Contents

[2. Introduction 2](#_Toc487528332)

[3. GRAPE User Interface 2](#_Toc487528333)

[4. Pipeline Tools 3](#_Toc487528334)

[5. Creating Pipeline 4](#_Toc487528335)

[6. Developing new nodes 4](#_Toc487528336)

[7. List of nodes 5](#_Toc487528337)

[7.1 BET 5](#_Toc487528345)

[7.2 Bias Correction 7](#_Toc487528346)

[7.3 Branch 8](#_Toc487528347)

[7.4 Curve fitting 9](#_Toc487528348)

[7.5 Delay 10](#_Toc487528349)

[7.6 Divider 11](#_Toc487528350)

[7.7 End 11](#_Toc487528351)

[7.8 Image Arithmetic 11](#_Toc487528352)

[7.9 Image Normalizer 12](#_Toc487528353)

[7.10 Image Reader 12](#_Toc487528354)

[7.11 Image Reformat 13](#_Toc487528355)

[7.12 Image Registration 14](#_Toc487528356)

[7.13 Image Segmentation 16](#_Toc487528357)

[7.14 Image Slicer 17](#_Toc487528358)

[7.15 Image Splitter 18](#_Toc487528359)

[7.16 Image Stack 18](#_Toc487528360)

[7.17 Image Statistics 18](#_Toc487528361)

[7.18 Image Writer 18](#_Toc487528362)

[7.19 Logical Operator 19](#_Toc487528363)

[7.20 Loop 19](#_Toc487528364)

[7.21 Parameter Reader 21](#_Toc487528365)

[7.22 Parameter Writer 21](#_Toc487528366)

[7.23 Relational Operator 22](#_Toc487528367)

[7.24 Select 22](#_Toc487528368)

[7.25 Sink 22](#_Toc487528369)

[7.26 Source 23](#_Toc487528370)

[7.27 Synch 23](#_Toc487528371)

[7.28 System call 24](#_Toc487528372)

# Introduction

GRAphical Pipeline Environment (GRAPE) is an open source Qt/C++ project used for the design and execution of algorithms. GRAPE provides a simple and flexible cross-platform algorithm development environment that allows users to generate new algorithms, analyses, and pipelines. GRAPE can be used for general visual programming tasks, but is primarily motivated by its intended applications in magnetic resonance imaging (MRI) data analysis, particularly in adaptive MRI scanning.

This guide describes the software architecture of GRAPE and the programming tools available to the user and the developer.

# GRAPE User Interface

The GRAPE graphical interface consists of the pipeline layout panel and the pipeline development toolbar (Fig. 1). The pipeline layout panel is an interactive canvas that allows the user to create and edit various modules of the analysis pipeline (as nodes), and control the direction and flow of data (as edges). The pipeline development toolbar includes basic drawing tools, execution buttons, and the module library. The module library shows all the available modules for the user, organized by both GRAPE built-in libraries and user-created libraries. From the toolbar, the user can also start and stop the execution of the current pipeline.

Selecting the Play button puts the pipeline in the run mode. Once the pipeline is in the run mode, editing of the modules is disabled. During execution, the modules show color cues indicating the current progress of the pipeline, and time-stamped events are written to a log file. Another feedback mechanism is available with image display modules programmed to display 2D, 3D, or 4D images, allowing the user to review the pipeline outputs at various stages. This is particularly useful during the initial development stages of analysis algorithms. The pipeline periodically checks the status of each node, and re-executes the nodes when their input data are changed. This is useful for nodes that require, for example, input from two separate branches in the pipeline. The user may interrupt the execution of the pipeline at any time by pressing the Stop button. Pipelines may also be saved, shared, and restored.

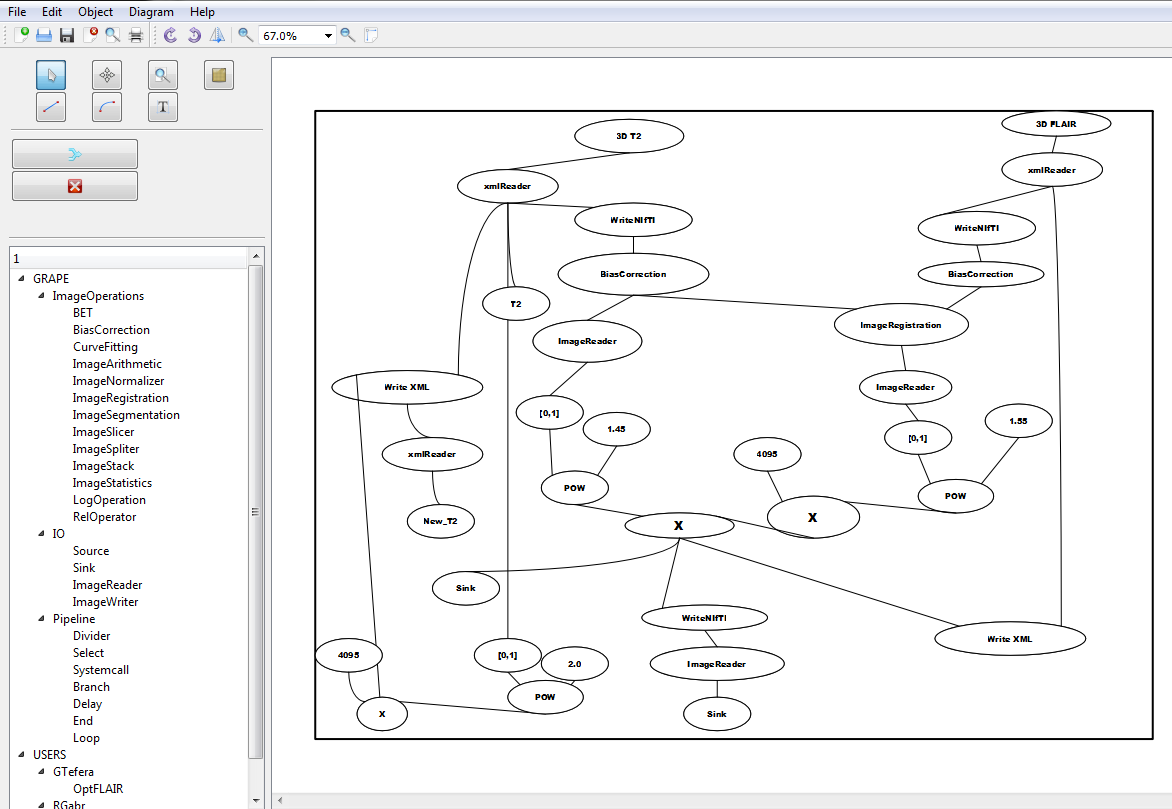


Fig. 1. Elements of the graphical user interface in GRAPE.

# Pipeline Tools

GRAPE includes basic programming elements such as logical operations, list selection, branching, and looping. These are all implemented as modules that the user can modify and use directly without the need for programming, allowing quick prototyping of new pipelines. In addition, these tools are also useful for pipelines that perform batch processing of multiple datasets. Special Source and Sink nodes provide convenient means to generate test data and parameters, to display analysis outputs, and to manually debug pipelines.

Basic image operations are implemented to facilitate the development of new algorithms. The image library includes modules for image arithmetic, slicing, splitting, statistics, and comparison. Widely used tools such as BET brain extraction (Smith 2002), no uniformity correction methods (Tustison et al. 2010), image registration (Collignon et al. 1995), and segmentation (Avants, Tustison, Wu, et al. 2011) are also available. Reading in and writing to popular image formats such as Analyze and NIFTI is supported. In addition, read/write is supported for the Philips XML/REC research format. The latter is crucial for communicating data with Philips MRI systems, allowing data exchange between the scanner and the analysis pipelines.

New user programs and image analysis functionalities in third party libraries are allowed through command line calls to executable files. The GRAPE interface allows the user to customize a system-call module to define the inputs and outputs to and from the executable program. This provides a high degree of flexibility for the user, avoids the need for re-programming tasks, and allows the extension of the program with a moderate effort.

# Creating Pipeline

An image analysis pipeline is built by selecting the desired modules from the available libraries and adding it to the canvas. For example, there are modules for reading data, performing image processing tasks, and writing output. The nodes are connected according to the desired data flow plan of the pipeline. For each module, a property dialog is defined and can be accessed from the module context menu. Through the module property dialog, various attributes such as the legend, rendering options and functional parameters can be modified.

# Developing new nodes

GRAPE is implemented in the Qt 5.4.2 C++ application development framework. It uses elements from the open-source projects JADE (https://sourceforge.net/projects/jade-diagram) used for diagram editing and from the Template Image Processing Library https://www.nitrc.org/projects/tipl/) for handling DICOM, NIFTI, and Analyze image file formats and some image processing functions. GRAPE was successfully compiled for Windows, Linux, and Mac operating systems without modifications, and was run on both standalone computers as well as on the Stampede Dell PowerEdge Linux cluster at the Texas Advanced Computing Center (Austin, TX, USA).

Developers can implement new modules by defining a Node Interface descriptor in the node definition file. The descriptor specifies the number of inputs and outputs, the type and default values of the inputs and outputs, and which inputs are mandatory. The Node Interface is used to populate the library viewer in the toolbar, and to dynamically generate the graphical representation of the node. The functional component of the module is implemented by a Node class. Each node implementation must define four basic methods: input assignment, output assignment, validation, and execution.

# List of nodes

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Nodes | Number of in Ports | Number of out Ports |
| 1 | Bet | 1 | 2 |
| 2 | Bias Correction | 2 | 1 |
| 3 | Branch | 2 | 2 |
| 4 | Curve Fitting | 1 | 2 |
| 5 | Delay | 1 | 1 |
| 6 | Divider | 1 | 3 |
| 7 | End | 1 | 0 |
| 8 | Image Arithmetic | 2 | 1 |
| 9 | Image Normalizer | 1 | 1 |
| 10 | Image Reader | 1 | 1 |
| 11 | Image Reformat | 1 | 1 |
| 12 | Image Registration | 2 | 1 |
| 13 | Image Segmentation | 4 | 2 |
| 14 | Image Slicer | 3 | 1 |
| 15 | Image Splitter | 1 | 2 |
| 16 | Image Stack | 7 | 1 |
| 17 | Image Statistics | 2 | 1 |
| 18 | Image Writer | 2 | 1 |
| 19 | Logical operator | 2 | 1 |
| 20 | Loop | 3 | 2 |
| 21 | Parameter Reader | 1 | 8 |
| 22 | Parameter Writer | 7 | 1 |
| 23 | Relational Operator | 2 | 1 |
| 24 | Select | 1 | 1 |
| 25 | Sink | 1 | 0 |
| 26 | Source | 0 | 1 |
| 27 | System Call | 5 | 3 |



## **BET**

**Purpose**: node BET is for brain-skull stripping. It works with node SOURCE, IMAGEREADER and SINK together in Fig.7.1.1.

**Input A** (in Fig.7.1.1): a string of input file name that can be set by node SOURCE (in ‘Data Value’, Fig.7.1.2) including:

* Input File Name: FLAIR3D.hdr (for example)
* Path: ../example/data/FLAIR3/

**Output B**: a string of output file name for stripped image, set by node BET at ‘Output File Prefix’(=bet\_ in Fig.7.1.3) that defines

* stripped image file name = [Output File Prefix] [Input File Name] = bet\_FLAIR3D(8 bit file)
* Path: same as input.

**Output C**: a string of output file name for mask image defined below

* Th stripped image file name = bet\_FLAIR3D\_mask (8 bit file)
* Path: same as input.

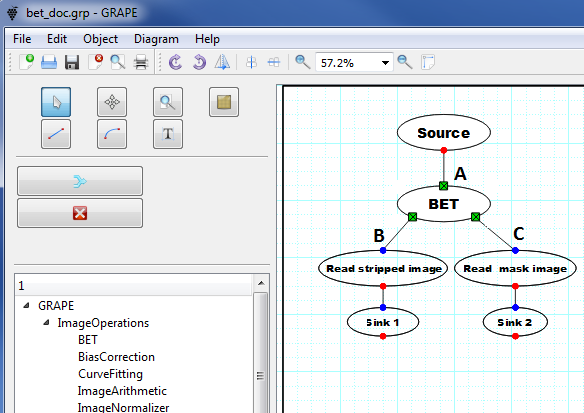
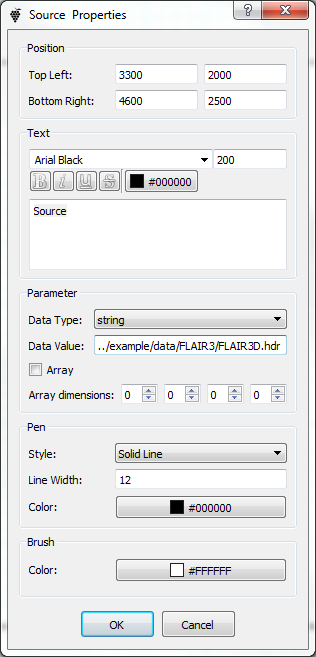
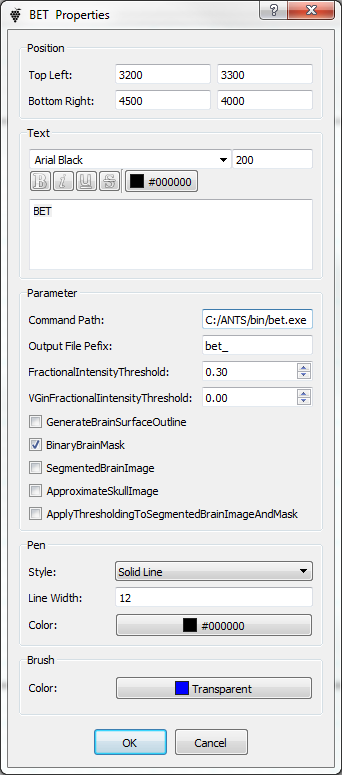
  

Fig. 7.1.1 Fig. 7.1.2 node source Fig. 7.1.3 node bet

**Usage:** set the following parameters before running node BET

* Command Path: full path to file bet.exe, for example, C:/ANTS/bin/bet.exe (Fig.7.1.3).
* FractionIntensityThreshold: used for bet-command[1], for example, 0.3 in Fig.7.1.3.
* VGinFractionIntensityThreshold: used for bet-command[1], for example, 0.0 in Fig.7.1.3.

Run pipeline: click button ‘>’ in Fig.7.1.1 that strips input image and displayed the stripped image (Fig.7.1.4) and its mask(Fig.7.1.5).

[1] bet command ( ˽ is a space letter)

[Command Path]˽[Input File]˽[Output File]˽-f˽[FractionIntensityThreshold]˽-m˽-g˽[VGinFractionIntensityThreshold]

Example:

C:/ANTS/bin/bet.exe ../example/data/FLAIR3/flair3d.hdr ../example/data/FLAIR3/bet\_flair3d -f 0.3 -m -g 0.0

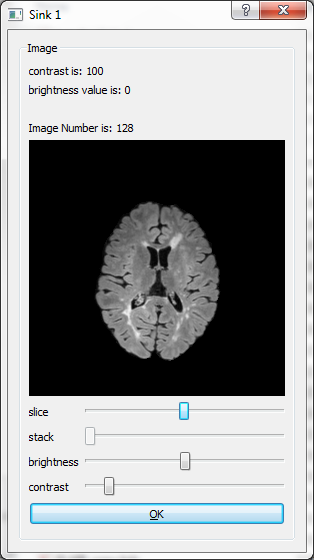
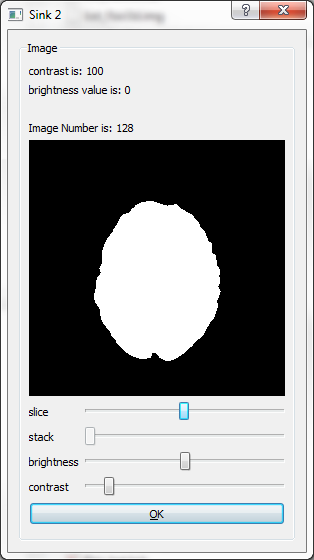
 

Fig-7.1.5 stripped Fig-7.1.6 mask

## **Bias Correction**

**Purpose**: node BIAS CORRECTION is for MRI bias field correction with two type options: N3 or N4. Fig.7.2.1 shows a test pipeline (Fig. 7.2.1).

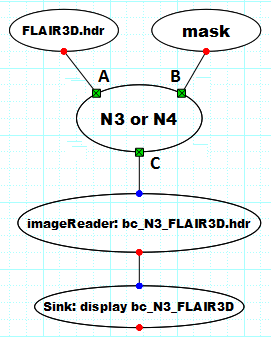
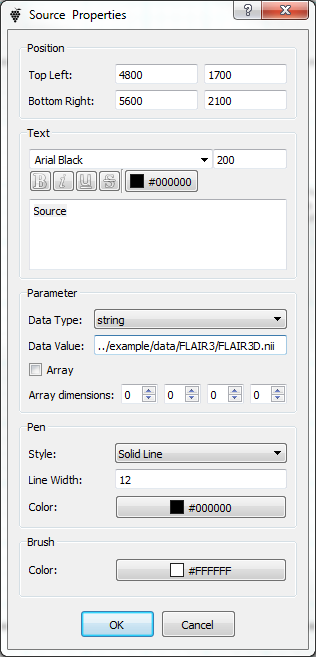
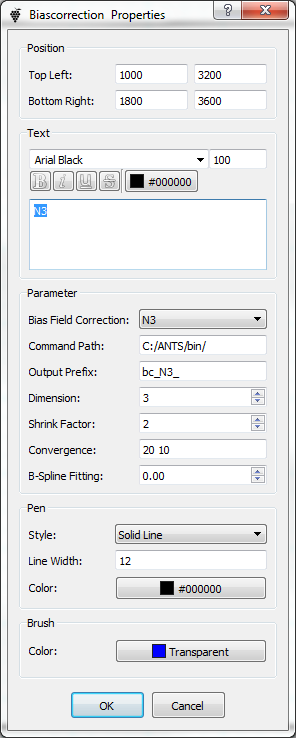
  

Fig 7.2.1 Bias correction Fig-7.2.2 source Fig-7.2.3 option N3

**Input A** (in Fig.7.2.1): string of input file name put by node SOURCE (in ‘Data Value’, Fig.7.2.2) including:

* Input File Name: flair3d.hdr (for example)
* Path: ../example/data/FLAIR3/

**Input B**: string of input mask file, set by another SOURCE with

* mask File Name: bet\_flair3d\_mask.hdr (for example)
* Path: same as input.

**Output C** (in Fig.7.2.1): string of output file name for bias-corrected image. It can be set by node BIASCORRECTION at ‘Output Prefix’(=bc\_N3\_ in Fig.7.2.3). This defines

* corrected image file name = [Output File Prefix] [Input File Name]=bc\_N3\_flair3d(32 bit file)
* Path: same as input.

**Usage:** set the following parameters before running node BIAS CORRECTION

* Bias Field Correction: N3/N4 (in Fig 7.2.3)
* Command Path: path to file ***N3BiasFieldCorrection***, for example, C:/ANTS/bin/ in Fig 7.2.3.
* Dimension: used for bias-command[2], for example, 3 (for 3D image; 2 for 2D image)
* Shrink Factor: used for bias-command[2], for example, 2 (in Fig.7.2.3).
* Convergence: used for bias-command[2], for example, 20 ˽ 10 (in Fig.7.2.3) for N3 (or N4).

Run pipeline.

[2] bias-command ( ˽ is a space letter)

***N3BiasFieldCorrection***˽imageDimension˽inputImage˽outputImage˽[shrinkFactor]˽ [maskImage]˽[numberOfIterations]˽[numberOfFittingLevels]˽[outputBiasField]

bias command example:

C:\ANTS\bin\N3BiasFieldCorrection 3 FLAIR3D.hdr bc\_N3\_FLAIR3D.hdr 2 20 10

([shrinkFactor]=2, [numberOfIterations]=20, [numberOfFittingLevels]=10)

## **Branch**

Purpose: node BRANCH has two inputs and two outputs decripted in Fig.7.3.1. The left input A is real input and the right one B is boolean for control. BRANCH can switch one of its outputs to connect the real input A.

**Function description**: an example pipeline Fig.7.3.1. Node SOURCE L sets BRANCH input A = string ‘I am here’(Fig.7.3.2). If set input B = 1 by SOURCE R (Fig.7.3.3), BRANCH output C will be active and pushes out this string to node SINK L. Instead, If BRANCH B = 0, BRANCH’s output D gets active for SINK R to display the string.

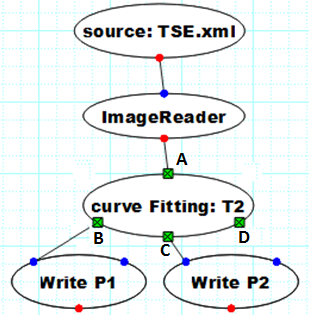
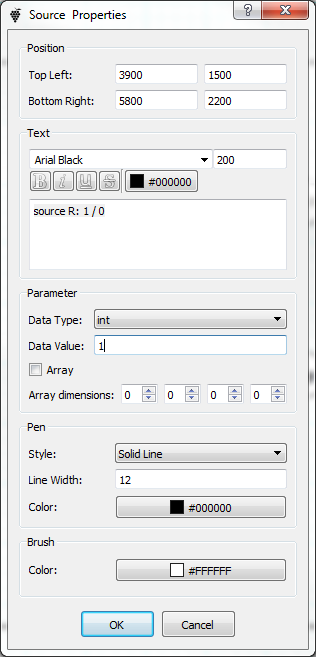
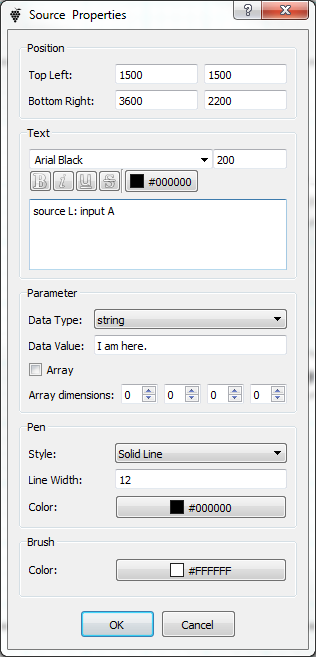
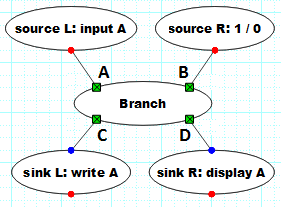


Fig-7.3.1 Branch Fig-7.3.2 source L Fig-7.3.3 source R Fig-7.4.1 CurveFitting

## **Curve fitting**

**Purpose**: node CURVEFITTING is for MRI parameter fitting. It can work in serveral fitting models: T1-IR, T2 and Looklock. Node CURVEFITTING has one input A and three outputs B, C and D as shown in Fig.7.4.1 of a test pipeline for T2-fitting. T2-fitting can generate two fitting-images P1 and P2.

**Input A**: image data provied by node IMAGEREADER that reads a image file set by node SOURCE (in ‘Data Value’, Fig.7.4.2) including:

* Input File Name: TSE.xml (for example)
* Path: ../example/data/TestDataXMLREC/

**Output B:** the first fittingimage data P1 (16 bit integer ) that can be written by node IMAGEWRITER A(Fig. 7.4.3), saved in name P1.nii in input path.

**Output C:** the second fittingimage data P2 (16 bit integer ) that can be written by node IMAGEWRITER B(Fig. 7.4.4) , saved in name P2.nii input path.

**Usage:** set the following options on node CURVEFITTING (Fig. 7.4.5) before running the pipeline:

* Curve Fitting: T2 (for example)
* xData: 9.5,90
* parValue: 2

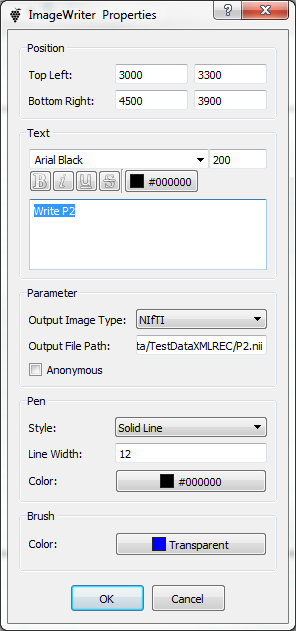
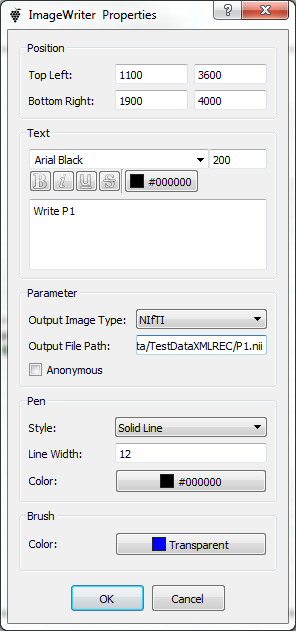
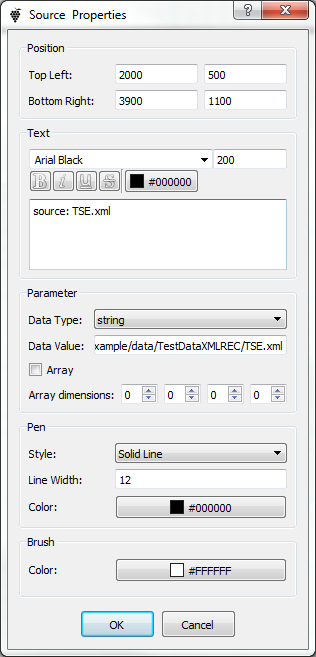
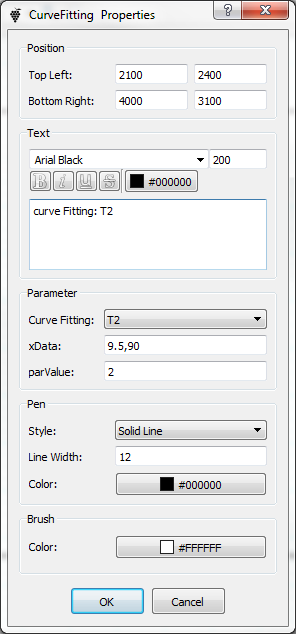
 

Fig-7.4.2 source Fig-7.4.3 Imagewriter Fig-7.4.4 Imagewriter Fig-7.4.5 curveFitting

## **Delay**

**Purpose**: node DELAY can delay its input going through to its output in msec.

**Input**: can be string, number and image data array. A test pipeline(Fig.7.5.1) pushes a string to node DELAY .

**Output:** same type as input, delay to appear on its output. Fig.7.5.1 displays the string 100.0 ms late.

**Usage**: set ‘Delay Time (msec)’=100.0 in Fig. 7.5.3.

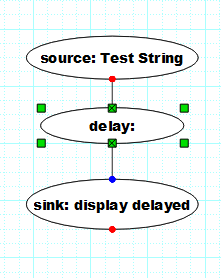
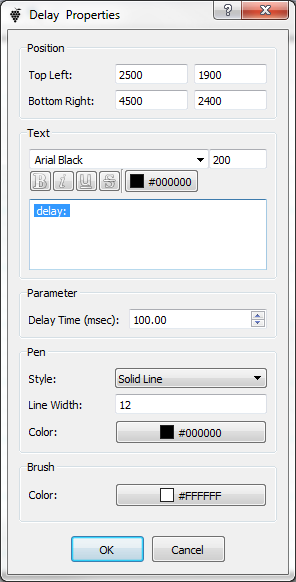
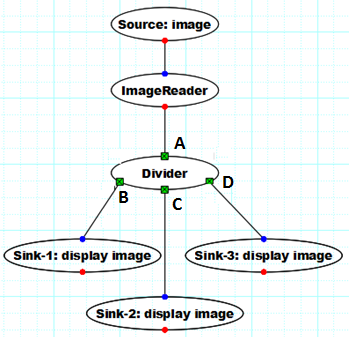
  

Fig-7.5.1 Use Delay Fig-7.5.3 node delay Fig-7.6.1 Divider

## **Divider**

**Purpose**: node DIVIDER has one input A and three outputs B, C and D shown in a test pipeline Fig. 7.6.1. That can tranfer input A to its three outputs, for example, Fig. 7.6.1 can display three image same as input.

**Input**: can be string, number and image data array.

**Output**: same type as input.

## **End**

Usage: put node END at the end of a pipeline to stop or exit.

## **Image Arithmetic**

**Purpose**: node IMAGEARITHMETIC has two inputs A and B, one output C as shown in Fig. 7.8.1. That can do an arithmetic calculation from A and B, result in an output C.

|  |  |
| --- | --- |
| **Input A and B**: can be number or image data.  **Output C**: depending on inputs by the table below  **Usage**: node IMAGEARITHMETIC has 8 models options, shown in Fig. 7.8.2.      Fig-7.8.1 a test pipeline of arithmetic Fig-7.8.2 models options on imageArithmetic |  |

## **Image Normalizer**

**Purpose**: node IMAGENORMALIZER is for MRI normalization. Fig.7.9.1 shows a test pipeline.

**Input:** image data

**Output:** image data normalized by the following equation in which *max\_value* and *min\_value* can be set in node property (Fig. 7.9.2).



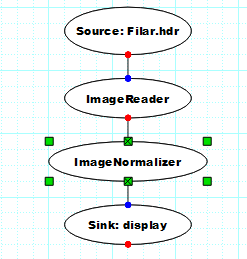
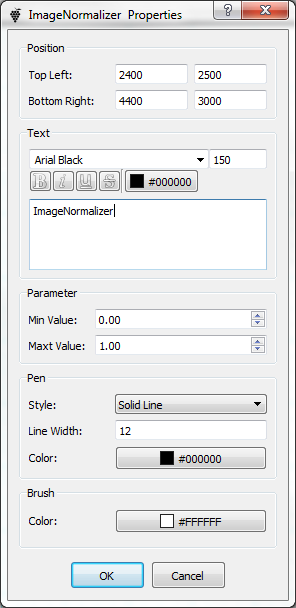
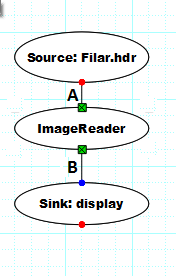
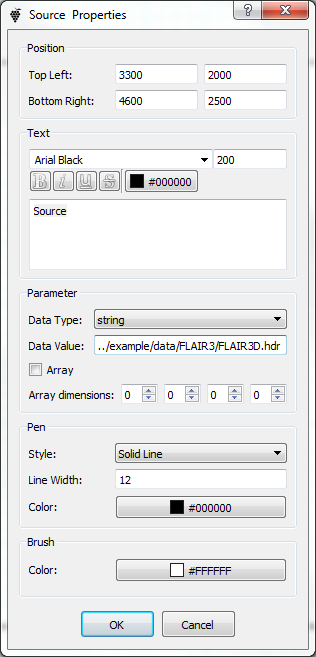
   

Fig. 7.9.1 test pipeline Fig. 7.9.2 imageArithmetic Fig. 7.10.1 (a) imagereader (b) node property

## **Image Reader**

**Purpose**: node IMAGEREADER can open an image file to read image data and output it. Fig.7.10.1 (a) shows it working in a test pipeline.

**Input A:** string of input file name provided by node SOURCE (in ‘Data Value’, Fig.7.10.1(b)) including:

* Input File Name: FLAIR3D.hdr (for example)
* Path: ../example/data/FLAIR3/

**Output B:** image data array

Usage: select an option of ‘Input Image Type’ (Fig. 7.10.2) before running the pipeline.

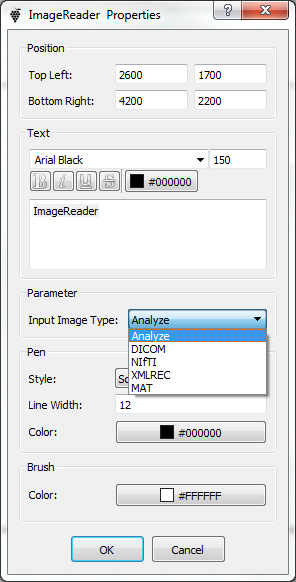
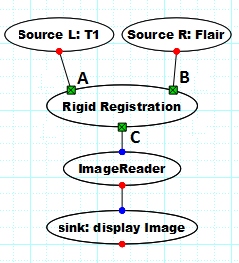
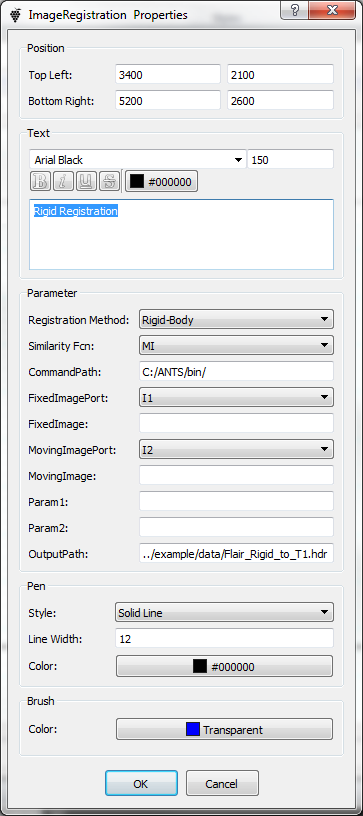
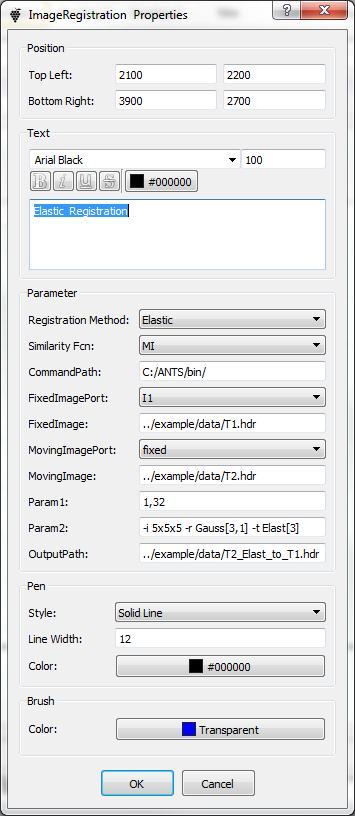
   

Fig. 7.10.2 node property Fig. 7.11.1 test pipeline Fig. 7.11.2 imageregistration property (a), (b)

## **Image Reformat**

**Purpose**: node IMAGEREFORMAT can reformat an image data array depending on setting the dimension oder A1, A2 ans A3 in its node property (Fig.7.11.3).

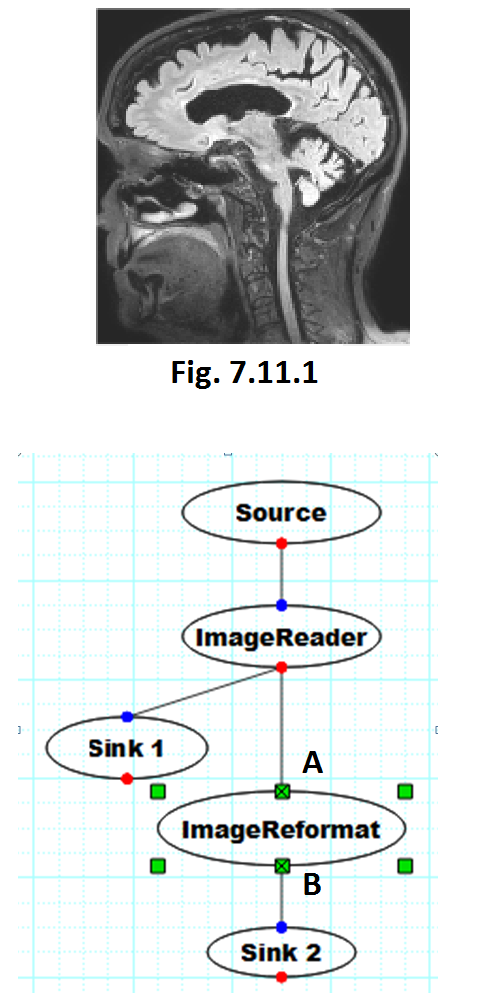
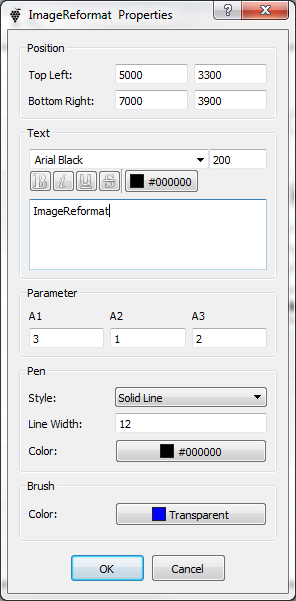
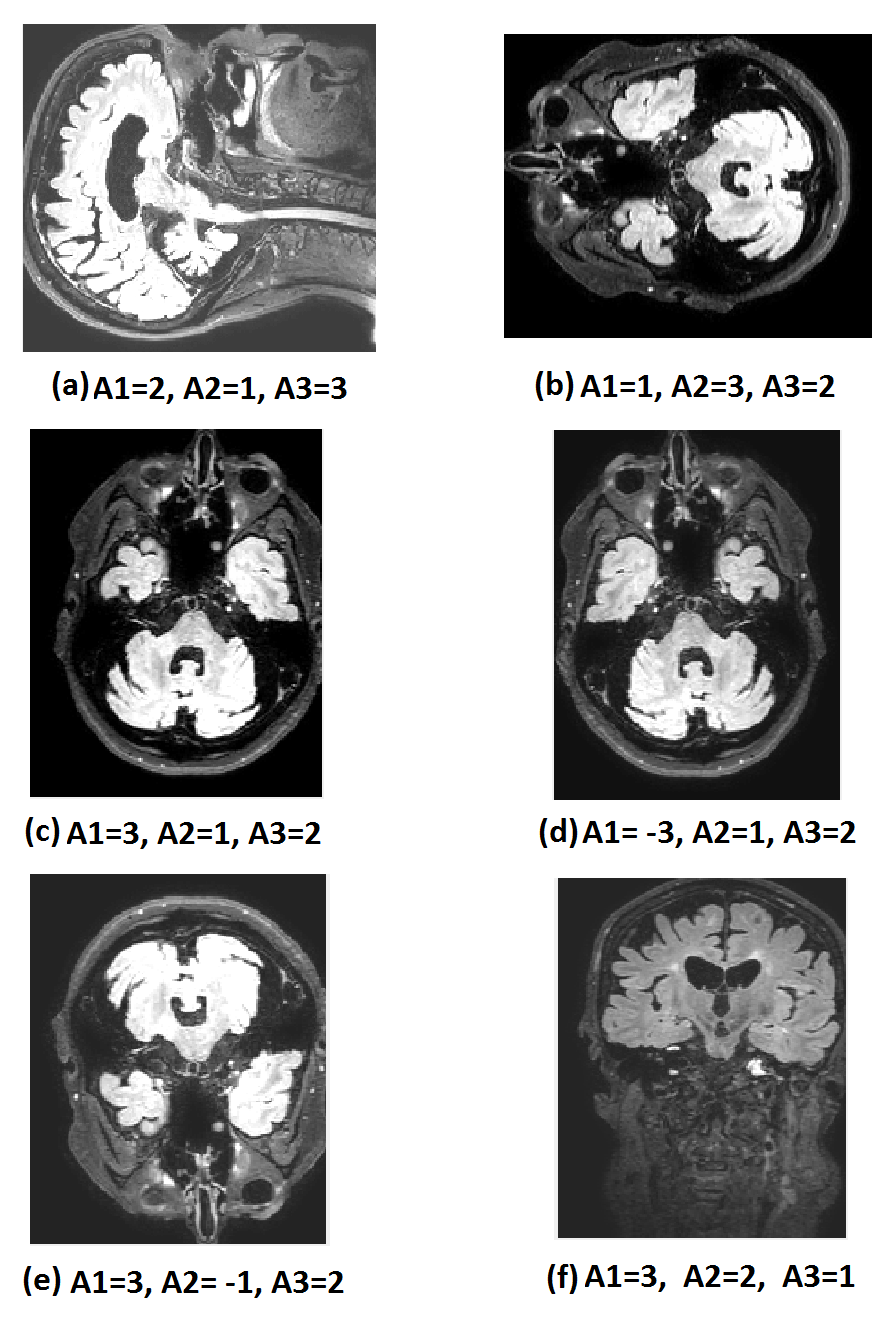
  

Fig. 7.11.2 Fig. 7.11.3 Fig.7.11-4 reformatted image

**Input A** (in Fig.7.11.2): input image array.

**Output** B:reformed image array

**Usage**: an input image (Fig.7.11.1) was displayed by sink 1 in a test pipe line (Fig.7.11.2), the image size is 198 x 218 x 167.

* If set dimension oder A1=1, A2=2 and A3=3 (Fig.7.11.3), then output image (displayed on sink 2) is same as input image(Fig.7.11.1).
* If set dimension oder A1=2, A2=1 and A3=3, then output image looks as Fig.7.11.4 (a)
* Fig.7.11.4 (b) ~ (f) show different dimension oder and related output images.

## **Image Registration**

**Purpose**: node IMAGEREGISTRATION is used to register a moving image to a fixed image. Four kind of registration methods (Fig.7.12.1) and three models (Fig.7.12.2) serve as options for image registration.

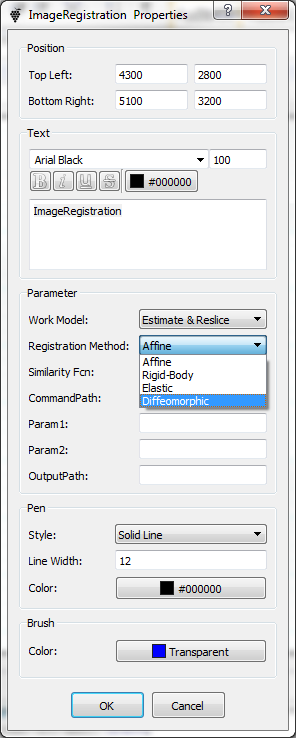
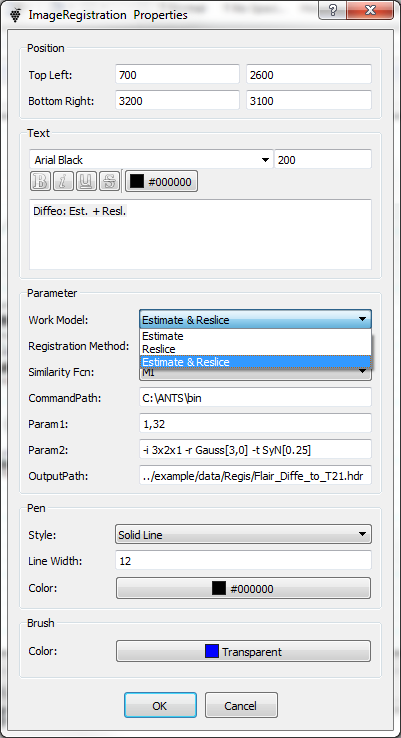
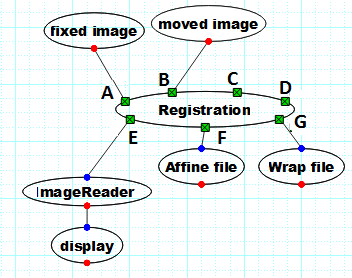
  

Fig.7.12.1 Fig.7.12.2 Fig. 7.12.3

**Performance:** node MAGEREGISTRATION works basing on ANTS tool[3] that provides open-source functionality for deformable image registration. node MAGEREGISTRATION calls two executable files of ANTS tool: *ANTS.exe* and *WarpImageMultiTransform.exe*. How to use the both of files can be set by ‘Work Mode’ in Fig.7.12.2:

* Estimate: only use *ANTS.exe* to create Affine-file and Warp-file;
* Reslice: only use *WarpImageMultiTransform.exe* (with Affine-file and Warp-file) to create registered image file;
* Estimate & Reslice: use both of *ANTS.exe* and *WarpImageMultiTransform.exe* to create registered image file.

[3] Brian B. Avants, el. at, Advanced Normalization Tools (ANTS), Penn Image Computing and Science Laboratory University of Pennsylvania.

**Input A** (in Fig.7.12.3): a string of input file name for fixed image (or reference image).

**Input B** : a string of input file name for moving image.

**Input C** : a string of input file name for Affine-file in ‘Reslice’ model.

**Input D** : a string of input file name for Warp-file in ‘Reslice’ model.

**Output E** : a string of registered image file name.

**Output F** : a string of Affine-file name.

**Output G** : a string of Warp-file name.

**Usage**: complete the following settings in node property (Fig.7.12.1~2) before running the node pipeline.

* Registration Method: one option of Affine, Body-Rigid, Elastic and Diffeomophic (Fig.7.12.1).
* Similarity Fcn: one option from MI, PR, CC, MSQ and PSE for ANTS-command[4].
* Command Path: a path to file *ANTS.exe*, for example, C:\ANTS\bin\(Fig.7.12.2).
* Para1 and Para2: required for ANTS-command[4] in model Elastic and Diffeomophic(Fig.7.12.2).
* OutputPath: a file name of the registered image file(Fig.7.12.2).

[4] Command examples: ( fixed image: T1; moving image: Flair; ˽ : space letter )

* Rigid-registration command: registered image name = Flair\_Rigid\_to\_T1 (for example)

*[Command Path]ANTS˽3˽-m˽[Similarity\_Fcn][T1.hdr,flair.hdr,1,32]˽-o˽Flair\_Rigid\_to\_T1˽-i˽0˽--rigid-affine-true*

*[Command Path]WarpImageMultiTransform˽3˽ flair.hdr˽ Flair\_Rigid\_to\_T1.hdr ˽-R ˽ T1.hdr ˽ Flair\_Rigid\_to\_T1Affine.txt*

* Affine-registration command: registered image name = Flair\_Affine\_to\_T1 (for example)

*[Command Path]ANTS˽3˽-m˽[Similarity\_Fcn][T1.hdr,flair.hdr,1,32]˽-o˽Flair\_Affine\_to\_T1˽-i˽0˽--number-of-affine 50x50x50 –rigid-affine false*

*[Command Path]WarpImageMultiTransform˽3˽ flair.hdr˽ Flair\_Affine\_to\_T1.hdr ˽-R ˽ T1.hdr ˽ Flair\_Affine\_to\_T1Affine.txt*

* Elastic-registration command: registered image name = *Flair2T1* (for example)

*[Command Path]ANTS˽3˽-m˽[Similarity\_Fcn][T1.hdr, flair.hdr,Para1]˽-o˽Flair2T1˽Para2*

*Para1 = 1,32*

*Para1 = -i˽3x2x1˽-r˽Gauss[3,1]˽-t˽Elast[3]*

*[Command Path]WarpImageMultiTransform˽3˽ flair.hdr˽ Flair2T1.hdr ˽-R ˽ T1.hdr ˽ Flair2T1Affine.txt ˽ Flair2T1Warp.nii.gz*

* Diffeomorphic-registration command: registered image name = *Flair2T1* (for example)

*[Command Path]ANTS˽3˽-m˽[Similarity\_Fcn][T1.hdr, flair.hdr,Para1]˽-o˽Flair2T1˽Para2*

*Para1 = 1,32*

*Para1 = -i˽3x2x1˽-r˽Gauss[3,0]˽-t˽Syn[0.25]*

*[Command Path]WarpImageMultiTransform˽3˽ flair.hdr˽ Flair2T1.hdr ˽-R ˽ T1.hdr ˽ Flair2T1Affine.txt ˽ Flair2T1Warp.nii.gz*

## **Image Segmentation**

**Purpose**: node IMAGESEGMENTATION is for MRI segmentation. A test pipeline Fig.7.13.1 shows an example of 3-channel segmentation.

**Input A**: a string of input file name on channel-1.

**Input B** : a string of input file name on channel-2 (optional).

**Input C** : a string of input file name on channel-3 (optional).

**Input D** : a string of input mask file name (optional).

**Output E** : not used.

**Output F** : a string of file name for segmented image.

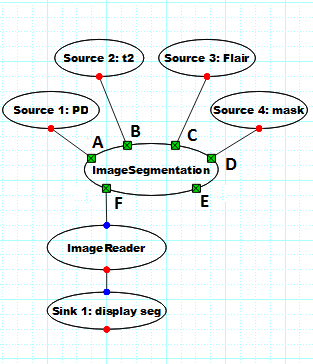
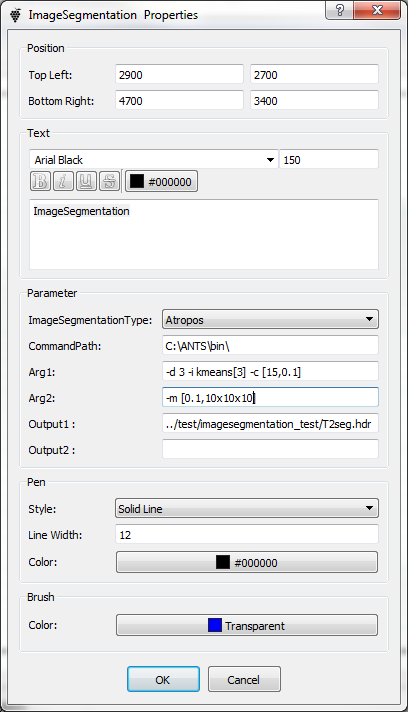
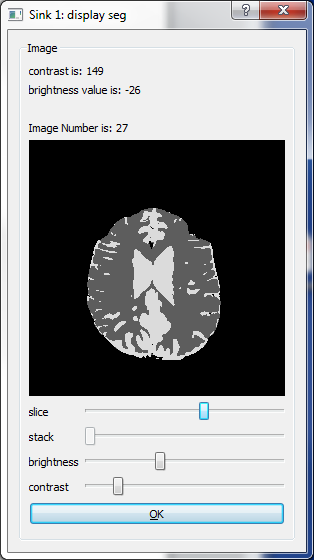
  

Fig. 7.13.1 test pipeline Fig. 7.13.2 imagesegmentation property Fig. 7.13.3 segmeted image

**Usage**: make the following settings in node property (Fig.7.13.2) before running the pipeline.

* ImageSegmentationType: Atropos
* Command Path: C:\ANTS\bin\ (full path to file atropos.exe).
* Arg1: part1 of atropos-command[5], for example (Fig.7.13.2), -d 3 -i kmeans[3] -c [15, 0.1]
* Arg2: part2 of in atropos-command[5], for example (Fig.7.13.2), -m [0.1, 10x10x10].
* Output1: full path file name of segmented image
* Output2: not used.

[5] astropos command example: 3-channel segmentation using PD, T2 and Flair ( ˽ is a space letter)

*Atropos.exe˽-d˽3˽-i˽kmeans[3]˽-c˽[15,0.1]˽-m˽[0.1,10x10x10]˽-a˽PD.hdr˽-a˽T2.hdr˽-a˽flair.hdr˽-o˽T2seg.hdr˽-x˽bet\_T2\_mask.hdr*

## **Image Slicer**

**Node function**: node IMAGESLICER can extract an volume image (3D) or a slice image (2D) from source image. A test pipeline Fig.7.14.1 inputs a 4D source image that includes two volums of 3D image with 44 slices. Node imageslicer-1 extracts a 3D image and imageslicer-2 extracts a slice from the 4D image.

**Input A1(A2)** : a numder of selected volume index.

**Input C1(C2)** : a numder of selected slice index.

**Input B1(B2)** : image data.

**Output D1(D2)** : 3D or 2D image data.

**Pipieline description:** (Fig.7.13.1)

* Extract 3D image from 4D image: set SOURCE-1 to output a number of 1 (Fig.7.13.2) that make A1=1 on node imageslicer-1. This extracts 3D image (volume-1) from the 4D source image because of no control on C1 (empty) .
* Extract a 2D image from 4D image: set SOURCE-2(Fig.7.13.3) for A2=0 on imageslicer-2 to select volume-0. Set SOURCE-3 for C2=25 on imageslicer-2. This setting extracts an image (slice) from volume-0 (3D image)
* Display 3D and 2D images on SINK 1 and SINK 2.

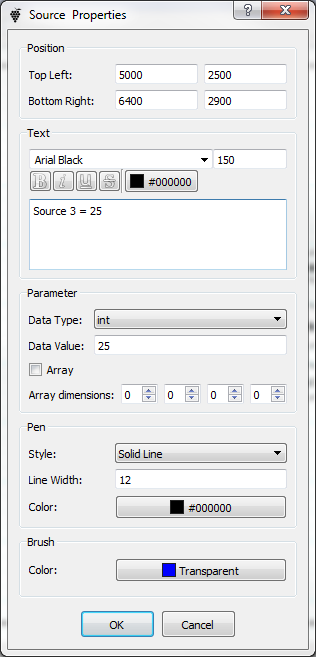
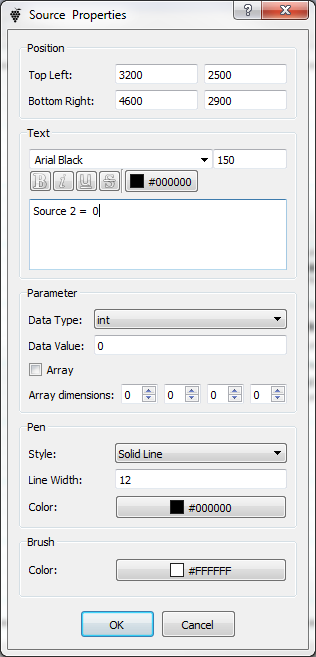
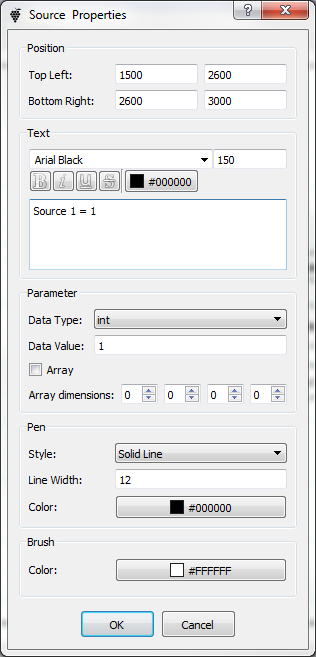
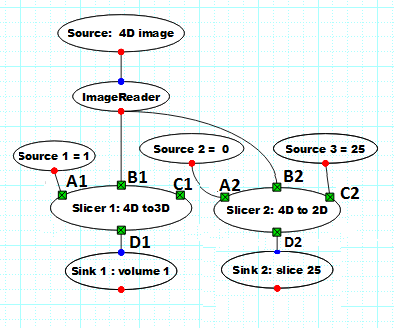


Fig. 7.14.1 test pipeline Fig. 7.14.2 source 1 Fig. 7.14.3 source 2 Fig. 7.14.4 source 3

## **Image Splitter**

**Node function**: node IMAGESPLITER can split a 4D image to two 3D images. Node SOURCE and IMAGEREADER in pipeline Fig.7.15 inputs a 4D source image (*FSE.XML/REC*) on node IMAGESPLITER that outputs 3D image (of *PD*) data on B and another (*T2*) on C to display.

**Input A** : 4D image data.

**Output B** : 3D image data.

**Output C** : 3D image data.

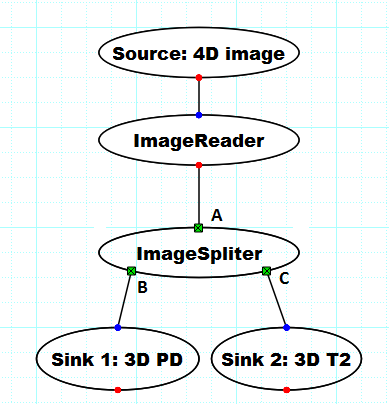
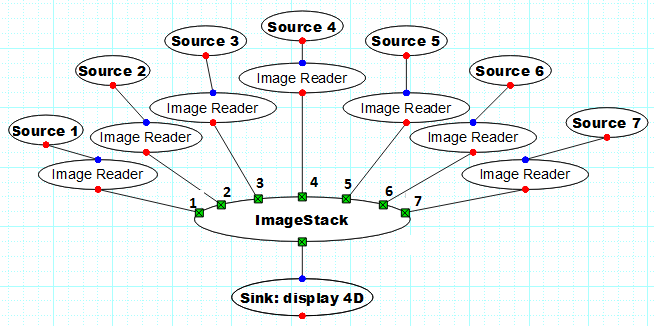
 

Fig. 7.15 imagespliter Fig. 7.16 ImageStack

## **Image Stack**

**Node function**: node IMAGESTACK stacks several (maximun 7) 3D images to a 4D image. Node SOURCE 1~7 and IMAGEREADER in pipeline Fig.7.16 input seven 3D image data on node IMAGESTACK that outputs a 4D image to display.

## **Image Statistics**

**Node function**: node IMAGESTATISTICS calculates image minimus, maximus, mean, median, standard deviation, variance and percentile (Fig.7.17.1). Node SOURCE in pipeline Fig.7.17.2 connects 3D image to node IMAGESTATISTICS that outputs a number of image mean.

**Input A** : 3D image data.

**Input B** : mask image data (option).

**Output C** : number.

## **Image Writer**

**Node function**: node IMAGEWRITER writes image data in a file (Fig.7.18.1) in four types: Analyze, DICOM, NIFTI and XML/REC that can be set in node property (Fig.7.18.2).

**Input A** : image data.

**Input B** : mask image data (option).

**Output C** : string of the output file name set by ‘Output File Path’ in node property (Fig.7.18.2).

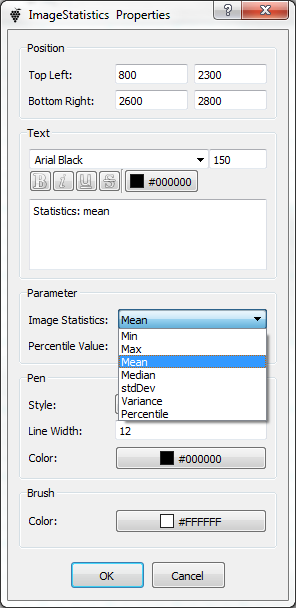
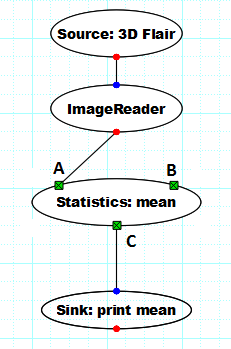
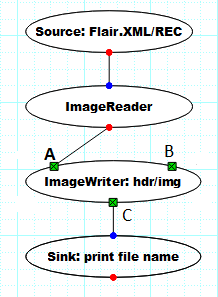
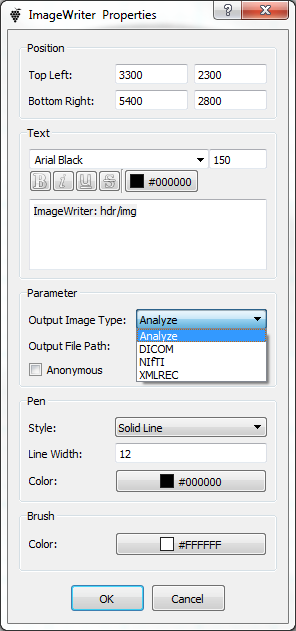
   

Fig. 7.17.1 imagestatistics Fig. 7.17.2 Fig. 7.18.1 Fig.7.18.2

## **Logical Operator**

**Node function**: node LOGOPERATION in Fig.7.19.1 performs a logical operation when its inputs A and B are boolean. It works on three logical types: AND, OR, NOT described by three sumbols &&, II and ~ at ‘LogOperation Type’ in Fig. 7.19.2. There are three kinds of logical output C listed in table 7.19.

**Node function**: node IMAGEWRITER is used to write image data in a file (Fig.7.19.1) in four types: Analyze, DICOM, NIFTI and XML/REC that can be set in node property (Fig.7.19.2).

**Input A** : image data.

**Input B** : mask image data (option).

**Output C** : string of the output file name set by ‘Output File Path’ in node property (Fig.7.19.2).

## **Loop**

**Node function**: node LOOP in test pipe line Fig.7.20.1, for example, generates a series of number for nood SELECT. Loop-parameters (start value, step and end value) are shown in node property of Fig. 7.20.4.

**Input A:** trigger to start. It can be set by any type, for example, a string in Fig. 7.20.3.

Input B: reset.

Input C: stop.

**Output D** : counter value.

**Output E** : end value.

**Output D** : counter value.

**Output E** : end value.

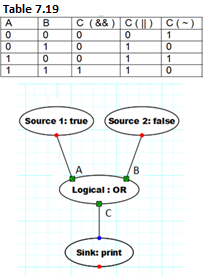
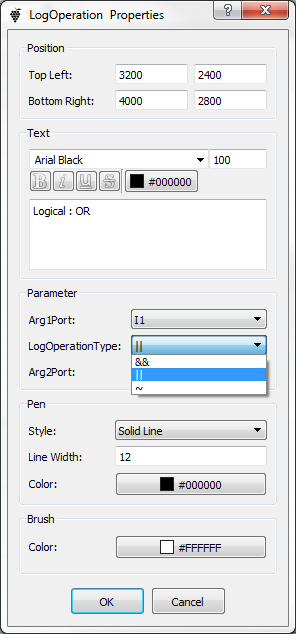
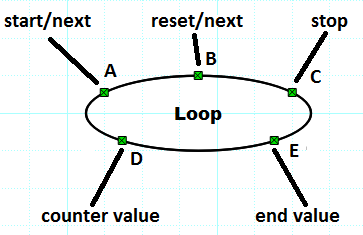
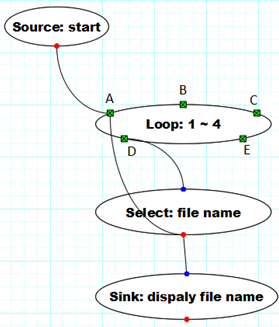
  

Fig. 7.19.1 Logical operator Fig. 7.19.2 Fig. 7.20.1 a loop pipeline Fig. 7.20.2

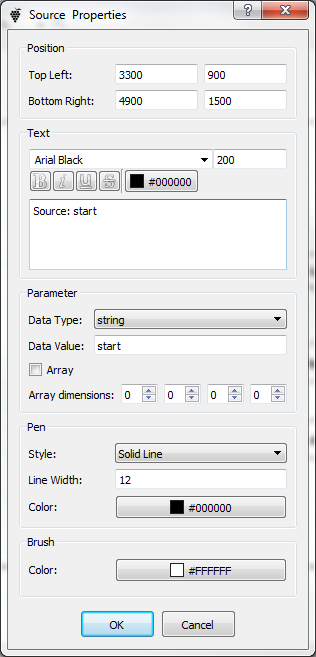
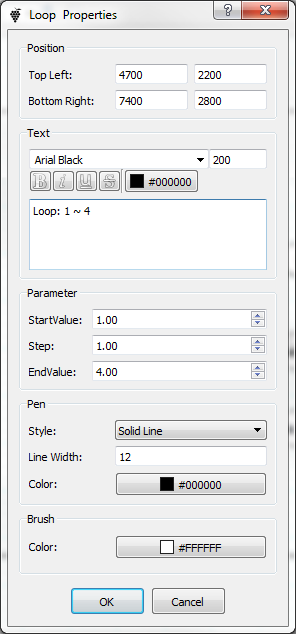
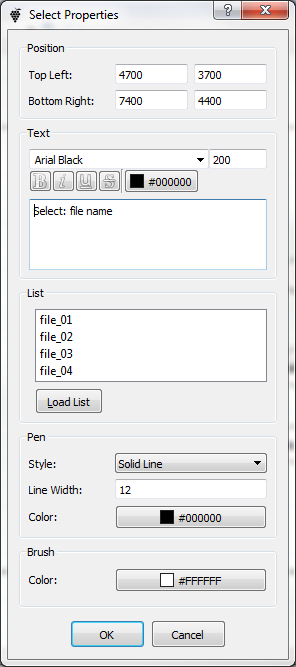
  

Fig. 7.20.3 input A Fig. 7.20.4 loop property Fig. 7.20.5 a list on SELECT

## **Parameter Reader**

**Node function**: node PARREADER(Fig. 7.21.1) reads text from a text file parfile.txt (Fig. 7.21.2) that includes 7 lines of text. Set the file name ‘parfile.txt’ in ‘Input File Path’ in node property (Fig. 7.20.3). Node PARREADER has eight output options from B to J. Fig. 7.21.1 selects B (the first text line) and D (third text line) to node PARWRITER that writes the text lines in file output.txt that was defined in Fig.7.21.4.

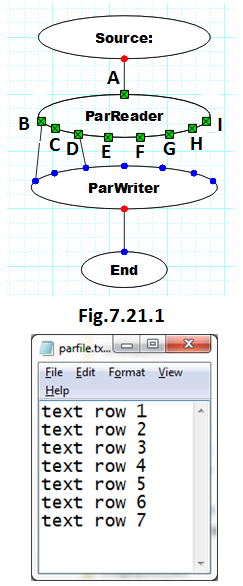
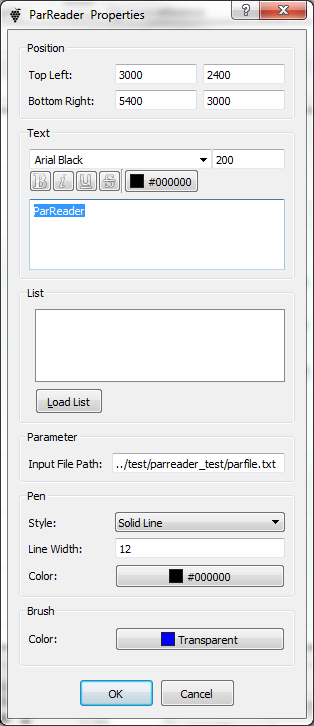
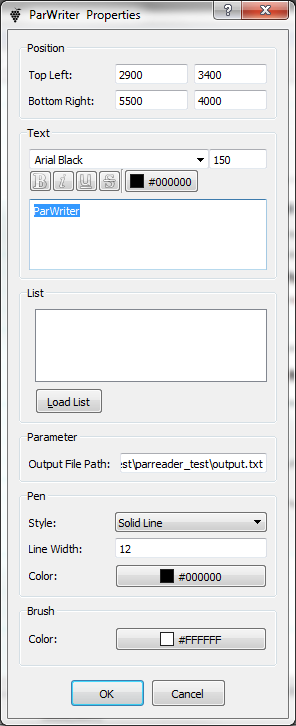
  

Fig. 7.21.2 Fig.7.21.3 parreader Fig. 7.21.4 parWriter

## **Parameter Writer**

**Node function**: node PARWRITER (in Fig.7.21.1) gets two text-strings from PARREADER on input-1 and input-3. Write them in a text file that is set in ‘Output File path’ in Fig. 7.21.4.

## **Relational Operator**

**Node function**: node RelOperator sets an relationship between its inputs A and B (Fig. 7.23.1), output a boolean result. There are seven options of relation types to select in node property (Fig. 7.23.2). For example, it outputs ‘false’ if set ‘RelOpertionType to ’>’ for A=6 and B=8 because A > B not true.

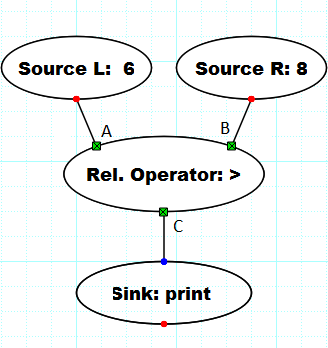
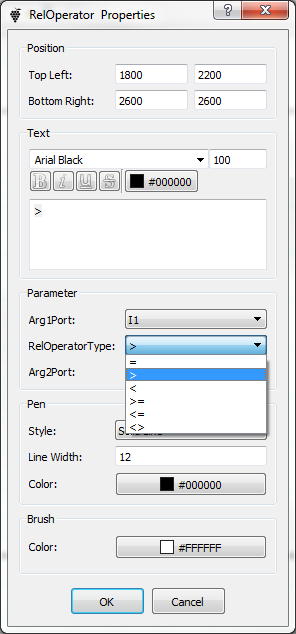
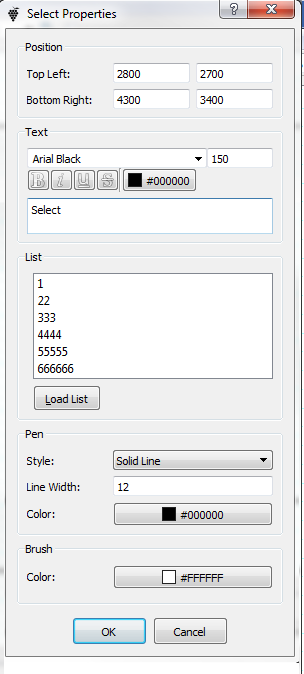
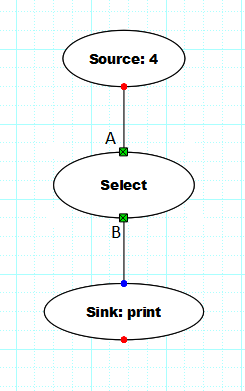
   

Fig. 7.23.1 RelOperator Fig. 7.23.2 node property Fig. 7.24.1 select Fig. 7.24.2

## **Select**

**Node function**: node SELECT has a list containing multiple text rows (Fig. 7.24.1). Set a index number on the input A (Fig. 7.24.2) to select a text row and output it. For example, node SOURCE inputs a number 4 to node SELECT that will output 4-th text row of ‘44444’ on node SINK.

## **Sink**

**Node function**: node SINK can receive an output (of string, number or image data) from a connected nodes (Fig. 7.24.2). SINK works in the following 3 types that are defined in ‘Output Type’ in Fig. 7.25.1.

* Image display: display an image if it receives an image data (Fig. 7.25.2);
* Print Text: display an number or a string if it receives number or a string;
* Write to file: write the received result in a file with a file name in ‘Output File Path’ in Fig. 7.25.3;

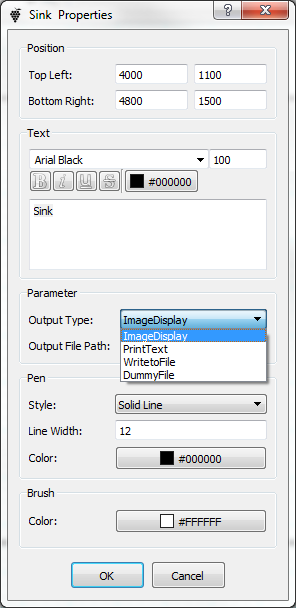
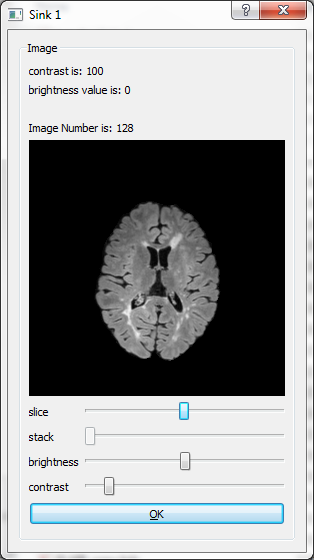
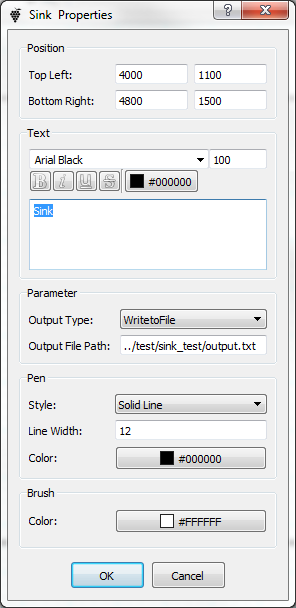
  

Fig. 7.25.1 type on sink Fig. 7.25.2 image Fig. 7.25.3 write a file

## **Source**

**Node function**: node SOURCE provides an input to the connected nodes. It has 8 data types for selection (Fig. 7.26.1). Fig. 7.26.2~4 set three data types: string, integer number and 2D random-number array.

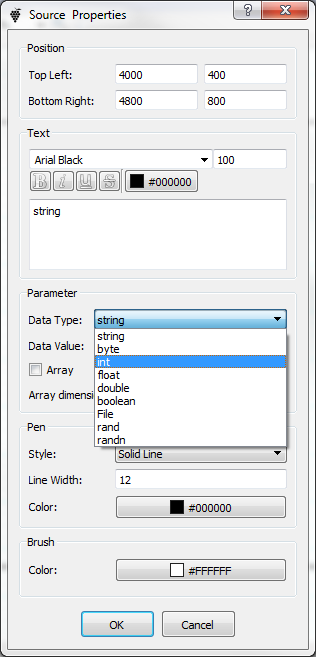
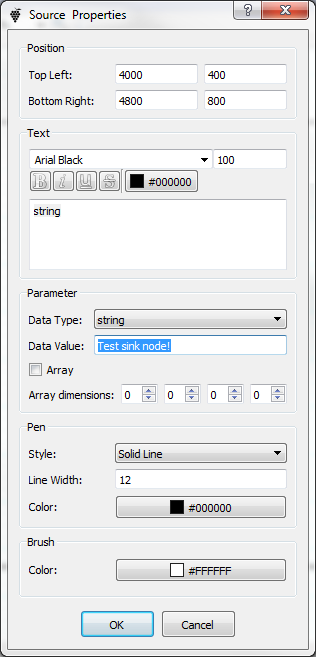
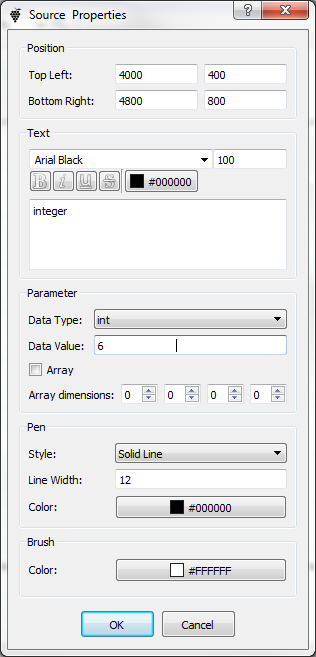
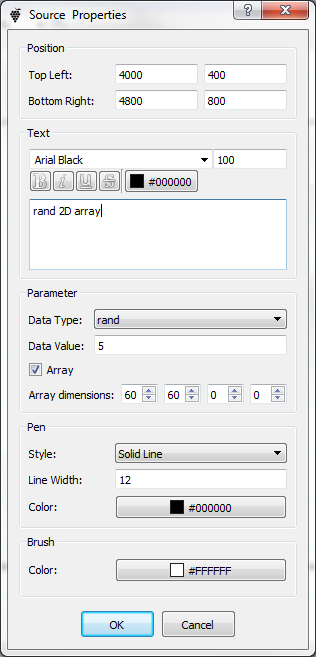
   

Fig. 7.26.1 data type Fig. 7.26.2 string Fig. 7.26.3 integer number Fig. 7.26.4 2D random array

## **Synch**

**Node function**: node SYNCH has three inputs A1~A3 and outputs B1~B3(Fig.7.27.1). It transfers A1 to B1, A2 to B2 and A3 to B3 when three inputs arrive at A1~A3 synchrously. Node SYNCH in Fig.7.27.1 waits for A2 to come 5 second late with A1 and A3 together. And then, it transfers A1~A3 (string 1, 2 and 3) to B1~B3 separately for PARWRITER to write B1~B3 in an text-file.

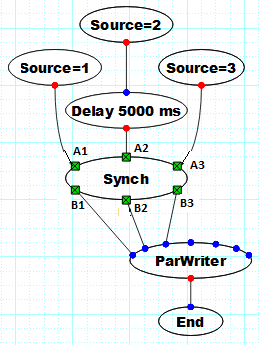
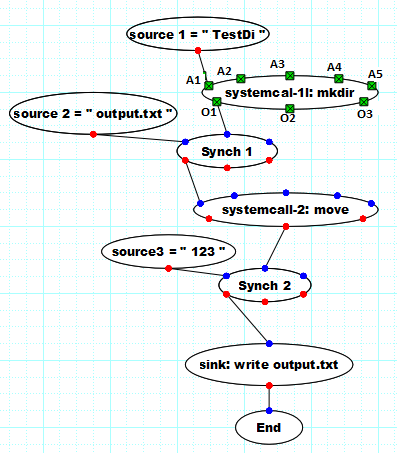
 

Fig. 7.27.1 node synch Fig. 7.28.1 node systemcall

## **System call**

**Node function**: node SYSTEMCALL is uesd to construct a command-line available to run on windows or linux system. Fig.7.28.1 shows the node SYSTEMCALL that has 5 input arguments A1~A5 and 3 outputs O1~O3. These parameters can be set up at Arg 1 ~ Arg 5 and output1~output3 in node property in Fig.7.28.2. Each of them links to port option, for example in Fig.7.28.2, including 7 selections below:

* I1~I5: input Arg1~Arg5 from input A1~A5. Agr1 Port=I1 in Fig.7.28.2 for A1=”testDir” linking to input node source 1. Output Port1=I1 for O1=”testDir”.
* fixed: enter a string in related edit-box.
* none: not used on node.

**Command**: system command. Fig.7.28.2 set the command=*mkdir* that connects to Agr1 from A1=”TestDir”. This constructs a command line *mkdir TestDir* by command + space + A1. The Command in Fig.7.28.2 can also be a batch file name and Arg1~Arg5 are able to provide parameters for use.

**Example decription** on Fig.7.28.1:

* Source 1 and systemcall-1 create a sub-folder *TestDir*, as explained above.
* After it done, source 2 inputs a file name of “*outuput.txt*” through node synch1 to systemcall-2 (Fig.7.23.3) that constructs a command line below. It moves the file *outuput.txt* to folder *TestDir*.

move *outuput.txt TestDir*/ *outuput.txt*

* Nodes source 3, synch2 and sink create a new *outuput.txt.*

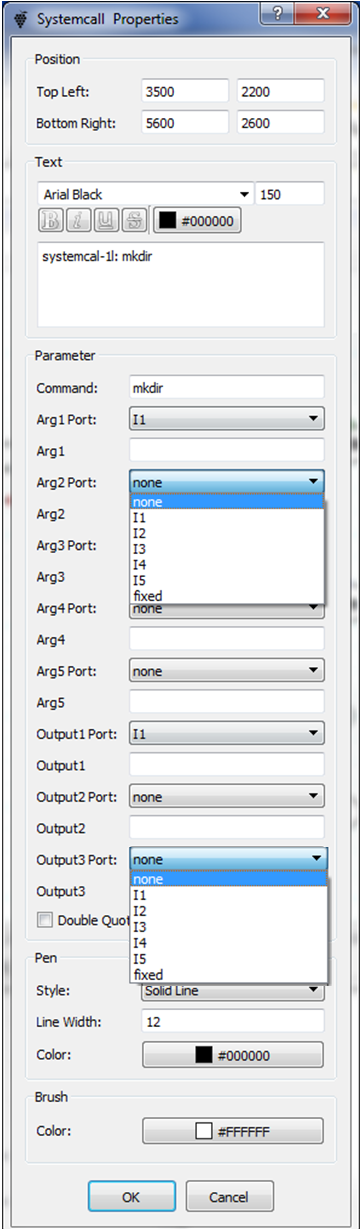
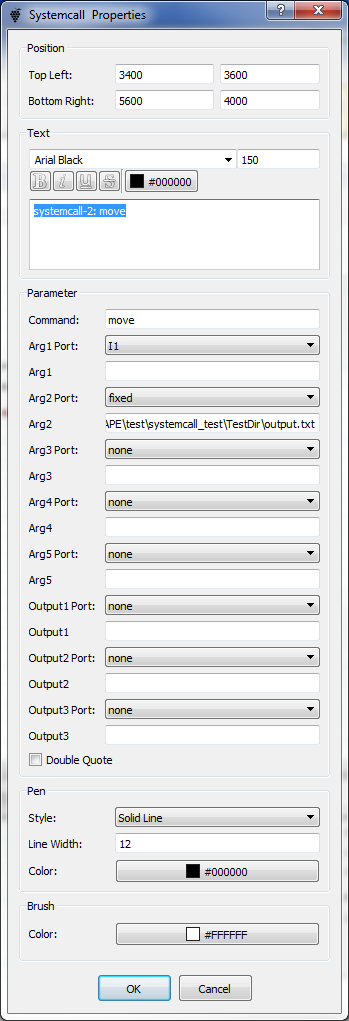
 

Fig.7.28.2 systemcall-1 Fig.7.28.3 systemcall-2