

SNAP Command Line Tutorial

Graph Processing

Issued July 2016

Updated January 2020

Luis Veci

SNAP Command Line Tutorial

This tutorial will show some examples of common processing done via the command line on Windows or Linux.

Graph Processing Framework (GPF)

The SNAP architecture provides a flexible Graph Processing Framework (GPF) allowing the user to create processing graphs for batch processing and customized processing chains. The Graph Builder, in SNAP-Desktop, allows the user to graphically assemble graphs from a list of available operators and connect operator nodes to their sources. Graphs can then be saved and batched processed from the GUI or from the command line.

The GPF is based on the Java Advanced Imaging (JAI) rendering chain. A graph is a set of nodes connected by edges. In this case, the nodes are the processing steps called operators. The edges will show the direction in which the data is being passed between nodes; therefore it will be a directed graph. A graph can have no loops or cycles, so it will be a Directed Acyclic Graph (DAG).

The sources of the graph will be the data product readers, and the sinks can be either a product writer or an image displayed. An operator can have one or more image sources and other parameters that define the operation. Two or more operators may be connected together so that the first operator becomes an image source to the next operator. By linking one operator to another, an imaging graph or processing chain can be created.

The GPF uses a Pull Model, where a request is made from the sink backwards to the source to process the graph. This request could be to create a new product file or to update a displayed image. Once the request reaches a source, the image is pulled through the nodes to the sink. Each time an image passes through an operator, the operator transforms the image, and it is passed down to the next node until it reaches the sink.

The graph processor will not introduce any intermediate files unless a writer is optionally added anywhere in the sequence. Tiles are processed in parallel according to the number of available cores.

Graphs offer the following advantages:

- no intermediate files written, no I/O overhead
- reusability of processing chains
- simple and comprehensive operator configuration
- reusability of operator configurations

Installing SNAP-Engine in Server Mode

The SNAP modules are separated between the SNAP Engine and the SNAP Desktop. The SNAP Engine consists of all core product models and processing functionality. A command-line interface able to execute

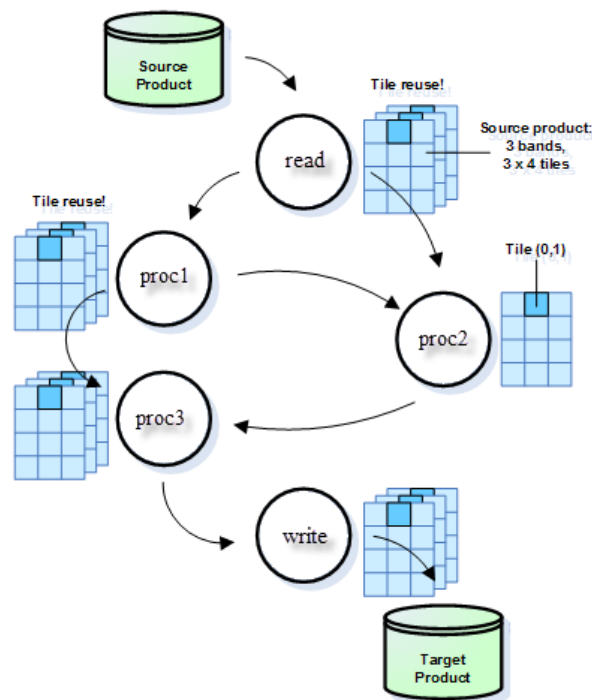


Figure 1 Tile Processing in the Pull Model Graph Processor

within a headless operating system shell would access only the SNAP Engine while the SNAP Desktop is used for creating a client graphical user interface.

The server-mode Toolbox is self-contained and does not depend on any GUI functionality for setup or operation.

For Linux, download the 64-bit unix installer from the STEP website (<http://step.esa.int>).

```
$ chmod +x esa-snap_sentinel_unix_4_0.sh
$ ./esa-snap_sentinel_unix_4_0.sh
```

Follow the installation instructions.

Updating SNAP from the Command Line

In headless environments, you can update modules from the command line without the graphical user interface. Only modules which are already installed can be updated. It is not possible to install new modules. Locate the SNAP executable in the bin directory of the installation folder of SNAP. The executable will be either `snap.sh`, `snap.command` or `snap64.exe` depending on your operating system. In the following commands `snap` is used as a place holder for the executable file. You need to replace it with the appropriate one for your system.

List available parameters

List the parameters with a description which can be passed to the executable.

```
snap --help
```

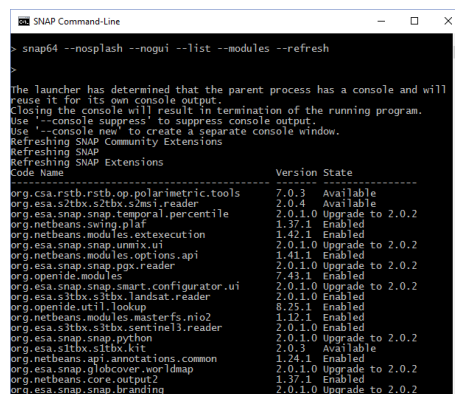
The parameter `--nogui` is missing in this list. This parameter prevents the SNAP GUI from being started.

List all modules

In order to get a list of all modules and the status if there is an update available you can call

```
snap --nosplash --nogui --modules --list --refresh
```

This will give you an output similar to what you can see in the following image.



```

> snap64 --nosplash --nogui --list --modules --refresh

The launcher has determined that the parent process has a console and will
reuse it for its own console output.
Closing the console will result in termination of the running program.
Use "--console suppress" to suppress console output.
Use "--console new" to create a separate console window.
Refreshing SNAP Community Extensions
Refreshing SNAP
Refreshing SNAP Extensions
Code Name                               Version State
-----
org.esa.rstb.rstb.op.polarimetric.tools  7.0.3 Available
org.esa.s2tbx.s2tbx.s2ms1.reader         2.0.4 Available
org.esa.snap.snap.temporal.percentile    2.0.1.0 Upgrade to 2.0.2
org.netbeans.swing.plaf                  1.37.1 Enabled
org.netbeans.modules.extexecution        1.42.1 Enabled
org.esa.snap.snap.umstx.ui               2.0.1.0 Upgrade to 2.0.2
org.netbeans.modules.options.api         1.41.1 Enabled
org.esa.snap.snap.pgx.reader              2.0.1.0 Upgrade to 2.0.2
org.openide.modules                       7.43.1 Enabled
org.esa.snap.snap.smart.configurator.ui  2.0.1.0 Upgrade to 2.0.2
org.esa.s1tbx.s1tbx.landsat.reader        2.0.1.0 Enabled
org.openide.util.lookup                   8.25.1 Enabled
org.netbeans.modules.masterfs.nio2       1.12.1 Enabled
org.esa.s1tbx.s1tbx.sentinel3.reader      2.0.1.0 Enabled
org.esa.snap.snap.python                  2.0.1.0 Upgrade to 2.0.2
org.esa.s1tbx.s1tbx.kit                    2.0.3 Available
org.netbeans.api.annotations.common       1.24.1 Enabled
org.esa.snap.globcover.worldmap          2.0.1.0 Upgrade to 2.0.2
org.netbeans.core.output2                 1.37.1 Enabled
org.esa.snap.snap.branding                2.0.1.0 Upgrade to 2.0.2

```

Update all modules

To update all modules which can be updated you need to call

```
snap --nosplash --nogui --modules --update-all
```

Update specific modules

In order to update just one or multiple specific module you can call:

```
snap --nosplash --nogui --modules --update org.esa.snap.snap.ndvi
org.esa.snap.snap.envisat.reader
```

Additional module related options can be found by calling:

```
snap --nosplash --nogui --modules --help
```

The SNAP Graph Processing Tool (GPT)

SNAP EO Data Processors are implemented as GPF operators and can be invoked via the Windows or UNIX command-line using the GPF Graph Processing Tool `gpt` which can be found in the bin directory of your SNAP installation.

The installation should have added the SNAP/bin folder to the system path in order to call the `gpt` from any folder.

Open a console window and type: `gpt -h`.

This will print out a short description of what the tool is for and describes the arguments and options of the tool. A list of available operators is displayed depending on the toolboxes installed.

Usage:

```
gpt <op>|<graph-file> [options] [<source-file-1> <source-file-2> ...]
```

Description:

This tool is used to execute SNAP raster data operators in batch-mode. The operators can be used stand-alone or combined as a directed acyclic graph (DAG). Processing graphs are represented using XML. More info about processing graphs, the operator API, and the graph XML format can be found in the SNAP documentation.

Arguments:

`<op>` Name of an operator. See below for the list of `<op>`s.
`<graph-file>` Operator graph file (XML format).
`<source-file-i>` The `<i>`th source product file. The actual number of source file arguments is specified by `<op>`. May be optional for operators which use the `-S` option.

Options:

`-h` Displays command usage. If `<op>` is given, the specific operator usage is displayed.
`-e` Displays more detailed error messages. Displays a stack trace, if an exception occurs.
`-t <file>` The target file. Default value is `./target.dim`.
`-f <format>` Output file format, e.g. 'GeoTIFF', 'HDF5', 'BEAM-DIMAP'. If not specified, format will be derived from the target filename extension, if any, otherwise the default format is 'BEAM-DIMAP'. Only used if the graph in `<graph-file>` does not specify its own 'Write' operator.
`-p <file>` A (Java Properties) file containing processing parameters in the form `<name>=<value>` or a XML file containing a parameter DOM for the operator. Entries in this file are overwritten by the `-P<name>=<value>` command-line option (see below). The following variables are substituted in

the parameters file:

\${gpt.operator} (given by the 'op' argument)
 \${gpt.graphFile} (given by the 'graph-file' argument)
 \${gpt.parametersFile} (given by the -p option)
 \${gpt.targetFile} (given by the -t option)
 \${gpt.targetFormat} (given by the -f option)

- c <cache-size> Sets the tile cache size in bytes. Value can be suffixed with 'K', 'M' and 'G'. Must be less than maximum available heap space. If equal to or less than zero, tile caching will be completely disabled. The default tile cache size is '512M'.
- q <parallelism> Sets the maximum parallelism used for the computation, i.e. the maximum number of parallel (native) threads. The default parallelism is '8'.
- x Clears the internal tile cache after writing a complete row of tiles to the target product file. This option may be useful if you run into memory problems.
- T<target>=<file> Defines a target product. Valid for graphs only. <target> must be the identifier of a node in the graph. The node's output will be written to <file>.
- S<source>=<file> Defines a source product. <source> is specified by the operator or the graph. In an XML graph, all occurrences of \${<source>} will be replaced with references to a source product located at <file>.
- P<name>=<value> Defines a processing parameter, <name> is specific for the used operator or graph. In an XML graph, all occurrences of \${<name>} will be replaced with <value>. Overwrites parameter values specified by the '-p' option.

Operators:

BandMaths	Create a product with one or more bands using mathematical expressions.
Collocate	Collocates two products based on their geo-codings.
EMClusterAnalysis	Performs an expectation-maximization (EM) cluster analysis.

...

With the gpt, you could process individual operators or you could process a graph of connected operators.

Type gpt operator-name -h to get usage information on each operator. The usage text of an operator also displays a template clipping of the operators configuration when used in a graph.

gpt Calibration -h

Usage:

gpt Calibration [options]

Description:

Calibration of products

Source Options:

-Ssource=<file> Sets source 'source' to <filepath>.
This is a mandatory source.

Parameter Options:

- PauxFile=<string> The auxiliary file
Value must be one of 'Latest Auxiliary File', 'Product Auxiliary File',
'External Auxiliary File'.
Default value is 'Latest Auxiliary File'.
- PcreateBetaBand=<boolean> Create beta0 virtual band
Default value is 'false'.
- PcreateGammaBand=<boolean> Create gamma0 virtual band
Default value is 'false'.
- PexternalAuxFile=<file> The antenna elevation pattern gain auxiliary data file.
- PoutputBetaBand=<boolean> Output beta0 band
Default value is 'false'.
- PoutputDNBand=<boolean> Output DN band
Default value is 'false'.
- PoutputGammaBand=<boolean> Output gamma0 band
Default value is 'false'.
- PoutputImageInComplex=<boolean> Output image in complex
Default value is 'false'.
- PoutputImageScaleInDb=<boolean> Output image scale
Default value is 'false'.
- PoutputSigmaBand=<boolean> Output sigma0 band
Default value is 'true'.
- PselectedPolarisations=<string,string,string,...> The list of polarisations
- PsourceBands=<string,string,string,...> The list of source bands.

Graph XML Format:

```
<graph id="someGraphId">
  <version>1.0</version>
  <node id="someNodeId">
    <operator>Calibration</operator>
    <sources>
      <source>${source}</source>
    </sources>
    <parameters>
      <sourceBands>string,string,string,...</sourceBands>
      <auxFile>string</auxFile>
      <externalAuxFile>file</externalAuxFile>
      <outputImageInComplex>boolean</outputImageInComplex>
      <outputImageScaleInDb>boolean</outputImageScaleInDb>
      <createGammaBand>boolean</createGammaBand>
      <createBetaBand>boolean</createBetaBand>
      <selectedPolarisations>string,string,string,...</selectedPolarisations>
      <outputSigmaBand>boolean</outputSigmaBand>
      <outputGammaBand>boolean</outputGammaBand>
      <outputBetaBand>boolean</outputBetaBand>
      <outputDNBand>boolean</outputDNBand>
    </parameters>
  </node>
</graph>
```

Calling GPT with an Operator on a Single Product

Let's suppose you wanted to read a product and convert it to another format. You could do that simply by using the Write operator.

gpt Write -h

Usage:

```
gpt Write [options]
```

Description:

Writes a data product to a file.

Source Options:

`-Ssource=<file>` The source product to be written.
This is a mandatory source.

Parameter Options:

`-PclearCacheAfterRowWrite=<boolean>` If true, the internal tile cache is cleared after a tile row has been written. Ignored if `writeEntireTileRows=false`.
Default value is 'false'.

`-PdeleteOutputOnFailure=<boolean>` If true, all output files are deleted after a failed write operation.
Default value is 'true'.

`-Pfile=<file>` The output file to which the data product is written.

`-PformatName=<string>` The name of the output file format.
Default value is 'BEAM-DIMAP'.

`-PwriteEntireTileRows=<boolean>` If true, the write operation waits until an entire tile row is computed.
Default value is 'true'.

Graph XML Format:

```
<graph id="someGraphId">
  <version>1.0</version>
  <node id="someNodeId">
    <operator>Write</operator>
    <sources>
      <source>${source}</source>
    </sources>
    <parameters>
      <file>file</file>
      <formatName>string</formatName>
      <deleteOutputOnFailure>boolean</deleteOutputOnFailure>
      <writeEntireTileRows>boolean</writeEntireTileRows>
      <clearCacheAfterRowWrite>boolean</clearCacheAfterRowWrite>
    </parameters>
  </node>
</graph>
```

The Write operator help shows that you can use the parameters `-Pfile` to specify an output file name and `-Pformat` to specify the file format.

To run gpt on an operator type:

```
gpt <OperatorName> [options] [<source-file-1> <source-file-2> ...]
```

To actually run an operator using the GPT, it is necessary to indicate the path to the source product(s), to the target product and to other operator-specific parameters which might be mandatory or specific.

To use the Write operator to write a product to HDF5 format:

```
gpt Write -Pfile=~/myoutput/myfile.h5 -Pformat=HDF5 pathToInputProductFile
```

The gpt can be run on several products in batch by using it within scripts and replacing the input files and output files.

Calling GPT with a Graph

Rather than calling each operator and specifying all its parameters, it is more convenient to pass the required settings in an xml-encoded graph file. It will then suffice to just pass the graph as parameter to the gpt.

To run gpt on a graph file type:

```
gpt <GraphFile.xml> [options] [<source-file-1> <source-file-2> ...]
```

Creating a Graph File

You can create your own graph files using a text editor.

The basic format of a graph XML file is:

```
<graph id="someGraphId">
  <version>1.0</version>
  <node id="someNodeId">
    <operator>OperatorName</operator>
    <sources>
      <sourceProducts>${sourceProducts}</sourceProducts>
    </sources>
    <parameters>
      ....
    </parameters>
  </node>
</graph>
```

You can use the operator help (gpt operatorName -h) to get a listing of the configuration of operator parameters within a graph.

For example, as shown above, the graph configuration for Calibration is:

```
<graph id="someGraphId">
  <version>1.0</version>
  <node id="someNodeId">
    <operator>Calibration</operator>
    <sources>
      <source>${source}</source>
    </sources>
    <parameters>
      <sourceBands></sourceBands>
      <auxFile>string</auxFile>
      <externalAuxFile>file</externalAuxFile>
      <outputImageInComplex>boolean</outputImageInComplex>
      <outputImageScaleInDb>boolean</outputImageScaleInDb>
      <createGammaBand>boolean</createGammaBand>
      <createBetaBand>boolean</createBetaBand>
      <selectedPolarisations>string,string,string,...</selectedPolarisations>
      <outputSigmaBand>boolean</outputSigmaBand>
      <outputGammaBand>boolean</outputGammaBand>
      <outputBetaBand>boolean</outputBetaBand>
```



```

    <outputDNBand>boolean</outputDNBand>
  </parameters>
</node>
</graph>

```

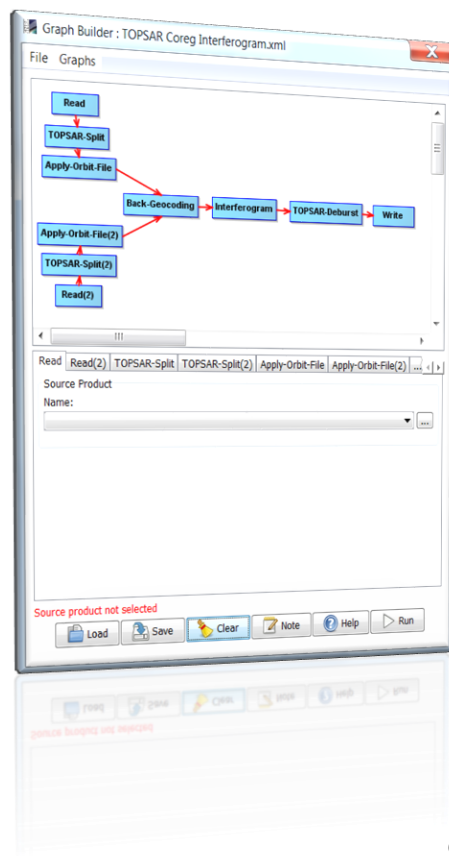
Replace the parameter data types (string, boolean, etc) with actual values.
Save the graph to an XML text file `calibrateGraph.xml`.

Note that in the list of available operators a *Read* and a *Write* operator exists. These are not needed because GPF will add those operators to the graph on its own for this simple case. However, they could be specified in more complex graphs.

To run this graph with `gpt` and write the output product to a specific path, type:

```
gpt calibrateGraph.xml -t ~/out/output.dim
```

Using a Graph Created by the GraphBuilder



The GraphBuilder in SNAP Desktop could be used to construct more complicated graphs with interconnected operators.

See the Sentinel-1 Toolbox [Graph Building tutorial](#) to learn more about how to drop in operators, connect them, and specify parameters.

When you save a graph, the parameters you have specified for the current data product(s) are also saved to the graph file. To reuse the graph from the command line using `gpt`, you may need to open the graph XML file in a text editor and remove or replace the value for some parameters in order to make the graph generic for any input product.

An example graph can be found in the `.snap/graphs` folder. The `.snap` folder is found in your home directory in Linux or in `c:\users\username\.snap` in Windows.

You could insert variables in the form `${variableName}` in place of a parameter value. You can then replace the `variableName` with a value at the command line. For example, if a parameter for a file included the variable for `${myFilename}`

```

<parameters>
  <file>${myFilename}</file>
</parameters>

```

```
gpt mygraph.xml -PmyFilename=pathToMyFile
```

Batch Processing Examples

The following are some examples which can be done with Windows batch files.

Calibration

For all envisat products in folder c:\ASAR run gpt Calibration and produce the output in the folder c:\output

```
for /r "c:\ASAR" %%X in (*.N1) do (gpt Calibration "%%X" -t "C:\output\%%~nX.dim")
```

Terrain Correction

For all dimap products in folder c:\data run the graph TC_Graph.xml and output to c:\output with the same name as the input file.

```
for /r C:\data %%X in (*.dim) do (gpt TC_Graph.xml -Pfile="%%X" -Ptarget="C:\output\%%~nX.dim")
```