AutoDoc

Generate documentation from GAP source code

2026.11.15

15 November 2026

Sebastian Gutsche
Max Horn

Sebastian Gutsche Email: gutsche@mathematik.uni-siegen.de
Homepage: https://algebra.mathematik.uni-siegen.de/gutsche/
Address: Department Mathematik
Universität Siegen
Walter-Flex-Straße 3
57072 Siegen
Germany

Max Horn Email: mhorn@rptu.de
Homepage: https://www.quendi.de/math
Address: Fachbereich Mathematik
RPTU Kaiserslautern-Landau
Gottlieb-Daimler-Straße 48
67663 Kaiserslautern
Germany

Abstract

AutoDoc is a GAP package whose purpose is to aid GAP package authors in creating and maintaining the documentation of their packages.

Copyright

© 2012–2026 by Sebastian Gutsche and Max Horn

This package may be distributed under the terms and conditions of the GNU Public License Version 2 or (at your option) any later version.

Acknowledgements

This documentation was prepared using the GAPDoc package [LN24].

Contents

1 What AutoDoc offers **4**
- 1.1 Core features . . . 4
- 1.2 Adopting AutoDoc incrementally . . . 4
- 1.3 Where to continue . . . 5

2 Getting started using AutoDoc **6**
- 2.1 Choose your workflow . . . 6
- 2.2 Creating a package manual from scratch . . . 6
- 2.3 Documenting code with AutoDoc . . . 7
- 2.4 Using AutoDoc in an existing GAPDoc manual . . . 9
- 2.5 Scaffolds . . . 12
- 2.6 Worksheets . . . 15

3 AutoDoc documentation comments **16**
- 3.1 Structuring the documentation . . . 17
- 3.2 Documenting declarations . . . 19
- 3.3 Other documentation comments . . . 22
- 3.4 Title page commands . . . 26
- 3.5 Plain text files . . . 27
- 3.6 Grouping . . . 29
- 3.7 Markdown-like formatting of text in AutoDoc . . . 30
- 3.8 Deprecated commands . . . 32

4 Reference **33**
- 4.1 AutoDoc worksheets . . . 33
- 4.2 Extracting manual examples . . . 33
- 4.3 The AutoDoc() function . . . 34

References **42**

Index **43**

Chapter 1

What AutoDoc offers

AutoDoc helps you create and maintain package manuals for GAP. It is built on top of GAPDoc and generates the XML input that GAPDoc consumes. So AutoDoc complements GAPDoc instead of replacing it.

The package is designed for a "mix and match" workflow: you can adopt only the parts that help you today, and add more over time.

1.1 Core features

- Build package manuals via one reproducible command, usually `gap makedoc.g`.
- Generate and maintain a title page from metadata in `PackageInfo.g`.
- Generate and maintain a main XML file that includes your chapters.
- Extract manual examples into `.tst` files via `extract_examples := true`.
- Document declarations directly in source files via AutoDoc comments beginning with `#!`.
- Organize chapter and section text either in source comments or in standalone `.autodoc` files.
- Combine AutoDoc comments, classic GAPDoc source comments, and hand-written XML chapters in one manual.

1.2 Adopting AutoDoc incrementally

You do not have to switch everything at once. Typical adoption paths include:

- Use only `AutoDoc` (4.3.1) to rebuild an existing GAPDoc manual.
- Enable only title-page generation, while keeping your custom main XML.
- Keep your existing title page but use generated entities such as `&VERSION;`, `&RELEASEYEAR;`, and `&RELEASEDATE;`.
- Add AutoDoc comments only for new code, and leave old documentation as-is.

- Keep existing XML chapters, or gradually add source comments and `.autodoc` files where they fit your preferred workflow.
- Later, gradually move chapters or source documentation to your preferred style.

This flexibility is central: AutoDoc should make your current setup better, without forcing a full rewrite first.

1.3 Where to continue

For practical setup and migration workflows, continue with chapter 2. For the command reference of documentation comments, including standalone `.autodoc` files, see chapter 3.

Chapter 2

Getting started using AutoDoc

This chapter gives practical workflows for adding AutoDoc to a package. For a high-level feature overview and adoption strategy, see chapter 1.

2.1 Choose your workflow

You can use AutoDoc in several ways, and it is fine to combine them:

- Start a new package manual from scratch (Section 2.2).
- Integrate AutoDoc into an existing GAPDoc manual (Section 2.4).
- Use only selected features, such as title-page generation or example extraction, while keeping everything else unchanged.

This incremental approach is encouraged: start with the smallest helpful step, then adopt additional features when useful.

2.2 Creating a package manual from scratch

Suppose your package is already up and running, but so far has no manual. Then you can rapidly generate a "scaffold" for a package manual using the `AutoDoc` (4.3.1) command like this:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( scaffold := true ) );
```

When called from a file such as `makedoc.g`, AutoDoc uses the directory containing that file as the package directory. Otherwise, it falls back to the current directory. In either case it then reads the `PackageInfo.g` file from that package directory. It extracts information about the package from it (such as its name and version, see Section 2.5.1). It then creates two XML files `doc/_main.xml` and `doc/title.xml` inside the package directory. Finally, it runs GAPDoc on them to produce a nice initial PDF and HTML version of your fresh manual.

To ensure that the GAP help system picks up your package manual, you should also add something like the following to your `PackageInfo.g`:

```
PackageDoc := rec(
  BookName  := ~.PackageName,
  ArchiveURLSubset := ["doc"],
  HTMLStart := "doc/chap0.html",
  PDFFile   := "doc/manual.pdf",
  SixFile   := "doc/manual.six",
  LongTitle := ~.Subtitle,
),
```

Congratulations, your package now has a minimal working manual. Of course it will be mostly empty for now, but it already should contain some useful information, based on the data in your `PackageInfo.g`. This includes your package's name, version and description as well as information about its authors. And if you ever change the package data, (e.g. because your email address changed), just re-run the above command to regenerate the two main XML files with the latest information.

Next of course you need to provide actual content (unfortunately, we were not yet able to automate *that* for you, more research on artificial intelligence is required). To add more content, you have several options: You could add further GAPDoc XML files containing extra chapters, sections and so on. Or you could use classic GAPDoc source comments. For details on either, please refer to (**GAPDoc: Distributing a Document into Several Files**), as well as Section 2.4 of this manual on how to teach the `AutoDoc` (4.3.1) command to include this extra documentation. Or you could use the special documentation facilities AutoDoc provides (see Section 2.3).

You will probably want to re-run the `AutoDoc` (4.3.1) command frequently, e.g. whenever you modified your documentation or your `PackageInfo.g`. To make this more convenient and reproducible, we recommend putting its invocation into a file `makedoc.g` in your package directory, with content based on the following example:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( autodoc := true ) );
QUIT;
```

Then you can regenerate the package manual from the command line with the following command:

```
gap makedoc.g
```

If you also want regression tests from your manual examples, enable `extract_examples := true`; this is explained in Section 2.4.1.

2.3 Documenting code with AutoDoc

AutoDoc supports several equally supported ways to organize your manual text. You can put chapter and section material directly into `#!` comments inside `.g`, `.gd`, or `.gi` files, you can put it into standalone `.autodoc` files, and you can mix either style with existing GAPDoc XML. Which arrangement is best is mostly a matter of taste and convenience. The detailed command reference for these styles is in chapter 3, especially Section 3.5.

To get one of your global functions, operations, attributes etc. to appear in the package manual, simply insert an AutoDoc comment of the form `#!` directly in front of it. For example:

```
#!
DeclareOperation( "ToricVariety", [ IsConvexObject ] );
```

This tiny change is already sufficient to ensure that the operation appears in the manual. In general, you will want to add further information about the operation, such as in the following example:

Important: the comment block must be immediately followed by the declaration it documents. Do not place other code between the comment block and the `Declare...` statement.

```
#! @Arguments conv
#! @Returns a toric variety
#! @Description
#!  Creates a toric variety out
#!  of the convex object <A>conv</A>.
DeclareOperation( "ToricVariety", [ IsConvexObject ] );
```

In these comment lines, you can freely use normal GAPDoc XML markup (with the usual exceptions for lines starting with `@`, which are interpreted as AutoDoc commands). So, for instance, tags like `<Ref>`, `<A>`, `<K>`, `<List>`, etc. can be written directly in `#!` comment text.

For chapter text, tutorial material, worked examples, and similar prose, some authors prefer to keep the material next to their code using `#! @Chapter`, `#! @Section`, and related commands, while others prefer standalone `.autodoc` files. In an `.autodoc` file you write the same commands without the `#!` prefix, and you list the file in `autodoc.files` or place it in a directory scanned via `autodoc.scan_dirs`. AutoDoc itself still uses the source-comment style in places, for example in `gap/Magic.gd`.

One caveat applies if you decide to keep prose in a standalone `.autodoc` file but want to interrupt it with the documentation of a declaration that is written in source comments. Today that requires the chunk mechanism, and that workflow is currently limited; see issue #60 in the AutoDoc issue tracker.

For a thorough description of what you can do with AutoDoc documentation comments, please refer to chapter 3.

Suppose you have not been using GAPDoc before but instead used the process described in section 2.2 to create your manual. Then the following GAP command will regenerate the manual and automatically include all newly documented functions, operations etc.:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( scaffold := true,
              autodoc := true ) );
```

If you are not using the scaffolding feature, e.g. because you already have an existing GAPDoc based manual, then you can still use AutoDoc documentation comments and standalone `.autodoc` files. Just make sure to first edit the main XML file of your documentation, and insert the line

```
<#Include SYSTEM "_AutoDocMainFile.xml">
```

in a suitable place. This means that you can mix AutoDoc documentation comments and `.autodoc` files freely with your existing documentation; you can even still make use of any existing GAPDoc documentation comments in your code.

The following command should be useful for you in this case; it still scans the package code for AutoDoc documentation comments and then runs GAPDoc to produce HTML and PDF output, but does not touch your documentation XML files otherwise.

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( autodoc := true ) );
```

2.4 Using AutoDoc in an existing GAPDoc manual

Even if you already have an existing GAPDoc manual, it might be interesting for you to use AutoDoc for two purposes:

First, with AutoDoc it is very convenient to regenerate your documentation.

Secondly, the scaffolding feature which generates a title page with all the metadata of your package in a uniform way is very handy. The somewhat tedious process of keeping your title page in sync with your `PackageInfo.g` is fully automated this way (including the correct version, release data, author information and so on).

There are various examples of packages using AutoDoc for this purpose only, e.g. io and orb.

In particular, this setup works well if you want to keep your existing manual structure and only adopt selected AutoDoc features first; for example automatic title-page data and predefined entities such as `&VERSION;`, `&RELEASEYEAR;` and `&RELEASEDATE;` (see Section 2.5.4).

2.4.1 Using AutoDoc on a complete GAPDoc manual

Suppose you already have a complete XML manual, with some main and title XML files and some documentation for operations distributed over all your `.g`, `.gd`, and `.gi` files. Suppose the main XML file is named `PACKAGENAME.xml` and is in the `/doc` subdirectory of your package. Then you can rebuild your manual by executing the following two GAP commands from a GAP session started in the root directory of your package:

```
LoadPackage( "AutoDoc" );
AutoDoc( );
```

Note that in particular, you do not have to worry about keeping a list of your implementation files up-to-date.

But there is more. AutoDoc can create `.tst` files from the examples in your manual to test your package. This can be achieved via

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( extract_examples := true ) );
```

Now files `PACKAGENAME01.tst`, `PACKAGENAME02.tst` and so appear in the `tst/` subdirectory of your package, and can be tested as usual using `Test` (**Reference: Test**) respectively `TestDirectory` (**Reference: TestDirectory**).

If you prefer to keep generated tests in a separate location, set `extract_examples.subdir` to a path relative to the package root:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( extract_examples := rec( subdir := "tst/generated" ) ) );
```

This writes the extracted examples into `tst/generated/` instead.

Either way the generated `.tst` files have to be regenerated whenever the manual changes, which tempts one into committing them. To avoid that, extract them when the tests run, using `AutoDocExtractExamples` (4.2.1) in your `tst/testall.g`:

```
LoadPackage( "mypkg" );
dirs := DirectoriesPackageLibrary( "mypkg", "tst" );
Add( dirs, AutoDocExtractExamples( "mypkg" ) );
TestDirectory( dirs, rec( exitGAP := true ) );
```

This reads your `makedoc.g`, so the settings describing the manual are not duplicated, but skips building the manual itself. Everything is written to a temporary directory, so the extracted tests need not be committed and the tests also run from a read-only package directory.

2.4.2 Setting different GAPDoc options

Sometimes, the default values for the GAPDoc command used by AutoDoc may not be suitable for your manual.

Suppose your main XML file is *not* named `PACKAGENAME.xml`, but rather something else, e.g. `main.xml`. Then you can tell AutoDoc to use this file as the main XML file via

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( gapdoc := rec( main := "main" ) ) );
```

AutoDoc can scan directories for documentation input automatically. In fact there are two separate scanning steps. The option `autodoc.scan_dirs` controls where it looks for source comments beginning with `#!` and for standalone `.autodoc` files. By default, it scans the package root directory and the subdirectories `gap`, `lib`, `examples` and `examples/doc`; the listed subdirectories are scanned recursively, while the package root itself is only scanned at top level. If you keep that kind of input in other directories, adjust `autodoc.scan_dirs`. The following example instructs AutoDoc to only search in the subdirectory `package_sources` of the package's root directory for those files.

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( autodoc := rec( scan_dirs := [ "package_sources" ] ) ) );
```

The separate option `gapdoc.scan_dirs` serves a different purpose: it controls where GAPDoc comments are searched for.

You can also specify an explicit list of files containing documentation, which will be searched in addition to any files located within the scan directories:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( autodoc := rec( files := [ "path/to/some/hidden/file.gd" ] ) ) );
```

Giving such a file does not prevent the standard `scan_dirs` from being scanned for other files.

Next, GAPDoc supports the documentation to be built with relative paths. This means, links to manuals of other packages or the GAP library will not be absolute, but relative from your documentation. This can be particularly useful if you want to build a release tarball or move your GAP installation around later. Suppose you are starting GAP in the root path of your package as always,

and the standard call of AutoDoc (4.3.1) will then build the documentation in the doc subdirectory of your package. From this directory, the gap root directory has the relative path ../../... Then you can enable the relative paths by

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( gapdoc := rec( gap_root_relative_path := "../../.." ) ) );
```

or, since ../../.. is the standard option for gap_root_relative_path, by

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( gapdoc := rec( gap_root_relative_path := true ) ) );
```

The same behavior is also available via the global option relativePath. This is particularly convenient in a short makedoc.g script, and it overrides gapdoc.gap_root_relative_path if both are given. Since this is a GAP global option, you can also set it outside the eventual AutoDoc (4.3.1) call and let it propagate through nested calls; see (**Reference: Options Stack**) for more information about GAP's global options system:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( autodoc := true ) : relativePath := "../../.." );
```

If you pass relativePath := true, then AutoDoc uses the default relative path ../../...

For example, if your makedoc.g reads the manual via AutoDoc (4.3.1), then the following command sets both relativePath and nopdf to true for that nested call:

```
Read( "makedoc.g" : nopdf, relativePath );
```

Finally, if you only want HTML and text output, you can suppress PDF generation with the global option nopdf:

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( autodoc := true ) : nopdf );
```

This is useful if pdflatex is unavailable, or when you want a faster documentation rebuild while editing.

You can also request the same behavior from the shell by setting the environment variable NOPDF before invoking makedoc.g. For example:

```
NOPDF=1 gap makedoc.g
```

If NOPDF is set, AutoDoc skips PDF generation even if no nopdf option is given in the GAP code.

2.4.3 Checklist for converting an existing GAPDoc manual to use AutoDoc

Here is a checklist for authors of a package PackageName, *which already has a GAPDoc based manual*, who wish to use AutoDoc to build the manual from now on. We assume that the manual is currently built by reading a file makedoc.g and that the main .xml file is named manual.xml.

The files PackageInfo.g, makedoc.g, doc/title.xml and doc/_main.xml (if it exists) will be altered by this procedure, so it may be wise to keep backup copies.

You should have copies of the AutoDoc files PackageInfo.g and makedoc.g to hand when reading these instructions.

- Copy `AutoDoc/makedoc.g` to `PackageName/makedoc.g`.
- Edit `PackageName/makedoc.g` as follows.
  - Change the header comment to match other files in your package.
  - After `LoadPackage("AutoDoc");` add `LoadPackage("PackageName");`.
  - In the `AutoDoc` record delete `autodoc := true;`.
  - In the `scaffold` record change the `includes` list to be the list of your `.xml` files that are contained in `manual.xml`.
  - If you do not have a bibliography you may delete the `bib := "bib.xml",` field in the scaffold. Otherwise, edit the file name if you have a different file. If you only have a `.bib` file (`manual.bib` or `bib.xml.bib` say) you should rename this file `PackageName.bib`.
  - In the `LaTeXOptions` record, which is in the `gapdoc` record, enter any LaTeX `newcommands` previously in `manual.xml`. (If there are none you may safely delete this record.) To illustrate this option, the AutoDoc file `makedoc.g` defines the command `\bbZ` by `\newcommand{\bbZ}{\mathbb{Z}}`, which may be used to produce the LaTeX formula $f : \mathbb{Z}^2 \to \mathbb{Z}$. However, note that this only works in the PDF version of the file.
- Now edit `PackageName/PackageInfo.g` as follows.
  - Delete any `PKGVERSIONDATA` chunk that may be there. One reason for converting your manual to use AutoDoc is to stop using entities such as `PACKAGENAMEVERSION`.
  - Copy the `AutoDoc` record from `AutoDoc/PackageInfo.g` to the end of your file (just before the final `"));"`.
  - Replace the `Copyright`, `Abstract` and `Acknowledgements` fields of the `TitlePage` record with the corresponding material from your `manual.xml`. (If you do not have an abstract, then delete the `Abstract` field, etc.)
  - In the `entities` record enter any entities previously stored in your `manual.xml`. (Again, if you have none, you may safely delete this record.) To illustrate this option the AutoDoc file `PackageInfo.g` defines entities for the names of packages io and PackageName.
- If you are using a GitHub repository, as well as running "`git add`" on files `makedoc.g`, `PackageInfo.g` and `doc/PackageName.bib`, you should run "`git rm -f`" on files `doc/manual.xml`, and `doc/title.xml`.

You should now be ready to run GAP and `Read("makedoc.g");` in your package root directory.

2.5 Scaffolds

2.5.1 Generating a title page

For most (if not all) GAP packages, the title page of the package manual lists information such as the release date, version, names and contact details of the authors, and so on.

All this data is also contained in your `PackageInfo.g`, and whenever you make a change to that file, there is a risk that you forget to update your manual to match. And even if you don't forget it, you of course have to spend some time to adjust the manual. GAPDoc can help to a degree with this via entities. Thus, you will sometimes see code like this in `PackageInfo.g` files:

```
Version        := "1.2.3",
Date           := "20/01/2015",
##  <#GAPDoc Label="PKGVERSIONDATA">
##  <!ENTITY VERSION "1.2.3">
##  <!ENTITY RELEASEDATE "20 January 2015">
##  <!ENTITY RELEASEYEAR "2015">
##  <#/GAPDoc>
```

However, it is still easy to forget both of these versions. And it doesn't solve the problem of updating package authors addresses. Neither of these is a big issue, of course, but there have been plenty examples in the past where people forget either of these two things, and it can be slightly embarrassing. It may even require you to make a new release just to fix the issue, which in our opinion is a sad waste of your valuable time.

So instead of worrying about manually synchronising these things, you can instruct AutoDoc to generate a title page for your manual based on the information in your `PackageInfo.g`. The following commands do just that (in addition to building your manual), by generating a file called `doc/title.xml`.

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( scaffold := rec( MainPage := false ) ) );
```

Note that this only outputs `doc/title.xml` but does not touch any other files of your documentation. In particular, you need to explicitly include `doc/title.xml` from your main XML file.

However, you can also tell AutoDoc to maintain the main XML file for you, in which case this is automatic. In fact, this is the default if you enable scaffolding; the above example command explicitly told AutoDoc not to generate a main page.

2.5.2 Generating the main XML file

The following generates a main XML file for your documentation in addition to the title page. The main XML file includes the title page by default, as well as any documentation generated from AutoDoc documentation comments.

```
LoadPackage( "AutoDoc" );
AutoDoc( rec( scaffold := true ) );
```

You can instruct AutoDoc to include additional XML files by giving it a list of filenames, as in the following example:

```
LoadPackage( "AutoDoc" );
AutoDoc(rec(
    scaffold := rec(
        includes := [ "somefile.xml", "anotherfile.xml" ]
    )
));
```

For more information, please consult the documentation of the `AutoDoc` (4.3.1) function.

2.5.3 What data is used from `PackageInfo.g`?

AutoDoc can use data from `PackageInfo.g` in order to generate a title page. Specifically, the following components of the package info record are taken into account:

PackageName
This is used to set the `<Title>` element of the title page.

Subtitle
This is used to set the `<Subtitle>` element of the title page.

Version
This is used to set the `<Version>` element of the title page, with the string "Version " prepended.

Date This is used to set the `<Date>` element of the title page.

Persons
This is used to generate `<Author>` elements in the generated title page.

PackageDoc
This is a record (or a list of records) which is used to tell the GAP help system about the package manual. Currently AutoDoc extracts the value of the `PackageDoc.BookName` component and then passes that on to GAPDoc when creating the HTML, PDF and text versions of the manual.

AutoDoc
This record controls extra settings used by AutoDoc while generating the manual. Specifically, its entries are used as default values for the entries of `optrec.scaffold` in `AutoDoc` (4.3.1) (see 4.3.3). In particular, `PkgInfo.AutoDoc.TitlePage` lets you add or override title-page elements coming from package metadata.

Typical fields you may set there include `TitleComment`, `Abstract`, `Copyright`, `Acknowledgements` and `Colophon`. For example, in `PackageInfo.g`:

```
SetPackageInfo( rec(
  ...
  AutoDoc := rec(
    TitlePage := rec(
      Copyright := "(C) 2026 Jane Doe",
      Acknowledgements := "Funded by Example Grant 1234."
    )
  )
) );
```

This inserts `<Copyright>` and `<Acknowledgements>` blocks into the generated `title.xml`.

`PkgInfo.AutoDoc.TitlePage` has exactly the same meaning as `scaffold.TitlePage` in `AutoDoc` (4.3.1) (see 4.3.3), which points back to this subsection.

2.5.4 Entities from `PackageInfo.g` and scaffold options

Besides title-page fields, you can define custom entities in `AutoDoc.entities` inside `PackageInfo.g`. They are written to `doc/_entities.xml`, so they can be used both by generated main files and by hand-written main XML files.

In addition, AutoDoc predefines the entities `VERSION`, `RELEASEYEAR` and `RELEASEDATE`, derived from package metadata. This is useful if you keep a custom title page or custom chapters and still want release information to stay synchronized with `PackageInfo.g`.

You can specify scaffold-related settings in `PackageInfo.g` and in your `makedoc.g` call at the same time; both records are merged, and values from `makedoc.g` take precedence when both define the same key.

2.6 Worksheets

For stand-alone documents that are not tied to a package, use `AutoDocWorksheet` (4.1.1). Its documentation and examples are in Section 4.1.

Chapter 3

AutoDoc documentation comments

You can document declarations of global functions and variables, operations, attributes etc. by inserting *AutoDoc* comments into your sources before these declarations. An AutoDoc comment always starts with `#!`. This is also the smallest possible AutoDoc command. If you want your declaration documented, just write `#!` at the line before the documentation. For example:

```
#!
DeclareOperation( "AnOperation",
                  [ IsList ] );
```

This will produce a manual entry for the operation `AnOperation`.

For declaration documentation, the associated declaration must appear immediately after the AutoDoc comment block. In particular, do not insert other code (such as `if false then` or assignments) between the comment block and the `Declare...` statement.

This also works for `InstallMethod` and `InstallOtherMethod`. In that case, AutoDoc uses the installed method's name and filter list to create a manual entry for the implemented item. For example,

```
#! @Description
#!  Returns the number of elements of <A>L</A>.
#! @Arguments L
#! @Returns a nonnegative integer
InstallMethod( CountElements, "for a list", [ IsList ],
    L -> Length( L ) );
```

leads to this:

```
<ManSection>
  <Meth Arg="L" Name="CountElements" Label="for IsList"/>
 <Returns>a nonnegative integer
</Returns>
 <Description>
  Returns the number of elements of <A>L</A>.
 </Description>
</ManSection>
```

The label is derived from the filter list; the method's description string `"for a list"` is not used.

Inside of AutoDoc comments, *AutoDoc commands* starting with `@` can be used to control the output AutoDoc produces. Any comment line that does *not* start with an AutoDoc command is treated as regular documentation text and may contain (almost) arbitrary GAPDoc XML markup, such as `<Ref>`, `<A>`, `<List>`, and similar tags. This lets you use the normal GAPDoc formatting toolbox directly inside AutoDoc comments.

For example:

```
#! @Description
#!  See <Ref Chap="Chapter_Tutorials"/> for setup details.
#!  The argument <A>obj</A> must satisfy <K>IsObject</K>.
```

As explained in chapter 1, you can combine AutoDoc comments with hand-written XML and classic GAPDoc comments. For practical setup and migration workflows, see chapter 2.

3.1 Structuring the documentation

The commands in this section create chapters, appendices, sections and subsections, and select where subsequent text and manual entries go.

3.1.1 `@Chapter` *name*

Sets the active chapter, all subsequent functions which do not have an explicit chapter declared in their AutoDoc comment via `@ChapterInfo` will be added to this chapter. Also all text comments, i.e. lines that begin with `#!` without a command, and which do not follow after `@Description`, will be added to the chapter as regular text; see 3.1.5 for the details. Additionally, the chapters label will be set to `Chapter_`*name*.

Example:

```
#! @Chapter My chapter
#!  This is my chapter.
#!  I document my stuff in it.
```

The `@ChapterLabel` *label* command can be used to set the label of the chapter to `Chapter_`*label* instead of `Chapter_`*name*.

Additionally, the chapter will be stored as `_Chapter_`*label*`.xml`.

The `@ChapterTitle` *title* command can be used to set a heading for the chapter that is different from *name*. Note that the title does not affect the label.

If you use all three commands, i.e.,

```
#! @Chapter name
#! @ChapterLabel label
#! @ChapterTitle title
```

`title` is used for the headline, `label` for cross-referencing, and `name` for setting the same chapter as active chapter again.

3.1.2 @Appendix *name*

This is analogous to @Chapter, but generates Appendix elements instead of Chapter elements. When scaffolding generates the main XML file, appendices created this way are included automatically after any files listed in scaffold.appendix.

Example:

```
@Appendix Supplementary material

@Section Additional tables
```

3.1.3 @Section *name*

Sets an active section like @Chapter sets an active chapter. The section automatically ends with the next @Section or @Chapter command.

```
#! @Section My first manual section
#!  In this section I am going to document my first method.
```

The @SectionLabel *label* command can be used to set the label of the section to Section_*label* instead of Chapter_chaptername_Section_*name*.

The @SectionTitle *title* command can be used to set a heading for the section that is different from *name*.

```
#! @Section A rather long section name
#! @SectionLabel ShortSec
#! @SectionTitle A short heading
```

Other parts of the manual can now refer to this section via <Ref Sect="Section_ShortSec"/>.

3.1.4 @Subsection *name*

Sets an active subsection like @Section sets an active section. The subsection automatically ends with the next @Subsection, @Section or @Chapter command. It also ends with the next documented function. Indeed, internally each function "manpage" is treated like a subsection.

```
#! @Subsection My first manual subsection
#!  In this subsection I am going to document my first example.
```

The @SubsectionLabel *label* command can be used to set the label of the subsection to Subsection_*label* instead of Chapter_chaptername_Section_sectionname_Subsection_*name*.

The @SubsectionTitle *title* command can be used to set a heading for the subsection that is different from *name*.

```
#! @Subsection Details on the input
#! @SubsectionLabel Input
#! @SubsectionTitle Input
```

This subsection is referred to via <Ref Subsect="Subsection_Input"/>.

3.1.5 Where plain text goes

A *plain text line* is a line of an AutoDoc comment, or of a `.autodoc` file, that is neither an AutoDoc command nor part of the documentation of a declaration (such as the text after `@Description`), an example, a chunk or a similar block. Such a line is appended to the active subsection, or if there is none to the active section, or else to the active chapter. In detail:

- Files are read one after another, in the order described for the `autodoc.files` and `autodoc.scan_dirs` options of `AutoDoc` (4.3.1). Each file starts with no active chapter, section or subsection.
- Text read while no chapter is active is discarded, and AutoDoc prints a warning.
- Activating a chapter, section or subsection that already exists, for example by repeating `@Chapter` *name* in another file, appends to its end.
- A documented declaration ends the active subsection. Text after it is added to the section, after the manual entry.
- Text appended to a chapter after one of its sections, or to a section after one of its subsections, is placed after that section resp. subsection. As GAPDoc marks where a section begins but not where it ends, readers see this text as part of the last section resp. subsection, so AutoDoc prints a warning.

For example, if one file contains

```
#! @Chapter Intro
#! @Section Basics
#! Some basic facts.
```

and a file read later contains

```
#! @Chapter Intro
#! More introductory text.
```

then “More introductory text.” appears at the end of section “Basics”. To avoid this, put all text that introduces a chapter before its first section, or move it into a section of its own.

3.2 Documenting declarations

In the bare form above, the manual entry for `AnOperation` will not contain much more than the name of the operation. In order to change this, there are several commands you can put into the AutoDoc comment before the declaration. Currently, the following commands are provided:

3.2.1 @Description *descr*

Adds the text in the following lines of the AutoDoc to the description of the declaration in the manual. Lines are until the next AutoDoc command or until the declaration is reached.

```
#! @Description
#!  Returns the largest entry of the list <A>L</A>.
#!  The list must not be empty.
DeclareOperation( "LargestEntry", [ IsList ] );
```

3.2.2 @Returns *ret val*

The string *ret_val* is added to the documentation, with the text "Returns: " put in front of it. This should usually give a brief hint about the type or meaning of the value returned by the documented function.

```
#! @Returns a positive integer
DeclareAttribute( "Exponent", IsGroup );
```

3.2.3 @Arguments *args*

The string *args* contains a description of the arguments the function expects, including optional parts, which are denoted by square brackets. The argument names can be separated by whitespace, commas or square brackets for the optional arguments, like "grp[, elm]" or "xx[y[z]]". If GAP options are used, this can be followed by a colon : and one or more assignments, like "n[, r]: tries := 100".

```
#! @Arguments G[, n]: tries := 100
DeclareGlobalFunction( "RandomGenerators" );
```

For DeclareGlobalName, using @Arguments or @Returns also makes AutoDoc document the item as a function. This is useful because DeclareGlobalName itself does not reveal whether the name denotes a function or a variable.

3.2.4 @ItemType *kind*

Overrides the kind of manual item created for the following declaration or installed method. The supported values are Attr, Cat, Coll, Constr, Fam, Filt, Func, InfoClass, Meth, Oper, Prop, Repr, and Var.

The values Cat, Coll, and Repr are AutoDoc-specific aliases. They emit Filt entries with the corresponding GAPDoc filter type.

This is useful when the source code alone does not determine which manual item kind should be emitted. For many declarations such as DeclareAttribute or DeclareProperty, AutoDoc already knows the intended type from the declaration itself.

It is useful for DeclareGlobalName, because that declaration can refer to either a function or a variable. AutoDoc defaults such entries to Var, but switches to Func as soon as @Arguments or @Returns indicates function-style documentation. You can still use @ItemType to override this explicitly.

It is useful for `DeclareSynonym`, because that declaration can document a function synonym or one of several filter-like synonyms. Use `@ItemType Filt`, `Cat`, `Coll`, or `Repr` to control which kind of filter entry should be emitted.

For example:

```
#! @ItemType Repr
DeclareSynonym( "IsSomethingRep", IsComponentObjectRep );
```

This makes AutoDoc emit a `<Filt Type="Representation" .../>` manual entry.

3.2.5 `@Group` *grpname*

Adds the following method to a group with the given name. See section 3.6 for more information about groups.

```
#! @Description
#!  The width of <A>obj</A>.
#! @Arguments obj
#! @Group Dimensions
DeclareAttribute( "Width", IsObject );

#! @Arguments obj
#! @Group Dimensions
DeclareAttribute( "Height", IsObject );
```

Both attributes end up in one manual entry, with the description given once.

3.2.6 `@Label` *label*

Adds label to the function as label.

If this is not specified, then for declarations that involve a list of input filters (as is the case for `DeclareOperation`, `DeclareAttribute`, etc.), a default label is generated from this filter list.

```
#! @Label testlabel
DeclareProperty( "AProperty",
                 IsObject );
```

leads to this:

```
<ManSection>
  <Prop Arg="arg" Name="AProperty" Label="testlabel"/>
 <Returns> <K>true</K> or <K>false</K>
</Returns>
 <Description>
 </Description>
</ManSection>
```

while

```
#!
DeclareProperty( "AProperty",
                 IsObject );
```

leads to this:

```
<ManSection>
  <Prop Arg="arg" Name="AProperty" Label="for IsObject"/>
 <Returns> <K>true</K> or <K>false</K>
</Returns>
 <Description>
 </Description>
</ManSection>
```

3.2.7 @ChapterInfo *chapter*, *section*

Adds the entry to the given chapter and section. Here, *chapter* and *section* are the respective titles. As an example, a full AutoDoc comment with all options could look like this:

```
#! @Description
#! Computes the list of lists of degrees of ordinary characters
#! associated to the $p$-blocks of the group $G$
#! with $p$-modular character table <A>modtbl</A>
#! and underlying ordinary character table ‘ordtbl‘.
#! @Returns a list
#! @Arguments modtbl
#! @Group CharacterDegreesOfBlocks
#! @Label chardegblocks
#! @ChapterInfo Blocks, Attributes
DeclareAttribute( "CharacterDegreesOfBlocks",
        IsBrauerTable );
```

3.3 Other documentation comments

There are also some commands which can be used in AutoDoc comments that are not associated to any declaration. This is useful for additional text in your documentation, examples, mathematical chapters, etc.

3.3.1 @BeginGroup *[grpname]*

Starts a group. All following documented declarations without an explicit @Group command are grouped together in the same group with the given name. If no name is given, then a new nameless group is generated. The effect of this command is ended when an @EndGroup command is reached.

See section 3.6 for more information about groups.

3.3.2 @EndGroup

Ends the current group.

```
#! @BeginGroup MyGroup
#!
DeclareAttribute( "GroupedAttribute",
                  IsList );

DeclareOperation( "NonGroupedOperation",
                  [ IsObject ] );

#!
DeclareOperation( "GroupedOperation",
                  [ IsList, IsRubbish ] );
#! @EndGroup
```

3.3.3 @GroupTitle *title*

Sets the subsection heading for the current group to *title*. In the absence of any @GroupTitle command, the heading will be the name of the first entry in the group. See 3.6 for more information.

```
#! @BeginGroup Dimensions
#! @GroupTitle Width and height
```

3.3.4 @BeginExample and @EndExample

@BeginExample marks the start of an example to be put into the manual. It differs from GAPDoc's <Example> (see (**GAPDoc: Log**)), in that it expects actual code (not in a comment) interspersed with comments, to allow for examples files that can be both executed by GAP, and parsed by AutoDoc. To achieve this, GAP commands are not preceded by a comment, while output has to be preceded by an AutoDoc comment. The gap> prompt for the display in the manual is added by AutoDoc. @EndExample ends the example block.

To illustrate this command, consider this input:

```
#! @BeginExample
S5 := SymmetricGroup(5);
#! Sym( [ 1 .. 5 ] )
Order(S5);
#! 120
#! @EndExample
```

This results in the following output:

Example

```
gap> S5 := SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
gap> Order(S5);
120
```

The AutoDoc command @Example is an alias of @BeginExample. If you enable extract_examples := true when calling AutoDoc (4.3.1), these examples can also be turned into .tst files (see Section 2.4.1).

3.3.5 `@BeginExampleSession` and `@EndExampleSession`

`@BeginExampleSession` marks the start of an example to be put into the manual, while `@EndExampleSession` ends the example block. It is the direct analog of GAPDoc's `<Example>` (see (**GAPDoc: Log**)).

To illustrate this command, consider this input:

```
#! @BeginExampleSession
#! gap> S5 := SymmetricGroup(5);
#! Sym( [ 1 .. 5 ] )
#! gap> Order(S5);
#! 120
#! @EndExampleSession
```

This results in the following output:

Example

```
gap> S5 := SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
gap> Order(S5);
120
```

It inserts an example into the manual just as `@Example` would do, but all lines are commented and therefore not executed when the file is read. All lines that should be part of the example displayed in the manual have to start with an AutoDoc comment (`#!`). The comment will be removed, and, if the following character is a space, this space will also be removed. There is never more than one space removed. To ensure examples are correctly colored in the manual, there should be exactly one space between `#!` and the `gap>` prompt. The AutoDoc command `@ExampleSession` is an alias of `@BeginExampleSession`.

3.3.6 `@BeginLog` and `@EndLog`

Works just like the `@BeginExample` command, but the example will not be tested. See the GAPDoc manual for more information. The AutoDoc command `@Log` is an alias of `@BeginLog`.

```
#! @BeginLog
Runtime();
#! 1234
#! @EndLog
```

3.3.7 `@BeginLogSession` and `@EndLogSession`

Works just like the `@BeginExampleSession` command, but the example will not be tested if manual examples are run. It is the direct analog of GAPDoc's `<Log>` (see (**GAPDoc: Log**)). The AutoDoc command `@LogSession` is an alias of `@BeginLogSession`.

```
#! @BeginLogSession
#! gap> Runtime();
#! 1234
#! @EndLogSession
```

3.3.8 @DoNotReadRestOfFile

Prevents the rest of the file from being read by the parser. Useful for unfinished or temporary files.

```
#! This will appear in the manual

#! @DoNotReadRestOfFile

#! This will not appear in the manual.
```

3.3.9 @BeginChunk *name*, @EndChunk, and @InsertChunk *name*

Text inside a @BeginChunk / @EndChunk part will not be inserted into the final documentation directly. Instead, the text is stored in an internal buffer.

That chunk of text can then later on be inserted in any other place by using the @InsertChunk *name* command.

If you do not provide an @EndChunk, the chunk ends at the end of the file.

```
#! @BeginChunk MyChunk
#! Hello, world.
#! @EndChunk

#! @InsertChunk MyChunk
## The text "Hello, world." is inserted right before this.
```

You can use this to define an example like this in one file:

```
#! @BeginChunk Example_Symmetric_Group
#! @BeginExample
S5 := SymmetricGroup(5);
#! Sym( [ 1 .. 5 ] )
Order(S5);
#! 120
#! @EndExample
#! @EndChunk
```

And then later, insert the example in a different file, like this:

```
#! @InsertChunk Example_Symmetric_Group
```

3.3.10 @BeginCode *name*, @EndCode, and @InsertCode *name*

Inserts the text between @BeginCode and @EndCode verbatim at the point where @InsertCode is called. This is useful to insert code excerpts directly into the manual.

```
#! @BeginCode Increment
i := i + 1;
#! @EndCode

#! @InsertCode Increment
## Code is inserted here.
```

3.3.11 `@LatexOnly` *text*, `@BeginLatexOnly`, and `@EndLatexOnly`

Code inserted between `@BeginLatexOnly` and `@EndLatexOnly` or after `@LatexOnly` is only inserted in the PDF version of the manual or worksheet. It can hold arbitrary LaTeX-commands.

```
#! @BeginLatexOnly
#! \include{picture.tex}
#! @EndLatexOnly

#! @LatexOnly \include{picture.tex}
```

3.3.12 `@NotLatex` *text*, `@BeginNotLatex`, and `@EndNotLatex`

Code inserted between `@BeginNotLatex` and `@EndNotLatex` or after `@NotLatex` is inserted in the HTML and text versions of the manual or worksheet, but not in the PDF version.

```
#! @BeginNotLatex
#! For further information see the PDF version of this manual.
#! @EndNotLatex

#! @NotLatex For further information see the PDF version of this manual.
```

3.3.13 `@Index` *key* [*entry text*]

Adds an index entry to the current documentation text. The command `@Index key entry text` generates `<Index Key="key">entry text</Index>`. If no entry text is provided, then the entry text is empty. To use keys containing spaces, wrap the key in double quotes, e.g. `@Index "my key" entry text`. The entry text is processed like normal documentation text, so markdown-like inline code such as `true` is supported.

```
#! @Index "coset enumeration" Coset enumeration, see ‘ToddCoxeter‘
```

generates

```
<Index Key="coset enumeration">Coset enumeration, see <Code>ToddCoxeter</Code></Index>
```

3.4 Title page commands

The following commands can be used to add corresponding parts to the title page of a document generated by AutoDoc.

- `@Title`
- `@Subtitle`
- `@Version`
- `@TitleComment`

- `@Author`
- `@Date`
- `@Address`
- `@Abstract`
- `@Copyright`
- `@Acknowledgements`
- `@Colophon`

Many of these values can (and for package manuals typically should) be extracted from `PackageInfo.g`. If you set them explicitly in comments, they override extracted or scaffold-defined values. This is usually most useful for worksheets created with `AutoDocWorksheet` (4.1.1), since worksheets do not have a `PackageInfo.g` file from which this information could be extracted.

For example, a worksheet could start like this:

```
#! @Title Groups made easy
#! @Subtitle A worked example
#! @Author Jane Doe
#! @Date 2026-11-07
#! @Abstract
#!  This worksheet shows how to set title page data.
```

`@Title`, `@Subtitle`, `@Version`, `@Author`, and `@Date` take only the rest of their line; the others also take all following text lines up to the next command.

3.5 Plain text files

Files that have the suffix `.autodoc` and are listed in the `autodoc.files` option of `AutoDoc` (4.3.1), resp. are contained in one of the directories listed in `autodoc.scan_dirs` or one of their subdirectories, are treated as standalone AutoDoc input files. They are meant for manual text that does not belong next to a single declaration: chapters, sections, tutorials, worked examples, index entries, chunks, title-page metadata, and similar prose-heavy material.

Conceptually, a `.autodoc` file uses the same parser as AutoDoc comments, but without the comment marker. In a `.autodoc` file, lines do not need to start with `#!` and in fact usually should not. This makes `.autodoc` files one convenient way to replace hand-written XML chapters when you prefer AutoDoc's command syntax and Markdown-like text features. However, it is not the only supported style: chapter and section commands inside source comments remain fully supported, and AutoDoc itself still uses that style in files such as `gap/Magic.gd`. For the surrounding workflow, see Chapter 2.

The most important difference from declaration comments is that a plain text file does *not* document a declaration by adjacency. There is no following `Declare...` or `InstallMethod` statement for AutoDoc to inspect. So commands whose meaning depends on such a declaration only make sense in source comments immediately before that declaration.

In practice, this gives the following rule of thumb.

- Use source comments beginning with `#!` to document declarations. This is the mode for `@Description`, `@Returns`, `@Arguments`, `@Label`, `@Group`, and `@ChapterInfo`.
- Use either `.autodoc` files or source comments for standalone manual structure and prose. Commands such as `@Chapter`, `@Section`, `@Subsection`, grouping commands, examples and logs, chunks and code insertion, `@Index`, title-page commands, Markdown-style headings, fenced code blocks, and ordinary GAPDoc XML markup work in both styles.

As a concrete example, the following file can serve as a complete chapter written in `.autodoc` format.

```
@Chapter Test
@Section First Section
@Subsection First Subsection

This text belongs directly to the manual chapter.
It can use XML tags such as <A>arg</A> or <Ref .../>.

@BeginExampleSession
gap> S5 := SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
gap> Size(S5);
120
@EndExampleSession

@Index "Worksheet Autoplain" Plain worksheet index entry with ‘true‘
```

This is essentially the style used in the worksheet fixture `tst/worksheets/autoplain.sheet/plain.autodoc`.

The same structural commands can also be used inside source comments, for example:

```
#! @Chapter Reference
#! @Section The AutoDoc() function
#! Some introductory text for this section.
```

This source-comment style is still fully supported and is used in `gap/Magic.gd`.

The mixed-workflow case is equally common. Suppose an existing manual still has a hand-written main XML file and perhaps some hand-written XML chapters. Then you can keep those files, include `_AutoDocMainFile.xml` from the main XML file as described in Chapter 2, and add one or more `.autodoc` files via `autodoc.files` or `autodoc.scan_dirs`. Those files can provide tutorial chapters or appendices, while declaration documentation continues to live in source comments and older XML chapters remain unchanged.

One caveat is worth keeping in mind. If you want a standalone `.autodoc` file to pause and resume around declaration documentation that is written in source comments, the current workaround is to use chunks. That interleaving workflow is currently limited; see issue #60 in the AutoDoc issue tracker.

So, while `.autodoc` files and source comments share most of the same text syntax, they can be combined freely. The main distinction is simply that declaration-bound commands attach metadata to a following declaration, while standalone manual text can live wherever you find it most convenient.

3.6 Grouping

In GAPDoc, it is possible to make groups of manual items, i.e., when documenting a function, operation, etc., it is possible to group them into suitable chunks. This can be particularly useful if there are several definitions of an operation with several different argument types, all doing more or less the same to the arguments. Then their manual items can be grouped, sharing the same description and return type information. You can give a heading to the group in the manual with the `@GroupTitle` command; if that is not supplied, then the heading of the first manual item in the group will be used as the heading.

Note that group names are globally unique throughout the whole manual. That is, groups with the same name are in fact merged into a single group, even if they were declared in different source files. Thus you can have multiple `@BeginGroup` / `@EndGroup` pairs using the same group name, in different places, and these all will refer to the same group.

Moreover, this means that you can add items to a group via the `@Group` command in the AutoDoc comment of an arbitrary declaration, at any time.

The following code

```
#! @BeginGroup Group1
#! @GroupTitle A family of operations

#! @Description
#!  First sentence.
DeclareOperation( "FirstOperation", [ IsInt ] );

#! @Description
#!  Second sentence.
DeclareOperation( "SecondOperation", [ IsInt, IsGroup ] );

#! @EndGroup

## .. Stuff ..

#! @Description
#!  Third sentence.
#! @Group Group1
KeyDependentOperation( "ThirdOperation", IsGroup, IsInt, "prime );
```

produces the following:

```
<ManSection Label="Group1">
<Heading>A family of operations</Heading>
  <Oper Arg="arg" Name="FirstOperation" Label="for IsInt"/>
  <Oper Arg="arg1,arg2" Name="SecondOperation" Label="for IsInt, IsGroup"/>
  <Oper Arg="arg1,arg2" Name="ThirdOperation" Label="for IsGroup, IsInt"/>
 <Description>
First sentence.
Second sentence.
Third sentence.
 </Description>
</ManSection>
```

3.7 Markdown-like formatting of text in AutoDoc

AutoDoc has some convenient ways to insert special format into text, like math formulas and lists. The syntax for them are inspired by Markdown and LaTeX, but do not follow them strictly. Neither are all features of the Markdown language supported. The following subsections describe what is possible.

3.7.1 Lists

One can create lists of items by beginning a new line with *, +, -, followed by one space. The first item starts the list. When items are longer than one line, the following lines have to be indented by at least two spaces. The list ends when a line which does not start a new item is not indented by two spaces. Of course lists can be nested. Here is an example:

```
#! The list starts in the next line
#! * item 1
#! * item 2
#!   which is a bit longer
#!   * and also contains a nested list
#!   * with two items
#! * item 3 of the outer list
#! This does not belong to the list anymore.
```

This is the output:
The list starts in the next line

- item 1
- item 2 which is a bit longer
  - and also contains a nested list
  - with two items
- item 3 of the outer list

This does not belong to the list anymore.

The *, -, and + are fully interchangeable and can even be used mixed, but this is not recommended.

3.7.2 Math modes

One can start an inline formula with a $, and also end it with $, just like in LaTeX. This will translate into GAPDocs inline math environment. For display mode one can use $$, also like LaTeX.

```
#! This is an inline formula: $1+1 = 2$.
#! This is a display formula:
#! $$ \sum_{i=1}^n i. $$
```

produces the following output:
This is an inline formula: $1+1=2$. This is a display formula:

$$\sum_{i=1}^{n} i.$$

3.7.3 Emphasize

One can emphasize text by using two asterisks (**) or two underscores (__) at the beginning and the end of the text which should be emphasized. Example:

```
#! **This** is very important.
#! This is __also important__.
#! **Naturally, more than one line
#! can be important.**
```

This produces the following output:
This is very important. This is *also important*. *Naturally, more than one line can be important.*

3.7.4 Inline code

One can mark inline code snippets by using backticks (‘) at the beginning and the end of the text which should be marked as code. Example:

```
#! Call function ‘foobar()‘ at the start.
```

This produces the following output:
Call function `foobar()` at the start.

3.7.5 Fenced code blocks

One can insert verbatim code blocks by placing the code between lines containing at least three backticks or at least three tildes. The opening fence may optionally be followed by an info string. The values `@listing`, `@example`, and `@log` select the corresponding GAPDoc element; any other value is currently ignored. If nothing is specified the default is to generate `<Listing>`. Example:

```
#! ‘‘‘@listing
#! if x = 2 then
#!   Print("1 + 1 = 2 holds, all is good\n");
#! fi;
#! ‘‘‘
#! ~~~@example
#! gap> [ 1 .. 3 ] ^ 2;
#! [ 1, 4, 9 ]
#! ~~~
#! ‘‘‘@log
#! #I  some log message
#! ‘‘‘
```

This produces the following output:

```
if x = 2 then
  Print("1 + 1 = 2 holds, all is good\n");
fi;
```

Example

```
gap> [ 1 .. 3 ] ^ 2;
[ 1, 4, 9 ]
```

Example

```
#I  some log message
```

3.8 Deprecated commands

The following commands used to be supported, but are not supported anymore.

`@EndSection`
: You can simply remove any use of this, AutoDoc ends sections automatically at the start of any new section or chapter.

`@EndSubsection`
: You can simply remove any use of this, AutoDoc ends subsections automatically at the start of any new subsection, section or chapter.

`@BeginAutoDoc` **and** `@EndAutoDoc`
: It suffices to prepend each declaration that is meant to appear in the manual with a minimal AutoDoc comment `#!`.

`@BeginSystem` *name*, `@EndSystem`, **and** `@InsertSystem` *name*
: Please use the chunk commands from subsection 3.3.9 instead.

`@AutoDocPlainText` **and** `@EndAutoDocPlainText`
: Use `.autodoc` files or AutoDoc comments instead.

Chapter 4

Reference

4.1 AutoDoc worksheets

4.1.1 AutoDocWorksheet

▷ `AutoDocWorksheet(`*`[filenames][,] [optrec]`*`)` (function)

The purpose of this function is to create stand-alone PDF and HTML files using AutoDoc without associating them with a package.

Instead of a package directory, you pass one filename or a list of filenames containing AutoDoc text from which the document is created. Settings are supplied via an optional record using the same entries as the *optrec* argument of `AutoDoc` (4.3.1). Alternatively, you may omit *filenames* and specify the files via `optrec.autodoc.files`.

A simple worksheet file can define title-page information and chapter content directly in the source file, including example blocks. If this is stored in `worksheet.g`, you can generate documentation via

Example

```
AutoDocWorksheet( "worksheet.g" );
```

This creates documentation in a `doc` subdirectory of the current directory.

Since worksheets do not have a `PackageInfo.g`, title-page fields are specified directly in the worksheet file.

For backwards compatibility, worksheet calls still accept GAP global options for specifying the option-record entries such as `dir`, `scaffold`, `autodoc`, `gapdoc`, and `extract_examples`. However, this feature is deprecated.

4.2 Extracting manual examples

4.2.1 AutoDocExtractExamples

▷ `AutoDocExtractExamples(`*`pkg[, makedoc]`*`)` (function)

Returns: a directory

Extracts the examples from the manual of the package *pkg* and returns the directory holding the resulting `.tst` files.

pkg is either the name of a package or a directory object pointing at one. The optional argument *makedoc* names the script to read, and defaults to `makedoc.g`.

The package's own `makedoc.g` is used as-is, so the settings which describe the manual — its source files, scaffolding and *`extract_examples`* options — are not duplicated. Only the parts of the manual needed to collect the examples are built, no HTML or PDF is produced, and everything is written to a temporary directory, so this works even when the package directory is read-only.

This is meant to be used from a package's `tst/testall.g`, so that extracted tests need not be committed to the repository:

```
LoadPackage( "mypkg" );
dirs := DirectoriesPackageLibrary( "mypkg", "tst" );
Add( dirs, AutoDocExtractExamples( "mypkg" ) );
TestDirectory( dirs, rec( exitGAP := true ) );
```

Note that `makedoc.g` is read in the usual way, so any other work it performs still happens; and a script ending in `QUIT` cannot be used this way.

4.3 The AutoDoc() function

4.3.1 AutoDoc

▷ `AutoDoc(`*`[pkgdir][,] [optrec]`*`)` (function)

Returns: nothing

This is the main function of the AutoDoc package. It can perform any combination of the following tasks:

1. It can (re)generate a scaffold for your package manual. That is, it can produce two XML files in GAPDoc format to be used as part of your manual: First, a file named `doc/_main.xml` which is used as main XML file for the package manual, i.e. this file sets the XML doctype and defines various XML entities, includes other XML files (both those generated by AutoDoc as well as additional files created by other means), tells GAPDoc to generate a table of contents and an index, and more. Secondly, it creates a file `doc/title.xml` containing a title page for your documentation, with information about your package (name, description, version), its authors and more, based on the data in your `PackageInfo.g`.

2. It can scan your package for AutoDoc based documentation, using documentation comments and commands. This produces additional XML files to be used as part of the package manual.

3. It can use GAPDoc to generate PDF, text and HTML (with MathJax enabled) documentation from the GAPDoc XML files it generated as well as additional such files provided by you. For this, it invokes `MakeGAPDocDoc` (**GAPDoc: MakeGAPDocDoc**) to convert the XML sources, and it also instructs GAPDoc to copy supplementary files (such as CSS style files) into your doc directory (see `CopyHTMLStyleFiles` (**GAPDoc: CopyHTMLStyleFiles**)).

These tasks can be enabled independently, so you can use as much or as little of AutoDoc as your package currently needs. For more information and some examples, please refer to Chapter 2.

If exactly one parameter is given, it is interpreted as `optrec` if it is a record, and as `pkgdir` otherwise.

The parameters have the following meaning:

pkgdir

This optional parameter is used to determine the package directory in which AutoDoc will operate, and to find the metadata concerning the package being documented. If given, it should be an `IsDirectory` object. If the argument is omitted, then AutoDoc checks if it was called from a `makedoc.g` file or similar, and if so, uses the directory this is contained in. Otherwise the current directory is used. In both cases, the specified directory must contain a `PackageInfo.g` file, and AutoDoc extracts all needed metadata from that. The `IsDirectory` form is for example useful if you have multiple versions of the package around and want to make sure the documentation of the correct version is built.

For backwards compatibility, it is also possible to pass a package name as this argument, which then is resolved to the package directory of the first instance of this package GAP knows about. However, this is deprecated, as it is unreliable in the presence of multiple copies of a package.

Note that when using `AutoDocWorksheet` (see 4.1), there is no parameter corresponding to *pkgdir* and so the "package directory" is always the current directory, and there is no metadata.

optrec

optrec can be a record with some additional options. The following are currently supported:

dir This should either be a string, in which case it must be a path *relative* to the specified package directory, or an `IsDirectory` object. (Thus, the only way to designate an absolute directory is with an `IsDirectory` object.) This option specifies where the package documentation (e.g. the GAPDoc XML or the manual PDF, etc.) files are stored and/or will be generated.
Default value: `"doc/"`.

scaffold
Controls whether and how to generate scaffold XML files for the package documentation; see 4.3.3.

autodoc
Controls whether and how to generate additional XML documentation files by scanning for AutoDoc documentation comments; see 4.3.5.

gapdoc
Controls whether and how to invoke GAPDoc to create HTML, PDF and text files from your various XML files; see 4.3.6.

extract_examples
Controls whether and how to extract the manual examples into test files; see 4.3.7.

The function also checks some GAP global options, which are not entries of *optrec*; see 4.3.8.

4.3.2 InfoAutoDoc

▷ `InfoAutoDoc` (info class)

Info class for the AutoDoc package. Set this to 0 to suppress info messages, 1 to allow most messages, and 2 to allow all messages including those that contain file paths.

This can be set by calling, for example, `SetInfoLevel(InfoAutoDoc, 0)`. Default value is 1.

4.3.3 Scaffold options

The entry *optrec.scaffold* of AutoDoc (4.3.1) controls whether and how to generate scaffold XML files for the package documentation.

The value should be either true, false or a record. If it is a record or true (the latter is equivalent to specifying an empty record), then this feature is enabled. It is also enabled if *optrec.scaffold* is missing but the package's info record in PackageInfo.g has an AutoDoc entry. In all other cases (in particular if *optrec.scaffold* is false), scaffolding is disabled.

If scaffolding is enabled, and PackageInfo.AutoDoc exists, then it is assumed to be a record, and its contents are used as default values for the scaffold settings.

If *optrec.scaffold* is a record, it may contain the following entries.

includes
: A list of XML files to be included in the body of the main XML file. They are included in the order in which they are given. If you specify this list and also are using AutoDoc to document your operations with AutoDoc comments, you can add _AutoDocMainFile.xml to this list to control at which point the documentation produced by AutoDoc is inserted. If you do not do this, it will be added after the last of your own XML files.

index
: By default, the scaffold creates an index. If you do not want an index, set this to false.

appendix
: This entry is similar to *optrec.scaffold.includes* but is used to specify files to include after the main body of the manual, i.e. typically appendices written directly in GAPDoc XML. Appendices created with @Appendix are included automatically after these files when scaffolding generates the main XML file.

bib
: The name of a bibliography file, in BibTeX or XML format. If this key is not set, but there is a file doc/PACKAGENAME.bib then it is assumed that you want to use this as your bibliography.

bibstyle
: Overrides the bibliography style used for LaTeX output. This is written as the Style attribute of the generated <Bibliography .../> element, so valid values are the bibliography style names understood by GAPDoc and BibTeX, such as alpha, alphaurl, or plain.

entities
: A record defining additional XML entities; see 4.3.4.

TitlePage
: A record with extra title-page fields for the generated manual. Field names are interpreted as title-page XML element names, and their values are written as element content. For example, you can set a custom acknowledgements block with

```
rec( Acknowledgements := "Many thanks to ..." )
```

If this is set in PackageInfo.g as PkgInfo.AutoDoc.TitlePage, it has the same meaning as this option; see subsection 2.5.3 in chapter 2 for details and an example.

For the list of valid title-page elements, see the GAPDoc manual, specifically section (**GAPDoc: TitlePage**).

MainPage
: If scaffolding is enabled, by default a main XML file is generated (this is the file which contains the XML doctype and more). If you do not want this (e.g. because you have a handwritten main XML file), but still want AutoDoc to generate a title page for you, you can set this option to `false`

document_class
: Sets the document class of the resulting PDF. The value can either be a string which has to be the name of the new document class, a list containing this string, or a list of two strings. Then the first one has to be the document class name, the second one the option string (contained in [ ]) in LaTeX.

latex_header_file
: Replaces the standard header from GAPDoc completely with the header in this LaTeX file. Please be careful here, and look at GAPDoc's `latexheader.tex` file for an example.

4.3.4 Scaffold entities

The entry *optrec.scaffold.entities* of `AutoDoc` (4.3.1) is a record whose keys are taken as entity names, set to the corresponding (string) values. For example, if you pass the record

```
rec( SomePackage := "<Package>SomePackage</Package>",
     RELEASEYEAR := "2015" )
```

then the following entity definitions are added to the XML preamble:

```
<!ENTITY SomePackage '<Package>SomePackage</Package>'>
 <!ENTITY RELEASEYEAR '2015'>
```

This allows you to write "&SomePackage;" and "&RELEASEYEAR;" in your documentation, which will be replaced by respective values specified in the entities definition.

Note that AutoDoc predefines several entities:

VERSION
: Set to the `Version` field of your package info record.

RELEASEYEAR
: Set to a string containing the release year, as derived from the `Date` field of your package info record.

RELEASEDATE
: Derived from the `Date` field of your package info record.

SomePackage
: The precise name of this entity is derived from the `PackageName` field of your package info record. Note that it is case sensitive. The content is defined as suggested by the example above.

4.3.5 Options for scanning AutoDoc comments

The entry *optrec.autodoc* of AutoDoc (4.3.1) controls whether and how to generate additional XML documentation files by scanning for AutoDoc documentation comments.

The value should be either true, false or a record. If it is a record or true (the latter is equivalent to specifying an empty record), then this feature is enabled. It is also enabled if *optrec.autodoc* is missing but the package depends (directly) on the AutoDoc package. In all other cases (in particular if *optrec.autodoc* is false), this feature is disabled.

If *optrec.autodoc* is a record, it may contain the following entries.

files
: A list of files (given by paths relative to the package directory) to be scanned for AutoDoc documentation comments. They are scanned in the order in which they are given, and always before the files scanned by *autodoc.scan_dirs*. Unless the order of files is important, it is usually more convenient to use *autodoc.scan_dirs*, see below.

scan_dirs
: A list of subdirectories of the package directory (given as relative paths) which AutoDoc then scans recursively for .gi, .gd, .g, and .autodoc files; all of these files are then scanned for AutoDoc documentation comments. The directories are scanned in the order in which they are given, and the files in each directory are scanned in lexicographic order. The special entries "." and "" still only scan the package root itself. This controls where AutoDoc looks for source comments beginning with #! and for standalone .autodoc files. It does not affect where GAPDoc looks for GAPDoc comments; that is controlled separately by *gapdoc.scan_dirs*.
Default value: [".", "gap", "lib", "examples", "examples/doc"].

4.3.6 Options for GAPDoc

The entry *optrec.gapdoc* of AutoDoc (4.3.1) controls whether and how to invoke GAPDoc to create HTML, PDF and text files from your various XML files.

The value should be either true, false or a record. If it is a record or true (the latter is equivalent to specifying an empty record), then this feature is enabled. It is also enabled if *optrec.gapdoc* is missing. In all other cases (in particular if *optrec.gapdoc* is false), this feature is disabled.

If *optrec.gapdoc* is a record, it may contain the following entries.

main
: The name of the main XML file of the package manual. This exists primarily to support packages with existing manual which use a filename here which differs from the default. In particular, specifying this is unnecessary when using scaffolding.
Default value: _main.xml when scaffolding is enabled for package manuals, otherwise PACKAGENAME.xml.

files
: A list of files (given by paths relative to the package directory) to be scanned for GAPDoc documentation comments. Usually it is more convenient to use *gapdoc.scan_dirs*, see below.

scan_dirs
: A list of subdirectories of the package directory (given as relative paths) which AutoDoc then

scans recursively for .gi, .gd and .g files; all of these files are then scanned for GAPDoc documentation comments. The special entries "." and "" still only scan the package root itself. This controls only where GAPDoc comments are searched for. It does not affect where AutoDoc looks for source comments beginning with #! or for .autodoc files; use *autodoc.scan_dirs* for that.
Default value: [".", "gap", "lib", "examples", "examples/doc"].

LaTeXOptions
Must be a record with entries which can be understood by SetGapDocLaTeXOptions (**GAPDoc: SetGapDocLaTeXOptions**). Each entry can be a string, which will be given to GAPDoc directly, or a list containing of two entries: The first one must be the string "file", the second one a filename. This file will be read and then its content is passed to GAPDoc as option with the name of the entry.

gap_root_relative_path
Either a boolean, or a string containing a relative path. By default (if this option is not set, or is set to false), references in the generated documentation referring to external documentation (such as the GAP manual) are encoded using absolute paths. This is fine as long as the documentation stays on only a single computer, but is problematic when preparing documentation that should be used on multiple computers, e.g., when creating a distribution archive of a GAP package.
Thus, if a relative path is provided via this option (or if it is set to true, in which case the relative path ../../.. is used), then AutoDoc and GAPDoc attempt to replace all absolute paths in references to GAP manuals by paths based on this relative path.

On a technical level, AutoDoc passes the relative path to the *gaproot* parameter of MakeGAPDocDoc (**GAPDoc: MakeGAPDocDoc**)

4.3.7 Options for extracting examples

The entry *optrec.extract_examples* of AutoDoc (4.3.1) must be either true or a record. If set to true, then all manual examples are extracted and placed into files tst/PACKAGENAME01.tst, tst/PACKAGENAME02.tst, ... and so on, with one file for each chapter. For chapters with no examples, no file is created.

If set to a record, it may contain the following entries:

subdir
A string or Directory() object selecting where the generated .tst files are written. The default is tst. If a string is given, it is interpreted relative to the package directory, so values such as tst/generated are supported.

units
This controls how examples are grouped into files. Recognized values are "Chapter" (default), "Section", "Subsection" or "Single". Depending on the value, one file is created for each chapter, each section, each subsection or each example. For all other values only a single file is created.

On a technical level, AutoDoc passes the value to the *units* parameter of ExtractExamples (**GAPDoc: ExtractExamples**).

skip_empty_in_numbering
: If set to `true` (the default), the generated files use filenames with strictly sequential numbering; that means that if chapter 1 (or whatever units are being used) contains no examples but chapter 2 does, then the examples for chapter 2 are put into the file `tst/PACKAGENAME01.tst`. If this option is set to `false`, then the chapter numbers are used to generate the filenames; so the examples for chapter 2 would be put into the file `tst/PACKAGENAME02.tst`.

4.3.8 Global options

`AutoDoc` (4.3.1) also checks the following GAP global options, i.e. options supplied via GAP's value-option syntax and visible through nested calls. These are not entries of *optrec*. See (**Reference: Options Stack**) for more information about GAP's global options system.

nopdf
: If this global option is set to `true`, then AutoDoc tells GAPDoc not to build the PDF version of the manual. HTML and text output are still generated.

  This is useful on systems without a working `pdflatex` installation, or when you only need the non-PDF outputs while iterating on the manual.

  For example:

```
AutoDoc( rec( autodoc := true ) : nopdf );
```

  Also, if the environment variable `NOPDF` is set, then AutoDoc behaves as if the global option *nopdf* had been enabled.

AutoDocExtractOnly
: If this global option is set, AutoDoc builds only as much of the manual as is needed to collect its examples: no HTML, PDF or manual index is produced, and all output is written below the given scratch directory instead of the package. The extracted tests end up in its `tst` subdirectory.

  The value is a directory, or `true` to have one chosen automatically. If the environment variable `AUTODOC_EXTRACT_ONLY` is set, AutoDoc behaves as if this option had been given; that is useful for a `makedoc.g` which ends in `QUIT` and therefore has to be run as a separate process.

  Rather than setting this by hand, use `AutoDocExtractExamples` (4.2.1).

relativePath
: This has the same effect as *gapdoc.gap_root_relative_path*, but as a GAP global option. It takes precedence over that record entry if both are specified.

  If *relativePath* is `true`, then the default relative path `../../..` is used. If it is a string, then that string is used as the relative path from the documentation directory to the GAP root.

  For example:

```
AutoDoc( rec( autodoc := true ) : relativePath := "../../.." );
```

In particular, a call such as

```
Read( "makedoc.g" : nopdf, relativePath );
```

sets both global options to `true`, and they remain visible to the `AutoDoc` (4.3.1) call inside `makedoc.g`.

References

[LN24] F. Lübeck and M. Neunhöffer. *GAPDoc (Version 1.6.7)*. RWTH Aachen, 2024. GAP package, https://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/index.html. 2

Index

@Appendix, 18
@Arguments *args*, 20
@BeginChunk *name*, 25
@BeginCode *name*, 25
@BeginExample, 23
@BeginExampleSession, 24
@BeginGroup, 22
@BeginLatexOnly, 26
@BeginLog, 24
@BeginLogSession, 24
@BeginNotLatex, 26
@Chapter, 17
@ChapterInfo, 22
@ChapterLabel, 17
@ChapterTitle, 17
@Description *descr*, 20
@DoNotReadRestOfFile, 25
@EndChunk, 25
@InsertChunk *name*, 25
@EndCode, 25
@EndExample, 23
@EndExampleSession, 24
@EndGroup, 23
@EndLatexOnly, 26
@EndLog, 24
@EndLogSession, 24
@EndNotLatex, 26
@Group *grpname*, 21
@GroupTitle, 23
@Index *key* [*entry text*], 26
@InsertCode *name*, 25
@ItemType *kind*, 20
@Label *label*, 21
@LatexOnly *text*, 26
@NotLatex *text*, 26
@Returns *ret_val*, 20
@Section, 18
@SectionLabel, 18
@SectionTitle, 18
@Subsection, 18
@SubsectionLabel, 18
@SubsectionTitle, 18

Abstract field in PackageInfo.g, 12
Acknowledgements field in PackageInfo.g, 12
AutoDoc, 34
AutoDocExtractExamples, 33
AutoDocWorksheet, 33

Bibliography field in makedoc.g, 12

Copyright field in PackageInfo.g, 12

Entities record in makedoc.g, 12

InfoAutoDoc, 35

LaTeXOptions record in makedoc.g, 12

makedoc.g, 7

Scaffold record in makedoc.g, 12