Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'gie.1' command

$ man gie.1

GIE(1) PROJ GIE(1)

NAME

gie - The Geospatial Integrity Investigation Environment

SYNOPSIS

gie [-hovql [args]] file[s]

DESCRIPTION

gie, the Geospatial Integrity Investigation Environment, is a regres? sion testing environment for the PROJ transformation library. Its pri? mary design goal is to be able to perform regression testing of code that are a part of PROJ, while not requiring any other kind of tooling than the same C compiler already employed for compiling the library.

-h, --help

Print usage information

-o <file>, --output <file>

Specify output file name

-v, --verbose

Verbose: Provide non-essential informational output. Repeat -v for more verbosity (e.g. -vv)

-q, --quiet

Quiet: Opposite of verbose. In quiet mode not even errors are reported. Only interaction is through the return code (0 on suc? cess, non-zero indicates number of FAILED tests)

-l, --list

List the PROJ internal system error codes

--version

Print version number

Tests for gie are defined in simple text files. Usually having the ex? tension .gie. Test for gie are written in the purpose-build command language for gie. The basic functionality of the gie command language is implemented through just 3 command verbs: operation, which defines the PROJ operation to test, accept, which defines the input coordinate to read, and expect, which defines the result to expect.

A sample test file for gie that uses the three above basic commands looks like:

```
<gie>
---------------------------------------------
Test output of the UTM projection
---------------------------------------------
operation +proj=utm +zone=32 +ellps=GRS80
---------------------------------------------
accept    12 55
expect    691_875.632_14   6_098_907.825_05
</gie>
```

Parsing of a gie file starts at <gie> and ends when </gie> is reached. Anything before <gie> and after </gie> is not considered. Test cases are created by defining an operation which accept an input coordinate and expect an output coordinate.

Because gie tests are wrapped in the <gie>/</gie> tags it is also pos? sible to add test cases to custom made init files. The tests will be ignore by PROJ when reading the init file with +init and gie ignores anything not wrapped in <gie>/</gie>.

gie tests are defined by a set of commands like operation, accept and expect in the example above. Together the commands make out the gie command language. Any line in a gie file that does not start with a command is ignored. In the example above it is seen how this can be used to add comments and styling to gie test files in order to make

them more readable as well as documenting what the purpose of the vari?

ous tests are.

Below the gie command language is explained in details.

EXAMPLES

1. Run all tests in a file with all debug information turned on

```
gie -vvvv corner-cases.gie
```

2. Run all tests in several files

```
gie foo bar
```

GIE COMMAND LANGUAGE

operation <+args>

Define a PROJ operation to test. Example:

```
operation proj=utm zone=32 ellps=GRS80
# test 4D function
accept    12 55 0 0
expect    691875.63214  6098907.82501  0  0
# test 2D function
accept    12 56
expect    687071.4391   6210141.3267
```

accept <x y [z [t]]>

Define the input coordinate to read. Takes test coordinate. The coordinate can be defined by either 2, 3 or 4 values, where the first two values are the x- and y-components, the 3rd is the z-component and the 4th is the time component. The number of components in the coordinate determines which version of the op? eration is tested (2D, 3D or 4D). Many coordinates can be ac? cepted for one operation. For each accept an accompanying expect is needed.

Note that gie accepts the underscore (_) as a thousands separa? tor. It is not required (in fact, it is entirely ignored by the input routine), but it significantly improves the readability of the very long strings of numbers typically required in projected coordinates.

See operation for an example.

expect <x y [z [t]]> | <error code>

Define the expected coordinate that will be returned from ac?
cepted coordinate passed though an operation. The expected coor?
dinate can be defined by either 2, 3 or 4 components, similarly
to accept. Many coordinates can be expected for one operation.
For each expect an accompanying accept is needed.
See operation for an example.
In addition to expecting a coordinate it is also possible to ex?
pect a PROJ error code in case an operation can't be created.
This is useful when testing that errors are caught and handled
correctly. Below is an example of that tests that the pipeline
operator fails correctly when a non-invertible pipeline is con?
structed.

```
operation   proj=pipeline step
            proj=urm5 n=0.5 inv
expect      failure pjd_err_malformed_pipeline
```

See gie --list for a list of error codes that can be expected.

tolerance <tolerance>

The tolerance command controls how much accepted coordinates can
deviate from the expected coordinate. This is handy to test that
an operation meets a certain numerical tolerance threshold. Some
operations are expected to be accurate within millimeters where
others might only be accurate within a few meters. tolerance
should

```
operation       proj=merc
# test coordinate as returned by ```echo 12 55 | proj +proj=merc``
tolerance       1 cm
accept          12 55
expect          1335833.89 7326837.72
# test that the same coordinate with a 50 m false easting as determined
# by ``echo 12 55 |proj +proj=merc +x_0=50`` is still within a 100 m
# tolerance of the unaltered coordinate from proj=merc
tolerance       100 m
```

accept 12 55

expect 1335883.89 7326837.72

The default tolerance is 0.5 mm. See proj -lu for a list of pos?

sible units.

roundtrip <n> <tolerance>

Do a roundtrip test of an operation. roundtrip needs a operation and a accept command to function. The accepted coordinate is passed to the operation first in it's forward mode, then the output from the forward operation is passed back to the inverse operation. This procedure is done n times. If the resulting co?

ordinate is within the set tolerance of the initial coordinate, the test is passed.

Example with the default 100 iterations and the default toler?

ance:

operation proj=merc

accept 12 55

roundtrip

Example with count and default tolerance:

operation proj=merc

accept 12 55

roundtrip 10000

Example with count and tolerance:

operation proj=merc

accept 12 55

roundtrip 10000 5 mm

direction <direction>

The direction command specifies in which direction an operation is performed. This can either be forward or inverse. An example of this is seen below where it is tested that a symmetrical transformation pipeline returns the same results in both direc?

tions.

operation proj=pipeline zone=32 step

proj=utm ellps=GRS80 step

```
            proj=utm  ellps=GRS80 inv
        tolerance 0.1 mm
        accept 12 55 0 0
        expect 12 55 0 0
        # Now the inverse direction (still same result: the pipeline is symmetrical)
        direction inverse
        expect 12 55 0 0
```

The default direction is "forward".

ignore <error code>

This is especially useful in test cases that rely on a grid that is not guaranteed to be available. Below is an example of that situation.

```
        operation proj=hgridshift +grids=nzgd2kgrid0005.gsb ellps=GRS80
        tolerance 1 mm
        ignore    pjd_err_failed_to_load_grid
        accept    172.999892181021551 -45.001620431954613
        expect    173                 -45
```

See gie --list for a list of error codes that can be ignored.

require_grid <grid_name>

Checks the availability of the grid <grid_name>. If it is not found, then all accept/expect pairs until the next operation will be skipped. require_grid can be repeated several times to specify several grids whose presence is required.

echo <text>

Add user defined text to the output stream. See the example be? low.

```
        <gie>
        echo ** Mercator projection tests **
        operation +proj=merc
        accept  0   0
        expect  0   0
        </gie>
```

which returns

```
------------------------------------------------------------------------------
Reading file 'test.gie'
** Mercator projection test **
------------------------------------------------------------------------------
total:  1 tests succeeded,  0 tests skipped,  0 tests failed.
------------------------------------------------------------------------------
```

skip Skip any test after the first occurrence of skip. In the example below only the first test will be performed. The second test is skipped. This feature is mostly relevant for debugging when writing new test cases.

```
<gie>
operation proj=merc
accept  0   0
expect  0   0
skip
accept  0   1
expect  0   110579.9
</gie>
```

STRICT MODE

New in version 7.1.

A stricter variant of normal gie syntax can be used by wrapping gie commands between <gie-strict> and </gie-strict>. In strict mode, com? ment lines must start with a sharp character. Unknown commands will be considered as an error. A command can still be split on several lines, but intermediate lines must end with the space character followed by backslash to mark the continuation.

```
<gie-strict>
# This is a comment. The following line with multiple repeated characters too
------------------------------------------------
# A command on several lines must use " \" continuation
operation proj=hgridshift +grids=nzgd2kgrid0005.gsb \
        ellps=GRS80
tolerance 1 mm
```

ignore pjd_err_failed_to_load_grid

accept 172.999892181021551 -45.001620431954613

expect 173 -45

</gie-strict>

BACKGROUND

More importantly than being an acronym for "Geospatial Integrity Inves? tigation Environment", gie were also the initials, user id, and USGS email address of Gerald Ian Evenden (1935--2016), the geospatial vi? sionary, who, already in the 1980s, started what was to become the PROJ of today.

Gerald's clear vision was that map projections are just special func? tions. Some of them rather complex, most of them of two variables, but all of them just special functions, and not particularly more special than the sin(), cos(), tan(), and hypot() already available in the C standard library.

And hence, according to Gerald, they should not be particularly much harder to use, for a programmer, than the sin()'s, tan()'s and hy? pot()'s so readily available.

Gerald's ingenuity also showed in the implementation of the vision, where he devised a comprehensive, yet simple, system of key-value pairs for parameterising a map projection, and the highly flexible PJ struct, storing run-time compiled versions of those key-value pairs, hence mak? ing a map projection function call, pj_fwd(PJ, point), as easy as a traditional function call like hypot(x,y).

While today, we may have more formally well defined metadata systems (most prominent the OGC WKT2 representation), nothing comes close being as easily readable ("human compatible") as Gerald's key-value system.

This system in particular, and the PROJ system in general, was Gerald's great gift to anyone using and/or communicating about geodata.

It is only reasonable to name a program, keeping an eye on the integ? rity of the PROJ system, in honour of Gerald.

So in honour, and hopefully also in the spirit, of Gerald Ian Evenden (1935--2016), this is the Geospatial Integrity Investigation Environ?

ment.

SEE ALSO

proj(1), cs2cs(1), cct(1), geod(1), projinfo(1), projsync(1)

BUGS

A list of known bugs can be found at https://github.com/OSGeo/PROJ/issues where new bug reports can be sub? mitted to.

HOME PAGE

https://proj.org/

AUTHOR

Thomas Knudsen

COPYRIGHT