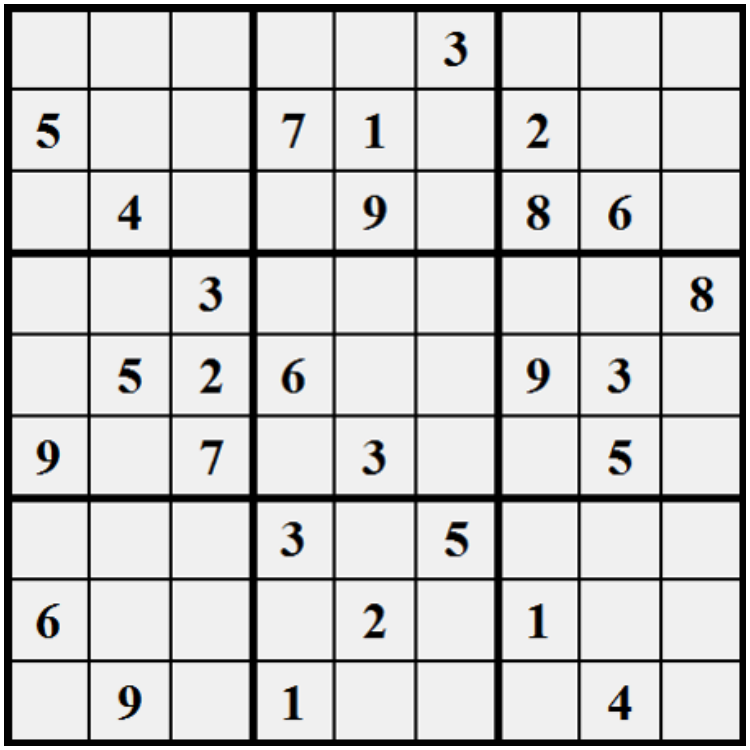
Institut Supérieur
d'Électronique de Paris

TIPE

# Le Rubik's Cube



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | 3 | | | |
| 5 | | | 7 | 1 | | 2 | | |
| | 4 | | | 9 | | 8 | 6 | |
| | | 3 | | | | | | 8 |
| | 5 | 2 | 6 | | | 9 | 3 | |
| 9 | | 7 | | 3 | | | 5 | |
| | | | 3 | | 5 | | | |
| 6 | | | | 2 | | 1 | | |
| | 9 | | 1 | | | | 4 | |

Laurent Tainturier & Alphonse Terrier

supervisé par
**M. Patrick COUVEZ**

**2016-2017**

# Table des matières

# Introduction

# Chapitre 1

# Présentation du sudoku

# Chapitre 2

# Électronique

# Chapitre 3

# Mécanique

# Chapitre 4

# Informatique

Tous les algorithmes sont implémentés en Python. Ils sont disponibles en annexe.

## 4.1 Résolution du sudoku

## 4.2 Reconnaissance du sudoku

## 4.3 Écriture et contrôle des moteurs

# Annexe A

# Fichier principal

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
import time
import threading

import lcd
import save
import numpy as np
import button as bt
import camera as cm
import resolution as rs


class Sudoku:
    def __init__(self):
        self.number = 0
        self.errors = []
        self.taille = (3, 3)
        self.power = True
        self.power_mode = True
        self.select_mode = None
        self.modes = ["Automatic", "Practice"]
        self.mode = self.modes[self.number]
        self.nb_cases = self.taille[0] * self.taille[1]
        self.sudoku = np.zeros((self.nb_cases, self.nb_cases), int)
        self.liste_position = []
        self.methode_resolution = "Backtracking"

        self.stop = Stop(self)
        self.camera = cm.Camera(self)
        self.resolution = rs.Resolution(self)
        self.button = bt.Buttons([11, 13, 15])

    def start(self):
        while self.power:
            lcd.write("Sudoku Plotter Welcome!")
            bt_pressed = self.wait(5)
            if bt_pressed[2]:
                self.quit()
            elif bt_pressed[1]:
                self.button.pressed = [0, 0, 0]
                lcd.write("Choose the mode")
                lcd.write(self.mode + "?", 2)
                time.sleep(4)
                while self.power_mode:
                    if self.button.pressed[0]:
                        self.button.pressed[0] = 0
                        self.number = 1 - self.number
                        self.mode = self.modes[self.number]
                        lcd.write(self.mode + "?", 2)
                        time.sleep(2)

```

```
55                      if self.button.pressed[1]:
56                          self.button.pressed[1] = 0
57                          self.select_mode = self.mode
58
59                      if self.button.pressed[2]:
60                          self.button.pressed[2] = 0
61                          self.quit()
62
63                      if self.select_mode == "Automatic":
64                          lcd.write("Sudoku Plotter Automatic Mode")
65                          time.sleep(2)
66                          self.stop.run()
67                          self.automaticMode()
68
69                      if self.select_mode == "Practice":
70                          lcd.write("Sudoku Plotter Practice Mode")
71                          time.sleep(2)
72                          self.practiceMode()
73
74                      if self.select_mode:
75                          self.select_mode = None
76                          self.start()
77
78                      time.sleep(1)
79              time.sleep(1)
80
81          lcd.write("Sudoku Plotter Goodbye!")
82          time.sleep(2)
83          self.power = False
84          self.stop.power = False
85          self.button.stop()
86          os.system("sudo shutdown -h now")
87
88      def automaticMode(self):
89          lcd.write("Photo Capture... Please Wait!")
90          self.camera.takePhoto()
91          if "camera_error" not in self.errors:
92              lcd.write("Succeeded!", 2)
93              time.sleep(2)
94          else:
95              lcd.write("Camera isn't connected!")
96              time.sleep(3)
97              lcd.write("Camera error Continue?")
98              time.sleep(4)
99              if self.button.pressed[2]: self.start()
100         lcd.write("Recognition...")
101         lcd.write("Please Wait!", 2)
102         os.system("sudo python extraction.py")
103         lcd.write("Succeeded!", 2)
104         time.sleep(2)
105         lcd.write("Solving... Please Wait!")
106         time.sleep(0.5)
107         self.sudoku = save.readSudoku()
108         self.sudoku, self.liste_position = self.resolution.start(self.sudoku, self.methode_resolution)
109         if "sudoku_insoluble" not in self.errors:
110             lcd.write("Succeeded!", 2)
111             print("finished /n", self.sudoku)
112             time.sleep(2)
113             lcd.write("Sudoku Plotter completed!", 2)
114
115         if self.errors and self.errors != ["camera_error"]:
116             lcd.write("Failed!", 2)
117             self.wait()
118             self.errors = []
119             self.start()
120
121     def practiceMode(self):
122         pass
123
124     def wait(self, sleep=2):
125         previous_time = time.time()
126         while time.time() - previous_time < sleep:
127             if self.button.pressed is not None:
```

```
                bt = self.button.pressed
                self.button.pressed = [0, 0, 0]
                return bt
            time.sleep(0.1)
        return [0, 0, 0]

    def setError(self, error):
        if error not in self.errors:
            self.errors.append(error)

    def quit(self):
        lcd.write("Sudoku Plotter Goodbye?")
        time.sleep(2)
        bt_pressed = self.wait(4)
        if bt_pressed[1] or bt_pressed[2]:
            self.power = False
            self.power_mode = False
            self.start()


class Stop(threading.Thread):
    def __init__(self, boss):
        threading.Thread.__init__(self)
        self.power = True
        self.boss = boss

    def start(self):
        while self.power:
            if self.boss.button[2]:
                self.boss.power = False
                self.boss.power_mode = False
                self.boss.start()
                self.power = False

"""
    def start(self):
        lcd.write("Sudoku Plotter Welcome!")
        time.sleep(2)
        while self.power:
            lcd.write("Sudoku Plotter Welcome!")
            print("Welcome")
            if self.button.pressed == 0:
                self.button.pressed = None
                self.number = 1 - self.number
                self.mode = self.modes[self.number]

            if self.button.pressed == 1:
                self.button.pressed = None
                self.select_mode = self.mode

            if self.button.pressed == 2:
                self.button.pressed = None
                lcd.write("Goodbye?", 2)
                while not self.button.pressed:
                    time.sleep(0.1)
                if self.button.pressed == 1:
                    self.power = False
                self.button.pressed = None

            while self.select_mode == "automatic":
                lcd.write("Automatic?", 2)
                if self.button.pressed == 2:
                    self.select_mode = None
                    self.button.pressed = None

            while self.select_mode == "practice":
                lcd.write("Practice?", 2)
                if self.button.pressed == 2:
                    self.select_mode = None
                    self.button.pressed = None

            time.sleep(2)
        print("Bye!")
```

```
"""

lcd = lcd.LCD()
sudoku = Sudoku()
sudoku.button.start()
sudoku.start()

"""
import resolution as rs
import camera as cm
import step_motor as stp
import os

lcd = lcd.LCD()
lcd.write("Sudoku Plotter Welcome!")
time.sleep(2)
lcd.write("Automatic?", 2)
time.sleep(2)
lcd.write("Practice?", 2)
time.sleep(2)
lcd.write("Step by step?", 2)
time.sleep(2)
lcd.write("Photo capture in progress...")
time.sleep(2)
lcd.write("Use the joystick to move or write")
time.sleep(2)
lcd.write("Joystick isn't connected!")
time.sleep(2)
lcd.write("Recognition...")
lcd.write("in progress...", 2)
time.sleep(2)
lcd.write("Failed!", 2)
time.sleep(2)
lcd.write("Succeeded!", 2)
time.sleep(2)
lcd.write("Solving...")
time.sleep(2)
lcd.write("Sudoku Plotter Goodbye!")
"""
```

# Annexe B

# Script de résolution des sudokus

```python
#!/usr/bin/env python3

import numpy as np
from time import time


class Resolution:
    """
    Classe permettant de résoudre un sudoku grâce à différentes méthodes, à savoir :
        - inclusion
        - exclusion
        - bactracking
        - ...
    Si le sudoku n'est pas résoluble, lève une erreur
    """

    def __init__(self, boss):
        self.boss = boss
        self.taille = self.boss.taille
        self.nb_cases = self.boss.nb_cases
        self.methode_resolution = None
        self.vitesse = None
        self.sudoku = np.zeros((self.nb_cases, self.nb_cases), int)
        self.liste_sudoku = []
        self.liste_position = []
        self.power = True

    def start(self, sudoku, methode, vitesse=None):
        self.sudoku = sudoku
        self.methode_resolution = methode
        self.vitesse = vitesse
        print(self.methode_resolution)
        zero_time = time()
        if not self.checkBeforeStart():
            self.boss.setError("sudoku_insoluble")
            return self.sudoku, []
        if self.methode_resolution == "Globale": self.optimale()
        elif self.methode_resolution == "Inclusion": self.inclusion()
        elif self.methode_resolution == "Exclusion": self.exclusion()
        elif self.methode_resolution == "Backtracking":
            self.createListe()
            self.backTracking()
        else: print("La méthode n'est pas reconnue")
        print(time() - zero_time, '\n')
        return np.copy(self.sudoku), self.liste_position

    def checkBeforeStart(self):
        for i in range(self.nb_cases):
            liste_ligne = []
            liste_colonne = []
            liste_carre = []
            x = 3 * (i // 3)
            y = 3 * (i % 3)
            for j in range(self.nb_cases):
```

```
                if self.sudoku[x + j // 3, y + j % 3] in liste_carre \
                        or self.sudoku[j, i] in liste_colonne \
                        or self.sudoku[i, j] in liste_ligne:
                    return False
                if self.sudoku[i, j] != 0:
                    liste_ligne.append(self.sudoku[i, j])
                if self.sudoku[j, i] != 0:
                    liste_colonne.append(self.sudoku[j, i])
                if self.sudoku[x + j // 3, y + j % 3] != 0:
                    liste_carre.append(self.sudoku[x + j // 3, y + j % 3])
        return True

    def createListe(self):
        self.liste_position = []
        liste_tailles_cases_vides = []
        liste_tailles = [[] for i in range(self.nb_cases - 1)]
        for x in range(self.nb_cases):
            for y in range(self.nb_cases):
                if not self.sudoku[x][y]:
                    self.liste_position.append((x, y))
                    liste_tailles_cases_vides.append(len(self.checkListe(x, y)))
                    liste_tailles[liste_tailles_cases_vides[-1] - 2].append((x, y))
        liste_position = []
        for i in range(self.nb_cases - 1):
            liste_position += liste_tailles[i]
        return liste_position

    def checkListe(self, x, y):
        """
        Renvoie la liste des valeurs possibles pour la case de coordonnées x et y
        :param x: int: ligne
        :param y: int: colonne
        :return: liste: list
        """
        liste = []
        if self.sudoku[x][y] == 0:
            liste = [i + 1 for i in range(self.nb_cases)]
            block_x = x - x % self.taille[0]
            block_y = y - y % self.taille[1]
            for i in range(self.nb_cases):
                if self.sudoku[x, i] in liste:
                    liste.remove(self.sudoku[x, i])
                if self.sudoku[i, y] in liste:
                    liste.remove(self.sudoku[i, y])
                if self.sudoku[block_x + i % self.taille[1], block_y + i // self.taille[0]] in liste:
                    liste.remove(self.sudoku[block_x + i % self.taille[1], block_y + i // self.taille[0]])
        return liste

    def optimale(self):
        self.liste_position = self.createListe()
        size = []
        for x, y in self.liste_position:
            size.append(len(self.checkListe(x, y)))
        self.backTracking()

    def inclusion(self):
        pass

    def exclusion(self):
        pass

    def backTracking(self):
        """
        Résoud un sudoku selon la méthode de backtracking
        :return: None or -1
        """
        self.liste_sudoku = []
        i = 0
        while self.power:
            x, y = self.liste_position[i]
            liste = self.checkListe(x, y)
            if liste:
                self.sudoku[x][y] = liste.pop(0)
```

```
128                 self.liste_sudoku.append(liste)
129                 i += 1
130             else:
131                 while not liste:
132                     i -= 1
133                     x, y = self.liste_position[i]
134                     try:
135                         liste = self.liste_sudoku[i]
136                     except IndexError:
137                         self.sudoku = self.boss.sudoku
138                         self.boss.setError("sudoku_insoluble")
139                         return -1
140                     if liste:
141                         self.sudoku[x][y] = liste.pop(0)
142                         self.liste_sudoku[i] = liste
143                         i += 1
144                         break
145                     else:
146                         self.liste_sudoku.pop(i)
147                         self.sudoku[x][y] = 0
148             if i < len(self.liste_position):
149                 self.power = False
150         self.power = True
151
152
153
154 if __name__ == '__main__':
155     class Boss:
156         def __init__(self):
157             self.taille = (3, 3)
158             self.nb_cases = 9
159             self.methode = 'Globale'
160             self.vitesse = 'Pas à Pas'
161             self.sudoku = np.array([[0, 0, 0, 0, 0, 0, 0, 1, 2],
162                                     [0, 0, 0, 0, 0, 0, 0, 0, 3],
163                                     [0, 0, 2, 3, 0, 0, 4, 0, 0],
164                                     [0, 0, 1, 8, 0, 0, 0, 0, 5],
165                                     [0, 6, 0, 0, 7, 0, 8, 0, 0],
166                                     [0, 0, 0, 0, 0, 9, 0, 0, 0],
167                                     [0, 0, 8, 5, 0, 0, 0, 0, 0],
168                                     [9, 0, 0, 0, 4, 0, 5, 0, 0],
169                                     [4, 7, 0, 0, 0, 6, 0, 0, 0]])
170             self.Resolution = Resolution(self)
171             self.Resolution.start(self.sudoku, self.methode, self.vitesse)
172             print(self.sudoku, '\n')
173
174         def setError(self, error):
175             if error == "sudoku_insoluble":
176                 print("Le sudoku n'est pas résoluble")
177
178
179     Boss()
```

# Annexe C

# Script de gestion de la caméra

```python
#!/usr/bin/env python3


class Camera:
    """
    Permet la gestion de la camera de la raspberry pi
    Si celle-ci n'est pas disponible ou le module 'picamera'
    n'a pas été installé correctement, lève une exception.
    """

    def __init__(self, boss):
        self.boss = boss
        self.camera = None
        self.tryError()

    def tryError(self):
        try:
            import picamera
            self.camera = picamera.PiCamera()
        except:
            self.boss.setError("camera_error")

    def takePhoto(self):
        try:
            self.camera.capture("Images/photos.jpg")
            print("The photo has been taken")
        except:
            self.boss.setError("camera_error")


if __name__ == '__main__':
    class Boss:
        def setError(self, error):
            if error == "module_camera":
                print("Le module 'picamera' n'a pas été installé correctement !")
            if error == "disponibilite_camera":
                print("La caméra n'est pas disponible !")

    Camera = Camera(Boss())
    Camera.takePhoto()
```