3 composantes.

$$Y = (100+50+230)/3 = 126$$

La nuance de gris correspondante est rgb(126,126,126) = 0x7E7E7E.

En appliquant cette technique à l'image du chrysanthème, on obtient :



Exercice 3 – Canaux R/G/B – Nuances de gris (45 min)

Dans la classe « Imaging.java » du package « bitwise » complétez les méthodes suivantes :

```
public static Matrix redChannel(Matrix img)
public static Matrix blueChannel(Matrix img)
public static Matrix greenChannel(Matrix img)
public static Matrix grayScale(Matrix img)
```

- Chaque méthode reçoit en argument une matrice qui représente une image en couleurs et renvoie une nouvelle matrice qui correspond au canal rouge/vert/bleu ou à la version en nuances de gris de l'image de départ.
- L'extraction des canaux R/G/B sera réalisée en utilisant uniquement des opérations logiques bit à bit et de décalages. Seul le calcul de la valeur moyenne $Y=(R+G+B)/3$ peut utiliser des opérations arithmétiques.
- Rappel : il ne faut pas confondre la couleur d'un point dans le canal rouge (ex : 0xA10000) et l'intensité de la composante rouge (ex : 0xA1) qui sert à calculer la nuance de gris.
- Vérifier le bon fonctionnement de votre programme en exécutant le plan de test JUnit et la méthode main().

Attention, ces exercices doivent être réalisés exclusivement avec des opérations de calcul matriciel. Il est interdit d'utiliser des boucles ou de faire appel à un constructeur (new).


4 Stéganographie (2h)

4.1 La perception des couleurs par l'œil humain

Le système RGB 24 bits permet de coder 2^{24} couleurs différentes, soit 16.777.216 couleurs. Cependant, l'œil humain n'est pas capable de discerner des couleurs très proches les unes des autres. On estime qu'il peut « seulement » distinguer environ un demi-million de couleurs.

De même, un écran d'ordinateur n'est généralement pas capable de restituer avec fidélité les 2^{24} couleurs différentes du système RGB. Sur un écran moyen, on peut parfois plafonner à quelques centaines de milliers ! En pratique, de nombreuses couleurs distinctes seront donc affichées ou perçues de la même manière.

Exemple : les deux couleurs suivantes sont indiscernables

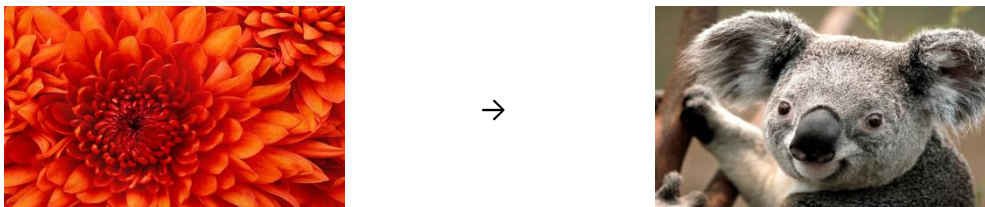
rgb(200,100,50)  rgb(205,105,50)

Nous allons utiliser cette caractéristique pour dissimuler une image à l'intérieure d'une autre. Ce procédé fait partie des techniques dites de « **stéganographie** ».

4.2 Dissimuler une image dans une autre

Là où la cryptographie s'attache à dissimuler le sens d'un message en le brouillant, la stéganographie s'attache à essayer de dissimuler l'existence même de ce message. Par exemple, en le cachant à l'intérieur d'un autre.

Exemple : qui pourrait deviner qu'un koala se cache dans ce beau chrysanthème ?



Pour réaliser notre tour de passe-passe, nous allons exploiter les limitations de l'œil humain qui ne distingue pas des couleurs très proches.

Rappelons qu'une composante RGB est codée sur 8 bits et permet donc 256 valeurs comprises entre 0 et 255. Considérons par exemple, le rouge rgb(255,0,0)

1	1	1	1	1	1	1	1	255
---	---	---	---	---	---	---	---	-----

Si on modifie, les 3 derniers bits de poids faibles pour les mettre à 0, on ne distingue quasiment aucun changement de couleur.

1	1	1	1	1	0	0	0	248
---	---	---	---	---	---	---	---	-----

En revanche, si on modifie un bit de poids fort, la différence est très sensible.

0	1	1	1	1	1	1	1	127
---	---	---	---	---	---	---	---	-----

On peut déduire de cette observation, que l'information principale au sujet d'une nuance de couleur est véhiculée par les bits de poids forts, tandis que les bits de poids faibles induisent uniquement de faibles variations difficilement perceptibles.

Pour cacher une image dans une autre, nous allons simplement remplacer un certain nombre de bits de poids faibles de l'image « hôte » par un nombre de bits de poids forts équivalent provenant de l'image secrète à dissimuler.

Image hôte	H ₇	H ₆	H ₅	H ₄	H ₃	H ₂	H ₁	H ₀
Image secrète	S ₇	S ₆	S ₅	S ₄	S ₃	S ₂	S ₁	S ₀
Stéganogramme	H ₇	H ₆	H ₅	H ₄	H ₃	S ₇	S ₆	S ₅

Pour extraire l'image dissimulée dans le stéganogramme, on réalise l'opération inverse, c'est-à-dire : on extrait les bits de poids faibles du stéganogramme et on les place dans les positions de poids forts.

Image extraite	S ₇	S ₆	S ₅	0	0	0	0	0
----------------	----------------	----------------	----------------	---	---	---	---	---

Il est clair que l'image extraite n'est pas identique à l'image de départ, les couleurs sont légèrement modifiées, mais elle reste néanmoins reconnaissable.

Exercice 4 – Stéganographie (2h)

- Dans la classe « Steganography.java » du package « stegano » commencez par compléter les méthodes suivantes :


```
public static int rgbLsbMask(int n)
public static int rgbMsbMask(int n)
```
- Ces 2 méthodes servent à créer des masques de n bits de poids faibles (respectivement de poids forts) que nous utiliserons pour extraire les informations utiles lors de la création ou de l'analyse du stéganogramme.

Exemples

- `rgbLsbMask(3)` → 00000111 00000111 00000111
- `rgbMsbMask(3)` → 11100000 11100000 11100000

- Complétez à présent les méthodes suivantes qui utiliseront `rgbLsbMask()` et `rgbLsbMask()` :

```
public static Matrix steganoCreate(Matrix publicImage, Matrix
secretImage, int hiddenBits)
```

```
public static Matrix steganoExtract(Matrix steganogram, int
hiddenBits)
```

- Testez vos methodes !

Attention, ces exercices doivent être réalisés exclusivement avec des opérations de calcul matriciel logiques bit à bit ou de décalage !

Bon travail !