tab2x Documentation

Release 1.2.6

Harry George

Feb 01, 2019

CONTENTS:

CHAPTER

ONE

TAB2X README

1.1 Objective

To generate Wayne Cripp's "Tab" tablature input using lilypond-style source.

Lute players need to generate tablature in french and italian styles. On Linux the normal music typesetting tool is lilypond, perhaps via frescobaldi. On the one hand, lilypond is easy to use but cannot enter tabulature (string-and-fret) and the lute tablature output is (so far) not very authentic. On the other hand, Cripps's Tab makes much better output but is excrutiatingly tedious to use.

Can we develoop an app to accept lilypond-like input, allow tablature input, generate valid lilypond and Cripps Tab outputs?

1.2 Using Tab2x

Usage:

```
tab2x.py --datadir=$DATADIR --in=$INNAME --out=$OUTNAME

$INNAME  is the input  file, e.g., myfile.hgt
$OUTNAME is the output file. The format is taken from the extension:

     $OUTNAME=myfile.ly   --> output in Lilypond format
     $OUTNAME=myfile.tab  --> output in Tab format
```

See test/go_ly and test/go_tab for full pipelines from hgt to pdf.

CHAPTER TWO

TUTORIAL

Contents

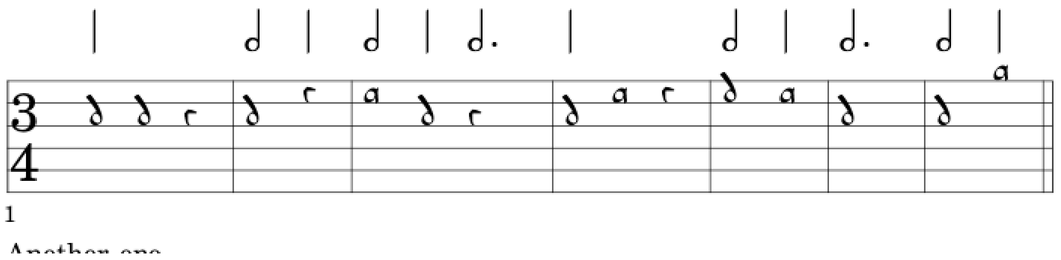
· *transpose*

· *include*

2.1 Why tab2x?

Lute players need “tablature” printouts. Tablature uses fret symbols on string lines instead of the modern stave. Think guitar tablature on steroids.

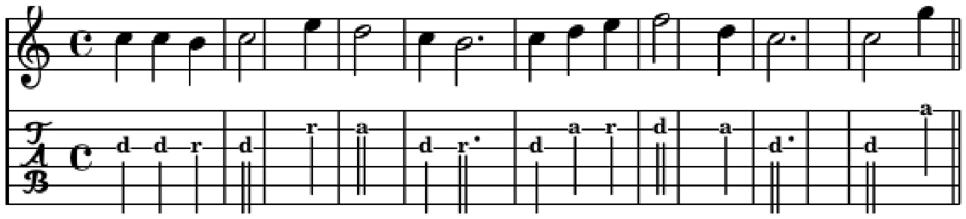
By far the most powerful lute tablature *output* tool is Wayne Cripps’s Tab software.



But the input is difficult:

```
%-----------------------------
{p9 The Coventry Carol}
%
b
S3-4
0-  d
x-  d
x-  c
b
w-  d
0- c
b
w- a
0-  d
w.  c
b
0-  d
x- a
x- c
b
w- d
0- a
b
w.  d
b
w-  d
0-a
bb
```

Lilypond (running under Frescobaldi) lute tablature output is barely usable.:



Lilypond has great programmability and pretty good input. But you cannot enter using tablature format.:

```
%Using myscoreTab function from hgg_lute.py

\myscoreTab c c {
  {
  c''4 c'' b'  \bar "|" c''2 e''4  \bar "|" d''2 c''4 b'2.  \bar "|"
  c''4 d'' e''  \bar "|" f''2 d''4  \bar "|" c''2.  \bar "|"
  c''2 g''4  \bar "||" }
  }  "p9 The Coventry Carol"
```

We need a way to enter music in mixed string/fret and modern notation, and then output to Tab or Lilypond. Thus get the best of both worlds. Further, we want the programmability and structuring features of Lilypond (reusable phrases, predefined functions, etc.).

`tab2x` is not a replacement for either Lilypond or Tab. It just makes them easier to use for a lute player. It does:

- Mixed string/fret and modern notation notes, with fret-hand and pluck-hand fingering markups.
- Multiple voices (e.g., melody and bass line), without struggling with Tab's "extension" mechanism `a&( )`.
- Multiple pieces of music in a given file, each with own title and settings.

By-the-way (BTW): The input format `.hgt` means "Harry George's Tablature", to distinguish it from the other formats out there.

2.2 Getting Started

2.2.1 Install Frescobaldi

See your distribution's install process, or go to Frescobaldi's site.

2.2.2 Install W Cripps's Tab

This is a classic *NIX config. . . make. . . .install process. Read the instructions at his site.

2.2.3 Install Python3

See your distribution's install process, or go to Python's site.

2.2.4 Install tab2x

tab2x is a normal python package, in a tar.gz tarball. It was developed and tested on Linux with python3. Should work immediately in Linux or in MS Windows+cygwin with python3 installed. On MS Windows native, the shell scripts will have to be revised to use whatever MS Windows uses for shell scripts these days.

Expand the tarball to get:

```
tab2x
  doc/
    build/
      html/
        index.html         #documentation <-- START HERE
        (various html files)
  data/
    Peg/
      peg1.hgt               #input file
      go_ly                  #hgt...lilypond...pdf...view
      go_tab                 #hgt...tab...pdf...view
    Scales/
      scales_major.hgt       #input file
      scales_minor.hgt       #input file
      go_ly                  #"go_ly major"  or "go_ly_minor" ...view
      go_tab                 #"go_tab major" or "go_tab_minor"...view

  tab2x/
    (the actual package)

  test/
    go_install             #typical installation process
    go_test                #run tests
    all_tests.py
    hgt_tests.py
    ly_tests.py
    tab_tests.py

    testdata/
      test_(testcase).hgt   #various testcases
      ly/                   #ly results and oracles
      tab/                  #tab results and oracles
      txt/                  #txt results and oracles
```

Examine `go_install` and edit as needed. Run it.

2.2.5 Install Lilypond site files

The Lilypond output is dramatically simplified by using site files with predefined functions. Copy the support/hgg.. files to your Lilypond site dir (e.g., /usr/share/lilypond/site)

Extra Credit: Read the support files to see if there is anything useful on other projects.

2.2.6 Use it

See the `data` dir. Examine `go_ly` and `go_tab`. Edit as needed. Run them to confirm you have a working pipeline from hgt to viewable pdf.

Set up a similar directory for your own work.

2.3 Basic framework

Use this framework for your work. Lilypond users will feel at home. Tab users will hopefully catch on quickly. Basic file framework:

```
book {
lyversion="2.21.0"
title="This is the top title"
revision="New"

{
%music goes here
)

} %end book
```

2.4 Music

2.4.1 Notes

Notes have pitch and duration. Pitch is either in LY-format or string/fret format:

```
c     %Ly-style C, too low for my lute per the stringTuning
c'    %Ly-style C, up one octave
c''   %Ly-style C, up two octave
ees   %Ly-style Eflat  ("es" = flat)
fis   %Ly-style Fsharp ("is" = sharp)
fis'' %Ly-style Fsharp, up two octaves
1a    %Tab-style string 1, fret a (open)
3c    %Tab-style string 3, fret c
3r    %Tab-style string 3, fret c  ("r" is an alias for "c")
r     %rest,    in Ly will show rest mark
s     %silence, in Ly will show nothing but will take space
```

Notes need duration, usually given with the note:

```
c'1   %whole note
c'2   %half note
c'4   %quarter note
c'8   %eighth note
c'16  %16th  note
c'2.  %dotted half
c'4.  %dotted quarter, etc.
```

But this is tedious if several notes have the same duration. If no duration is given, then assume last given duration:

```
c'4 d' e' f''8 g'' a''1

Quarter notes c' d' e'.   Then eighth notes f'' g''
Then whole note a''
```

2.4.2 Chords

Chords wrap several notes in `<...>`. The notes do not get durations, but a duration is given (or assumed) for the whole chord:

```
<c' e' g'>4.
```

2.4.3 Bar lines

tab2x counts note durations and inserts bar lines based on time signature. The user can do 2 things:

- Insert "check bars" ("|") where you think there should be a bar. Tab2x will warn if it doesn't align with the calculated bars.
- Insert double bar ("||") to end a piece. Tab2x will not do this automatically. But it will detect a double bar and avoid putting an automatic bar at the same place.

Thus:

```
c'4 d'2 ees' | 1a4 2b2  3c  |
c'4 1a d' e' | ees' 3c  ||
```

2.4.4 Linebreaks

Both Lilypond and TYab require the user to supply linebreaks – otherwise the line gets too crowded and the tool complains. tab2x does linebreaks on the basis of number of notes, or number of measures.

The default is counting notes. A Book or a Piece can set its own basis and counts:

```
piece {
  title="15-25 notes per line"
  linebreak_basis="notes"
  min_notes_per_line=15
  max_notes_per_line=25

  time="3/4"
  c'4 d'2 ees' | 1a4 2b2  3c  |
  c'4 1a d' e' | ees' 3c  ||

} %end piece

piece {
  title="4 measures per line"
  linebreak_basis="measures"
  measures_per_line=4

  time="3/4"
  c'4 d'2 ees' | 1a4 2b2  3c  |
  c'4 1a d' e' | ees' 3c  ||

} %end piece
```

2.4.5 Slurs

Slurs wrap 2 or more notes, and may cross bar lines. If a slur starts and ends on the same note, it is techincally a tie, but we treat it the same.:

```
a'4 (b'4 e'4 f'4) |  g'2 (d'2 | d'2)
```

Lilypond users: Notice that the HGT slur covers the notes *inside* the "("…")", thus *b'4 e'4 f'4*. Lilypond by contrast would start the with the note *before* the "(", thus *a'4 b'4 e'4 f'4*. Yes, we could have replicated Lilypond treatment here, but that is just too funky to contemplate. Ly_writer converts to the correct (for Ly) placement.

2.4.6 Fingering

Mark the fret-finger and the plucking-finger:

```
\fi \f2 \f3 \f4  %fret-fingers 1,2,3,4
\fp \fi \fm \fa  %pluck-fingers p,i,m,a

e.g., c'4 \f1 d'2 ees' \fm
```

2.4.7 Forced string

Sometimes need to force a non-typical string for the note. E.g., lower string with higher fret – maybe to accommodate other fingering problems. Enter the note in string/fret format. The system will recognize non-normal string (using higher-than-needed fret), and label it as a forced string choice.

2.4.8 Text

To place text above a note, enter it as a quoted text after the note:

```
c'4 d'2 ees' "half close" | 1a4 2b2  3c "full close" |
```

2.4.9 Multivoice

If you need a bass line, there are two approaches

"multivoice" markup:

```
{
<<
  {
    %melody voice goes here
  }
\\
  {
    %bass voice goes here
  }
>>
}
```

This works great in Ly but in Tab it is dicey.

"chord" markup:

```
<d d'>4 e' f' <e g'>4
```

Here, d is the bass note and starts same time as the melody d'. The melody continues on for a while until there is a new bass note in a chord. The visual effect is OK – the lute player will realize the bass line is held until the next bass note arrives.

The chord markup is the simplest right now.

2.5 File structure

We've already introduced the basic "Book" framework, with a book and an embedded phrase of music. But we can do more than that.

2.5.1 Named phrase

Suppose the music is structured as phrases ABABC, maybe with transition notes between. It would be a pain to do this typing out every note. Instead:

```
phraseA={c'4 d' e'}
phraseB={e'4 d' c'}
a_phrase3_with_a_long_name={a''4. a''8}
phraseC=a_phrase3_with_a_long_name  %a more convenient alias

%the music:
\phraseA \phraseB 2b2 "half close"
\phraseA \phraseB 1a2 "full close"
\phraseC
```

2.5.2 Pieces

This is the normal case – a book with one or more "pieces" of music. In fact, we don't bother doing lilypond tablature for anything but pieces. You can choose the style of lilypond tablature:

```
book {

   piece {   %typically with its own settings
   title="Piece 1"
   lytabstyle="simple"    %modern stave plus tablature
                          %tablature just has strings and frets
   key="c \major"
   time="3/4"
   { c'4 d' | e'2. | f' ||}
   } %end piece 1

   piece {
   title="Piece 2"
   lytabstyle="full"    %modern stave plus tablature
                        %tablature includes rhythm markings
   key="ees \major"
   time="4/4"
   { c'4 d' d'| e'2. d' | f'1 ||}
   } %end piece 2
} %end book
```

2.5.3 Code for X

Suppose there is something that needs to be printed to the Ly file or the Tab file, without messing with the whole system. Use the function `codefor`

```
raw1=codefor TXT @{Hello from raw1@}

raw2=codefor LY @{ ees'' @}

raw3=codefor TAB @{
b
[Hello from raw3}
W-b
b
W-c
[another line}
@}

{ \raw1 \raw2 \raw3 ||}
```

When printing to LY, only raw2 will be in the ly file. When printing to Tab, only raw3 will be in the tab file. Often this is used for settings, e.g. to choose french vs italian tablature.

2.5.4 Builtin Functions

These are defined in functions.py. If you want to add more builtins, see examples bi_samp01, bisamp02. And of course transpose and include.

transpose

Transpose a phrase. Arguments are provided in line with the function:

```
... \transpose (initial pitch) (final pitch) (what to transpose)
```

E.g., transpose phraseA up by an octave. Replace the function and its arguments with the result:

```
... \transpose c c' \phraseA
```

The definition of phraseA is left as-is, in case you need to use it elsewhere.

include

Include a file written in hgt format. Arguments are in line:

```
... \include (filename) ...
```

E.g.,

```
... \include "testdata/samp01.hgt" ...
```

The function reads the file, parses it to a chunk_seq, and inserts that chunk_seq where the ftn and its arg were.

CHAPTER THREE

DEVELOPER'S GUIDE

Contents

3.1 Sitemap

```
tab2x
  AUTHORS
  CHANGES     #points to doc/build/index
  COPYING
  INSTALL     #how to install
  README      #points to doc/build/README
  dist/       #distribution packaging
  data/       #sample data
   Peg/
     peg1.hgt              #input file
     go_ly                 #hgt...lilypond...pdf...view
     go_tab                #hgt...tab     ...pdf...view
   Scales/
     scales_major.hgt      #input file
     scales_minor.hgt      #input file
     go_ly                 #"go_ly major"  or "go_ly_minor" ...view
     go_tab                #"go_tab major" or "go_tab_minor"...view

  setup.py
  scripts/
     tab2x.py      #hgt in... ly or tab out
  tab2x/
    hgt_parser.py #ply-based lexer/parser for hgt format
    model.py      #AST objects, and their supporting functions.
    instrument.py #string tuning et all for instruments
    Ly_writer.py  #outputs .ly files
    Tab_writer.py #outputs .tab files
    Txt_writer.py #outputs .txt files
    (various)     #autogernated by ply
  test/
    go_test       #run all_tests.py
    all_tests.py  #run (fmt) tests, where fmt is txt/ly/tab
    (fmt)_tests.py#hgt in... fmt out
    view_(fmt)    #view (fmt)/result_(testcase).(fmt)
    approve_(fmt) #copy (fmt)/result_(testcase).(fmt)
                     to (fmt)/oracle_(testcase).(fmt)
    testdata/
      test_(testcase).hgt   # test input
      ly/
         result_(testcase).ly
         oracle_(testcase).ly
      tab/
         result_(testcase).tab
         oracle_(testcase).tab
      txt/
         result_(testcase).txt
         oracle_(testcase).txt

    doc/
      go                   #run sphinx for html
      go_pdf               #run Sphinx for latex then pdflatex for PDF
      source/
        conf.py            #Sphinx config
        _static            #templates
        index.rst          #master file
        README.rst         #intro
        devguide.rst       #Developers' Guide (this doc)
        tutorial.rst       #How to use tab2x.py
      build/
        html/
          (various).html #Sphinx output
        latex/
          (various).tex  #Sphinx output
    doc2/
      (same structure as doc, but more esoteric docs)
      source/
        design_history.rst   #day-by-day diary of the project
```

3.2 Architecture

The idea is to parse:

```
book {
  %line comment
  %(
     block comment
  %}

  %phrases may be named or anonymous, with own context for
  %settings, and mixed lynotes, hgtnotes, bars, and phrase refs

  piece {   %printed right here
  title="Piece 1"
   {time="3/4" 6a1 ees''}   %anonymous phrase with own settings
  } %end piece


  piece { %another piece
  title="Piece 2"
  phrase_1= {         %named phrase, saved but not printed here
    c'4 fis''2. g''| 1b4 2c4. ees' ||}
  }

  { a''4 \phrase_1  1b2}  %use the named phrase, with other stuff
  }%end piece

  }%end book
```

The syntax is defined in hgt_parser.py. The AST is defined in model.py The AST is built during the parse and saved on a ContextStack, as Context-inheriting objects (Book, Piece, Voice, Phrase).

hgt_parser.py… Hgt_parser.run() can run just the lexer (for debugging), or can do the full parse. Choice is by "if 0…" vs "if 1…" statement.

The parser returns "result", which can then be manipulated or just dumped out with a "to_(format)" method.

3.3 Edit Cycle

The test paradigm is this:

- tab2x/test/go_test runs all_tests.py.
- all_tests.py runs hgt_tests.py, ly_tests.py, tab_tests.py. Edit it to choose what to run at a given time.
- Each of the format tests (e.g., hgt_tests.py) has a setup/mkfile/check/teardown framework.
  - `mkfile` uses the name of the testcase is used to find the input, e.g. test_book.hgt, to run the parser and then generate output with the appropriate backend (.txt, .ly, or .tab). This is named, e.g., result_book.txt.
  - `check` compares the result file to the corresponding known-good file, e.g., oracle_book.txt. If the files are identical, the test passes, else it fails. This means the output files can;t have changing content (e.g., date/time, temp file names, object ids). If necessary, that can be controlled with the "test_p flag in each module.
- Adding new grammar may require editing some or all of the input files. Adding new output features may change some or all of the output files, requiring you to reassess the oracle.

- After a test run, check for failures (the test didn't even run) – e.g., pythonic syntax errors or a "None" where you expected an object. Fix those.
- When the test runs, but fails, that means the result and the oracle do not match. Examine the result file with `view_(format)`, e.g.:

  ```
  view_tab book
  ```

  This runs the tools to convert the result file into a pdf, and displays the pdf.
- Examine the display. Two cases:
  - Something went wrong. The conversion tool didn't run, or it ran but the result were strange. Examine the result file, and edit and view until that works, and then revise the code to make that happen.
  - It is actually a better output. Accept this new result as the new oracle.:

    ```
    ./approve_tab book
    ```

3.3.1 New feature

Backup tab2x tree, or make a distribution tarball. Or both.

Determine a new feature. Sketch out an example with hgt input and desired ly and tab output.

Make a test case for it in each of the (format)_tests.py files and add to the suite. Edit all_tests to just do hgt_tests, and edit hgt_tests to do just the new test. Run `go_test`.

Once that works, edit hgt_tests.py to do all the available tests. Run `go_test`, and determine if you have mangled something else.

Then edit all_tests.py to run only ly_tests.py. Get that working. Then edit to run only tab_tests.py.

Finally turn on all tests in all formats. go_test should run clean.

3.3.2 Debugging and Troubleshooting

Basic rules:

- Make the app and its testing machine runnable, No visual inspections needed once an oracle has been approved.
- Make small test cases, just big enough to capture the problem
- Arrange to quickly re-create the problem state and examine accessible values. Most bugs are obvious once you isolate them in the problem state and check values of live variables.
- lexer/parser bugs tend to be non-obvious even when you are staring at them. First, just do the lexer scan to see if the right tokens are found. Next, run the parser with debug=True and check parser.out for odd transitions. If there are shift/reduce conflicts, try rethinking your grammar.

Some people use interactive debuggers to get to the problem state and then insert a "watch" flag and read values. That it pretty slow (lots of mouse actions).

Some people throw in a bunch of print statements, and then can't remember where the print was fired. Also, print can't be turned on and off module by module or within a method.

My `debug(ftn,"...")` approach is crude but effective. And it works in every language I've ever used. `go_test` (using small test inputs) takes you immediately to the problem state, and the debug tells your where you are.

3.4 Major Design Decisions

(See also Decision History)

3.4.1 Parser choice

I wanted a Context Free Grammar, and a very pythonic treatment. I've hand-written LL(1) recursive descent in the past, and have use Ply LALR(1) for a couple of projects. Went with Ply as powerful enough for whatever I ran into.

It took several days to get the hang of the tokens and productions. A key insight was how to do (prod)_found (e.g., p_book_found) to initialize an object on the stack before its contents were filled in.

3.4.2 Context Stack

Contexts have their own local namespaces but also need to find namespaces deeper in the stack. After some experiments I went with freezing the Contexts maps as it want onto the stack. So even if it came off the stack and was referenced elsewhere, it remembered its namespaces at creation time.

This is driven by the same considerations as the whole "stackless" runtime approach.

3.4.3 Regex for notes

Notes, rests, silences, and ids all look a lot alike. I ended up doing inclusive regex to get in the ballpark, and then prioritizing how it was interpreted.

Tried doing regexes with/without durations embedded. It looks nice to do the durations separately, until you realize they are almost string/fret notes in themselves, and aren't actually NUMBERs. E.g., is `s2.` a variable name or a silence note? Is a separate `2` a number or a halfnote duration?

So I just grab it all in t_ID and tease it apart from there.

If you decide you can solve this more cleanly... make really good backups before you start. I''ve done a couple of cycles on this one.

3.4.4 Pitch storage

There are lots of ways to capture pitches. I went with (octave,step), because it covers all octaves, does not get confused by accidentals and en-harmonics, and makes scale analysis and transpositions easier.

You can either do (octave,step) and juggle when you go past an octave boundary, or do all steps on a piano (1...88) and use modulo to get to the octave and scalestep. I found (octave,step) easier on my brain.

3.4.5 Lute octaves

See instruments.Lute8.

My 8-course Lute actually sounds down in the bass clef. But we mostly want to read in the treble clef. This is like a male singer reading treble clef alongside a soprano, but sounding an octave lower.

I chose to declare the string tunings 1 octave high, and treat everything else by-the-book. It comes out OK in Lilypond modern, Lilypond tablature, and Tab tablature.

Given that decision we can generate maps (string,fret)<–>(octave,step). These are generated by mk_os2sf, using the chosen instrument string tuning. After running the mk function, copy-and-paste into Lute8 class data section.

When going (octave,step)–>(string,fret) there are of course multiple was to the the pitch. Use a lower string and a higher fret. But we make the maps using the lowest available frets (thus the highest available string), assuming this is easier on the player. It is possible to force the lower-string-higher-fret solution, but we want to mark it as unusual, and provide the string number.

The "real" string tuning is available, commented out, so you can try it in mk_os2sf.

3.4.6 Bars and LineBreaks

You have to do linebreaks in Ly and Tab or else the output becomes cluttered and unreadable/unplayable. There is some reasonable number of notes to place on a given line. Of course you want to break at measure bars. min_notes_per_line and max_notes_per_line are heuristically determined. They should be settable for each Piece.

The right answer is Lilypond's: Autobarring based on time signature, with manually entered check-bars so we humans don't get lost. model.Writer does this.

A Piece eventually ends in a doublebar ("||"). We don't want to do an autobar and linebreak just before this, so we need to lookahead. But lookahead isn't simply a matter of checking the last object in the current chiunk_seq – we might be getting a whole series of chunk-seqs fornm various referenced phrases. Therefore, we do a simple queue of length 1, for pending bar and linebreak. If a doublebar shows up, we ignore the pending. if anything else shows up, we release the pending.

CHAPTER FOUR

CHANGES

Contents

4.1 Kickoff

2018-12-12T:06:54:43: Initialized project. Folllowed by lots of 0.xx development releases.

4.2 1.00 (2019-01-16)

Initial release.

4.3 1.20 (2019-01-18)

Reworked Writer and Tab's Multivoice.

4.4 1.21 (2019-01-19)

Added slurs.

4.5 1.22 (2019-01-20)

Allow Contexts (Book, Piece, Voice, Phrase) to set min/max notes-per-line, thus controlling layout.

Fixed function “transpose” to handle Multivoice.

4.6 1.23 (2019-01-21)

Revised autobar to handle final doublebar. Removed compile date from Ly tests. Converted to Poulan-style interpretation of duration flagging.

- In Cripps’ Tab manual, a vertical bar with not flags is a quarter note
- In Poulan, a vertical bar with no flags is a whole note.

4.7 1.25 (2019-01-27)

Cleaned up autobar, and added linbreaks via measure count.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search