

# **Aims tutorial**

---

## Aims tutorial

---

---

## Table of Contents

Foreward .....	1
Basic command lines .....	2
AimsFileConvert: Performs file format and data conversion .....	2
AimsSubVolume: Carve a subvolume in the input volume .....	2
AimsThreshold: Threshold on data .....	3
AimsGraphMesh: Performs graph storage conversion and sub-buckets meshing. This command is an improved version of AimsGraphConvert .....	3
AimsRoiFeatures: Compute scalar features (mean, volume ...) from regions of interest. ....	4
Conversion .....	5
AimsDiffusionBundleToRoi: conversion from bundles.bundles to ROI graph .....	5
AimsGraphConvert: conversion from label image to ROI graph .....	5
Table of format conversions .....	5
Calculation of images .....	7
AimsLinearComb: sum 2 activation maps .....	7
Handling meshes .....	8
Creation of a cube mesh from a point list .....	8
AimsZCat: concatenates volumes (along Z axis), meshes or buckets .....	8
Handling referentials and transformations .....	9
Coordinates systems in AIMS .....	9
AIMS and Anatomist .....	9
SPM .....	10
Changing between SPM and AIMS .....	12
Issues .....	13
Technical details .....	13
Compute the inverse transformation .....	15
Compose a transformation .....	15
Handling graphs .....	16
Copy a set of graph attributes to another graph .....	16
Rigid registration .....	17
AimsManualRegistration: manual registration between 2 volumes from 3 specific landmarks. ....	17
AimsMIRregister: registration based on mutual information. ....	17
Advanced level .....	19
Get a symmetrical ROI .....	19
Programming with AIMS in Python language .....	21
Using data structures .....	21
Module importation .....	21
IO: reading and writing objects .....	21
Volumes .....	22
Meshes .....	28
Textures .....	31
Buckets .....	32
Graphs .....	33
Other examples .....	34
Using algorithms .....	34
PyAIMS / PyAnatomist integration .....	38

---

## List of Figures

1. Select label .....	3
2. Viewing of non-meshed and meshed ROI .....	4
1. Sum of 2 activation maps .....	7
1. Referentials and normalization transformations .....	15
1. 3D volume: value 12 at voxel (100, 100 ,60) .....	22
2. Distance example .....	23
3. Thresholded Audio-Video T-map .....	24
4. Downsampled anatomical image .....	25
5. 3D volume containing a cube .....	25
6. Flying saucer mesh .....	29
7. Inflated mesh .....	30
8. Inflated mesh with timesteps .....	31
9. Computed time-texture vs 3D fusion .....	32
10. Thresholded T1 MRI .....	35
11. Closing of a thresholded T1 MRI .....	35
12. Head mesh .....	36
13. Generated icosahedron and arrow .....	37
14. Interpolated vs not interpolated texture .....	
15. Aimsalgo resampling .....	38
16. 3D volume modified with numpy .....	40
17. Modified cut mesh .....	41

---

## List of Tables

1. Table of format conversions .....	6
1. Summary to preserve the ROI volume .....	20

---

# Foreward

## Disclaimer:

All contributors of BrainVISA project are very happy to propose tools, libraries, demonstration data and documentation available free of charge on <http://brainvisa.info>. The BrainVISA project is focused on neuroimaging and the development of tools only in a context of research. Although every care has been taken by all contributors of the BrainVISA project no warranty can be given in respect of the accuracy, reliability, up-to-dateness or completeness of the information contained within BrainVISA project. In the same way no one will be responsible or liable for any loss or damage of any sort.

Any contributor reserves the right to alter or remove the content, in full or in part, without prior notice.

In order to work through the following sections, please download the demonstration data from one of the following links:

- [ftp://ftp.cea.fr/pub/dsv/anatomist/data/demo\\_data\\_td\\_2007.zip](ftp://ftp.cea.fr/pub/dsv/anatomist/data/demo_data_td_2007.zip)
- [ftp://ftp.cea.fr/pub/dsv/anatomist/data/demo\\_data\\_td\\_2007.tar.gz](ftp://ftp.cea.fr/pub/dsv/anatomist/data/demo_data_td_2007.tar.gz)
- Section *Exemple data* from <http://brainvisa.info/downloadpage.html>

For more information concerning the installation, please refer to [the handbook of BrainVISA](#).

In order to read the help of one commande line, either tape `name_commmande -h` or refer to the list of commande lines on <http://brainvisa.info>.

---

# Basic command lines

## AimsFileConvert: Performs file format and data conversion

**HELP:** [command help for AimsFileConvert](#)

**DATA:** data\_for\_anatomist/objects/3d/gis/subject01.ima.

**EXAMPLE:** here is the conversion from GIS to ANALYZE format.

```
prompt% AimsFileConvert -i subject01.ima -o subject01.img
```

**NOTE:** if you work with dicom files, all slices of the same acquisition must be located in the same directory.

## AimsSubVolume: Carve a subvolume in the input volume

**HELP:** [command help for AimsSubVolume](#)

**DATA:** data\_for\_aims/AimsSubVolume/diff\_data.ima, diffusion volume of size [256, 256, 23, 22].

**EXAMPLE 1: GET T2 VOLUME FROM DIFFUSION DATA.** In order to get the first volume of the 4th dimension, which corresponds to a T2 volume, you can use the AimsSubVolume command in this way:

```
prompt%
prompt% AimsSubVolume -i diff_data.ima -o t2.ima -t 0 -T 0
Input volume dimensions : 256 256 23 22
Output volume dimensions : 256 256 23 1
prompt%
```

**NOTE:** concerning a volume of size [x, y, z, 3] has 3 time step, indices start at 0, so the time index must have a value between 0 and 2. This information can be read in the header file or into an anatomist browser.

**EXAMPLE 2: GET THE FIRST 3 DIFFUSION VOLUMES FROM THE DIFFUSION DATA.** In order to get the first 3 volumes of diffusion data only:

```
prompt% AimsSubVolume -i diff_data.ima -o voll.ima vol2.ima vol3.ima -t 1 2 3 -T 1 2 3
Input volume dimensions : 256 256 23 22
Output volume dimensions : 256 256 23 1
Input volume dimensions : 256 256 23 22
Output volume dimensions : 256 256 23 1
Input volume dimensions : 256 256 23 22
Output volume dimensions : 256 256 23 1
prompt%
```

**EXAMPLE 3: GET ALL DIFFUSION VOLUMES FROM DIFFUSION DATA.** In order to remove the T2 data from diffusion data and to keep only the diffusion data:

```
prompt%% AimsSubVolume -i diff_data.ima -o diff.ima -t 1 -T 21
Input volume dimensions : 256 256 23 22
Output volume dimensions : 256 256 23 21
prompt%
```

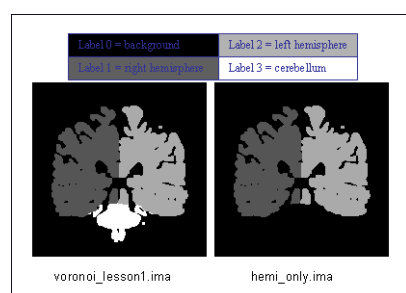
## AimsThreshold: Threshold on data

**HELP:** [command help for AimsThreshold](#)

**DATA:** data\_for\_aims/AimsThreshold/voronoi\_subject01.ima.

**EXAMPLE: select a label.** For instance, your image is a label volume with 4 values: label 0 = background, label 1 = one hemisphere, label 2 = second hemisphere and label 3 = cerebellum. If you want to remove the cerebellum, you can set up a threshold to keep all values lower than 3:

```
prompt% AimsThreshold -i voronoi_lesson1.ima -o hemi_only.ima -m lt -t 3
```



**Figure 1. Select label**

## AimsGraphMesh: Performs graph storage conversion and sub-buckets meshing. This command is an improved version of AimsGraphConvert

**HELP:** [command help for AimsGraphMesh](#)

**DATA:** data\_for\_anatomist/objects/roi/basal\_ganglia.arg and  
data\_for\_anatomist/objects/roi/basal\_ganglia.data.

**EXAMPLE: mesh a ROI graph.** The viewing will be enhanced if the ROI graph is meshed.

```
prompt% AimsGraphMesh -i basal_ganglia.arg -o mesh_basal_ganglia.arg
Warning: wrong filename_base in graph, trying to fix it
filename_base : mesh_basal_ganglia.data
bound : (121 ,127 ,66)
reading slice : 67
getting interface : done
processing mesh : done
clearing interface : done
bound : (151 ,153 ,66)
reading slice : 67
getting interface : done
processing mesh : done
clearing interface : done
bound : (153 ,137 ,71)
reading slice : 72
getting interface : done
processing mesh : done
```

```
clearing interface : done
....
saving all
```

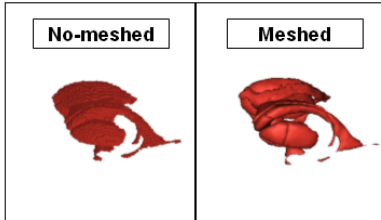


Figure 2. Viewing of non-meshed and meshed ROI

## AimsRoiFeatures: Compute scalar features (mean, volume ...) from regions of interest.

**HELP:** [command help for AimsRoiFeatures](#)

**DATA:** data\_for\_anatomist/objects/roi/anat\_demo\_roi.ima and  
AimsRoiFeatures/masque\_thalamus\_gauche.ima

Here is an example using a binary mask (so all voxels are set to 1, in other words there is a label called 1) and a volume:

```
prompt% AimsRoiFeatures -i masque_thalamus_gauche.ima --imageStatistics
1:anat_demo_roi.ima -o roi_features.txt
prompt% more features.txt
attributes = {
'format': 'features_1.0',
'content_type': 'roi_features',
'1': {
'point_count': 6502,
'volume': 6857.58,
'1': {
'mean': 48.797,
'stddev': 8.32696,
'min': 21.0003,
'max': 69.9999,
'median': 50,
},
},
},
}
```

---

# Conversion

## AimsDiffusionBundleToRoi: conversion from bundles.bundles to ROI graph

**HELP:** [command help for AimsDiffusionBundleToRoi](#)

**DATA:** no data.

**EXAMPLE:**

```
prompt% AimsDiffusionBundleAnalysis -i my_bundles.bundles -g my_bundles.arg
```

## AimsGraphConvert: conversion from label image to ROI graph

**HELP:** [command help for AimsGraphConvert](#)

**DATA:** no data.

**EXAMPLE 1:**

```
prompt% AimsGraphConvert -i label_image.ima -o label_graphe.arg --bucket
```

**EXAMPLE 2:** Mesh the graph.

```
prompt% AimsGraphMesh -i label_graphe.arg -o m_label_graphe.arg
```

## Table of format conversions

Here are some very useful command lines to convert data. However, all command options are not explained in details. Please refer to the command help.

Input (format/type)	Output (format/type)	commande line	Note
GIS	MINC	prompt% AimsFileConvert my_volume.ima my_volume.mnc	
.bundles	.arg	prompt% AimsDiffusionBundleAnalysis -i my_bundles.bundles -g my_bundles.arg	
Label_image (volume)	.arg	prompt% AimsGraphConvert -i label_image.ima -o label_graphe.arg --bucket	
.arg	Label_image (volume)	prompt%	You will find roi_Volume.ima

Input (format/type)	Output (format/type)	commande line	Note
		AimsGraphConvert -i roi.arg -o roi.arg --volume	in roi.data directory
Volume	Cluster graph	prompt% AimsClusterArg -i volume.ima -o cluster.arg	
Volume	Mesh	prompt% AimsMesh -i label_image.ima -o label_image.mesh	
Mesh	Ascii	prompt% AimsMesh2Ascii input.mesh mesh.txt	
Binary image	Mesh	prompt% AimsMesh -i binary_mask.ima -o mask.mesh	The output file will be mask_1_0.mesh

**Table 1. Table of format conversions**

---

# Calculation of images

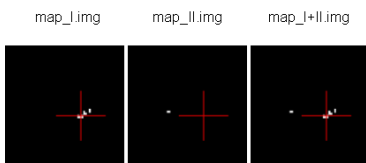
## AimsLinearComb: sum 2 activation maps

**HELP:** [command help for AimsLinearComb](#)

**DATA:** no data.

**EXAMPLE: Sum of 2 activation maps.** For instance, if you have 2 binary activation maps which have been obtained by functional analysis, and if you want to do a fusion of both, then you can create a new volume which will be the sum of map\_I and map\_II.

```
prompt% AimsLinearComb -i map_I.img -j map_II.img -o map_I+II.img
options parsed
i1      : map_I.img
a       : 1
b       : 1
c       : 1
d       : 1
e       : 0
i2      : map_II.img
fileout : map_I+II.img
type    :
Reading image map_I.img...
done
reading image map_II.img...
done
processing...
done
writing result...
done
```



**Figure 1. Sum of 2 activation maps**

**NOTE:** image dimensions must be the same.

**NOTE:** you can use this command line to add several volumes; first add up map\_I and map\_II to create map\_I+II and if you have a third volume, map\_III to fusion with map\_I and map\_II, then you can use AimsLinearComb with map\_I+II and map\_III.

**NOTE:** advanced use, you can use multiplicative and divisor coefficients for each volume.

---

# Handling meshes

## Creation of a cube mesh from a point list

**HELP:** [command help for AimsConvexHull](#)

**DATA:** no data.

**EXAMPLE:**

- Write a the following text file and save under cube.txt:

```
BEGIN-----CUT HERE-----BEGIN
8
0 0 0
10 0 0
0 10 0
10 10 0
0 0 10
10 0 10
0 10 10
10 10 10
END-----CUT HERE-----END
```

- ```
prompt% AimsConvexHull -i cube.txt -o cube.mesh
```

## AimsZCat: concatenates volumes (along Z axis), meshes or buckets

**HELP:** [command help for AimsZCat](#)

**DATA:**

- `database_brainvisa/demo/subject01/tri/subject01_Lhemi.mesh`
- `database_brainvisa/demo/subject01/tri/subject01_Rhemi.mesh`

**EXAMPLE: concatenation both right and left hemisphere meshes**

```
prompt% AimsZCat -i subject01_Lhemi.mesh subject01_Rhemi.mesh -o
right_and_left_hemisphere.mesh
```

---

# Handling referentials and transformations

## Coordinates systems in AIMS

Here is a description of the coordinates systems used in Aims and Anatomist, and what I have understood of how SPM handles its referentials.

### AIMS and Anatomist

#### Internally

Anatomist uses AIMS to handle its referentials so behaves exactly the same way.

Aims tries to work internally in an image-specific referential, but with always the same orientation. This orientation is axial with the following coordinates system:

- X axis: right to left
- Y axis: front to back
- Z axis: top to bottom
- origin: the center of the *first* voxel: the voxel in the top, right, front corner

If you look at it you will realize that this referential is in *radiological* convention and is *indirect*. This is, in my opinion, a bad choice, but it's a bit too late to change.

Once loaded in memory, all voxels should be organized in this order. As a consequence, images in Anatomist are always displayed in radiological mode, whatever the actual orientation of data on disk.

#### Externally

Images on disk, depending on their format and acquisition modes, are not necessarily in this orientation. When a different orientation is detected, images are flipped in memory at load-time to fit the standard AIMS orientation. And when images are written back to disk, they may also be flipped back according to the specific format needs.

### Transformations

By default, AIMS doesn't apply any transformation other than flipping images at load time as described just before.

But transformations can be provided in some Aims commands or loaded in Anatomist to apply coordinates changes. Then coords transformations are applied on the fly when processing or displaying data which are not in the same referential.

There is no special referential (such as a common central working referential).

Transformation files used by AIMS (.trm files) are ASCII files looking like this:

```
Tx Ty Tz
R11 R12 R13
R21 R22 R23
R31 R32 R33
```

Tx, Ty, Tz are the translation components while the Rij coefficients are the linear matrix part. When used, these coefficients are applied as a "standard" 4x4 transformation matrix:

$$M = \begin{bmatrix} R11 & R12 & R13 & Tx \\ R21 & R22 & R23 & Ty \end{bmatrix}$$

```
[ R31  R32  R33  Tz ]
[  0   0   0   1 ]
```

## MINF files

AIMS (and Anatomist) writes an additional header file which can store any additional information: the `.minf` header (for Meta-INformation) when saving its data (images, meshes, and any other data), and reads it if it is present when loading data files. This meta-header has the shape displayed by the `AimsFileInfo` command, and may be saved in "python dictionary" or XML formats. The MINF file has the same file name as the main data file, with the `.minf` extension added (`toto.img.minf` for instance).

The MINF header may contain referentials and transformations information. When present, this information is stored in a few fields:

- **referential** may store a unique identifier (a cryptic characters string) to identify the AIMS referential for the current data file. If several data files refer to the same identifier, then they share the same referential and are considered to have coordinates in the same system.
- **referentials** may store a list of target referentials for transformations specified in the `transformations` field. Both fields must have the same number of entries. Referentials are identified by character strings, either as unique identifiers or generic names (not necessarily unique). Some standard common referentials have specific names: "Talairach-MNI template-SPM" for the MNI normalization referential (used by SPM for instance), or "Talairach-AC/PC-Anatomist" for the referential based on anterior and posterior commissures used by the BrainVISA anatomical segmentation pipeline.
- **transformations** may store a list of transformation matrices, each going from the AIMS data referential to the corresponding referential specified in the `referentials` field (same position in the list). Each transformation is a 4x4 matrix written as 16 numbers in rows, and assumes all coordinates are in millimeters.
- **storage\_to\_memory** may store the disk orientation information, by providing the transformation between the disk storage voxels order and the memory orientation (the AIMS referential). This transformation is in the same shape as the `transformations` field, except that it is not a list, and the transformation is in voxels, not in mm.

For instance a MINF file may look like the following (in "python dictionary" format, here):

```
attributes = {
  'storage_to_memory' : [ 1, 0, 0, 0, 0, -1, 0, 62, 0, 0, -1, 45, 0, 0, 0, 1 ],
  'referentials' : [ 'Coordinates aligned to another file or to anatomical truth' ],
  'transformations' : [ [ -1, 0, 0, 78, 0, -1, 0, 75, 0, 0, -1, 84, 0, 0, 0, 1 ] ],
  'referential': 'be9724cc-eceb-d831-a83e-335e12b80f14',
}
```

The referentials and transformations information in the MINF header may reflect information already stored in the specific format header (Analyze origin, or NIFTI-1 `qform` and `sform`, or MINC transformation).

## SPM

### Internally

Internally, SPM *thinks* things are always in the same orientation, which is also axial but with different axes:

- X axis: left to right
- Y axis: back to front
- Z axis: bottom to top

- origin: the center of the voxel specified by the *origin* field of the SPM image header. This origin is specified in voxels and starts counting from 1 (not 0) like a matlab array index does.

This is a *neurological* convention orientation. The axes happen to be exactly the contrary of what is done in AIMS. Bad luck... But this referential is direct so cannot be considered worse than in AIMS...

Working on the coordinate transformations for years and regularly getting headaches from it, I am still not 100% sure of what I say here, so if I'm wrong, please correct me by sending a message on BrainVisa forum (<http://brainvisa.info/forum/>). Especially, I'm not sure that SPM99 and SPM2 really use the same referentials.

### Externally

SPM handles input Analyze images in two different orientations: axial radiological and axial neurological orientations. This orientation is **not specified** in SPM-Analyze format image files, so **you** have to tell how they are oriented. This is done in SPM by a flipping flag set somewhere in SPM defaults configuration (`default.analyze.flip` in SPM2).

This is specific to SPM-Analyze format, and does not apply to NIFTI-1 or Minc formats. Hopefully the Analyze format is now obsolete and will disappear with time, but there are still existing files...

This flipping flag has changed in form and meaning between SPM99 and SPM2.

As I have understood:

- **SPM99:**

- SPM99 uses the flipping flag only when normalizing images, indicating that the normalization process must perform or not a flip towards the normalization template. A clear indication of it is that the flag is part of the normalization parameters and is not present in other parts of SPM.
- Otherwise, SPM99 does not bother about the orientation of images. This is to say: even when displaying images, radiological images will be displayed with the left on the right of the display window, and neurological images with the left on the left, regardless of the flipping flag.
- Normalized images are **always** in neurological orientation whatever the orientation of input unnormalized images. Consequently, after normalizing a radiological image, loading both a normalized image and an unnormalized one in SPM will display them with different orientations.
- I am not sure if normalization templates have to be necessarily in neurological orientation or not but I guess yes because there is no way to indicate that the template is in radiological orientation.
- Normalization matrices for radiological data contain a X axis flip (negative 1st coefficient)

- **SPM2:**

- SPM2 uses the flipping flag at load time: radiological images are systematically flipped when loaded (and flipped back when rewritten so as to keep their radiological orientation on disk). This is true for **all Analyze/SPM** image files.
- This means all processings use it. As a consequence, all images are displayed in neurological orientation, left on the left, even for radiological images.
- But as this flag is global in SPM, *all* SPM images are considered to be in the same orientation: you cannot mix radio and neuro images. What I am pointing out here is only valid for SPM format images: SPM2 also handles MINC format, and Minc images contain orientation information.
- Normalized images are now in the same orientation as the input unnormalized image. Normalizing a radiological image will result in a normalized file in the radiological orientation (in SPM format). **This is not what SPM99 used to do.**
- SPM2 does not understand SPM99 and vice versa: no compatibility at all (neither forward nor backward): if you are using

the radiological convention (like we are), loading in SPM2 an image normalized by SPM99 will result in a spurious flip and incorrect display and processing. This means you cannot use with SPM2 an image database built with SPM99.

- Normalization templates can be in either orientation. More precisely, I guess the template must be in the orientation specified by the flipping flag, or in Minc format in neurological orientation. I'm not completely sure of this. But this is perhaps an explanation of why the standard normalization template is now in Minc format and not in SPM format (otherwise its interpretation would depend on a user-defined flag).
- Last minute: I have just discovered that SPM now sometimes produces images with negative voxel sizes. I guess it is a kind of flipping indication, but I don't know from what to what else. And we know that all radiological images don't have this negative voxel size feature. So my opinion is that it's not reliable at all (at least unless you exactly know which version of SPM has written each image and this info is not available). This sign information is ignored in the current version of AIMS.

The headache goes on...

- **SPM5 ans SPM8:**
  - SPM5 now uses the NIFTI-1 format for all output. NIFTI-1 specifies orientations and possibly transformations to standard referentials in its format, so this is a very good thing. Many problems are now solved.
  - Otherwise I guess SPM5 behaves essentially like SPM2.
  - The only little imperfection is that when SPM5 performs normalizations towards the MNI template, it does not indicate in output image that the target referential is the MNI template, but an unspecified other referential instead. So **you** have to know the target referential and specify it when needed (for instance in Anatomist).

## Transformations

SPM uses a common central referential to work in. Every image can provide a transformation matrix to this referential. Such a transformation may be specified in different ways:

- an optional `.mat` file with the same name as the SPM format image. this was the way SPM99 and SPM2 behaved with Analyze format. But with SPM5 and SPM8, using NIFTI-1 formats avoids this need.
- If this `.mat` is not provided, then the file format header information is considered. NIFTI-1 provides full affine transformation matrices, but Analyze has only the origin translation, which is considered to be the only transformation needed to reach the central referential. If the `.mat` file is specified, information contained in it overrides some of the header information (including the origin).

Normalization files (`*_sn3d.mat` for SPM99, `*_sn.mat` for SPM2 and newer) contain transformations to the referential of a normalization template (either a standard one provided with the SPM software distribution, or a custom user-made one). This transformation contains an affine part (matrix), and optionally, depending on the normalization type, a non-linear part (coefficients on a functions base as far as I know but I don't know much about this part). Information about the input and template images are also included (dimensions, and origins or voxels-to-template transformation).

Normalized images are in the referential of the normalization template used, but not necessarily with the same bounding box, resolution and field of view.

SPM99 and SPM2 use normalization files with different names and different contents. They are not compatible, even if there is some common and similar information in them.

## Changing between SPM and AIMS

Due to the different internal orientations of the coordinate systems, going from SPM to AIMS and vice versa causes some serious

problems.

### Normalization

SPM normalization files are in matlab (.mat) format. AIMS cannot read the proprietary matlab format, so such files cannot be directly imported in AIMS.

However, the scipy module for Python language can read them. So we have made Python scripts in PyAims and in BrainVisa to convert SPM matrices to AIMS .trm format. **Only the affine part can be converted**, because AIMS only use matrices for transformations, and non-linear information cannot fit into a matrix. Look at the `AimsSpmNormalizationConvert.py` program, and the `SPMsn3dToAims` process in BrainVISA.

As the orientation is different in SPM and AIMS, a transformation to a template image is not the same as a transformation to a normalized image with a different field of view. So, when converting SPM normalization matrices, the normalized image must be also provided, otherwise BrainVisa can only give the transformation to the normalization template. Note the difference.

## Issues

### Unnormalized SPM/Analyze images

It is impossible to guess the orientation of such images if you don't know how they were acquired. This means you have to manually specify their orientation, either for all images in SPM, or in BrainVisa when importing them into a database. BrainVisa tags them so it knows everything afterwards and avoids mistakes. SPM does not.

### Normalized SPM/Analyze images

Normalizing the same image in radiological orientation with SPM99 and SPM2 results in normalized images in different orientations. When you import normalized images coming from another site, you have to know whether they have been normalized by SPM99 or by SPM2, and if the original image was in radiological or neurological orientation.

I think the normalization file (\*\_sn3d.mat for SPM99, or \*\_sn.mat for SPM2) contains enough information to retrieve the orientation of input and template images, so can disambiguate the situation.

### Reading SPM/Analyze origin

The origin field of SPM format is the position of the referential origin, in voxels and starting from 1 (not from 0). In fact it's a matlab array index. So it is given in the orientation of the image on disk. AIMS flips SPM images on several axes when loading them, so the origin information also has to be flipped. Flipping it needs to know the image dimensions.

AIMS referentials have their origin in the first voxel, (almost) in the corner of the image, and normally don't use the SPM origin. But the origin information is read and maintained. Anatomist can, if asked for, make a transformation going from AIMS origin (corner) to the SPM origin. This allows to display several aligned SPM images in Anatomist with the correct correspondance. However after this translation, the coordinates are still in AIMS orientation (radiological and indirect), not in SPM, so the coordinates do not correspond to what they are in SPM.

To compare coordinates of SPM images in Anatomist and SPM, another transformation has to be applied in Anatomist, with all the flips included. Anatomist can directly use the SPM/MNI normalization referential.

### Other formats (GIS etc)

Up to now, GIS images are considered being always in AIMS orientation unless specified in their AIMS meta-header (.minf file, see [the corresponding paragraph](#)). No flips are applied.

The Minc and NIFTI-1 IO plugins take orientation into account and flip data accordingly when reading / writing files.

I am not sure if other formats (Dicom, Ecat...) can specify an image orientation or not. If they do, the current release of AIMS will probably not take it into account.

## Technical details

### SPM normalization matrices conversion to AIMS world

SPM99 and SPM2 don't use the same format of normalization files, but both provide more or less the following information:

- An affine transformation matrix, called `Affine`, transforming coordinates from the template space to the input space, both sides in voxels arrays index, and indexed from 1 (not from 0)
- A voxels-to-mm transformation matrix for the input image, transforming voxels of the image into a mm position in the SPM internal orientation, taking the origin into account, and possibly rotations if the format supports it (NIFTI). This matrix is called `VF.mat` in SPM2 and also performs flipping, and called `MF` in SPM99 but doesn't seem to contain the flipping information. However for nomrrialization this millimetric referential is quite undefined and we will not really use it.
- Another voxel-to-mm matrix for the template image: `VG.mat` in SPM2, or `MG` in SPM99.
- Input and template image dimensions in voxels and a bit more: `VF.dim` and `VG.dim` in SPM2, or `Dims` in SPM99.

**Notations:**

- 3 images: input (I call it Anatomy to be clearer), template, and normalized images. I use the suffixes A, T and N for coordinates on these 3 images.
- I use the same name for a given referential and coordinates in this referential: for instance RAA is both the AIMS referential of the anatomical image and a coordinates vector in it. I don't bother about standardized math notations: I don't remember them and haven't been using math anymore for many years. Don't ask me too much.
- AIMS referentials:
  - RAA: anatomy (in mm, radio convention, origin in 1st voxel)
  - RAAv: anatomy (in voxels, radio convention, origin in 1st voxel)
  - RAA<sub>d</sub>: anatomy (in voxels, disk storage ordering)
  - RAN: normalized (in mm...)
- SPM referentials:
  - RSA: anatomy, in voxels
  - RST: template, in voxels
  - RSCT: template, "central" in mm
  - RSCN: normalized, "central" in mm. Actually, RSCT and RSCN are the same.
- Transformation matrices:
  - `Affine`: the SPM affine matrix (voxels): RST to RSA
  - `At`: SPM voxels to AIMS voxels transformation. This is only to take the array indexing starting at 1 in Matlab. So `At` is a  $(-1, -1, -1)$  voxel translation. It can be used between RSA and RAA<sub>d</sub>, and either between RSN and RAN<sub>d</sub>.
  - AIMS: RAA to RAN, what we want to calculate. Here again, I'm maybe not using correctly math notations. I mean:  $RAN = AIMS * RAA$ .
  - `VsA`: Aims voxels to mm anat
  - `S2MA Aims "storage_to_memory"` anat matrix: disk voxels to Aims voxels.

- A2T: Aims anat-mm to template space-mm: RAA to RSCT, what we want to calculate if no normalized image is used.
- TN: normalized, Aims-mm to SPM-central-mm: RAN to RSCN
- TCN: template to normalized in SPM-mm: RSCT to RSCN. This transformation is identity in fact because the template and normalized images are in the same referential internally in SPM, but it's maybe clearer if I mention it.

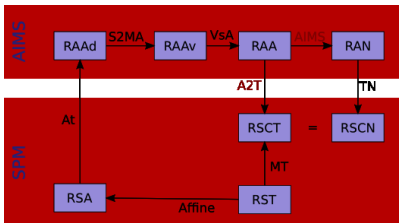


Figure 1. Referentials and normalization transformations

**Resolution:**

We want first A2T, then AIMS, provided Affine, S2MA and MT

$$A2T = MT * ( VsA * S2MA * At * Affine )^{-1}$$

$$AIMS = TN^{-1} * A2T$$

## Compute the inverse transformation

**HELP:** [command help for AimsInvertTransformation](#)

**DATA:** no data.

```
prompt% AimsInvertTransformation -i R1_TO_R2.trm -o R2_TO_R1.trm
```

## Compose a transformation

**HELP:** [command help for AimsComposeTransformation](#)

**DATA:** no data.

Let's imagine you have 3 referentials: R1, R2 and R3. You know R1->R2 (R1\_TO\_R2.trm) and R2->R3 (R2\_TO\_R3.trm). To compute R1->R3:

```
prompt% AimsComposeTransformation -i R2_TO_R3.trm R1_TO_R2.trm -o R1_TO_R3.trm
```

NOTE\_1: be aware, the order of transformation matrices is very important, this one is right *-i R2\_TO\_R3.trm R1\_TO\_R2.trm* but the following is completely wrong *-i R1\_TO\_R2.trm R1\_TO\_R3.trm*.

NOTE\_2: if you have R3\_TO\_R2.trm and not R2\_TO\_R3.trm, you must first inverse this transformation matrix by using [AimsInvertTransformation](#) .

---

# Handling graphs

## Copy a set of graph attributes to another graph

**HELP:** [command help for AimsGraphConvert](#)

**DATA:** no data.

This case generally happens when working on automatically labelled sulci graphs. The nodes labels are given as the *label* attribute (automatic recognition labels), and you sometimes need to copy them to the *name* attribute (manual labelling). You have 2 possibilities to do it: manually or automatically.

**MANUALLY:** you can verify each value of *label* attribute and correct it if necessary. To do so, change the value of the *name* attribute in a browser window (using a right-click on a graph node), and save the graph as a new graph (right-click on the graph in Anatomist control window and select *File => Save*).

**AUTOMATICALLY:** you can use the *AimsGraphConvert* commandline. The following example shows how to use it:

```
prompt% AimsGraphConvert -i subjectAuto.arg -o subjectAutoName.arg -c label -d name
```

This command has many other options, but for the current application, the useful ones are:

- *-i option*: input graph, for instance an autolabelled.
- *-o option*: output file name.
- *-c option*: attribute to be copied.
- *-d option*: destination attribute.

---

# Rigid registration

## AimsManualRegistration: manual registration between 2 volumes from 3 specific landmarks.

**KEYWORDS:** .trm file.

**HELP:** [command help for AimsManualRegistration](#)

**DATA:** no data.

### EXAMPLE: MANUAL REGISTRATION BETWEEN 2 VOLUMES FROM 3 SPECIFIC LANDMARKS.

This example is based on 2 volumes (i.e. registration from image\_1 to image2) using specific points (i.e. anatomical landmarks). By using the AimsManualRegistration command line, you will obtain a transformation matrix from image\_1 to image\_2 as a .trm file.

- For each volume, [draw a ROI](#) with exactly 3 regions (each region must contain only 1 voxel) with the same name per region (ie voxel\_1, voxel\_2, and voxel\_3). So, you have the following ROIs: ROI\_image\_1.arg and ROI\_image\_2.arg with the structure of each ROI is composed by the regions: voxel\_1, voxel\_2 and voxel\_3.
- The command line is as follows:

```
prompt% AimsManualRegistration -f ROI_image_1.arg -t ROI_image_2.arg -o  
ROI_image_1_TO_ROI_image_2.trm
```

**NOTE:** [to load a transformation, please refer to \*The Anatomist getting started guide\*](#).

## AimsMIRegister: registration based on mutual information.

**KEYWORDS:** .trm file.

**HELP:** [command help for AimsMIRegister](#)

**DATA:**

- data\_for\_aims/Registration/anat.img
- data\_for\_aims/Registration/fonc.ima

This command can appear complex because a lot of options are available. In this section, we are going to try to define a reasonable use. The easier use is the following:

```
prompt% AimsMIRegister -r anat.img -t fonc.ima --dir fonc_TO_anat.trm --inv  
anat_TO_fonc.trm
```

Here are some options to optimize the registration. It is not advisable to use the other because the implementation is not totally finished.

**Initialization of registration: --gcinit, --seuilref and --seuiltest.** There are 2 modes for initialization of registration. The **--gcinit 1** mode (default mode) allows an initialization with the center of gravity. It works with **--seuilref** (0.05 by default) and **--seuiltest** (0.1 by default) options. These thresholds preserve a percentage of intensity according to the maximum intensity. Historically, this

command was created to register PET and T1 RMI modalities, so to decrease the signal of PET data, a threshold was performed. And the **--gcinit 0** mode allows an initialization with coordinates by using of **--Tx, --Ty, --Tz, --dTx, --dTy, --dTz, --Rx, --Ry, --Rz, --dRx, --dRy, --dRz**. Parameters beginning with a **d** corresponds to the exploration step, which is voxel=2 by default.

**Speeding up the process**, reference image can be damaged with a reduction factor according to the principle of a pyramid with **--refstartpyr**:

- **--refstartpyr 1**: 1 voxel for 2 in the 3 directions = reduction of factor 8
- **--refstartpyr 2**: 1 voxel for 4 in the 3 directions = reduction of factor 64
- **--refstartpyr 3**: 1 voxel for 8 in the 3 directions = reduction of factor 512

---

# Advanced level

## Get a symmetrical ROI

### HELP:

- [command help for AimsMidPlaneAlign](#)
- [command help for AimsLinearComb](#)
- [command help for AimsResample](#)
- [command help for AimsThreshold](#)
- [command help for AimsFlip](#)

**DATA:** no data.

The purpose of this section is for instance to compare the ROI measurement for both hemispheres following the realignment of the brain by using a symmetric axe:

- [Draw a ROI](#) on the T1 MRI and export it as a mask to work with a .ima file (image) and not a .arg (graph): roi.ima
- Use the AimsMidPlaneAlign command line to realign the image and compute the transformation (superposition of the interhemispheric plane with the plane  $x=\text{dimX}/2$ ).

```
prompt% AimsMidPlaneAlign -i rmiT1.ima -o align_rmiT1.ima
```

NOTE: the transformation matrix is located in the input file directory with the following name rmiT1.ima\_TO\_align.ima.trm.

- Do a linear combination if the ROI is a binary image :

```
prompt% AimsLinearComb -i roi.ima -o linearComb_roi.ima -a 16000
```

NOTE: please refer to the table in NOTE\_2 if the image is not binary.

- Resample the ROI with the previously calculated transformation:

```
prompt% AimsResample -i linearComb_roi.ima -o resample_roi.ima -m  
rmiT1.ima_TO_align.ima.trm
```

- Perform a threshold to 8000 to preserve a correct volume because the resampling widely extend the symmetric roi volume:

```
prompt% AimsThreshold -i resample_roi.ima -o threshold_roi.ima -m ge -t 8000
```

- Get the symmetrical ROI by using AimsFlip as follows:

```
prompt% AimsFlip -i threshold_roi.ima -o sym_roi.ima -m XX
```

- Each ROI can be in both referentials which are T1 and T1\_align. In order to change the coordinate system, you apply a .trm. For instance, if you want the ROIs in T1 referential, you must resample the sym\_roi.ima with the inverse of

T1MRI.ima\_TO\_align.ima.

```
prompt% AimsInvertTransformation -i rmiT1.ima_TO_align.ima.trm -o
align.ima_TO_rmiT1.ima.trm
```

Then, resample the sym\_roi.ima:

```
prompt% AimsResample -i sym_roi.ima -o sym_roi_RT1.ima -m align.ima_TO_rmiT1.ima.trm
```

- Analyze/compare the ROIs by using [AimsRoiFeatures](#).

NOTE\_1: be aware that the procedure presented below is not formal. In fact, many variations can be processed, the modality (PET, CT ...), how the ROI is obtained (draw on the original referential, or after the realignment) or where it comes from (i.e. created by an other process), what is the type of ROI value (binary, label image ...).

NOTE\_2: here is a summary to help you compute and/or do a threshold of your ROI to preserve a correct volume (the resampling leads to volume changes):

| Max value            | commande line                                                                                                                                                                                    |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Binary image (max=1) | <pre>prompt% AimsLinearComb -i roi.ima -o roi.ima -a 16000 prompt% AimsResample -i roi.ima -o roi.ima -m motion.trm prompt% AimsThreshold -i roi.ima -o roi.ima -m ge -t 8000</pre>              |
| Max=max_value        | <pre>prompt% AimsLinearComb -i roi.ima -o roi.ima -a 16000 -b max_value prompt% AimsResample -i roi.ima -o roi.ima -m motion.trm prompt% AimsThreshold -i roi.ima -o roi.ima -m ge -t 8000</pre> |

**Table 1. Summary to preserve the ROI volume**

NOTE\_3: for more information on the algorithm used by AimsMidPlaneAlign, please refer to *Prima S, Ourselin S, and Ayache N. Computation of the mid-sagittal plane in 3-D brain images. IEEE Trans Med Imaging. 2002 Feb;21(2):122-38.*

---

# Programming with AIMS in Python language

AIMS is a C++ library, but has python language bindings: **PyAIMS**. This means that the C++ classes and functions can be used from python. This has many advantages compared to pure C++:

- Writing python scripts and programs is much easier and faster than C++: there is no fastidious and long compilation step.
- Scripts are more flexible, can be modified on-the-fly, etc
- It can be used interactively in a python interactive shell.
- As pyaims is actually C++ code called from python, it is still fast to execute complex algorithms. There is obviously an overhead to call C++ from python, but once in the C++ layer, it is C++ execution speed.

A few examples of how to use and manipulate the main data structures will be shown here.

The data for the examples in this section can be downloaded here: [ftp://ftp.cea.fr/pub/dsv/anatomist/data/demo\\_data.zip](ftp://ftp.cea.fr/pub/dsv/anatomist/data/demo_data.zip). To use the examples directly, users should go to the directory where this archive was uncompressed, and then run `ipython` from this directory. A cleaner alternative, especially if no write access is allowed on this data directory, is to make a symbolic link to the "data\_for\_anatomist" subdirectory:

```
cd $HOME
mkdir bvcourse
cd bvcourse
ln -s <path_to_data>/data_for_anatomist .
ipython
```

More information about the API is available [here](#).

## Using data structures

### Module importation

In python, the `aimsdata` library is available as the `soma.aims` module.

```
import soma.aims
# the module is actually soma.aims:
vol = soma.aims.Volume( 100, 100, 100, dtype='int16' )
```

or:

```
from soma import aims
# the module is available as aims (not soma.aims):
vol = aims.Volume( 100, 100, 100, dtype='int16' )
# in the following, we will be using this form because it is shorter.
```

### IO: reading and writing objects

Reading operations are accessed via a single `read()` function, and writing through a single `write()` function. The `read()` function reads any object from a given file name, in any supported file format, and returns it:

```
from soma import aims
obj = aims.read( 'data_for_anatomist/subject01/subject01.nii' )
print obj
obj2 = aims.read( 'data_for_anatomist/subject01/Audio-Video_T_map.nii' )
print obj2
```

```
obj3 = aims.read( 'data_for_anatomist/subject01/subject01_Lhemi.mesh' )  
print obj3
```

The returned object can have various types according to what is found in the disk file(s).

Writing is just as easy. The file name extension generally determines the output format. An object read from a given format can be re-written in any other supported format, provided the format can actually store the object type.

```
from soma import aims  
obj2 = aims.read( 'data_for_anatomist/subject01/Audio-Video_T_map.nii' )  
aims.write( obj2, 'Audio-Video_T_map.ima' )  
obj3 = aims.read( 'data_for_anatomist/subject01/subject01_Lhemi.mesh' )  
aims.write( obj3, 'subject01_Lhemi.gii' )
```

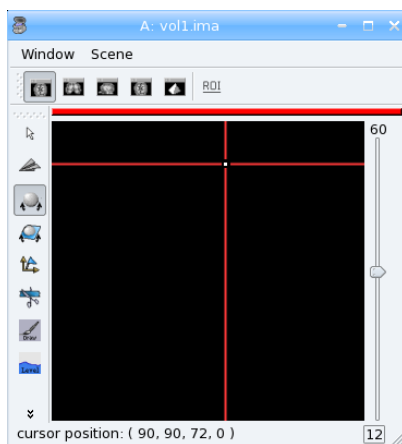
**Exercise:** write a little file format conversion tool

## Volumes

Volumes are array-like containers of voxels, plus a set of additional information kept in a header structure. In AIMS, the header structure is generic and extensible, and does not depend on a specific file format. Voxels may have various types, so a specific type of volume should be used for a specific type of voxel. The type of voxel has a code that is used to suffix the Volume type: `Volume_S16` for signed 16-bit ints, `Volume_U32` for unsigned 32-bit ints, `Volume_DOUBLE` for 32-bit floats, `Volume_RGBA` for RGBA colors, etc.

### Building a volume

```
# create a 3D volume of signed 16-bit ints, of size 192x256x128  
vol = aims.Volume( 192, 256, 128, dtype='int16' )  
# fill it with zeros  
vol.fill(0)  
# set value 12 at voxel ( 100, 100, 60 )  
vol.setValue( 12, 100, 100, 60 )  
# get value at the same position  
x = vol.value( 100, 100, 60 )  
# set the voxels size  
vol.header()[ 'voxel_size' ] = [ 0.9, 0.9, 1.2, 1. ]  
print vol.header()
```



**Figure 1.** 3D volume: value 12 at voxel (100, 100 ,60)

### Basic operations:

Whole volume operations:

```
# multiplication, addition etc
vol *= 2
vol2 = vol * 3 + 12
vol /= 2
vol3 = vol2 - vol - 12
vol4 = vol2 * vol / 6
```

Voxel-wise operations:

```
# fill the volume with the distance to voxel ( 100, 100, 60 )
vs = vol.header()[ 'voxel_size' ]
pos0 = ( 100 * vs[0], 100 * vs[1], 60 * vs[2] ) # in millimeters
for z in xrange( vol.getSizeZ() ):
    for y in xrange( vol.getSizeY() ):
        for x in xrange( vol.getSizeX() ):
            # get current position in an aims.Point3df structure, in mm
            p = aims.Point3df( x * vs[0], y * vs[1], z * vs[2] )
            # get relative position to pos0, in voxels
            p -= pos0
            # distance: norm of vector p
            dist = p.norm()
            # set it into the volume
            vol.setValue( dist, x, y, z )

# save the volume
aims.write( vol, 'distance.nii' )
```

Now look at the "distance.nii" volume in Anatomist.

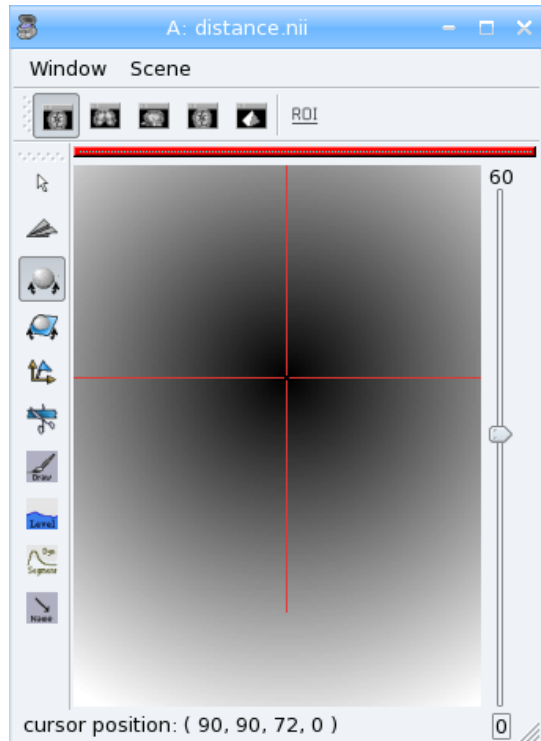


Figure 2. Distance example

**Exercise:** Make a program which loads the image data\_for\_anatomist/subject01/Audio-Video\_T\_map.nii and thresholds it so as to keep values above 3.

```
from soma import aims
vol = aims.read( 'data_for_anatomist/subject01/Audio-Video_T_map.nii' )
for z in xrange( vol.getSizeZ() ):
    for y in xrange( vol.getSizeY() ):
        for x in xrange( vol.getSizeX() ):
            if vol.value( x, y, z ) < 3.:
                vol.setValue( 0, x, y, z )

aims.write( vol, 'Audio-Video_T_thresholded.nii' )
```

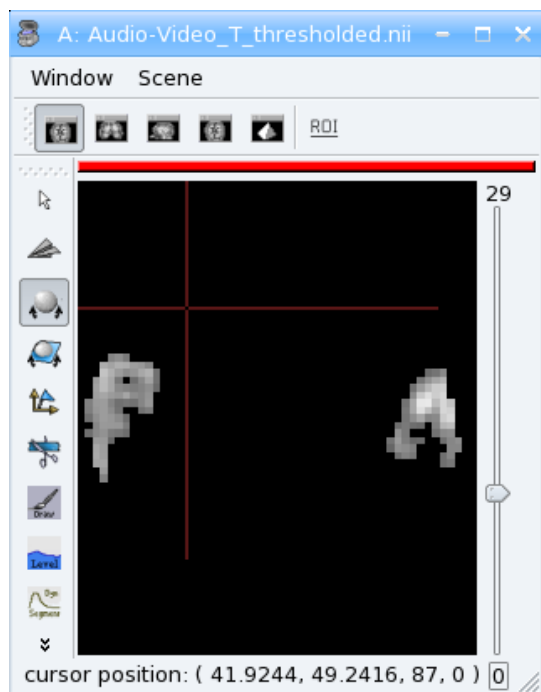


Figure 3. Thresholded Audio-Video T-map

**Exercise:** Make a program to downsample the anatomical image `data_for_anatomist/subject01/subject01.nii` and keeps one voxel out of two in every direction.

```
from soma import aims
vol = aims.read( 'data_for_anatomist/subject01/subject01.nii' )
# allocate a new volume with half dimensions
vol2 = aims.Volume( vol.getSizeX() / 2, vol.getSizeY() / 2, vol.getSizeZ() / 2,
dtype='DOUBLE' )
# set the voxel size to twice it was in vol
vs = vol.header()[ 'voxel_size' ]
vs2 = [ x * 2 for x in vs ]
vol2.header()[ 'voxel_size' ] = vs2
for z in xrange( vol2.getSizeZ() ):
    for y in xrange( vol2.getSizeY() ):
        for x in xrange( vol2.getSizeX() ):
            vol2.setValue( vol.value( x*2, y*2, z*2 ), x, y, z )

aims.write( vol2, 'resampled.nii' )
```

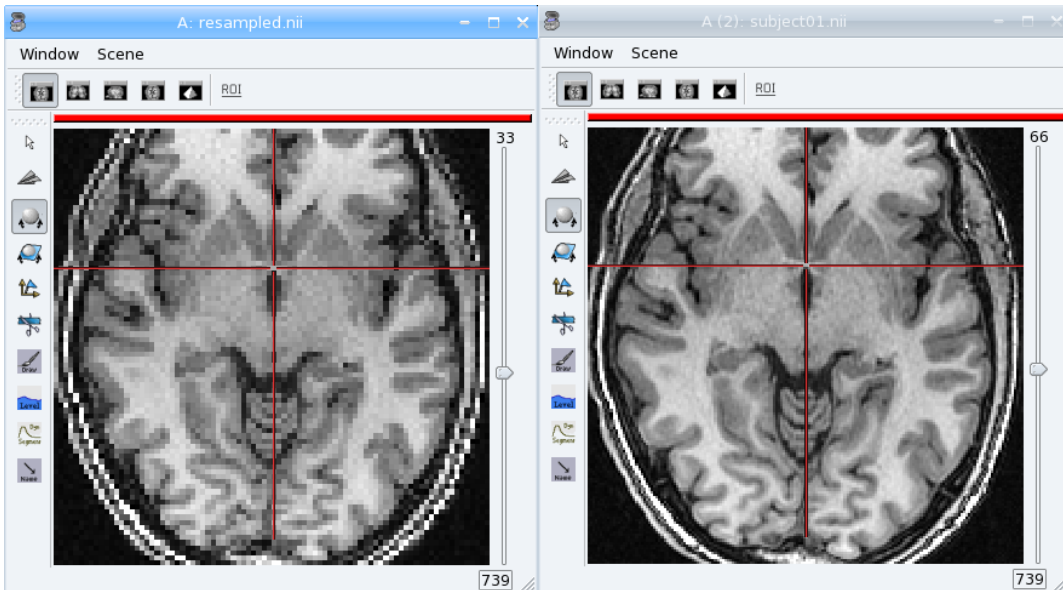
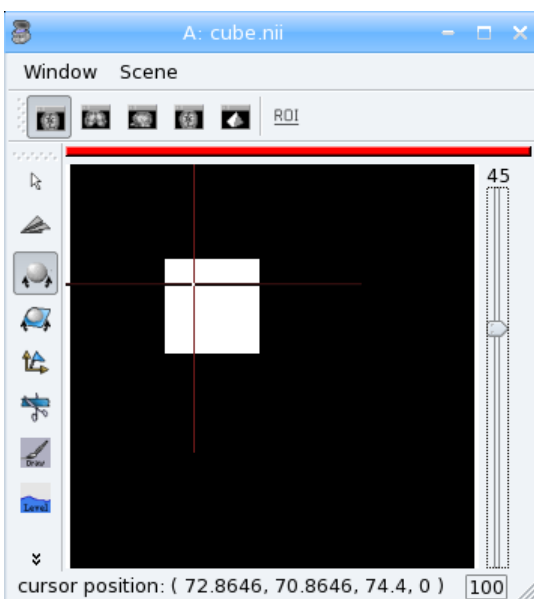


Figure 4. Downsampled anatomical image

The first thing that comes to mind when running these examples, is that they are *slow*. Indeed, python is an interpreted language and loops in any interpreted language are slow. In addition, accessing individually each voxel of the volume has the overhead of python/C++ bindings communications. The conclusion is that that kind of example is probably a bit too low-level, and should be done, when possible, by compiled libraries or specialized array-handling libraries. This is the role of **numpy**.

Accessing numpy arrays to AIMS volume voxels is supported:

```
import numpy
vol.fill( 0 )
arr = numpy.array( vol, copy=False )
# set value 100 in a whole sub-volume
arr[60:120, 60:120, 40:80] = 100
# note that arr is a shared view to the volume contents,
# modifications will also affect the volume
aims.write( vol, "cube.nii" )
```



### Figure 5. 3D volume containing a cube

Now we can re-write the thresholding example using numpy:

```
from soma import aims
vol = aims.read( 'data_for_anatomist/subject01/Audio-Video_T_map.nii' )
arr = numpy.array( vol, copy=False )
arr[ numpy.where( arr < 3. ) ] = 0.
aims.write( vol, 'Audio-Video_T_thresholded2.nii' )
```

Here, `arr < 3.` returns a boolean array with the same size as `arr`, and `numpy.where()` returns arrays of coordinates where the specified condition is true.

The distance example, using numpy, would like the following:

```
from soma import aims
import numpy
vol = aims.Volume( 192, 256, 128, 'S16' )
vol.header()[ 'voxel_size' ] = [ 0.9, 0.9, 1.2, 1. ]
vs = vol.header()[ 'voxel_size' ]
pos0 = ( 100 * vs[0], 100 * vs[1], 60 * vs[2] ) # in millimeters
arr = numpy.array( vol, copy=False )
# build arrays of coordinates for x, y, z
x, y, z = numpy.ogrid[ 0.:vol.getSizeX(), 0.:vol.getSizeY(), 0.:vol.getSizeZ() ]
# get coords in millimeters
x *= vs[0]
y *= vs[1]
z *= vs[2]
# relative to pos0
x -= pos0[0]
y -= pos0[1]
z -= pos0[2]
# get norm, using numpy arrays broadcasting
arr[:, :, :, 0] = numpy.sqrt( x**2+y**2+z**2 )
# and save result
aims.write( vol, 'distance2.nii' )
```

This example appears a bit more tricky, since we must build the coordinates arrays, but is way faster to execute, because all loops within the code are executed in compiled routines in numpy. One interesting thing to note is that this code is using the famous "array broadcasting" feature of numpy, where arrays of heterogeneous sizes can be combined, and the "missing" dimensions are extended.

### Copying volumes or volumes structure, or building from an array

To make a deep-copy of a volume, use the copy constructor:

```
vol2 = aims.Volume( vol )
vol2.setValue( 12, 100, 100, 60 )
# now vol and vol2 have different values
print 'vol.value( 100, 100, 60 ):', vol.value( 100, 100, 60 )
print 'vol2.value( 100, 100, 60 ):', vol2.value( 100, 100, 60 )
```

If you need to build another, different volume, with the same structure and size, don't forget to copy the header part:

```
vol2 = aims.Volume( vol.getSizeX(), vol.getSizeY(), vol.getSizeZ(), vol.getSizeT(),
'FLOAT' )
vol2.header().update( vol.header() )
```

Important information can reside in the header, like voxel size, or coordinates systems and geometric transformations to other

coordinates systems, so it is really very important to carry this information with duplicated or derived volumes.

You can also build a volume from a numpy array:

```
arr = numpy.array( numpy.diag( xrange( 40 ) ), dtype=numpy.float32 ).reshape( 40, 40, 1 ) \
    + numpy.array( xrange( 20 ), dtype=numpy.float32 ).reshape( 1, 1, 20 )
# WARNING: the array must be in Fortran ordering for AIMS, at least at the moment
# whereas the numpy addition always returns a C-ordered array
arr = numpy.array( arr, order='F' )
arr[ 10, 12, 3 ] = 25
vol = aims.Volume( arr )
print 'vol.value( 10, 12, 3 ):', vol.value( 10, 12, 3 )
# data are shared with arr
vol.setValue( 35, 10, 15, 2 )
print 'arr[ 10, 15, 2 ]:', arr[ 10, 15, 2 ]
arr[ 12, 15, 1 ] = 44
print 'vol.value( 12, 15, 1 ):', vol.value( 12, 15, 1 )
```

#### 4D volumes

4D volumes work just like 3D volumes. Actually all volumes are 4D in AIMS, but the last dimension is commonly of size 1. In `value()` and `setValue()` methods, only the first dimension is mandatory, others are optional and default to 0, but up to 4 coordinates may be used. In the same way, the constructor takes up to 4 dimension parameters:

```
from soma import aims
# create a 4D volume of signed 16-bit ints, of size 30x30x30x4
vol = aims.Volume( 30, 30, 30, 4, 'S16' )
# fill it with zeros
vol.fill(0)
# set value 12 at voxel ( 10, 10, 20, 2 )
vol.setValue( 12, 10, 10, 20, 2 )
# get value at the same position
x = vol.value( 10, 10, 20, 2 )
# set the voxels size
vol.header()[ 'voxel_size' ] = [ 0.9, 0.9, 1.2, 1. ]
print vol.header()
```

Similarly, 1D or 2D volumes may be used exactly the same way.

#### The older `AimsData` classes

For historical reasons, another set of classes may also represent volumes. These classes are the older API in AIMS, and tend to be obsolete. But as they were used in many many routines and programs, they have still not been eradicated. Many C++ routines build volumes and actually return those older classes, so we could not really hide them, and they also have python bindings. These classes are `aims.AimsData_<type>`. Converting from and to `aims.Volume_` classes is rather simple since the newer `Volume` classes are used internally in the `AimsData` API.

```
from soma import aims
# create a 4D volume of signed 16-bit ints, of size 30x30x30x4
vol = aims.Volume( 30, 30, 30, 4, 'S16' )
vol.header()[ 'voxel_size' ] = [ 0.9, 0.9, 1.2, 1. ]
advol = aims.AimsData( vol )
# vol and advol share the same header and voxel data
vol.setValue( 12, 10, 10, 20, 2 )
print 'advol.value( 10, 10, 20, 2 ):', advol.value( 10, 10, 20, 2 )
advol.setValue( 44, 12, 12, 24, 1 )
print 'vol.value( 12, 12, 24, 1 ):', vol.value( 12, 12, 24, 1 )
```

And, in the other direction:

```
from soma import aims
# create a 4D volume of signed 16-bit ints, of size 30x30x30x4
advol = aims.AimsData( 30, 30, 30, 4, 'S16' )
advol.header()[ 'voxel_size' ] = [ 0.9, 0.9, 1.2, 1. ]
vol = advol.volume()
# vol and advol share the same header and voxel data
vol.setValue( 12, 10, 10, 20, 2 )
print 'advol.value( 10, 10, 20, 2 ):', advol.value( 10, 10, 20, 2 )
advol.setValue( 44, 12, 12, 24, 1 )
print 'vol.value( 12, 12, 24, 1 ):', vol.value( 12, 12, 24, 1 )
```

AimsData has a bit richer API, since it includes minor processing functions that have been removed from the newer Volume for the sake of API simplicity and minimalism.

```
# minimum / maximum
print 'min:', advol.minimum(), 'at', advol.minIndex()
print 'max:', advol.maximum(), 'at', advol.maxIndex()
# clone copy
advol2 = advol.clone()
advol2.setValue( 12, 4, 8, 11, 3 )
# now advol and advol2 have different values
print 'advol.value( 4, 8, 11, 3 ):', advol.value( 4, 8, 11, 3 )
print 'advol2.value( 4, 8, 11, 3 ):', advol2.value( 4, 8, 11, 3 )
# Border handling
# Border width is th 5th parameter of AimsData constructor
advol = aims.AimsData( 192, 256, 128, 1, 2, 'S16' )
advol.header()[ 'voxel_size' ] = [ 0.9, 0.9, 1.2, 1. ]
advol.fill( 0 )
advol.setValue( 15, 100, 100, 60 )
vol = advol.volume()
# then vol is 4 voxels wider in each direction, and shifted:
print 'vol.value( 100, 100, 60 ):', vol.value( 100, 100, 60 )
# ... it is 0, not 15...
print 'vol.value( 102, 102, 62 ):', vol.value( 102, 102, 62 )
# here we get 15
# some algorithms require this border to exist, otherwise fail or crash...
from soma import aimsalgo
aimsalgo.AimsDistanceFrontPropagation( advol, 0, -1, 3, 3, 3, 10, 10 )
aims.write( advol, 'distance3.nii' )
```

## Meshes

### Structure

A surfacic mesh represents a surface, as a set of small polygons (generally triangles, but sometimes quads). It has two main components: a vector of vertices (each vertex is a 3D point, with coordinates in millimeters), and a vector of polygons: each polygon is defined by the vertices it links (3 for a triangle). It also optionally has normals (unit vectors). In our mesh structures, there is one normal for each vertex.

```
from soma import aims
mesh = aims.read( 'data_for_anatomist/subject01/subject01_Lhemi.mesh' )
vert = mesh.vertex()
print 'vertices:', len( vert )
poly = mesh.polygon()
print 'polygons:', len( poly )
norm = mesh.normal()
print 'normals:', len( norm )
```

To build a mesh, we can instantiate an object of type `aims.AimsTimeSurface_<n>`, with  $n$  being the number of vertices by

polygon. Then we can add vertices, normals and polygons to the mesh:

```
# build a flying saucer mesh
from soma import aims
import numpy
mesh = aims.AimsTimeSurface( 3 )
# a mesh has a header
mesh.header()[ 'toto' ] = 'a message in the header'
vert = mesh.vertex()
poly = mesh.polygon()
x = numpy.cos( numpy.ogrid[ 0.:20 ] * numpy.pi / 10. ) * 100
y = numpy.sin( numpy.ogrid[ 0.:20 ] * numpy.pi / 10. ) * 100
z = numpy.zeros( 20 )
c = numpy.vstack( ( x, y, z ) ).transpose()
vert.assign( [ aims.Point3df( 0., 0., -40. ), aims.Point3df( 0., 0., 40. ) ] + [
aims.Point3df( x ) for x in c ] )
pol = numpy.vstack( ( numpy.zeros( 20, dtype=numpy.int32 ), numpy.ogrid[ 3:23 ],
numpy.ogrid[ 2:22 ] ) ).transpose()
pol[ 19, 1 ] = 2
pol2 = numpy.vstack( ( numpy.ogrid[ 2:22 ], numpy.ogrid[ 3:23 ], numpy.ones( 20,
dtype=numpy.int32 ) ) ).transpose()
pol2[19,1] = 2
poly.assign( [ aims.AimsVector(x,'U32',3) for x in numpy.vstack( ( pol, pol2 ) ) ] )
# write result
aims.write( mesh, 'saucer.mesh' )
# automatically calculate normals
mesh.updateNormals()
```

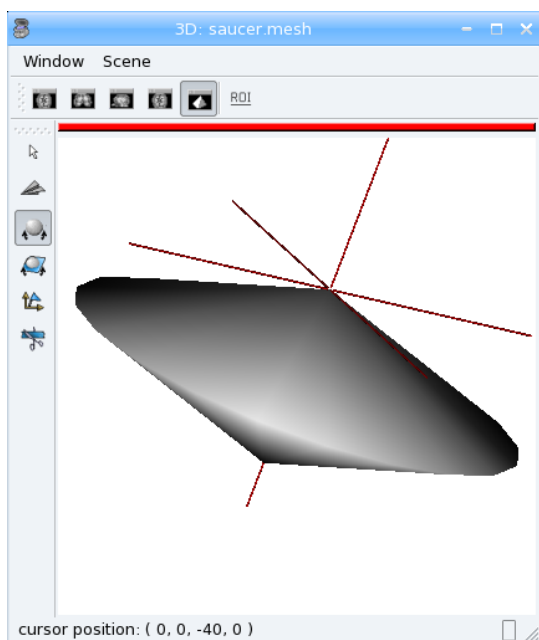


Figure 6. Flying saucer mesh

### Modifying a mesh

```
# slightly inflate a mesh
from soma import aims
import numpy
mesh = aims.read( 'data_for_anatomist/subject01/subject01_Lwhite.mesh' )
vert = mesh.vertex()
```

```
varr = numpy.array( vert )
norm = numpy.array( mesh.normal() )
varr += norm * 2 # push vertices 2mm away along normal
vert.assign( [ aims.Point3df(x) for x in varr ] )
mesh.updateNormals()
aims.write( mesh, 'subject01_Lwhite_semiinflated.mesh' )
```

Now look at both meshes in Anatomist...

Alternatively, without numpy, we could have written the code like this:

```
from soma import aims
mesh = aims.read( 'data_for_anatomist/subject01/subject01_Lwhite.mesh' )
vert = mesh.vertex()
norm = mesh.normal()
for v, n in zip( vert, norm ):
    v += n * 2

mesh.updateNormals()
aims.write( mesh, 'subject01_Lwhite_semiinflated.mesh' )
```

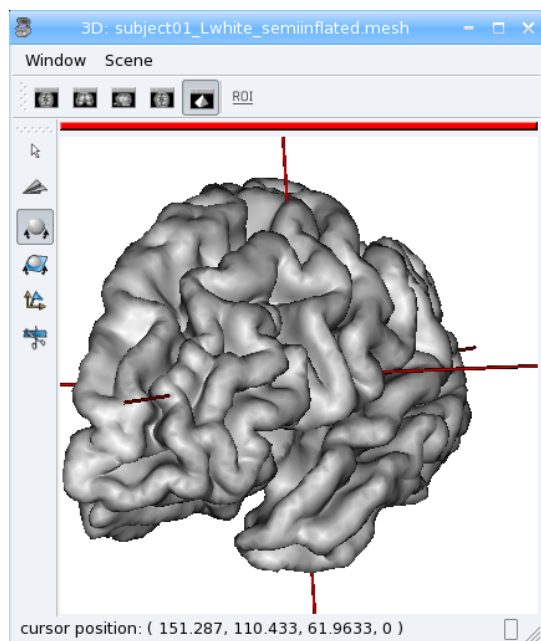


Figure 7. Inflated mesh

## Handling time

In AIMS, meshes are actually time-indexed dictionaries of meshes. This way a deforming mesh can be stored in the same object. To copy a timestep to another, use the following:

```
from soma import aims
mesh = aims.read( 'data_for_anatomist/subject01/subject01_Lwhite.mesh' )
# mesh.vertex() is equivalent to mesh.vectex( 0 )
mesh.vertex( 1 ).assign( mesh.vertex( 0 ) )
# same for normals and polygons
mesh.normal( 1 ).assign( mesh.normal( 0 ) )
mesh.polygon( 1 ).assign( mesh.polygon( 0 ) )
print 'number of time steps:', mesh.size()
```

**Exercise:** make a deforming mesh that goes from the original mesh to 5mm away, by steps of 0.5 mm

```

from soma import aims
import numpy
mesh = aims.read( 'data_for_anatomist/subject01/subject01_Lwhite.mesh' )
vert = mesh.vertex()
varr = numpy.array( vert )
norm = numpy.array( mesh.normal() )
for i in xrange( 1, 10 ):
    mesh.normal( i ).assign( mesh.normal() )
    mesh.polygon( i ).assign( mesh.polygon() )
    varr += norm * 0.5
    mesh.vertex( i ).assign( [ aims.Point3df(x) for x in varr ] )

mesh.updateNormals()
aims.write( mesh, 'subject01_Lwhite_semiinflated_time.mesh' )

```

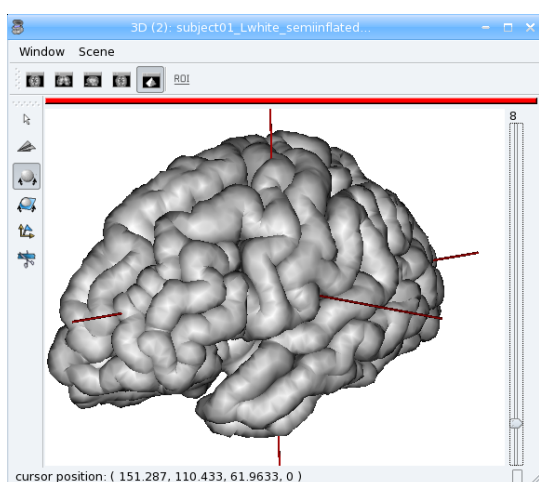


Figure 8. Inflated mesh with timesteps

## Textures

A texture is merely a vector of values, each of them is assigned to a mesh vertex, with a one-to-one mapping, in the same order. A texture is also a time-texture.

```

from soma import aims
tex = aims.TimeTexture( 'FLOAT' )
t = tex[0] # time index, inserts on-the-fly
t.reserve( 10 ) # pre-allocates memory
for i in xrange( 10 ):
    t.append( i / 10. )

```

**Exercise:** make a time-texture, with at each time/vertex of the previous mesh, sets the value of the underlying volume data\_for\_anatomist/subject01/subject01.nii

```

from soma import aims
mesh = aims.read( 'subject01_Lwhite_semiinflated_time.mesh' )
vol = aims.read( 'data_for_anatomist/subject01/subject01.nii' )
tex = aims.TimeTexture( 'FLOAT' )
vs = vol.header()[ 'voxel_size' ]
for i in xrange( mesh.size() ):
    t = tex[i]
    vert = mesh.vertex( i )
    t.reserve( len( vert ) )

```

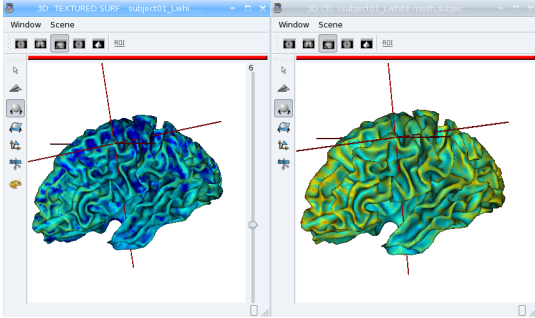
```

for p in vert:
    t.append( vol.value( *[ int( round(x/y) ) for x,y in zip( p, vs ) ] ) )

aims.write( tex, 'subject01_Lwhite_semiinflated_texture.tex' )

```

Now look at the texture on the mesh (inflated or not) in Anatomist. Compare it to a 3D fusion between the mesh and the MRI volume.



**Figure 9. Computed time-texture vs 3D fusion**

**Bonus:** We can do the same for functional data. But in this case we may have a spatial transformation to apply between anatomical data and functional data (which may have been normalized, or acquired in a different referential).

```

from soma import aims
import numpy
mesh = aims.read( 'subject01_Lwhite_semiinflated_time.mesh' )
vol = aims.read( 'data_for_anatomist/subject01/Audio-Video_T_map.nii' )
# get header info from anatomical volume
f = aims.Finder()
f.check( 'data_for_anatomist/subject01/subject01.nii' )
anathdr = f.header()
# get functional -> MNI transformation
m1 = aims.AffineTransformation3d( vol.header()[ 'transformations' ] [1] )
# get anat -> MNI transformation
m2 = aims.AffineTransformation3d( anathdr[ 'transformations' ] [1] )
# make anat -> functional transformation
anat2func = m1.inverse() * m2
# include functional voxel size to get to voxel coordinates
vs = vol.header()[ 'voxel_size' ]
mvs = aims.AffineTransformation3d( numpy.diag( vs[:3] + [ 1. ] ) )
anat2func = mvs.inverse() * anat2func
# now go as in the previous program
tex = aims.TimeTexture( 'FLOAT' )
for i in xrange( mesh.size() ):
    t = tex[i]
    vert = mesh.vertex( i )
    t.reserve( len( vert ) )
    for p in vert:
        t.append( vol.value( *[ int(round(x)) for x in anat2func.transform( p ) ] ) )

aims.write( tex, 'subject01_Lwhite_semiinflated_audio_video.tex' )

```

See how the functional data on the mesh changes across the depth of the cortex. this demonstrates the need to have a proper projection of functional data before dealing with surfacic functional processing.

## Buckets

"Buckets" are voxels lists. They are typically used to represent ROIs. A BucketMap is a list of Buckets. Each Bucket contains a list of voxels coordinates.

```
from soma import aims
bck_map=aims.read( 'data_for_anatomist/roi/basal_ganglia.data/roi_Bucket.bck' )
print 'Bucket map: ', bck_map
print 'Nb buckets: ', bck_map.size()
for i in xrange(bck_map.size()):
    b=bck_map[i]
    print "Bucket ", i, ", nb voxels: ", b.size()
    if b.keys():
        print " Coordinates of the first voxel: ", b.keys()[0].list()
```

## Graphs

Graphs are data structures that may contain various elements. They can represent sets of smaller structures, and also relations between such structures. The main usage we have for them is to represent ROIs sets, sulci, or fiber bundles.

A graph contains:

- properties of any type, like a volume or mesh header.
- nodes (also called vertices), which represent structured elements (a ROI, a sulcus part, etc), which in turn can store properties, and geometrical elements: buckets, meshes...
- optionally, relations, which link nodes and can also contain properties and geometrical elements.

## Properties

Properties are stored in a dictionary-like way. They can hold almost anything, but a restricted set of types can be saved and loaded. It is exactly the same thing as headers found in volumes, meshes, textures or buckets.

```
from soma import aims
graph = aims.read( 'data_for_anatomist/roi/basal_ganglia.arg' )
print graph
print 'properties:', graph.keys()
for p, v in graph.iteritems():
    print p, ':', v

graph[ 'gudule' ] = [ 12, 'a comment' ]
```

*Note:* Only properties declared in a "syntax" file may be saved and re-loaded. Other properties are just not saved.

## Vertices

Vertices (or nodes) can be accessed via the vertices() method. Each vertex is also a dictionary-like properties set.

```
for v in graph.vertices():
    print v['name']
```

To insert a new vertex, the addVertex() method should be used:

```
v = graph.addVertex( 'roi' )
print v
v[ 'name' ] = 'new ROI'
```

## Edges

An edge, or relation, links nodes together. Up to now we have always used binary, unoriented, edges. They can be added using the `addEdge()` method. Edges are also dictionary-like properties sets.

```
v2 = graph.vertices().list()[1]
e = graph.addEdge( v, v2, 'roi_link' )
print graph.edges()
# get vertices linked by this edge
print e.vertices()
```

## Adding meshes or buckets in a graph vertex or relation

Setting meshes or buckets in vertices properties is OK internally, but for saving and loading, additional consistency must be ensured and internal tables update is required. Then, use the `aims.GraphManip.storeAims` function:

```
mesh = aims.read( 'data_for_anatomist/subject01/subject01_Lwhite.mesh' )
# store mesh in the 'roi' property of vertex v of graph graph
aims.GraphManip.storeAims( graph, v, 'roi', mesh )
```

## Other examples

There are other examples for pyaims referenced [here](#).

## Using algorithms

AIMS contains, in addition to the different data structures used in neuroimaging, a set of algorithms which operate on these structures. Currently only a few of them have Python bindings, because we develop these bindings in a "lazy" way, only when they are needed. The algorithms currently available include data conversion, resampling, thresholding, mathematical morphology, distance maps, the mesher, some mesh generators, and a few others. But most of the algorithms are still only available in C++.

### Volume Thresholding

```
from soma import aims, aimsalgo
# read a volume with 2 voxels border
vol = aims.read( 'data_for_anatomist/subject01/subject01.nii', border=2 )
# use a thresholder which will keep values above 600
ta = aims.AimsThreshold( aims.AIMS_GREATER_OR_EQUAL_TO, 600, intype=vol )
# use it to make a binary thresholded volume
tvol = ta.bin( vol )
aims.write( tvol, 'thresholded.nii' )
```

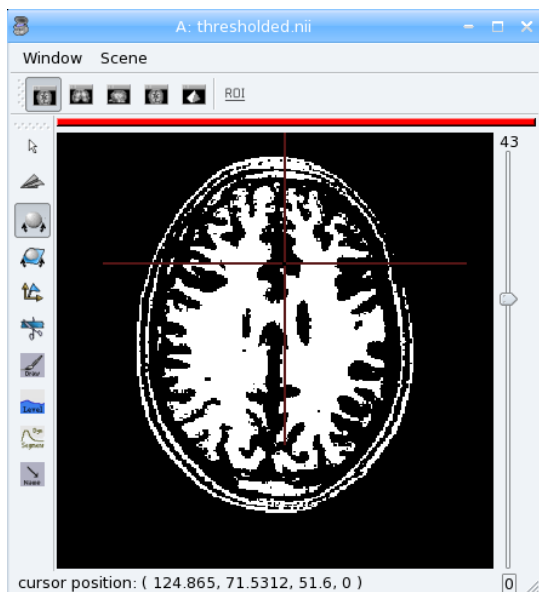


Figure 10. Thresholded T1 MRI

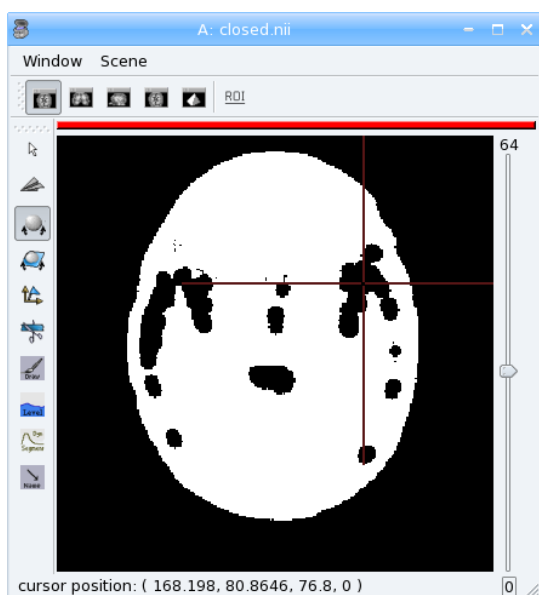
## Warning

Some algorithms need that the volume they process have a **border**: a few voxels all around the volume. Indeed, some algorithms can try to access voxels outside the boundaries of the volume which may cause a segmentation error if the volume doesn't have a border. That's the case for example for operations like erosion, dilation, closing. There's no test in each point to detect if the algorithm tries to access outside the volume because it would slow down the process.

In the previous example, a 2 voxels border is added by passing a parameter *border=2* to *aims.read* function.

## Mathematical morphology

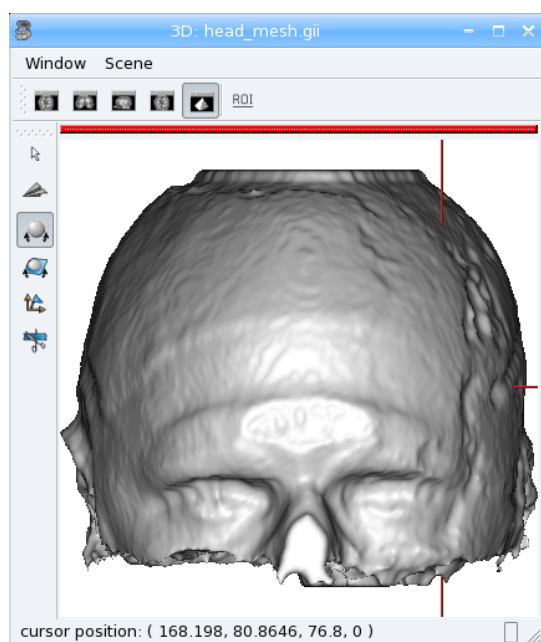
```
# apply 5mm closing
clvol = aimsalgo.AimsMorphoClosing( tvol, 5 )
aims.write( clvol, 'closed.nii' )
```



**Figure 11. Closing of a thresholded T1 MRI**

## Mesher

```
m = aimsalgo.Mesher()
mesh = aims.AimsSurfaceTriangle() # create an empty mesh
# the border should be -1
clvol.fillBorder( -1 )
# get a smooth mesh of the interface of the biggest connected component
m.getBrain( clvol, mesh )
aims.write( mesh, 'head_mesh.gii' )
```



**Figure 12. Head mesh**

The above examples make up a simplified version of the head mesh extraction algorithm in `VipGetHead`, used in the T1 pipeline.

## Surface generation

The `aims.SurfaceGenerator` allows to create simple meshes of predefined shapes: cube, cylinder, sphere, icosehedron, cone, arrow.

```
from soma import aims
center = ( 50, 25, 20 )
radius = 53
mesh1 = aims.SurfaceGenerator.icosahedron( center, radius )
# this dictionary-based generation will work correctly in pyaims >= 3.2.1
mesh2 = aims.SurfaceGenerator.generate( { 'type' : 'arrow', 'point1' : [ 30, 70, 0 ],
    'point2' : [ 100, 100, 100 ], 'radius' : 20, 'arrow_radius' : 30,
    'arrow_length_factor' : 0.7, 'facets' : 50 } )
# get the list of all possible generated objects and parameters:
print aims.SurfaceGenerator.description()
```

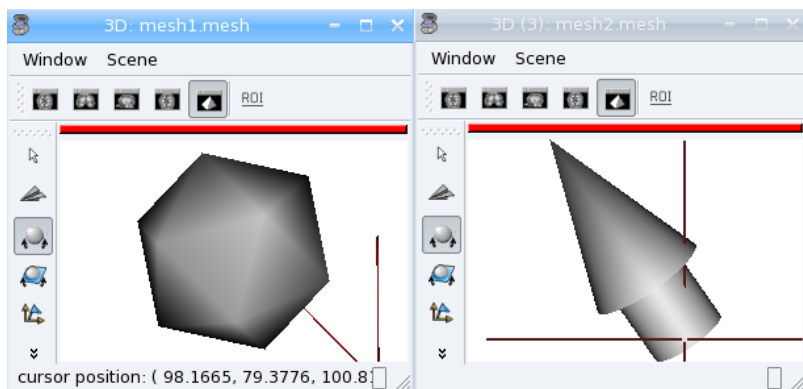


Figure 13. Generated icosahedron and arrow

## Interpolation

Interpolators help to get values in millimeters coordinates in a discrete space (volume grid), and may allow voxels values mixing (linear interpolation, typically).

```

from soma import aims
# load a functional volume
vol = aims.read( 'data_for_anatomist/subject01/Audio-Video_T_map.nii' )
# get the position of the maximum
pmax, maxval = aims.AimsData_DOUBLE( vol ).maxIndex()
# set pmax in mm
vs = vol.header()[ 'voxel_size' ]
pmax = [ x * y for x,y in zip( pmax, vs ) ]
# take a sphere of 5mm radius, with about 200 vertices
mesh = aims.SurfaceGenerator.sphere( pmax[:3], 5., 200 )
vert = mesh.vertex()
# get an interpolator
interpolator = aims.aims.getLinearInterpolator( vol )
# create a texture for that sphere
tex = aims.TimeTexture_FLOAT()
tx = tex[0]
tx2 = tex[1]
tx.reserve( len( vert ) )
tx2.reserve( len( vert ) )
for v in vert:
    tx.append( interpolator.value( v ) )
    # compare to non-interpolated value
    tx2.append( vol.value( *[ int( round(x/y) ) for x,y in zip( v, vs ) ] ) ) )

aims.write( tex, 'functional_tex.gii' )
aims.write( mesh, 'sphere.gii' )

```

Look at the difference between the two timesteps (interpolated and non-interpolated) of the texture in Anatomist.

## Types conversion

The Converter\_\*\_\* classes allow to convert some data structures types to others. Of course all types cannot be converted to any other, but they are typically used to convert volumes from a given voxel type to another one. A "factory" function may help to build the correct converter using input and output types. For instance, to convert the anatomical volume of the previous examples to float type:

```

from soma import aims
vol = aims.read( 'data_for_anatomist/subject01/subject01.nii' )
print 'type of vol:', type( vol )

```

```
c = aims.Converter( intype=vol, outtype=aims.Volume('FLOAT') )
vol2 = c( vol )
print 'type of converted volume:', type( vol2 )
print 'value of initial volume at voxel (50,50,50):', vol.value( 50, 50, 50 )
print 'value of converted volume at voxel (50,50,50):', vol2.value( 50, 50, 50 )
```

## Resampling

Resampling allows to apply a geometric transformation or/and to change voxels size. Several types of resampling may be used depending on how we interpolate values between neighbouring voxels (see interpolators): nearest-neighbour (order 0), linear (order 1), spline resampling with order 2 to 7 in AIMS.

```
from soma import aims, aimsalgo
import math
vol = aims.read( 'data_for_anatomist/subject01/subject01.nii' )
# create an affine transformation matrix
# rotating pi/8 along z axis
tr = aims.AffineTransformation3d( aims.Quaternion( [ 0, 0, math.sin( math.pi/16 ),
math.cos( math.pi/16 ) ] ) )
tr.setTranslation( ( 100, -50, 0 ) )
# get an order 2 resampler for volumes of S16
resp = aims.ResamplerFactory_S16().getResampler( 2 )
resp.setDefaultValue( -1 ) # set background to -1
resp.setRef( vol ) # volume to resample
# resample into a volume of dimension 200x200x200 with voxel size 1.1, 1.1, 1.5
resampled = resp.doit( tr, 200, 200, 200, ( 1.1, 1.1, 1.5 ) )
# Note that the header transformations to external referentials have been updated
print resampled.header()[ 'referentials' ]
print resampled.header()[ 'transformations' ]
aims.write( resampled, 'resampled.nii' )
```

Load the original image and the resampled in Anatomist. See how the resampled has been rotated. Now apply the NIFTI/SPM referential info on both images. They are now aligned again, and cursor clicks correctly go to the same location on both volume, whatever the display referential for each of them.

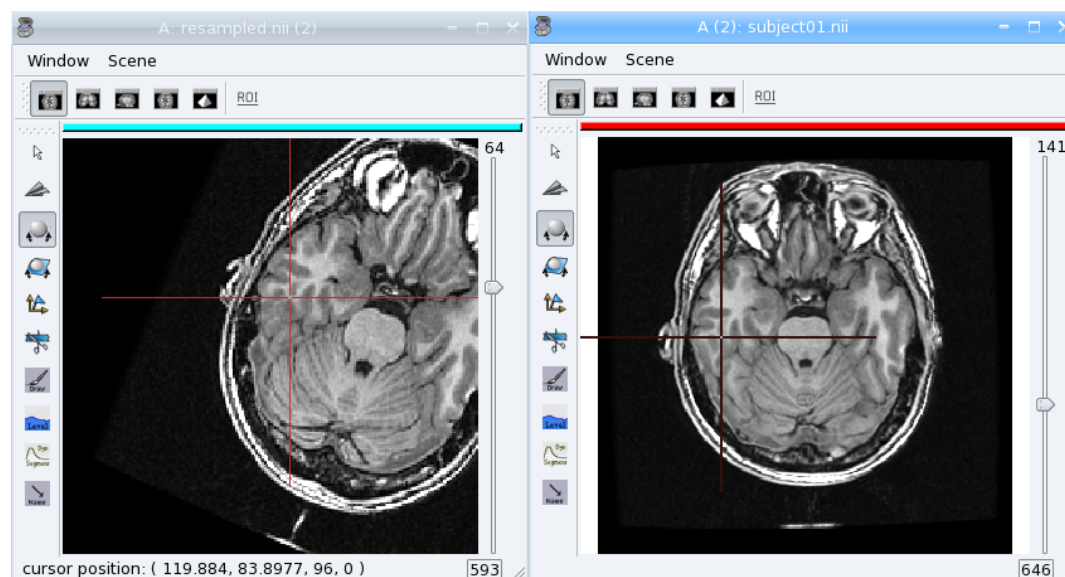


Figure 15. Aimsalgo resampling

## PyAIMS / PyAnatomist integration

### Running PyAnatomist with PyAims

When PyAnatomist is run in "direct" mode (library bindings), it is possible to share objects between processing routines in pyaims and viewing models in Anatomist, and interact on them directly and interactively.

For an interactive shell with Anatomist rendering enabled, IPython should be run in an appropriate mode for graphical events loop to run. If Anatomist is built using Qt3:

```
ipython -qthread
```

If Anatomist is built using Qt4:

```
ipython -q4thread
```

Then, in IPython, start Anatomist in direct mode:

```
import anatomist.direct.api as ana  
a = ana.Anatomist()
```

Now Anatomist main window is here.

### Sharing objects between Aims and Anatomist

Objects loaded in pyaims may be wrapped as Anatomist objects:

```
from soma import aims  
vol = aims.read( 'data_for_anatomist/subject01/subject01.nii' )  
anavol = a.toAObject( vol )  
win = a.createWindow( 'Axial' )  
win.addObject( anavol )
```

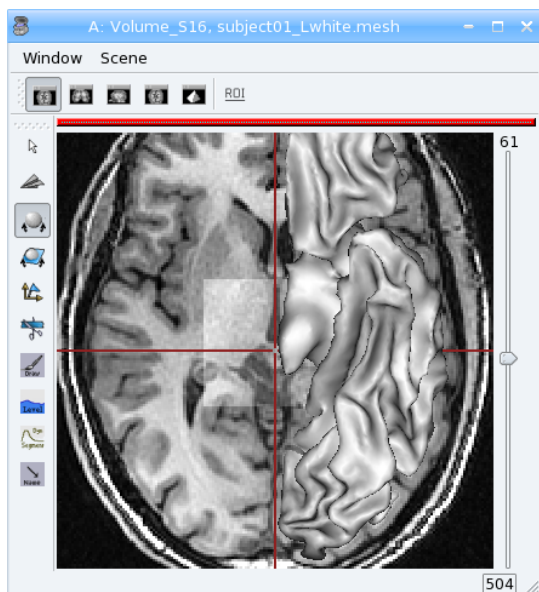
Or, in the other way, Anatomist objects may be exported as AIMS objects:

```
anamesh = a.loadObject( 'data_for_anatomist/subject01/subject01_Lwhite.mesh' )  
mesh = a.toAimsObject( anamesh )  
win.addObject( anamesh )
```

### Interactive modifications in objects

Objects may be modified on AIMS side. Then Anatomist must be notified of such modifications so that views on the objects can be refreshed.

```
import numpy  
arr = numpy.array( vol, copy=False )  
arr[ 100:150, 100:150, 50:80 ] += 200  
anavol.setChanged() # say this object has changed  
anavol.notifyObservers() # refresh views and other interactions
```

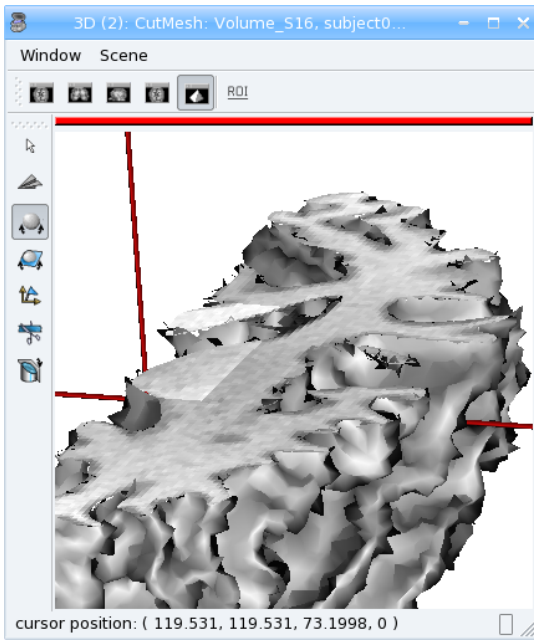


**Figure 16. 3D volume modified with numpy**

It is also possible to interact on objects created within Anatomist:

```
cutmesh = a.fusionObjects( [ anavol, anamesh ], method='FusionCutMeshMethod' )
win2 = a.createWindow( '3D' )
win2.addObject( cutmesh )
# find the cut sub mesh in cutmesh children
anacutsubmesh = filter( lambda x: x.name.startswith( 'CutSubMesh' ), cutmesh.children
)[0]
# get the Aims mesh from it
cutsubmesh = a.toAimsObject( anacutsubmesh )
# now modify the mesh, adding a random value to vertices along the normal
vert = cutsubmesh.vertex()
norm = cutsubmesh.normal()
for v,n in zip( vert, norm ):
    v += n * numpy.random.randn()

anacutsubmesh.setChanged()
anacutsubmesh.notifyObservers()
```



**Figure 17. Modified cut mesh**