

TP4 - Imagerie 4D

Ce TP à été réalisé par **Odorico Thibault** et **Isnel Maxime** le **Mercredi 27/11/2019**.

TP4 - Imagerie 4D

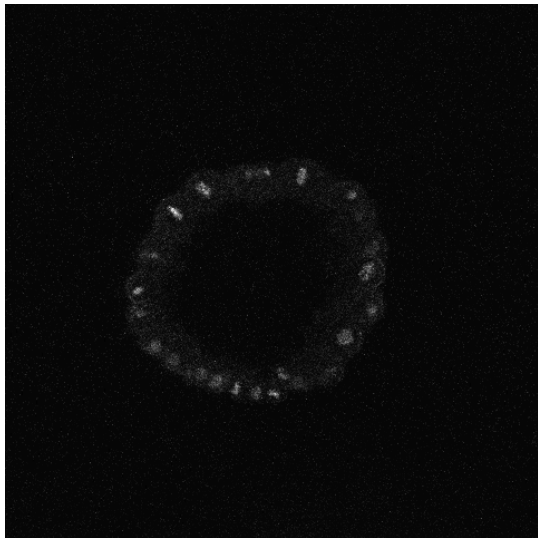
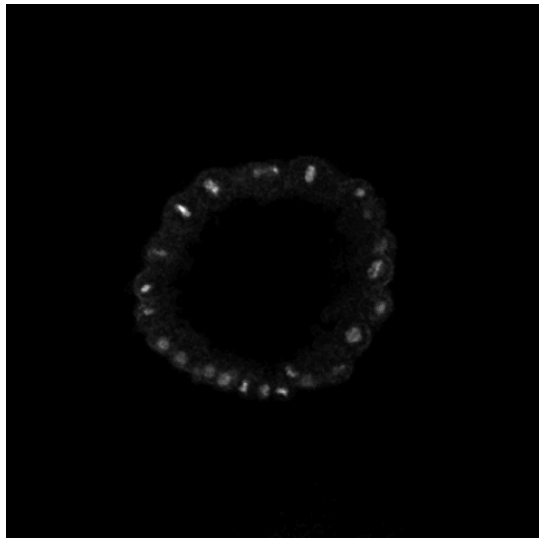
Suivi de cellules

1. Adoucissement de l'image
2. Segmentation et Morphologie
3. Translation

[Code Source](#)

Suivi de cellules

1. Adoucissement de l'image

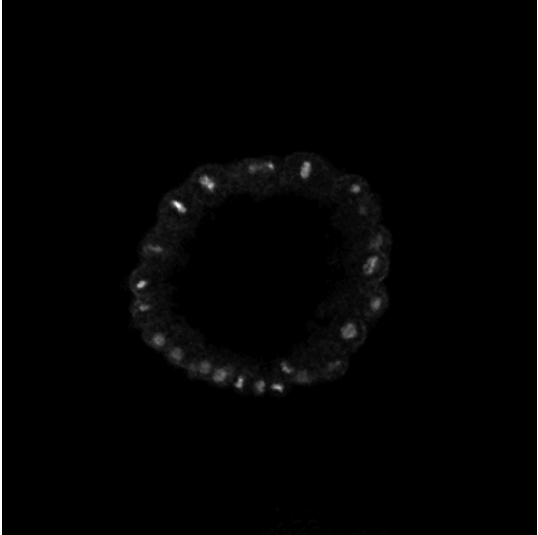
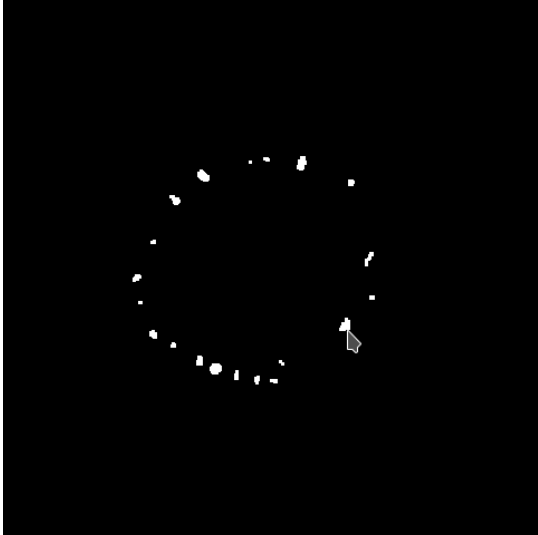
Image brut	Image filtre médian
	

Comme on peut le voir sur les images ci-dessus, le filtre médian permet d'adoucir grandement le bruit des images et de mettre en évidence les structures importantes de l'image. C'est donc un bon outils pour faciliter la segmentation des cellules dans notre cas.

2. Segmentation et Morphologie

Avec notre programme l'utilisateur doit cliquer au centre d'une cellule dont l'intensité est dans la moyenne des autres cellule.

Au clic de l'utilisateur la moyenne des pixels M_p (dans un rayon de 4 pixels autours du curseur) est calculée (la moyenne permet de réduire les erreurs d'imprécisions). Ensuite on seuil l'image avec cette valeur M_p et pour finir on applique à la suite : une dilatation, 2 érosions et enfin une dilatation sur les voxels pour éliminer le bruit restant.

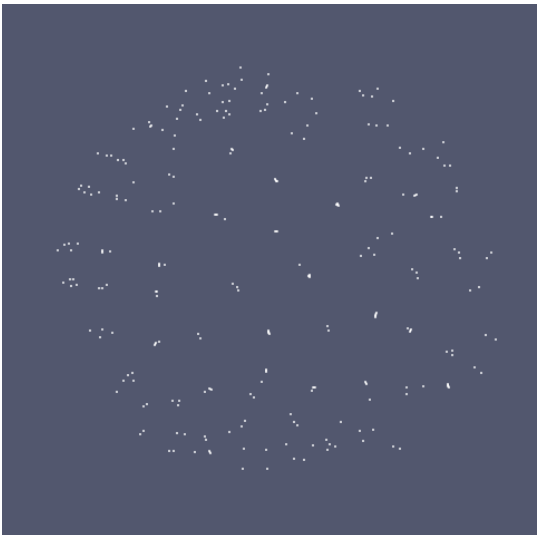
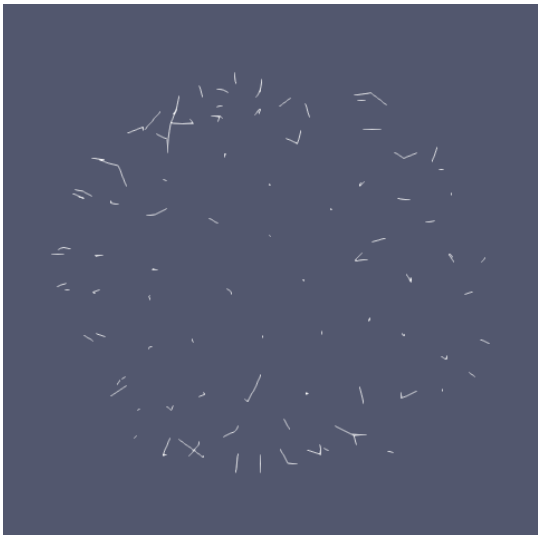
Image de référence	Image segmenté + opération morphologique
	

3. Translation

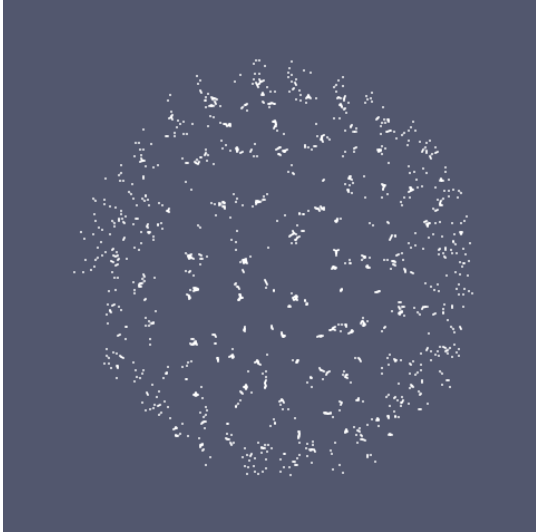
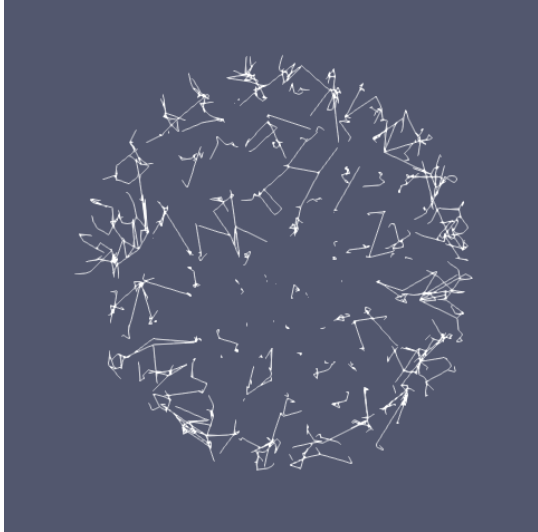
CImg permet de labelliser les structures isolés dans une image, cela nous permet de calculer une position pour chaque cellules facilement.

Pour connaître le déplacement des cellules dans chaque images 3D il nous suffit ensuite de calculer pour chaque position d'une images I_i la différence avec les positions les plus similaires dans l'image I_{i+1} on obtient ainsi les vecteurs de déplacement de chaque cellules de I_i vers I_{i+1} .

En écrivant ces vecteurs dans un format .obj puis en les lisant avec ParaView on obtiens les trajectoires 3D suivantes :

Image stack-[0-3].hdr Points	Image stack-[0-3].hdr Trajectoires
	

Avec une plus grande quantité d'images 3D on obtient les trajectoires suivantes :

Image stack-[0-20].hdr Points	Image stack-[0-20].hdr Trajectoires
	

Les trajectoires formées par les lignes ont bien l'air cohérentes avec le mouvement qu'accomplissent les cellules dans le temps. Cependant il aurait tout de même pu être possible d'améliorer les résultats en se basant sur d'autres critères que la distance la plus proche entre deux points de 2 images. On aurait possiblement pu se baser sur un système multi critère, par exemple en se basant sur la **similarité de position** comme nous l'avons fait mais en ajoutant un critère de **similarité d'intensité de voxel** représentant une cellule. On peut très bien imaginer pouvoir attribuer un poids à chaque critère pour faire varier leur importance et ainsi obtenir des résultats plus ou moins bon.

Code Source

```

1  #include <iostream>
2  #include <algorithm>
3  #include <limits>
4  #include <string>
5  #include <array>
6  #include <vector>
7  #include <thread>
8  #include <chrono>
9  #include <map>
10
11 #include "CImg.h"
12
13 using namespace cimg_library;
14 using namespace std::chrono_literals;
15
16 template<class T>
17 using vec3 = std::array<T, 3>;
18 using vec3i = vec3<int>;
19 using vec3u = vec3<unsigned int>;
20 using vec3f = vec3<float>;
21 using vec3d = vec3<double>;
22
23 template<class T>
24 vec3<T>& operator+=(vec3<T>& left, const vec3<T>& right)
25 {
26     left[0] += right[0];

```

```

27     left[1] += right[1];
28     left[2] += right[2];
29
30     return left;
31 }
32
33 template<class T>
34 vec3<T> operator-(const vec3<T>& left, const vec3<T>& right)
35 {
36     return {left[0] - right[0],
37             left[1] - right[1],
38             left[2] - right[2]};
39 }
40
41 template<class T, class U>
42 vec3<T>& operator/=(vec3<T>& left, U value)
43 {
44     left[0] /= value;
45     left[1] /= value;
46     left[2] /= value;
47
48     return left;
49 }
50
51 template<class T, class U>
52 vec3<T> operator/(const vec3<T>& left, U value)
53 {
54     return {left[0] / value,
55             left[1] / value,
56             left[2] / value};
57 }
58
59 template<class T, class U>
60 vec3<T> to_vec3(const vec3<U>& v)
61 {
62     return {(T)v[0], (T)v[1], (T)v[2]};
63 }
64
65 template<class T>
66 double distance_squared(const vec3<T>& left, const vec3<T>& right)
67 {
68     double dist = 0;
69
70     for (size_t i = 0 ; i < left.size() ; i++)
71         dist += pow(left[i] - right[i], 2);
72
73     return dist;
74 }
75
76 template<typename T>
77 bool operator<(const vec3<T>& left, const vec3<T>& right)
78 {
79     if (left[0] < right[0])
80     {
81         return true;
82     }
83     else if (left[0] == right[0])
84     {

```

```

85         if (left[1] < right[1])
86         {
87             return true;
88         }
89         else if (left[1] == right[1])
90         {
91             return left[2] < right[2];
92         }
93     }
94
95     return false;
96 }
97
98 template<class T>
99 std::ostream& operator<<(std::ostream& os, const vec3<T>& v)
100 {
101     return os << '(' << v[0] << ", " << v[1] << ", " << v[2] << ')';
102 }
103
104 template<class T>
105 std::ostream& operator<<(std::ostream& os, const std::vector<T>& vect)
106 {
107     for (const auto& v : vect)
108         os << v << '\n';
109
110     return os;
111 }
112
113 // renvoie la moyenne des intensité dans une région dont le centre est
114 // vox_coord
115 double region_intensity(const CImg<float>& img, const vec3i& vox_coord,
116 int region_radius)
117 {
118     double sum = 0;
119     unsigned int count = 0;
120
121     int min_x = vox_coord[0] - region_radius;
122     int min_y = vox_coord[1] - region_radius;
123
124     min_x = min_x < 0 ? 0 : min_x;
125     min_y = min_y < 0 ? 0 : min_y;
126
127     for (int x = min_x ; x < min_x + 1 + 2 * region_radius; ++x)
128     {
129         for (int y = min_y ; y < min_y + 1 + 2 * region_radius ; ++y)
130         {
131             sum += img(x, y, vox_coord[2]);
132             count++;
133         }
134     }
135
136     return sum / count;
137 }
138
139 // il faut que l'image soit seuillée avant de récupérer les centroids
140 std::vector<vec3d> get_cells_centroid(const CImg<float>& thresholded_img)
141 {
142     std::cerr << "Obtention des labels pour chaque cellules...\n";

```

```

141
142     auto labels = thresholded_img.get_label();
143
144     std::map<size_t, vec3d> label_pos_sum_map;
145     std::map<size_t, size_t> label_pos_count_map;
146
147     std::cerr << "Calcul des barycentres de chaque cellules...\n";
148
149     for (int x = 0 ; x < labels.width() ; ++x)
150     {
151         for (int y = 0 ; y < labels.height() ; ++y)
152         {
153             for (int z = 0 ; z < labels.depth() ; ++z)
154             {
155                 auto label = labels(x, y, z);
156
157                 label_pos_sum_map[label] += {(double)x, (double)y,
(double)z};
158                 label_pos_count_map[label] ++;
159             }
160         }
161     }
162
163     std::cerr << "Nombre de barycentres trouvés : " <<
label_pos_sum_map.size() << '\n';
164
165     std::vector<vec3d> barycentres(label_pos_sum_map.size());
166
167     size_t i = 0;
168
169     for (auto& pair : label_pos_sum_map)
170     {
171         auto& label = pair.first;
172         auto& barycentre_sum = pair.second;
173
174         barycentres[i] = barycentre_sum / label_pos_count_map[label];
175         i++;
176     }
177
178     return barycentres;
179 }
180
181 // recupère le centroid le plus proche de "centroid" dans "centroids"
182 vec3d get_closest_centroid(const vec3d& centroid, const
std::vector<vec3d>& centroids)
183 {
184     auto closest = [&centroid](const vec3d& left, const vec3d& right){
return distance_squared(centroid, left) < distance_squared(centroid,
right) ? true : false; };
185
186     return *std::min_element(centroids.begin(), centroids.end(), closest);
187 }
188
189 std::vector<vec3d> get_translations(const std::vector<vec3d>& left, const
std::vector<vec3d>& right)
190 {
191     size_t min_size = std::min(left.size(), right.size());
192

```

```

193     std::vector<vec3d> translations(min_size);
194
195     for (size_t i = 0 ; i < min_size ; i++)
196     {
197         translations[i] = get_closest_centroid(left[i], right) - left[i];
198     }
199
200     return translations;
201 }
202
203 // ecrit la trajectoire des cellule dans un format .obj
204 void write_3d_trajectory(std::ostream& os, const
std::vector<std::vector<vec3d>>& image_4d_centroids, const
std::vector<std::vector<vec3d>>& image_4d_translations)
205 {
206     size_t line_count = 0;
207
208     std::cerr << "Ecriture des sommets...\n";
209
210     for (size_t i = 0 ; i < image_4d_translations.size() ; i++)
211     {
212         for (size_t j = 0 ; j < image_4d_translations[i].size() ; j++)
213         {
214             os << "v " << image_4d_centroids[i][j][0] << ' '
215                 << image_4d_centroids[i][j][1] << ' '
216                 << image_4d_centroids[i][j][2] << '\n';
217             os << "v " << image_4d_centroids[i][j][0] +
image_4d_translations[i][j][0] << ' '
218                 << image_4d_centroids[i][j][1] +
image_4d_translations[i][j][1] << ' '
219                 << image_4d_centroids[i][j][2] +
image_4d_translations[i][j][2] << '\n';
220
221             line_count++;
222         }
223     }
224
225     std::cerr << "Ecriture des indices de (" << line_count << ")
lignes...\n";
226
227     if (line_count > 0)
228     {
229         for (size_t i = 0 ; i < line_count ; i+=2)
230         {
231             os << "l " << i + 1 << ' ' << i + 2 << '\n';
232         }
233     }
234 }
235
236 int main(int argc, char **argv)
237 {
238     if(argc < 2)
239     {
240         std::cerr << "Usage : " << argv[0] << " <image-0.hdr> <image-
1.hdr>... <image-N.hdr>\n";
241         exit(EXIT_FAILURE);
242     }
243

```

```

244     std::cerr << "Récupération du nom des images...\n";
245
246     size_t i = 1;
247     std::vector<std::string> names;
248
249     while (argv[i] != NULL)
250     {
251         names.push_back(argv[i]);
252         std::cerr << argv[i] << '\n';
253         i++;
254     }
255
256     std::cerr << "Lecture des images...\n";
257
258     vec3i image_3d_dimensions;
259     vec3f image_3d_voxel_size;
260     std::vector< CImg<float> > image_4d(names.size());
261
262     i = 0;
263
264     for (const auto& name : names)
265     {
266         image_4d[i].load_analyze(name.c_str(),
image_3d_voxel_size.data());
267         image_3d_dimensions = {image_4d[i].width(), image_4d[i].height(),
image_4d[i].depth()};
268
269         std::cerr << "Dimensions[" << i << "] = " <<
image_3d_dimensions[0] << ", " << image_3d_dimensions[1] << ", " <<
image_3d_dimensions[2] << ")\n";
270         std::cerr << "Voxel_size[" << i << "] = " <<
image_3d_voxel_size[0] << ", " << image_3d_voxel_size[1] << ", " <<
image_3d_voxel_size[2] << ")\n";
271
272         i++;
273     }
274
275     std::cerr << "Application du filtre médian...\n";
276
277     for (auto& image_3d : image_4d)
278     {
279         image_3d.blur_median(1);
280     }
281
282     std::cerr << "Création d'un pointer d'image 3d courante...\n";
283
284     size_t current_image_3d_index = 0;
285     CImg<float> current_image_3d = image_4d[current_image_3d_index];
286
287     std::cerr << "Creation de l'affichage...\n";
288
289     CImgDisplay visu_window(current_image_3d, "Visualisation");
290
291     CImg<float> visu(500, 400, 1, 3, 0);
292
293     vec3i displayed_slice = {current_image_3d.width()/2,
current_image_3d.height()/2, current_image_3d.depth()/2};
294

```



```

295     /* Slice corresponding to mouse position: */
296     vec3i coord = {0, 0, 0};
297
298     /* The display visu_window corresponds to a MPR view which is
decomposed into the following 4 quadrants:
299     2 = original slice size=x y      0 size = z y
300     1 = size = x z                  -1 corresponds to the 4th quarter
where there is nothing displayed */
301     int visu_window_plane = 2;
302     bool visu_window_redraw = true;
303
304     bool thresholded = false;
305     bool centroided = false;
306     bool translated = false;
307
308     std::vector<std::vector<vec3d>> image_4d_centroids(image_4d.size());
309     std::vector<std::vector<vec3d>> image_4d_translations(image_4d.size()
- 1);
310
311     std::cerr << "Veuillez cliquer a l'interieur d'une cellule ayant un
couleur représentative pour seuillez l'image !\n";
312
313     while(!visu_window.is_closed() && !visu_window.is_keyESC()) // Main
loop
314     {
315         // Affiche l'image 3D suivante
316         if(visu_window.is_keyARROWRIGHT())
317         {
318             if (current_image_3d_index + 1 < image_4d.size())
319                 current_image_3d_index++;
320             else
321                 current_image_3d_index = 0;
322
323             current_image_3d = image_4d[current_image_3d_index];
324
325             // std::cerr << "current_image_3d_index : " <<
current_image_3d_index << '\n';
326             visu_window_redraw = true;
327             std::this_thread::sleep_for(0.1s);
328         }
329
330         // Affiche l'image 3D precedente
331         if(visu_window.is_keyARROWLEFT())
332         {
333             if (current_image_3d_index > 0)
334                 current_image_3d_index--;
335             else
336                 current_image_3d_index = image_4d.size() - 1;
337
338             current_image_3d = image_4d[current_image_3d_index];
339
340             // std::cerr << "current_image_3d_index : " <<
current_image_3d_index << '\n';
341             visu_window_redraw = true;
342             std::this_thread::sleep_for(0.1s);
343         }
344
345         // Réinitialise l'image visualisée

```

```

346     if(visu_window.is_key('r'))
347     {
348         current_image_3d = image_4d[current_image_3d_index];
349         visu_window_redraw = true;
350     }
351
352     if(visu_window.is_key('e'))
353     {
354         current_image_3d.erode(3, 3, 3);
355         visu_window_redraw = true;
356     }
357
358     if(visu_window.is_key('d'))
359     {
360         current_image_3d.dilate(3, 3, 3);
361         visu_window_redraw = true;
362     }
363
364     // Sauvegarde l'image
365     if(visu_window.is_key('s'))
366     {
367         current_image_3d.save_analyze("out.hdr",
image_3d_voxel_size.data());
368     }
369
370     // Traitement d'image par défaut
371     if(visu_window.is_key('f'))
372     {
373         current_image_3d.threshold(28);
374         current_image_3d.dilate(3, 3, 3);
375         current_image_3d.erode(3, 3, 3);
376         current_image_3d.erode(3, 3, 3);
377         current_image_3d.dilate(3, 3, 3);
378
379         visu_window_redraw = true;
380     }
381
382     if(visu_window.is_key('w'))
383     {
384         if (!translated)
385         {
386             std::cerr << "Veuillez d'abord calculer les trajectoire
(touche 't') avant de pouvoir écrire les trajectoire 3d !\n";
387             std::this_thread::sleep_for(0.1s);
388             continue;
389         }
390
391         std::cerr << "Ecriture de de la trajectoire 3d des cellules en
cours...\n";
392         write_3d_trajectory(std::cout, image_4d_centroids,
image_4d_translations);
393         std::cerr << "fait\n";
394         std::this_thread::sleep_for(0.2s);
395     }
396
397     if(visu_window.is_key('t'))
398     {
399         if (!centroided)

```

```

400     {
401         std::cerr << "Veuillez d'abord calculer les centroids
(touche 'c') avant de récupérer les translations !\n";
402         std::this_thread::sleep_for(0.1s);
403         continue;
404     }
405
406     std::cerr << "Calcul des translations de l'image 4d en
cours...\n";
407     for (size_t i = 0 ; i < image_4d_translations.size() ; i++)
408     {
409         image_4d_translations[i] =
get_translations(image_4d_centroids[i], image_4d_centroids[i + 1]);
410     }
411     std::cerr << "fait !\n";
412     std::cerr << "Appuyez sur 'w' pour ecrire la trajectoire dans
un flux\n";
413     std::this_thread::sleep_for(0.1s);
414     translated = true;
415 }
416
417 if(visu_window.is_key('c'))
418 {
419     if (!thresholded)
420     {
421         std::cerr << "Veuillez seuiller l'image (clicker sur une
cellule représentative de l'image) avant de récupérer les centroids !\n";
422         std::this_thread::sleep_for(0.1s);
423         continue;
424     }
425
426     // il faut que l'image soit seuiller avant de récupérer les
centroids
427     std::cerr << "Calcul des barycentres de l'image 4d en
cours...\n";
428     std::vector<vec3d> cells_centroid =
get_cells_centroid(current_image_3d);
429     size_t i = 0;
430     for (const auto& image_3d : image_4d)
431     {
432         image_4d_centroids[i] = get_cells_centroid(image_3d);
433         i++;
434     }
435     std::cerr << "fait !\n";
436     std::cerr << "Appuyez sur 't' pour calculer la trajectoire des
cellules\n";
437     centroided = true;
438 }
439
440     // Mouse left : clicker sur une cellule representative pour
seuiller l'image
441     if((visu_window.button() & 1) && (visu_window_plane != -1) &&
!thresholded)
442     {
443         // Seuillage de l'image
444
445         const int radius = 4;
446         vec3i vox_coord = {coord[0], coord[1], coord[2]};

```

```

447
448         double intensity = region_intensity(current_image_3d,
vox_coord, radius);
449
450         std::cerr << "Calcul de la moyenne des intensités de la
region de centre " << vox_coord << " et de rayon " << radius << "...\\n";
451
452         std::cerr << "Segmentation des cellules de l'image 4d en
cours...\\n";
453         for (auto& image_3d : image_4d)
454         {
455             std::cerr << "Seuillage sur l'intensité calculée (" <<
intensity << ")...\\n";
456             image_3d.threshold(intensity);
457             std::cerr << "Elimination du bruit...\\n";
458             image_3d.dilate(3, 3, 3);
459             image_3d.erode(3, 3, 3);
460             image_3d.erode(3, 3, 3);
461             image_3d.dilate(3, 3, 3);
462         }
463         std::cerr << "fait !\\n";
464         std::cerr << "Appuyez sur 'c' pour calculer les barycentres
des cellules\\n";
465
466         current_image_3d = image_4d[current_image_3d_index];
467
468         visu_window_redraw = true;
469         thresholded = true;
470     }
471
472     // Mouse right
473     if((visu_window.button() & 2) && (visu_window_plane != -1))
474     {
475         for(size_t i = 0; i < coord.size(); i++)
476             displayed_slice[i] = coord[i];
477
478         visu_window_redraw = true;
479     }
480
481     // Gère le défilement des projections de l'image 3D
482     if(visu_window.mouse_x() >= 0 && visu_window.mouse_y() >= 0)
483     {
484         int mX = visu_window.mouse_x()*
(image_3d_dimensions[0]+image_3d_dimensions[2])/visu_window.width();
485         int mY = visu_window.mouse_y()*
(image_3d_dimensions[1]+image_3d_dimensions[2])/visu_window.height();
486
487         if (mX >= image_3d_dimensions[0] && mY <
image_3d_dimensions[1])
488         {
489             visu_window_plane = 0;
490             coord[1] = mY;
491             coord[2] = mX - image_3d_dimensions[0];
492             coord[0] = displayed_slice[0];
493         }
494         else
495         {

```

```

496         if (mX < image_3d_dimensions[0] && mY >=
image_3d_dimensions[1])
497         {
498             visu_window_plane = 1;
499             coord[0] = mX;
500             coord[2] = mY - image_3d_dimensions[1];
501             coord[1] = displayed_slice[1];
502         }
503         else
504         {
505             if (mX < image_3d_dimensions[0] && mY <
image_3d_dimensions[1])
506             {
507                 visu_window_plane = 2;
508                 coord[0] = mX;
509                 coord[1] = mY;
510                 coord[2] = displayed_slice[2];
511             }
512             else
513             {
514                 visu_window_plane = -1;
515                 coord[0] = 0;
516                 coord[1] = 0;
517                 coord[2] = 0;
518             }
519         }
520     }
521     // visu_window_redraw = true;
522 }
523
524 if(visu_window.wheel())
525 {
526     displayed_slice[visu_window_plane] =
displayed_slice[visu_window_plane] + visu_window.wheel();
527
528     if(displayed_slice[visu_window_plane] < 0)
529     {
530         displayed_slice[visu_window_plane] = 0;
531     }
532     else
533     {
534         if(displayed_slice[visu_window_plane] >=
image_3d_dimensions[visu_window_plane])
535         {
536             displayed_slice[visu_window_plane] =
image_3d_dimensions[visu_window_plane] - 1;
537         }
538     }
539
540     visu_window.set_wheel();
541     visu_window_redraw = true;
542 }
543
544 if(visu_window_redraw)
545 {
546     visu_window.display(current_image_3d.get_projections2d(displayed_slice[0],
displayed_slice[1], displayed_slice[2]));

```

```
547         visu_window_redraw=false;
548     }
549 }
550
551     return 0;
552 }
553
```