On scripting GRASS GIS: Building location-independent command line tools

Peter Loewe
RapidEye AG, Molkenmarkt 30, 14476 Brandenburg an der Havel, Germany
`loewe@rapideye.de`

Keywords: GRASS GIS, GRASS Scripting, FOSS GIS Workflows, Embedded GIS, Bash, Python

Abstract

This paper discusses scripting techniques within the context of GRASS GIS. After an overview over scripting for interactive GRASS sessions, it is shown how GRASS GIS-provided functionality can be used for external applications. This approach of external scripting allows for the application of GRASS GIS-based functionality to be used for standalone applications and embedding in larger automated workflows.

On Scripting

What's in a GRASS Script

Scripting allows to to re-use workflows which have been previously developed in interactive sessions of a user (expert) and a GIS by automatization. The Geographic Resources Analysis Support System (GRASS) GIS, a Free and Open Source (FOSS) application consists of a variety of independent software modules, each of them providing a unique GIS processing capability. During an interactive GRASS session, the GRASS modules are applied to the designated set of geodata (”GRASS location”) by the user (expert). So a script in GRASS GIS is a control structure which orchestrates the execution of underlying GRASS GIS modules. The control structure is wrapped around the GRASS modules and accesses them via their defined interfaces. Examples are provided in [1] and [2].

Scripts are an integral part of GRASS GIS: A significant amount of the available GRASS GIS functionality is actually provided by scripts, instead of modules consisting of compiled binary code. The number of GRASS modules which are scripts can be checked by examining the ”scripts” directory of a GRASS GIS distribution. An overview over currently available GRASS scripts beyond the scope of the standard GRASS GIS distribution is available in [3].

New GRASS functionality can most effectively be implemented on sourcecode level, using the GRASS libraries and C-API. However, this approach requires a good understanding of the internals of GRASS GIS. By comparison, scripting allows for fast prototyping and does not require source-code-level knowledge. The downside is a generally slower execution speed than compiled C-code.

Kinds of GRASS Scripts

On a high level, GRASS scripts fall in two general categories: **Module-Scripts**, like those included in every GRASS GIS distribution, provide additional GIS functionality to the user by combining multiple pre-existing GRASS modules. The other category are **workflow-scripts**. These provide for bulk tasks like the repeated use of processing chains (i.e. "apply the same computation on [1 .. n] data layers").

One exception to the rule is the **g.mremove** script [4], which falls in both categories: Being part of the standard GRASS distribution makes it a module script. But it is also a workflow-script, as it allows for bulk removal of data layers, instead of the repeated use of the module **g.remove**.

Languages for Scripting

While Unix-Shells and Perl have been previously the most widely used programming languages for scripting, Python has been recently adopted as the preferred object-oriented scripting language by the GRASS community: Starting with GRASS 7.0, all scripts included in the standard GRASS GIS distributions are coded in Python. Add-on scripts provided by the user community continue to be formulated in various programming languages [3]. For the examples given in this paper, the Bourne Again Shell (bash) is still used. The code can be easily translated into Python commands.

The Joy of Boilerplating

The effort to create a script is only justified by adequate savings in time, effort and ease when re-using the workflow. For this, two factors are crucial: Obviously, the primary requirement is the correct implementation of the algorithm, but equally important are standard-conforming "friendly" user interfaces. GRASS started out 25 years ago as a system to be accessed via a (unix)shell as a command line interface (CLI). During the evolution of GRASS, graphical user interfaces (GUI) have been added to allow interaction via graphical icons and visual indicators. Therefore, GRASS scripts should support support both means of interaction (CLI and GUI). This can be easily provided by the *g.parser* module [5]. It provides standardized input/output interfaces and help-page templates, both for the use of the module of a CLI and GRASS GUIs. This approach is colloquially referred to as "**boilerplating**", as it provides a convenient means to create a high-quality front-end for the users with little effort. Further, a script can also be started by a mouseclick if it is integrated in an overall GRASS GUI-Manager (like Tcl/Tk or wxPython) by including it to the GUI configuration scripts.

Automating GRASS GIS

Apart from applying GRASS scripts during interactive GRASS sessions, it is possible to automatically deploy GRASS scripts without user interaction. This allows for automated GRASS-based processing without the need for user interaction.

During the startup of a GRASS session, the GRASS environment has to be linked to the current geodata work environment (GRASS term: 'location') and its metadata (i.e. projection). This information is defined by a set of environment variables. The setting of these shell variables is automated, but can also be done manually or within scripts. The latter technique will be put to use in the External Scripting Section.

Scripted GRASS Sessions

Since the release of GRASS 6.3, a GRASS session can be automatically started to execute a preselected script: After setting a specific environment variable (i.e. GRASS_BATCH_JOB) [9] to the path of an external GRASS script, this script will be executed when the next GRASS session is started, implicitly assuming the availability of a GRASS location. This approach can not be used in situations when no GRASS locations have been set up before.

External Scripting

External scripting invokes GRASS functionality outside of an interactive or scripted GRASS session. This allows to create stand-alone scripts and has been available since GRASS Version 5.x : 'Once a minimum number of environment variables have been set, GRASS commands can be integrated into shell, CGI, PERL, PHP and other scripts' [2, p.290]. Initially, his approach appears less convenient than variable-controlled scripted GRASS sessions. However, it enables full control over the location settings. This allows to start computations literally 'from scratch', creating GRASS locations in the process.

Example: External Scripting

The following example showcases the necessary steps required for an external script to initialize a GRASS session. It demonstrates how to proceed from an unprojected location to a projected one.

External GRASS Initialisation

To invoke GRASS without a proper location available in the filesystem, a temporary mock-up has to be created. This is done in a user-defined folder, by setting up the following directory structure: On the highest level, the location directory ($THE_LOCATION), provides metadata and projection information. For each explicitly named region of interest (GRASS term: mapset), a subfolder is created within the location-directory. As each GRASS location is required to contain one mapset named PERMENENT, this is the recommended name for the intitial mapset to be created. In the following code fragment, the directories are referred to by the variables $THE_LOCATION and $THE_MAPSET while the variable

$GRASS_DBASE_EXAMPLE refers to the temporary directory. In addition to the described folder stucture, a file is required by GRASS GIS to contain the location's metadata. This ASCII file, by default named 'WIND', is set up for an location lacking projection information ('proj:0'), defining an area of singular extent. These settings serve as placeholders to be updated during the following steps. The file is also copied into a second instance ('DEFAULT_WIND'). In a last step, the database driver is defined by creating the file 'VAR'.

```
# Create a WIND file with minimal information and no projection:
echo "proj:       0
zone:       0
north:      1
south:      0
east:       1
west:       0
cols:       1
rows:       1
e-w resol:  1
n-s resol:  1
top:        1
bottom:     0
 cols3:      1
rows3:      1
depths:     1
e-w resol3: 1
n-s resol3: 1
t-b resol:  1
" > ${GRASS_DBASE_EXAMPLE}/$THE_LOCATION/$THE_MAPSET/WIND

# Copy WIND-file to DEFAULT_WIND:
cp ${GRASS_DBASE_EXAMPLE}/$THE_LOCATION/$THE_MAPSET/WIND \
 ${GRASS_DBASE_EXAMPLE}/$THE_LOCATION/$THE_MAPSET/DEFAULT_WIND

# Set default database driver:
echo "DBF_DRIVER: dbf
DB_DATABASE : $GISDBASE/$LOCATION_NAME/$MAPSET/dbf/
" > ${GRASS_DBASE_EXAMPLE}/$THE_LOCATION/$THE_MAPSET/VAR
```

[**Code Snippet 1**: First step of GRASS GIS initialization]

When GRASS is started without a reference to a specific location, it attempts to refer to the last previously used location. This mechanism is exploited to point GRASS to the mock-up location. The following code snippet demonstrates how the required references are stored in an ASCII file. The standard file used for this is named '.grassrc6' (for GRASS versions 6.x) but in the mock-up case, any arbitrary name can be used because the filename is stored in a shell variable in the last step.

```
echo "GISDBASE: ${GRASS_DBASE_EXAMPLE}
LOCATION_NAME: $THE_LOCATION
MAPSET: $THE_MAPSET
" > ${GRASS_DBASE_EXAMPLE}/$THE_GRASSRC
```

[**Code Snippet 2**: Second step of GRASS GIS initialization: Creation of the GRASSRC-file]

As a final step, the shell variables required for GRASS have to be set to match the newly created directories and files.

```
# $GISBASE points to the GRASS installation to be used:
 export GISBASE=/opt/grass

# Extend $PATH for the default GRASS scripts:
 export PATH=$PATH:$GISBASE/bin:$GISBASE/scripts
```

```
# Add GRASS library information:
 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GISBASE/lib

# use process ID (PID) as lock file number:
 export GIS_LOCK=$$

# Backup previously existing GRASS references
# export GISRC_BACKUP=${GISRC}

# path to GRASS settings file:
 export GISRC=${GRASS_DBASE_EXAMPLE}/$THE_GRASSRC

db.connect driver=dbf database='$GISDBASE/$LOCATION_NAME/$MAPSET/dbf/'
```

[**Code Snippet 2**: Third step of GRASS GIS initialization: Export of paths and settings]

At this point, a fully operational GRASS GIS environment has been established within the script.

Auto-generating a location from external data

After the initialization stage, GRASS modules can be used within the mock-up location. Geodata can be imported, inluding the generation of additional locations based on the projecton information derived from the input data. The GRASS module **r.in.gdal** [6] is used for raster data while the module **v.in.ogr** [10] is used for vector import.

```
# Read the raster data stored in the GeoTIFF "example.tif" in directory /tmp
# into to the layer 'raster_layer' in the new location 'raster_location'
r.in.gdal -e input=/tmp/example.tif output=raster_layer location=raster_location

# Read the line-vectors stored in the shapefile "example.shp" in directory /tmp
# into to the layer 'vector_layer' in the new location 'vector_location'
v.in.ogr -e dsn=/tmp output=vector_layer layer=example.shp type=line location=vector_location
```

Switching over to the new location is achieved by setting the shell variables accordingly. By default, the data is stored in the newly created location within the mapset 'PERMANENT'.

If the external data is lacking proper projection information (i.e. a Shapefile is provided without a .prj-file), the approach described in the next section can be used to explictly define the projection.

Projection Changes within a Script

Using the module **g.proj** [15] allows to advance from the initial mock-up location to a projected location: The necessary parameters can either be provided to the **g.proj** module or a European Petroleum Survey Group (EPSG) [13] numeric code can be used. It must be noted that **g.proj** will only work on the PERMANENT mapset of a location.

The example uses the EPSG code "4326", which refers to geographical coordinates using the WGS84 ellipsoid:

```
# Override the current projection setting
 g.proj --quiet -c epsg=4326
# Define the extent of the region of interest
 g.region --quiet -s  n=90 s=-90 w=-180 e=180 res=1
```

[**Code Snippet 4**: The usage of g.proj to override the current projection with a EPSG code]

Shifting the Focus between Multiple Locations

After a projected location containing data has been established, either by definition in the script or external data, it might become necessary to write out data in another projection. For this, it is necessary to transfer the results into an additional location in the output projection, using **r.out.gdal** [7] and **v.out.ogr** [11] to finally export the actual data.

The export location is created as described before: The sequence of defining a GISRC-file pointing to the location and mapset, exporting the GIS_LOCK variable and finally pointing the GISRC-Variable to the new GISRC-file is repeated, thereby changing the focus of the GRASS session to the addedd location. Once this has been accomplished, the reprojection modules **r.proj** [8] and **v.proj** [12] can be used to transfer the data into the location.

After providing the required output data products, all locations can be safely removed from the file system. Care should be taken to restore the cached value of the variable $GRASSRC for upcoming interactive GRASS sesssions.

Fields of Applications

The approach of External Scripting of GRASS GIS is beneficial for all repetitive processing tasks without the need of user interaction. Such tasks can be either stand alone applications or part of larger workflows, where geospatial processing is only a subtask.

GRASS scripting also provides alternatives to the set-up of up Open Web Services (OWS), like Web Mapping Service (WMS) or Web Processing Service (WPS) to provide automated geospatial map-products or to do geoprocessing. Such GIS-produced maps can be used to regularly update Webpages. A classic example by Neteler [14] provides a global map of earthquake epicenters. This is accomplished by importing and processing external earthquake information in a GRASS-based externally scripted workflow.

Another noteworthy aspect about using automated scripts is the option to reduce the functionality and thereby to minimize the footprint of the GIS: The footprint of a contemporary "off the shelf" GRASS GIS installation on the filesystem is about 50Mb. If GRASS functionality is implemented in an automated workflow, GRASS can be reduced to exactly those modules which are required in the workflow. All other GRASS modules would be a waste of potentially critical ressources. This approach is especially useful when implementing workflows on embedded systems, such as environmental sensors with limited system ressources.

Conclusion

This paper provided a technical overview on the current options to deploy scripts based on GRASS modules. It was demonstrated how inherent GRASS GIS-functionality can be wrapped in scripts to be applied beyond the scope of an interactive GRASS GIS session.

Beyond the technical challenge to "think script", the significance of being able to provide ready-to-use tools to a wide audience is emphasized: The empowering of audiences lacking previous FOSS GIS exposure to perform challenging geospatial tasks (while hiding the complexities of the actual GRASS GIS solution) is expected to broaden the user base and overall impact of Free and Open Source Softare (FOSS) GIS applications.

References

1. Löwe P: Niederschlagserosivität: Eine Fallstudie aus Südafrika, basierend auf Wetterradar und Open Source GIS, VDM Verlag, 2008, ISBN 978-3-8364-5018-8
2. Neteler, M. and Mitášová H.: Open Source GIS: A GRASS GIS Approach, Kluwer Academic Publishers Group, 2002, ISBN 1-4020-7088-8
3. GRASS AddOns
   http://grass.osgeo.org/wiki/GRASS_AddOns
4. GRASS GIS module to remove data base element files from the user's current mapset.
   http://grass.fbk.eu/grass64/manuals/html64_user/g.mremove.html
5. GRASS module to provide canonic interfaces for scripts
   http://grass.itc.it/grass64/manuals/html64_user/g.parser.html
6. Raster data import
   http://grass.itc.it/grass64/manuals/html64_user/r.in.gdal.html
7. Raster data export
   http://grass.itc.it/grass64/manuals/html64_user/r.out.gdal.html
8. Reprojection of raster data from other GRASS locations with differing projection
   http://grass.itc.it/grass64/manuals/html64_user/r.proj.html
9. Overview over GRASS environment variables and setting options
   http://grass.itc.it/grass64/manuals/html64_user/variables.html
10. Vector data import
    http://grass.itc.it/grass64/manuals/html64_user/v.in.ogr.html
11. Vector data export
    http://grass.itc.it/grass64/manuals/html64_user/v.out.ogr.html
12. Reprojection of vector data from other GRASS locations with differing projection
    http://grass.itc.it/grass64/manuals/html64_user/v.proj.html
13. Overview over EPSG codes
    http://www.epsg.org/Geodetic.html
14. Automated Earthquake Map based on GRASS GIS
    http://grass.itc.it/spearfish/php_grass_earthquakes.php
15. GRASS module to manipulate projection settings
    http://www.grass.itc.it/grass64/manuals/html64_user/g.proj.html