Cygpack

Utilities for working with GEMPACK under Cygwin
Release 1, 2013–01

R.A. McDougall
Badri Narayanan G

This note describes release 1 of Cygpack, a suite of utilities for working with GEMPACK under Cygwin, developed by R.A. McDougall and Badri Narayanan G at the Center for Global Trade Analysis, Purdue University.

Table of Contents

Acknowledgments

We thank the developers of GEMPACK for permission to distribute `for2exe`, a work derivative from the GEMPACK batch file '`ltg.bat`'. We thank Michael Jerie for helpful advice.

1 Introduction

GEMPACK (General Equilibrium Modelling PACKage) is a suite of economic modeling software designed for building and solving applied general equilibrium models. Recent releases of GEMPACK run only under Microsoft Windows, and some GEMPACK utilities, notably ViewHAr, have never been released for any operating system other than Windows. GEMPACK is developed and maintained at the Centre of Policy Studies/Impact Project, Monash University.

Cygwin is a software package providing a Linux-like working environment on Windows, originally developed by Cygnus Solutions, later by Red Hat, and currently by employees of Red Hat and independent volunteers. At the Center for Global Trade Analysis, some staff members have found it a useful supplement to GEMPACK in various tasks pertaining to development and maintenance of the GTAP data base.

Cygpack is a small collection of software utilities developed at the Center for working with GEMPACK on Cygwin.

2 Cygwin at the Center

In tasks pertaining to maintenance and development of the GTAP data base, while performing data handling almost exclusively with GEMPACK, we find Cygwin programs useful in various ancillary tasks.

We find the Bash shell a pleasant and capable user interface and scripting language.

We use GNU Make to combine shell commands into sequences to perform complex tasks for individual applications in which the procedure is defined mostly in user-compiled executables. For that purpose, it offers several advantages over the obvious alternative, shell scripting. It provides intrinsic error handling, relieving the developer of the need to add explicit error handling to each command. It remakes all and only those files that are out of date, so that we don't waste time remaking up-to-date files, don't forget to remake out-of-date files, and don't need a separate script for every possible out-of-date combination. It allows us to specify any intermediate file as the target, so that we don't need separate scripts for partial runs. For general-purpose utilities, however, in which the procedure is defined mostly in shell commands, we prefer shell scripts.

Unix utilities assist in various text processing tasks. For instance, to run a TABLO-generated program over all GTAP regions, we need command files for all regions, differing only in the country codes included in file names; we generate them automatically from a single template file using the Sed utility.

We use Awk and Perl in more complex text processing tasks and in those few data handling tasks to which TABLO is unsuited.

This tool collection — Bash, GNU Make, Unix text processing utilities, Perl — is used mostly in preprocessing and in post-construction analysis. For the main GTAP construction program, we rely on an older established tool set, including the Windows command prompt and Opus Make, with some use of Unix text processing utilities. In that tool set we do not include Perl, but rely for text scripting exclusively on Awk.

3 Terminology

We need some terminology to talk about files and directories, and this is a convenient place to define it.

A file may be a regular file or a directory. A file name includes a file-within-directory part, or *base name*, and may also include a *directory part*, which (if present) is the name of the file's parent directory. If the base name contains one or more period characters '`.`', the last period character and any following characters comprise the *file name extension*; otherwise, the extension is empty. Removing the extension from the file name leaves the *file name stem*; removing it from the base name leaves the *base name stem*. But if just the first character of the base name is a period, then the extension is empty, and the stem is the whole name (so the extension never comprises the whole base name, and the stem is never empty). So for instance in the file name '`share/doc/cygpack.pdf`', '`cygpack.pdf`' is the base name, '`share/doc`' is the directory part, '`.pdf`' is the extension, '`share/doc/cygpack`' is the file name stem, and '`cygpack`' is the base name stem; while in the file name '`.bashrc`', '`.bashrc`' is the base name, the file name stem, and the base name stem, and the directory part and the extension are empty.

4 Cygpack

In working with GEMPACK and Cygwin together, we have developed some utilities to perform frequently occurring tasks. Some of those utilities, like the ones described here, give Bash shell users access to capabilities otherwise available at the Windows command prompt; others provide (as far as we are aware) new capabilities. For some time these have remained unpublished, but we find it necessary now to release a few, to support the release of other materials that rely on them. This minimal initial release comprises just two utilities, For2exe and Tab2exe, and a Make file template. As we release it, needing a name to call it by, we name it Cygpack.

In general, we want Cygpack programs to meet expectations conditioned by familiarity with GNU/Linux utilities, modified to suit the simplicity of the programs, the smallness of the user base, and the narrow range of environments targeted (i.e., Cygwin only). More particularly:

- Programs install into the '`/usr/local`' tree. In particular, programs intended to be called by users install into '`/usr/local/bin`', and programs intended to be called by other programs install into '`/usr/local/lib/`*`user-program`*'.
- Programs not intended to be executed directly by users we install into '`/usr/local/lib`'. Under the Filesystem Hierarchy Standard, this should contain only architecture-dependent files, but we install there also architecture-independent files such as shell scripts and Windows batch files.
- Installation is done by '`make; make install`', omitting the preliminary '`configure`'.
- Options are preferred to non-option arguments in supplying information only rarely required (perhaps to override a frequently used default setting), or for which the valid settings are predefined and few.
- Wherever reasonable, for any information that can be supplied with an option, there is a default setting.
- Programs generally observe GNU/Linux conventions in handling arguments. In particular:
  - Options are case-sensitive.
  - Every short option (other than '`--`') has a long synonym.
  - Long options begin with two minus signs '`--`'.
  - Every program has options '`--help`' and '`--version`'.
  - Long options may be abbreviated to unambiguous prefixes. For example, if a program's only long options are '`--help`' and '`--version`', '`--version`' may be abbreviated to '`--vers`' or even '`--v`'. On the other hand, if a program has long options '`--stack`' and '`--sti`', '`--sti`' must not be abbreviated at all.
  - The option '`--`' marks the end of options. So to use a program with a file whose name begins with a minus sign, without the program's mistaking the filename argument for an option, the user may preceed the filename argument with '`--`'.

    That doesn't mean that beginning file names with '`-`' is a good idea. On the contrary, it is liable to make trouble in interactive work, since the file name requires protection with '`--`' in so many commands. Furthermore, it may lead to trouble in

indirect calls. If, for instance, a user attempts to compile Fortran generated from a TABLO source file '`-pg.tab`' with the command

```
for2exe -- -pg
```

she may find that while we have taken care within `for2exe` to avoid interpreting the file name arguments as options, we have found no way to make the Fortran compiler do likewise, and that the attempt fails with a "Can't compile . . ." message.

  - Unrecognized options are not ignored, but trigger an error exit.
  - An argument to a short option may be separated from it by space, but need not be, for example, `-S400000` or `-S 400000`; an argument to a long option must be separated by an equals sign '`=`', for example, `--stack=400000`. In the latter case, there must be no space before or after the equals sign.
  - Contrary to GNU convention, once the first non-option argument is encountered, all remaining arguments are treated as non-option arguments, even if they begin with a minus sign '`-`'.
  - Redundant non-option arguments are silently ignored.

- Regarding unusual characters in file names, Cygpack imposes no restrictions beyond those imposed by GEMPACK (that is, we don't think it does; we don't check this thoroughly). In particular, it suppports file names with embedded space characters (we do check this). We support those file names less as a service to users than to ourselves; testing with them helps keep our code clean, whether or not they serve any real need. Guidance on file names suitable for use with GEMPACK is available in for instance section 7.1, "Allowed file and directory names", in the 2011–10–10 edition of the GEMPACK manual.
- In filename arguments, in cases where GEMPACK requires a specific filename extension (for example, '`.tab`' for TABLO source files), omission of the extension from the argument is allowed but not required.
- Subject to file and directory permissions, the user may call programs from any directory to work with files in any directory. In particular, the files that the program is to read or write need not be in the user's working directory.
- Both Unix- and Windows-form file names are accepted.
- Filename arguments are case-insensitive (on our system at least, though we don't know how well this will generalize across other systems).
- Successful operation is silent.
- Exit status is zero if successful, non-zero otherwise.
- Every program has a manual page.

In discussion of individual programs, conformity with those expectations is implicitly assumed, except where divergence is explicitly noted.

The authors offer no undertaking whatsoever of user support for Cygpack. They do not rule out the possibility that some problem reports may raise their interest sufficiently to lead them to respond to them.

As noted above, this release contains just three components:

`for2exe` Generate an executable program from TABLO-generated Fortran source.

`tab2exe` Generate an executable program from TABLO source.

`cygpack.mk`
Serve as a template for Cygpack-aware Make description files.

Future circumstances may or may not conduce to more expansive releases.

Besides the Cygwin base packages, Cygpack requires the make and texinfo packages for installation, and the cygutils package for operation. For Bash, it requires version 4 or higher.

5 For2exe

`for2exe` generates an executable program from TABLO-generated Fortran:

```
for2exe [-S size] file
```

The argument *`file`* is the file name of the Fortran program. The *`file`* argument may contain a directory part, and may omit the extension '`.for`'.

A stack size argument may be supplied with the option '`-S` *`size`*' or '`--stack=`*`size`*'. In use with Lahey-Fujitsu Fortran (LF95), if a stack size is supplied, it must be numeric; if not supplied, the size defaults to 400000. In use with other compilers, the option is ignored.

The output file has the extension '`.exe`', and the same file name stem as the Fortran program. An implication is that it is written to the same directory as that from which the Fortran program is read. That directory also serves for all intermediate and diagnostic files.

`for2exe` provides for the Cygwin Bash shell capability similar to that provided for the Windows command prompt by the GEMPACK batch file '`ltg.bat`'.

Every GEMPACK installation contains an '`ltg.bat`' file adapted to a particular Fortran compiler. The `for2exe` utility comprises several working scripts corresponding to those various '`ltg.bat`' files, and a wrapper script that detects the compiler in use and calls the appropriate working script.

The current GEMPACK release at the time of writing, GEMPACK 11, works with 32- and 64-bit Intel Fortran, 32- and 64-bit GNU Fortran (GFortran), and LF95. The previous release, GEMPACK 10, works with Intel Fortran and LF95. The For2exe user executable, '`/usr/local/bin/for2exe`', determines for which compiler GEMPACK is configured, and then calls either '`ifor2exe`' for Intel Fortran, '`gfor2exe`' for GFortran, or '`lfor2exe`' for LF95. Those working scripts, not meant to be called directly by users, are installed in '`/usr/local/lib/for2exe`'.

Each working script is a free translation into Bash of the corresponding version of '`ltg.bat`'. In the essential operations, options file preparation and `gptask` and compiler invocation, it follows the original exactly, but in parameter processing, error handling, and cleanup, it may diverge slightly.

We have tested the script only with GEMPACK Releases 10 and 11, and only with the Fortran compiler versions we happen to have on hand: Intel Fortran Version 12.1 (configured as recommended by the GEMPACK developers), the GEMPACK-compatible distribution of GNU Fortran 4.4.4, and LF95 Express Release 7.10.02. How likely it is to work with other versions, or across the many possible configurations of Intel Fortran, we do not attempt to guess, except that we would not expect it to work with Intel Fortran versions 10 or earlier.

One reason why future GEMPACK or compiler versions may break `for2exe` is its need to inspect some aspects of their configuration. It determines the compiler in use with GEMPACK by looking at the first line of the file '`libs/gpcomp.use`' under the GEMPACK directory.[1] With Intel Fortran, it sets the environment variables `PATH`, `LIB`, and `INCLUDE` according to their settings at the Windows command prompt after running '`ifortvars.bat`'. With GFortran, it sets `PATH` according to its setting after running '`gfortvars.bat`'. Future changes in GEMPACK or compiler setup may defeat these arrangements.

[1] We thank Michael Jerie for suggesting this approach.

Attempts to compile source files with names beginning with a minus sign '`-`' are likely to fail.

There are some small differences in functionality between `for2exe` and `ltg.bat`:

- At the time of writing, it is not clear whether `ltg.bat` supports compilation of source files not in the working directory. It does not reject filename arguments that contain directory components, but it does sometimes fail to compile the named files. At the time of writing, it compiles out-of-directory files successfully with LF95, but not with Intel Fortran or GFortran. `for2exe` does support compilation of out-of-directory files.
- After generating the executable, `ltg.bat` deletes the Fortran source file. That has the odd implication that if run twice, the first run frustrates the second. Used with `make`, moreover, it means that `make` is liable to find the executable out of date, and to insist on remaking it and its dependencies, however fresh they may in fact be. Diverging from `ltg.bat` in that respect, `for2exe` does not remove the Fortran source.

  Besides the main Fortran source file '*`stem`*`.for`', `tablo.exe` produces several auxiliary Fortran files, with names of the form '*`stem`*`_[cmuv][0-9].for`' or '*`stem`*`_[cmuv][0-9][0-9].for`'. Those files too `ltg.bat` removes and `for2exe` retains.
- `ltg.bat` and `for2exe` call `gptask.exe` to divide the main Fortran file '*`stem`*`.for`' into smaller files '*`stem`*`__[0-9].for`' or '*`stem`*`__[0-9][0-9].for`'. `for2exe` removes just names of those forms, but `ltg.bat` removes all names of the form '*`stem`*`__*.for`'. So `ltg.bat` may sometimes remove too much, in unlikely circumstances such as compiling files generated by TABLO from '`pg.tab`' in the presence of files generated from '`pg__x.tab`'. On the other hand, it is possible that `for2exe` may sometimes remove too little.
- Compilation also produces module and object files with extensions '`.mod`' and '`.obj`'. These files sometimes, but not always, have stems similar to those of the auxiliary Fortran files. With the Intel and GNU compilers, for instance, if the stem of the main Fortran file contains space or period '`.`' characters, then for the '`.mod`' files, the part of the stem before the last space or period character is removed; so from a main Fortran source file '`stylized Johansen.for`', we get auxiliary Fortran files with names like '`stylized Johansen_c11.for`', but module files with names like '`johansen_c11.mod`'.

  `ltg.bat` deals with this by deleting all '`.obj`' and '`.mod`' files. `for2exe` deletes '`.obj`' and '`.mod`' files selectively, but may sometimes select wrongly. Knowing of the special treatment of file stems containing space or period characters with the Intel and GNU compilers, we have adapted `for2exe` to it; but we have undertaken no systematic investigation of object and module file naming, and there are likely to be other irregularities of which we remain unaware and which we have not addressed.

In unsuccessful use, if the script fails in attempting a `gptask` or compiler run, it saves files '*`stem`*`.stderr`' and '*`stem`*`.stdout`'. One or both of these may be empty; either or both, if non-empty, may provide some guidance to the cause of failure.

It will be apparent to the reader that there is no claim of independence in content or expression of `for2exe` from `ltg.bat`, and that former is released only with the consent of the developers of the latter. It will also be apparent that any expectation of user support for `for2exe` from the developers of `ltg.bat` would be utterly unreasonable.

6 Tab2exe

`tab2exe` generates an executable program from TABLO source:

```
tab2exe [-s stifile] [-S size] [-t directory] tabfile
```

where *`tabfile`* is the name of the TABLO source file, *`directory`* is the directory to which the executable is to be written (by default, the same as that from which the source file is read), and *`stifile`* is the name of a TABLO stored input file.

The file name argument *`tabfile`* may contain a directory part, and may omit the extension '`.tab`'.

By default, the output files are written to the directory from which the input ('`.tab`') file is read. To specify a different directory, use the option '`-t` *`directory`*' or '`--target-directory=`*`directory`*'. The target directory must already exist; if it does not, `tab2exe` does not create it, but exits unsuccessfully. If it does exist, the '`.axs`', '`.axt`', '`.exe`', and '`.min`' files are written to it; for convenience in editing, the '`.inf`' file is always written to the directory containing the '`.tab`' file.

By default, `tab2exe` calls `tablo` with the command-line option '`-wfp`'. That implies that other options take their default values, that no condensation is done, and that the output files have the same base name stem as the inputs. If that is not what is wanted, the user may indicate otherwise in a stored input file with the option '`-s` *`stifile`*' or '`--sti=`*`stifile`*'.

To stored input files used with `tab2exe`, one special restriction applies: file name stems must be supplied without a directory part; that is, just base name stems should be supplied. That is because `tab2exe` runs `tablo.exe` in a special temporary directory. It creates that directory, makes it the working directory, copies the input files into it, creates the output files in it, copies them from it to their final destination, and finally removes it. So the files referred to in the stored input file are not the copies in the original source or final destination directories, but the copies in the temporary directory.

For use with LF95, a stack size other than the default may be supplied as an argument to the option '`-S` *`size`*' or '`--stack==`*`size`*', as with `for2exe` (see Chapter 5 [For2exe], page 8).

Running `tab2exe` combines two operations, generating Fortran code from TABLO and generating an executable program from Fortran. Combining the two operations in a single program is motivated less by a desire to save keystrokes than by some small difficulties with the programs run separately:

- Besides the main Fortran file '*`stem`*`.for`', TABLO generates several auxiliary files '*`stem`*`_*.for`'. `for2exe` removes its own intermediate files but not the prerequisite Fortran files, but that leaves the directory cluttered with files of no interest in normal use. `ltg.bat` removes the prerequisite Fortran files; that leaves the directory tidy but has its own disadvantages (see Chapter 5 [For2exe], page 8).
- Name clashes can arise between Fortran files generated from different TABLO source files. For example, running TABLO on '`gtap.tab`' generates, among other outputs, an auxiliary Fortran file '`gtap_c0.tab`'. If the user also happens to have a TABLO source file '`gtap_c0.tab`', then running TABLO on '`gtap.tab`' and then on '`gtap_c0.tab`' clobbers the auxiliary Fortran file from the first run, while running on '`gtap_c0.tab`' and then '`gtap.tab`' clobbers the main Fortran file from the first run.

- `for2exe` and `ltg.bat` both attempt to remove their own intermediate files, but have some difficulty in determining exactly what those are. For example, in compiling and linking the GTAP model, `ltg.bat` would be liable to mistake a file '`gtap__e3.for`' for one of its own intermediate files, and to delete it. `for2exe` would leave it, but would be liable mistakenly to delete '`gtap__42.for`'. Furthermore, we're not sure that `for2exe` always deletes all its own intermediate files.

As users, we haven't suffered from these difficulties anything worse than directory clutter. As developers, we have to take some stance on them, and we want it to be robust with respect to future changes in GEMPACK.

`tab2exe` addresses these difficulties by presenting `tablo.exe` and `for2exe` as a single operation, and performing it in a temporary directory. In successful use, it creates a temporary directory, copies '*stem*`.tab`' into it from the source directory (where *stem* is the stem of the base name of the TABLO source file), calls `tablo.exe` and `for2exe` from within it, copies to the target directory files '*stem*`.axs`', '*stem*`.axt`', '*stem*`.exe`', '*stem*`.inf`', and '*stem*`.min`'. and removes the temporary directory with all its contents. That ensures that a `tab2exe` run never clutters the user's working directory with intermediate files, and never clobbers extraneous files. By default the target directory is the same as the source directory, that is, the directory containing the TABLO source file, but the '`-t`' option may be used to select a different directory.

In unsuccessful use, if the '`.inf`' file is available, `tab2exe` copies it to the source directory. Depending on where and how the operation aborts, it may or may not retain the temporary directory. Broadly speaking, it retains it whenever it seems likely to provide guidance to the cause of failure. If it retains it, it writes to standard error a message reporting its location.

If `tab2exe` fails because of an unsuccessful TABLO run, and the temporary directory is retained, it may contain files '`tablo.stdout`' and '`tablo.stderr`' recording whatever TABLO wrote to standard output and standard error in that unsuccessful run. In these circumstances, however, a file '`gpxx1.log`' is likely to be present, and to prove more informative.

The batch file '`tabltg.bat`', supplied in current versions of the standard GEMPACK distribution and the Bundle package, provides similar but not identical functionality for the Windows command prompt. It combines `tablo.exe` and `ltg.bat` calls into a single operation, but performs it in the current rather than a temporary directory.

7 A Make file template

In '`/usr/local/share/make`' we provide a Make file template '`cygpack.mk`'. From this or similar templates are written many of our Make description files.

Our main purpose in discussing it here is to describe certain common facilities which those Make files provide to their users. But first, we note certain practices which must be followed by developers in order to provide those features:

- Following standard practice, the dummy target `all` remains the first target in the Make file.
- The prerequisites of `all` are the default targets of the build.
- The variable `fprogs` is a list of stems of names of TABLO source files from which Fortran code is to be generated.
- The variable `gprogs` is a list of stems of names of TABLO source files from which GEMSIM auxiliary files are to be generated.
- The variable `source_files` is a list of program source files.
- The variable `in_files` is a list of data input files.
- The variable `clean_files` is a list of derived files.

To keep code local, we define these variables incrementally. So, for example, the rule for turning lead into gold might be accompanied by incremental assignment statements, in a cluster like:

```
clean_files += gold.log gold.har
in_files += lead.har
source_files += gold.cmf
fprogs += transmute
gold.har : transmute.exe gold.cmf lead.har
        ./transmute.exe -cmf gold.cmf -los gold.log
```

It is not necessary explicitly to add '`transmute.tab`' to `source_files`, or '`transmute.exe`' to `clean_files`; this is accomplished automatically as a consequence of including `transmute` in `fprogs`.

If these practices are followed, then:

- The Make file need contain no explicit rules to generate executable programs or GEMSIM auxiliary files from Fortran source, except for programs requiring special instructions, such as are recorded in a TABLO stored input file.
- The lists `source_files`, `in_files`, and `clean_files` are disjoint.
- The files listed in `source_files` and `in_files` are necessary and jointly sufficient for the build.
- If no extraneous files are present at the beginning of the build, then the files present at the end of the build are just those listed in `source_files`, `in_files`, and `clean_files`.

If these practices are followed, the Make file provides the following facilities:

- To make the default targets, just type:

  ```
  make
  ```

- To remove all derived files:

```
make clean
```

- To write to standard output a list of all program source files:

```
make source_list
```

- To write to standard output a list of all data input files:

```
make in_list
```

- To write to standard output a list of all derived files:

```
make clean_list
```