

Project 3 Writeup

In the beginning...

Point features podem ser usados para localizar um conjunto esparso de localizações correspondentes em diferentes imagens. Existem duas abordagens principais para encontrar *features points* e suas correspondências. A primeira era encontrar recursos em uma imagem que possam ser rastreados com precisão usando a tecnologia de busca local, já a segunda combina em detectar recursos em todas as imagens em consideração, e em seguida, combinar os recursos com base em suas aparências locais (Szeliski, 2010).

Para conseguir detectar os *features* devemos comparar os dois pontos em uma imagem em que pode ser usado o somatório do quadrado da diferença. Depois de detectar os *features*, devemos combinar determinando quais recursos vêm de locais correspondentes em imagens diferentes. Uma vez extraído as *features* e os descritores de duas ou mais imagens, é preciso estabelecer algumas preliminares para combinar *features* entre as imagens. Para isso é necessário duas etapas, a primeira é selecionar uma estratégia de combinação, na qual determina quais correspondências serão passada para um processo adicional e a segunda etapa é criar estruturas de dados e algoritmos eficientes para realizar essa correspondência o mais rápido possível (Szeliski, 2010).

Para o *local feature matching*, foi utilizado a linguagem Octave, nas quais implementamos as funções *get_features_patch()*, *match_features()*, *get_features_sift()* e *calc_magnitude()*. Parte do código já foi disponibilizado pronto, tal como, a posição de algumas *features* interessantes, que é o retorno da função *cheat_interest_points()*. Após ter implementado o *get_features_patch()* e o *match_features()*, nessa respectiva ordem, aproveitamos parte da primeira função implementada para implementar *get_features_sift()*, já que existe uma certa semelhança e por fim implementamos o *calc_magnitude()*.

Interesting Implementation Detail

A implementação começou na função *get_features_patch()* que recebe como parâmetro a imagem original, os vetores retornados na função *cheat_interest_points* e a largura do descritor da janela da imagem (código 1). O algoritmo começa criando a matriz onde será armazenado os valores, que foram capturados de certos intervalos da imagem original e que foram normalizados (linha 4). A partir dos vetores recebidos como parâmetro percorremos cada um deles (linhas 10-67). Em cada posição do vetor x, existe um valor que corresponde a posição no eixo x da imagem original, e o mesmo vale para o vetor y. As coordenadas no eixo x e y correspondem a um ponto central da imagem, e a partir desse ponto capturamos um intervalo de *pixels* que estão ao redor, que é calculado para o eixo x nas linhas 13 e 17, e no eixo y nas linhas 21 e 25. Afim de evitar, valores que não poderão ser acessados criamos uma condição (linha 32), verificando se os valores dos intervalos são válidos. Caso seja válido o intervalo de *pixels* para o eixo x e eixo y, capturamos essa sub-região na imagem original (linha 34). Com esses pedaços da

imagem, que são matrizes bidimensionais, transformamos eles em um arranjo de uma única dimensão (linhas 43-48). Com esse arranjo de uma única dimensão, normalizamos ele (linhas 51-62). Porém, durante a normalização tivemos alguns casos onde os maiores e menores valores eram iguais, e consequentemente resultando num valor igual a zero e uma divisão por zero não existe, por isso foi criada a condição presente na linha 57. Após a normalização, armazenamos na variável *features*(linha 65), e após rodar para todos as coordenadas de x e y, é retornado essa variável.

Listing 1: código da função *get_features_patch()*

```

1 function [features] = get_features_patch(image, x, y,
2                                         descriptor_window_image_width)
3 % Matriz que armazena valores que foram
4 % capturados das posições
5 features = zeros(size(x,1), 289, 'single');
6
7 % Pego o tamanho da imagem que foi passado por parametro
8 % , isso porque
9 % no for eu posso ter valores que ultrapassem o tamanho
10 % original da imagem
11 [xOriginal yOriginal] = size(image);
12
13 for k = 1:size(x)
14     % Dada uma imagem, pegar uma feature da imagem 16x16
15     % Pega a posição minima do eixo x
16     x1 = x(k) - (descriptor_window_image_width / 2);
17     x1 = uint32(x1);
18
19     % Pega a posição maxima do eixo x
20     x2 = x(k) + (descriptor_window_image_width / 2);
21     x2 = uint32(x2);
22
23     % Pega a posição minima do eixo y
24     y1 = y(k) - (descriptor_window_image_width / 2);
25     y1 = uint32(y1);
26
27     % Pega a posição maxima do eixo y
28     y2 = y(k) + (descriptor_window_image_width / 2);
29     y2 = uint32(y2);
30
31     % Evita que valores negativos sejam acessados
32     if(x1 > 0 && x2 > 0 && y1 > 0 && y2 > 0)
33         % Depois verifico se não tem algum valor que vai
34         % uma posição que não esteja dentro dos valores da
35         % foto
36         if(x1 <= xOriginal && x2 <= xOriginal && y1 <=

```

```

33     yOriginal && y2 <= yOriginal)
34     % Pego um intervalo da imagem original
35     pieceImage = image(x1 : x2, y1: y2);
36
37     % Pega as dimensões de X e Y
38     [pieceX pieceY] = size(pieceImage);
39
40     % Faço um array unidimensional
41     z = 1;
42
43     % Coloco os valores no array
44     for i = 1:pieceX
45         for j = 1:pieceY
46             array(z) = pieceImage(i, j);
47         z++;
48     end
49
50     % Normalizo esse array
51     maxValue = max(array);
52     minValue = min(array);
53
54     % Percorro o vetor e vou normalizando o mesmo
55     for i = 1:size(array)
56         % Quando minValue e maxValue era zero
57         if(minValue == maxValue)
58             array(i) = 1;
59         else
60             array(i) = (array(i) - minValue) / (maxValue -
61                 minValue);
62         endif
63     end
64
65     % Tem que transpor o array para caber na matriz de
66     % features
67     features(k, :) = array';
68     endif
69   endfor
70 endfunction

```

Após ter implementado a função *get_features_patch()* e ter aplicado para as duas imagens, teremos duas matrizes bidimensionais. Com esses arranjos bidimensionais, iremos realizar um cálculo para calcular a distância entre as duas *features* passadas por parâmetro, na função *match_features()* (código 2). O cálculo basicamente envolve realizar a subtração da linha um de uma das *features* menos a linha um da outra. Com o

resultado dessa subtração, somamos todos os valores de todas as posições e elevamos ao quadrado (linhas 12-13). Depois da linha um da *feature* um com a linha dois da *feature* dois, e assim por diante. Esse processo foi feito também com todas as linhas da *feature* um com todas as linhas da *feature* dois. Durante esse processo obtivemos o primeiro e o segundo menor valor e a linha que ocorre (linhas 17-19) e guardamos as linhas na matriz *matches* (linhas 24-25). Com o primeiro e o segundo menores valores, calculamos a distância através da fórmula 1, onde $d1$ é o menor valor e $d2$ é o segundo menor valor, e armazenamos em um vetor chamado *confidences* (linha 29-30). Porém, na hora de calcular a distância, também tivemos problemas com o resultado das divisões, pois davam igual a zero, para sanar esse problema foi atribuído o menor valor como sendo a distância, por isso a existência da condição na linha 28. Por fim, ordenamos os valores presentes no vetor *confidences* com todas as distâncias em ordem crescente(linha 36), na qual esse conjunto de valores e a matriz *matches* são os valores de retorno dessa função.

$$NNDR = \frac{d1}{d2} \quad (1)$$

Listing 2: código da função *match_features()*

```

1 function [matches, confidences] = match_features(
2     features1, features2)
3 % Percorro a linha das features1
4 for i = 1:size(features1, 1)
5     % Acha o primeiro e o segundo menor valor
6     minValue = e^200;
7     minAnterior = e^200;
8     minJ = 1;
9
10    % Percorro a linha das features2
11    for j = 1:size(features2, 1)
12        % Calcula a diferença entre as duas imagens
13        value = features1(i, :) - features2(j, :);
14        value = sum(value.^2);
15
16        % Pego o menor valor e o segundo menor valor
17        if(value < minValue)
18            minAnterior = minValue;
19            minValue = value;
20            minJ = j;
21        endif
22    end
23
24    % Guardo a posição do menor valor
25    matches(i, 1) = i;
26    matches(i, 2) = minJ;

```

```

27 % NNDR = d1 / d2 = ||Da-Db| / ||Da-Dc| formula 4.18
28 if(minValue / minAnterior == 0)
29     confidences(i) = minValue;
30 else
31     confidences(i) = 1 / (minValue / minAnterior);
32 end
33 end
34
35 % Ordeno o vetor de confidências
36 [confidences, ind] = sort(confidences, 'ascend');
37 endfunction

```

Após ter implementado as funções `get_features_patch()` e `match_features()`, partimos para a implementação do SIFT *Scale-Invariant Feature Transform*, na função `get_features_sift()`. Aproveitamos boa parte do código, da função `get_features_patch()` e fizemos algumas alterações, (código 3). As principais diferenças entre as funções é o que acontece depois de obter um intervalo da imagem, à partir da linha 37. Com um pedaço da imagem, que tem 16 linhas por 16 colunas, dividimos em oito pedaços cada um deles contendo quatro linhas e quatro colunas, que é feito no `for` das linhas 43-60. Para cada um desses pedaços calculamos a magnitude através de uma função implementada por nós mesmo (linha 49), chamada `calc_magnitude()` (código 4), na qual iremos explicar o seu funcionamento posteriormente, mas basicamente adicionamos os valores retornados por essa função na matriz de `features` (linha 53). Esse processo citado anteriormente, é feito para todos os oitos pedaços. Após ter sido calculado a magnitude para as oito sub-regiões, aplicamos um filtro Gaussiano (linhas 64-65), com as dimensões de 16 por 16, já que o pedaço da nossa imagem contém as mesmas dimensões, e um valor de sigma igual a 2, já que para esse valor tivemos poucos valores igual a zero. Após isso normalizamos os valores (linhas 68-78), e mais uma vez como na função `get_features_patch()` ocorreu de nós termos maiores valores e menores valores que são iguais, por isso da condição na linha 72.

Listing 3: código da função `get_features_sift()`

```

1 function [features] = get_features_sift(image, x, y,
2                                         descriptor_window_image_width)
3
4 %Placeholder that you can delete. Empty features.
5 features = zeros(size(x,1), 289, 'single');
6
7 % Pego o tamanho da imagem que foi passado por parametro
8 % , isso porque
9 % no for eu posso ter valores que ultrapassem o tamanho
10 % original da imagem
11 [xOriginal yOriginal] = size(image);
12
13 % O x e y vindo por parametro sao vetores, com as
14 % posicoes

```

```
11 % que foram atribuidas no cheat_interest_points
12 % O tamanho do vetor de x e y sao iguais
13 for k = 1:size(x)
14     % Dada uma imagem, pegar uma feature da imagem 16x16
15     % Pega a posicao minima do eixo x
16     x1 = x(k) - (descriptor_window_image_width / 2);
17     x1 = uint32(x1);
18
19     % Pega a posicao maxima do eixo x
20     x2 = x(k) + (descriptor_window_image_width / 2 - 1);
21     x2 = uint32(x2);
22
23     % Pega a posicao minima do eixo y
24     y1 = y(k) - (descriptor_window_image_width / 2);
25     y1 = uint32(y1);
26
27     % Pega a posicao maxima do eixo y
28     y2 = y(k) + (descriptor_window_image_width / 2 - 1);
29     y2 = uint32(y2);
30
31     % Evita que valores negativos sejam acessados
32     if(x1 > 0 && x2 > 0 && y1 > 0 && y2 > 0)
33
34         % Depois verifico se nao tem algum valor que vai
            acessar
            % uma posicao que nao esteja dentro dos valores da
            foto
35         if(x1 <= xOriginal && x2 <= xOriginal && y1 <=
            yOriginal && y2 <= yOriginal)
            % Pego um intervalo da imagem original
36         pieceImage = image(x1 : x2, y1: y2);
37
38         % A partir do pedaço da imagem que retiramos
            quebramos ela em 8 pedacos
39         pieceY1 = 1;
40         pieceY2 = 4;
41         for i = 1:4
42             pieceX1 = 1;
43             pieceX2 = 4;
44             for j = 1:4
45                 % Para esse pedaco de imagem calculamos
                    % a magnitude desse pedaco
46                 result = calc_magnitude(pieceImage(pieceX1 :
                    pieceX2, pieceY1 : pieceY2));
47
48                 % Coloco no vetor de features
49
50
51
```

```

52         pos = (i - 1) * 4 + j;
53         features(k, ((pos - 1) * 8) + 1 : ((pos - 1)
54             * 8) + 8) = result;
55
56         pieceX1 = pieceX2 + 1;
57         pieceX2 = pieceX1 + 3;
58     end
59         pieceY1 = pieceY2 + 1;
60         pieceY2 = pieceY1 + 3;
61     end
62 end
63 % Aplico o gaussiano
64 filtroGaussiano = fspecial('gaussian', [16 16], 2);
65 features(k, :) = conv2(features(k, :), filtroGaussiano,
66                         'same');
67
68 % Pego o maior e o menor
69 maxValue = max(features(k, :));
70 minValue = min(features(k, :));
71
72 % Percorro o vetor e vou normalizando o mesmo
73 if(minValue == maxValue)
74     features(k, :) = 1;
75 else
76     for z = 1:size(features, 2)
77         features(k, z) = (features(k, z) - minValue) / (
78             maxValue - minValue);
79     end
80 end
end

```

Afim de facilitar a compreensão e legibilidade do código, criamos uma função para o cálculo da magnitude (chama de *calc_magnitude()*, que é invocada na linha 48 do código 3) que recebe como parâmetro um pedaço da imagem original (código 4). Então primeiramente montamos o filtro de Sobel na horizontal e na vertical (linhas 3-22). Após isso, com o filtro feito montamos uma borda na imagem recebida como parâmetro e aplicamos os dois filtros na imagem original (linhas 25-28), e calculamos a magnitude através do módulo da imagem na qual foi aplicada o filtro na horizontal e na vertical (linha 30). Depois através da função *atan2* obtemos os valores em radiano das imagens que foram aplicadas os filtros(linha 33). Com os valores de radianos, convertemos os mesmos para graus através da seguinte equação 2 (e no código na linha 36). Como todos valores convertidos para graus, contabilizamos a quantidade de ângulos que estão nos seguintes intervalos: 0° a 45°, 45° a 90°, 90° a 135°, 135° a 180°, 180° a 225°, 225° a 270°, 270° a 315° e 315° a 360°, para representar esses oito intervalos possíveis criamos um

vetor de tamanho oito (linha 39). A contabilização é feita no *for* das linhas 42-49, em que o cálculo para descobrir o intervalo que o ângulo pertence é dada pela equação , (e no código na linha 47). Porém, durante alguns teste que realizamos percebemos que alguns pontos que foram retornados lá pela função *cheat_interest_points()*, retornavam pontos que acarretavam em ângulos superiores a 360° , por isso foi feito a condição da linha 44, em que para valores superiores ou igual a 360° consideramos que está no último intervalo.

$$rad2deg = (radiano * \frac{180}{\pi}) + 180 \quad (2)$$

$$intervalo = \left\lfloor \frac{\hat{\text{ângulo}}}{45} \right\rfloor$$

Listing 4: código da função *calc_magnitude()*

```

1 function [result] = calc_magnitude(image)
2 % Cria os filtros
3 filtroSobelH = zeros(3, 3, "int8");
4 filtroSobelV = zeros(3, 3, "int8");
5
6 % Monta o filtro na horizontal
7 filtroSobelH(1, 1) = -1;
8 filtroSobelH(1, 2) = -2;
9 filtroSobelH(1, 3) = -1;
10
11 filtroSobelH(3, 1) = 1;
12 filtroSobelH(3, 2) = 2;
13 filtroSobelH(3, 3) = 1;
14
15 % Monta o filtro na vertical
16 filtroSobelV(1, 1) = -1;
17 filtroSobelV(2, 1) = -2;
18 filtroSobelV(3, 1) = -1;
19
20 filtroSobelV(1, 3) = 1;
21 filtroSobelV(2, 3) = 2;
22 filtroSobelV(3, 3) = 1;
23
24 % Replicando a magnitude
25 imagemSobel = padarray(image, [1 1], "replicate");
26 imagemSobelX = filter2(filtroSobelH, imagemSobel, 'valid
   ');
27 imagemSobelY = filter2(filtroSobelV, imagemSobel, 'valid
   ');
28

```

```

29 % Calculando a magnitude
30 magnitude = abs(imagemSobelX) + abs(imagemSobelY);
31
32 % Calcula o arctangente
33 direcoes = atan2(imagemSobelY, imagemSobelX);
34
35 % Converte radiano para graus
36 direcoes = (direcoes * (180 / pi)) + 180;
37
38 % Crio um vetor de 8 linhas, porque podemos ter oito
   ângulos
39 result = zeros(1, 8, "int8");
40
41 % Percorro vetor de direcoes
42 for x = 1 : size(direcoes(:, 1)
43   % Alguns casos o ângulo estava dando um valor acima de
      360°
44   if(floor(direcoes(:, x) / 45) >= 8)
45     result(1, 8) += magnitude(:, x);
46   else
47     result(1, floor(direcoes(:, x) / 45) + 1) +=
        magnitude(:, x);
48   end
49 end
50 end

```

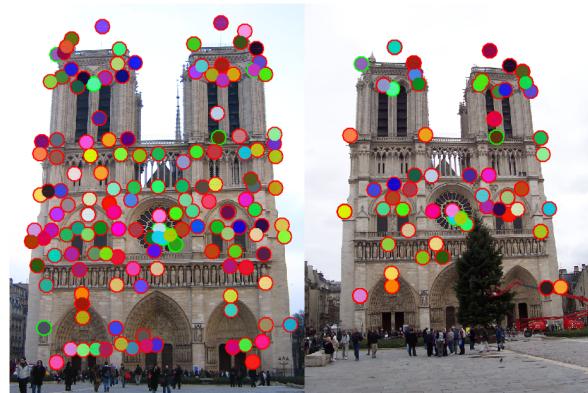
A Result

Os resultados obtidos foram realizados nas imagens da igreja de Notre Dame, do Monte Rushmore e do palácio Episcopal de Gaudi. Para cada uma delas aplicamos as funções *get_features_patch()* e *get_features_sift()*, sendo que na segunda função, realizamos a aplicação do filtro Gaussiano e sem o filtro Gaussiano.

Para todos os casos e as imagens na qual executamos, esperávamos resultados de acurácia mais próximos do que estimado, entretanto não foi isso que aconteceu. Os resultados que iremos mostrar a seguir podem ser vistos com mais detalhes na pasta chamada *results*. O resultado aplicando a função *get_features_patch()* pode ser visto na tabela 1, e as imagens com *features* da igreja de Notre Dame (figura 1), do Monte Rushmore (2) e do palácio Episcopal de Gaudi (figura 3), respectivamente.

Table 1: resultados obtidos com a função *get_features_patch()*

Imagen	Pontos bons / ruins / total	Precisão	Acurácia obtida / esperada
Notre Dame	9 / 140 / 149	6,04%	9,00% / \approx 40%
Monte Rushmore	1 / 125 / 126	0,79%	1% / \approx 25%
Episcopal Gaudi	0 / 146 / 146	0%	0% / \approx 7%

Figure 1: localização das *features* usando a função *get_features_patch()* na igreja de Notre Dame.Figure 2: localização das *features* usando a função *get_features_patch()* do Monte Rushmore.

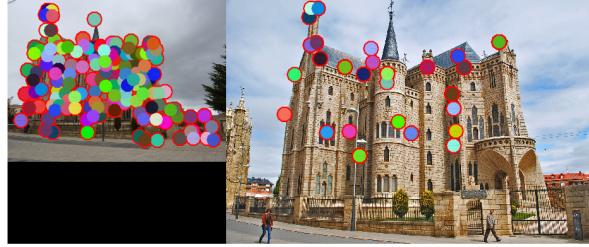


Figure 3: localização das *features* usando a função *get_features_patch()* do palácio Episcopal Gaudi.

Como dito anteriormente, o resultado foi surpreendente, pois esperávamos um resultado muito bom, mas não chegamos próximo do valor esperado. Não sabemos se as alterações que fizemos no algoritmo *get_features_patch()* e *match_features()*, já que existiam alguns casos que foram necessários tratar, como a condição da linha 57 no *get_features_patch()* em que conosco aconteceu o caso de termos o maior valor e o menor valor sendo igual, consequentemente ocasionando a divisão por zero. Isso também ocorreu na função *match_features()*, em que tivemos que tratar uma condição (linha 28) que está imprevista, em que a divisão do primeiro menor valor e do segundo menor valor é igual a zero. Já para a função *get_features_sift()*, não foi diferente da primeira função, os valores de acurácia obtido foram bem inferiores ao que eram esperados, tanto quando aplicamos o filtro Gaussiano quanto sem. Porém, o que nos surpreendeu foi a diferença entre a aplicação e sem a aplicação do filtro Gaussiano. Os resultados obtidos podem ser visto na tabela 2 e 3.

Table 2: resultados obtidos com a função *get_features_sift()* e usando o filtro Gaussiano

Imagen	Pontos bons / ruins / total	Precisão	Acurácia obtida / esperada
Notre Dame	4 / 145 / 149	2,68%	4% / ≈70%
Mount Rushmore	2 / 124 / 126	1,59%	2% / ≈40%
Episcopal Gaudi	1 / 145 / 146	0,68%	1% / ≈15%

Table 3: resultados obtidos com a função *get_features_sift()* e sem o filtro Gaussiano

Imagen	Pontos bons / ruins / total	Precisão	Acurácia obtida / esperada
Notre Dame	6 / 143 / 149	4,03%	6% / ≈70%
Mount Rushmore	2 / 124 / 126	1,59%	2% / ≈40%
Episcopal Gaudi	2 / 145 / 146	1,37%	2% / ≈15%

Comparando as duas tabelas acimas, podemos notar que quando não usamos o filtro Gaussiano o resultado que obtivemos foram dois casos com um resultado superior e um que se manteve o mesmo resultado, porém deveria ter sido ao contrário. Tentamos outros valores de sigma quando construímos o filtro Gaussiano, porém percebemos que para valores de sigma muito altos, os valores de acurácia ficaram bem piores. Além disso, acreditamos que talvez isso tenha acontecido, pois talvez tenhamos tratado de maneira errônea a condição da linha 44 da função `calc_magnitude()`, em casos que os valores de magnitude tem um ângulo superior a 360°. Outro caso que pensamos que talvez também tenhamos ter tratado de maneira errada foi na aplicação do filtro Gaussiano (linha 65 da função `get_features_sift()`), no momento da normalização para valores que tem o maior e o menor valor sendo igual (linha 72 da função `get_features_sift()`) ou na função `match_features_()`, onde tivemos que tratar o caso da divisão do primeiro menor valor e o segundo valor ocasionavam em uma divisão que não existe (linha 28). A seguir estão demonstradas as *features* de cada uma das duas imagens com a aplicação do filtro Gaussiano e sem o filtro, na respectiva ordem, primeiramente da igreja de Notre Dame, o Monte Rushmore e o palácio Episcopal Gaudi (figuras: [4](#), [5](#), [6](#), [7](#), [8](#) e [9](#)).

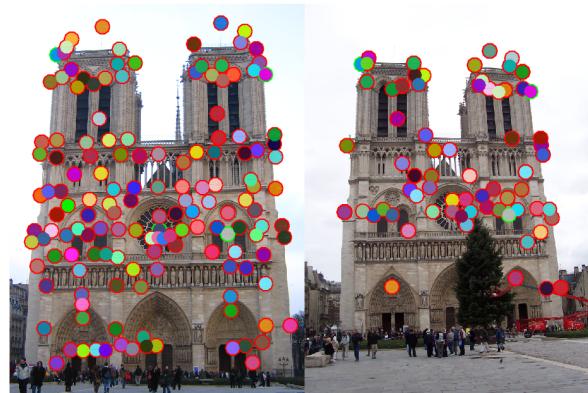


Figure 4: localização das *features* usando a função `get_features_sift()` na igreja de Notre Dame e com o filtro Gaussiano.



Figure 5: localização das *features* usando a função `get_features_sift()` na igreja de Notre Dame e sem o filtro Gaussiano.



Figure 6: localização das *features* usando a função `get_features_sift()` no Monte Rushmore e com o filtro Gaussiano.



Figure 7: localização das *features* usando a função *get_features_sift()* no Monte Rushmore e sem o filtro Gaussiano.



Figure 8: localização das *features* usando a função *get_features_sift()* no palácio Episcopal Gaudi e com o filtro Gaussiano.

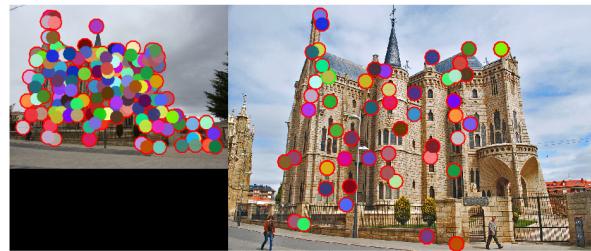


Figure 9: localização das *features* usando a função `get_features_sift()` no palácio Episcopal Gaudi e sem o filtro Gaussiano.

Enfim, acreditamos que os resultados obtidos em ambos os casos (utilizando as funções `get_features_patch()` e `textit{get_features_sift()}`) podem ter ocorrido, porque esquecemos de algum passo ou porque tratamos de maneira errada alguma etapa. Com relação a função do SIFT, acreditamos que se tivéssemos implementado ela por completa, e não somente apenas uma parte dela o resultado teria sido melhor do que foi obtido.

References

SZELISKI, R. *Computer Vision: Algorithms and Applications*. [S.l.]: Springer, 2010.