Système de simulation

Ancien système (TimeSignal, Signal4D, SpatialConfig, BOLDModel, ...)

Tout d'abord un petit retour sur le premier système mis en place dans pyhrf, pour mettre en avant ses limites et les erreurs de développements.
Ce système utilise principalement la fonction pyhrf.boldsynth.scenario.createBold qui elle même constrtuit un objet par brique de simulation (eg NRL, HRF, ...) et les rassemble tous dans un BOLDModel object.

Désavantages :

- La fonction createBold n'offre que des noms de scenarios comme paramètres et on ne peut donc pas changer facilement par exemple la variance de bruit (il faut changer la valeur en dur dans une sous-fonction). C'est d'ailleurs pour ça que dans pas mal de scripts, elle n'est pas utilisée directement. Le corps de cette fonction est alors recopié et adapté. On se retrouve avec pas mal de code dupliqué et des scripts qui divergent souvent
- il y a trop de design inutile : la plupart des briques de simulation hérite de Signal4D et TimeSignal. L'objectif au départ de ces classes était de fournir des opérations d'ensemble haut niveau (concaténation, ajout, rééchantillonnage, ...)
- A l'usage, ça n'apporte finalement rien par rapport à utiliser directement des numpy array.
- pour gérer la géométrie, comme par exemple le mapping entre indices dans un tableau plat et coordonnées 3D, la classe SpatialMapping est instanciée et cet objet est utilisé par toutes les briques de simulation. Là encore, beaucoup de design inutile. Historiquement, c'était pour gérer explicitement ce mapping alors qu'on peut le faire implicitement avec un mask numpy.
- le nombre de condition est plus ou moins figé à 2 maximum.

Pour résumé sur cet ancien système :

- pas assez modulaire: difficile de modifier une brique de simulation, d'en ajouter de nouvelles ...
- trop de design: il y a 2 voire 3 couches de classe avant d'atteindre la fonction qui produit vraiment les données
- difficile à scripter

Nouveau système pipeline-like

- exposer au maximum les fonctions qui produisent vraiment les données.
- travailler avec des numpy array autant que possible, sans reposer sur de nouvelles classes.
- sauvegarder vers du nifti pour avoir des données simulées qui puissent s'utiliser comme les données réelles en entrées des analyses.

Fonctions "unitaires"

Un premier point important est que chaque brique de simulation est simplement une fonction qui ne renvoie qu'une seule quantité.
On évite donc des trucs du genre:

```
def create_labels_and_nrls(params):
     labels = read_labels()
     nrls = numpy.random.randn()
     return labels, nrls
```

où la génération des labels et des nrls est liée et donc il est difficile par la suite de ne vouloir que changer la génération des nrls.
On préfèrera :

```
def create_labels(params_labels):
      return random_labels

def create_nrls(labels, params_nrls):
     return random_nrls_from_labels
```

Applatissement des données

Le deuxième point important est la gestion de la géométrie. Manipuler des tableaux 3D n'est pas très pratique et on veut éviter d'avoir à boucler sur les 3 dimensions de l'espace quand on veut boucler sur les voxels. On préfère donc manipuler des tableaux 1D

applatis. Pour conserver l'information spatiale, on sauvegarde par ailleurs une liste d'adjacence (ou graphe) qui indique quelles positions sont voisines desquelles.
Exemple:

```
#define some data and mask:
data = numpy.arange(4*5*6).reshape(4,5,6)
mask = numpy.zeros((4,5,6), dtype=int)
mask[1:3,1:3,1:2] = 1

#flatten data using mask
flat_data_inside_mask = data[numpy.where(mask)]
graph = pyhrf.boldsynth.graph.graph_from_lattice(mask)

#unflatten:
data_vol = numpy.zeros(mask.shape, dtype=flat_data_inside_mask.dtype)
data_vol[numpy.where(mask)] = flat_data_inside_mask
```

Dans le cadre des données simulées, le graphe n'est utile que lorsqu'on en a besoin (lapalissade...). La plupart du temps, les cartes de labels sont des cartes déssinées à la main et donc pas besoin de graphe. Il n'y a pas d'autre élément qui nécessite d'information spatiale. On peut ne travailler tout le temps sur les données applaties et reshaper à la fin.

Pipeline

Pour gérer l'aspect hierarchique de la simulation, un système de pipeline est proposé.
Le principe est de brancher les arguments d'une fonction aux sorties d'autres fonctions.
Exemple:

```
# simulation bricks:
def create_quantity_1(param_1, param_2):
      return numpy.random.randn( ....)

def create_quantity_2(quantity_1, param_3):
      return quantity_1 + numpy.random.randn(...)

# simulation pipeline:
simulation_steps = {
    'param_1' : 2.3,
    'param_2' : 'valeur',
    'quantity_1' : create_quantity_1,
    'param_3' : 4,
    'quantity_2' : create_quantity_2,
}
simulation_pipeline = UpdateGraph(simulation_steps)
simulation_pipeline.update_all()
simulation = simulation_pipeline.get_values()

# All simulated quantities are gathered in a dict:
# simulation['quantity_1']
# simulation['quantity_2']
```

En fait, la partie simulation pipeline est équivalente à :

```
p_1 = 2.3
p_2 = 'valeur'
q_1 = create_quantity(param_1=p1, param_2=p2)
p_3 = 4
quantity_2 = create_quantity(quantity_1=q_1, param_3=p3)

simulation = {
     ...
    'quantity_1' : q_1,
    'quantity_2' : quantity_2,
    }
```

L'interface est donc assez légère, à travers un dictionnaire python. Il y a même un peu moins de code répété puisque les paramètres ne sont définis qu'une fois alors que le code explicite réécrit chaque paramètre pour chaque appel de fonction.
Les fonctionnalités apportées par ce système de pipeline (class UpdateGraph):
- le branchement des briques est implicite (pas besoin de s'occuper de l'ordre)
- les quantités peuvent être sauvées dans du cache. Lors d'un appel suivant d'une même fonction avec exactement les mêmes paramètres d'entrée, le résultat sera lu depuis le disque plutôt que réévalué. Ca permet d'aller plus vite mais également de sauvergarder un résultat généré aléatoirement pour le réutiliser plus tard.
- il est possible d'avoir un visuel du pipeline sous forme de graphe (cf fichier joint). Pratique pour savoir ce qu'on fait.

Export vers nifti

Pour ce qui est de la sauvegarde vers du nifti, c'est fait un peu à la main, quantité par quantité.
Extrait de la fonction pyhrf.boldsynth.scenarios.simulation_save_vol_outputs :

```
if simulation.has_key('drift'):
        fn_drift = op.join(output_dir, 'drift.nii')
        drift_vol = unflatten_array(simulation['drift'], mask_vol, flat_axis=1)
        write_volume(drift_vol, fn_drift, vol_meta)
```

Le problème est ici de devoir écrire une étape de sortie pour chaque brique de simulation alors que la chose pourrait être rendue implicite.
Il faudrait savoir, pour une quantité donnée, quelle est la signification des axes pour pouvoir orienter convenablement le volume lors de la sauvegarde en nifti. Il faudrait également pouvoir sauvegarder dans le nifti les informations associées aux axes pour pouvoir les recharger par la suite.
Pour ce faire, numpy devrait justement intégrer un système sémantique sur les axes. Mais ce n'est pas encore pour tout de suite ...
En attendant, j'avais codé la classe Cuboid, qui encapsule un numpy.ndarray et permet de faire ce genre de chose. Elle supporte maintenant des entrées / sorties vers du nifti.

L'avantage de nommer les axes est aussi de stocker explicitement une information qui tient d'habitude à une convention et d'éviter des bugs en utilisant un axe à la place d'un autre (eg confondre l'axe "condition" et l'axe "classe de mélange")

TODO: *Affaire à suivre, faire une page de Wiki dédiée ...*

Milestones

A partir de tout ça, voici une proposition pour fixer ce système de simu:

1. commencer par le simple et le plus sûr : écrire les briques de simulation (ou les extraire de code déjà existant)

- Une fonction = une quantité. Essayer de ne retourner qu'une seule valeur.
- Bien documenter chaque fonction en terme de paramètre d'entrée, de valeur de retour et surtout de convention sur les axes (cf confusion axes condition et classe de mélange plus haut).
- Faire quelques sanity checks sur les paramètres d'entrée en début de fonction.
- Continuons à tout mettre dans pyhrf.boldsynth.scenarios

Exemple d'une brique de simulation :

```
def create_stim_induced_signal(nrls, rastered_paradigm, hrf, condition_defs,
                               dt, hrf_territories=None):
    """
    Create a stimulus induced signal from neural response levels, paradigm and
    HRF (sum_{m=1}^M a^m X^m h)
    For each condition, compute the convolution of the paradigm binary sequence
    'rastered_paradigm' with the given HRF and multiply by nrls. Finally
    compute the sum over conditions.

    Return a bold array of shape (nb scans, nb voxels)
    """

    assert nrls.ndim == 2 #condition, voxel
    assert rastered_paradigm.ndim == 2 #condition, scan
    assert rastered_paradigm.shape[0] == nrls.shape[0] #consitency of nb cond
    assert rastered_paradigm.shape[0] == len(condition_defs) #consitency of nb cond
```

```
    npos = nrls.shape[1]
    duration_dt = len(hrf[:,0]) + len(rastered_paradigm[0]) - 1
    bold = np.zeros((duration_dt , npos))

    for ipos in xrange(npos):
        h = hrf[:,ipos]
        for ic,c in enumerate(condition_defs):
            activity = np.array(rastered_paradigm[ic,:], dtype=float)
            activity[np.where(activity==1)] = nrls[ic,ipos]

            bold[:,ipos] += np.convolve(activity, h)

    return bold
```

2. Dans des scripts, pour notre utilisation de tous les jours, utiliser ces briques de simulation directement ou bien en utilisant le système de pipeline et formuler des critiques / besoins. Faire ressortir des redondances et donc des factorisations possibles.

3. Tester la classe Cuboid

Exemple d'utilisation.

Quelques ex
h3. Avec / sans Cuboid

Files

simulation_graph.png	145 KB	19/09/2011	Vincent, Thomas