Annals of examinations

(In French until 2020). Corrections are available on the course web page (`http://imagine.enpc.fr/~monasse/Info/`).

1 Final examination on machine 2022

1.1 Power Spectrum of a Function

The power spectrum of a function is the modulus of its Fourier frequencies. A periodic function can be decomposed into a linear combination of sine and cosine functions at different frequencies. Each one is the sum of two complex exponentials with opposite frequencies. A simple formula extracts the coefficients of the linear combination. We will implement it and observe the power spectrum of some usual functions.[1]

It is more important to deliver a clean code (commented and correctly indented) that *compiles* than answering all questions. For that, check after each step that the build works. Write in comments of the code the question number. At the end, create an archive with source code and file `CMakeLists.txt` to upload on educnet.

The following functions and constant from #include <cmath> will be used: `sqrt`, `cos`, `sin`, `exp`, `log` and `M_PI` (π). Consistently labeling methods as const when appropriate will be appreciated.

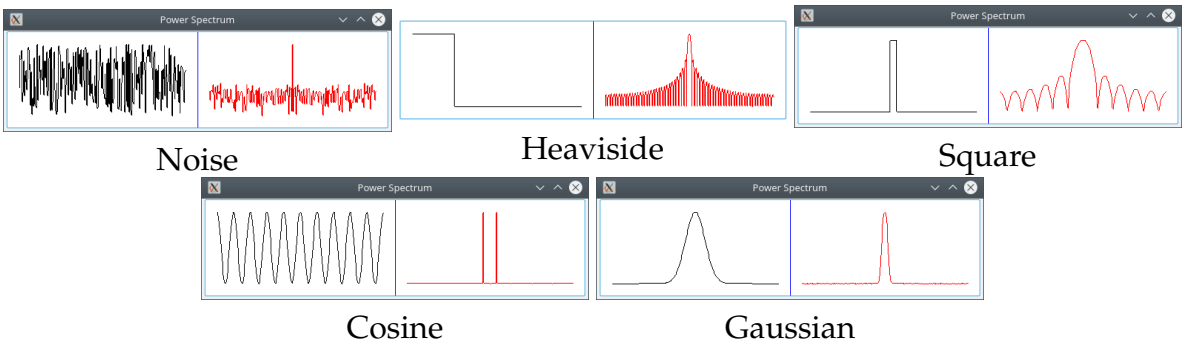


Figure 1: Some usual functions (black) and their power spectrum (red).

[1] More fun with the discrete Fourier transform will be illustrated in the course "Algorithmics and Data Structures", along with a fast algorithm to compute it.

1.2 Complex numbers

1. Create an Imagine++ project. In a file *separate* from the one containing your `main` function, define the class `complex` storing real and imaginary parts as float.[2]

2. Define two constructors for the class, the first one taking a single float for the real part (representing a real number), the second one taking two for real and imaginary parts.

3. Define the operators for addition and subtraction of two complexes outside the class.

4. Define the method `bar` returning the conjugate $\bar{z}$ of its complex argument z.

5. Define the methods `modulus` and `modulus2` of class complex. The latter is $|z|^2 = z\bar{z} \in \mathbb{R}$ and the former its square root.

6. Define the operators for multiplication and division of two complexes outside the class.[3] For the latter, remember that $1/z = \bar{z}/|z|^2$. The real $1/|z|^2$ can be computed and the formula implemented as a multiplication of three terms.

7. Define operator+= and operator$*$= inside the class.

8. Define the function `exp` computing the exponential of a complex number. Remember that $e^{i\theta} = \cos\theta + i\sin\theta$.

9. Following the model of Section 12.2.3 of the lecture notes, define operator<< so that we can write cout << z. Use compact display, see next question.

10. In a function `test_4ops()`, called from main(), define complex constants zero, one and i and make it print the following complex numbers:
    `0 1 i -i 1+i 1-i 2+3i`
    Notice it is the compact format we require, the following represent the same with a bad format:
    `0+0i 1+0i +1i -1i 1+1i 1-1i 23i`
    (the last one a bug because float im=3; cout<<im; does not display the $+$).

11. Make the same function display
    `i*i = -1.  (1+i)(1-i) = 2.  1/i = -i.  exp(i pi)+1 = -8.74228e-08i`
    (notice the small numerical error $< 10^{-7}$ in Euler's formula). Naturally, the right hand side of each equality is computed in your code by the corresponding operation.

[2] The standard C++ library has already a type `std::complex`, but it is more exciting to create our own!

[3] A benefit of defining them outside the class combined with the constructor taking a single argument is that we can write in the code $3+4*i$ when i is complex, reals 3 and 4 are automatically cast to complex type.

1.3 Signal

12. In a new separate file, define a class `Signal` storing an array of float numbers whose number of elements is not known in advance. Two fields `n` and `nmax`, always with $n \leq$`nmax`, control the array: `nmax` is the current size of the allocated array and `n` the actual numbers of elements used.

13. Write a constructor with no argument, the signal is initially empty with `nmax`=0 and the array not allocated.

14. Write the destructor.

15. Write the copy constructor. *Bonus*: in which question and why is it crucial to have the copy constructor defined? (answer as comments in code).

16. Write a method `size` returning the actual number of elements and `operator[]` for element access, the latter checking with `assert` that the index makes sense.

17. Write a method `add` appending its float argument to the signal. If $n =$`nmax` we must (re)allocate an array: if $n = 0$ we put `nmax`$= 1$, otherwise we double `nmax`.

18. Write a method `bounds` computing the minimum and maximum values of the signal.

19. Write a private method `fourier(int k)` computing the coefficient of index k of the Fourier transform:

$$\hat{f}[k] = \sum_{j=0}^{n-1} f[j]\, e^{-2ijk\pi/n} \in \mathbb{C}. \tag{1}$$

    Do not recompute the complex exponential for each term of the sum, use the recurrence relation $e^{-2i(j+1)k\pi/n} = e^{-2ik\pi/n}\, e^{-2ijk\pi/n}$ (geometric sequence).

20. Write a method `modFourier` returning a new signal with the same length and with coefficients $\log(1 + |\hat{f}[k]|)$, $0 \leq k < n$ (the power spectrum).

21. The second half of the Fourier coefficients should rather be considered as negative frequencies, so write a method `shift` that swaps the two halves of the signal (index $k = 0$ is at the middle, -1 at its left, $+1$ at its right, etc).

1.4 Display power spectra

22. The equation for the affine function $y = ax+b$ passing through (x_1, y_1) and (x_2, y_2) has coefficients $a = (y_1 - y_2)/(x_1 - x_2)$ and $b = (x_1y_2 - x_2y_1)/(x_1 - x_2)$. Write the function computing a and b (if $x_1 = x_2$ take $a = 0$, $b = (y_1 + y_2)/2$).

23. Write a function `draw` taking a signal S, coefficients a_x, b_x, a_y, b_y and a color, and drawing a polygonal line joining points $(a_xk + b_x,\ a_yS[k] + b_y)$.

24. Write another function `draw` taking a signal S and displaying the graph of S in the left half of a window of size $w \times h = 512 \times 128$ and its power spectrum in the right half. Each graph is displayed with a scale adapted to the bounds of the signal and letting `margin`$= 16$ pixels of white padding around. A blue line at the middle separates the graphs.

25. Write and call from main a function `noise` generating a signal with random values in $[0, 1]$ and displaying it and its power spectrum. The number of samples is a constant `nsamples`$= w/2$.

26. Same with function `heaviside`: the signal is 1 if $t < c = n/4$, 0 otherwise.

27. Same with function `square` equal to 1 only if $|k - n/2| < r = 5$.

28. Same with function `cosine`: $f[k] = \cos(2if k\pi/n)$ with $f = 10$ an integer.

29. Same with function `gaussian`: $f[k] = \exp(-(k - n/2)^2/\sigma^2)$ ($\sigma = 15$).

30. Modify the function `draw` of Question 24 so that it ends with waiting for a user clicks and returns false in case of right click. Otherwise, the pointer abscissa converted to signal coordinates $X = (x - b_x)/a_x$ is computed.

31. Modify the different signals so that the parameters are chosen interactively by the user until a right click: $c = X$, $r = f = \sigma = |X - n/2|$.

2 Midterm examination on machine 2022

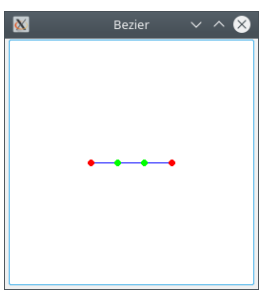
2.1 Bézier Curves

A cubic Bézier curve is a parametric curve used a lot in computer graphics. The curve $P(s)$, $(0 \leq s \leq 1)$, is parameterized by 4 control points $P_0 \dots P_3$:

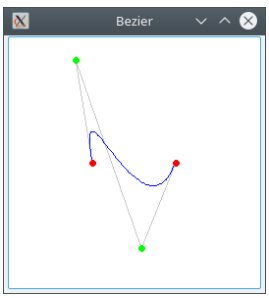
$$P(s) = (1-s)^3 P_0 + 3(1-s)^2 s P_1 + 3(1-s)s^2 P_2 + s^3 P_3. \quad (2)$$

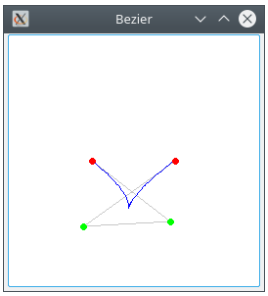
Note that $P(0) = P_0$ and $P(1) = P_3$ but P_1 and P_2 are not on the curve in general. Our program explores the various shapes the curve can take. Without loss of generality, we will assume $P_0 = (0, 0)$ and $P_3 = (1, 0)$.

It is more important to deliver a clean code (commented and correctly indented) that *compiles* than answering all questions. For that, check after each step that the build works. Write in comments of the code the question number. At the end, create an archive with source code and file `CMakeLists.txt` to upload on educnet.



Initial Bezier curve





Two Bezier curves after motion of the control points. Notice the cusp in the right image.

```
bool track(int& x, int& y) {
  Event e;
  while(true) {
    getEvent(-1,e);
    if(e.type == EVT_BUT_OFF)
      return false;
    if(e.type == EVT_MOTION) {
      x = e.pix[0];
      y = e.pix[1];
      return true;
    }
  }
}
```

Function used in question 15.

2.2 Point structure

1. Create an Imagine++ project. In a file *separate* from the one containing your `main` function, define the structure `point` taking coordinates of type `float`.

2. Define the operators for addition and subtraction of two points.

3. Define the operators for multiplication and division by a `float`. For multiplication, define the two operators: `float*point` and `point*float`.

4. Define a function `affine` taking a point and applying the similarity $P \to a*P+S$ with parameters a the zoom factor and S a point (shift). It will be used to map standard coordinates of the curve to pixel coordinates for drawing.

5. Define a function `rotate` that rotates a point P around a center C with an angle α expressed in degrees. Functions $\cos$ and $\sin$ from #include <cmath> take their argument in radians.

2.3 Bézier curve

6. In a new separate file, define a structure `Bezier` storing an array of four points.

7. Write a function `initBezier` that returns a new curve with $P_1 = (1/3, 0)$ and $P_2 = (2/3, 0)$.

8. To draw the curve, we will zoom and shift the coordinates for display. The window will be square with `dim`×`dim` pixels, point P_0 will be displayed at (`dim`/3,`dim`/2) and P_3 at (2`dim`/3,`dim`/2). Values of s will be discretized uniformly at `npoints`$= 100$ values. Define the adequate constants, `dim`$= 512$.

9. The function `draw` takes a curve and displays it: draw a blue line from $P(s)$ to $P(s + \delta s)$ (use `affine` to apply the transform) with $\delta s = 1/$`npoints`.

10. Add in the previous function the display of the 4 points (red for extremities, green for P_1 and P_2), disks of `radius`$= 3$ pixels.

11. Make the `main` function display an initial Bezier curve and wait for a click to continue.

2.4 Animation

12. Write a function `animate` in the main file that applies 100 iterations of rotation around P_0 of $P_1 = (3/4, 0)$ and of $P_2 = (2/3, 0)$ around P_3. At iteration i, P_1 rotates of $i * 10°$ and P_2 of $i * 3°$. After each display, a small pause is observed. At the end, the function should wait a point click.

13. Enrich the function `draw` by linking the control points by lines in gray color of intensity 200. This should be done at the beginning of the function, so that it does not overlap with the rest.

2.5 User interaction

14. Write a function `selectPoint`, taking a curve and waiting until the user clicks on P_1 or P_2. A right click exits and returns false, whereas a left click stores the pixel coordinates of the click and the point number selected. While the left click occurs elsewhere, it continues waiting for a click. Be careful that the click coordinates are in pixels while the points are around the interval $[0, 1]$, so that an affine transform must be applied to compare.

15. Write a function `interactive`. It displays an initial Bézier curve and loops undefinitely until `selectPoint` returns with a right click. Inside the loop, it lets the user move the selected controlled point and displays interactively the curve. The function `track` (see figure) is used to detect a mouse motion or the mouse button release. In case of motion, the shift from the previous position in pixels is applied to the control point. Be careful that the scale of display is not the same as the point coordinates.

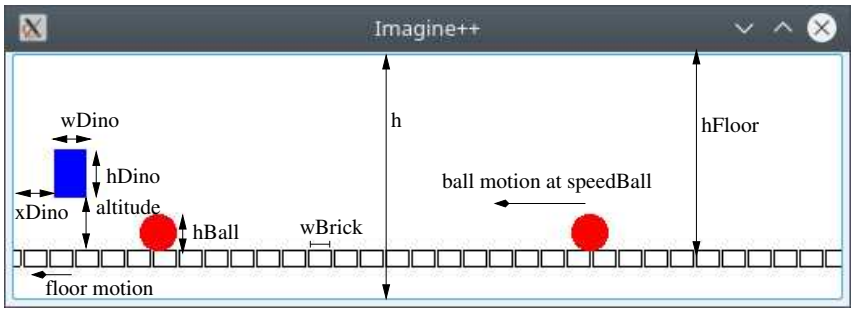
16. *Bonus*: (i) Move the interactive control in a separate program (within the same project CMakeLists.txt), and (ii) create a library for the common functions of the two programs.

3 Final examination on machine 2021

3.1 Dinosaur Game

The goal is to build a simplified version of the Dinosaur Game, a basic game included in the Google Chrome web browser. A dinosaur has to avoid obstacles moving its way by jumping. Our dinosaur will be represented by a rectangle and the obstacles are balls. The game is registered so that it can be viewed backward and forward after the user lost.
Important: All methods of classes take a time `t0` (integer) as parameter.

> **It is more important to deliver a clean code (commented and correctly indented) that *compiles* than answering all questions. For that, check after each step that the build works. At the end, create an archive with source code and file `CMakeLists.txt` to upload on educnet.**




```
int keyboard() {
  Event e;
  do {
    getEvent(0,e);
    if(e.type == EVT_KEY_ON)
      return e.key;
  } while(e.type!=EVT_NONE);
  return 0;
}
```

Figure 2: Blue jumping dinosaur and red balls. Altitude is variable (when the dinosaur jumps), computed question 4. The function `keyboard` returns the pressed key code (without waiting), 0 if none. You can copy-paste it from Practical#8.

3.2 Happy jumping dino

1. Create a new project and a basic `main` function opening a window. In a separate file `dino.h` write the constants: wDino=20 and hDino=30 the dimensions of the dinosaur, xDino=25 the abscissa of the dinosaur (fixed, the decor is moving), $w \times h$=512 $\times$ 5$*$hDino the window dimensions, hFloor=h-hDino the base ordinate of the dinosaur when not jumping.

2. Write a class `Dino` and its constructor. It only needs an integer `t` storing the time of the last jump start. A jump lasts for a constant time tJump=20 and reaches a height hJump=3*hDino (yes, the dinosaur is heavy but is a high jumper!). Initially, the dinosaur is not jumping, so we put its `t` as sufficiently negative.

3. Method `jump` registers that a jump starts. Method `jumping` indicates if the jump is still in process.

4. Method `altitude` computes the altitude above the floor (0 if not jumping). According to Newton's gravitation law, it has a parabolic evolution given by equation

$$h = \text{hJump} * \left(1 - \left(1 - 2\frac{t_0 - t}{\text{tJump}}\right)^2\right). \tag{3}$$

(Proof: $t_0 = t \Rightarrow h = 0$, $t_0 = t + \text{tJump}/2 \Rightarrow h = \text{hJump}$, $t_0 = t + \text{tJump} \Rightarrow h = 0$)

5. In a separate file, define a class `Recorder`. It has a field of type `Dino`. First this class will be used to play the game, the recording part will be coded in Section 3.4.

6. Method `Recorder::display` clears the window and draws the dinosaur at the current time.

7. Method `Recorder::action` calls the function `keyboard`: if the space bar key is pressed, the dinosaur is set to start jumping but only if the last jump has finished.

8. In the `main`, let the user make the dino jump on demand.

9. `Recorder::display` draws the floor, composed of disjoint rectangles of width wBrick=16 and height 10 pixels. The floor is shown moving to the left (as dinosaur moves to the right), the bricks are shifted 1 pixel at each time increment.

3.3 Life becomes harder with moving balls

10. A ball has diameter hBall=3/4 hDino and will be moving with a constant speed speedBall=8 pixels per time increment. It stores a time and abscissa for an initial position. Write class `Ball` in `dino.h` and a constructor, initially at abscissa 1000 for time 0.

11. `Ball::set` records a time and abscissa as initial position.

12. `Ball::center` returns the abscissa of its center at current time. It is based on initial position and speed, moving to the left (toward the poor dino).

13. `Ball::reInit` is used to recycle a ball that was dodged by the jumping dino if it went out of screen: it restarts at abscissa xBase (a method parameter) plus a random gap between one and three times tJump*speedBall pixels, but it must appear to the right, so that the result is set to at least `w` (the window width). The function returns true if recycling took place.

14. Add an array of nBalls=3 balls (nBalls is a constant) in class `Recorder`. In the constructor of the class, the ball are regularly spaced from abscissa w with a space of tJump*speedBall pixels.

15. Insert the display of balls in `Recorder::display` and call `reInit` on all balls in method `action`. The parameter xBase of `reInit` is the position of preceding ball.

16. In the main, let the user play the game with a span of 20 milliseconds for each time increment.

17. Add a method `Dino::crash` taking a ball and indicating whether the ball intersects the rectangle of the dinosaur. The squared distance to the ball center can be computed

$$d^2 = \max(0, h - \text{hBall}/2)^2 + \max(0, \text{xDino} - c)^2 + \max(0, c - \text{xDino} - \text{wDino})^2, \quad (4)$$

with h the altitude of the dino and c the abscissa of the ball center.

18. Add method `Recorder::crash` indicating if the dino crashes with one of the balls. Insert the crash test in the game, letting it finish when it happens.

3.4 Recorder replay

19. Add a structure `Action`, a triplet (t, x, i) with t the time, x the abscissa and i the ball number. An action is either a dino jump start (then x and i are set negative) or a ball reinitialization.

20. The recorder will store all actions during the game. Insert a dynamic array `actions` in `Recorder` reserving initially the space for one action. Modify constructor and destructor accordingly. The management of the array follows this principle: `nmax` actions are initially allocated, but $n = 0$, the actual number of stored actions.

21. Write method `Recorder::record` storing a new action: if n=nmax, double nmax and reallocate the array to leave space for the new action.

22. In `Recorder::action`, store actions if they happen (jump and reinitialization of a ball). In the constructor, record also actions for the initial positions of the balls.

23. Write `Recorder::set` taking a time and resetting the game at this time: find the preceding stored jump in recorded actions, and for each ball the preceding reinitialization.

24. When the game is finished, let the user visualize it back and forth by arrow keys.

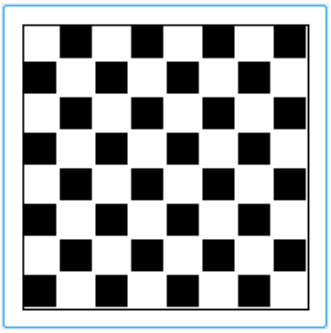
4 Midterm examination on machine 2021

4.1 The Queens Game

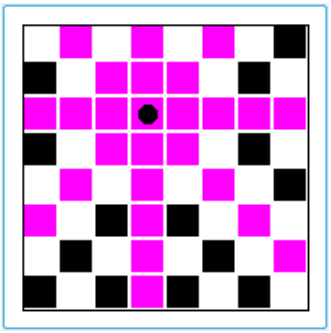
The goal of the game is to place as many queens as possible on a chessboard so that they do not threaten each other. Each queen threatens the cases on same row, column, or diagonal. For most board dimensions, it is possible to have a solution with one queen per row.

It is more important to deliver a clean code (commented and correctly indented) that *compiles* than answering all questions. For that, check after each step that the build works. At the end, create an archive with source code and file `CMakeLists.txt` to upload on educnet.

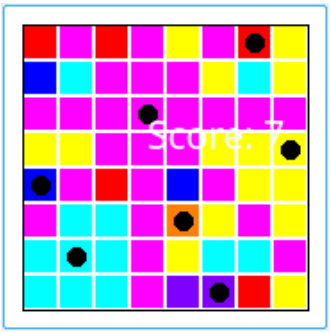
A variant of the queens game.



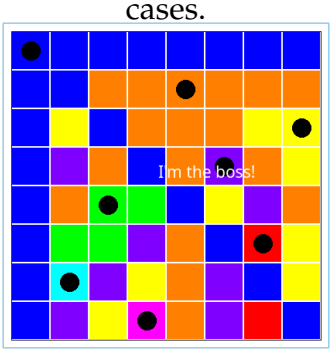
Initial board.



One queen and threatened cases.



Human player, score=7.



Machine, perfect score 8 queens.

4.2 Basic display

1. In a file *separate* from the one containing your `main` function, define the constants of the project: the dimension $n = 8$ (8×8 chessboard), a border (in pixels) so that the chessboard is not displayed right at window's edge, and a zoom factor $z = 100$ for display of each case of the board.

2. Define a structure `Board` comprising a 2D array of cases, each element encoding the color of the case.

3. Define a function `init` that puts the board in an alternance of black and white cases.

4. Define functions `draw` and `drawCase`, respectively for the whole board and for an individual case. Leave a one pixel wide space between adjacent cases.

5. In the main function, create, initialize and display the chessboard.

4.3 Human player

6. Define a function `getCase` converting pixel coordinates (x, y) into case coordinates (row and column). This function will be used to know the case of the board where a mouse click happens.

7. Define the function `centerCase` that returns the pixel coordinates of the *center* of the case at given row and column.

8. Write a function `isFree` that indicates whether a case at given row and column is threatened by a queen of not (white and black indicate free cases, any other color means it is not free).

9. Write a function `nFree` counting the number of free cases of a board.

10. Each queen is associated to a color. Write an array of `n`$= 8$ fixed colors (excluding black and white).

11. Write a function `forbid` that takes a board and a case to paint the free cases threatened by a queen placed there in the fixed color given by the column of the queen. Instead of writing 8 loops, use an auxiliary function taking a position and a direction vector $(dx, dy) \in \{-1, 0, 1\} \times \{-1, 0, 1\} \setminus \{(0, 0)\}$.

12. Using the previous functions, in a function `play` inside the file containing the main function, present an empty board to the player, let her click cases until no more free cases are left. After each click on a free case, the board is repainted to show the threatened case and the queen represented by a black disk.

13. Make the program write the score of the player (number of queens). It may be in a console instead of the graphical window if you do not know how to do it.

4.4 Machine player

A permutation P of $\{0, \ldots, n-1\}$ is these n integers written in any order. The idea is to place queens at all positions $(i, P(i))$ and check that they do not threaten each other. To generate all permutations, we can use the following algorithm:

(a) Take two arrays $t = \emptyset$ and $u = (0, \ldots, n-1)$

(b) If $u = \emptyset$, t is a permutation

(c) Otherwise, take all arrays t'_i deduced from t by appending a single element u_i of u and iterate (b) with arrays t'_i and $u' = u$ minus u_i.

14. Write a structure `tab` storing an array of `n` integers. We will use some arrays with fewer than `n` elements, but we will use a variable to store the actual number of elements.

15. The function `permute` takes two such structures t and u and the actual number of elements in t (if t has k elements, u has $n - k$) to generate all permutations.

16. For each permutation, a function `check` is called, which builds the corresponding board and verifies if it is a winning configuration.

17. In that case, the filled board is shown and a message bragging about the result is displayed. A mouse click is expected to continue searching for more configurations.

5 Examen final sur machine 2020: énoncé

5.1 Jeu 10×10

Nous allons programmer le jeu 10×10, une variante de Tetris mais sans animation ni gravité. On a des pièces de différentes formes qu'il faut placer sur une grille, de taille 10×10 normalement. Toute ligne ou colonne de la grille remplie est détruite et rapporte des points. Le jeu se termine quand un nouvelle pièce apparaît et qu'il n'y a plus d'espace libre où la déposer. Utilisez un projet Imagine++ de base. Reportez dans le code en commentaire les questions auxquelles vous répondez (`Q1`, `Q2`, etc).

Il est important de livrer un code clair (commenté et indenté) et qui compile sans warning, même s'il ne répond pas à toutes les questions. Pour cela, vérifiez à chaque étape que votre programme compile et se lance correctement. À la fin de l'examen, nettoyez votre code (indentation...) et vérifiez encore qu'il compile. Créez alors une archive contenant votre code et le fichier `CMakeLists.txt`.

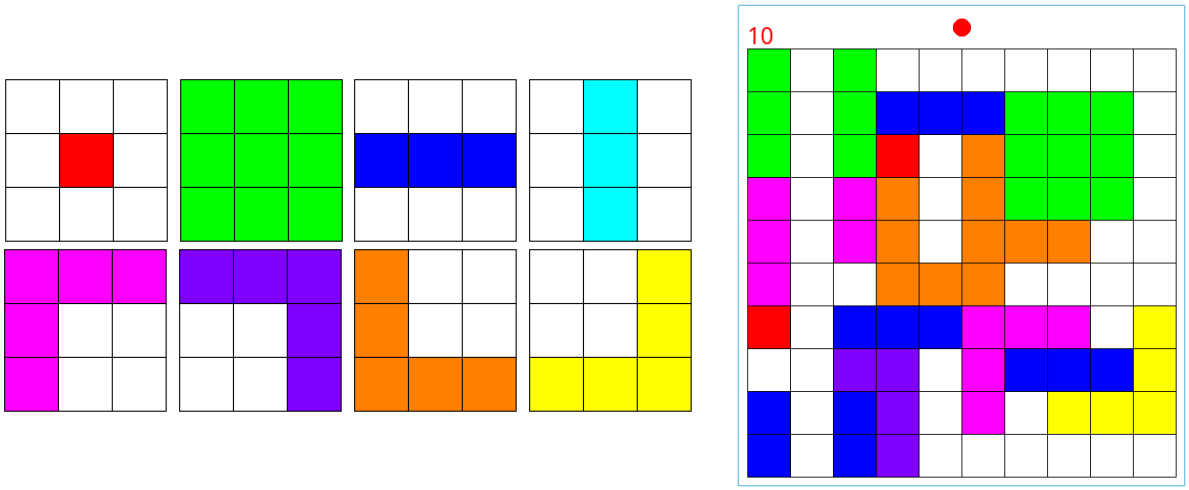


Figure 3: Les 8 pièces du jeu 10×10, composées d'au maximum 3×3 cases (cases blanches non occupées par la pièce), chacune avec sa couleur. À droite, le jeu en cours de partie. Une colonne remplie du plateau 10×10 a été détruite, d'où un score affiché de 10 points. Le voyant rouge en haut indique que la pièce courante (un L orange), située au centre, ne peut être déposée là car elle recouvre au moins une case occupée. Le joueur doit la déplacer à un emplacement libre avant de la déposer.

5.2 Pièces

1. On laisse de chaque côté `marge`$= 10$ pixels avec le bord de la fenêtre, sauf en haut avec `margeScore`$= 50$ pixels. On a une grille carrée de `size`$= 10$ cases de côté et chaque case occupe $z \times z$ pixels avec $z = 50$. Ouvrir une fenêtre de taille correcte pouvant afficher tout cela.

2. Implémenter une fonction `dessine_grille` qui dessine la grille.

3. Dans un fichier séparé, créer une classe `Piece`, composée de jusqu'à 3×3 cases (un booléen indique si la case est remplie), une couleur et une position (x, y) de la case centrale dans la grille.

4. Ajouter un constructeur à la classe. Celui-ci prend une des 8 formes de pièce au hasard (à chaque forme est associée une couleur unique). Sa position est à peu près centrée dans la grille. Pour les 4 pièces en forme de L, essayez de ne pas faire 4 fois des codes similaires.

5. Ajouter une méthode `bloc(i,j)` ($-1 \leq i, j \leq 1$) permettant de savoir si la case centrale $+(i, j)$ est occupée et une méthode `couleur`.

6. Ajouter une méthode `dessine`.

7. Tester cette classe dans une fonction `test_deplacement`: les touches flèches du clavier déplacent une pièce d'une case dans la direction indiquée (on ne vérifie pas si on sort de la grille), la touche "Entrée" termine la fonction. Utiliser la fonction `getKey` d'Imagine++ qui renvoie le code de la touche appuyée par l'utilisateur.

5.3 Plateau

8. Créer une classe `Plateau` composé de $w \times h$ cases. Potentiellement $w \neq h$ et ces dimensions ne sont pas fixées a priori, même si nous n'utiliserons qu'un plateau avec $w = h =$`size`.

9. Écrire constructeur et destructeur de cette classe. Les cases sont initialement blanches (cases libres).

10. Écrire sa méthode `dessine`. Les méthodes qui suivent prennent une pièce en paramètre, que nous appelons p.

11. La méthode `contient`, à implémenter, indique si p ne déborde pas du plateau.

12. La méthode `verifie` indique si p n'occupe que des cases libres (blanches).

13. La méthode `positions` compte le nombre d'emplacements possibles de p sans recouvrement.

14. La méthode `absorbe` "fige" p en copiant sa couleur dans les cases du plateau qu'elle occupe.

15. La fonction `jeu` est une première version qui affiche une nouvelle pièce, laisse le joueur la déplacer avec les flèches du clavier, la fige (si l'emplacement est libre) lors de l'appui sur la touche "Entrée" et passe à la pièce suivante. Le jeu s'arrête lorsque la nouvelle pièce n'a aucun emplacement libre possible.

5.4 Jeu final

16. Écrire une méthode `Plateau::detruis` qui compte les lignes et colonnes remplies, les détruit (remises à blanc) et renvoie le nombre effectif de cases détruites (attention, ne pas compter double une case participant au remplissage d'une ligne et d'une colonne).

17. Incorporer cette fonctionnalité dans le jeu.

18. Tenir le compte et afficher le score (nombre de cases détruites) dans le jeu.

19. Indiquer par un disque vert ou rouge si l'emplacement courant de la pièce est libre ou pas.

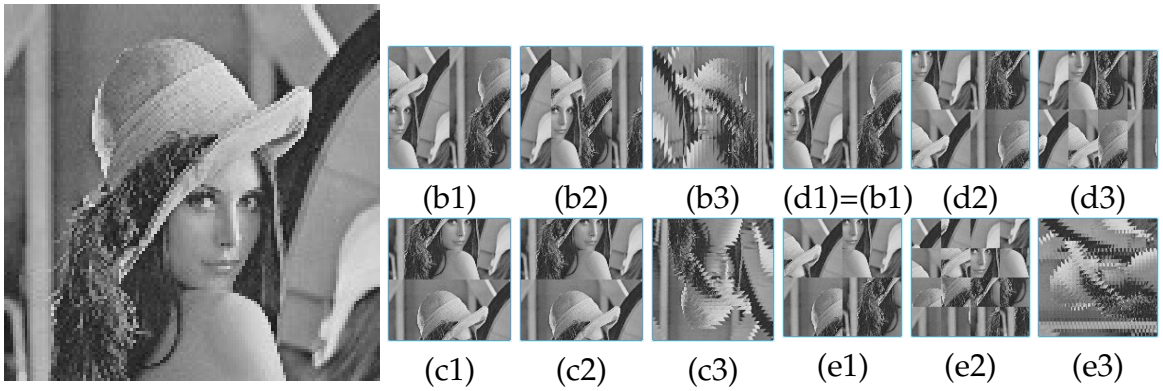
6 Examen partiel sur machine 2020: énoncé

6.1 Kaléidoscope

Nous allons à partir d'une image faire un kaléidoscope. On sépare l'image en blocs de dimensions identiques et on applique une permutation (une bijection) entre ces blocs. On sépare alors chacun des blocs obtenus en sous-blocs de la même manière pour les permuter également et ainsi de suite pour les sous-sous-blocs. Au bout de suffisamment d'itérations, on retombe sur l'image initiale (voir figure page suivante).

Utilisez comme projet Imagine++ de base celui de l'exercice 2 (avec l'image). Reportez dans le code en commentaire les questions auxquelles vous répondez (`Q1`, `Q2`, etc). Le type `Img` dans le code s'utilise un peu comme une structure (c'est en fait une classe) représentant un tableau bi-dimensionnel de pixels de type `Color`. Pour une variable `img`, le pixel en haut à gauche est Color c = img(0,0);. On peut écrire aussi par img(1,0)=c; (ces deux lignes copient le pixel haut-gauche dans son voisin à droite). La fonction `load` permet de charger un fichier image en mémoire. Pour créer un nouvelle image *ex nihilo*, utiliser `Img img(w,h);` pour une dimension $w \times h$. On peut demander sa taille à une image par img.width() et img.height(). Le pixel en bas à droite est img(img.width()−1,img.height()−1).

Il est plus important de livrer un code clair (commenté et indenté) et qui compile sans warning, même s'il ne répond pas à toutes les questions. Pour cela, vérifiez à chaque étape que votre programme compile et se lance correctement. À la fin de l'examen, nettoyez votre code (indentation...) et vérifiez qu'il compile. Créez alors une archive contenant votre code, le fichier `CMakeLists.txt` et l'image.



(a) image initiale

Figure 4: (b) Application de `miroirh` (Q11) aux itérations 1, 2 et plus tard. (c) Idem pour `miroirv` (Q12). (d) Itérations 1, 2, 3 de `miroirhv` (Q13). (e) `tourne` (Q15) aux itérations 1, 2 et plus tard.

6.2 Outils de base

1. Dans un fichier *séparé*, créer une structure `Bloc` qui sert à désigner un rectangle de l'image (coordonnées entières). Les fonctions de cette section seront placées dans ce fichier.

2. Écrire une fonction `voisinh` qui prend un bloc en argument et renvoie celui juste à sa droite.

3. De même `voisinv` pour le bloc juste en-dessous.

4. Programmer un `operator<=` qui indique si son argument gauche est inclus dans son argument droite.

5. Écrire une fonction `copie` qui prend une image et un bloc pour renvoyer la sous-image correspondante.

6. Écrire une fonction `colle` qui prend une image à la fois source et destination, une image (plus petite) à coller et la position (x, y) du coin haut-gauche de collage.

7. Écrire une fonction `translate` qui prend une image à la fois source et destination, un bloc et le vecteur de déplacement (dx, dy) indiquant où dupliquer ce bloc de l'image (on peut supposer que les zones source et destination ne se recouvrent pas).

8. Sécuriser `copie` et `colle` par des assert vérifiant qu'il n'y a pas de débordement.

9. Écrire une fonction `presente_image` qui affiche l'image en argument et fait une pause d'environ une demi-seconde (remplacer la pause par une attente de clic dans la phase de mise au point du code).

6.3 Effets miroir

10. Écrire une fonction `miroirh` qui prend une image et un bloc et échange les deux moitiés côte à côte (utiliser le copier-coller défini ci-dessus).

11. Écrire une fonction `miroirh` prenant seulement une image et appelant de façon répétée la précédente: on échange les deux moitiés (on présente le résultat), puis les moitiés de chacune (on présente le résultat) et ainsi de suite jusqu'à obtenir l'image initiale inversée horizontalement. Faire une deuxième itération pour retrouver l'image originale. Pour les boucles, utiliser `voisinh` et `operator<=` des blocs.

12. Faire de même avec un miroir vertical.

13. Définir un miroir mixte (horizontal/vertical, fonction `miroirhv`) qui échange les deux moitiés horizontales (présente le résultat), puis chacune des moitiés verticalement (présente le resultat), puis chacun des blocs à nouveau horizontalement et ainsi de suite. On obtient enfin l'image tournée de $180°$. Itérer une deuxième fois pour retrouver l'image initiale. Attention, on a besoin dans ce cas de deux boucles imbriquées pour appliquer l'échange de moitiés sur tous les blocs.

6.4 Rotation

14. Définir une fonction `tourne` prenant une image et un bloc, séparant le bloc en quatre quarts et les faisant tourner. On pourra utiliser le principe que pour passer de la liste (a, b, c, d) à (b, c, d, a), on peut faire: tmp=a; a=b; b=c; c=d; d=tmp;

15. La fonction `tourne` prenant simplement une image fait touner les quatre quarts, puis chacun des quatres quarts de ces derniers, et ainsi de suite jusqu'à obtenir l'image tournée de $90°$. Itérer quatre fois pour retrouver l'image initiale.

6.5 Bonus

16. Les fonctions `tourne` et `miroirhv` requièrent que les images soient carrées de taille 2^n (c'est le cas de l'image *Lena* fournie, 512×512). Pour une image autre, extraire la plus grande sous-image centrée vérifiant ces conditions.

17. Question algorithmique: pourquoi l'implémentation de ces kaléidoscopes par fonction récursive ne donnerait pas l'effet souhaité? (répondre en commentaire dans le code).

18. Pour les ambitieux: plutôt qu'appliquer les permutations élémentaires brutalement, on peut les faire de façon progressive en faisant les translations par étapes (translation de n pixels en n translations de 1 pixel) pour un effet plus fluide. Le plus simple est de garder l'image avant translation, d'en faire une copie, copier les blocs dans l'image initiale pour coller dans la copie. Une fois toutes les translations effectuées, on peut remplacer l'image initiale par la copie finale.