Chessbot with Computer Vision

En este Notebook de Jupyter mostraremos el funcionamiento por partes de manera simple e interactiva de nuestro bot de ajedrez basado en visión por computador. Para ilustrar adecuadamente y de forma amena y comprensible las entrañas del programa dividiremos la explicación en los siguientes apartados:

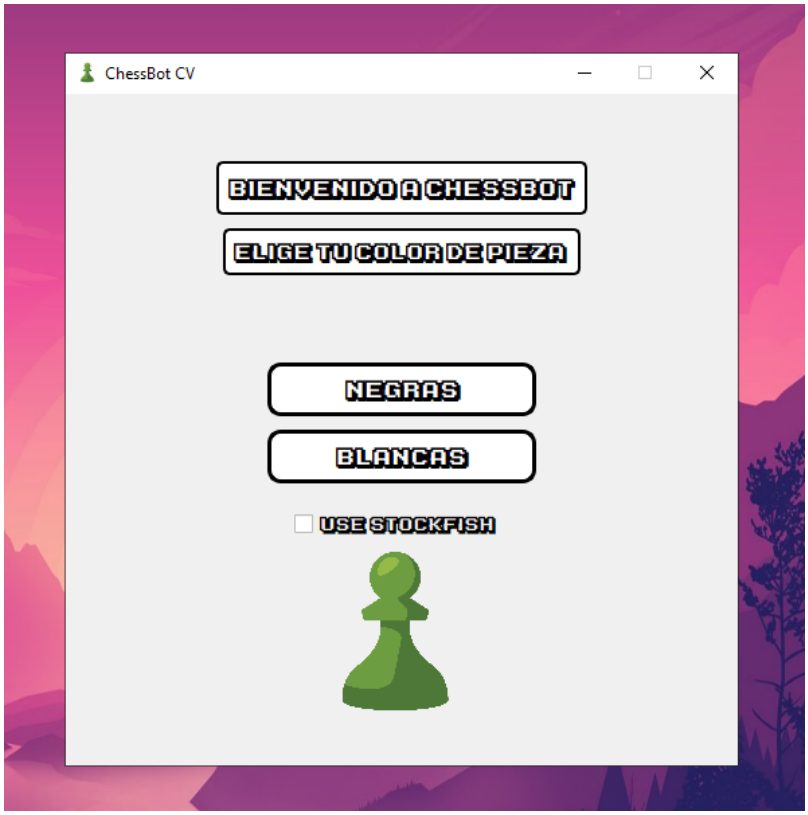
1. Captura de pantalla
2. Reconocimiento del tablero
3. Análisis del mejor movimiento
4. Movimiento automático por ordenador
5. Conclusiones
6. Demo del proyecto

Todo el código puede encontrarse en el siguiente enlace en github completo y con su documentación.

0. Prompt al usuario

Un paso 0, muy trivial pero esencial para nuestro analizador, es preguntar al usuario con que piezas va a jugar la partida (blancas o negras), ya que es imposible determinar, dado una configuración aleatoria de tablero sin esta información, que piezas estamos jugando. También damos la opción de usar StockFish (un potente motor de ajedrez ya creado) o nuestra propia implementación.

Para este paso hemos usado la libreria **Pyqt5** la cual nos permite crear interfaces de usuario gráficas (GUI) que interactuen con nuestro código.



0.1. Motivación del proyecto

La motivación que obtenemos para este proyecto es ver las capacidades de interacción de la visión por computador con aplicaciones virtuales como juegos. Particularmente el ajedrez nos gusta jugarlo, y pensamos que podría ser divertido y podríamos aprender mucho de diseñar un proyecto de este tipo.

1. Captura de pantalla

El primer paso que debemos realizar si queremos que nuestro bot juegue al ajedrez de forma autónoma es alguna forma de darle información primitiva, que en este caso serán capturas de lo que estamos viendo en el ordenador y con ello procesar las imágenes y transformarlas en información útil.
Para realizar este paso usaremos la librería de PyAutoGUI, la cual nos permitirá tomar capturas de pantalla, pero también identificar las ventanas activas y maximizarlas o minimizarlas.
Dicho esto, el proceso que realizamos en este apartado es el siguiente:

1. Obtendremos una lista de las ventanas abiertas
2. Comprobamos que Chess.com esté abierto en alguna de ellas y no en más de una, puesto que de lo contrario no estaríamos jugando una partida
3. Una vez localizada la venta procedemos primero a minimizarla y luego a maximizarla para tenerla en pantalla completa
4. Finalmente comienza el bucle en el que se va tomando esporádicamente una captura de lo que hay en pantalla y toca procesarla

2. Reconocimiento del tablero

Este es uno de los pasos esenciales de nuestra aplicación. A través de distintos métodos y algoritmos y con ayuda de la librería **opencv** conseguiremos analizar una imagen que contenga un tablero de la pagina de Chess.com y "virtualizar" la información que vemos por pantalla a otra que podamos manejar nosotros para posteriormente proceder al calculo de las jugadas.

Para transmitir de forma sencilla el funcionamiento del programa analizaremos uno por uno los métodos más importantes que nos van a ir permitiendo procesar la imagen.

Paso 1

Todas las imagenes que recopila nuestra capturadora del apartado 1 pasan por la clase **ChessboardAnalizer** , la cual se encarga en primer lugar de localizar de forma precisa el tablero en la pantalla.

Supongamos que la imagen de arriba es una captura cualquiera de una partida de ajedrez, observamos que hay mucha información que no necesitamos (basicamente todo aquello que no sea el tablero), por lo que nuestro objetivo será recortar la imagen para acabar solo con el tablero, el cual contiene la información relevante. Nuestra aproximación a este problema es la siguiente:

In [1]:
```
import cv2 as cv
import numpy as np
import os
import math
```

```
import matplotlib.pyplot as plt
import matplotlib
import imutils
import io
import pyautogui
import chess
import chess.svg
from ipywidgets import interactive, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)
```

In [2]:
```
def getContours(board_image):

        #Convertir a escala de grises la imagen
        gray = cv.cvtColor(board_image, cv.COLOR_BGR2GRAY)
        #Calculamos el threshold adaptativo
        th = cv.adaptiveThreshold(gray,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,cv.THRE

        #Printeamos la imagen
        plt.subplot(2,2,1)
        plt.title("Image con Threshold")
        plt.imshow(cv.cvtColor(th, cv.COLOR_BGR2RGB))

        #Encontramos los contornos de la imagen
        contours, _ = cv.findContours(th, cv.RETR_TREE,cv.CHAIN_APPROX_SIMPLE)
        return contours
```

Esta primera función se encarga de extraer los contornos de la imagen. La idea es ver todos los posibles contornos cerrados en la imagen, y aquel con mayor perímetro y con forma cuadrada debe ser el de nuestro tablero. Por ello, primero pasamos la imagen a gris y posteriormente aplicamos un threshold adaptativo que diferencie perfectamente cada cuadrado para posteriormente aplicar la funcion **findContours**.

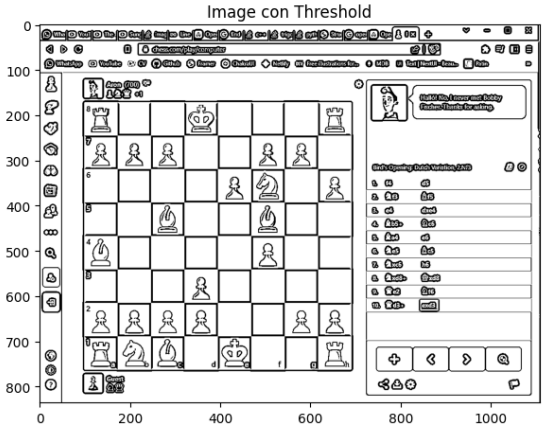
In [3]:
```
def findBoard(board_image,contours):
        #Suma máxima del tamaño de los contornos (width + height)
        maxSum = 0
        #Aquí almacenamos el contorno más grande en perímetro/2
        contornoMax = []
        #Variables para luego cropear la imagen
        xSquare, ySquare, wSquare, hSquare = 0, 0, 0, 0
        #Iteramos los contornos
        for cnt in contours:
            approx = cv.approxPolyDP(cnt, 0.05 * cv.arcLength(cnt, True), True)
            if len(approx) == 4:
                x, y, w, h = cv.boundingRect(cnt)
                ratio = float(w) / h

                if w + h > maxSum and 0.9 <= ratio <= 1.1:  # Buscamos el cuadra
                    maxSum = w + h
                    contornoMax = cnt
                    xSquare, ySquare, wSquare, hSquare = x, y, w, h  # Guardamos

        # Recorte de la imagen
        cropped_chessboard = board_image[ySquare+5:ySquare + hSquare -5,
                                         xSquare+5:xSquare +
                                         wSquare - 5]
        #Dibujamos el contorno detectado
        board_image = cv.drawContours(board_image, [contornoMax], -1, (0, 255, 0
```

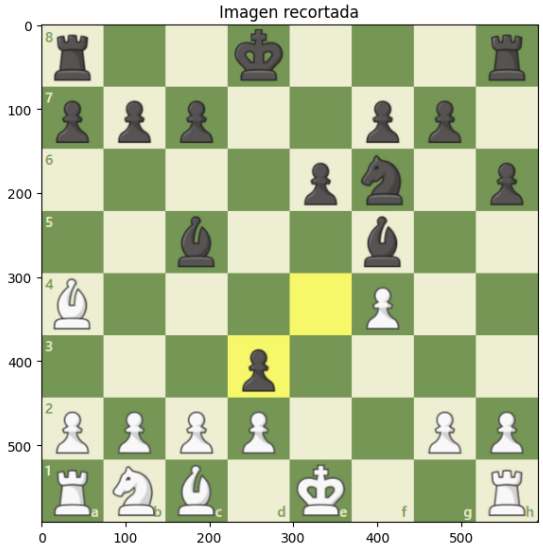
```
    return cropped_chessboard
```

Una vez tenemos todos los contornos en un array podemos empezar a iterar entre ellos y ver cuantos de ellos tienen 4 lados y el ratio entre la altura y el ancho los hace aproximadamente un cuadrado. Con aquellos contornos que pasan este filtro tratamos de encontrar el que tiene el perímetro máximo y finalmente con aquel que tiene el perímetro máximo podemos extraer su posición y usarla para recortar la imagen inicial y obtener únicamente el tablero.

In [4]:
```
image_raw = cv.imread('images/CapturaRaw.png',cv.IMREAD_UNCHANGED)

contours = getContours(image_raw)
cropped_board = findBoard(image_raw,contours)

plt.subplot(2,2,2)
plt.title("Imagen original")
plt.imshow(cv.cvtColor(image_raw, cv.COLOR_BGR2RGB))

plt.subplot(2,2,3)
plt.title("Imagen recortada")
plt.imshow(cv.cvtColor(cropped_board, cv.COLOR_BGR2RGB))
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x276831d4b50>
```





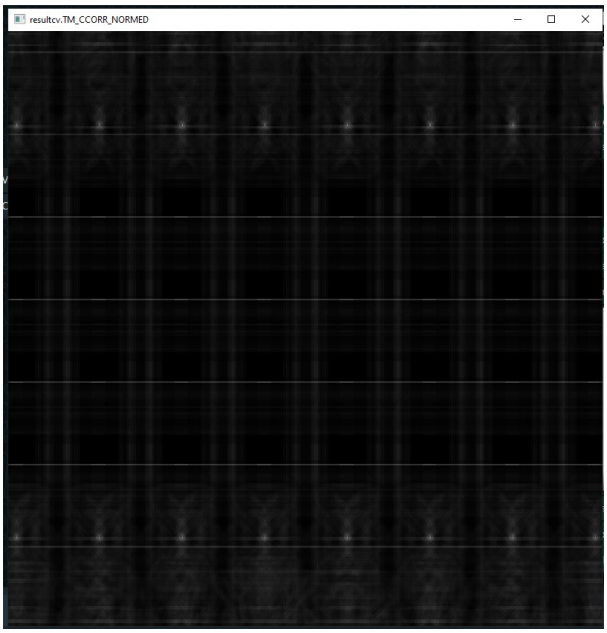
¡Ahí la tenemos! Como vemos nuestro algoritmo consigue detectar y recortar sin ningún problema el tablero. Esto nos facilita en gran medida el reconocimiento de piezas que ahora veremos.

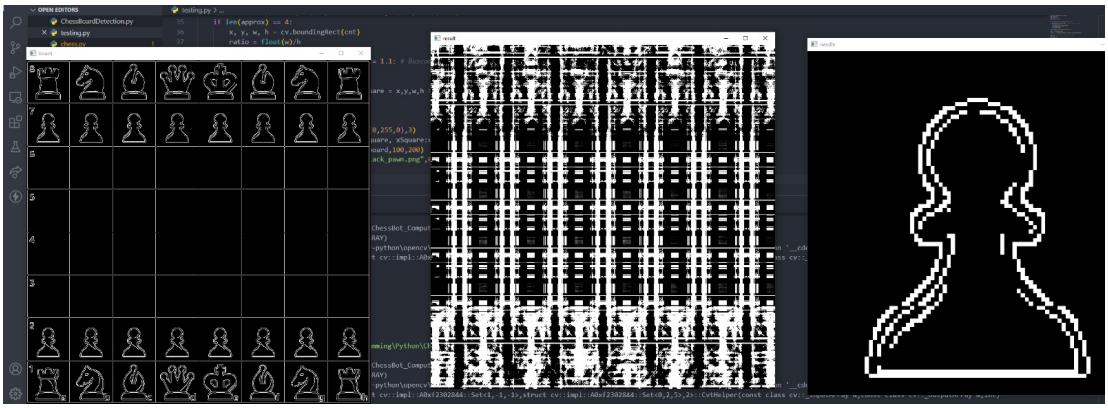
Es importante remarcar que este no es la primera ocurrencia que tuvimos para resolver este problema, probamos anteriormente a aplicar otros algoritmos como **Hough** o **NCC**, sin embargo, estos no resultaron adecuados para esta aplicación y no daban buenos resultados. La actual implementación utiliza el algoritmo de contornos de Suzuki, proporcionado por opencv, para el cual se puede encontrar una detallada explicación aquí.

Paso 2

En este paso nuestro objetivo será identificar todas las piezas presentes, así como los espacios vacíos para poder virtualizar correctamente el tablero.

Nuestra primera aproximación a este problema fue usar NCC directamente aplicado al tablero entero con las imagenes de las distintas piezas recortadas para ver donde se localizaban los puntos máximos y así conseguir hacer una correspondencia.





En la imagen de arriba hemos puesto una prueba que realizamos intentando detectar los peones, y se puede observar que aperecen puntos blancos (valores máximos) en los

lugares donde deberían estar los peones en la configuración inicial del tablero. Sin embargo, esta método hacia muy dificil distinguir si no habia ninguna pieza (ya que siempre ibamos a encontrar un valor máximo por muy "pequeño" que fuese) y además era poco precisa la predicción del tipo de pieza de esta forma.

Finalmente optamos por usar NCC pero fue necesario hacerlo invariante a escala, debido a que en función a la resolución de pantalla, el template fijo que usabamos no detectaba bien las piezas con distintos tamaños de tablero.

Para no hacer la explicación muy extensa lo mejor será ir paso a paso explicando cada una de las funciones del código.

In [5]:
```
def divideSquares(cropped_board):

        img_width = cropped_board.shape[0] #Anchura del tablero
        square_size = int(img_width / 8)  # Tamaño del lado del cuadrado

        squares_arr = [] #Array que contenga las posiciones de arriba izquierda

        #Añadimos las posiciones en la pantalla de los cuadrados
        for f in range(8):
            for c in range(8):
                squares_arr.append(
                    (int(f * square_size), int(c * square_size)))

        return squares_arr, square_size
```

En primer lugar dividimos nuestro tablero en 64 cuadrados con la función **divideSquares()** , la cual nos devuelve el tamaño y posiciones, en la imagen del tablero, de cada una de las celdas.

In [6]:
```
squares_arr, square_size = divideSquares(cropped_board)
print("Array de celdas: ", squares_arr)
print("Tamaño de la celda: ", square_size , " pixeles")
```

```
Array de celdas:  [(0, 0), (0, 74), (0, 148), (0, 222), (0, 296), (0, 370), (0,
444), (0, 518), (74, 0), (74, 74), (74, 148), (74, 222), (74, 296), (74, 370),
(74, 444), (74, 518), (148, 0), (148, 74), (148, 148), (148, 222), (148, 296),
(148, 370), (148, 444), (148, 518), (222, 0), (222, 74), (222, 148), (222, 22
2), (222, 296), (222, 370), (222, 444), (222, 518), (296, 0), (296, 74), (296,
148), (296, 222), (296, 296), (296, 370), (296, 444), (296, 518), (370, 0), (37
0, 74), (370, 148), (370, 222), (370, 296), (370, 370), (370, 444), (370, 518),
(444, 0), (444, 74), (444, 148), (444, 222), (444, 296), (444, 370), (444, 44
4), (444, 518), (518, 0), (518, 74), (518, 148), (518, 222), (518, 296), (518,
370), (518, 444), (518, 518)]
Tamaño de la celda:  74  pixeles
```

Teniendo en memoria las posiciones de las 64 celdas nuestra idea fue la siguiente; podemos ir iterando por todas las celdas del tablero y mirar con los distintos templates que tenemos de las piezas a ver cual de estos nos da la mejor correspondencia y asignar así a la celda la pieza que nos diese el mayor valor NCC.

Sin embargo hay que añadir algunos elementos a esta idea para que resulte efectiva:

1. Detectar primero si el cuadrado está vacío

2. Añadir invarianza a escala a la función **matchTemplate()** de opencv
3. Distinguir si la pieza es negra o blanca

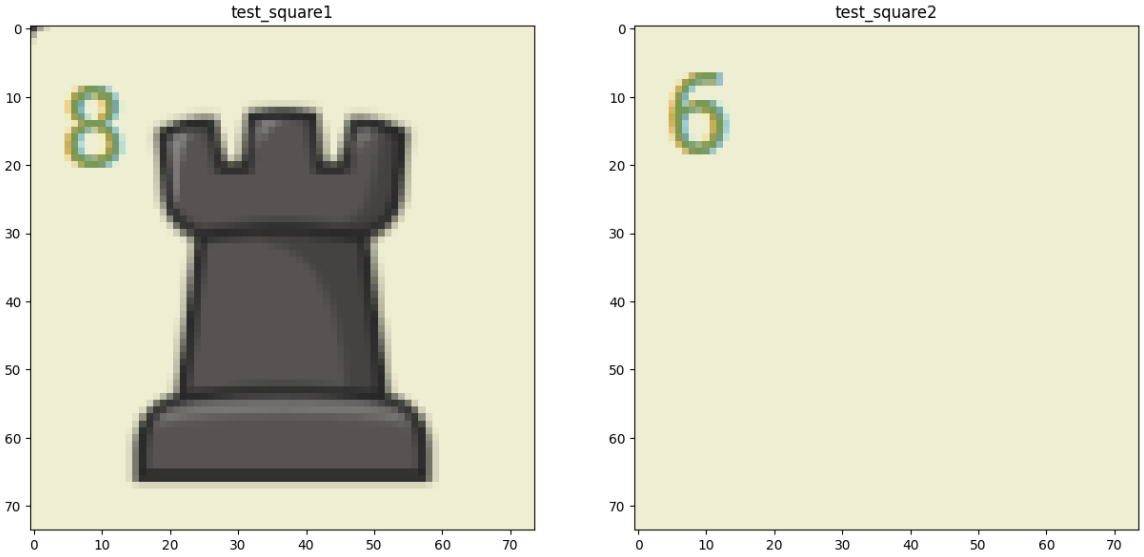
Detectar cuadrados vacios

In [7]:
```
def isBlankSquare(square):

    image = cv.cvtColor(square,cv.COLOR_BGR2RGB)

    flattened_img = image.reshape((-1,3))
    result = np.unique(flattened_img, axis=0, return_counts = True)
    image_size = len(flattened_img)
    ocurrences = result[1]
    max_ocurrence = np.amax(ocurrences)

    return max_ocurrence/image_size > 0.8
```

Como indica el nombre de la función, esta se encarga de detectar si una celda está vacía. Para ello simplemente crea un diccionario de todos los posibles colores en la imagen (combinaciones RGB) y calcula su frecuencia. Tras calcular este "histograma", concluimos en que si hay un tipo de color que represente a más del 80% de los pixeles de la imagen, entonces nos encontramos ante una celda sin pieza. Este umbral es importante definirlo ya que pueden existir celdas con números o algunas indicaciones propias del juego que no la hagan perfectamente uniforme.

In [8]:
```
#Tomamos 3 cuadrados de referencia distintos
test_square1 = cropped_board[square_size*0:square_size*0+square_size,square_size
test_square2 = cropped_board[square_size*2:square_size*2+square_size,square_size
test_square3 = cropped_board[square_size*7:square_size*7+square_size,square_size

plt.subplot(1,2,1)
plt.title("test_square1")
plt.imshow(cv.cvtColor(test_square1, cv.COLOR_BGR2RGB))

plt.subplot(1,2,2)
plt.title("test_square2")
plt.imshow(cv.cvtColor(test_square2, cv.COLOR_BGR2RGB))

print("isBlankSquare(test_square1): ", isBlankSquare(test_square1))
print("isBlankSquare(test_square2): ", isBlankSquare(test_square2))
```

```
isBlankSquare(test_square1):  False
isBlankSquare(test_square2):  True
```

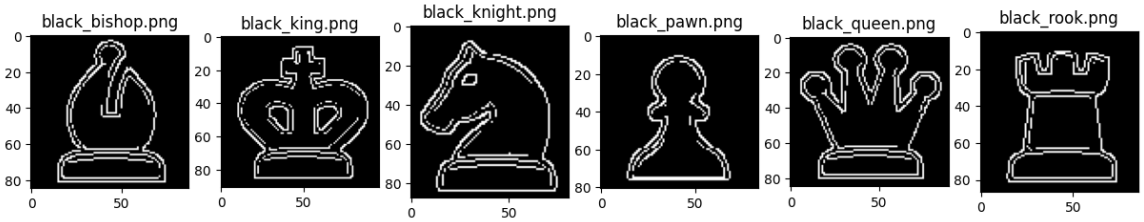


Con esto eliminamos también buena parte del tiempo de ejecución que necesitará nuestro método personalizado de matchTemplate(), al descartar muy rapidamente buena parte de los cuadrados.

Invarianza a escala para matchTemplate()

Comencemos primero explicando como detectamos que pieza hay en cada celda. En primer lugar, para hacer los matches más sencillos no utilizamos una imagen normal, si no que primero aplicamos Canny para extraer la información de los bordes, y así posteriormente "comparar" la imagen con unos templates personalizados a los que también previamente hemos aplicado Canny, resultando en los siguientes:

In [9]:
```
directory = "images/template_images"

column = 1

for filename in os.listdir(directory):
    template = cv.imread(directory + "/" + filename,cv.IMREAD_UNCHANGED)
    plt.subplot(1,6,column)
    plt.title(filename)
    plt.imshow(cv.cvtColor(template, cv.COLOR_BGR2RGB))
    column += 1
```



Realmente solo necesitamos 6 templates, porque las figuras son practicamente idénticas al realizar Canny con ambos colores, además de esta forma solo comparamos 6 veces en lugar de 12, a coste de posteriormente tener que identificar el color de cada pieza (pero esta operación no es ni mucho menos tan costosa).

Al probar estos templates nos encontramos con el siguiente problema; en diferentes resoluciones de pantalla o con la ventana del navegador en tamaños pequeños los

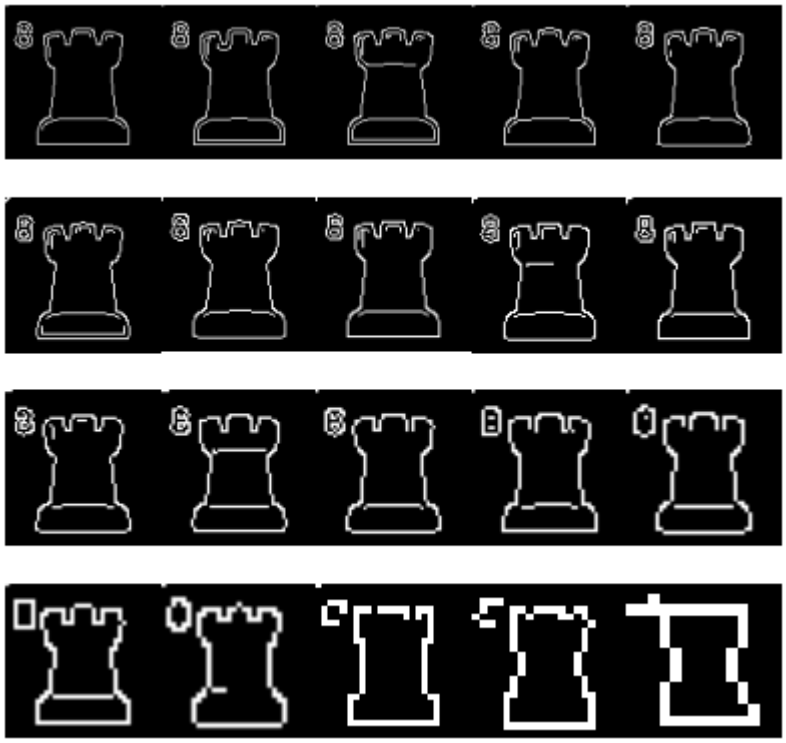
matches son muy malos. Esto se debe a que el método matchTemplate no es invariante a escala, por lo que para solucionarlo ideamos la siguiente función:

In [10]:
```
def getBestScaleMatch(original,template):

    (tH, tW) = template.shape[:2]

    found = None
    bestMatch = 0.0

    for scale in np.linspace(0.2, 1.6, 20)[::-1]:
        #reescalamos la imagen original
        resized = imutils.resize(original, width = int(original.shape[1] * scale
        #calculamos ratio
        r = original.shape[1] / float(resized.shape[1])

        #Si el template es más grande que la imagen cambiada de tamaño no nos si
        if resized.shape[0] < tH or resized.shape[1] < tW:
            break

        #Aplicamos Canny a la imagen original
        original_edged = cv.Canny(resized,100,200)
        result = cv.matchTemplate(original_edged, template, cv.TM_CCORR_NORMED)
        _, maxVal, _, maxLoc = cv.minMaxLoc(result) #Precision

        #Maximización de los resultados
        if found is None or maxVal > found[0]:
            found = (maxVal, maxLoc, r)

        if maxVal > bestMatch:
            bestMatch = maxVal

    return bestMatch
```

Esta función lo que hace es escalar la imagen original en distintas dimensiones y compararla con una plantilla para ver si obtenemos un mejor match al variarla ya que nuestras imagenes no siempre van a tener las mismas dimensiones que aquellas que vengan del tablero que estamos capturando.

Veamos un pequeño ejemplo de su funcionamiento:

In [11]:
```
c = 1

for scale in np.linspace(0.2, 1.6, 20)[::-1]:
    #reescalamos la imagen original
    resized = imutils.resize(test_square1, width = int(test_square1.shape[1] * s
    original_edged = cv.Canny(resized,100,200)
    plt.subplot(4,5,c)
    plt.axis('off')
    plt.imshow(original_edged, cmap="gray")
    plt.gcf().set_size_inches(5, 5)
    c += 1

plt.subplots_adjust(wspace=0, hspace=0)
```



Esta sería el escalado de la **imagen original** que le proporcionamos a la función y que utiliza para intentar asemejarse lo máximo posible a la **plantilla**.

Distinguir entre piezas blancas o negras

Por último, si hemos detectado una pieza, nos queda la duda de si esta es blanca o negra, por lo que necesitamos de una función que nos diga el color de esta:

In [12]:
```
def isPieceWhite(cropped_square):
    s_window = 5
    #Pasamos a gris
    cropped_square = cv.cvtColor(cropped_square,cv.COLOR_BGR2GRAY)

    w = cropped_square.shape[0]
    h = cropped_square.shape[1]
    center = (int(w/2),int(h/2))
    #Cogemos una unica ventana en el centro de la imagen que es donde va a estar
    _, binarized_square = cv.threshold(cropped_square,127,255,cv.THRESH_BINARY)
    cropped_square = binarized_square[center[1]+int(h/3.5):center[1]+int(h/3),ce


    whitePixels = 0
    flattened_img = np.array(cropped_square).flatten()
    for pixel in flattened_img:
        #100 es el threshold de intensidad para considerar pixel blanco
        if pixel > 100: whitePixels = whitePixels +1

    #Si hay una proporcion grande de pixeles blancos (la ventana quizas se ha de
    return whitePixels > int(flattened_img.shape[0]/1.5)
```

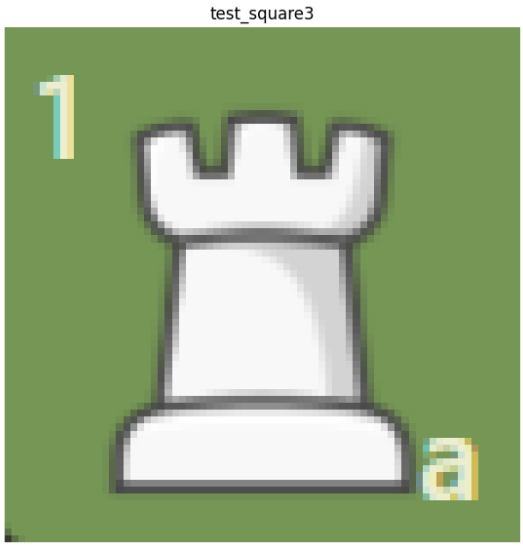
La función puede parecer más complicada de lo que es, pero en realidad lo que hace es mirar una pequeña ventana por el centro de la imagen (la cual previamente ha binarizado) y cuenta la cantidad de pixeles blancos y negros. Si finalmente la mayor

parte de los píxeles contados son blancos, la pieza resultante es blanca y de igual forma con las piezas negras. Podría haberse hecho mirando un unico pixel binarizado de la imagen pero como cada pieza es distinta en forma es posible que ese pixel no estuviese siempre dentro de la pieza, por ello es la razón de usar una ventana pequeña.

In [13]:
```
print("isPieceWhite(test_square1): ", isPieceWhite(test_square1))
print("isPieceWhite(test_square3): ", isPieceWhite(test_square3))

plt.subplot(1,2,1)
plt.axis('off')
plt.title("test_square2")
plt.imshow(cv.cvtColor(test_square1, cv.COLOR_BGR2RGB))

plt.subplot(1,2,2)
plt.axis('off')
plt.title("test_square3")
plt.imshow(cv.cvtColor(test_square3, cv.COLOR_BGR2RGB))
```

```
isPieceWhite(test_square1):  False
isPieceWhite(test_square3):  True
```

Out[13]: <matplotlib.image.AxesImage at 0x2770016bf40>



¡Toca juntarlo todo!

Ya tenemos los métodos esenciales para detectar las piezas del tablero solo falta juntarlo todo y ver los resultados.

Por medio añadiremos un par de funciones, una que nos convertirá el resultado del tablero virtualizado en la notación **FEN** ampliamente utilizada en el ajedrez para describir configuraciones de tableros, y otra para imprimir por pantalla un array bidimensional.

In [14]:
```
def array2fen(chess_array):
    # StringIO es mas eficiente para concatenar
    with io.StringIO() as s:
        for row in range(8):
            empty = 0
            for cell in range(8):
                c = chess_array[row][cell]
                if c != 0:
```

```
                    if empty > 0:
                        s.write(str(empty))
                        empty = 0
                    #escribir en notación FEN la pieza
                    s.write(index2piece[abs(c)].upper(
                    ) if c > 0 else index2piece[abs(c)].lower())
                else:
                    empty += 1
            if empty > 0:
                s.write(str(empty))
            s.write('/')
        # Move one position back to overwrite last '/'
        s.seek(s.tell() - 1)
        s.write(' w KQkq - 0 1')
        return s.getvalue()
```

In [15]:
```
def printArrayBoard(chess_array):
    for f in range(8):
        print("")
        for c in range(8):
            if(chess_array[f,c] < 0):
                print(chess_array[f,c], "|",end = '')
            else:
                print(chess_array[f,c], " |",end = '')
    print("")
```

In [16]:
```
#Conversión de nombres de ficheros a códigos de piezas
filename2piece = {
        "images/template_images\\black_pawn.png": -1,
        "images/template_images\\black_knight.png": -2,
        "images/template_images\\black_bishop.png": -3,
        "images/template_images\\black_rook.png": -4,
        "images/template_images\\black_queen.png": -5,
        "images/template_images\\black_king.png": -6,
    }

#Conversión de código de pieza a notación FEN
index2piece = {
    1: "p",
    2: "n",
    3: "b",
    4: "r",
    5: "q",
    6: "k",
}
```

In [17]:
```
def classifyPieces( squares_array, square_size, cropped_chessboard):

        #Iteramos por todos los cuadrados y vamos añadiendo piezas
        directory = 'images/template_images'
        #Valores
        precision_values = []
        board = np.zeros((8,8), dtype=np.int64)

        for (x, y) in squares_array:
            #Cropeamos las imagenes
            chess_square = cropped_chessboard[x:x + square_size,y:y + square_siz

            #Filtramos primero si es un cuadrado vacío
```

```
            if isBlankSquare(chess_square):
                #Si no tiene nada continuamos a la siguiente iteración
                continue

            for filename in os.listdir(directory):
                f = os.path.join(directory, filename)
                # Si es un archivo
                if os.path.isfile(f):
                    current_img = cv.imread(f, 0)
                    best_match = getBestScaleMatch(chess_square,current_img)
                    precision_values.append((f, best_match))

            #print(precision_values)

            top_piece = "",
            top_value = -math.inf

            for (name, precision) in precision_values:
                if precision > top_value:
                    top_value = precision
                    top_piece = name

            #posicionamiento en tablero virtual
            x_piece = int(x / int(square_size))
            y_piece = int(y / int(square_size))


            piece_value = filename2piece[top_piece]

            #Evaluamos si la pieza es blanca para cambiarle el valor
            if(isPieceWhite(chess_square)):
                piece_value = -piece_value

            #Metemos la pieza en la matriz
            board[x_piece][y_piece] =  piece_value

            #Vaciamos array
            precision_values.clear()

        # Creación de string para notación FEN
        fen = array2fen(board)

        return board, fen
```

Recordemos una vez más el tablero inicial del que partiamos:

In [18]:
```
imagen_inicial = cv.imread('images/CapturaRaw.png',cv.IMREAD_UNCHANGED)

plt.subplot(1,1,1)
plt.title("Imagen original")
plt.imshow(cv.cvtColor(imagen_inicial, cv.COLOR_BGR2RGB))
```

Out[18]:
```
<matplotlib.image.AxesImage at 0x2770018aa40>
```

Imagen original

Y lo que conseguimos obtener a partir de ella

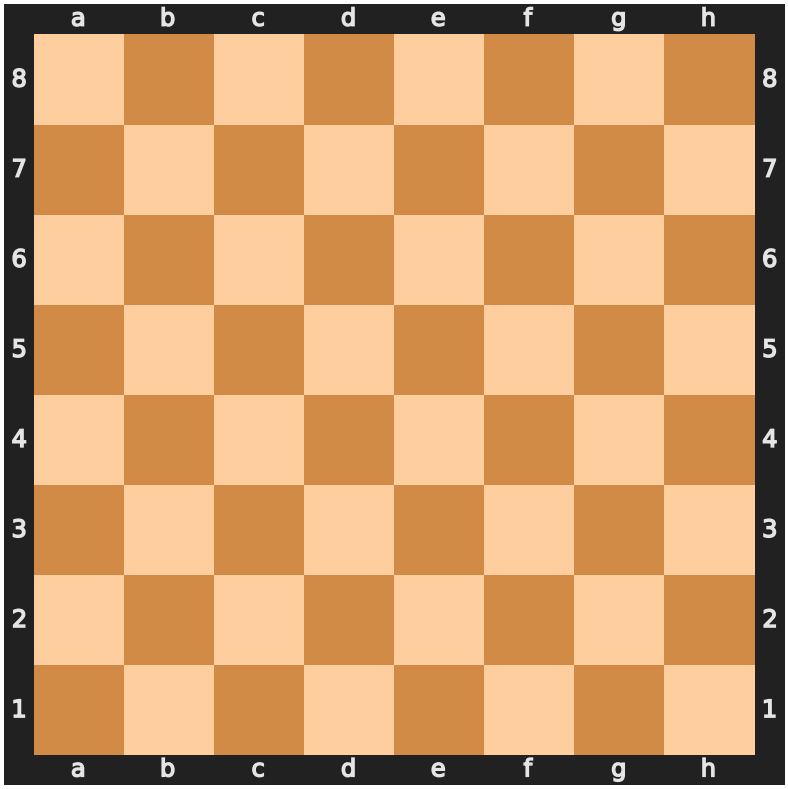
In [19]:
```
board, fen = classifyPieces(squares_arr, square_size,cropped_board)
printArrayBoard(board)
print("")
print("Tablero en notación FEN: ", fen)
```

```
-4 |0  |0  |-6 |0  |0  |0  |-4 |
-1 |-1 |-1 |0  |0  |-1 |-1 |0  |
0  |0  |0  |0  |-1 |-2 |0  |-1 |
0  |0  |-3 |0  |0  |-3 |0  |0  |
3  |0  |0  |0  |0  |1  |0  |0  |
0  |0  |0  |-1 |0  |0  |0  |0  |
1  |1  |1  |1  |0  |0  |1  |1  |
4  |2  |3  |0  |6  |0  |0  |4  |

Tablero en notación FEN:  r2k3r/ppp2pp1/4pn1p/2b2b2/B4P2/3p4/PPPP2PP/RNB1K2R w
KQkq - 0 1
```

Incluso teniendo la notación FEN y una librería de utilidades de ajedrez para Python podemos renderizar la siguiente imagen:

In [20]:
```
board = chess.Board(fen)
board
```

Out[20]:



¡Genial! En principio hemos sido capaces de obtener todas las posiciones y piezas correctamente a través de únicamente una captura de nuestra pantalla completa. Ahora damos paso a la siguiente parte de esta libreta, el motor de ajedrez que nos permitirá decidir los mejores movimientos.

3. Análisis del mejor movimiento

En este apartado, vamos a hacer uso de la librería Chess, puesto que muchas de las funcionalidades triviales de un tablero de ajedrez serían complicadas de implementar y al importarlas simplemente haremos uso de ellas para crear nuestro propio analizador. Además vamos a importar otra librería para poder ir mostrando el tablero en esta libreta.

In [21]:
```
import chess
import chess.svg
from IPython.display import SVG
```

Valor de las casillas

La principal funcionalidad a conseguir en este apartado es, dada una notación **FEN** de un tablero, encontrar el mejor movimiento posible según a qué piezas les toque hacer el siguiente movimiento. Para ello, lo primero que haremos será definir unas tablas de evaluación que permitirá a nuestro analizador saber cuánto valor numérico tiene cada pieza según la casilla del tablero que esté ocupando en el turno a analizar:

In [22]:
```
pawntable = [
 0,  0,  0,  0,  0,  0,  0,  0,
 5, 10, 10,-20,-20, 10, 10,  5,
 5, -5,-10,  0,  0,-10, -5,  5,
 0,  0,  0, 20, 20,  0,  0,  0,
```

```
 5,  5, 10, 25, 25, 10,  5,  5,
10, 10, 20, 30, 30, 20, 10, 10,
50, 50, 50, 50, 50, 50, 50, 50,
 0,  0,  0,  0,  0,  0,  0,  0]

knightstable = [
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  5,  5,  0,-20,-40,
-30,  5, 10, 15, 15, 10,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 10, 15, 15, 10,  0,-30,
-40,-20,  0,  0,  0,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50]

bishopstable = [
-20,-10,-10,-10,-10,-10,-10,-20,
-10,  5,  0,  0,  0,  0,  5,-10,
-10, 10, 10, 10, 10, 10, 10,-10,
-10,  0, 10, 10, 10, 10,  0,-10,
-10,  5,  5, 10, 10,  5,  5,-10,
-10,  0,  5, 10, 10,  5,  0,-10,
-10,  0,  0,  0,  0,  0,  0,-10,
-20,-10,-10,-10,-10,-10,-10,-20]

rookstable = [
  0,  0,  0,  5,  5,  0,  0,  0,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
  5, 10, 10, 10, 10, 10, 10,  5,
 0,  0,  0,  0,  0,  0,  0,  0]

queenstable = [
-20,-10,-10, -5, -5,-10,-10,-20,
-10,  0,  0,  0,  0,  0,  0,-10,
-10,  5,  5,  5,  5,  5,  0,-10,
  0,  0,  5,  5,  5,  5,  0, -5,
 -5,  0,  5,  5,  5,  5,  0, -5,
-10,  0,  5,  5,  5,  5,  0,-10,
-10,  0,  0,  0,  0,  0,  0,-10,
-20,-10,-10, -5, -5,-10,-10,-20]

kingstable = [
 20, 30, 10,  0,  0, 10, 30, 20,
 20, 20,  0,  0,  0,  0, 20, 20,
-10,-20,-20,-20,-20,-20,-20,-10,
-20,-30,-30,-40,-40,-30,-30,-20,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30]
```

Hay una tabla para cada tipo de pieza del tablero, y cada nos servirá tanto para las piezas blancas como para las negras pues, al estar ambos bandos en igualdad de condiciones, tan solo tenemos que invertir verticalmente el tablero para obtener la evaluación del bando contrario. Las tablas están diseñadas de la siguiente manera:

- **Peón:** el programa se ve animado a avanzar los peones y no dejarse los peones del centro inmóviles.
- **Caballo:** la tabla está diseñada para que los caballos vayan al centro y eviten los bordes del tablero.
- **Alfil:** los alfiles simplemente evitarán los bordes y las esquinas para aprovechar al máximo su rango de movimiento.
- **Torre:** las torres tratarán de ocupar la fila 7 (fila 2 en caso de las piezas negras) y evitar las columnas 'a' y 'h' (bordes del tablero)
- **Dama:** la dama combinará las reglas del alfil y la torre, evitar las esquinas y bordes del tablero y mantenerse en el centro.
- **Rey:** el rey se mantendrá tras los peones y se verá animado a enrocarse.

Las tablas no han sido desarrolladas por nosotros, las hemos obtenido del siguiente enlace.

Por comodidad y claridad de código, juntaremos estas tablas en un array. Además también crearemos arrays para los tipos de pieza y los valores de cada una de estas.

In [23]:
```
pieces = [chess.PAWN, chess.KNIGHT, chess.BISHOP, chess.ROOK, chess.QUEEN, chess
tables = [pawntable, knightstable, bishopstable, rookstable, queenstable, kingst
values = [100, 320, 330, 500, 900]
```

Analizador

Para el analizador, implementaremos una clase denominada **ChessEngine**, que será la encargada de hacer todo el trabajo de este apartado.

Es importante matizar que, en el ajedrez, la evaluación de un tablero es un número real (generalmente con dos decimales), que será positivo si van ganando las piezas blancas y negativo en caso de que vayan ganando las negras. Cada pieza tiene un valor numérico, siendo el peón la de menor valor con 1 punto, seguido del caballo y el alfil con 3 puntos, la torre con valor de 5 puntos y la dama con valor de 9 ó 10 puntos (esto depende de la persona a la que le preguntes, nosotros utilizaremos el valor de 9). Al rey no se le asigna un valor, pues no puede ser capturado y por tanto se considera que su valor es "infinito".

Aclarar que como hemos programado utilizando clases, primero meteremos toda la clase en la misma celda para que pueda compilar y luego iremos definiendo y explicando las funciones una a una para mayor claridad de la libreta

El primer paso de este analizador (función **__init__**) será evaluar la posición que nos han dado, previamente calculada mediante visión por computador, y para ello diseñaremos un algoritmo que procederá de la siguiente forma:

1. Contará cuántas piezas hay de cada tipo por cada color.
2. Calculará la diferencia de material entre ambos bandos, usando las 'values' dadas en la celda de arriba
3. Calculará el valor de cada pieza según la casilla en la que se encuentra, e irá calculando la diferencia entre las piezas blancas y las negras.

4. La evaluación final será la suma de la diferencia de material más la evaluación total de cada tipo de pieza.

In [24]:
```
class ChessEngine:


    def __init__(self, custom_board):

        global pawntable, knightstable, bishopstable, rookstable, queenstable, k

        self.board_value = -9999
        self.board = custom_board

        # Obtenemos el numero de piezas que tienen ambos jugadores
        white_pawns = len(self.board.pieces(chess.PAWN, chess.WHITE))
        white_knights = len(self.board.pieces(chess.KNIGHT, chess.WHITE))
        white_bishops = len(self.board.pieces(chess.BISHOP, chess.WHITE))
        white_rooks = len(self.board.pieces(chess.ROOK, chess.WHITE))
        white_queen = len(self.board.pieces(chess.QUEEN, chess.WHITE))
        black_pawns = len(self.board.pieces(chess.PAWN, chess.BLACK))
        black_knights = len(self.board.pieces(chess.KNIGHT, chess.BLACK))
        black_bishops = len(self.board.pieces(chess.BISHOP, chess.BLACK))
        black_rooks = len(self.board.pieces(chess.ROOK, chess.BLACK))
        black_queen = len(self.board.pieces(chess.QUEEN, chess.BLACK))

        # Calculamos la diferencia de material
        materialDiff = 100 * (white_pawns - black_pawns) + 320 * (white_knights
                white_bishops - black_bishops) \
                    + 500 * (white_rooks - black_rooks) + 900 * (white_queen - b

        # Calculamos la evaluacion segun las matrices definidas arriba
        # La primera linea ira sumando las piezas que tenemos en el tablero segu
        # La segunda linea flipea el tablero y evalua
        eval_pawns = sum(
            [pawntable[i] for i in self.board.pieces(chess.PAWN, chess.WHITE)])
        eval_pawns = eval_pawns + sum([
            -pawntable[chess.square_mirror(i)]
            for i in self.board.pieces(chess.PAWN, chess.BLACK)
        ])

        eval_knights = sum([
            knightstable[i]
            for i in self.board.pieces(chess.KNIGHT, chess.WHITE)
        ])
        eval_knights = eval_knights + sum([
            -knightstable[chess.square_mirror(i)]
            for i in self.board.pieces(chess.KNIGHT, chess.BLACK)
        ])

        eval_bishops = sum([
            bishopstable[i]
            for i in self.board.pieces(chess.BISHOP, chess.WHITE)
        ])
        eval_bishops = eval_bishops + sum([
            -bishopstable[chess.square_mirror(i)]
            for i in self.board.pieces(chess.BISHOP, chess.BLACK)
        ])

        eval_rooks = sum([
```

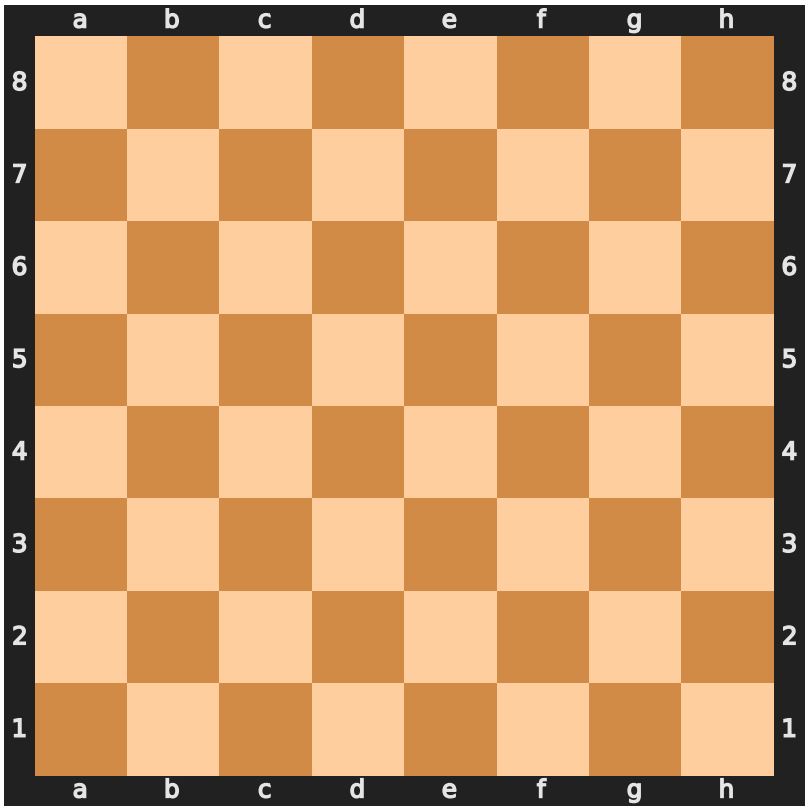
```
            rookstable[i] for i in self.board.pieces(chess.ROOK, chess.WHITE)
        ])
        eval_rooks = eval_rooks + sum([
            -rookstable[chess.square_mirror(i)]
            for i in self.board.pieces(chess.ROOK, chess.BLACK)
        ])

        eval_queen = sum([
            queenstable[i] for i in self.board.pieces(chess.QUEEN, chess.WHITE)
        ])
        eval_queen = eval_queen + sum([
            -queenstable[chess.square_mirror(i)]
            for i in self.board.pieces(chess.QUEEN, chess.BLACK)
        ])

        eval_king = sum([
            kingstable[i] for i in self.board.pieces(chess.KING, chess.WHITE)
        ])
        eval_king = eval_king + sum([
            -kingstable[chess.square_mirror(i)]
            for i in self.board.pieces(chess.KING, chess.BLACK)
        ])

        self.board_value = materialDiff + eval_pawns + eval_knights + eval_bishc

        print("Board value is: ", self.board_value)
        #return board_value

    def evaluate_board(self):


        if self.board.is_checkmate():
            if self.board.turn:
                return -9999
            else:
                return 9999
        if self.board.is_stalemate() or self.board.is_insufficient_material():
            return 0
        evaluation = self.board_value
        if self.board.turn:
            return evaluation
        else:
            return -evaluation

    def update_evaluation(self, move, side):
        global tables, values

        moving_piece = self.board.piece_type_at(move.from_square)

        if side:  #blancas juegan
            self.board_value = self.board_value - tables[moving_piece -
                                                         1][move.from_square]
            #Actualizamos el enroque
            if (move.from_square == chess.E1) and (move.to_square == chess.G1):
                self.board_value = self.board_value - rookstable[chess.H1]
                self.board_value = self.board_value + rookstable[chess.F1]
            elif (move.from_square == chess.E1) and (move.to_square
                                                     == chess.C1):
                self.board_value = self.board_value - rookstable[chess.A1]
                self.board_value = self.board_value + rookstable[chess.D1]
```

```
        else:
            self.board_value = self.board_value + tables[moving_piece -
                                                         1][move.from_square]
            # Actualizamos el enroque
            if (move.from_square == chess.E8) and (move.to_square == chess.G8):
                self.board_value = self.board_value - rookstable[chess.H8]
                self.board_value = self.board_value + rookstable[chess.F8]
            elif (move.from_square == chess.E8) and (move.to_square
                                                     == chess.C8):
                self.board_value = self.board_value - rookstable[chess.A8]
                self.board_value = self.board_value + rookstable[chess.D8]

        if side:
            self.board_value = self.board_value + tables[moving_piece -
                                                         1][move.to_square]
        else:
            self.board_value = self.board_value - tables[moving_piece -
                                                         1][move.to_square]

        #Actualizamos diferencia de material
        if move.drop is not None:
            if side:
                self.board_value = self.board_value + values[move.drop - 1]
            else:
                self.board_value = self.board_value - values[move.drop - 1]

        #Actualizamos posibles promociones de piezas
        if move.promotion is not None:
            if side:
                self.board_value = self.board_value + values[
                    move.promotion - 1] - values[moving_piece - 1]
                self.board_value = self.board_value - tables[moving_piece - 1][m
                            + tables[move.promotion-1][move.to_square]
            else:
                self.board_value = self.board_value - values[
                    move.promotion - 1] + values[moving_piece - 1]
                self.board_value = self.board_value + tables[moving_piece - 1][m
                            - tables[move.promotion - 1][move.to_square]
            return move

    def make_move(self, move):

        self.update_evaluation(move, self.board.turn)
        self.board.push(move)
        return move

    def unmake_move(self):

        movement = self.board.pop()
        self.update_evaluation(movement, not self.board.turn)
        return movement

    def alphabeta(self, alpha, beta, depth):

        best_score = -9999
        if depth == 0:
            return self.quiesce(alpha, beta)

        for move in self.board.legal_moves:
            self.make_move(move)
```

```
            score = -self.alphabeta(-beta, -alpha, depth - 1)
            self.unmake_move()
            if score >= beta:
                return score
            if score > best_score:
                best_score = score
            if score > alpha:
                alpha = score

        return best_score

    def quiesce(self, alpha, beta):

        stand_pat = self.evaluate_board()
        if stand_pat >= beta:
            return beta
        if alpha < stand_pat:
            alpha = stand_pat

        for move in self.board.legal_moves:
            if self.board.is_capture(move):
                self.make_move(move)
                score = -self.quiesce(-beta, -alpha)
                self.unmake_move()

                if score >= beta:
                    return beta
                if score > alpha:
                    alpha = score

        return alpha

    def selectmove(self, depth):

        try:
            move = chess.polyglot.MemoryMappedReader(
                "Perfect2017.bin").weighted_choice(self.board).move
            print("OPENING ES: ", move)
            # movehistory.append(move)
            return move
        except:
            bestMove = chess.Move.null()
            bestValue = -99999
            alpha = -100000
            beta = 100000
            for move in self.board.legal_moves:
                self.make_move(move)
                self.boardValue = -self.alphabeta(-beta, -alpha, depth - 1)
                if self.boardValue > bestValue:
                    bestValue = self.boardValue
                    bestMove = move
                if self.boardValue > alpha:
                    alpha = self.boardValue
                self.unmake_move()

            # movehistory.append(bestMove)
            return bestMove
```

Por ejemplo, si le damos la posición inicial de una partida a nuestro motor, sacará la siguiente evaluación:

In [25]:
```
fen = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
board = chess.Board(fen)
display(SVG(chess.svg.board(board=board,size=400)))
engine = ChessEngine(board)
```




```
Board value is:  0
```

La siguiente función, **evaluate_board**, es muy simple y únicamente servirá para comprobar si se ha dado jaque mate o no, o si la partida ha terminado en tablas por rey ahogado o material insuficiente:

In [26]:
```
    def evaluate_board(self):


        if self.board.is_checkmate():
            if self.board.turn:
                return -9999
            else:
                return 9999
        if self.board.is_stalemate() or self.board.is_insufficient_material():
            return 0
        evaluation = self.board_value
        if self.board.turn:
            return evaluation
        else:
            return -evaluation
```

Una vez tenemos esto, encontramos una problemática cuando empezamos a desarrollar nuestro motor. Al principio intentamos hacer uso de estas dos funciones para evaluar cada posible movimiento que se puede hacer en una posición, pero iterar por todo el

tablero en cada posible movimiento es una operación muy costosa que ocupaba bastante tiempo (llegando a usar varios minutos para sacar el mejor movimiento en una posición).
Por esta razón, se nos ocurrió mantener esta evaluación inicial que hacemos como el análisis "general" de la posición, y el motor sólo tiene que restar a la evaluación la casilla desde la que se ha movido la pieza (puesto que ya no estará ahí) y sumar el valor que tiene la pieza en la casilla a la que va a moverse. Además otros movimientos especiales como las capturas, el enroque o las promociones de los peones necesitan una consideración única, pues o se modifican más de dos casillas o algunas piezas se convierten en otras.

Con esto en mente, implementamos nuestra siguiente función **update_evaluation**, que se compondrá de los siguientes pasos:

1. Dependiendo del color, resta (juegan blancas) o suma (juegan negras) a la evaluación la casilla desde la que se ha movido la pieza elegida.
2. Si el movimiento elegido es un enroque, tiene que actualizar también la casilla desde y a la que se mueve la torre, pues para hacer el enroque la pieza elegida para realizar el movimiento **siempre** es el rey. En caso de tocar la torre antes del rey, el enroque no será un movimiento posible de acuerdo con las reglas del ajedrez.
3. Suma (o resta según el color que esté jugando) la casilla a la que se ha movido la pieza.
4. Si se ha capturado una pieza (comprobado con **move.drop**), se añade el valor de esa pieza al bando que la haya capturado.
5. Si el movimiento ha sido una promoción de un peón, se eliminan los valores relacionados al movimiento del peón y se actualiza la casilla con la pieza elegida para reemplazar al peón.

Una vez hecho todo esto, tenemos lista la función.

In [27]:
```
    def update_evaluation(self, move, side):
        global tables, values

        moving_piece = self.board.piece_type_at(move.from_square)

        if side:  #blancas juegan
            self.board_value = self.board_value - tables[moving_piece -
                                                         1][move.from_square]
            #Actualizamos el enroque
            if (move.from_square == chess.E1) and (move.to_square == chess.G1):
                self.board_value = self.board_value - rookstable[chess.H1]
                self.board_value = self.board_value + rookstable[chess.F1]
            elif (move.from_square == chess.E1) and (move.to_square
                                                     == chess.C1):
                self.board_value = self.board_value - rookstable[chess.A1]
                self.board_value = self.board_value + rookstable[chess.D1]
        else:
            self.board_value = self.board_value + tables[moving_piece -
                                                         1][move.from_square]
            # Actualizamos el enroque
            if (move.from_square == chess.E8) and (move.to_square == chess.G8):
                self.board_value = self.board_value - rookstable[chess.H8]
```

```
                self.board_value = self.board_value + rookstable[chess.F8]
            elif (move.from_square == chess.E8) and (move.to_square
                                                     == chess.C8):
                self.board_value = self.board_value - rookstable[chess.A8]
                self.board_value = self.board_value + rookstable[chess.D8]

        if side:
            self.board_value = self.board_value + tables[moving_piece -
                                                         1][move.to_square]
        else:
            self.board_value = self.board_value - tables[moving_piece -
                                                         1][move.to_square]

        #Actualizamos diferencia de material
        if move.drop is not None:
            if side:
                self.board_value = self.board_value + values[move.drop - 1]
            else:
                self.board_value = self.board_value - values[move.drop - 1]

        #Actualizamos posibles promociones de piezas
        if move.promotion is not None:
            if side:
                self.board_value = self.board_value + values[
                    move.promotion - 1] - values[moving_piece - 1]
                self.board_value = self.board_value - tables[moving_piece - 1][m
                            + tables[move.promotion-1][move.to_square]
            else:
                self.board_value = self.board_value - values[
                    move.promotion - 1] + values[moving_piece - 1]
                self.board_value = self.board_value + tables[moving_piece - 1][m
                            - tables[move.promotion - 1][move.to_square]
            return move
```

Búsqueda de movimientos

Ya tenemos implementado todo lo relacionado al análisis de un tablero, pero todavía tenemos que desarrollar la búsqueda para que nuestro motor encuentre qué movimientos posibles hay en la posición y cuáles son los mejores.
Para esto primero vamos a definir dos funciones simples que nos sirvan para hacer y deshacer movimientos y poder mandar posiciones nuevas al analizador, llamadas **make_move** y **unmake_move**:

In [28]:
```
    def make_move(self, move):

        self.update_evaluation(move, self.board.turn)
        self.board.push(move)
        return move

    def unmake_move(self):

        movement = self.board.pop()
        self.update_evaluation(movement, not self.board.turn)
        return movement
```

Para implementar la búsqueda estuvimos investigando cuáles eran las mejores maneras de buscar movimientos posibles en un tablero de ajedrez sin ocupar mucho tiempo, y terminamos encontrando técnicas como el Minimax, Alpha-Beta pruning o Quiescence search, las cuales implementaremos en nuestro motor pues son fáciles de entender y suficientemente eficientes para el proyecto que queremos llevar a cabo.

Cabe destacar antes de exponer estos algoritmos que los motores de ajedrez funcionan con un término denominado *profundidad*, que representa cuántos movimientos en el futuro es capaz de ver y analizar correctamente el motor.

El primer algoritmo que enseñamos es la **Quiescence search**. Es una búsqueda realizada cuando hemos alcanzado la profundidad deseada. La lógica detrás de esta búsqueda es que si, por ejemplo, al terminar un análisis concluimos con un movimiento en el que la dama toma un peón, el analizador puede pensar que ha conseguido un punto de ventaja. Pero si en el siguiente movimiento perdemos la dama por culpa de otra pieza, habremos fallado estrepitosamente en nuestra búsqueda por un movimiento adecuado. Para la implementación se define una evaluación estática **stand_pat** para establecer un límite inferior a la puntuación. Si este límite ya es igual o mayor que **beta** (la respuesta de nuestro oponente) podemos devolver este valor, pues nuestro movimiento no supone un peligro. En caso contrario, la búsqueda continúa manteniendo el valor **stand_pat** como límite inferior si supera a **alpha** para ver qué movimientos pueden aumentar éste **alpha**. El objetivo de este algoritmo es evitar el Efecto Horizonte, que ocurre cuando un motor de ajedrez trata de pospone una pérdida inevitable de material perdiendo más material en el proceso.

La implementación queda de la siguiente forma, función **quiesce**:

In [29]:
```
    def quiesce(self, alpha, beta):

        stand_pat = self.evaluate_board()
        if stand_pat >= beta:
            return beta
        if alpha < stand_pat:
            alpha = stand_pat

        for move in self.board.legal_moves:
            if self.board.is_capture(move):
                self.make_move(move)
                score = -self.quiesce(-beta, -alpha)
                self.unmake_move()

                if score >= beta:
                    return beta
                if score > alpha:
                    alpha = score

        return alpha
```

El segundo algoritmo que vamos a explicar es el **Minimax** mediante **Alpha-Beta pruning**. El **Minimax** es un algoritmo que se basa en la siguiente consideración: la evaluación del mejor movimiento del bando que mueve (jugador *max*) se corresponde

con la evaluación del peor movimiento que el oponente (jugador *min*) puede realizar. El problema de este algoritmo es que iterará por todos los movimientos posibles y todas las contestaciones posibles a este movimiento, por tanto decidimos implementar el **Alpha-Beta pruning** que es una mejora de este algoritmo **Minimax** pero utilizando la técnica de ramificación y poda.

En esta mejora del algoritmo, si se encuentra un buen movimiento y estamos tratando de encontrar alternativas a ese movimiento, si encontramos **una sola refutación** para el movimiento alternativo el algoritmo deja de buscar por esa rama y tratará de seguir mirando alternativas en lugar de iterar por todas las contestaciones posibles al movimiento alternativo. Por ejemplo, si encuentra un movimiento que gana un caballo, y luego encuentra un movimiento alternativo que gana una torre pero **nos hace perder la dama**, esa alternativa quedará desechada y se buscará otra en lugar de iterar por todas las demás posibles contestaciones, porque sabemos que el primer movimiento es mejor que el segundo.
Para profundidades mayores a dos, es decir, mirar más de dos movimientos en el futuro, se complica un poco más pues a la respuesta de un jugador puede haber otra respuesta más. En estos casos se mantienen un límite inferior *Alpha* y uno superior *Beta*, ya que si un movimiento es muy malo no lo consideraremos pero si un movimiento es demasiado bueno tampoco lo consideraremos, pues se supone que el oponente va a ser capaz de verlo y no permitir que ocurra.

La implementación quedaría de la siguiente forma, en la función **alphabeta**:

In [30]:
```
    def alphabeta(self, alpha, beta, depth):

        best_score = -9999
        if depth == 0:
            return self.quiesce(alpha, beta)

        for move in self.board.legal_moves:
            self.make_move(move)
            score = -self.alphabeta(-beta, -alpha, depth - 1)
            self.unmake_move()
            if score >= beta:
                return score
            if score > best_score:
                best_score = score
            if score > alpha:
                alpha = score

        return best_score
```

Selección de movimiento

Ya definidas las diferentes formas que tenemos de analizar tableros y buscar los posibles movimientos, tan sólo nos queda definir un algoritmo que junte todas estas funciones para seleccionar el movimiento más adecuado dada una posición, nuestra función **selectmove**.

Esta función primero hará uso de la librería **chess.polyglot** (que importaremos a continuación), la cual nos permite utilizar libros de aperturas para nuestro motor. La pregunta es, **¿por qué utilizar un libro de aperturas, si se supone que nuestro motor ya analiza los mejores movimientos?**. La realidad es que las aperturas son partes *ya estudiadas* en el mundo del ajedrez, comienzos de partidas que **ya han sido jugadas**, y como tal los movimientos más óptimos ya han sido jugados miles de veces, aunque estos movimientos en un principio no se correspondan con la evaluación que pueda hacer un motor de ajedrez. Por ello, el motor intentará buscar un movimiento en el libro de aperturas **Perfect2017.bin** y, en caso de que no encuentre ninguno, hará toda la evaluación por sí mismo, primero inicializando las variables necesarias y luego procediendo con la búsqueda.

Nosotros lo hemos implementado de la siguiente forma:

In [31]:
```
import chess.polyglot
```

In [32]:
```
    def selectmove(self, depth):

        try:
            move = chess.polyglot.MemoryMappedReader(
                "Perfect2017.bin").weighted_choice(self.board).move
            print("OPENING ES: ", move)
            return move
        except:
            bestMove = chess.Move.null()
            bestValue = -99999
            alpha = -100000
            beta = 100000
            for move in self.board.legal_moves:
                self.make_move(move)
                self.boardValue = -self.alphabeta(-beta, -alpha, depth - 1)
                if self.boardValue > bestValue:
                    bestValue = self.boardValue
                    bestMove = move
                if self.boardValue > alpha:
                    alpha = self.boardValue
                self.unmake_move()

            return bestMove
```

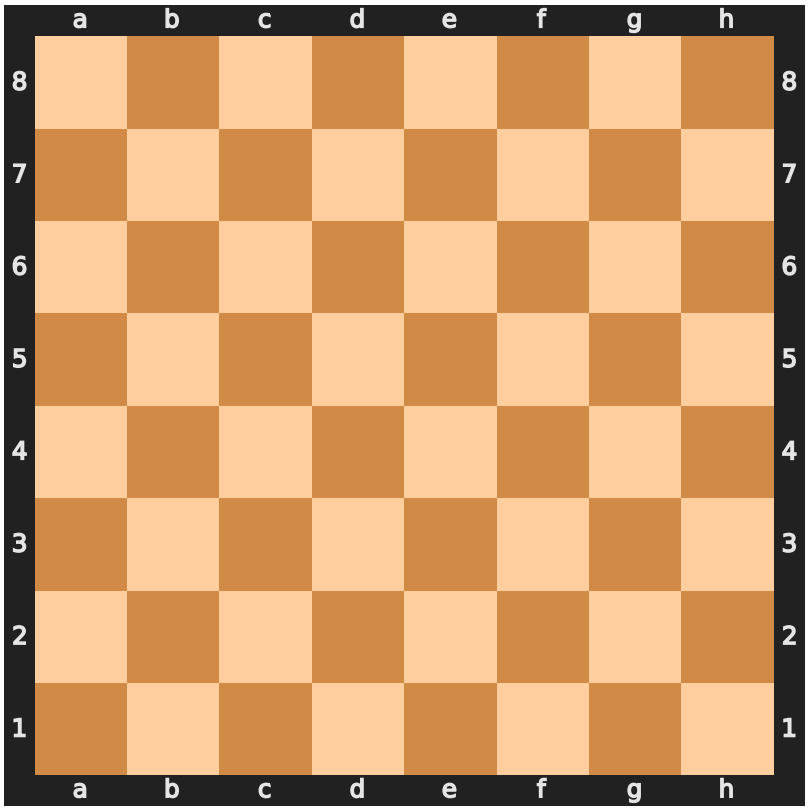
Demostración

Con todo esto, nuestro motor ya está listo para evaluar posiciones y obtener los mejores movimientos, y a continuación pondremos dos ejemplos de partida.

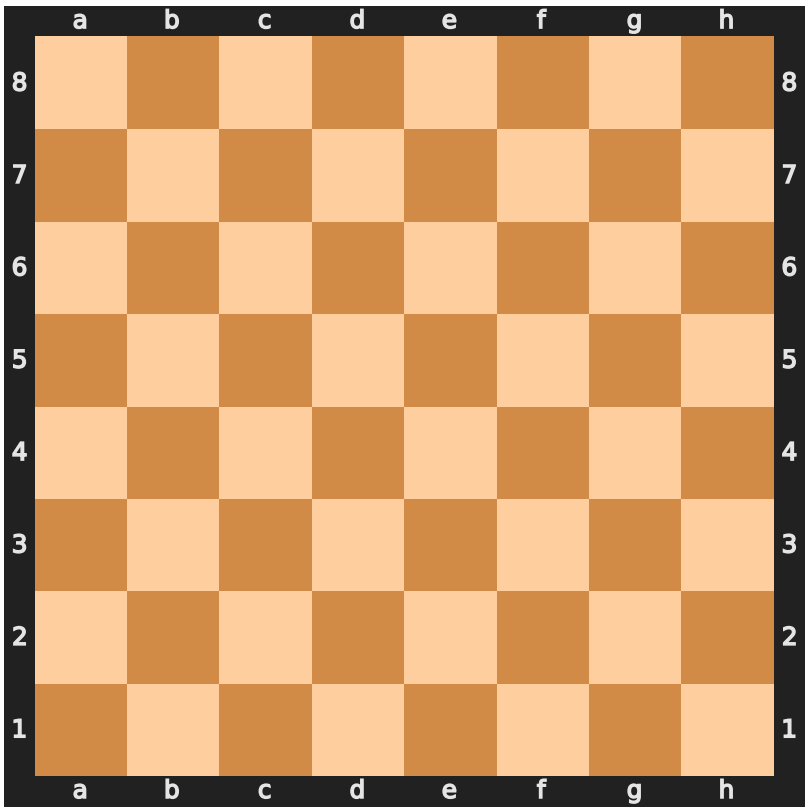
En el primer ejemplo, usaremos una posición conocida por ser una de las posibles posiciones de la **Apertura Italiana:**:

In [33]:
```
fen = "r1bqkb1r/pppp1ppp/2n2n2/4p3/2B1P3/5N2/PPPP1PPP/RNBQK2R w KQkq - 4 4"
board = chess.Board(fen)
display(SVG(chess.svg.board(board=board,size=400)))
engine = ChessEngine(board)
```

```
board.push(engine.selectmove(3))
display(SVG(chess.svg.board(board=board,size=400)))
```



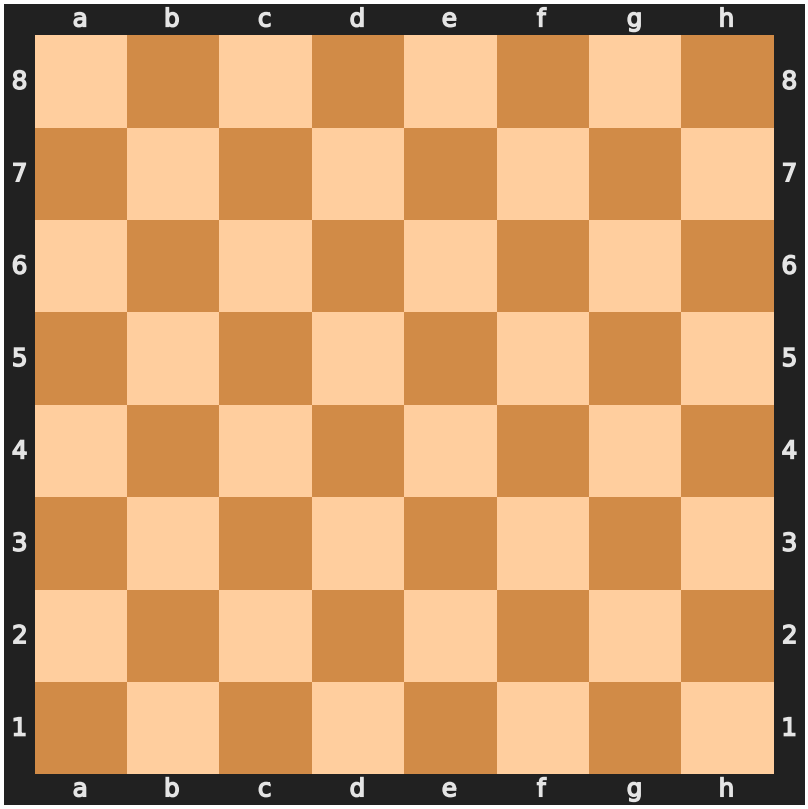

```
Board value is:  -30
OPENING ES:  d2d3
```



En este caso vemos que ha seguido el libro de aperturas, pues el movimiento da pie a una de las variantes modernas de la **Defensa de dos caballos** que han jugado las piezas negras en este caso.

Ahora, vamos a plantearle una posición de una partida que jugamos Pablo y yo recientemente. La posición es la siguiente:

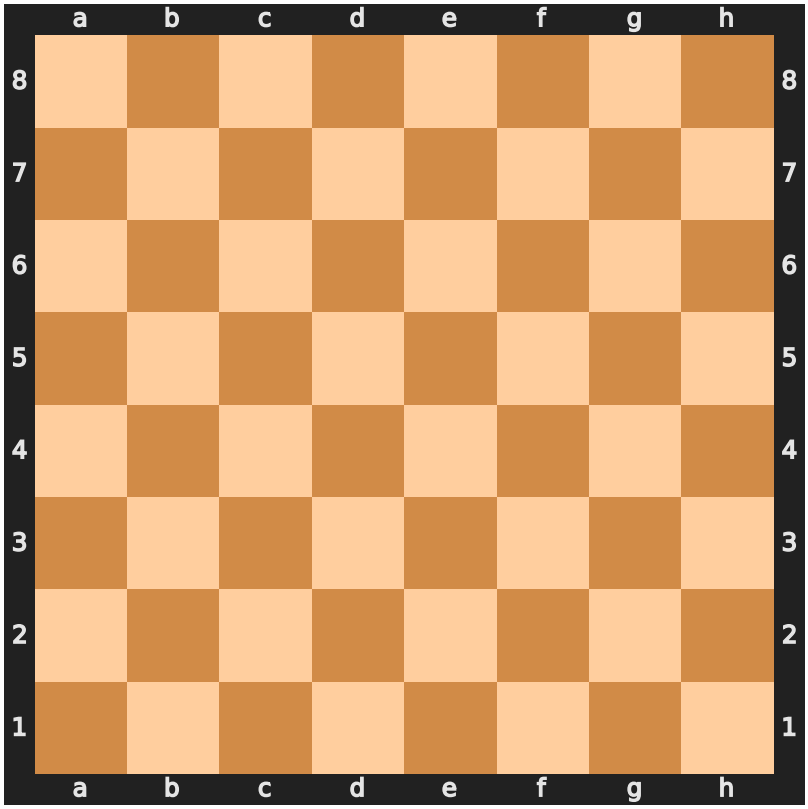
In [34]:
```
fen = "2kr3r/ppp3p1/1b1qpp2/nP1p1b1p/P2P2n1/2N2NPB/2P1PP1P/1RBQK1R1 w - - 0 1"
board = chess.Board(fen)
display(SVG(chess.svg.board(board=board,size=400)))
```



Como podemos ver, muchas piezas han sido movidas en esta partida y no estamos ante ninguna posición típica de aperturas. Las piezas negras disponen de ventaja posicional, y sabemos que el último movimiento de negras ha sido **Dd6**, es decir, ha movido la dama a la casilla d6 del tablero. Ante esta posición las blancas tienen muchas opciones distintas, veamos qué hace nuestro motor:

In [35]:
```
engine = ChessEngine(board)
#engine.selectmove(3)
board.push(engine.selectmove(3))
display(SVG(chess.svg.board(board=board,size=400)))
```

```
Board value is:  -40
```



El movimiento elegido por el motor ha sido **Af4**, movimiento que desarrolla el alfil y ataca a la dama. No es quizás el mejor movimiento, pero desarrolla una pieza y ataca a otra pieza importante del contrario, además de poner un alfil apuntando hacia el medio, en concordancia a lo que programamos cuando implementamos las tablas.

4. Movimiento automático por ordenador

Para este apartado volvemos a hacer uso de la librería PyAutoGUI, con la que podremos especificar acciones específicas para realizar con el ratón en la pantalla. La idea es que una vez obtuviesemos el movimiento deseado para el tablero de ajedrez, el cual podría ser por ejemplo "a2a3", convirtiésemos esa cadena en una serie de instrucción que representase arrastrar un peon hacia la casilla de alante. Con esto en mente diseñamos la siguiente clase:

In [36]:
```
class AutoMover:

    def __init__(self, x_top_left,y_top_left,square_size,color):
        self.x_top_left = x_top_left
        self.y_top_left = y_top_left
        self.square_size = square_size
        self.color = color

        if color == 'w':
            self.letter2number = {"a":0,"b":1,"c":2,"d":3,"e":4,"f":5,"g":6,"h":
        else:
            self.letter2number = {"h":0,"g":1,"f":2,"e":3,"d":4,"c":5,"b":6,"a":



    def moveToSquare(self,square_notation):
        x = self.letter2number[square_notation[0:1]]
```

```
        if self.color == 'w':
            y = 8 - int(square_notation[1:2])
            offsetx = self.square_size/2
            offsety = offsetx
        else:
            y = int(square_notation[1:2])
            offsetx = self.square_size/2
            offsety = -self.square_size/2


        pyautogui.moveTo(self.x_top_left + x*self.square_size + offsetx,self.y_t


    def clickAndMoveTo(self,square_notation):
        x = self.letter2number[square_notation[0:1]]
        if self.color == 'w':
            y = 8 - int(square_notation[1:2])
            offsetx = self.square_size/2
            offsety = offsetx
        else:
            y = int(square_notation[1:2])
            offsetx = self.square_size/2
            offsety = -self.square_size/2
        offset = self.square_size/2
        pyautogui.dragTo(self.x_top_left + x*self.square_size + offset,self.y_tc

    def movePiece(self,starting_pos, ending_pos):
        self.moveToSquare(starting_pos)
        self.clickAndMoveTo(ending_pos)
```

In [37]:
```
#Un ejemplo de movimiento automático del ratón
mover = AutoMover(500,200,50,'w')
mover.movePiece("a1","b3")
```

El código en sí no tiene mucho misterio, solo había que calibrar un poco las posiciones para que estuviesen centradas en los cuadrados del tablero y hacer correctamente la traducción de coordenadas de ajedrez a coordenadas en los pixeles de la pantalla. Por supuesto, para poder calcular todo esto a esta clase la tenemos que inicializar con parámetros que hemos calculado previamente como la posición del tablero en la pantalla y el tamaño de celda del tablero.

5. Conclusiones

Limitaciones

- No se puede usar con los estilos de piezas y tableros personalizados que te permite chess.com, puesto que utilizamos *template matching* para las piezas.
- Los tiempos de reconocimiento del tablero pueden ser elevados en máquinas de bajo rendimiento, y a su vez su complejidad aumenta a mayor resolución de pantalla (puesto que se procesan más píxeles).
- El programa usa la API de Windows, por tanto no funciona para otros sistemas operativos.

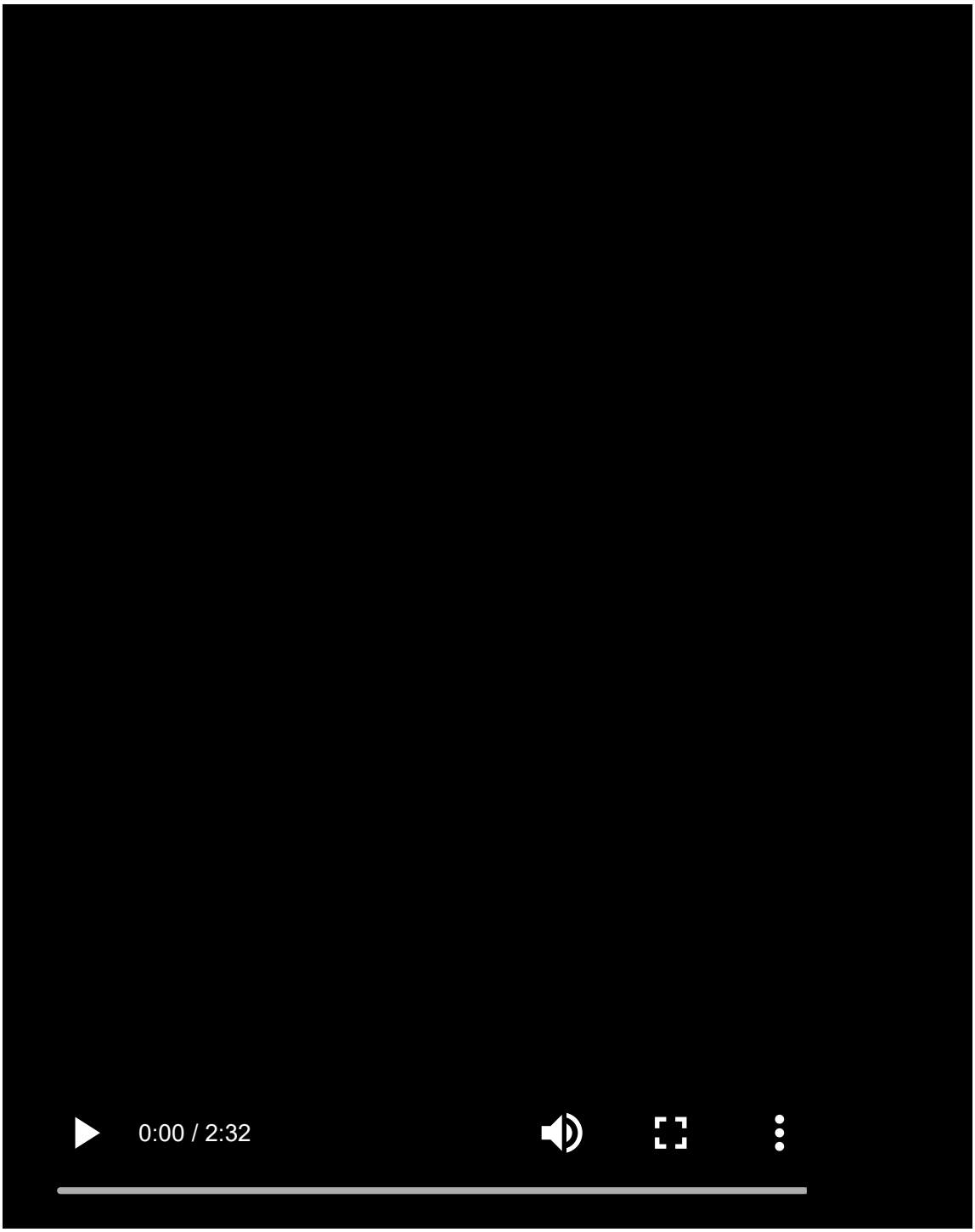
- La complejidad de los algoritmos de búsqueda de movimiento aumenta muchísimo con cada aumento de la profundidad de búsqueda, pues hay que iterar por muchos movimientos más.
- Una partida de ajedrez tiene distintas fases, y las tablas que tenemos para la evaluación de casillas no son las más adecuadas para la fase final de una partida. Con algo más de tiempo y disponibilidad se podría adaptar para todas las fases del juego pero es más complicado.
- Obviamente, esta forma de jugar se considera hacer trampas en todas las webs de ajedrez y, por tanto, tu cuenta será baneada en caso de usar esta herramienta contra jugadores reales.

Posibles optimizaciones/características

- Cambiar el algoritmo de detección de piezas, para mejorar el tiempo de ejecución y poder aplicar el programa a piezas y tableros personalizados.
  - Se nos ocurre solucionar el problema mediante distintas técnicas como el uso de inteligencia artificial o reconociemiento de piezas con una red neuronal. También podríamos usar técnicas distintas de extracción de características de una imagen como los momentos de Hu.
- Adaptación a otros sistemas operativos.
  - Solucionar esto es simple, pero para nuestra demostración preferimos ajustarnos sólo a Windows, pues es el sistema operativo que utilizamos.
- Optimizar el motor de juego para mayores profundidades.
  - Podríamos hacerlo programando en otro lenguaje más eficiente, como C, o entrenando una red neuronal para que aprenda de partidas ya jugadas.
- Adaptar el motor de juego a fases avanzadas del juego.
  - Cambiando las tablas de evaluación dependiendo de lo que ocurra en el tablero podríamos conseguir esto.

Finalmente, creemos que hemos conseguido una implementación relativamente eficiente, capaz de adaptarse a prácticamente cualquier tipo de ordenador y de jugar sin ningún problema una partida completa con buenos resultados.

6. Demo



>

In [ ]: