Analyse de l'étape de propagation de segmentation
version git 5c6bf9c du 2018-05-04 (dernière version remontée par G. Michelin)

Contents

1 Overview

1.1 Notations

- $S_t^\star$: segmentation à t
- $S_{t+\delta t\leftarrow t}^\star = S_t^\star \circ \mathcal{T}_{t\leftarrow t+\delta t}$: segmentation à t projetée dans $I_{t+\delta t}$. C'est une nouvelle notation par rapport à [1], mais l'image est pas mal utilisée.
- $\tilde{S}_{t+1}$: segmentation de $I_{t+\delta t}$ par ligne de partage des eaux avec les graines $S_{t+1\leftarrow t}^e = S_t^e \circ \mathcal{T}_{t\leftarrow t+\delta t}$
- $\hat{S}_{t+1}$: segmentation de $I_{t+\delta t}$ par ligne de partage des eaux avec les graines Seeds_{t+1}. Cette image peut être amenée à changer
- S_t^e : cellules de $S_t^\star$ érodées (pour servir de graines)
- $S_{t+\delta t\leftarrow t}^e = S_t^e \circ \mathcal{T}_{t\leftarrow t+\delta t}$: cellules de $S_t^\star$ érodées (pour servir de graines) puis projetées dans $I_{t+\delta t}$.
- Seeds_{t+1} : image des graines calculées avec les paramètres optimaux. Cette image peut être amenée à changer
- $\mathcal{T}_{t\leftarrow t+\delta t}$: transformation non-linéaire permet tant de rééchantillonner I_t sur $I_{t+\delta t}$

1.2 Overview

L'appel se fait par le fichier `4-astec.py`. C'est là que se fait la boucle sur le temps pour segmenter successivement tous les points de temps.

La segmentation d'un point de temps $t+\delta t$ se fait par l'appel de la fonction `segmentation_propagation()` (section 3.16, page 36) avec en paramètres les images fusionnées, I_t et $I_{t+\delta t}$ ainsi que la segmentation à t, $S_t^\star$.

La plupart des paramètres (longueurs, volumes) sont en unité voxelique.
`segmentation_propagation()` (section 3.16, page 36) fait les opérations suivantes

- `non_linear_registration()` de `ASTEC/CommunFunctions/cpp_wrapping.py` qui calcule la transformation non-linéaire $\mathcal{T}_{t\leftarrow t+\delta t}$, à partir des images fusionnées
- `segmentation_propagation_seeds_init_and_deform()` (section 3.14, page 32), qui calcule S_t^e et $S_{t+\delta t\leftarrow t}^e$

  $$S_{t+\delta t\leftarrow t}^e = S_t^e \circ \mathcal{T}_{t\leftarrow t+\delta t}$$

  soit les graines projetées (cellules de $S_t^\star$ érodées puis transformées dans $I_{t+\delta t}$) [1, section 2.3.3.4]
- `apply_trsf()` de `ASTEC/CommunFunctions/cpp_wrapping.py` qui calcule $S_{t+\delta t\leftarrow t}^\star = S_t^\star \circ \mathcal{T}_{t\leftarrow t+\delta t}$, soit la segmentation à t projetée à $t+\delta t$
- Une autre image d'intensité $I_{t+\delta t}$ peut être calculée avec le rehaussement de membrane
- `segmentation_propagation_from_seeds()`, (section 3.15, page 32) avec en paramètres l'image de segmentation à t, $S_t^\star$, l'image d'intensité à $t+\delta t$ sur un octet, $I_{t+\delta t}$, $S_{t+1\leftarrow t}^e = S_t^e \circ \mathcal{T}_{t\leftarrow t+\delta t}$, soit les graines projetées (cellules de $S_t^\star$ érodées puis transformées dans $I_{t+\delta t}$). La transformation pourrait se faire en dehors de la fonction.

  - Calcul de $\tilde{S}_{t+1}$ (une première segmentation de $I_{t+\delta t}$) par ligne de partage des eaux avec les graines $S_{t+1\leftarrow t}^e = S_t^e \circ \mathcal{T}_{t\leftarrow t+\delta t}$
  - `get_seeds()` (section 3.8, page 13). Pour chaque cellule, et pour un ensemble de valeurs de $h \in [h_{min}, h_{max}]$ (avec un pas de 2, paramètre en dur), `get_seeds()` calcule (avec `cell_propagation()`) le nombre de h-minima qui sont strictement inclus dans la cellule c de $\tilde{S}_{t+1}$ ($\text{Count}^h(c)$ de [1, section 2.3.3.5, page 71]). Cette opération de sélection des h-minima strictement inclus dans la cellule c de $\tilde{S}_{t+1}$ est faite plusieurs fois, on peut donc la mutualiser.

– `get_back_parameters()` (section 3.9, page 15), qui détermine pour chaque cellule, selon les nombres de h-minima inclus, le nombre optimal de graines à considérer et les valeurs de paramètres (h, σ), associés (on prend le plus grand h donnant ce nombre de graines). Il y a ici une différence significative entre la description dans la thèse et ce qui est mis en œuvre.

D'après la thèse [1, page 72], on utilise $N_2(c)$ le nombre de h donnant exactement 2 graines et $N_{2^+}(c)$, la plus grande valeur de h donnant 2 graines. De plus $N_2(c)$ est défini par $N_{2^+}(c) - N_{2^-}(c) + 1$, où $N_{2^-}(c)$ est la plus petite valeur de h donnant 2 graines.

D'après [1], h peut être vu comme une mesure de contraste (si la hauteur de la membrane entre 2 cellules adjacentes est plus petite que h, alors il n'y aura qu'une seule Une première hypothèse est que le premier h donnant plus de 2 graines est une mesure du "bruit" (considéré comme un contraste) du "fond" de la cellule, ce premier h est donc $N_{2^-}(c) - 1$. Il y a une vraie division si le contraste de la membrane entre les deux cellules après division est suffisamment élevé (ce contraste est estimé par $N_{2^+}(c)$) et si ce contraste est suffisamment différent du contraste du bruit, donc si la différence $N_{2^+}(c) - (N_{2^-}(c) - 1) = N_2(c)$ est suffisamment élevée. D'où le critère

$$s(c) = N_{2^+}(c).N_2(c) > \tau \tag{1}$$

On calcule le score $s(c) = N_{2^+}(c).N_2(c)$ que l'on compare à τ. Il y aura division si $s(c) = N_{2^+}(c).N_2(c) > \tau$.

* si plusieurs h donnent 2 graines, on prend le plus grand de ces h
* s'il y a toujours au moins 3 graines, on prend 3 graines (et le plus grand h donnant 3 graines) et la plus petite des 3 régions résultant de ces graines sera fusionnée avec celle avec laquelle elle partage la plus grande frontière. [En fait c'est faux, car dans `get_seeds_from_optimized_parameters()` qui sera ensuite appelé dans `segmentation_propagation_from_seeds()` on crée les graines avec `extract_seeds(..., accept_3_seeds=False)` : on ne numérotera que les 2 premières graines.]

L'implémentation est sensiblement différente

* On a $h_{min} = 4$, $h_{max} = 18$ et on parcourt les h avec un pas δh de 2. L'ensemble des h testés n'est $[h_{min}, h_{max}] \subset \mathbb{N}$ mais $\{4, 6, 8, 10, 12, 14, 16, 18\}$.
* On calcule NB_2 par `nb_2=np.sum(np.array(s)==2)`, c'est le nombre de h qui donnent 2 graines. Cette valeur n'est égale à $N_2(c) = N_{2^+}(c) - N_{2^-}(c) + 1$ que si $\delta h = 1$
* On calcule NB_3 par `nb_3=np.sum(np.array(s)>=2)`, c'est le nombre de h qui donnent 2 graines ou plus. Cette valeur n'a rien à voir avec $N_{2^+}(c)$. On a évidemment $NB_3 \geq NB_2$

La règle implémentée (cf Alg. 1) est

1. S'il existe des h donnant 1 ou 2 graines (la question de la division se pose donc)
   (a) si le score $s(c) = NB_2 \cdot NB_3 \geq \tau$, alors on garde 2 graines. Comme $\tau = 25$, cela signifie que s'il y a au moins 5 valeurs de h qui donnent 2 graines, alors on divisera la cellule.
   (b) sinon ($s(c) = NB_2 \cdot NB_3 < \tau$) et il existe des h donnant 1 graine, alors on garde une graine
   (c) sinon ($s(c) = NB_2 \cdot NB_3 < \tau$ et il n'existe pas de h donnant 1 graine) on garde 2 graines [ce cas doit difficilement survenir, il faut que beaucoup de h ne donnent aucune graine, puis que les suivants donnent au moins 2 graines].
2. sinon (il n'existe pas de h donnant 1 ou 2 graines) et il existe des h donnant 3 graines, alors on garde 3 graines [en fait seules les 2 premières seront effectivement gardées].
3. sinon (il n'existe pas de h donnant 1 ou 2 ou 3 graines), on dit qu'il n'y a pas de graines

On récupère le premier h donnant le nombre choisi de graines. Comme les h sont parcourus par ordre décroissant, c'est donc le plus grand h donnant ce nombre de graines qui est retenu.

Quelques remarques:

* Le score dépend du nombre de h utilisés pour le calcul des graines, donc de h_{min}, de h_{max} et du pas entre 2 h successifs.

```
if N_1(c) > 0 or N_2(c) > 0 then
    // Il y a des h donnant 1 ou 2 graines
    if NB_2 · NB_3 ≥ τ then
        // Rq:  [1, page 72] utilise une inégalité stricte
        // τ est fixé à 25
        #seeds_opt = 2
    else if N_1(c) > 0 then
        #seeds_opt = 1
    else
        #seeds_opt = 2
    end
else if N_3(c) > 0 then
    #seeds_opt = 3
else
    // cas où N_1(c) = N_2(c) = N_3(c) = 0
    #seeds_opt = 0
end
```

Algorithm 1: Choix du nombre de graines

- `get_seeds_from_optimized_parameters()` (section 3.10, page 17), qui va construire une image de graines Seeds_{t+1} à partir des nombres de graines calculés par `get_back_parameters()`, et l'image de segmentation correspondante $\hat{S}_{t+1}$
  - Pour les cellules qui ont des graines ($\#seeds \neq 0$, cf Alg. 1), on récupère les h-minima (pour les paramètres optimaux trouvés) inclus dans la cellule (on les recalcule donc avec `extract_seeds()` (section 3.5, page 11), qui numérote aussi les composantes).
    Rq: si $\#seeds = 3$ on ne garde que les 2 premières composantes numérotées, ce qui me semble bizarre comme choix.
  - Pour le fond, on récupère (avec `cell_propagation()`) les h_{max}-minima strictement inclus dans le fond. Plutôt que de garder plusieurs numéros/labels pour ces graines, on pourrait toutes les mettre à 1, cela supprimerait des opérations (eg renumérotation de la segmentation) car ces graines ne seront pas remises en cause plus tard.
  - Pour les cellules sans graines ($\#seeds = 0$, cf Alg. 1, cas où $N_1(c) = N_2(c) = N_3(c) = 0$), on regarde leur volume dans $\tilde{S}_{t+1}$. S'il est assez grand (supérieur à 100 voxels), on récupère la projection de la cellule mère érodée comme graine ($S^e_{t+\delta t \leftarrow t} = S^e_t \circ \mathcal{T}_{t \leftarrow t+\delta t}$)

  Un appel à l'algorithme de ligne de partage des eaux permet de calculer l'image de segmentation $\hat{S}_{t+1}$ à partir des graines Seeds_{t+1}. Cet appel pourrait être sorti de la fonction.
- `volume_checking()` (section 3.12, page 22). C'est une fonction assez complexe. Elle vise à corriger les erreurs faites à l'étape précédente. C'est plus ou moins bien décrit dans [1, section 2.3.3.6, page 73]. Une des préoccupations est que les tests (informatiques) sont mal faits, ce qui peut (en théorie) faire que des cas échappent aux traitements (ce qui est peut être voulu), ou que des cas soient traités deux fois (ce qui serait bizarre).
  - On classe les cellules (ou les couples (mère, fille(s))) selon les variations de volume entre la mère (calculé dans $S^\star_t$ ou issu du linéage), et le volume des filles (calculé dans $\hat{S}_{t+1}$). On calcule
    $$volume_{ratio} = 1 - \frac{vol_{mother}}{\sum_{fille} vol_{fille}}$$
    - `bigger` contient les couples (mère, fille(s)) avec une augmentation de volume de plus de 10% ($vol_{mother} < \sum_{fille} vol_{fille}$)

- `lower` contient les couples (mère, fille(s)) avec une diminution de volume de plus de 10% ($vol_{mother} > \sum_{fille} vol_{fille}$)
- `to_look_at` contient les couples (mère, fille(s)) avec une diminution de volume de plus de 50%, c'est donc un sous-ensemble de `lower` ($vol_{mother} >> \sum_{fille} vol_{fille}$)
- `too_little` contient les couples (mère, fille) où la fille a un volume de moins de 1000 voxels

* Traitement de `to_look_at` (cf Alg. 2, [1, point (1), section 2.3.3.6, page 73]).
  - On recalcule le nombre de graines optimales $\#seeds_{opt}$, comme dans l'Alg. 1, pour les cas $N_1(c) > 0$ *or* $N_2(c) > 0$, sauf que l'on garde le plus petit h donnant le nombre de graines $\#seeds_{opt}$ (c'était le plus grand dans `get_back_parameters()`).
  - On peut maintenant avoir trois graines (liste `to_fuse_3` qui sera traitée plus tard).
  - la gestion des correspondances (filiation) est mal faite : des graines sont re-numérotées, mais les filiations précédentes ne sont pas effacées.

```
1  if N_1(c) > 0 or N_2(c) > 0 then
2      if N_{2+}(c).N_2(c) ≥ τ then
3          #seeds_opt = 2
4      else if N_1(c) > 0 then
5          #seeds_opt = 1
6      else
7          #seeds_opt = 2
8      end
9      if #seeds_opt = 1 and N_2(c) > 0 then
           // si on avait 1 graine optimale, mais des h donnant 2 graines
           // on (re)calcule les graines (#seeds) pour le premier h qui donne 2 graines
           // et on ne garde que les graines incluses dans la cellule
10         if #seeds = 2 and ... then
               // la seconde partie de la condition me semble bizarre, et serait fausse
                  tout le temps ...
               // C'est censé détecter le cas où sur les 2 graines, l'une est en dehors de
                  la cellule dans Ŝ_{t+1}
11             on passe à 2 graines
12     if (#seeds_opt = 1 or #seeds_opt = 2) and ∃n ≥ 3|N_n(c) > 0 then
           // cette condition peut être vérifiée en même temps que la précédente
           // si on a par exemple #seeds_opt = 1, N_2(c) > 0 et N_3(c) > 0
           // une cellule peut être traitée 2 fois ...
13         on récupère les h_min-minima (il y en a donc au moins 3)
14     else if #seeds_opt = 1 then
           // On a donc ∀n ≥ 3|N_n(c) = 0
           // cette condition peut être vérifiée en même temps que la première
           // si on a par exemple #seeds_opt = 1 et N_2(c) > 0
15         on récupère la graine correspondante de S^e_{t+δt←t} // graine "projetée", censée être plus
              grande
16 else if N_3(c) > 0 then
       // C'est le cas où il n'y a pas de h donnant 1 ou 2 graines.  On avait auparavant
          pris les 2 premières graines.
17     on permet d'avoir 3 graines, et on en fusionnera 2 plus tard
```

Algorithm 2: Traitement de `to_look_at` dans `volume_checking()`.

* Traitement de `too_little`. On enlève la graine correspondant à la fille de volume trop petit dans Seeds_{t+1}. Si c'est la seule fille, la mère n'a donc plus de descendance.
* S'il y a eu des changements précédemment, on recalcule une image de segmentation $\hat{S}_{t+1}$ et les volumes des cellules de $\hat{S}_{t+1}$. Sinon, on pourrait sortir de la fonction, ce qui n'est pas fait.
* Recalcul de la liste `lower` (le seuil de 10% est en dur et non réglable par une variable). A priori, il faudrait recalculer aussi les autres listes.
* Traitement de `lower`.
  · On fait la différence entre la cellule mère de $S^{\star}_{t+\delta t \leftarrow t}$ et les cellules filles de $\hat{S}_{t+1}$, et on regarde si le label majoritaire dans cette différence est le fond. Si c'est le fond, on va mettre le couple (mère, fille(s)) dans une liste `exterior_correction`, sinon on ne fait rien.
  · Traitement de `exterior_correction` [1, début du point (2), section 2.3.3.6, page 73] avec les morphosnakes dans des sous-images (`perform_ac()`, section 3.11, page 21), c'est ce qui peut prendre du temps.
  Dans `perform_ac()`, on calcule une norme de gradient (vient de `cpp_wrapping.py`), puis $g(I) = \frac{1}{\sqrt{1+\alpha|\nabla G_\sigma * I|}}$ comme dans [2, Eq. (24)]. Toutefois, cette équation est faite pour avoir une image de potentiel avec de faibles valeurs pour les contours (ie les membranes). Elle n'est donc pas adaptée pour nos images (une inversion des valeurs de l'image pourrait être plus adéquate).
  · Réincorporation des résultats dans $\hat{S}_{t+1}$.
  Attention, s'il y avait plusieurs filles (une division), elles sont fusionnées et la filiation ne donne plus qu'une fille.
* Traitement de `to_fuse_3`. On élimine la plus petite des 3 cellules et on l'affecte à celle des 2 autres avec laquelle elle partage le plus de frontière.

– `outer_correction()` ([1, fin du point (2), section 2.3.3.6, page 73]) (section 3.13, page 31) On réalise une ouverture morphologique sur le masque de l'embryon (un ouvert est toujours inclus dans l'objet de départ) et on enlève, pour les cellules filles de `exterior_correction`, les points enlevés par l'ouverture.

2 4-astec.py

```
[...]
   115 ### Building paths from nomenclature.py and parameters file
   116
   117 path_fuse_exp = replaceFlags(path_fuse_exp, p)
   118 print "Fused data will be searched in directory %s"%replaceFlags(path_fuse_exp,
   119                                                                   p)
   120 assert os.path.isdir(path_fuse_exp), "Provided fuse directory '%s' not found"\
   121                                      %path_fuse_exp
   122 path_fuse_exp_files = replaceFlags(path_fuse_exp_files, p)
   123
   124 path_seg = replaceFlags(path_seg, p)
   125 path_seg_exp = replaceFlags(path_seg_exp, p)
   126 path_seg_exp_files = replaceFlags(path_seg_exp_files, p)
   127 path_seg_exp_lineage = replaceFlags(path_seg_exp_lineage, p)
   128 path_seg_exp_lineage_test = replaceFlags(path_seg_exp_lineage_test, p)
   129 path_seg_exp_reconstruct=replaceFlags(path_seg_exp_reconstruct,p)
   130 path_seg_exp_reconstruct_files=replaceFlags(path_seg_exp_reconstruct_files,p)
   131 if not p.astec_keep_reconstruct_files:
   132     path_seg_exp_reconstruct_files = None
   133 path_log_file = replaceFlags(path_seg_logfile, p)
[...]
   178 ######################################
   179 ### Segmentation Propagation Stuff ###
   180 ######################################
   181
   182 # ASTEC  segmentation propagation
   183
   184 # Read the lineage tree (in case it was previously created)
   185 lin_tree_information=read_lineage_tree(path_seg_exp_lineage)
   186
   187 begin=p.begin+p.raw_delay
   188 end=p.end+p.raw_delay
   189
   190 ####SAFETY CHECK AFTER RELAUNCH
[...]
   228 ### PROCESS PROPAGATION SEGMENTATION
   229 for t in range(begin, end):
   230     time_segment=t+p.delta #Time point of Segmentation
   231     print 'Starting the segmentation at ' + str(time_segment)
   232     fused_file_ref=replaceTIME(path_fuse_exp_files, t) #Previous image file
   233     fused_file=replaceTIME(path_fuse_exp_files, time_segment) #To be segmented
   234     segmentation_file_ref=replaceTIME(path_seg_exp_files, t) #Prev. seg file
   235     segmentation_file=replaceTIME(path_seg_exp_files, time_segment) #Output seg
   236     reconstruct_file=None
   237
   238     if p.astec_keep_reconstruct_files:
   239         reconstruct_file=replaceTIME(path_seg_exp_reconstruct_files, \
   240                                      time_segment)
   241     #  TEMPORARY FOLDER
   242     temporary_folder=replaceTIME(os.path.join(path_seg_exp,'TEMP_'+FLAG_TIME),\
```

```
                                t)
    os.system("mkdir -p " + temporary_folder ) # Make temporary folder

    vf_file=replaceTimes( \
            os.path.join( \
            temporary_folder,'VF_t'+FLAG_TIMEREF+'_on_t'+FLAG_TIMEFLO+'.inr' \
            ), {FLAG_TIMEREF:t,FLAG_TIMEFLO:time_segment})
    h_min_files=replaceTIME(os.path.join(temporary_folder, \
                            'h_min_t$TIME_h$HMIN_s$SIGMA.inr'),time_segment)
    seed_file=replaceTIME(os.path.join(temporary_folder,'Seed_t$TIME.inr'),t)
    print vf_file
    print h_min_files
    print seed_file

    #PROCESS PROGATION SEGMENTATION
    seg_from_opt_h, lin_tree_information=segmentation_propagation( \
        t,fused_file_ref,segmentation_file_ref, fused_file,  \
        seed_file, vf_file, h_min_files,  \
        p.astec_h_min_min, p.astec_h_min_max, p.astec_sigma1,  \
        lin_tree_information, p.delta, p.astec_nb_proc, \
        membrane_reconstruction_method=p.astec_membrane_reconstruction_method,\
        fusion_u8_method=p.astec_fusion_u8_method, \
        flag_hybridation=p.astec_flag_hybridation, \
        RadiusOpening=p.astec_RadiusOpening, Thau=p.astec_Thau, \
        MinVolume=p.astec_MinVolume,  \
        VolumeRatioBigger=p.astec_VolumeRatioBigger, \
        VolumeRatioSmaller=p.astec_VolumeRatioSmaller, \
        MorphosnakeIterations=p.astec_MorphosnakeIterations, \
        NIterations=p.astec_NIterations, DeltaVoxels=p.astec_DeltaVoxels, \
        rayon_dil=p.astec_rayon_dil, \
        sigma_membrane=p.astec_sigma_membrane, \
        manual=p.astec_manual, \
        manual_sigma=p.astec_manual_sigma, \
        hard_thresholding=p.astec_hard_thresholding, \
        hard_threshold=p.astec_hard_threshold, \
        sensitivity=p.astec_sensitivity, \
        sigma_TV=p.astec_sigma_TV, \
        sigma_LF=p.astec_sigma_LF, \
        sample=p.astec_sample, \
        keep_membrane=False, keep_all=False,  nb_proc_ACE=p.astec_nb_proc_ace,\
        min_percentile=p.astec_min_percentile, \
        max_percentile=p.astec_max_percentile, \
        min_method=p.astec_min_method, max_method=p.astec_max_method,\
        sigma_hybridation=p.astec_sigma_hybridation, \
        path_u8_images=reconstruct_file, \
        verbose=True)
```

Appel à `segmentation_propagation()`, cf section 3.16, page 36

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `fused_file_ref` : nom de l'image d'intensite (fusionnée) à t, I_t
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S_t^\star$

- `fused_file` : nom de l'image d'intensite (fusionnée) à $t+1$, I_{t+1}
- `seed_file` : nom de l'image des graines
- `vf_file` : nom de la transformation non-linéaire $\mathcal{T}_{t \leftarrow t+1}$
- `h_min_files` : nom générique des images de h_{min} (paramétré par `TIME`, `HMIN`, et `SIGMA`)

```
290     #SAVE OUTPUT
291     print 'Write the segmentation in ' + segmentation_file
292     imsave(segmentation_file, seg_from_opt_h)
293     #Save the current lineage tree
294     write_lineage_tree(path_seg_exp_lineage,lin_tree_information)
295     os.system("rm -rf  " + temporary_folder ) #delete temporary folder
296
297 print 'ASTEC SEGMENTATION DONE'
298
299 ### PROCESS LINEAGE TREE FILE VERIFICATION
300 print 'PROCESS LINEAGE TREE VERIFICATION'
301 image_dict_seg=imageDict(path_seg_exp_files.replace(FLAG_TIME,"*"))
302 report=pkl_lineage_test(lin_tree_information, image_dict_seg, \
303                         file_out=path_seg_exp_lineage_test)
304 print report
305 print 'LINEAGE TREE FILE VERIFICATION DONE'
```

3 ASTEC/ASTEC.py

3.1 compute_volumes()

```
14 def compute_volumes(im, labels = None, real = True):
15     """
16     Return a dictionary, { label: volume }
17     im : SpatialImage (label image)
18     labels : list of labels for which to compute the volumes (if None, all volumes are computed)
19     """
20     labels = np.unique(im)
21
22     volume = nd.sum(np.ones_like(im), im, index=np.int16(labels))
23     return dict(zip(labels, volume))
```

3.2 create_seed()

```
26 def create_seed(parameters):
27     """
28     Erodes the label i in the binary image tmp
29     tmp : binary SpatialImage
30     max_size_cell : size max allow for a cell (here put at np.inf)
31     size_cell : size of the cell to erode
32     iterations : maximum number of iterations for normal cells
33     out_iterations : maximum number of iterations for exterior
34     bb : bounding box if tmp in the global image (necessary when computing in parallel)
35     i : label of the cell to erode
36     """
37     tmp, max_size_cell, size_cell, iterations, out_iterations, bb, i=parameters
```

```
38      nb_iter=iterations
39      if i==1:
40          nb_iter=out_iterations
41          opened=nd.binary_erosion(tmp, iterations=nb_iter)
42          while len(nd.find_objects(opened))!=1 and nb_iter>=0:
43              nb_iter-=1
44              opened=nd.binary_erosion(tmp, iterations=nb_iter)
45      else:
46          opened=nd.binary_erosion(tmp, iterations=nb_iter)
47          while len(nd.find_objects(opened))!=1 and nb_iter>=0:
48              nb_iter-=1
49              opened=nd.binary_erosion(tmp, iterations=nb_iter)
50      if max_size_cell<size_cell:
51          num=1
52      else:
53          num=i
54      return opened, num, bb
```

3.3 create_seeds()

Retourne une `SpatialImage` où les cellules de taille supérieure à `min_size_cell` (1000) sont érodées d'au plus 10 itérations pour les cellules et 25 pour le fond.

```
57  def create_seeds(seg, max_size_cell=np.inf, min_size_cell=1000, iterations=10, out_iterations=25
58      """
59      Erodes all the labels in the segmented image seg
60      seg : Segmentation to erode (SpatialImage)
61      max_size_cell : size maximum of a cell in number of voxels
62      min_size_cell : size minimum of a cell in number of voxels
63      iterations : maximum number of iterations for normal cells
64      out_iterations : maximum number of iterations for exterior
65      nb_proc : number maximum of processors allowed to be used
66      """
67      from multiprocessing import Process, Queue, Pool
68      bboxes=nd.find_objects(seg)
69      seeds=np.zeros_like(seg)
70      a=np.unique(seg)
71      pool=Pool(processes=nb_proc)
72      count=0
73      mapping=[]
74      for i in a:
75          tmp=seg[bboxes[i-1]]==i
76          size_cell=np.sum(tmp)
77          if size_cell>min_size_cell:
78              count+=1
79              mapping.append((tmp, \
80                              max_size_cell, size_cell, \
81                              iterations, out_iterations, \
82                              bboxes[i-1], i))
83      outputs=pool.map(create_seed, mapping)
84      pool.close()
85      pool.terminate()
```

```
86      for seed, num, bb in outputs:
87          seeds[bb][seed]=num
88      return SpatialImage(seeds, voxelsize=seg.voxelsize)
```

3.4 cell_propagation()

Appelé par `get_seeds()` (section 3.8, page 13) et `get_seeds_from_optimized_parameters()` (section 3.10, page 17)

Pour une sous-image contenant une cellule et une sous-image correspondant contenant des minima étiquetés, sélectionne les minima strictement inclus dans la cellule, et renvoie leur nombre total. Cette procédure est similaire à `extract_seeds()` (section 3.5, page 11), toutefois `extract_seeds()` renvoie un nombre de graines d'au plus 3, et en sélectionne au plus 2.

```
92 def cell_propagation(parameters):
93      """
94      Return the seeds in seeds_not_prop stricly included in cell c in seg_c
95      seg_c : segmented image (SpatialImage)
96      c : label of the cell to process
97      seeds_not_prop : image of seeds (SpatialImage)
98      """
99      seg_c, c, seeds_not_prop=parameters
```

- `seg_c` : une sous-image où la cellule est à c et le reste à 1.
- `c` : le label de la cellule
- `seeds_not_prop` : la sous-image des h-minima étiquetés

```
100     if len(np.unique(seg_c))!=2: # DON'T MESS WITH THE SEEDS ! YOU NEED ONE AND ONLY ONE !!
101         return
102     seg_out=None
103     labels=list(np.unique(seeds_not_prop[seg_c==c]))
```

on récupère les labels qui sont dans la cellule

```
104     #if 0 in labels:
105     labels.remove(0)#TODO Check if 0 is inside labels list
106     final_labels=[]
107     for l in labels:
108         if (seg_c[seeds_not_prop==l]==c).all():
```

on vérifie que le h-minimum du label est entièrement dans la cellule

```
109             final_labels.append(l)
110     nb=len(final_labels)
111     return seg_out, nb, final_labels, c
```

Retourne `None`, le nombre de h-minima, la liste des labels des h-minima, le label de la cellule

3.5 extract_seeds()

Appelé par `get_seeds_from_optimized_parameters()` (section 3.10, page 17) et `volume_checking()` (section 3.12, page 22). Cette procédure est similaire à `cell_propagation()` (section 3.4, page 11). Cependant `extract_seeds()` renvoie le nombre total de graines dans la cellule d'au plus 3, en en sélectionnant au plus 2, alors que `cell_propagation()` les dénombre toutes.

```
113 def extract_seeds(seg_c, c, path_seeds_not_prop=None, bb=None, accept_3_seeds=False):
```

Appelé par `get_seeds_from_optimized_parameters` (section 3.10, page 17). Il y a des similarités avec `cell_propagation()`

- `seg_c` : sous-image où la cellule est à c et le reste à 1
- `c` : label de la cellule
- `path_seeds_not_prop` : sous-image des h-minima, s'appelait `seeds_ex` lors de l'appel. Les h-minima ontété étiquetés par `find_local_minima()`

```
114     """
115     Return the seeds from seeds_not_prop stricly included in cell c from seg_c (the labels of th
116     seg_c : segmented image (SpatialImage)
117     c : label of the cell to process
118     seeds_not_prop : image of seeds (can be the path to the image or the SpatialImage)
119     bb : if seeds_not_prop is a path then bb is the bounding box of c in seeds_not_prop
120     accept_3_seeds : True if 3 seeds can be accepted as a possible choice
121     """
122     if type(path_seeds_not_prop)!=SpatialImage:
123         seeds_not_prop_out=imread(path_seeds_not_prop)
124         seeds_not_prop=seeds_not_prop_out[bb]
125     else: ## Then path_seeds_not_prop is the actual image we want to work with
126         from copy import deepcopy
127         seeds_not_prop=deepcopy(path_seeds_not_prop)
128     labels=list(np.unique(seeds_not_prop[seg_c==c]))
```

Récupère les labels des graines qui intersectent la cellule

```
129     labels.remove(0)
130     final_labels=[]
131     for l in labels:
132         if (seg_c[seeds_not_prop==l]==c).all():
133             final_labels.append(l)
```

Test si le label est entièrement inclus dans la cellule

```
134     if len(final_labels)==1:
135         return (1, (seeds_not_prop==final_labels[0]).astype(np.uint8))
136     elif len(final_labels)==2:
137         return (2, ((seeds_not_prop==final_labels[0]) +
138                     2*(seeds_not_prop==final_labels[1])).astype(np.uint8))
139     elif len(final_labels)==3 and not accept_3_seeds:# "too much seeds in the second extraction"
140         return (3, ((seeds_not_prop==final_labels[0]) +
141                     2*(seeds_not_prop==final_labels[1])).astype(np.uint8))
```

`accept_3_seeds` est à `False` par défaut, et ne peut pas être modifié lors de l'appel d'`extract_seeds()` par `get_seeds_from_optimized_parameters`, donc en cas de 3 graines, on ne numérote que les 2 premières, mais on renvoie un nombre de 3 graines quand même.

Les lignes ci-après ne sont utiliées que par `volume_checking()`. C'est le cas où il y a eu une grosse diminution de volume (plus de 50%) entre la mère et les filles, et qu'il n'existe pas de h donnant 1 ou 2 graines.

```
142     elif len(final_labels)==3 and accept_3_seeds: #"accept 3 seeds !"
143         return (3, ((seeds_not_prop==final_labels[0]) +
144                     2*(seeds_not_prop==final_labels[1]) +
145                     3*(seeds_not_prop==final_labels[2])).astype(np.uint8))
```

Retourne le nombre de labels, ainsi qu'une sous-image avec les graines étiquettées à partir de 1.

3.6 slices_dilation()

```
147 def __slices_dilation(slices, maximum=[np.inf, np.inf, np.inf]):
148     return tuple([slice(max(0, s.start-1), min(s.stop+1, maximum[i])) for i, s in enumerate(slic
149
150 def slices_dilation(slices, maximum=[np.inf, np.inf, np.inf], iterations=1):
151     for i in range(iterations):
152         slices=__slices_dilation(slices, maximum)
153     return slices
```

3.7 to_u8()

```
156 def to_u8(im, lt=0):
157     """
158      Return a SpatialImage in unsigned int
159     im : SpatialImage
160     lt : if the smallest value in the intensity image can be "predicted"
161     """
162     from copy import deepcopy
163     imcp=deepcopy(im)
164     tmp=imcp[:,:,imcp.shape[2]/3]
165     fper=np.percentile(tmp[tmp>=lt], 1)
166     nper=np.percentile(tmp[tmp>=lt], 99)
167     imcp[imcp<fper]=fper
168     imcp[imcp>nper]=nper
169     #im-=fper
170     np.subtract(imcp, fper, out=imcp, casting='unsafe')
171     return SpatialImage(np.uint8(np.linspace(0, 255, nper-fper+1), casting='unsafe')[imcp], voxe
```

3.8 get_seeds()

Appelé par `segmentation_propagation_from_seeds()`, section 3.15

Compte les h-minima strictement inclus dans la cellule pour un ensemble de valeurs de h

```
174 def get_seeds(seg, h_min_min,h_min_max, sigma, cells, fused_file, path_h_min, bounding_boxes, nb
```

- `seg` : image de segmentation $\tilde{S}_{t+1}$. S'appelait `segmentation` lors de l'appel
- `h_min_min` : plus petite valeur de h pour le calcul des h-minima
- `h_min_max` : plus grande valeur de h pour le calcul des h-minima
- `sigma` : σ pour le lissage gaussien avant le calcul des h-minima
- `cells` : liste des cellules
- `fused_file` : image originale I_{t+1}, les graines sont donc calculées sur l'image originale en 2 octets, et non sur une image transformée (eg par GLACE)
- `path_h_min` : nom générique pour les images de h-minima

- `bounding_boxes` : boites englobantes des cellules

```
175     """
176     Return the number of seeds found for each cell in seg for different h_min values (from h_min
177     seg : Segmented image (SpatialImage)
178     h_min_max : starting maximum value of h_min
179     sigma : sigma of the gaussian smoothing (in voxels)
180     cells : cells contained in seg
181     fused_file : path (?) towards the fused image on which to perform the local minima detection
182     path_h_min : format of h minima file names
183     bounding_boxes : bounding boxes of the cells in seg (to fasten the computation)
184     verbose : verbose mode (False or True)
185     """
186     from multiprocessing import Pool
187     nb_cells={}
188     treated=[]
189     parameters={}
190     mask=None
191     temp_path_h_min=path_h_min.replace('$HMIN',str(h_min_max))
192     if not os.path.exists(temp_path_h_min):
193         seeds_not_prop, mask=find_local_minima(temp_path_h_min, fused_file, h_min_max, sigma=sig
194     else:
195         seeds_not_prop=imread(temp_path_h_min)
```

`find_local_minima()` fait successivement un lissage gaussien, un calcul des h-minima (renvoie une image de "différence"), puis un seuillage par hysteresis avec un seuil bas de 1 et un seuil haut à h (les composantes sont étiquetées). Renvoie `seeds_not_prop`, `SpatialImage` résultat du seuillage par hystérésis, et `mask`, image de "différence" résultat du calcul des h-minima. Du fait de la relation d'ordre pour les h-minima, on peut calculer les prochains (avec h plus petit) dans cette image. `find_local_minima()` est dans `ASTEC/CommunFunctions/cpp_wrapping.py`

```
196
197     h_min=h_min_max
198     tmp_nb=[]
199     checking=True
200     while (checking):
201         mapping=[]
202         tmp_nb=[]
203         for c in cells:
204             if not c in treated:
205                 bb=slices_dilation(bounding_boxes[c], maximum=seg.shape, iterations=2)
206                 seg_c=np.ones_like(seg[bb])
207                 seg_c[seg[bb]==c]=c
208                 mapping.append((seg_c, c, seeds_not_prop[bb]))
```

Pour chaque cellule c, on dilate sa boite englobante, puis on construit une sous-image où la cellule est à c et le reste à 1.

```
209
210         pool=Pool(processes=nb_proc)
211         outputs=pool.map(cell_propagation, mapping)
```

on passe à `cell_propagation()` (section 3.4, page 11)

- `seg_c` : une sous-image où la cellule est à c et le reste à 1.
- `c` : le label de la cellule
- `seeds_not_prop[bb]` : la sous-image des h-minima étiquetés

```
212         pool.close()
213         pool.terminate()
```

`cell_propagation()` retourne

- `seg_c` : `None`,
- `nb` : le nombre de h-minima strictement inclus dans la cellule
- `labels` : liste des labels de ces h-minima
- `c` : le label de la cellule

```
214         for seg_c_p, nb, labels, c in outputs:
215             tmp_nb.append(nb)
216             nb_cells.setdefault(c, []).append(nb)
217             parameters.setdefault(c, []).append([h_min, sigma])
218 
219         h_min-=2
220         checking=h_min>=h_min_min and (((np.array(tmp_nb)<=2) & (np.array(tmp_nb)!=0)).any() or 
221         if checking :
222             temp_path_h_min=path_h_min.replace('$HMIN',str(h_min))
223             if not os.path.exists(temp_path_h_min):
224                 seeds_not_prop, mask=find_local_minima(temp_path_h_min,fused_file, h_min, mask=ma
225             else:
226                 seeds_not_prop=imread(temp_path_h_min)
227             if seeds_not_prop is None:
228                 checking=False
229     return nb_cells, parameters
```

Retourne

- `nb_cells` : liste pour chaque cellule, du nombre de h-minima strictement inclus dans la cellule de $\tilde{S}_{t+1}$
- `parameters` : liste pour chaque cellule, des paramètres de calcul des h-minima (h et σ)

3.9 `get_back_parameters()`

Appelé par `segmentation_propagation_from_seeds()`, section 3.15

```
232 def get_back_parameters(nb_cells, parameters, lin_tree, cells,Thau=25):
```

- `nb_cells` : liste pour chaque cellule, du nombre de h-minima strictement inclus dans la cellule de $\tilde{S}_{t+1}$. C'est le $\text{Count}^h(c)$ de [1, section 2.3.3.5, page 71].
- `parameters` : liste pour chaque cellule, des paramètres de calcul des h-minima (h et σ)
- `lin_tree` : fichier de linéage
- `cells` : liste des cellules
- `Thau` : τ pour le calcul du score $s(c) = N_{2^+}(c).N_2(c) > \tau$ [1, page 72].

```
233     """
234     Return the correct h-minima value for each cell
235     nb_cells : { cell: [#seeds, ] }: dict, key: cell, values: list of #seeds
236     parameters : { cell: [[h_min, sigma], ]}: dict matching nb_cells, key: cell, values: list of
237     """
238     lin_tree_back={ v:k for k, val in lin_tree.iteritems() for v in val }
239     right_parameters={}
240     cells_with_no_seed=[]
241     ## 2 plateau size vs noise ##
242     for c, s in nb_cells.iteritems():
243         nb_2=np.sum(np.array(s)==2)
244         nb_3=np.sum(np.array(s)>=2)
245         score=nb_2*nb_3
```

D'après [1, page 72], on utilise $N_2(c)$ le nombre de h donnant exactement 2 graines et $N_{2^+}(c)$, la plus grande valeur de h donnant 2 graines. De plus $N_2(c)$ est défini par $N_{2^+}(c) - N_{2^-}(c) + 1$, où $N_{2^-}(c)$ est la plus petite valeur de h donnant 2 graines. On calcule le score $s(c) = N_{2^+}(c).N_2(c)$ que l'on compare à τ. Il y aura division si $s(c) = N_{2^+}(c).N_2(c) > \tau$.

- si plusieurs h donnent 2 graines, on prend le plus grand de ces h
- s'il y a toujours au moins 3 graines, on prend 3 graines (et le plus grand h donnant 3 graines) et la plus petite des 3 régions résultant de ces graines sera fusionnée avec celle avec laquelle elle partage la plus grande frontière. [En fait c'est faux, car dans `get_seeds_from_optimized_parameters()` qui sera ensuite appelé dans `segmentation_propagation_from_seeds()` on crée les graines avec `extract_seeds(..., accept_3_seeds=False)` : on ne numérotera que les 2 premières graines.]

L'implémentation est sensiblement différente

- On a $h_{min} = 4$, $h_{max} = 18$ et on parcourt les h avec un pas δh de 2. L'ensemble des h testés n'est $[h_{min}, h_{max}] \subset \mathbb{N}$ mais $\{4, 6, 8, 10, 12, 14, 16, 18\}$.
- On calcule NB_2 par `nb_2=np.sum(np.array(s)==2)`, c'est le nombre de h qui donnent 2 graines. Cette valeur n'est égale à $N_2(c) = N_{2^+}(c) - N_{2^-}(c) + 1$ que si $\delta h = 1$
- On calcule NB_3 par `nb_3=np.sum(np.array(s)>=2)`, c'est le nombre de h qui donnent 2 graines ou plus. Cette valeur n'a rien à voir avec $N_{2^+}(c)$. On a évidemment $NB_3 \geq NB_2$

La règle implémentée est

1. S'il existe des h donnant 1 ou 2 graines (la question de la division se pose donc)
   (a) si le score $s(c) = NB_2 \cdot NB_3 \geq \tau$, alors on garde 2 graines. Comme $\tau = 25$, cela signifie que s'il y a au moins 5 valeurs de h qui donnent 2 graines, alors on divisera la cellule.
   (b) sinon ($s(c) = NB_2 \cdot NB_3 < \tau$) et il existe des h donnant 1 graine, alors on garde une graine
   (c) sinon ($s(c) = NB_2 \cdot NB_3 < \tau$ et il n'existe pas de h donnant 1 graine) on garde 2 graines [ce cas doit difficilement survenir, il faut que beaucoup de h ne donnent aucune graine, puis que les suivants donnent au moins 2 graines].
2. sinon (il n'existe pas de h donnant 1 ou 2 graines) et il existe des h donnant 3 graines, alors on garde 3 graines [en fait seules les 2 premières seront effectivement gardées].
3. sinon (il n'existe pas de h donnant 1 ou 2 ou 3 graines), on dit qu'il n'y a pas de graines

On récupère le premier h donnant le nombre choisi de graines. Comme les h sont parcourus par ordre décroissant, c'est donc le plus grand h donnant ce nombre de graines qui est retenu.

```
246         if (s.count(1) or s.count(2))!=0:
247             if score>=Thau:
248                 h, sigma=parameters[c][np.where(np.array(s)==2)[0][0]]
249                 nb_final=2
250             elif s.count(1)!=0:
251                 h, sigma=parameters[c][np.where(np.array(s)==1)[0][0]]
252                 nb_final=1
253             else:
254                 h, sigma=parameters[c][np.where(np.array(s)==2)[0][0]]
255                 nb_final=2
256             right_parameters[c]=[h, sigma, nb_final]
257         elif s.count(3)!=0:
258             h, sigma=parameters[c][s.index(3)]
259             right_parameters[c]=[h, sigma, 3]
260         else:
261             cells_with_no_seed.append(c)
262             right_parameters[c]=[0, 0, 0]
263     return right_parameters, cells_with_no_seed
264
```

Retourne un tableau avec, pour chaque cellule c, les valeurs de h, σ, et le nombre de graines, ainsi que la liste des cellules sans graines (c'est redondant, puisque ce sont les cellules avec 0 graines).

3.10 get_seeds_from_optimized_parameters()

Appelé par `segmentation_propagation_from_seeds()`, section 3.15, page 32

```
266 def get_seeds_from_optimized_parameters(t, seg, cells, cells_with_no_seed, right_parameters,delta
```

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `seg` : image de segmentation $\tilde{S}_{t+1}$, `SpatialImage`. S'appelait `segmentation` lors de l'appel
- `cells` : liste des cellules
- `cells_with_no_seed` : liste des cellules sans graines
- `right_parameters` : (H, σ, nombre de graines) pour chaque cellule
- `delta_t` :
- `bounding_boxes` : boites englobantes pour les cellules
- `im_ref` : I_{t+1} sur un octet, `SpatialImage`. S'appelait `im_fused_8` lors de l'appel
- `seeds` : $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$, soit les graines projetées (cellules de $S^\star_t$ érodées puis transformées dans I_{t+1}), `SpatialImage`
- `parameters` : liste pour chaque cellule, des paramètres de calcul des h-minima (h et σ)

```
267     """
268     Return the seed image from the locally parametrized h-minima operator
269     t : time
270     seg : propagated segmentation (seg at t deformed on t+dt)
271     cells : list of cells in seg
272     cells_with_no_seed : list of cells with no correct parameters
273     right_parameters : dict of the correct parameters for every cells
274     delta_t : dt
275     bounding_boxes : bounding boxes of the cells in seg (to fasten the computation)
```

```
276     im_ref : Intensity image at time t+dt (on which to permorm the watershed)
277     seeds : Propagated seeds from segmentation at time t (when no correct parameters were found)
278     parameters : ?
279     h_min_max : starting maximum value of h_min
280     sigma : sigma of the gaussian smoothing (in voxels)
281     path_h_min : format of h minima file names
282     """
283     seeds_from_opt_h=np.zeros_like(seg, dtype=np.uint16)
284     label_max=2
285     corres={}
286     divided_cells=[]
287     h_min_information={}
288     sigma_information={}
289     sigma_done=[]
290     h_min_done=[]
291     seeds_images={}
292     for c in cells:
293      print 'get_seeds_from_optimized_parameters on '+str(c)
294         if c in cells_with_no_seed:
295             continue
296         if not seeds_images.has_key((right_parameters[c][0], right_parameters[c][1])):
297             path_seeds_not_prop=path_h_min.replace('$HMIN',str(right_parameters[c][0])).replace(
298             seeds_images[(right_parameters[c][0], right_parameters[c][1])]=imread(path_seeds_not
299         bb=slices_dilation(bounding_boxes[c], maximum=seg.shape, iterations=2)
300         seg_c=np.ones_like(seg[bb])
301         seg_c[seg[bb]==c]=c
302         seeds_ex=seeds_images[(right_parameters[c][0], right_parameters[c][1])][bb]
```

Pour chaque cellule (qui a des graines), on dilate sa boite englobante, et on crée une sous-image, `seg_c`, où la cellule est à c et le reste à 1. On récupère aussi la sous-image correspondante, `seeds_ex`, des h-minima correspondant aux paramètres de la cellule. C'est tout-à-fait similaire à ce qui était fait dans `get_seeds()` (section 3.8, page 13).

```
303         nb, seeds_c=extract_seeds(seg_c, c, seeds_ex)
```

Appel à `extract_seeds()` (section 3.5, page 11)

- `seg_c` : sous-image où la cellule est à c et le reste à 1
- `c` : label de la cellule
- `seeds_ex` : sous-image des h-minima

`extract_seeds()` (re)calcule les graines strictement incluses dans la cellule c et renvoie une sous-image des graines numérotées à partir de 1

- `nb` : le nombre de graines
- `seeds_c` : sous-image des graines numérotées à partir de 1

Toutefois, le nombre de graines ne peut être que dans $[1, 2, 3]$ et il ne peut y avoir au plus que 2 graines numérotées. C'est pour ça que le cas `nb==3` est identique au cas `nb==2` ci-dessous.

```
304         if nb==1:
305             corres[c]=[label_max]
306             h_min_information[(t+delta_t)*10**4+label_max]=right_parameters[c][0]
307             sigma_information[(t+delta_t)*10**4+label_max]=right_parameters[c][1]
```

```
308             seeds_from_opt_h[bb]+=seeds_c*label_max
309             label_max+=1
310         elif nb==2:
311             corres[c]=[label_max, label_max+1]
312             divided_cells.append((label_max, label_max+1))
313             seeds_from_opt_h[bb][seeds_c==1]=label_max
314             h_min_information[(t+delta_t)*10**4+label_max]=right_parameters[c][0]
315             sigma_information[(t+delta_t)*10**4+label_max]=right_parameters[c][1]
316             label_max+=1
317             seeds_from_opt_h[bb][seeds_c==2]=label_max
318             h_min_information[(t+delta_t)*10**4+label_max]=right_parameters[c][0]
319             sigma_information[(t+delta_t)*10**4+label_max]=right_parameters[c][1]
320             label_max+=1
321         elif nb==3:
322             corres[c]=[label_max, label_max+1]
323             divided_cells.append((label_max, label_max+1))
324             seeds_from_opt_h[bb][seeds_c==1]=label_max
325             h_min_information[(t+delta_t)*10**4+label_max]=right_parameters[c][0]
326             sigma_information[(t+delta_t)*10**4+label_max]=right_parameters[c][1]
327             label_max+=1
328             seeds_from_opt_h[bb][seeds_c==2]=label_max
329             h_min_information[(t+delta_t)*10**4+label_max]=right_parameters[c][0]
330             sigma_information[(t+delta_t)*10**4+label_max]=right_parameters[c][1]
331             label_max+=1
```

A la fin de cette etape, `seeds_from_opt_h` contient les graines numerotees (la numérotation commence à 2, voir l'initialisation de `label_max`) des cellules qui ont des graines (donc pas celles de la liste `cells_with_no_seeds`).

```
332
333 print 'Create Background seed'
334     c=1
335     seg_c=np.ones_like(seg)
336     seg_c[seg!=c]=0
337     sigma_out=sigma
338     key_min = (h_min_max, sigma_out)
339     for k in seeds_images.iterkeys():
340      if k[0]<key_min[0]:
341      key_min = k
```

Pour la grain du fond, on récupère dans `seg_c` la "cellule" fond de $\tilde{S}_{t+1}$ (`seg_c` a donc des 1 pour le fond et des 0 ailleurs), segmentation par ligne de partage des eaux de I_{t+1} avec les graines $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$. On lui affecte des paramètres fictifs (h_{max}, σ).

```
342
343     print 'Cell propagation'
344     seeds_not_prop=seeds_images[key_min]
345     parameters=(seg_c, c,seeds_not_prop)
346     seg_c_p, nb, labels, c=cell_propagation(parameters)
```

On appelle `cell_propagation()` (section 3.4, page 11) avec les graines issues du calcul des h_{max}-minima. `cell_propagation()` retourne `None`, le nombre de h-minima, la liste des labels des h-minima, le label de la cellule

```
347     corres[1]=[]
348     exterior_corres=[]
349     for l in labels:
350         seeds_from_opt_h=seeds_from_opt_h.astype(np.uint16)
```

On caste `seeds_from_opt_h` en `uint16`, mais il avait été construit dans ce type : `seeds_from_opt_h=np.zeros_like(seg, dt`

```
351         exterior_corres.append(label_max)
352         seeds_from_opt_h[seeds_not_prop==l]=label_max
353         label_max+=1
```

On a récupéré ici toutes les graines correspondant au fond dans l'image des h_{max}-minima. Autre choix, on aurait pu toutes les mettre à 1.

```
354
355     print 'Cells with not Seed'
356     for c in cells_with_no_seed:
357         if np.sum(seg==c)>Volum_Min_No_Seed:
358             seeds_from_opt_h[seeds==c]=label_max
359             h_min_information[(t+delta_t)*10**4+label_max]=right_parameters[c][0]
360             corres[c]=[label_max]
361             label_max+=1
```

Pour les cellules "sans graines", on regarde si leur volume (dans $\tilde{S}_{t+1}$) est suffisamment grand (supérieur à 100). Si oui, on récupère alors la graine correspondante dans $S^e_{t+1 \leftarrow t} = S^e_t \circ \mathcal{T}_{t \leftarrow t+1}$ (cellules de $S^\star_t$ érodées puis transformées).

```
362
363     print 'Watershed '
364     seg_from_opt_h=watershed(SpatialImage(seeds_from_opt_h, voxelsize=seeds_from_opt_h.voxelsize
```

Ligne de partage des eaux dans I_{t+1}

```
365     for l in exterior_corres:
366         seg_from_opt_h[seg_from_opt_h==l]=1
367     corres[1]=[1]
```

On met les cellules correspondant au fond à 1. Cela n'aurait pas été nécessaire si toutes les graines du fond avaient été mises à 1.

```
368
369     return seeds_from_opt_h, seg_from_opt_h, corres, exterior_corres, h_min_information, sigma_i
```

Retourne

- `seeds_from_opt_h` : une `SpatialImage` contenant les graines
- `seg_from_opt_h` : une `SpatialImage` contenant la segmentation
- `corres` : un tableau contenant les filiations
- `exterior_corres` : une liste contenant la filiation pour le fond (inutile ?)
- `h_min_information` : une liste contenant les valeurs de h utilises pour chaque cellule
- `sigma_information` : une liste contenant les valeurs de σ utilises pour chaque cellule
- `divided_cells` : une liste contenant les couples de cellules soeurs
- `label_max` : le plus grand label utilis (+1)

3.11 perform_ac()

Appelé par volume_checking() (section 3.12, page 22).

```
372 def perform_ac(parameters):
373     """
374     Return the shape resulting of morphosnake operation on image I using image S as an initialis
375     m : label of the cell to work on
376     daughters : list of the daughters of cell m (to keep track when working in parallel)
377     bb : bounding boxe of m
378     I : intensity image to perform active contours on (SpatialImage)
379     S : segmented image to perform active contours from (SpatialImage, must contain the label m)
380     """
381
382
```

- m : label de la cellule mère
- daughters : labels des cellules filles
- bb : boite englobante de la cellule mère transformée et dilatée (15 fois)
- I : sous-image de l'image originale définie par la boite englobante (ce n'est pas nécessairement une image sur 2 octets). S'appelait im_ref_tmp
- S : sous-image de $S_t^\star$ transformée dans I_{t+1}, soit $S_t^\star \circ \mathcal{T}_{t \leftarrow t+1}$. S'appelait seg_ref_tmp
- MorphosnakeIterations : MorphosnakeIterations=10
- NIterations : NIterations=200
- DeltaVoxels : DeltaVoxels=10**3

```
383     m, daughters, bb, I, S,MorphosnakeIterations,NIterations,DeltaVoxels=parameters
384     import os
385     from scipy import ndimage as nd
386     import morphsnakes
387     cell_num=m
```

On érode le complémentaire de la mère ?

```
388     Sb=nd.binary_erosion(S!=cell_num, iterations=MorphosnakeIterations, border_value=1)#[:,:,sl]
389     image_input='tmp_'+str(cell_num)+'.inr'
390     gradient_output='tmp_out_'+str(cell_num)+'.inr'
391     imsave(image_input, I)
```

On calcule une norme de gradient (vient de cpp_wrapping.py), puis on calcule

$$g(I) = \frac{1}{\sqrt{1 + \alpha |\nabla G_\sigma * I|}}$$

c'est l'eq. (24) de [2]. Toutefois, le design de cette fonction $g()$ est d'être faible aux contours. Si on considère que l'image des membranes est une image de norme de gradient, il aurait suffi de l'inverser.

```
392     gradient_norm(image_input,gradient_output)
393     gI = imread(gradient_output)
394     os.system('rm -f '+image_input+' '+gradient_output)
395     gI=1./np.sqrt(1+100*gI)
396
```

```
397
398     macwe = morphsnakes.MorphGAC(gI, smoothing=3, threshold=1, balloon=1)
399     macwe.levelset = Sb
400     bef=np.ones_like(Sb)
401     from copy import deepcopy
402     for i in xrange(NIterations):
403         beff=deepcopy(bef)
404         bef=deepcopy(macwe.levelset)
405         macwe.step()
406         if np.sum(bef!=macwe.levelset)<DeltaVoxels or np.sum(beff!=macwe.levelset)<DeltaVoxels:
407             break
408     out=macwe.levelset
409     tmp=nd.binary_fill_holes(out)
410     cell_out=(out.astype(np.bool) ^ tmp)
411     return m, daughters, bb, cell_out
```

3.12 volume_checking()

Appelé par segmentation_propagation_from_seeds() (section 3.15, page 32)

```
414 def volume_checking(t,delta_t,seg, seeds_from_opt_h, seg_from_opt_h, corres, divided_cells, boun
415     label_max, exterior_corres, parameters, h_min_information, sigma_information, segmentation_f
416     nb_proc=26,Thau= 25,MinVolume=1000,VolumeRatioBigger=0.5,VolumeRatioSmaller=0.1,MorphosnakeI
```

- t : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- delta_t :
- seg : image de segmentation $\tilde{S}_{t+1}$, SpatialImage. S'appelait segmentation lors de l'appel. C'est la segmentation de I_{t+1} avec les graines projetées $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$,
- seeds_from_opt_h : une SpatialImage contenant les graines (obtenues avec les paramètres optimaux pour chaque cellule)
- seg_from_opt_h : une SpatialImage contenant la segmentation obtenue avec les graines de seeds_from_opt_h
- corres : un tableau contenant les filiations de chaque cellule de la segmentation à t
- divided_cells : une liste contenant les couples de cellules soeurs
- bounding_boxes : boites englobantes pour les cellules
- right_parameters : (h, σ, nombre de graines) optimaux pour chaque cellule
- im_ref : I_{t+1} sur un octet, SpatialImage. S'appelait im_fused_8 lors de l'appel
- im_ref16 : I_{t+1} sur un ou deux octet, SpatialImage. Peut être identique à im_ref. S'appelait im_fused lors de l'appel.
- seeds : $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$, soit les graines projetées (cellules de $S^\star_t$ érodées puis transformées dans I_{t+1}) SpatialImage
- nb_cells : liste pour chaque cellule, du nombre de h-minima strictement inclus dans la cellule de $\tilde{S}_{t+1}$. C'est le Count$^h(c)$ de [1, section 2.3.3.5, page 71].
- label_max : le plus grand label utilis pour les graines (+1)
- exterior_corres : une liste contenant la filiation (les labels des graines) pour le fond (inutile ?)
- parameters : liste pour chaque cellule, de tous les paramètres de calcul des h-minima (h et σ)
- h_min_information : une liste contenant les valeurs de h utilises pour chaque cellule
- sigma_information : une liste contenant les valeurs de σ utilises pour chaque cellule
- segmentation_file_ref : nom de l'image de segmentation à t, $S^\star_t$

- `segmentation_file_trsf` : nom de l'image de segmentation $S_t^\star$ transformée dans I_{t+1}, soit $S_t^\star \circ \mathcal{T}_{t \leftarrow t+1}$. S'appelait `path_seg_trsf` lors de l'appel.
- `path_h_min` : nom générique pour les images de h-minima
- `volumes_t_1` : liste des volumes des cellules à t
- ...

```
417     """
418     Return corrected final segmentation based on conservation of volume in time
419     seg : propagated segmentation (seg at t deformed on t+dt) (SpatialImage)
420     seeds_from_opt_h : optimized seeds (SpatialImage)
421     seg_from_opt_h : segmented image from seeds_from_opt_h (SpatialImage)
422     corres : mapping of cells at time t to cells at t+dt in seg_from_opt_h
423     divided_cells : list of cells that have divided between t and t+dt
424     bounding_boxes : bounding boxes of the cells in seg (to fasten the computation)
425     right_parameters : list of parameters used to create seeds_from_opt_h
426     im_ref : image to segment at time t+dt 8 bits (SpatialImage)
427     im_ref16 : image to segment at time t+dt in 16 bits (SpatialImage)
428     seeds : Propagated seeds from segmentation at time t
429     nb_cells : { cell: [#seeds, ] }: dict, key: cell, values: list of #seeds
430     label_max : maximum label in seg_from_opt_h
431     exterior_corres : list of cells that have been corrected for issue in exterior
432     parameters : { cell: [[h_min, sigma], ]}: dict matching nb_cells, key: cell, values: list of
433     h_min_information : { cell: h_min}: dict associating to each cells the h_min that allowed its
434     sigma_information : { cell: sigma}: dict associating to each cells the sigma that allowed its
435     segmentation_file_ref : path to the segmentation at time t
436     segmentation_file_trsf : path to the segmentation at time t resampled at t+1
437     vf_file : path to the vector field that register t into t+dt
438     path_h_min : format of h-minima files
439     volumes_t_1 : cell volumes at t (provient de la lecture du lin\'eage)
440     """
441
442     # seg_origin : original segmentation (SpatialImage)
443
444     seg_origin=imread(segmentation_file_ref)
445
446     volumes_from_opt_h=compute_volumes(seg_from_opt_h)
447     if volumes_t_1=={}:
448         volumes=compute_volumes(seg_origin)
449     else:
450         volumes=volumes_t_1
```

- `volumes_from_opt_h` : volumes des cellules de la segmentation `seg_from_opt_h` à $t+1$
- `volumes` : volumes des cellules de la segmentation `segmentation_file_ref` à t (a priori provient de la lecture du linéage lors de l'appel : `volumes_t_1`).

```
451
452     bigger=[]
453     lower=[]
454     to_look_at=[]
```

```
455     too_little=[]
456     for mother_c, sisters_c in corres.iteritems():
457         if mother_c!=1:
458             volume_ratio=1-(volumes[mother_c]/np.sum([volumes_from_opt_h.get(s, 1) for s in sist
459             if not (-VolumeRatioSmaller<volume_ratio<VolumeRatioSmaller):
460                 if (volume_ratio>0) and (volumes_from_opt_h.get(s, 1)!=1):
461                     bigger.append((mother_c, sisters_c))
462                 elif volumes_from_opt_h.get(s, 1)!=1 :
463                     lower.append((mother_c, sisters_c))
464                 if volume_ratio<-VolumeRatioBigger:
465                     to_look_at.append(mother_c)
466             else :
467                 for s in sisters_c:
468                     if volumes_from_opt_h[s]<MinVolume:
469                         too_little.append((mother_c, s))
```

Calcule $volume_{ratio} = 1 - vol(mother)/\sum vol(daughter)$

1. Si $vol(mother)/\sum vol(daughter) \geq 1 + VRS$ ou $1 - VRS > vol(mother)/\sum vol(daughter)$ (variation de volume de plus de 10%)
   (a) Si $\sum vol(daughter) > vol(mother)$, on met le couple (mère, liste des filles) dans `bigger`
   (b) sinon (si $\sum vol(daughter) \leq vol(mother)$), on met le couple (mère, liste des filles) dans `lower`
   (c) Si $vol(mother)/\sum vol(daughter) \geq 1 + VRB$ (variation de volume de plus de 50%), on ajoute la mère dans `to_look_at`
2. sinon, si une cellule fille s a un trop petit volume (< 1000), alors on ajoute le couple (mère, fille) dans `too_little`

Notes:

- Les mères dans `to_look_at` sont aussi dans `lower`
- Le cas $vol(mother)/\sum vol(daughter) \leq 1 + VRB$ n'est pas considéré

```
470
471     to_fuse_3=[]
472     change_happen=False
```

`to_look_at` contient la liste des cellules de $S_t^\star$ dont les filles ont une grande diminution de volume (plus de 50%). `nb_cells` contient le nombre de graines (de h-minima) pour les différentes valeurs de h. `nb_cells` a été calculé par `get_seeds()` (section 3.8, page 13).

```
473     for c in to_look_at:
474         s=nb_cells[c]
475         nb_2=np.sum(np.array(s)==2)
476         nb_3=np.sum(np.array(s)>=2)
477         score=nb_2*nb_3
478         if (s.count(1) or s.count(2))!=0:
479             if score>=Thau:
480                 h, sigma=parameters[c][np.where(np.array(s)==2)[0][-1]]
481                 nb_final=2
482             elif s.count(1)!=0:
483                 h, sigma=parameters[c][np.where(np.array(s)==1)[0][-1]]
```

```
484                 nb_final=1
485             else:
486                 h, sigma=parameters[c][np.where(np.array(s)==2)[0][-1]]
487                 nb_final=2
488             right_parameters[c]=[h, sigma, nb_final]
```

On recalcule le score $s(c) = N_{2^+}(c).N_2(c) > \tau$ (déjà calculé dans `get_back_parameters()` (section 3.9, page 15). Les premières lignes (479 à 488) sont identiques aux lignes (247 à 256) de `get_back_parameters()`, excepté que c'est le plus petit h qui donne ce nombre de graines qui est retenu (plutôt que le plus grand – l'indice [-1] au lieu de [0])

1. Si le score donne une graine (`nb_final` $= 1$) et il existe des h qui donnent 2 graines, on récupère le premier h qui donne 2 graines et on passe donc à 2 graines (mais il y a un test bizarre)
2. Si le score donne une ou deux graine(s) (`nb_final` $= 1$ `or nb_final` $= 2$) et il existe un h qui donne plus de 2 graines, on récupère les paramètres liés au plus petit h (`parameters[c][-1]`), donc forcément plus de 2 graines. Notons que le cas où `nb_final` $= 1$ et qu'il y a des h qui donnent 2 et plus graines est traité 2 fois.

```
489             if nb_final==1 and s.count(2)!=0:
490                 h, sigma=parameters[c][s.index(2)]
```

Premier cas : la sélection optimale donne 1 graine, et il existe des h qui donnent 2 graines, on récupère le premier h qui donne 2 graines, donc le plus grand d'entre eux.

```
491                 path_seeds_not_prop=path_h_min.replace('$HMIN',str(h)).replace('$SIGMA',str(sigma
492                 bb=slices_dilation(bounding_boxes[c], maximum=seg.shape, iterations=2)
493                 seg_c=np.ones_like(seg[bb])
494                 seg_c[seg[bb]==c]=c
495                 nb, seeds_c=extract_seeds(seg_c, c, path_seeds_not_prop, bb)
```

Comme dans `get_seeds_from_optimized_parameters()` (section 3.10, page 17), on crée une sous-image, `seg_c`, où la cellule est à c et le reste à 1. `extract_seeds()` renvoie le nombre de graines ainsi qu'une sous-image avec ces graines étiquetées.

```
496                 if nb==2 and (seg_from_opt_h[bb][seeds_c!=0]==0).any(): #If we can found 2 seeds
```

On vérifie que l'on a bien 2 graines (ce doit être le cas, ce sont les mêmes opérations que pour `get_seeds()`, et que `seg_from_opt_h[bb][seeds_c!=0]==0).any()` : on prend la sous-image `seg_from_opt_h[bb]`, on la masque par `[seeds_c!=0]`, on a donc un tableau avec la liste des labels de `seg_from_opt_h[bb]` "en-dessous" des graines de `seeds_c` et on regarde s'il y a des points à 0, ce qui est a priori nécessairement faux, puisque les labels vont de 1 (le fond) à `label_max`-1 ...

Sans doute il aurait fallu tester avec `seeds_from_opt_h`, auquel cas on aurait bien testé l'apparition d'une nouvelle graine (par rapport à une graine (avec un h grand) qui se divise en 2 graines (avec un h petit).

Si oui, on efface dans les graines `seeeds_from_opt_h` la graine précédente (`corres[c][0]`) et on y ajoute les 2 nouvelles graines. On met à jour l'information des paramètres dans `h_min_information` et `sigma_information`. Notons qu'il aurait fallu y enlever les informations liées à `corres[c][0]`.

ex: `del h_min_information[(t+delta_t)*10**4+corres[c][0]]`

```
497                     change_happen=True
498                     seeds_from_opt_h[seeds_from_opt_h==corres[c][0]]=0
499                     corres[c]=[label_max, label_max+1]
500                     divided_cells.append((label_max, label_max+1))
501                     seeds_from_opt_h[bb][seeds_c==1]=label_max
```

```
502                     h_min_information[(t+delta_t)*10**4+label_max]=h
503                     sigma_information[(t+delta_t)*10**4+label_max]=sigma
504                     label_max+=1
505                     seeds_from_opt_h[bb][seeds_c==2]=label_max
506                     h_min_information[(t+delta_t)*10**4+label_max]=h
507                     sigma_information[(t+delta_t)*10**4+label_max]=sigma
508                     label_max+=1
509             if (nb_final==1 or nb_final==2) and (np.array(s)>2).any():
510                 h, sigma=parameters[c][-1]
```

Second cas : la sélection optimale donne 1 graine ou 2 graines, et il existe des h qui donnent plus de 2 graines, on récupère le dernier h (le plus petit testé, donc h_{min}) qui donne forcément plus de graines.

Les cas où l'ensemble des h donne 1, 2 et plus graines est vérifié par les 2 conditions, et est donc traité 2 fois ?!

```
511                 path_seeds_not_prop=path_h_min.replace('$HMIN',str(h)).replace('$SIGMA',str(sigma
512                 seeds_image=imread(path_seeds_not_prop)
513                 bb=slices_dilation(bounding_boxes[c], maximum=seg.shape, iterations=2)
514                 seg_c=np.zeros_like(seg_from_opt_h[bb])
515                 for daughter in corres[c]:
516                     seg_c[seg_from_opt_h[bb]==daughter]=1
```

On crée une sous-image `seg_c` de 0 pour la cellule c dans laquelle on met à 1 les cellules filles de c (rappel, il y a une diminution de volume de plus de 50% entre la mère et les filles)

```
517                 seeds_c=np.zeros_like(seg_from_opt_h[bb])
518                 seeds_c[(seg_c==1) & (seeds_image[bb]!=0)]=1
519                 seeds_c[(seg[bb]==c) & (seg_c!=1) & (seeds_image[bb]!=0)]=2
```

On crée une sous-image `seeds_c` de 0 pour la cellule c dans laquelle on met à

- 1 : l'intersection des cellules filles de c (marquées dans `seg_c`) et des h_{min}-minima
- 2 : l'intersection de la cellule c de $\tilde{S}_{t+1}$ et des h_{min}-minima (qui ne sont pas dans les cellules filles). Ces points sont donc des graines en-dehors des cellules filles.

```
520                 if 2 in seeds_c:
```

Si il existe des graines en dehors des cellules filles (donc des graines permettant de "récupérer" plus de matière), on efface dans les graines `seeeds_from_opt_h` les graines des filles précédentes (`corres[c]`) et on y ajoute les 2 ensembles de nouvelles graines. On met à jour l'information des paramètres dans `h_min_information` et `sigma_information`. Notons qu'il aurait fallu y enlever les informations liées à `corres[c]`.

ex: `del h_min_information[(t+delta_t)*10**4+corres[c][0]]`

Par ailleurs, on a forcément plus de 3 graines : les graines à l'intérieur des cellules filles auront un label, et celles à l'extérieur un autre. La division cellulaire est réalisée ainsi (alors qu'il pouvait déjà y avoir une division cellulaire (cas `nb_final` $==2$) !)

```
521                     change_happen=True
522                     for daughter in corres[c]:
523                         seeds_from_opt_h[seeds_from_opt_h==daughter]=0
524                     corres[c]=[label_max, label_max+1]
525                     divided_cells.append((label_max, label_max+1))
526                     seeds_from_opt_h[bb][seeds_c==1]=label_max
527                     h_min_information[(t+delta_t)*10**4+label_max]=h
```

```
528                     sigma_information[(t+delta_t)*10**4+label_max]=sigma
529                     label_max+=1
530                     seeds_from_opt_h[bb][seeds_c==2]=label_max
531                     h_min_information[(t+delta_t)*10**4+label_max]=h
532                     sigma_information[(t+delta_t)*10**4+label_max]=sigma
533                     label_max+=1
534             elif nb_final==1:
```

On est donc dans le cas où `(np.array(s)>2).any()` est faux, donc il n'y a que une ou deux graines. S'il y a deux graines, cela a déjà été traité dans le premier test (ligne 489). On efface alors la graine de `seeds_from_opt_h` et on la remplace par la graine projetée de `seeds` ($S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$). Comme on ne change pas le nombre de labels, pourquoi incrémenter `label_max` de 1 ?

```
535                 change_happen=True
536                 seeds_from_opt_h[seeds_from_opt_h==corres[c][0]]=0
537                 seeds_from_opt_h[seeds==c]=corres[c][0]
538                 label_max+=1
539         elif s.count(3)!=0:
```

cas où il y a 0 ou trois graines (ou plus) dans tous les h-minima (s'il y a 0 ou 4 graines ou plus, cela échappe au test).

```
540             h, sigma=parameters[c][s.index(3)]
```

Premier cas : on récupère le premier 3 qui donne 2 graines, donc le plus grand d'entre eux. Si les premiers h ne donnent pas 0 graines, c'est donc h_{max}.

```
541             path_seeds_not_prop=path_h_min.replace('$HMIN',str(h)).replace('$SIGMA',str(sigma));
542             bb=slices_dilation(bounding_boxes[c], maximum=seg.shape, iterations=2)
543             seg_c=np.ones_like(seg[bb])
544             seg_c[seg[bb]==c]=c
```

Appel à `extract_seeds()` (section 3.5, page 11), avec `accept_3_seeds=True`. On va donc numéroter les 3 graines.

```
545             nb, seeds_c=extract_seeds(seg_c, c, path_seeds_not_prop, bb, accept_3_seeds=True)
546             change_happen=True
547             #addition to correct 0-boolean error when len(corres[c])>1
548             for ci in range(len(corres[c])):
549                 seeds_from_opt_h[seeds_from_opt_h==corres[c][ci]]=0
550             #seeds_from_opt_h[seeds_from_opt_h==corres[c]]=0
551             divided_cells.append((label_max, label_max+1))
552             seeds_from_opt_h[bb][seeds_c==1]=label_max
553             h_min_information[(t+delta_t)*10**4+label_max]=h
554             sigma_information[(t+delta_t)*10**4+label_max]=sigma
555             label_max+=1
556             seeds_from_opt_h[bb][seeds_c==2]=label_max
557             h_min_information[(t+delta_t)*10**4+label_max]=h
558             sigma_information[(t+delta_t)*10**4+label_max]=sigma
559             label_max+=1
560             seeds_from_opt_h[bb][seeds_c==3]=label_max
561             h_min_information[(t+delta_t)*10**4+label_max]=h
562             sigma_information[(t+delta_t)*10**4+label_max]=sigma
563             label_max+=1
564             to_fuse_3.append([c, (label_max-1, label_max-2, label_max-3)])
565
```

`to_fuse_3` contient donc les couples (mère, triplets de labels des filles) lorsqu'il y a 3 soeurs.

```
566
567
```

On traite les cas où une fille a un trop petit volume. `too_little` contient le couple `c=[mother_c, daughter_c]`. On enlève cette fille des graines de `seeds_from_opt_h`. Si c'était la seule fille, on enlève la mère de la table des correspondances.

```
568     if too_little!=[]:
569         for c in too_little:
570             #for d in corres[c]:
571             seeds_from_opt_h[seeds_from_opt_h==c[1]]=0
572             tmp=corres[c[0]]
573             tmp.remove(c[1])
574             if tmp==[]:
575                 corres.pop(c[0])
576             else:
577                 corres[c[0]]=tmp
578             change_happen=True
579
```

S'il y a eu des changements dans les graines, on recalcule une image de segmentation. Sinon on pourrait arreter là.

`bigger` n'a pas été utilisé, `lower` non plus, et va être recalculé.

```
580     if change_happen:
581         seg_from_opt_h=watershed(SpatialImage(seeds_from_opt_h,voxelsize=seeds_from_opt_h.voxels
582         for l in exterior_corres:
583             seg_from_opt_h[seg_from_opt_h==l]=1
584
585         volumes_from_opt_h=compute_volumes(seg_from_opt_h)
586
```

On recalcule `lower` (on rappelle que les cellules présentes dans `to_look_at` étaient aussi dans `lower`, il y a donc eu des changements dans la liste (s'il y a eu des changements dans les graines).

A priori, cela est inutile s'il n'y a pas eu de changements.

Note : il aurait fallu utiliser `VolumeRatioSmaller` et non `0.1`

```
587     lower=[]
588     for mother_c, sisters_c in corres.iteritems():
589         if mother_c!=1:
590             volume_ratio=1-(volumes[mother_c]/np.sum([volumes_from_opt_h.get(s, 1) for s in siste
591             if not (-.1<volume_ratio<.1):
592                 if (volume_ratio<0) and volumes_from_opt_h.get(s, 1)!=1 :
593                     lower.append((mother_c, sisters_c))
594
```

On regarde donc s'il y a des diminutions de volumes de plus de 10%.

```
595     exterior_correction=[]
596     if lower!=[]:
597         from copy import deepcopy
598         #tmp=apply_trsf(segmentation_file_ref, vf_file, nearest=True, lazy=False)
599         tmp=imread(segmentation_file_trsf)
```

segmentation_file_trsf : $S_t^\star$ transformée dans I_{t+1}, soit $S_t^\star \circ \mathcal{T}_{t \leftarrow t+1}$.

```
600         old_bb=nd.find_objects(tmp)
601         for mother_c, sisters_c in lower:
```

- cell_before : sous-image booléenne écrivant la cellule c dans $S_t^\star$
- cell_after : sous-image booléenne écrivant les cellules filles de c dans la nouvelle segmentation seg_from_opt_h
- lost : masque de la sous-image de seg_from_opt_h masquée par le XOR de cell_before et cell_after, c'est donc à la fois ce qui a été perdu et ce qui a été gagné [Rq: sur ma machine, le XOR se comporte comme un OR ...]

```
602             cell_before=tmp[old_bb[mother_c-1]]==mother_c
603             cell_after=np.zeros_like(cell_before)
604             for c in sisters_c:
605                 cell_after+=seg_from_opt_h[old_bb[mother_c-1]]==c
606             lost=seg_from_opt_h[old_bb[mother_c-1]][cell_after^cell_before]
607             max_share=0
608             share_lab=0
609             size={}
```

lost contient les labels de la différence. On cherche donc à vérifier si le plus grand label est le fond, ce qui suppose que c'est le fond qui a gagné sur la cellule.

```
610             for v in np.unique(lost):
611                 size[v]=np.sum(lost==v)
612                 if np.sum(lost==v)>max_share:
613                     max_share=np.sum(lost==v)
614                     share_lab=v
```

On vérifie que 1 (le fond) est bien le label le plus représenté dans la partie perdue. On vérifie aussi que le fond était présent dans la boite englobante de la cellule mere deformée (il aurait fallu vérifier que le fond était adjacent à la cellule mère). Si oui, on va lancer des corrections.

```
615             if share_lab==1 and 1 in tmp[old_bb[mother_c-1]]:
616                 exterior_correction.append((mother_c, sisters_c))
617         from multiprocessing import Pool
618         pool=Pool(processes=nb_proc)
619         mapping=[]
620         for m, daughters in exterior_correction:
621             bb=slices_dilation(old_bb[m-1], maximum=im_ref.shape, iterations=15)
622             im_ref_tmp=deepcopy(im_ref16[bb])
623             seg_ref_tmp=deepcopy(tmp[bb])
624             mapping.append((m, daughters, bb, im_ref_tmp, seg_ref_tmp,MorphosnakeIterations,NIte
```

Appel à perform_ac(), cf section 3.11, page 21.

- m : label de la cellule mère
- daughters : labels des cellules filles
- bb : boite englobante de la cellule mère transformée et dilatée (15 fois)
- im_ref_tmp : sous-image de l'image originale définie par la boite englobante (ce n'est pas nécessairement une image sur 2 octets)

- `seg_ref_tmp` : sous-image de $S_t^\star$ transformée dans I_{t+1}, soit $S_t^\star \circ \mathcal{T}_{t \leftarrow t+1}$
- `MorphosnakeIterations` : `MorphosnakeIterations=10`
- `NIterations` : `NIterations=200`
- `DeltaVoxels` : `DeltaVoxels=10**3`

On récupère ensuite le résultat du morphosnake. S'il y a plusieurs cellules filles, on n'en garde qu'une (la division n'est plus considérée). Note : si le morphosnake est appliqué à deux cellules adjacentes, les parties communes dans le fond atteintes par les 2 morphosnakes seront attribuées à la première qui sera traitée.

Le cas de 3 cellules filles n'est pas considéré.

```
625         outputs=pool.map(perform_ac, mapping)
626         pool.close()
627         pool.terminate()
628         for m, daughters, bb, cell_out in outputs:
629             seg_from_opt_h[bb][seg_from_opt_h[bb]==1 & cell_out]=daughters[0]
630             if len(daughters)==2:
631                 seg_from_opt_h[bb][seg_from_opt_h[bb]==daughters[1]]=daughters[0]
632                 if tuple(daughters) in divided_cells:
633                     divided_cells.remove(tuple(daughters))
634             corres[m]=[daughters[0]]
```

Traitement de `to_fuse_3` : couples (mère, triplets de labels des filles) lorsqu'il y a 3 soeurs. On élimine le plus petit label et on l'attribue au label avec lequel il a la plus grande frontière.

```
635     for c, tf in to_fuse_3:
636         bb=slices_dilation(bounding_boxes[c], maximum=seg.shape, iterations=2)
637         seg_c=np.ones_like(seg_from_opt_h[bb])
638         seg_c[seg_from_opt_h[bb]==tf[0]]=tf[0]
639         seg_c[seg_from_opt_h[bb]==tf[1]]=tf[1]
640         seg_c[seg_from_opt_h[bb]==tf[2]]=tf[2]
641         v1=np.sum(seg_c==tf[0])
642         v2=np.sum(seg_c==tf[1])
643         v3=np.sum(seg_c==tf[2])
644         vol_cells_to_f=[v1, v2, v3]
645         cell_to_f=np.argmin(vol_cells_to_f)
646         tmp=nd.binary_dilation(seg_c==tf[cell_to_f])
647         p1=tf[np.argsort(vol_cells_to_f)[1]]
648         p2=tf[np.argsort(vol_cells_to_f)[2]]
649         im_tmp=np.zeros_like(seg_c)
650         im_tmp[seg_c==p1]=p1
651         im_tmp[seg_c==p2]=p2
652         im_tmp[tmp==False]=0
653         p1_share=np.sum(im_tmp==p1)
654         p2_share=np.sum(im_tmp==p2)
655         if p1_share>p2_share:
656             seg_from_opt_h[seg_from_opt_h==tf[cell_to_f]]=p1
657         else:
658             seg_from_opt_h[seg_from_opt_h==tf[cell_to_f]]=p2
659         corres[c]=[p1, p2]
660         divided_cells.append((p1, p2))
661
662     return seg_from_opt_h, bigger, lower, to_look_at, too_little, corres, exterior_correction
```

- `seg_from_opt_h` : image de segmentation
- `bigger` : liste des couples (mère, filles) où les filles ont un volume plus grand (de plus de 10%) de celui de la mère
- `lower` : liste des couples (mère, filles) où les filles ont un volume plus petit (de plus de 10%) de celui de la mère
- `to_look_at` : liste des couples initiaux (mère, filles) où les filles avaient un volume plus petit (de plus de 50%) de celui de la mère. A priori, les filles ont été recalculées, donc cette liste n'est plus à jour.
- `too_little` : liste des couples initiaux (mère, fille) où la fille avait un petit volume (inférieur à 1000 voxels). Ces filles ont a priori disparu.
- `corres` : table des correspondances (mère, filles), il peut y rester des incohérences.
- `exterior_correction` : liste des couples (mère, filles) où les filles ont un volume plus petit (de plus de 10%) de celui de la mère, et proches du fond, sur lesquels on lancera le morphosnake.

3.13 outer_correction()

Appelé par `segmentation_propagation_seeds_init_and_deform()` (section 3.15, page 32).

```
665 def outer_correction(seg_from_opt_h, exterior_correction,segmentation_file_ref,RadiusOpening=20)
666     """
667     Return an eroded segmentation correcting for potential errors in the morphsnake
668     seg_from_opt_h : segmentated image (SpatialImage)
669     exterior_correction : list of cells that have been corrected using morphsnake algorithm
```

- `seg_from_opt_h` : image de segmentation
- `exterior_correction` : liste des couples (mère, filles) où les filles ont un volume plus petit (de plus de 10%) de celui de la mère, et proches du fond, sur lesquels on a lancé le morphosnake.
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S_t^\star$
- `RadiusOpening` : `RadiusOpening`= 20

```
670     """
671     if exterior_correction!=[]:
672      image_input=segmentation_file_ref.replace('.inr','.seg_from_opt_h.inr')
673         imsave(image_input, SpatialImage(seg_from_opt_h!=1, voxelsize=seg_from_opt_h.voxelsize).a
674         image_output=segmentation_file_ref.replace('.inr','.seg_out_h.inr')
```

Ouverture morphologique (érosion puis dilatation) du masque de l'embryon, cela va lisser les cellules touchant le fond par l'extérieur.

```
675         morpho(image_input,image_output,' -ope -R '+str(RadiusOpening))
676         opened=imread(image_output)
677         cells_to_correct=[i for j in exterior_correction for i in j[1]]
678         os.system('rm -f '+image_input+' '+image_output)
```

XOR entre le masque de l'embryon et son ouverture (qui y est inclus). La différence se trouve donc uniquement dans les cellules de `seg_from_opt_h`.

```
679         to_remove=opened^(seg_from_opt_h>1)
680
```

Plutôt que de faire une boucle sur les cellules, on pourrait juste les cellules à corriger de `seg_from_opt_h` par le masque de l'ouverture ?

```
681         for c in cells_to_correct:
682             seg_from_opt_h[((seg_from_opt_h==c) & to_remove).astype(np.bool)]=1
683     return seg_from_opt_h
```

3.14 segmentation_propagation_seeds_init_and_deform()

```
686 def segmentation_propagation_seeds_init_and_deform(t, segmentation_ref, fused_file, seeds_file,
```

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `segmentation_ref` : image (ie `SpatialImage`) de segmentation à t, $S_t^\star$
- `fused_file` : nom de l'image d'intensite (fusionnée) à $t+1$, I_{t+1}
- `vf_file` : nom de la transformation non-linéaire $\mathcal{T}_{t\leftarrow t+1}$
- `delta_t` : pas de temps, l'image à segmenter est $I_{t+\delta t}$, généralement $\delta t = 1$

```
687     """
688     Steps 2 to 3 of segmentation propagation:
689     create seeds from reference segmentation, then resample it by transformation application
690     -> generation of seeds_file containing the image of seeds which will be used for segmentatio
691     """
692     print 'Create The Seeds from '+str(t)
693
694     seeds_ref=create_seeds(segmentation_ref, max_size_cell=np.inf)
```

Appel à `create_seeds()`, cf section 3.3, page 10. `seeds_ref` est la `SpatialImage` S_t^e [1, section 2.3.3.4] où les cellules de taille supérieure à `min_size_cell` (1000) sont érodées d'au plus 10 itérations pour les cellules et 25 pour le fond.

```
695     imsave(seeds_file, SpatialImage(seeds_ref, voxelsize=seeds_ref.voxelsize))
696
697     print 'Deform Seeds with vector fields from '+str(t)+' to '+str(t+delta_t)
698     apply_trsf(seeds_file, vf_file , path_output=seeds_file, template=fused_file,nearest=True, la
```

La dernière ligne calcule $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$ [1, section 2.3.3.4].

3.15 segmentation_propagation_from_seeds()

```
702 def segmentation_propagation_from_seeds(t, segmentation_file_ref, fused_file,  fused_file_u8 , s
703     RadiusOpening=20,Thau=25,MinVolume=1000,VolumeRatioBigger=0.5,VolumeRatioSmaller=0.1,Morphos
704     verbose=False):
```

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S_t^\star$
- `fused_file` : ce peut être l'image originale I_{t+1} ou l'image à segmenter sur 1 octet. S'appelait `graylevel_file` lors de l'appel
- `fused_file_u8` : c'est l'image à segmenter sur 1 octet. S'appelait `graylevel_file_u8` lors de l'appel
- `seeds_file` : nom de l'image $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$, soit les graines projetées (cellules de $S_t^\star$ érodées puis transformées dans I_{t+1})
- `path_seg_trsf` : nom de l'image de segmentation $S_t^\star$ transformée dans I_{t+1}, soit $S_t^\star \circ \mathcal{T}_{t\leftarrow t+1}$
- `path_h_min` : nom générique des images de h_{min} (paramétré par `TIME`, `HMIN`, et `SIGMA`),

```
    """
    Steps 4 to 9 of segmentation propagation:
    - initial watershed
    - computation of h-minima (get_seeds method)
    - optimal h selection for each cell (get_back_parameters method)
    - build a seeds image from previous information and new segmentation by watershed (get_seeds
    - morphosnake if needed (called from volume_checking method)
    - last corrections with a morphological opening (outer_correction method)
    Returns seg_from_opt_h, lin_tree_information
        seg_from_opt_h : SpatialImage of the segmentation at t+delta_t
        lin_tree_information : updated lineage tree
    """
    from copy import deepcopy
    lin_tree=lin_tree_information.get('lin_tree', {})
    tmp=lin_tree_information.get('volumes_information', {})
    volumes_t_1={k%10**4: v for k, v in tmp.iteritems() if k/10**4 == t}
    h_min_information={}


    print 'Perform watershed with the seeds from method "segmentation_propagation_seeds_init_and
    im_fused=imread(fused_file)
    im_fused_8=imread(fused_file_u8)



    segmentation=watershed(seeds_file, im_fused_8, temporary_folder=os.path.dirname(path_seg_trsf
```

Calcule la segmentation $\tilde{S}_{t+1}$ par ligne de partage des eaux de I_{t+1} avec les graines $S^e_{t+1 \leftarrow t} = S^e_t \circ \mathcal{T}_{t \leftarrow t+1}$.
`segmentation` est une `SpatialImage`

```
    seeds=imread(seeds_file)
    if delSeedsASAP:
        cmd='rm %s'%seeds_file
        if verbose:
            print cmd
        os.system(cmd)
    cells=list(np.unique(segmentation))
    cells.remove(1)
    bounding_boxes=dict(zip(range(1, max(cells)+1), nd.find_objects(segmentation)))
```

Calcul des bounding boxes pour les cellules

```
    treated=[]

    print 'Estimation of the local h-minimas at '+str(t+delta_t)
    nb_cells, parameters=get_seeds(segmentation, h_min_min,h_min_max, sigma, cells, fused_file,
```

Appel à `get_seeds()`, cf section 3.8, page 13. Attention, les graines (minima locaux) sont calculées sur les image originales, encodées sur 2 octets, et non sur des images transformées (par `to_u8()`, ...)

- `segmentation` : image de segmentation $\tilde{S}_{t+1}$
- `h_min_min` : plus petite valeur de h pour le calcul des h-minima
- `h_min_max` : plus grande valeur de h pour le calcul des h-minima

- `sigma` : σ pour le lissage gaussien avant le calcul des h-minima
- `cells` : liste des cellules
- `fused_file` : image originale I_{t+1}
- `path_h_min` : nom générique pour les images de h-minima
- `bounding_boxes` : boites englobantes des cellules

Retourne

- `nb_cells` : liste pour chaque cellule, du nombre de h-minima strictement inclus dans la cellule de $\tilde{S}_{t+1}$. C'est le $\text{Count}^h(c)$ de [1, section 2.3.3.5, page 71].
- `parameters` : liste pour chaque cellule, des paramètres de calcul des h-minima (h et σ)

```
744
745     right_parameters, cells_with_no_seed=get_back_parameters(nb_cells, parameters, lin_tree, cel
```

Appel à `get_back_parameters()`, cf section 3.9, page 15.

- `nb_cells` : liste pour chaque cellule, du nombre de h-minima strictement inclus dans la cellule de $\tilde{S}_{t+1}$. C'est le $\text{Count}^h(c)$ de [1, section 2.3.3.5, page 71].
- `parameters` : liste pour chaque cellule, de tous les paramètres de calcul des h-minima (h et σ)
- `lin_tree` : fichier de linéage
- `cells` : liste des cellules
- `Thau` : τ pour le calcul du score $s(c) = N_{2^+}(c).N_2(c) > \tau$ [1, page 72].

Retourne un tableau avec, pour chaque cellule c, les valeurs de h, σ, et le nombre de graines, ainsi que la liste des cellules sans graines (c'est redondant, puisque ce sont les cellules avec 0 graines).

```
746
```

Appel à `get_seeds_from_optimized_parameters()`, cf section 3.10, page 17.

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `segmentation` : image de segmentation $\tilde{S}_{t+1}$, `SpatialImage`
- `cells` : liste des cellules
- `cells_with_no_seed` : liste des cellules sans graines
- `right_parameters` : (h, σ, nombre de graines) optimaux pour chaque cellule
- `delta_t` :
- `bounding_boxes` : boites englobantes pour les cellules
- `im_fused_8` : I_{t+1} sur un octet, `SpatialImage`
- `seeds` : $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$, soit les graines projetées (cellules de $S^\star_t$ érodées puis transformées dans I_{t+1}), `SpatialImage`
- `parameters` : liste pour chaque cellule, des paramètres de calcul des h-minima (h et σ)

```
747     print 'Applying volume correction '+str(t+delta_t)
748     seeds_from_opt_h, seg_from_opt_h, corres, exterior_corres, h_min_information, sigma_informat
749         right_parameters, delta_t, bounding_boxes, im_fused_8, seeds, parameters, h_min_max, patl
```

Retourne

- `seeds_from_opt_h` : une `SpatialImage` contenant les graines
- `seg_from_opt_h` : une `SpatialImage` contenant la segmentation

- `corres` : un tableau contenant les filiations
- `exterior_corres` : une liste contenant la filiation pour le fond (inutile ?)
- `h_min_information` : une liste contenant les valeurs de h utilises pour chaque cellule
- `sigma_information` : une liste contenant les valeurs de σ utilises pour chaque cellule
- `divided_cells` : une liste contenant les couples de cellules soeurs
- `label_max` : le plus grand label utilis (+1)

```
750
751     print 'Perform volume checking '+str(t+delta_t)
```

Appel à `volume_checking()`, cf section 3.12, page 22.

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `delta_t` :
- `segmentation` : image de segmentation $\tilde{S}_{t+1}$, `SpatialImage`
- `seeds_from_opt_h` : une `SpatialImage` contenant les graines (obtenues avec les paramètres optimaux pour chaque cellule)
- `seg_from_opt_h` : une `SpatialImage` contenant la segmentation obtenue avec les graines de `seeds_from_opt_h`
- `corres` : un tableau contenant les filiations de chaque cellule de la segmentation à t
- `divided_cells` : une liste contenant les couples de cellules soeurs
- `bounding_boxes` : boites englobantes pour les cellules
- `right_parameters` : (h, σ, nombre de graines) optimaux pour chaque cellule
- `im_fused_8` : I_{t+1} sur un octet, `SpatialImage`
- `im_fused` : I_{t+1} sur un ou deux octet, `SpatialImage`
- `seeds` : $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$, soit les graines projetées (cellules de $S^\star_t$ érodées puis transformées dans I_{t+1}) `SpatialImage`
- `nb_cells` : liste pour chaque cellule, du nombre de h-minima strictement inclus dans la cellule de $\tilde{S}_{t+1}$. C'est le $\text{Count}^h(c)$ de [1, section 2.3.3.5, page 71].
- `label_max` : le plus grand label utilis pour les graines (+1)
- `exterior_corres` : une liste contenant la filiation (les labels des graines) pour le fond (inutile ?)
- `parameters` : liste pour chaque cellule, de tous les paramètres de calcul des h-minima (h et σ)
- `h_min_information` : une liste contenant les valeurs de h utilises pour chaque cellule
- `sigma_information` : une liste contenant les valeurs de σ utilises pour chaque cellule
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S^\star_t$
- `path_seg_trsf` : om de l'image de segmentation $S^\star_t$ transformée dans I_{t+1}, soit $S^\star_t \circ \mathcal{T}_{t\leftarrow t+1}$
- `path_h_min` : nom générique pour les images de h-minima
- `volumes_t_1` : liste des volumes des cellules à t (provient de la lecture du linéage)
- ...

```
752     seg_from_opt_h, bigger, lower, to_look_at, too_little, corres, exterior_correction = volume_
753         im_fused_8, im_fused, seeds, nb_cells, label_max, exterior_corres, parameters, h_min_inf
754         nb_proc=nb_proc,Thau=Thau, MinVolume=MinVolume,VolumeRatioBigger=VolumeRatioBigger,Volum
```

`volume_checking()` retourne

- `seg_from_opt_h` : image de segmentation
- `bigger` : liste des couples (mère, filles) où les filles ont un volume plus grand (de plus de 10%) de celui de la mère

- `lower` : liste des couples (mère, filles) où les filles ont un volume plus petit (de plus de 10%) de celui de la mère
- `to_look_at` : liste des couples initiaux (mère, filles) où les filles avaient un volume plus petit (de plus de 50%) de celui de la mère. A priori, les filles ont été recalculées, donc cette liste n'est plus à jour.
- `too_little` : liste des couples initiaux (mère, fille) où la fille avait un petit volume (inférieur à 1000 voxels). Ces filles ont a priori disparu.
- `corres` : table des correspondances (mère, filles), il peut y rester des incohérences.
- `exterior_correction` : liste des couples (mère, filles) où les filles ont un volume plus petit (de plus de 10%) de celui de la mère, et proches du fond, sur lesquels on lancera le morphosnake.

```
755
756     print 'Perform Outer Correction '+str(t+delta_t)
```

Appel à `outer_correction()` (section 3.13, page 31)

- `seg_from_opt_h` : image de segmentation
- `exterior_correction` : liste des couples (mère, filles) où les filles ont un volume plus petit (de plus de 10%) de celui de la mère, et proches du fond, sur lesquels on a lancé le morphosnake.
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S_t^\star$ (le nom sert juste pour créer des noms d'images intermédiaires qui seront effacées)
- `RadiusOpening` : `RadiusOpening`= 20

```
757     seg_from_opt_h = outer_correction(seg_from_opt_h, exterior_correction,segmentation_file_ref,
758
759     print 'Compute Volumes'+str(t+delta_t)
760     volumes=compute_volumes(seg_from_opt_h)
761     volumes_information={}
762     for k, v in volumes.iteritems():
763         volumes_information[(t+delta_t)*10**4+k]=v
764     for m, d in corres.iteritems():
765         if m!=1:
766             daughters=[]
767             for c in d:
768                 if c in volumes:
769                     daughters.append(c+(t+delta_t)*10**4)
770                 else:
771                     print str(c) +' is not segmented'
772             if len(daughters)>0:
773                 lin_tree[m+t*10**4]=daughters
774     lin_tree_information['lin_tree']=lin_tree
775     lin_tree_information.setdefault('volumes_information', {}).update(volumes_information)
776     lin_tree_information.setdefault('h_mins_information', {}).update(h_min_information)
777     lin_tree_information.setdefault('sigmas_information', {}).update(sigma_information)
778
779     return seg_from_opt_h, lin_tree_information
```

3.16 segmentation_propagation()

```
782 def segmentation_propagation(t, fused_file_ref, segmentation_file_ref, fused_file , seeds_file,v
783     membrane_reconstruction_method=None, fusion_u8_method=0, flag_hybridation=False,
```

```
784     RadiusOpening=20,Thau=25,MinVolume=1000,VolumeRatioBigger=0.5,VolumeRatioSmaller=0.1,Morphosm
785     rayon_dil=3.6, sigma_membrane=0.9, manual=False, manual_sigma=7, hard_thresholding=False, ha
786     keep_membrane=False, keep_all=False,  path_u8_images=None, nb_proc_ACE=7,
787     min_percentile=0.01, max_percentile=0.99, min_method='cellinterior', max_method='cellborder'
788     verbose=False):
```

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `fused_file_ref` : nom de l'image d'intensite (fusionnée) à t, I_t
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S_t^\star$
- `fused_file` : nom de l'image d'intensite (fusionnée) à $t+1$, I_{t+1}
- `seeds_file` : nom de l'image des graines (s'appelait `seed_file` lors de l'appel)
- `vf_file` : nom de la transformation non-linéaire $\mathcal{T}_{t\leftarrow t+1}$
- `path_h_min` : nom générique des images de h_{min} (paramétré par `TIME`, `HMIN`, et `SIGMA`), s'appelait `h_min_files` lors de l'appel

```
789     '''
790     Return the propagated segmentation at time t+dt and the updated lineage tree and cell informa
791     t : time t
792     fused_file_ref : path format to fused images
793     segmentation_file_ref : path format to segmentated seeds_images
794     fused_file : fused image at t+dt
795     vf_file : path format to transformation
796     path_h_min : path format to h-minima files
797     h_min_max : maximum value of the h-min value for h-minima operator
798     sigma : sigma value in voxels for gaussian filtering
799     lin_tree_information : dictionary containing the lineage tree dictionary, volume information
800     delta_t : value of dt (in number of time point)
801     nb_proc : number maximum of processors to allocate
802
803     # Modules choice
804
805     membrane_reconstruction_method : if not set or set to 0, the input fused_file is not processe
806                                      if set to 1, the GLACE reconstruction method is going to be
807                                      if set to 2, the GACE reconstruction method is going to be
808
809     fusion_u8_method : select method to convert fused_file into a 8 bits images for the segmenta
810                        if set to 0 (default), calling the historical "to_u8" method
811                        if set to 1, calling the mc_adhocFuse function which enhances the fused i
812                        knowing the semgnetation propagation from previous time point
813
814     flag_hybridation : if set to True and if the membrane_reconstruction_method parameter is pro
815                        then the reconstructed gray level image
816                        used for semgentation_propragation_from_seeds is goind to be ahybridation
817                        fused_file and the result of image reconstruction by the specified method
818
819     path_u8_images : default is None. If provided, saves a copy of the u8 image used for watersh
820
821
822
823     # Glace Parameters (if membrane_reconstruction_method is set to 1 or 2):
```

```
824     # membrane_renforcement
825     sigma_membrane=0.9  # membrane enhancement parameter (in real units, a
826                         # priori 0.9 um is a good choice for data like
827                         # Patrick/Ralph/Aquila)
828     # anisotropicHist /!\ critical step
829     sensitivity=0.99    # membrane binarization parameter, /!\ if failure,
830                         # one should enter in "manual" mode of the function
831                         # anisotropicHist via activation of 'manual' option
832
833     manual=False        # By default, this parameter is set to False. If
834                         # failure, (meaning that thresholds are very bad,
835                         # meaning that the binarized image is very bad),
836                         # set this parameter to True and relaunch the
837                         # computation on the test image. If the method fails
838                         # again, "play" with the value of manual_sigma...
839                         # and good luck.
840     manual_sigma=15     # Axial histograms fitting initialization parameter
841                         # for the computation of membrane image binarization
842                         # axial thresholds (this parameter is used iif
843                         # manual = True).
844                         # One may need to test different values of
845                         # manual_sigma. We suggest to test values between 5 and
846                         # 25 in case of initial failure. Good luck.
847
848     hard_thresholding=False   # If the previous membrane threshold method
849                               # failed, one can force the thresholding with a
850                               # "hard" threshold applied on the whole image.
851                               # To do so, this option must be set to True.
852     hard_threshold=1.0        # If hard_thresholding = True, the enhanced
853                               # membranes image is thresholded using this
854                               # parameter (value 1 seems to be ok for
855                               # time-point t001 of Aquila embryo for example).
856
857     # Tensor voting framework
858     sigma_TV=3.6    # parameter which defines the voting scale for membrane
859                     # structures propagation by tensor voting method (real
860                     # coordinates).
861                     # This parameter shoud be set between 3 um (little cells)
862                     # and 4.5 um(big gaps in the binarized membrane image)
863     sigma_LF=0.9    # Smoothing parameter for reconstructed image (in real
864                     # coordinates). It seems that the default value = 0.9 um
865                     # is ok for classic use.
866     sample=0.2      # Parameter for tensor voting computation speed
867                     # optimisation (do not touch if not bewared)
868     rayon_dil=3.6   # dilatation ray for propagated ROI from time t to t+1
869                     # (default: 3.6, in real coordinates)
870
871     nb_proc_ACE=7   # number of processors for ACE (7 is recommanded)
872
873     '''
874     segmentation_ref=imread(segmentation_file_ref);
```

```
875
876
877     print 'Compute Vector Fields from '+str(t)+' to '+str(t+delta_t)
878     non_linear_registration(fused_file_ref,\
879                             fused_file, \
880                             vf_file.replace('.inr','_affine.inr'), \
881                             vf_file.replace('.inr','_affine.trsf'),\
882                             vf_file.replace('.inr','_vector.inr'),\
883                             vf_file);
```

Calcul de la transformation non-linéaire $\mathcal{T}_{t\leftarrow t+1}$. `non_linear_registration()` est dans `ASTEC/CommunFunctions/cpp_wrappi`

```
884
885     cmd='rm -f '+vf_file.replace('.inr','_affine.inr')+' '+vf_file.replace('.inr','_affine.trsf')
886     if verbose:
887         print cmd
888     os.system(cmd)
889
890     segmentation_propagation_seeds_init_and_deform(t, segmentation_ref, fused_file, seeds_file, v
```

Appel à `segmentation_propagation_seeds_init_and_deform()`, cf section 3.14, page 32. `seeds_file` est le nom de l'image $S^e_{t+1\leftarrow t} = S^e_t \circ \mathcal{T}_{t\leftarrow t+1}$ [1, section 2.3.3.4], soit les graines projetées (cellules de $S^\star_t$ érodées puis transformées dans I_{t+1}).

```
891
892
893     # graylevel image construction for segmentation propagation
894
895     # defining temporary file paths
896     graylevel_file=vf_file.replace('.inr','_graylevel.inr')         # The first input gray level
897     graylevel_file_u8=vf_file.replace('.inr','_graylevel_u8.inr')   # The second input gray level
898     fused_file_u8=vf_file.replace('.inr','_fuse_u8.inr')            # Temporary file
899     path_seg_trsf=vf_file.replace('.inr','_seg_trsf.inr')           # Temporary file
900
901     # segmentation propagation
902     apply_trsf(segmentation_file_ref, path_trsf=vf_file, path_output=path_seg_trsf, template=fuse
```

`path_seg_trsf` est le nom de l'image de segmentation $S^\star_t$ transformée dans I_{t+1}, soit $S^\star_t \circ \mathcal{T}_{t\leftarrow t+1}$

```
903
904     # transformation file deletion
905     cmd='rm -f '+vf_file
906     if verbose:
907         print cmd
908     os.system(cmd)
909
910
911     # fused file u8 vconversion if needed
912     if flag_hybridation or not membrane_reconstruction_method:
913         if  fusion_u8_method==1:
914             mc_adhocFuse(fused_file, path_seg_trsf, fused_file_u8, min_percentile=min_percentile
915                          min_method=min_method, max_method=max_method, sigma=sigma_hybridation, v
916         else:
```

```
917             imsave(fused_file_u8, to_u8(imread(fused_file)))
918
919     # Switch membrane_reconstruction_method
920     if not membrane_reconstruction_method:
921         copy(fused_file_u8, graylevel_file_u8, verbose=verbose)
922         copy(fused_file, graylevel_file, verbose=verbose)
923     if membrane_reconstruction_method == 1:
924         # GLACE reconstruction
925         GLACE_from_resampled_segmentation(fused_file, path_seg_trsf, labels_of_interest='all', ba
926         path_output=graylevel_file, rayon_dil=rayon_dil,
927         sigma_membrane=sigma_membrane, manual=manual, manual_sigma=manual_sigma, hard_thresholdi
928         hard_threshold=hard_threshold, sensitivity=sensitivity, sigma_TV=sigma_TV, sigma_LF=sigma
929         keep_membrane=keep_membrane, keep_all=keep_all,  nb_proc=nb_proc_ACE, verbose=verbose)
930     if membrane_reconstruction_method == 2:
931         # GACE reconstruction
932         out=GACE(fused_file, binary_input=False, path_output=graylevel_file,
933         sigma_membrane=sigma_membrane, manual=manual, manual_sigma=manual_sigma, hard_thresholdi
934         hard_threshold=hard_threshold, sensitivity=sensitivity, sigma_TV=sigma_TV, sigma_LF=sigma
935         keep_membrane=keep_membrane, keep_all=keep_all, verbose=verbose)
936
937
938
939     # reconstructed image and fused image hybridation if needed
940     if membrane_reconstruction_method:
941         if flag_hybridation:
942             Arit(fused_file_u8, graylevel_file, graylevel_file, Mode='max', Type='-o 1', verbose=
943         copy(graylevel_file, graylevel_file_u8, verbose=verbose)
944
945     # temporary images deletion
946     if os.path.exists(fused_file_u8):
947         cmd='rm -f '+fused_file_u8
948         if verbose:
949             print cmd
950         os.system(cmd)
951
952     # u8 image copy if asked
953     if path_u8_images:
954         copy(graylevel_file_u8, path_u8_images, verbose=verbose)
955
956     # segmentation propagation stuff from seeds
957     seg_from_opt_h, lin_tree_information = segmentation_propagation_from_seeds(t, segmentation_fi
958                                                                                path_h_min, h_min_
959                                                                                RadiusOpening=Radi
960                                                                                VolumeRatioSmalle
961                                                                                NIterations=NItera
962                                                                                delSeedsASAP=True
```

Appel à `segmentation_propagation_from_seeds()`, cf section 3.15, page 32.

- `t` : temps pour l'image de référence, on va segmenter l'image suivante, ie $t+1$
- `segmentation_file_ref` : nom de l'image de segmentation à t, $S_t^\star$
- `graylevel_file` : ce peut être l'image originale I_{t+1} ou l'image à segmenter sur 1 octet

- `graylevel_file_u8` : c'est l'image à segmenter sur 1 octet
- `seeds_file` : nom de l'image $S^e_{t+1 \leftarrow t} = S^e_t \circ \mathcal{T}_{t \leftarrow t+1}$, soit les graines projetées (cellules de $S^\star_t$ érodées puis transformées dans I_{t+1})
- `path_seg_trsf` : nom de l'image de segmentation $S^\star_t$ transformée dans I_{t+1}, soit $S^\star_t \circ \mathcal{T}_{t \leftarrow t+1}$
- `path_h_min` : nom générique des images de h_{min} (paramétré par `TIME`, `HMIN`, et `SIGMA`),

```
963
964     # temporary images deletion
965     if os.path.exists(path_seg_trsf):
966         cmd='rm -f '+path_seg_trsf
967         if verbose:
968             print cmd
969         os.system(cmd)
970
971     if os.path.exists(graylevel_file):
972         cmd='rm -f '+graylevel_file
973         if verbose:
974             print cmd
975         os.system(cmd)
976
977     if os.path.exists(graylevel_file_u8):
978         cmd='rm -f '+graylevel_file_u8
979         if verbose:
980             print cmd
981         os.system(cmd)
982
983     return seg_from_opt_h, lin_tree_information
```

4 5-postcorrection.py

```
[...]
    25 from nomenclature import *
    26 from lineage import write_tlp_from_lin_tree,read_lineage_tree,\
    27 write_lineage_tree,timeNamed,timesNamed
    28 from post_correction import apply_cell_fusion,remove_too_little_branches
    29 from lineage_test import pkl_lineage_test, imageDict
[...]
   146 ##########################################
   147 ### Segmentation Post-Correction Stuff ###
   148 ##########################################
   149
   150 # Post correction of the segmentation
   151
   152 # Read the lineage tree (in case it was previously created)
   153 lin_tree_information=read_lineage_tree(path_seg_exp_lineage)
```

Appel à `remove_too_little_branches()` (section 5.8, page 48).

```
   157 # Lineage Tree Post-correction
   158
   159 ### CORRECT THE LINEAGE TREE WITH THE CELLS VOLUMES
   160 lin_tree_cor, new_volumes, to_fuse, been_fused=\
   161 remove_too_little_branches(lin_tree_information['lin_tree'], \
   162 lin_tree_information['volumes_information'], \
   163 p.postcor_Volume_Threshold, soon=p.postcor_Soon)
   164
```

Appel à `apply_cell_fusion()` (cf section 5.3, page 44).

Notons que le linéage mis à jour n'est pas passé à `apply_cell_fusion()` qui prend en entrée le linéage avant correction. De même les nouveaux volumes (qui ne peuvent pas avoir pris en compte les fusions de branches où il y a division), ne seront pas tenus en compte. .

```
   166 ### APPLYING THE CORRECTION ON THE IMAGES
   167 apply_cell_fusion(lin_tree_information['lin_tree'], \
   168    lin_tree_information['volumes_information'], \
   169    to_fuse,os.path.join(path_seg_exp,path_seg_exp_files), \
```

Il faut vérifier si les volumes (mis à jour) sont bien remontés. .

```
   170                   os.path.join(path_post_exp,path_post_exp_files),p.begin+p.raw_delay, \
   171    p.end+p.raw_delay, p.delta)
   172
   173 #SAVE THE NEW LINEAGE TREE
   174 lin_tree_information['lin_tree']=lin_tree_cor
   175
   176
   177 ### SAVE THE FINAL LINEAGE TREE
   178 write_lineage_tree(path_post_exp_lineage,lin_tree_information)
[...]
```

5 ASTEC/post_correction.py

5.1 timeNamed()

```
13 def timeNamed(filename, time):
14     time_point = ('00' + str(time))[-3:]  # Format time on 3 digit
15     return filename.replace('$TIME', time_point)+".inr"
16
```

5.2 fuse_cells()

```
18 def fuse_cells(couple, lin_tree, bb, im, t):
19     """
20     Return an segmented image where the cells in couple have been fused
21     couple : tuple of cells to fuse, couple[0] is fused to couple[1]
22     lin_tree : lineage tree
23     bb : bounding box of the couple of cells in ``couple``
24     im : segmented image to fuse
25     t : time point of im
26     """
27     inv_lin_tree={ v : k for k, values in lin_tree.iteritems() for v in values }
28     give=couple[0]%10**4
29     get=np.array(couple[1])%10**4
```

give est la cellule à enlever, get est un tableau de cellules avec lesquelles fusionner

```
30     possibles=np.array(lin_tree[inv_lin_tree[t*10**4+give]])
31     possibles=possibles[possibles!=10**4*t+give]%10**4
32     possibles=np.array([p for p in possibles if p in get])
33     get=possibles
```

on récupère l'ensemble des cellules avec lesquelles fusionner qui sont les soeurs de la cellule à enlever. En cas de division le long de la branche soeur, cela suppose une certaine mise à jour de l'arbre de linéage (cf lignes 253 à 260 de remove_too_little_branches(), section 5.8).

```
34     if len(get)==1:
35         final_get=get[0]
36     else:
37         if bb is None:
38             bb=nd.find_objects(im)
39         im_tmp=im[bb[give-1]]
40         im_tmp_d=nd.binary_dilation(im_tmp==give)
41         borders_b=im_tmp_d & (im_tmp!=give)
42         borders=im_tmp[borders_b]
43         count=[]
44         for g in get:
45             count.append([g, np.sum(borders==g)])
46         count=np.array(count)
47         final_get=count[np.argmax(count[:,1]), 0]
48     return bb, (give, final_get)
49
```

5.3 apply_cell_fusion()

```
52 def apply_cell_fusion(lin_tree,volumes, to_fuse, segmented_files,segmented_fused_files,begin, en
53     """
54     Fuse all the cells from the information given by the lineage correction for a time series
55     begin : starting point
56     end : ending point
57     step : dt
58     path : path to the working folder
59     folder : folder containing the segmentations (in path)
60     path_lin_tree : path to the lineage tree
61     to_fuse_file : path the the file containing the informations on the cells to fuse
62     path_file_format : format of the segmented files
63     """
64     print 'Process cell fusion'
65 
66     inv_lin_tree={ v : k for k, values in lin_tree.iteritems() for v in values }
67 
68     for t in range(begin, end+1, step):
69         current_state=deepcopy(lin_tree)
70         print ' ->Cell fusion at '+str(t)
71         bb=None
72         time=('00'+str(t))[-3:]
73         time_prev=('00'+str(t-1))[-3:]
74         tf=to_fuse.get(t, '')
75         ext_cor=[]
76         segmented_file=timeNamed(segmented_files,t)
77         im=imread(segmented_file)
78         treated=[]
79         mapping=[]
80         if tf!='':
81             if im is None:
82                 im=imread(file_name)
83             if mapping==[]:
84                 mapping=np.array(range(np.max(im)+1), dtype=np.uint16)
85             getting=[]
86             for couple in tf:
87                 bb, trsf=fuse_cells(couple, lin_tree, bb, im, t)
88                 getting.append(trsf)
```

`trsf` est un tuple (ancien label, nouveau label) calculé par `fuse_cells()`, on les conserve dans l'array `getting`

```
89             getting={i:j for i,j in getting}
```

on transforme l'array `getting` en un dictionnaire où `getting(ancien label) = nouveau label`

```
90             roots=set(getting.keys()).difference(set(getting.values()))
```

on a les labels de `getting` qui n'ont pas de prédécesseur

```
91             final_pairing={}
92             for r in roots:
93                 n=r
```

```
                to_change=[]
                while getting.has_key(n):
                    to_change.append(n)
                    n=getting[n]
                    final=n
                for n in to_change:
                    getting[n]=final
```

pour les labels de `roots`, on les suit pour les avoir tous, et on leur donne le même label (ici le dernier)

```
            for give, get in getting.iteritems():
                inv_lin_tree={ v : k for k, values in lin_tree.iteritems() for v in values }
                if give<mapping.shape[0]:
                    mapping[give]=get
                tmp=lin_tree.get(10**4*t+get, [])
                tmp.extend(lin_tree.get(10**4*t+give, []))
                lin_tree[inv_lin_tree[10**4*t+get]].remove(10**4*t+give)
                lin_tree[10**4*t+get]=tmp
                lin_tree.pop(10**4*t+give, None)
                volumes[10**4*t+get]=volumes[10**4*t+get]+volumes[10**4*t+give]
```

Mise à jour du linéage et des volumes

```
        if mapping!=[]:
            im=SpatialImage(mapping[im], voxelsize=im.voxelsize)

        segmented_fused_file=timeNamed(segmented_fused_files,t)
        imsave(segmented_fused_file, im)

```

5.4 get_volumes()

```
def get_volumes(n, vol, lin_tree):
    """
    Return the volumes of cell n and its progeny up to division
    n : starting cell
    vol : dictionary of volumes
    lin_tree : lineage tree
    """
    tmp=[vol[n]]
    while len(lin_tree.get(n, ''))==1:
        tmp.append(vol[n])
        n=lin_tree[n][0]
    return tmp
```

5.5 soon_to_divide()

```
def soon_to_divide(c, tree, reverse_tree, len_min,time_begin,time_end):
    """
    Return if a division is too soon (mother divided to late or cell divided too early)
    c : daughter cell to check
    tree : lineage tree
    reverse_tree : reversed lineage tree
```

```
137     len_min : minimum length allowed
138     time_begin: Starting time point
139     time_end : final time of the movie
140     """
141     cell=deepcopy(tree[reverse_tree[c]])
142     cell.remove(c)
143     cell=cell[0]
```

`reverse_tree[c]` est la cellule parente de `c`, celle où il y a bifurcation. `tree[reverse_tree[c]]` est donc l'ensemble des cellules filles de celle-ci, filles dont `c` fait partie. Au final, on récupère la sœur de `c`

```
144     out=[cell]
145     while len(tree.get(cell, []))==1:
146         cell=tree[cell][0]
147         out.append(cell)
```

`out` est donc la branche sœur de la branche dont `c` est la première cellule.

```
148     cell=reverse_tree[c]
149     out2=[cell]
150     while reverse_tree.has_key(cell):
151         cell=reverse_tree[cell]
152         out2.append(cell)
```

`out2` est la branche qui remonte et qui part de la cellule mère de `c`. Elle devrait s'arrêter à la première bifurcation et non remonter jusqu'au début

```
153     return (len(out)<len_min and out[-1]/10**4!=time_end) or (len(out2)<len_min and out2[-1]/10*
```

Cette fonction renvoir `True` si l'une des branches, sœur ou mère, est trop courte (à condition que la branche sœur ne soit pas une branche finissant au dernier point de temps, ou que la branche mère ne commence pas au premier point de temps). Comme la branche mère est mal construite, elle commence toujours au premier point de temps ...

5.6 branches_to_delete()

```
159 def branches_to_delete(tree, volumes, threshold, reverse_tree, soon=False,ShortLifespan=25,Pearso
160     """
161     Return the list of cells that have to be removed sorted from the yougest to the oldest
162     tree : lineage tree
163     volumes : dictionary of volumes information
164     reverse_tree : reversed lineage tree
165     soon : True if the cell life span has to be taken into account
166     """
167     from scipy.stats.stats import pearsonr
168     nodes=list(set(tree.keys()).union(set([v for values in tree.values() for v in values])))
169     leaves=set(nodes)-set(tree.keys())
170     last_time=max(leaves)/10**4
171     leaves_to_delete=[l for l in leaves if (((l/10**4)<last_time) or (volumes[l]<threshold))]
```

Cellules terminales des branches candidates pour un effacement.

- la cellule n'est pas au dernier instant de la séquence
- ou elle a un petit volume

```
172     branche_max_val=0
173     branche_max=[]
174     for l in leaves_to_delete:
```

Pour chaque branche potentielle

```
175         b=[]
176         b.append(l)
177         while len(tree.get(reverse_tree.get(l, ''), ''))==1:
178             l=reverse_tree[l]
179             b.append(l)
180         b.reverse()
```

La branche est extraite, la division n'est pas incluse.

```
181         if (min(b)>branche_max_val):
```

On considère la branche si sa division (`min(b)` est la première cellule après la division) est plus tardive. Note: ce test prend aussi en compte le numéro de la cellule ...

```
182             if len(b)<ShortLifespan: #Time steps length
183                 branche_max=list(b)
184                 branche_max_val=min(b)
```

On peut effacer la branche si elle est trop courte (même si elle commence au tout début de la séquence ...).

```
185             elif reverse_tree.get(min(b), '')!='':
```

Sinon (la branche n'est pas trop courte), et si elle ne commence pas au tout début de la séquence

```
186                 n1, n2=tree[reverse_tree[min(b)]]
187                 tmp1, tmp2=get_volumes(n1, volumes, tree), get_volumes(n2, volumes, tree)
188                 common_len=min(len(tmp1), len(tmp2))
189                 P=pearsonr(tmp1[:common_len-1], tmp2[:common_len-1])
```

`pearsonr()` renvoie une erreur si l'une des séquences est constante. On pourrait aussi tester si `common_len` est suffisamment grand

```
190                 if P[0]<-PearsonThreshold:# and deriv>4*10**4:
191                     branche_max=list(b)
192                     branche_max_val=min(b)
193                 elif soon and soon_to_divide(min(b), tree, reverse_tree,ShortLifespan,time_begin
```

si `soon` est vrai, appel à `soon_to_divide()`

```
194                     branche_max=list(b)
195                     branche_max_val=min(b)
196
197     branche_max.reverse()
198     return branche_max
```

5.7 get_sisters()

```
200 def get_sisters(lin_tree_out, mother, init, end):
201     """
202     Build the list of all the progeny of the sister cells of init
203     lin_tree_out : lineage tree
```

```
204     mother : common mother of sisters and init cell
205     init : cell id
206     end : final time point
207     """
208     sisters=[s for s in lin_tree_out[mother] if s!=init]
209     out=[]
210     tmp=sisters
211     while tmp!=[] and sisters[0]/10**4<=end:
```

ne vérifie pas si `sisters` est multiple (s'il y a eu division)

```
212         out.append(sisters)
213         tmp=[]
214         for s in sisters:
215             if lin_tree_out.has_key(s):
216                 tmp.extend(lin_tree_out[s])
217         sisters=tmp
218     return out
```

5.8 remove_too_little_branches()

```
223 def remove_too_little_branches(lin_tree, volumes, threshold=2000, soon=False,ShortLifespan=25,Pea
224     """
225     Build a corrected lineage tree (based on volume correlation and cell life span).
226     Return a list of cells to fuse, new volumes and new lineage tree
227     lin_tree : lineage tree
228     volumes : dictionary of volumes
229     threshold : volume low threshold for final cells (in voxels)
230     soon : True if the cell life span has to be taken into account
231     """
232     print 'Process remove too little branches'
233     reverse_tree={v:k for k, values in lin_tree.iteritems() for v in values}
234     branches=branches_to_delete(lin_tree, volumes, threshold, reverse_tree, soon=soon,ShortLifesp
235     lin_tree_out=deepcopy(lin_tree)
236     to_fuse={}
237     new_volumes=deepcopy(volumes)
238     while list(branches)!=[]:
239         reverse_tree={v:k for k, values in lin_tree_out.iteritems() for v in values}
240         b=branches
241         last=b[-1]
242         mother=reverse_tree.get(last, '')
243         if mother!='':
244             tmp=get_sisters(lin_tree_out, mother, last, b[0]/10**4)
```

`tmp` est une liste de liste, où chaque liste correspond aux sœurs à fusionner pour chaque instant (comme `get_sisters()` passe à travers les divisions, il peut y avoir plusieurs sœurs pour un seul point de temps). `b` est ordonné à partir de la feuille, tandis que `tmp` est ordonné à partir de la division.

```
245             for t in tmp:
246                 time=t[0]/10**4
247                 cell=[c for c in b if c/10**4==time]
248                 to_fuse.setdefault(time, []).append((cell[0], t))
```

`to_fuse` est un dictionary où les clefs sont les instants (time point). Les entrées sont des listes, et chaque élément de la liste est un tuple (cellule à enlever, [cellule(s) avec qui fusionner]) où le premier élément est une cellule de la branche à enlever (la cellule de cet instant) et le second élément la liste de ses sœurs avec laquelle il faut la fusionner. `to_fuse` sera utilisé par la fonction `apply_cell_fusion()` (section 5.3) pour effectivement faire la fusion dans les images.

```
249                 if len(t)==1 and new_volumes.has_key(t[0]) and new_volumes.has_key(cell[0]):
250                     new_volumes[t[0]]=new_volumes[cell[0]]+new_volumes[t[0]]
```

La mise à jour des volumes ne se fait que pour les cellules à fusionner qui n'ont qu'une seule sœur. Il aurait fallu supprimer `cell[0]` du dictionnaire des volumes ...

```
251             i=1
252             while (i<=len(b)) and (b[-i]/10**4 in [time_cell[0]/10**4 for time_cell in tmp]):
253                 n=b[-i]
```

On parcourt `b` dans l'inverse (de la fin vers la début), `b` étant construit depuis la feuille

```
254                 sis=tmp[np.argwhere(np.array([time_cell[0]/10**4 for time_cell in tmp])==n/10**4
```

- `time_cell[0]/10**4 for time_cell in tmp` est la liste des time point du sous-arbre sœur
- `np.array([time_cell[0]/10**4 for time_cell in tmp])==n/10**4` est une liste de booléen indiquant les sœurs du même time point que le point de la branche
- `np.argwhere()` donne le tableau d'indice où les booléens sont vrais
- `tmp[np.argwhere(np.array([time_cell[0]/10**4 for time_cell in tmp])==n/10**4)[0, 0]]` est la liste des sœurs du point à fusionner
- `sis` est la première de ces soeurs

```
255                 lin_tree_out[reverse_tree[n]].remove(n)
256                 lin_tree_out.pop(n, None)
257                 if i+1<=len(b):
258                     lin_tree_out.setdefault(sis, []).append(b[-(i+1)])
259                     reverse_tree[b[-(i+1)]]=sis
260                 i+=1
261         else:
262             for n in b:
263                 lin_tree_out.pop(n, None)
264                 reverse_tree.pop(n, None)
```

on traite ici les branches dont la mère n'existe pas, donc commençant ex nihilo ou au début de la séquence. Il manque le traitement des volumes, et la fusion (avec le fond ?) des cellules effacées.

```
265         branches=branches_to_delete(lin_tree_out, new_volumes, threshold, reverse_tree, soon=soo
266
267     #### FUSE TOO EARLY DIVISIONS
268     from scipy.stats.stats import pearsonr
269     cells_list=[(i[0],i[1]) for i in lin_tree_out.values() if len(i)==2]
```

extrait les premières cellules après une bifurcation, revient donc à regarder l'ensemble des bifurcations.

```
270     P={}
271     deriv={}
272     for n1, n2 in cells_list:
273         tmp1, tmp2=get_volumes(n1, new_volumes, lin_tree_out), get_volumes(n2, new_volumes, lin_
274         common_len=min(len(tmp1), len(tmp2))
275         if len(tmp1)>8 and len(tmp2)>8:
276             P[(n1, n2)]=pearsonr(tmp1[:common_len-1], tmp2[:common_len-1])
```

Si la longueur des branches est suffisamment longue (¿8), on calcule le coefficient de Pearson entre les volumes des 2 branches

```
277     to_check=[k for k, v in P.iteritems() if v[0]<-.80] #.80 WHAT ?
```

On examine donc les branches qui ont un coefficient de corrélation de Pearson plus petite que -0.8 (c'était -0.9 pour la première étape)

```
278     scores_window={}
279     for c1, c2 in to_check:
280         tmp1, tmp2=get_volumes(c1, volumes, lin_tree_out), get_volumes(c2, volumes, lin_tree_out
281         scores=[]
282         common_len=min(len(tmp1), len(tmp2))
283         for i in range(0, common_len-4):
284             scores.append(pearsonr(tmp1[i:i+5], tmp2[i:i+5])[0])
285         scores_window[(c1, c2)]=np.array(scores)
```

On calcule le coefficient de corrélation de Pearson sur une fenêtre glissante de longueur 5 le long de la branche.

```
286
287     been_fused={}
288     for c, scores in scores_window.iteritems():
289         if (np.array(scores)<-.8).all(): #.8 WHAT ?
290             first_size=len(scores)-1
291         else:
292             out=scores[1:]-scores[:-1]
```

`out[i] = score[i+1] - score[i]` c'est donc la différence (dérivée) du score

```
293             sizes=np.argsort(out)[::-1]
```

`sizes` donne les indices de `out` pour un tri décroissant (de la plus grande dérivée à la plus petite)

```
294             sizes=np.array([s for s in sizes if scores[s]<-.8])
```

`sizes` donne les indices de `out[i] = score[i+1] - score[i]` pour un tri décroissant, avec `score[i] < -0.8`

```
295             first_size=sizes[sizes<len(scores)/2]
```

On ne récupère que les indices qui sont dans la première moitié, donc les plus proches de la bifurcation

```
296             if first_size!=[]:
297                 first_size=first_size[0]
298             else:
299                 first_size=-1
300         c1, c2=c
301         for i in range(first_size+1):
```

```
302             next_c1=lin_tree_out.get(c1, [0])[0]
303             next_c2=lin_tree_out.get(c2, [0])[0]
304             lin_tree_out.setdefault(c1, []).extend(lin_tree_out.get(c2, []))
305             for tmp_c in lin_tree_out[c1]:
306                 reverse_tree[tmp_c]=c1
307             lin_tree_out[reverse_tree[c1]]=[c1]
308             lin_tree_out.pop(c2, None)
309             time=c1/10**4
310             to_fuse.setdefault(time, []).append((c2, [c1]))
311             new_volumes[c1]=new_volumes[c1]+new_volumes[c2]
312             c1=next_c1
313             c2=next_c2
314             been_fused[c1]=1
315     return lin_tree_out, new_volumes, to_fuse, been_fused
```

References

[1] Léo Guignard. *Quantitative analysis of animal morphogenesis: from high-throughput laser imaging to 4D virtual embryo in ascidians.* Theses, Université Montpellier, December 2015.

[2] P. Màrquez-Neila, L. Baumela, and L. Alvarez. A morphological approach to curvature-based evolution of curves and surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(1):2–17, Jan 2014.