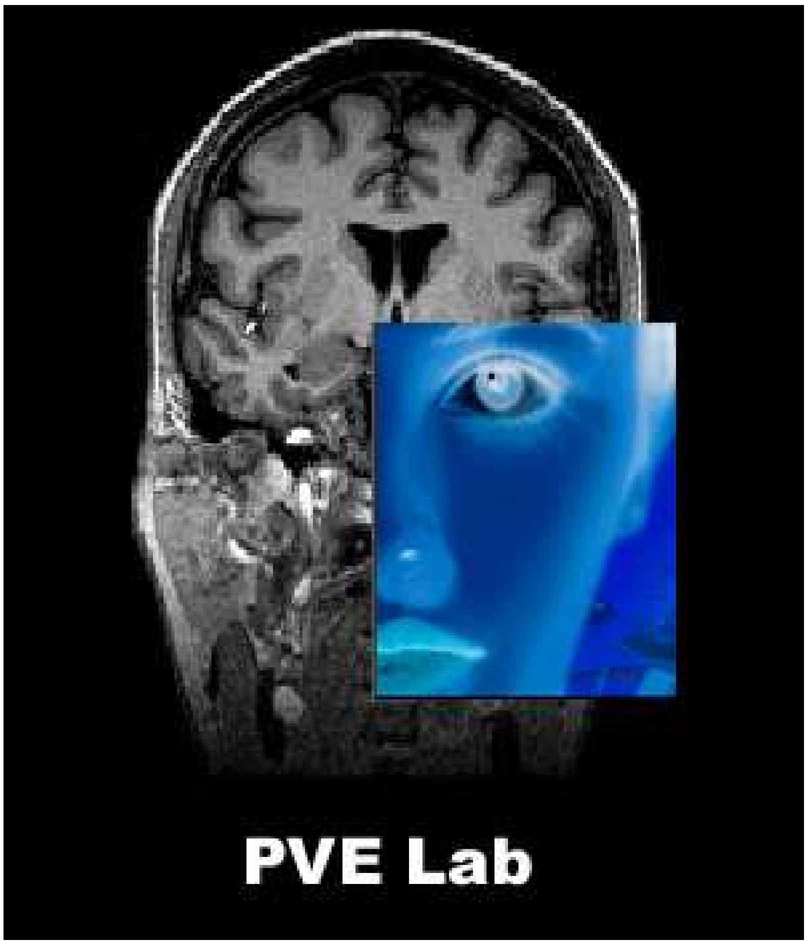

PVE Lab

## 1 Partial Volume Error

Intro.
What is PVE project
Groups implementation of methods. Tasks in a PVE correction process.

Set-up at pipeline w. given methods, log all necessary information ´structure
, and control a process…
One success criteria for the PVEout (Partial Volume Effect) project within the European 5th framework is to make a common interface for a product that can do a full partial volume correction of a PET/SPECT scan using a high resolution MR scan.
At the Firenze meeting, Italy, mid November 2002, the partners agreed to make a proposal on how the different methods in the PVE process could be integrated through a user-friendly graphic user interface (GUI). Further, the partners agreed in using the Matlab program from MathWorks Inc. for implementing the GUI.
A major advantage using Matlab as platform is the non-dependency of the operating system (Linux or Windows), the worldwide use of Matlab within medical community, and an easy way to realize the GUI.

The Copenhagen group, NRU, would make the preliminary description. The result is given in the following document where a full (preliminary) description on how an user friendly interface could be realized, code be written and, different methods integrated. Further a proposed prototype of a GUI is created in Matlab.

# 2 The pipeline program

In the following a pipeline program is introduced. The pipeline program offers an easy way to set-up a pipeline containing one or more steps in a user-defined order. A step in the pipeline is refereed as a task and within each task there exist none or more programs each realising the given. Such a program for a task is refereed as a method.

The pipeline program is based on a documented data structure where information of the individual pipeline is stored. The data structure can be seen as a read/writeable database refereed as the project for the actual pipeline. For each project a unique working directory is automatically created, where files generated through a pipeline are stored. In this way files within different projects are not mixed up and more pipeline programs can be executed at the same time.

A user-friendly interface gives a good overview of a project and makes the pipeline program easy-to-use. Some features are as followed:
- to select other of available methods than default selected
- to save user defined settings for a actual pipeline
- to load and continue a not finished project
- inspect used settings and other information for a finished project

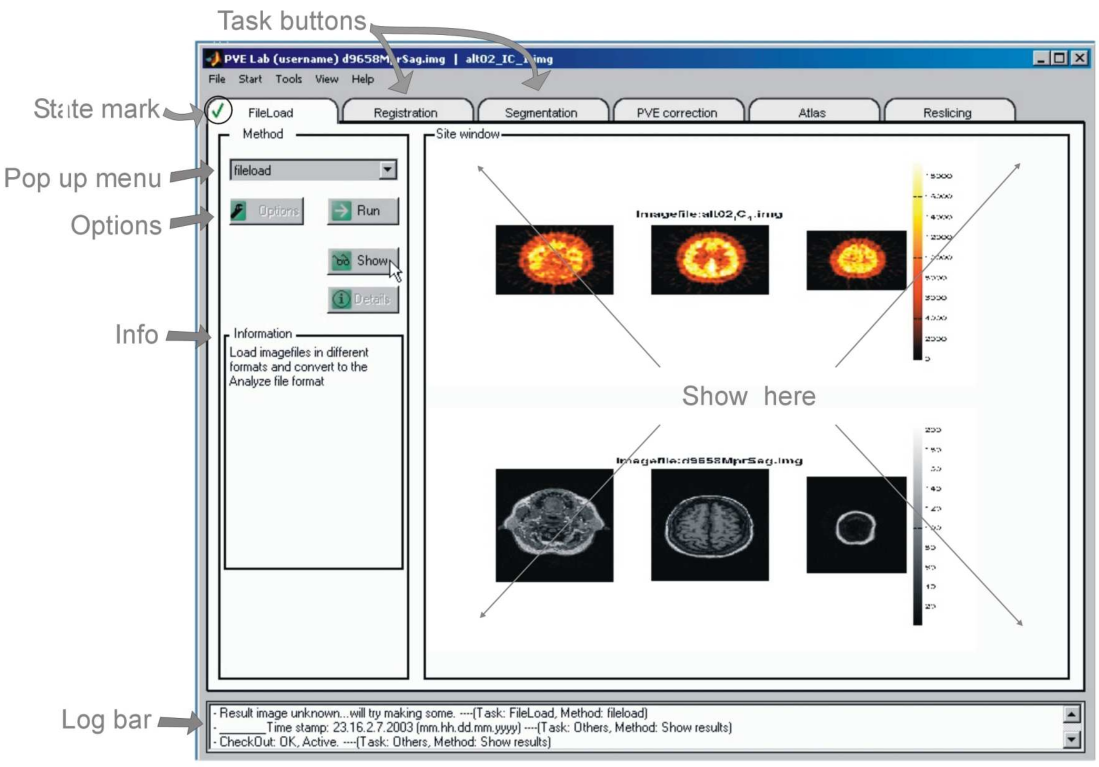
The data structure is independent of the user-interface, which means that the pipeline program in principle can be executed without any user-interface (GUI). This gives the possibility to run the pipeline program on clusters of computers. Of cause this feature expect that methods in the actual pipeline are fully automatic and either does not need any user-interface.

To set-up a pipeline two-steps must be done:
1) Each method must interface the pipeline program through a simple wrapper that shields a method from the pipeline program.
2) Create a set-up file defining execution order of the tasks and needed information of the different methods. Guideline to full-fill and examples are given.

Because …
A user implementing a new pipeline can stay focused on the given two steps, if errors are found/detected.

## *2.1 The user-interface*

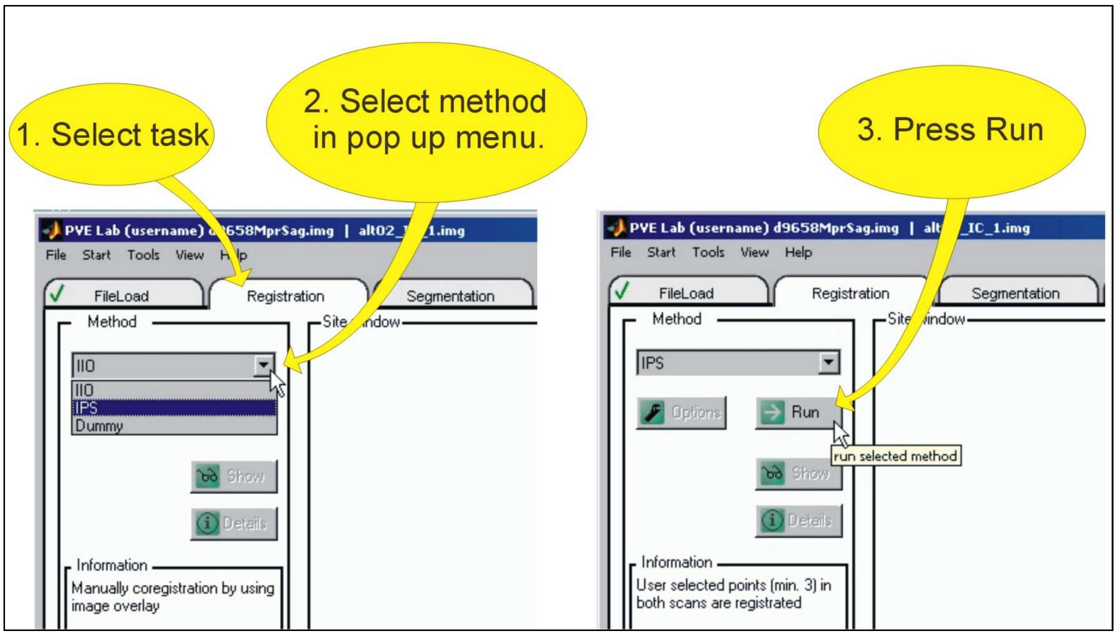
Users using a already existing set-up of a pipeline.



Load imagefiles in different formats and convert to the Analyze file format

```
- Result image unknown...will try making some. ----(Task: FileLoad, Method: fileload)
- ________Time stamp: 23.16.2.7.2003 (mm.hh.dd.mm.yyyy) ----(Task: Others, Method: Show results)
- CheckOut: OK, Active. ----(Task: Others, Method: Show results)
```

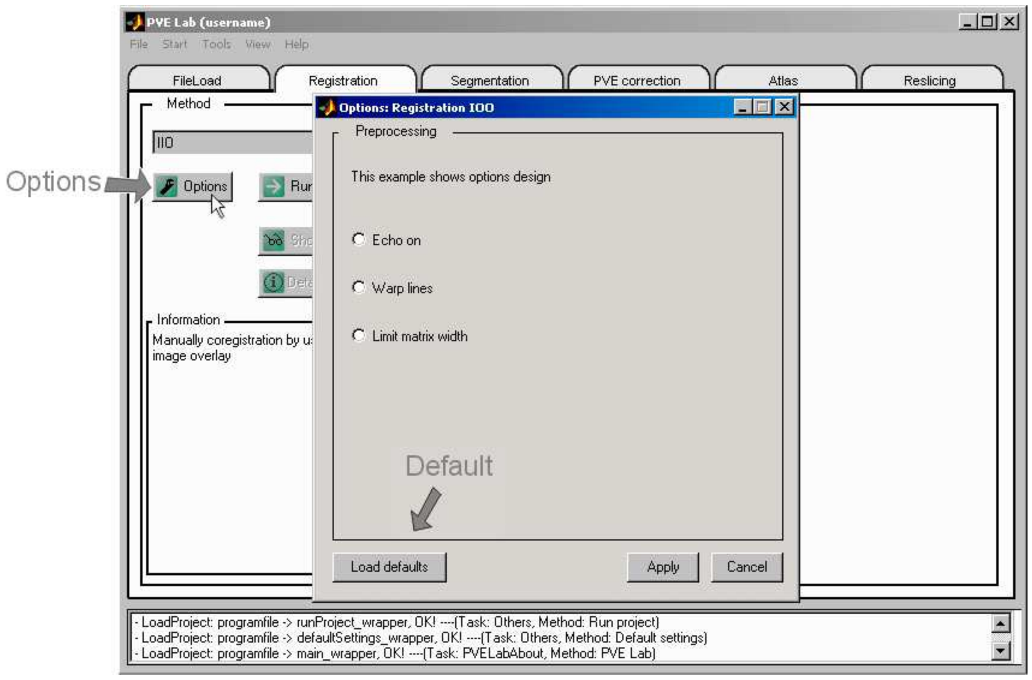
**Figure 1**



Manually coregistration by using image overlay

User selected points (min. 3) in both scans are registrated

**Figure 2**



- LoadProject: programfile -> runProject_wrapper, OK! ---(Task: Others, Method: Run project)
- LoadProject: programfile -> defaultSettings_wrapper, OK! ---(Task: Others, Method: Default settings)
- LoadProject: programfile -> main_wrapper, OK! ---(Task: PVELabAbout, Method: PVE Lab)

**Figure 3**

→✓!

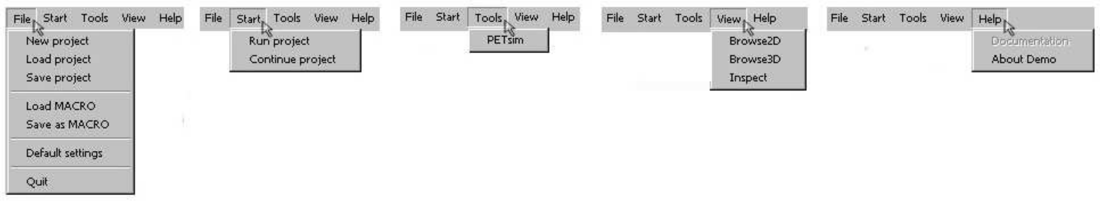


**Figure 4**

The PVElab offer methods to do a PVE correction of a low resolution PET or SPECT scan (functional image) given a high resolution MR scan (structural image).

This demo presents our ideas behind the PVElab design. This document explains how the design and layout of the GUI should work.

The PVElab GUI is build about tasks. Each task is linked to a task button in the top. Each task can be accomplished by a number of methods. Registration for example can be done by IIO, IPS etc.

The results of a method are displayed in the right frame. The idea with PVElab is that it displays the selected method and offers a possibility to quickly inspect the results. If more interaction from the user is required, a new figure can be created.

The method is selected from the pop up menu. The info field displays clues to guide the user through all the tasks. When a method is selected a short method description is displayed in the info field.
A log is also created, so you can see and document how your results were obtained. You can keep an eye on the log because it is also displayed at the bottom of the figure.

Pressing the 'Options' button can configure each method. In the options figure it is possible to reset the method to default settings.

The tasks display their state by a small symbol. The states are I'm next  - Completed - Error .

You can save all your work in a project. This project contains all the settings for all tasks. When you load your project you can change some options and run the calculations again.

A summary of the current menu structure:

## *2.2 The project*

As introduced each project contain a data structure where all information for a pipeline are stored. Information regards settings for the tasks and available methods in the pipeline, results, log information and status of the pipeline process.

In the following section all fields of the data structure in a project will be explained shortly. This section is intended for programmers who want to implement new tasks and/or methods. All so users who want a deeper understand of the data structure or like to retrieve extended information from the project.
How to write wrapper functions that interface to the pipeline program and setting up pipelines, see section **Error! Reference source not found.**.

### 2.2.1 Restrictions using project

The data structure of a project is in general a READ-ONLY structure, unless else is given and a field is marked with a start (*) in the following sections. A pipeline in the pipeline program is a collection of different tasks and methods implemented by different programmers. It is therefore obviously, not respecting given restrictions may cause to instability, not generality and misunderstanding using the pipeline program.

Do not store larges data amounts in the project such as images. Matlab controls the project where pointers not are available. The data structure is therefore transferred between functions, each making a copy of the project. This can make the pipeline program slow and using unnecessary memory if lot of data is stored in the project.

### 2.2.2 Syntax and indices

To navigate in a project one must understand the basic syntax of a data structure and used indices.

A data structure is build up of blocks containing one or more fields of those illustrated in Figure 5.

Type of fields:

- field0: Contain one or more given type of fields: field0 to field4.
- field1: One variable of type string, number or a field.
- field2(index): Vector a specific type of variables, as numbers or strings with the same length.
- field3{index}:Vector of none-specific type of variables or fields.
- field{index1,index2}: As field3, but with an extra dimension.

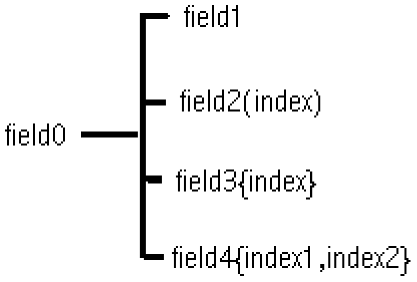


**Figure 5**

To access field1 to field4 in structure field0 do:
Reading a field into variable A:

*A=field0.field1;*
*A=field0.field2(index);*

Writing variable to a field:

*field0.field1=A;*
*field0.field2(index)=A;*

Used indices navigating a project in the pipeline program:

- **Index**: not a specific index.
- **TaskIndex**: Selected task of a defined order in the pipeline. First task is 1
- **MethodIndex**: Selected method of available methods in a task: First method is 1
- **ModalityIndex**: Selected modality of available modalities: First modality is 1
- **ImageIndex**: Selected image of available images: First image is 1 (default)
- **FunctionIndex**: Selected function of available functions: First function is 1

### 2.2.3 The project field

The project has four fields each containing a structure, which are explained in the following sections.

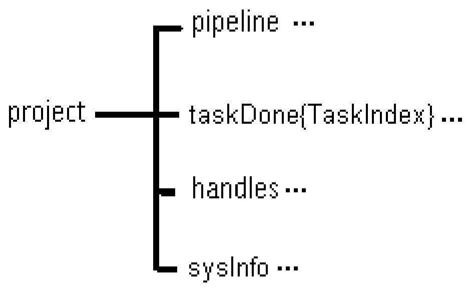


**Figure 6**

<u>Fields of the project:</u>

- **pipeline**: Contain both settings for the used pipeline and status information of the pipeline process.
- **taskDone{TaskIndex}**: Information of a finished tasks in the pipeline process.
- **Handles**: Handles/pointers to used figures in the pipeline program.
- **sysInfo**: General information of used directories and files.

### 2.2.4 The pipeline field and sub-fields

The pipeline field contain all information regarding set-up and process status and control of a pipeline. The field *taskSetup{TaskIndex,MethodIndex}* contain a sub-field of settings for the given method of a task. This *task{,}*-field sets up a given pipeline and is therefore refereed as a *set-up file.* The set-up file is an input parameter when starting the pipeline program.
The other fields in the pipeline field are mainly process status and control of a pipeline.

The pipeline field is shown in Figure 7 and a short description of each field is given in the following.

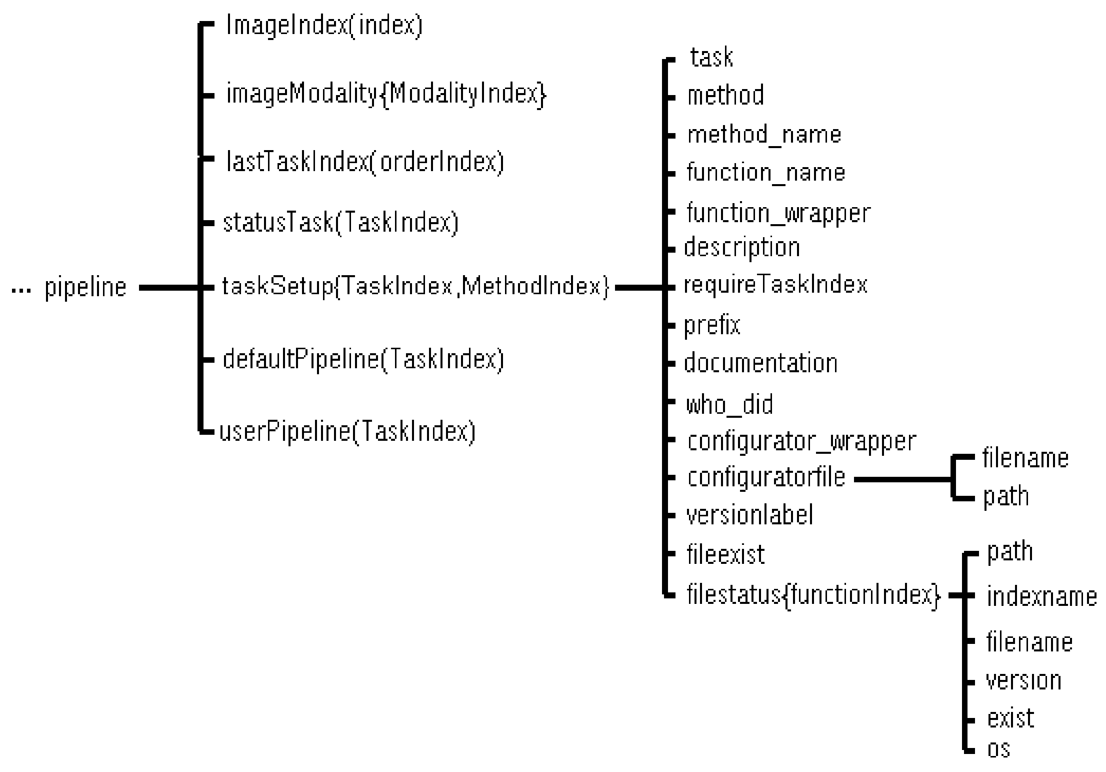


**Figure 7**

<u>Fields of pipeline:</u>

- **LastTaskIndex(Index)**: Vector containing the index to the last succeeded task in an increasing order.
- **StatusTask(TaskIndex)**: Vector containing a status-flag of each task: 0= Ready, 1=Active/runing and 2=Done (if succeed)
- **DefaultPipeline(TaskIndex)**:Vector containing index to the method which is by default chosen when PVElab is started.
- **UserPipeline(TaskIndex):** Vector containing index to the methods used in the pipeline of a PVE correction process. A user could have changed the default method indices.

- **taskSetup{TaskIndex,MethodIndex}**: Cell-array containing a sub-entry to a setup structure of a given method in a given task.
  - **task**: Task that method belongs to. The task name will appear in the pipeline program.
  - **method**: Name of the method (e.g short-term) to appear in the pipeline program.
  - **method_name**: Full name of method.
  - **function_name**: Program/function realising the method.
  - **function_wrapper**: Wrapper that interface method and the pipeline program.
  - **description**: Information of method, will appear in the pipeline program (information window).
  - **requireTaskIndex**: Index of which tasks has to be finished before current task.
  - **prefix**: Added to output files of actual task.
  - **documentation**: Text or link how to get further documentation of given method. Will appear in 'about menu' of the pipeline program.
  - **who_did**: Name of group and year. Will appear in 'about menu' of the pipeline program.
  - **configuratorfile:** (if exist) text file with settings for actual method
    - **filename**: Name of file
    - **path**: path to file
  - **configurator_wrapper**: (if exist) Wrapper to interface a configurator and the pipeline program.
  - **versionlabel**: Text label identifying version number of given method in a matlab file.
  - **fileexist**: Ready flag if all needed functions in the task field are found by the pipieline program. 1=exist , 0=do not exist and method can not be executed.
  - **filestatus{functionIndex}**: File status of functions for a given method in the current field of tasks{,}, which should be available to the pipeline program.
    - **path**: Where function is found
    - **indexname**: Field name
    - **filename**: Name of file for the actual function.
    - **version**: Found version label.
    - **exist**: If file exist = 1 else = 0.
    - **os**: Actual operation system

### 2.2.5 The taskDone field and its sub-fields

Information for each finished task in the pipeline is stored in this field. This regards from selected method, settings and program files. If an error is detected the field is cleaned up and deletes all generated files, except input and program files.

In the field 'userdata' additional user information can be written either into a variable or a sub structure. All other fields are read-only.

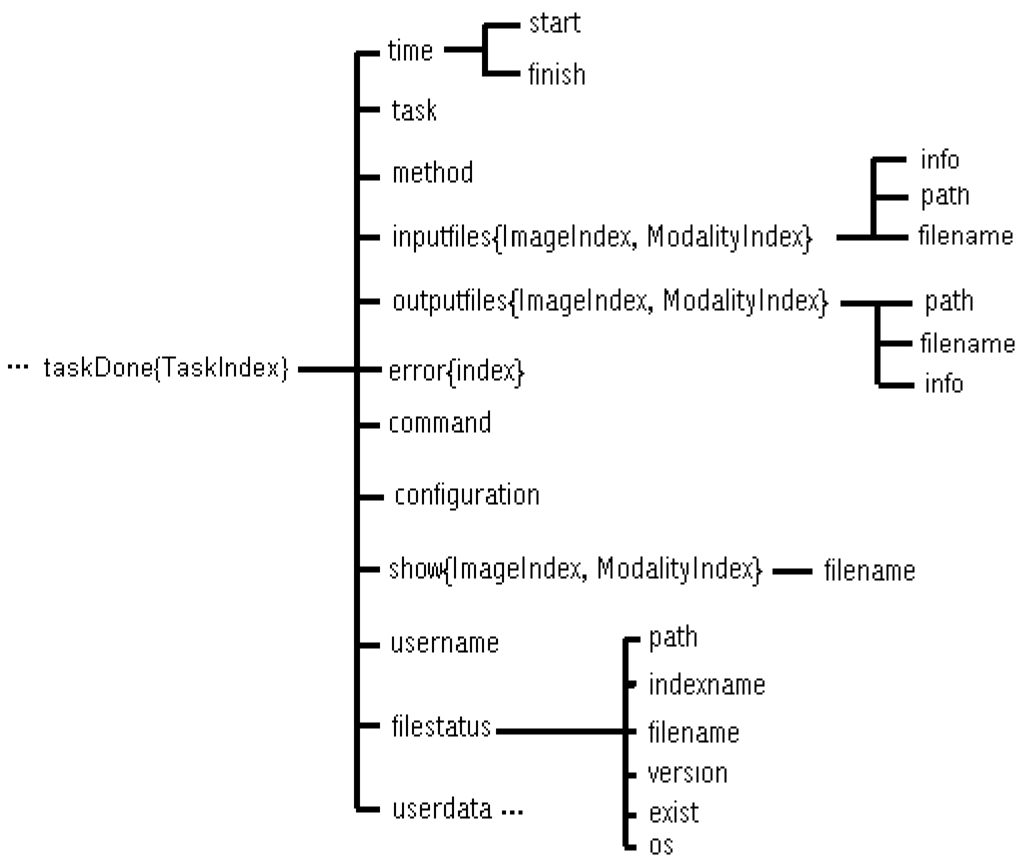


**Figure 8:**

<u>Fields of taskDone{TaskIndex}:</u>

- **Username**: username of logged in the operation system.
- **time**: Evaluation time for given method.
  - **start:** Start time(yy.dd.hh.mm.ss)
  - **finish:** End time (yy.dd.hh.mm.ss)
- **task**: Name of task given in: *project.pipeline.{,}.task*
- **method**: Tag of selected method given in: *project.pipeline.{,}.method*
- **inputfiles{ImageIndex,ModalityIndex}** (filenames are automatic made before the method is executed)
  - **filename**: header (*.hdr) or image (*.img) filename of input image, given a modality. Expected image format: Analyze.
  - **Path:** path of input image file of given modality.
  - **Info:** (optional user information) Additional text information of image.
- **outputfiles{ImageIndex,ModalityIndex}**:(file names are automatic made)

  - **filename filename**: header (*.hdr) or image (*.img) filename of output image, given a modality. Expected image format: Analyze
  - **path**: path of input image file of given modality.
  - **Info:** (optional user information) Additional text information of image.
- **error{index}:** cell-array of detected errors. Detected errors will clean up the field and delete generated files. For a finished task the error field is empty.
- **command**: A commands given to the method
- **configuration:** Used configuration parameters: NOT READY/IMPLEMENTED.
- **show{ImageIndex,ModalityIndex}**:(file names are automatic made) Show snapshot of the output data.
  - **filename**: Images (*.bmp,*jpg or like) that gives a snapshot of the output result from given method.
    Note: The images are shown in the Result window of the pipeline program. These images are not made by the pipeline program, but by the programmer for the given method.
- **filestatus{functionIndex}**: File status of functions for a given method in the current field of tasks{,}, which should be available to the pipeline program.
  - **path**: Where function is found
  - **indexname**: Field name
  - **filename**: Name of file for the actual function.
  - **version**: Found version label.
  - **exist**: If file exist = 1 else = 0.
  - **os**: Actual operation system: Linux, Windows and ect.
- **userdata**: A user defined entry for the actual method. Here additional information can be stored as sub-entries the user needs or like to handle to the following tasks in the pipeline.

### 2.2.6 The handles field

Handles to axes that give the possibility for a method to present the user for progressing data or show images in the user interface of the pipeline program. If the pipeline program is executed without the user-interface the handles will not be available but empty.

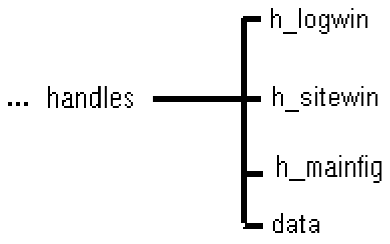


**Figure 9:**

Fields of handles:

- **h_logwin:** Handle to the log window where log information are presented to the user.
- **h_sitewin**: Handle to the site window, where progress data and images of results can be presented to the user.

- **h_mainfig**: handle to the main figure of the user-interface, called mainGUI.

### 2.2.7 The info field

In this field system information for the loaded pipeline and for the pipeline program are found.

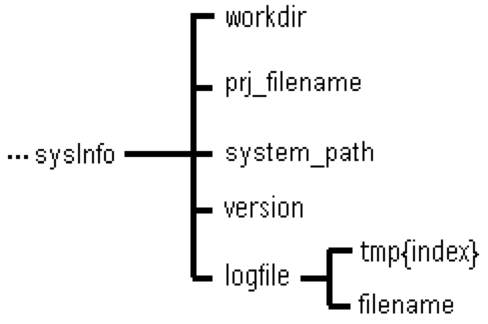


**Figure 10:**

<u>Fields of sysInfo:</u>

- **workdir**: Directory where all files generated in a pipeline process are saved.
  Note: A *workdir* is automatically created for each new project when the first task in the pipeline is finished.
- **systemdir**: Directory where system files for the pipeline program are placed.
- **Prj_filename**: filename of project with extension '**.prj*', saved in *workdir*.
- **version**: version number of the loaded pipeline set-up file.
- **logfile**: A log file exist for each project, and saved in the *workdi*r with extension '**.log*'. Log information are transferred to the log bar via *project.handles.h_log* if the user-interface is used.
  – **tmp{index}:** Temporary cell-array of log data is saved (added) to the log filename when a method has finish.
  – **filename**: name of log file with extension '**.log*', saved in *workdir*.

## 3 Realising a pipeline

Before continuing into deeper levels of the pipeline program where the process, and set-up of a pipeline in the pipeline program are explained, a summarize is given.

Preparation must be done to set-up a pipeline in the pipeline program. First define name and order of the tasks in a pipeline. Then within each task, name and program of available methods is considered and methods tested. In section XX a preparation guideline can be found

After preparation the next step is implementation of:
- A set-up file to initialise a pipeline in the pipeline program.
- Wrappers that interface the pipeline program and a given method.

- Wrappers that interface the pipeline program and a configurator for a given method.

## 3.1 Matlab and the pipeline program

Varargin, varargout

## 3.2 Connection diagram of the pipeline program

In the following the connectivity between user-interface, wrappers and files in the pipeline program are explained and all are referenced to Figure 11.

Robustness, transparency and generality are key words of the pipeline program.
- Robustness: The pipeline program is not affected by a crashing method.
- Transparency: All necessary information regards set-up, used settings and log information are automatic collected and are fully available within the pipeline program.
- Generality: The pipeline program is easy to use, independent of operation systems, and different type of executable realising a method can be used.

A wrapper is a function, which interface to/from the pipeline program. Basically two type of wrappers exist in the pipeline program: the *main_wrapper* and the *function_wrapper / configurator_wrapper*.

The user-interface is handled by a function called *mainGUI*, section 2.1. Here handles to the site and the log window are given, in this way progressing data, results and log information can be presented for the user.

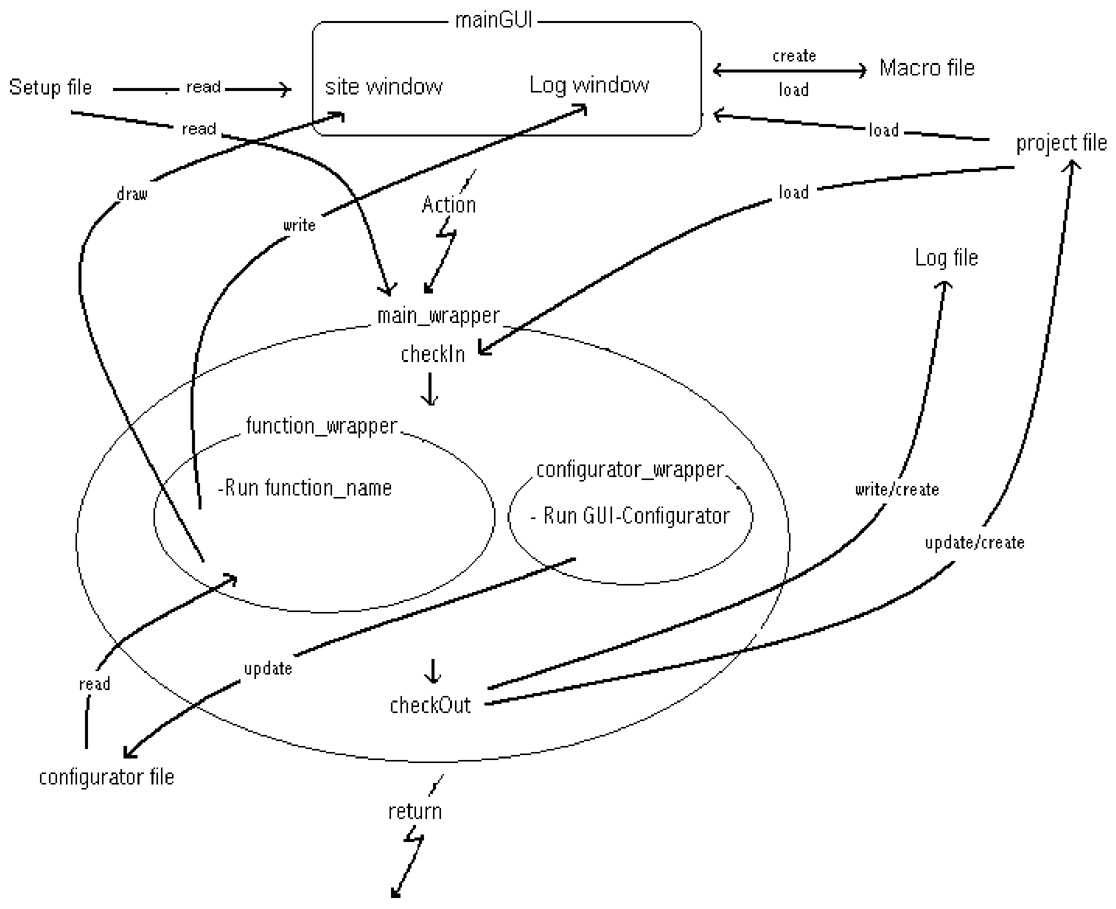


**Figure 11:**

### *3.3 A function_wrapper*

The function wrapper is responsible to make the pipeline program general in a way so that different methods can be implemented without confliction. It is typically called by the main_wrapper, but also called by another function-wrapper. (Figure 11) Some methods are very alike in this case the same function-wrapper can be used.

The only restrictions making a function wrapper are those for reading/writing the project (see section 2.2.1), and that a number of input- output arguments are fixed:

*project =MethodName_Wrapper(project,TaskIndex,MethodIndex,varargin)*

Where *projec*t is the data structure and the indices are refereed to the field of the project with the set-up for the actual method given in *project.pipeline.taskSetup{TaskIndex,MethodIndex}* (section 2.2.4). *Varargin* holds an arbitrary number of user input arguments. These are only understood by the given function.

The syntax for programming a function wrapper:

1) Retrieve information from the project such as input/output names which are automatic made, given *project.taskDone{TaskIndex}*

2) Prepare configuration of the method either using

*project.pipeline.taskSetup{TaskIndex,MethodIndex}.configuratorfile* or an other field in the data structure.

3) Execute the method found in:
   *project.pipeline.taskSetup{TaskIndex,MethodIndex}.function_name*
   OR
   Insert code instead of a function. This require in the *taskSetup{,}* field in the project that *function_name=function_*wrapper for the given method.

4) User data can be added to the project: *project.taskDone{TaskIndex}.userdata*, or the available standard fields in the project as given in section 2.2.5.

An example on how programming a function wrapper is shown in section 4.1.

## 3.4 A configurator_wrapper

Some methods can be configured with different settings. A configurator user-interface can be set-up, so the user has the possibility to change the settings, an example is shown section 2.1. This user-interface is not a part of the pipeline program, but can be started by the configurator_wrapper. New settings will be used next time the method is executed if they are stored in the data structure of the project *project.pipeline.taskSetup{TaskIndex,MethodIndex}.configuratorfile.* Note: It should always be possible to restore default settings through the configurator user-interface.

The syntax for making a configurator wrapper:
1) jk
2) fg
3) ghj
4) fg

An example on how to make a configurator wrapper is shown in section 4.2.

## 3.5 The main_wrapper

The robustness of the pipeline program is insured by the main_wrapper. The main wrapper is executing programs by refereed to in the taskSetup by the indices TaskIndex and MethodIndex.  field found in *project.pipeline.taskSetup{TaskIndex,MethodIndex},* see section 2.2.4 for possible functions and programs.

In a pipeline it is important to control process, so two methods can not be executed at the time, the program is available, the input arguments exist etc. etc.
The process
To control the process in a pipeline

### 3.5.1 checkIn

### 3.5.2 checkOut

### *3.6 The mainGUI*

### *3.7 Preparation: A task guideline*

Tasks defining tasks and order of a pipeline…
Task named ‘Others’: typical functions linked to a menu…
Task ‘end’: Always last task: Information of actual pipeline…

### 3.8 Preparation: A method guideline

In the following a guideline is given when preparation which to full file the set-up file is given

**NOTE:** By default it is expected**:**

- That a method reads and save input/output MR- PET-images in the Analyze format.
- That the orientation between MR and PET is the same.
- That the function realising a method can be executed on Linux-, Unix- and window platforms.

**Method:** Tag name of method name to appear in the pipeline program.

**Method_name**: Full name of method.

**Task:** Name of the task the given method belongs to/ is realising.

**Version label**: A text string to identify the version number of the method in a Matlab file.

**Description**: Information given via the information window to the user about selected method and its special features.

**Documentation**: Article or where (e.g. www) the method is published.

**Who did**: Who did the method (group), city and year.

**Require task index**: Tasks to be done before method may run.

**Function name:** Name of the function that is realising the method. The function is executed within the Matlab environment. The same name for the function and method can be used.

**Function syntax**: Syntax for the function.

**Input arguments**: Description of the input parameters.

**Output arguments**: Description of the output parameters.

**Code type**: Matlab, executable or like to be executed from Matlab.

**Configuration**: (NOT READY…under construction) If exist, the configurator will be used to set-up/initialise the method before execution. Further the configurator gives the user the possibility to change the set-up through a user-interface.

# 4 How to…

### *4.1 How to write a function wrapper*

In following are given an example of a function_wrapper called *registrate_wrapper*. The function_wrapper is used in the co-registration task, and the same function is used for both the IIO and the ISP methods realising a co-registration task.

Note that in this example the filename and path of an AIR-file, containing the rigid transformation matrix after co-registration, is saved in the sub-entry *userdata* of the project. All other information to set up the  IIO or the ISP is given in the structure of the project.

```
function project=registrate_wrapper(project,TaskIndex,MethodIndex,varargin)
% function_wrapper for the two registration methods: IIO_method and IPS_method
%
% Input:
%   project: Structure for actual PVE correction process
%   TaskIndex: Index in the project-structure for actual task
%   MethodIndex: Index in the project-structure for actual method
%   varargin: Extra input arguments, not used
%
% Outout:
%   project: structure for actual PVE correction process
% ____________________________________________
% T. Dyrby, 170303, NRU
%
%SW version: 170303TD

%_______ Check if registration function is returning data...
if(~isstruct(project))
   % No project structure is given!!
    return
end%

%_______ Add to project: AIR file where to store registration matrix
name=project.taskDone{TaskIndex}.inputfiles{1,1}.name;
[tmp,AIRfile,extAIR,tmp]=fileparts(name);

project.taskDone{TaskIndex}.userdata.AIRfile.name=[AIRfile,'.air'];
project.taskDone{TaskIndex}.userdata.AIRfile.path=project.info.workdir;%Save in workdir

%_______ Init matrix to Coreg function. Defined in header of 'Coreg.m'
name=project.taskDone{TaskIndex}.inputfiles{1,1}.name;
path=project.taskDone{TaskIndex}.inputfiles{1,1}.path;
files.STD=fullfile('',path,name);

name=project.taskDone{TaskIndex}.inputfiles{1,2}.name;
path=project.taskDone{TaskIndex}.inputfiles{1,2}.path;
files.RES{1}=fullfile('',path,name);

name=project.taskDone{TaskIndex}.userdata.AIRfile.name;
path=project.taskDone{TaskIndex}.userdata.AIRfile.path;
files.AIR=fullfile('',path,name);

%_______ Possible to load a default registration matrix
files.A{1}=eye(4,4);% Initial registration matrix

%_______ Get handle to registration method only used for UIWAIT
h_registration=feval(project.pipeline.taskSetup{TaskIndex,MethodIndex}.function_name);
set(h_registration,'Visible','off');

%_______ Call registration method
parent=project.handles.h_mainfig;% Parent handle
ReturnFcn=project.pipeline.taskSetup{TaskIndex,MethodIndex}.function_wrapper;% Where to return afterwards
visualizer=project.pipeline.taskSetup{7,3}.function_name;% Which method to browse:Could be in a configurator...

feval(project.pipeline.taskSetup{TaskIndex,MethodIndex}.function_name,'Batch',h_registration,files,parent,ReturnF
cn,visualizer);
%Coreg('Batch',h,files,parent,ReturnFcn,,visualizer);
% OR
%Registrate('Batch',h,files,parent,ReturnFcn,visualizer);

%____ Wait until function (figure object) return (is finished)
uiwait(h_registration)
```

**Figure 12 example of a implemented function_wrapper, called *registrate_wrapper*, which belong to the co-registration task. The same function_wrapper is used for the two methods realising the co-registration: *Coreg(…)* and the *Registrate(...)*.**

## *4.2 How to write a configurator wrapper*

## *4.3 How to make a set-up file*

```
function tasks=setupPVELab;
% Setting up the user defined information about
% the different methods and which task it belongs.
%
% The information is stored in the project structure:
%  project.pipeline.tasks{TaskIndex,MethodIndex}
%
%___________________________________________
%SW version: 170303TD, T. Dyrby, 120303, NRU

%__________ Add image information
pipeline.imageModality={'Structural','Receptor'};
pipeline.imageIndex=1;

%_________________________________________________________
%_________________________TASK: FILELOAD
TaskIndex=1;% FileLoad
MethodIndex=1;% FileLoad
tmpTask.task='FileLoad';

tmpTask.method='fileload';
tmpTask.method_name='FileLoad';
tmpTask.function_name='FileLoad_wrapper';
tmpTask.description='Load imagefiles in different formats and convert to the Analyze
file format';
tmpTask.require_taskindex=0;%0=no tasks has to be done
tmpTask.function_wrapper='FileLoad_wrapper';
tmpTask.prefix='';
tmpTask.documentation='No available';
tmpTask.who_did='NRU, 2003';
tmpTask.configurator_wrapper='';
tmpTask.configuratorfile.name='';
tmpTask.configuratorfile.path='';
tmpTask.versionlabel='SW version';
taskSetup{TaskIndex,MethodIndex}=tmpTask;

%_________________________________________________________
%______________________TASK: Registration
TaskIndex=2;% Registration
MethodIndex=1;% IIO
tmpTask.task='Registration';

tmpTask.method='IIO';
tmpTask.method_name='Interactive Image Overlay';
tmpTask.function_name='Coreg';
tmpTask.description='Manually coregistration by using image overlay';
tmpTask.require_taskindex=1;
tmpTask.function_wrapper='registrate_wrapper';
tmpTask.prefix='r_';
tmpTask.documentation='None available';
tmpTask.who_did='NRU, 2003';
tmpTask.configurator_wrapper='';
tmpTask.configuratorfile.name='';
tmpTask.configuratorfile.path='';
tmpTask.versionlabel='SW version';
taskSetup{TaskIndex,MethodIndex}=tmpTask;

%___________________________________________________________
TaskIndex=2;% Registration
MethodIndex=2;% IPS
tmpTask.task='Registration';

tmpTask.method='IPS';
tmpTask.method_name='Interactive Point Selection';
tmpTask.function_name='Registrate';
tmpTask.description='User selected points (min. 3) in both scans are registrated';
tmpTask.require_taskindex=1;
tmpTask.function_wrapper='registrate_wrapper';
tmpTask.prefix='r_';
tmpTask.documentation='None available';
tmpTask.who_did='NRU, 2003';
tmpTask.configurator_wrapper='';
tmpTask.configuratorfile.name='';
tmpTask.configuratorfile.path='';
tmpTask.versionlabel='SW version';
taskSetup{TaskIndex,MethodIndex}=tmpTask;
```

**Figure 13. Example of setting up a method for a given task. Here is shown: the first task:'Fileload' with one metod:'fileload'. And the second task:'Registration' with two methods:'IIO' and 'IPS'.**

## *4.4 How to make a method guideline*

**Method:** IIO

**Method name**: Interactive Image Overlay

**Task**: Co-registration

**Version label**: version

**Description**: Interactive Image Overlay (IIO) a manual method for co-registration (6-DOF) high resolutions MR scan and a low resolution PET/SPECT brain scan.

**Documentation:**

**Who did**: NRU, Copenhagen, 2000-2003

**Require task index**: 1 (Files load)

**Function name**: coreg

**Function syntax**: coreg(*'batch'*,[],*Filename*,*ParentHandle*,*ReturnFucn*)

**Input parameters**:

*Filename*: structure containing a high resolution MR scan and one or more low resolutions PET/SPECT given in a cell array. All in the Analyze image format.

Example of *Filename* structure:

```
Filename.RES: Name and path for high resolution MR scan
Filename.STD{}: Name and path of one or more low resolution PET/SEPCT
scans
Filename.A: If exist, a given co-registration matrix is used as
initiated co-registration
Filename.AIR: If filename exist, co-registration matrix is saved in
air-format.
```

*ParentHandle*: Figure handle to parent/main figure so it can be found when returning fra Coreg.

Example:
*ParentHandle*=h_mainfig

*ReturnFucn*: Function to call exiting the Coreg program.

Example when exiting from Coreg in Matlab:

*feval(ReturnFucn,'ReturningData',ParentHandle,Aall);*

where

- *ReturnFucn='checkOut'*
- *'ReturningData'*: Parameter telling that co-registration matrix is returned by Coreg.
- *ParentHandle=h_mainfig.*

- *Aall* cell array contain co-registration matrix.

**Output parameters**: None, is using the '*ReturnFucn*'

**Code type**: Matlab code, (figure object)

**Configuration**: none

### *4.5 File formats and directories*

The file name of the first loaded modality given in *project.pipeline.modality{1}…*
In general: Use prefix given in the set-up added to the fileb
Name of *project.pipeline.inputfile.name{1,1}* or/and with an extension.
Files are only made if *workdir* exist, which is generated when the first task is done or an existing project file is loaded.

- **Workdir**: unique project directory
  - Path: sub directory to *project.pipeline.inputfile.path{1,1}* with prefix 'WORK'. If directory exist an index is added to the prefix, 'WORK_*index*'.
- **Log file**: Text file w. log information
  - Filename: *project.pipeline.inputfile.filename{1,1}* with extension '*.log'
  - Path: *workdir*
- **Project file**: Matlab data file.
  - Filename: *project.pipeline.inputfile.filename{1,1}* with extension '*.prj'.
  - Path: *workdir*
- **Set-up file**: A Matlab m-file w. a function setting up a pipeline
  - Filename: *project.pipeline.inputfile.filename{1,1}* with extension '*.prj'.
  - Path: Within the Matlab path
- **Macro file**: Mathlab data file, w. user defined set-up of a pipeline
  - Filename: *project.pipeline.inputfile.filename{1,1}* with extension '*.mco'.
  - Path: user defined
- **Output files:** Generated image files in Analyze format
  - Filename: *project.pipeline.inputfile.filename{1,1}* add prefix given in the set-up file.
  - Path: *workdir*
- **Configurator file**: Text file containing in parameters for a method
  - Filename: user defined
  - Path: Within the Matlab path.
- **Function_wrappers, configurator_wrapper:** Interfacing the pipeline program.
  - Filename: Given in the set-up file.
  - Path: Within the Matlab path
- **Method program files:** Realising a task

- Filename: Given in the set-up file.
- Path: Within the Matlab path