Version 0.1

PDFTOOLBOX

ari.feiglin@gmail.com

PDFTOOLBOX offers a variety of tools for creating documents in plain TEX. These include packages for structuring documents, coloring documents, etc. PDFTOOLBOX is a collection of packages intended to be used with plain TEX. It is intended to be self-contained and does not promise compatibility with other packages.

PDFTOOLBOX is still experimental and may be subject to breaking changes. If you have an important document relying on it, the author advises keeping

PDFTOOLBOX is known to not interact with the `color`, `xcolor`, `tikz` and all related packages. This may or may not be changed in the future.

This documentation is split into sections corresponding to the different collections in PDFTOOLBOX. These are:

(1) Data manipulation: counters, dictionaries, etc.

(2) Document structure: layouts, table of contents, indices, etc.

(3) Graphics: colors, diagrams, colored boxes, etc.

Contents

1 pdfData

The pdfData section of the PDFTOOLBOX toolbox is meant for creating instances of and manipulating datatypes.

1.1 Arrays

In the `pdfData/arrays` file, PDFTOOLBOX defines various macros for creating and manipulating arrays. PDFTOOLBOX defines two types of arrays, which are different in the macros used for them and the way they are stored internally.

- (Normal) arrays: these arrays are stored in the traditional way: an array $[1;2;3]$ is stored in a macro whose meaning is equivalent to `\X{1}\X{2}\X{3}`. Manipulation of the array is done by defining `\X`, and then executing the array macro.
- Macro arrays: these arrays are stored in a collection of macros: each element is stored in its own indexed macro. So an array $[1;2;3]$ will be stored in three macros, whose values are 1, 2, 3 respectively.

All arrays are zero-indexed.

1.1.1 Normal Arrays

`\createarray {`⟨*name*⟩`}`: creates an (normal) array whose name is *name*.

`\ensurearray {`⟨*name*⟩`}`: ensures that an array by the name of *name* exists.

`\localizearray {`⟨*name*⟩`}`: localizes (see localization) the array named by *name*.

`\appendarray {`⟨*name*⟩`}{`⟨*value*⟩`}`: appends *value* to the end of the array array named by *name*. *value* is inserted according to `\currdef`.

`\prependarray {`⟨*name*⟩`}{`⟨*value*⟩`}`: prepends *value* to the end of the array array named by *name*. *value* is inserted according to `\currdef`.

`\appendarraymany {`⟨*name*⟩`}{{`⟨*value1*⟩`}{`⟨*value2*⟩`}...{`⟨*valueN*⟩`}}`: appends *value1* through *valueN* to the end of the array array named by *name*. Each *value* is inserted according to `\currdef`.

`\arraylen {`⟨*name*⟩`}`: expands to the length of the array specified by *name*.

`\getarraylen {`⟨*name*⟩`}`⟨*macro*⟩: inserts the length of the array specified by *name* into the macro *macro*.

`\arraymap {`⟨*name*⟩`}`⟨*macro*⟩: if the array specified by *name* is equivalent to `[x0;...;xN]` then doing `\arraymap{name}\X` will execute `\X{x1}{0}...\X{xN}{N}`.

`\indexarray {`⟨*name*⟩`}{`⟨*i*⟩`}`⟨*macro*⟩: Puts the ith element in the array specified by *name* into the macro *macro*.

`\removearray {`⟨*name*⟩`}{`⟨*i*⟩`}`⟨*macro*⟩: Removes the ith element in the array specified by *name* and places it into the macro *macro*.

`\removeitemarray {`⟨*name*⟩`}{`⟨*value*⟩`}`: Removes all instances of *value* from the array specifed by *name* (comparison is done using `\ifx` on macros containing *value* and the current index).

`\printarray {`⟨*name*⟩`}`: Prints the array specified by *name*.

`\copyarray {`⟨*src*⟩`}{`⟨*dest*⟩`}`: Copies the array *src* into *dest*.

`\concatenatearrays {`⟨*arr1*⟩`}{`⟨*arr2*⟩`}{`⟨*dest*⟩`}`: Concatenates the arrays *arr1* and *arr2* and places the result into a new array *dest*.

`\initarray {`⟨*name*⟩`}{`⟨*x1*⟩`,...,`⟨*xN*⟩`}`: Creates a new array by the name of *name* equivalent to `[x1;...;xN]`.

`\findarray {`⟨*name*⟩`}{`⟨*value*⟩`}`: Checks if the value *value* exists in the array *name* (checking is done via `\ifx`). If the value exists, the value `\True` is placed into `\@return@value`, otherwise it is equal to `\False`.

`\uniqueappendarray {`⟨*name*⟩`}{`⟨*value*⟩`}`: Appends *value* to the array *name* only if it does not already exist in *name* (`\@return@value` is set accordingly).

`\convertarray {`⟨*src*⟩`}{`⟨*dest*⟩`}`: Converts a normal array *src* to a macro array *dest*.

`\mergesort {⟨src⟩}{⟨dest⟩}`: Sorts the array *src* and places the result in *dest*.

1.1.2 Macro Arrays

`\createmarray {⟨name⟩}`: Creates a macro array by the name of *name*.

`\localizemarray {⟨name⟩}`: Localizes (see localization) the macro array specified by *name*.

`\appendmarray {⟨name⟩}{⟨value⟩}`: Appends *value* to the macro array specified by *name*.

`\printmarray {⟨name⟩}`: Prints the macro array specifed by *name*.

`\convertmarray {⟨src⟩}{⟨dest⟩}`: Converts the macro array *src* into a normal array *dest*.

`\copymarray {⟨src⟩}{⟨dest⟩}`: Copies the macro array *src* into *dest*.

`\initmarray {⟨name⟩}{⟨x1⟩,...,⟨xN⟩}`: Creates a macro array *name* whose value is equivalent to [`x1`, ..., `xN`].

`\findmarray {⟨name⟩}{⟨value⟩}⟨macro⟩`: Searches for *value* in the macro array *name*. If found, sets `\@return@value` to `\True` and *macro* to the index where *value* was found. Otherwise `\@return@value` is set to `\False`.

1.2 Stacks

In the `pdfData/stacks.tex` file, PDFTOOLBOX offers macros for creating and manipulating stack data-structures. PDFTOOLBOX offers two types of stacks, which differ in how they store their data. They are generally used for different purposes:

- Normal stacks: these are normal stacks which store just the values given.
- Macro stacks: these stacks are meant to store only macros: they store both the definition and name of the macro.

1.2.1 Normal Stacks

`\createstack {⟨name⟩}`: Creates a normal stack by the name of *name*.

`\stackpush {⟨name⟩}{⟨value⟩}`: Pushes the value *value* onto the stack specified by *name*.

`\stackdecrement {⟨name⟩}`: Pops from the top of the stack specified by *name* (deleting the value).

`\stackpop {⟨name⟩}⟨macro⟩`: Pops from the top of the stack specified by *name* into *macro*.

`\stacktop {⟨name⟩}⟨macro⟩`: Places the top of the stack specified by *name* into the macro *macro* without popping.

1.2.2 Macro Stacks

Macro stacks store macros, as opposed to values. When pushing a macro `\X` onto the stack, not only is the meaning of `\X` stored, but so is its name.

`\createmacrostack {⟨name⟩}`: Creates a macro stack by the name of *name*.

`\macrostackpush {⟨name⟩}⟨macro⟩`: Pushes the macro *macro* onto the macro stack specified by *name*.

`\macrostackdecrement {⟨name⟩}`: Pops from the top of the macro stack specified by *name* (deleting the value).

`\macrostackset {⟨name⟩}`: If the top of the macro stack specified by *name* has name `\X` and value `value`, sets `\X` to `value`.

`\macrostackpop {⟨name⟩}`: Pops from the top of the macro stack specified by *name* (same as `\macrostackset`, but also pops the value off of the stack).

`\macrostacktop {⟨name⟩}⟨macro1⟩⟨macro2⟩`: If the top of the macro stack specified by *name* is (`\X`, `value`), then `\X` is placed into *macro1*, and `value` into *macro2*.

1.3 Localization

Using macro stacks, PDFTOOLBOX allows for *localization*. This gives the user the ability to create block scopes (as opposed to just plain-ol' TEX groups). The usage is simple and as follows:

(1) The user enters a scope using `\beginscope`.

(2) The user *localizes* a macro `\X` by doing `\localize\X`.

(3) The user exits the scope using `\endscope`. Once the scope is exited, the previous definition of localized macros is restored.

So for example,

```
\def\X{0}
\beginscope
  \localize\X
  \def\X{1}
  \X
  \beginscope
    \def\X{2}
    \X
  \endscope
  \X
\endscope
\X
```

Will output 1 2 2 0. As opposed to

```
\def\X{0}
\bgroup
  \def\X{1}
  \X
  \bgroup
    \def\X{2}
    \X
  \egroup
  \X
\egroup
\X
```

Which will output 1 2 1 0.

1.4 Counters

In the `pdfData/counters.tex`, PDFTOOLBOX implements counters. Counters are simple wrappers over plain-TEX counters. They hold integer values, are mutable, and can be made dependent on one another so that when one is altered another is set to zero.

`\createcounter {`⟨*name*⟩`}[`⟨*c1*⟩`,...,`⟨*cN*⟩`]`: Creates a counter by the name *name* dependent on counters *c1,...,cN*.

`\adddependentcounter {`⟨*secondary*⟩`}{`⟨*primary*⟩`}`: Makes the *secondary* counter dependent on the *primary* one; whenever *primary* is (non-independently; see e.g. `\seticounter`) altered, *secondary* is set to zero.

`\zerodependents {`⟨*primary*⟩`}`: Sets to zero all counters dependent on *primary*.

`\setcounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Sets *counter* to *amount* (zeroing all counters dependent on *counter*).

`\advancecounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Advances *counter* by *amount* (zeroing all counters dependent on *counter*).

`\seticounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Sets *counter* to *amount* (*without* zeroing all counters dependent on *counter*).

`\advanceicounter {`⟨*counter*⟩`}{`⟨*amount*⟩`}`: Advances *counter* by *amount* (*without* zeroing all counters dependent on *counter*).

`\counter {`⟨*name*⟩`}`: Returns the TEX counter corresponding to the PDFTOOLBOX counter *name*. Useful for example when printing the value of a counter: simply do `\the\counter{name}`.

1.5 Dictionaries

In the `pdfData/dictionaries.tex` file, PDFTOOLBOX implements dictionaries (also colloquially known as "hashmaps" or "maps"). These are simple maps between keys and values.

`\createdict {`⟨*name*⟩`}`: Creates a dictionary by the name *name*.

`\adddict {`⟨*name*⟩`}{`⟨*key*⟩`}{`⟨*value*⟩`}`: Adds the (*key* : *value*) key-value pair to the dictionary specified by *name*.

`\indexdict {`⟨*name*⟩`}{`⟨*key*⟩`}`: Expands to the value of *key* in the dictionary *name*.

`\keyindict {`⟨*name*⟩`}{`⟨*key*⟩`}`: Sets `\@return@value` according to if *key* is found in the dictionary *name*.

1.6 Key-Value Parameters

In `pdfData/key-value.tex`, PDFTOOLBOX implements the ability to pass key-value parameters to macros.

`\mapkeys {`⟨*options*⟩`}{`⟨*input*⟩`}`: Maps the key-value pairs given in *input* according to *options*. *options* is itself a set of key-value pairs, where the value of each key is an array which may contain:

- `name` (required): the name of the macro to give the value of the key;
- `required`: added if the key is required;
- `definition`: what definition macro to use for defining the value (e.g. `\def`, `\edef`);
- `mapping`: how to map the input to the value: the input is defined relative to `definition` into a macro wrapped with `mapping`;
- `default`: the default value of the key.

Or the value may be empty (no array), which means it is *valueless* and acts as a boolean flag.

So for example, you may have a macro defined like so:

```
\def\puthi#1{Hello (#1)}

\def\getinput#1{%
   \mapkeys{
      first={
         name=fst,
         required,
         definition=\edef,
         mapping=\puthi%
      },
      second={
         name=snd,
         default=S. Lurp%
      }%
   }{#1}%
}

\getinput{first=pdftoolbox}
(\fst) (\snd)
```

This will output `(Hello (pdftoolbox)) (S. Lurp)`.

`\keyexists {`⟨*key*⟩`}`⟨*macro*⟩`\lastkeys`: This is an internal command, added to this documentation only due to its usefulness. Given a key name *key*, this macro checks if it exists in the map corresponding to the last call to `\mapkeys` (the macro itself is more versatile, but we restrict it to this case). If the key does not exist, then *macro* is set to `\_nul`. This is useful with valueless keys.

`\mapkeys` is a bit finnicky when it comes to spaces and commas, but the rule is simple: place a comment at the end of each list. That means that within each key's array, you must place a comment at the end (otherwise an extraneous space is added to the value), and after the last key's array you must place a comment.

2 pdfDstruct

3 pdfDstruct

The pdfDstruct section of the PDFTOOLBOX toolbox is for managing the structure of your documents.

3.1 Layout

In `pdfDstruct/layout.tex`, PDFTOOLBOX provides a macro `\setlayout` for setting up the layout of the document. The use is

`\setlayout {[page width=`⟨*wd*⟩`,] [page height=`⟨*ht*⟩`,] [horizontal margin=`⟨*mwd*⟩`,] [vertical margin=`⟨*vwd*⟩`]}`

3.2 Hyperlinks

In `pdfDstruct/hyperlinks.tex`, PDFTOOLBOX provides macros for creating and managing hyperlinks.

`\anchor [`⟨*type*⟩`]{`⟨*name*⟩`}`: Creates an anchor (a reference, if you will) to the current point in the document.

`\gotoanchor [`⟨*type*⟩`]{`⟨*name*⟩`}{`⟨*material*⟩`}`: Creates a clickable field containing *material* which, when clicked, will go to the anchor labeled with the type *type* and name *name*.

`\url {`⟨*url*⟩`}{`⟨*material*⟩`}`: Creates a clickable field containing *material* which, when clicked, will redirect to the url *url*.

`\createbordertype {`⟨*type*⟩`}{`⟨*color*⟩`}{`⟨*wd*⟩`}`: Sets the border type of anchor type *type* to be of color *color* and width *wd*. Urls have border type `url`. If a type doesn't have a specified border type, the `default` one is used.

3.3 Fonts

In `pdfDstruct/fonts.tex`, PDFTOOLBOX provides macros for accessing and controlling fonts.

`\addfont {`⟨*name*⟩`}{`⟨*sizes*⟩`}`: This will add a font by the name *name* so that it is accessible by PDFTOOLBOX. *sizes* is a key-value dictionary which specifies the font codes for different sizes of the font. For example, in `pdfDstruct/fonts.tex` is the usage:

```
\addfont{rm}{%
  default=cmr10,
  5pt=cmr5,
  6pt=cmr6,
  7pt=cmr7,
  8pt=cmr8,
  9pt=cmr9,
  10pt=cmr10,
  12pt=cmr12,
  17pt=cmr17
}
```

So now PDFTOOLBOX has access to the computer modern roman font (`cmr`) at the sizes specified. The purpose of the default size is for when a size is not available. For example, requesting the `rm` font at size 13 will give you `cmr10 at 13pt`. The default size is required.

PDFTOOLBOX provides the following fonts:

rm: `cmr` it: `cmti` bf: `cmbx` sc: `cmcsc` mi: `cmmi` sy: `cmsy` ex: `cmex` sl: `cmsl`
ss: `cmss` tt: `cmtt` msam: `msam` msbm: `msbm` eufm: `eufm` rsfs: `rsfs`

`\applyfontcode` ⟨*font code*⟩: Applies the font specified by *font code*. For example, `\applyfontcode cmr10` will set the font to `cmr10`.

`\setfontfamily {`⟨*font*⟩`}{`⟨*family*⟩`}`: Sets math font family *family* to the font *font* (which is specified by `\addfont`). For example, `\setfontfamily{rm}{0}` sets the alpha-numeric font family to `rm`.

`\setfont {`⟨*font*⟩`}`: Sets the current font to *font*. The current font is stored in the macro `\currfont`.

`\setscale {`⟨*scale*⟩`}`: Sets the current font scale to *scale*. The current font scale is stored in the macro `\currscale`.

`\setfontandscale {`⟨*font*⟩`}{`⟨*scale*⟩`}`: Sets the current font to *font* and scale to *scale*.

PDFTOOLBOX also provides the following font switches (which are simple wrappers around `\setfont` which also set `\fam`):

`\bf, \it, \bb, \sf, \sl, \frak, \scr`

`\mathfonttable {`⟨*family*⟩`}[`⟨*offset*⟩`]{`⟨*table*⟩`}`: The `\mathfonttable` macro's purpose is to define multiple mathematical characters for the same family. *table* consists of a sequence of macros followed by numbers (e.g. `\square0`) which correspond to the name of the macro and the math type (in this case 0: ordinary/`\mathord`). `\mathfonttable` will iterate over *table* and `\mathchardef` the macro to be equal to the character at the current position in family *family* of the type specified. If *offset* is specified, it will start iterating over the family starting from the offset.

More explicitly, if *family* is `X` and the `i`th index in the table is `\X N`, then the macro does essentially

```
\mathchardef\X = XNi
```

To skip over an index, simply write `\__`.

Using `\mathfonttable`, PDFTOOLBOX defines the following:

\boxdot: ⊡	\boxplus: ⊞	\boxtimes: ⊠	\square: □
\blacksquare: ■	\diamond: ◇	\blackdiamond: ⧫	\rotateclockwise: ↻
\rotatecounterclockwise: ↺	\rightleftharpoons: ⇌	\leftrightharpoons: ⇋	\boxminus: ⊟
\Vdash: ⊩	\Vvdash: ⊪	\vDash: ⊨	\twoheadrightarrow: ↠
\twoheadleftarrow: ↞	\leftleftarrows: ⇇	\rightrightarrows: ⇉	\upuparrows: ⇈
\downdownarrows: ⇊	\upharpoonright: ↾	\downharpoonright: ⇂	\upleftharppon: ↿
\downleftharpoon: ⇃	\rightarrowtail: ↣	\leftarrowtail: ↢	\leftrightarrows: ⇆
\rightleftarrows: ⇄	\Lsh: ↰	\Rsh: ↱	\rightsquigarrow: ⇝
\leftrightsquigarrow: ↭	\looparrowleft: ↫	\looparrowright: ↬	\circeq: ≗
\succsim: ≿	\gtrsim: ≳	\gtrapprox: ⪆	\multimap: ⊸
\therefore: ∴	\because: ∵	\Doteq: ≑	\triangleq: ≜
\precsim: ≾	\lesssim: ≲	\lessapprox: ⪅	

3.4 Hooks

PDFTOOLBOX provides a tool, inspired by LaTeX, called *hooks* (source in `pdfDstruct/hooks.tex`). Hooks are simply snippets of code that can be inserted into macros and then altered later. An example is given at the end of this section.

`\createhook {`⟨*name*⟩`}`: Creates a hook by the name of *name*.

`\appendtohook {`⟨*name*⟩`}{`⟨*code*⟩`}`: Appends *code* to the hook specified by *name*.

`\prependtohook {`⟨*name*⟩`}{`⟨*code*⟩`}`: Prepends *code* to the hook specified by *name*.

`\callhook {`⟨*name*⟩`}`: Calls the hook specified by *name*.

PDFTOOLBOX provides a builtin hook called `end` which is executed by `\bye`. Throughout the document, you can add macros to an array called `document data`, then all these definitions are written to the file `\jobname.data` by the `end` hook.

Specifically, you can use the `\docdata` macro to add a macro to the document's data, e.g. if you have a macro `\name` which has the author's name (say, S. Lurp), you can do `\docdata\name`, and this will write the line `\gdef\name{S. Lurp}` to the data file. Then at the beginning of the document next compilation, you can load all definitions in the data file.

3.5 Indices

In `pdfDstruct/index.tex`, PDFTOOLBOX provides macros for creating an index. The index is organized into *categories* and *items* within each category, and an associated *value*. A category may be something like "manifolds" and an item within this category may be "topological" which has a value corresponding to the page number where topological manifolds are defined.

`\indexize {`⟨*options*⟩`}`: Adds an item to the index, specified by options, which has fields:

(1) `category` (required): the category of the item;

(2) `item`: the item of the item;

(3) `value` (required): the value of the item;

(4) `expand value` (valueless): added if `value` should be expanded (e.g. if `value` is a macro corresponding to the page number, it needs to be expanded);

(5) `add hyperlink` (valueless): whether or not the item's values should be hyperlinked.

`\seealso {`⟨*options*⟩`}`: Adds a "see also" item to the index: one which redirects to another index item. *options* is a map which has fields:

(1) `category` (required): the category of the item;

(2) `item`: the item of the item;

(3) `dest` (required): the destination of the "see also" (e.g. if the item is "wedge product", you may want to also see "exterior product", and so the destination may be "exterior product");

(4) `hyperlink`: an anchor to link to;

(5) `index link` (valueless): a flag of whether or not the anchor is within the index.

To link to an item within the index, suppose of category `C` and item `I`, set `hyperlink` to `C:I` (or just `C:` if `I` is empty), and set `index link`.

`\index`: Prints the index.

`\addtoindex {`⟨*category*⟩`}[`⟨*item*⟩`]`: Adds an item to the index of category *category* and item *item*. Its value is `\@defaultindexval` (by default `\the\pageno`), and `expand value` and `add hyperlink` are set.

3.6 Lists

In `pdfDstruct/lists.tex`, PDFTOOLBOX provides macros for creating lists of text.

There are two types of lists: unenumerated and enumerated. Unenumerated lists start with `\blist` and end with `\elist`. Each item begins with `\item`. The symbol used for each bullet point is determined by the nested depth of the list. For a depth of N, the symbol used is stored in the macro `\liststyleN`.

Similarly enumerated lists start with `\benum` and end with `\elist`. Each item begins with `\item`, and the style for the enumeration is determined by the depth of the list. For a depth of N, the nth element is styled with `\enumstyleN{n}`.

To add text in between items (not as part of the list), you can use `\mtext`.

3.7 Table of Contents

In `pdfDstruct/tableofcontents.tex`, PDFTOOLBOX provides macros for creating and displaying tables of content.

`\addtoccontent {`⟨*title*⟩`}{`⟨*value*⟩`}{`⟨*depth*⟩`}{`⟨*anchor*⟩`}`: Adds content to the table of contents whose title is *title* (e.g. chapter name), value is *value* (e.g. page number), depth is *depth*, and is linked to the anchor *anchor*. The depth *depth* determines the style used in the table (see `\settocdepthformat`).

`\tableofcontents`: Prints the table of contents.

`\settocdepthformat {`⟨*depth*⟩`}{`⟨*options*⟩`}`: Sets the format of the table of contents at the depth *depth*. *options* is a map with the following fields:

- `title`: the style switch for the title (default is `\setfont{rm}`);
- `value`: the style switch for the value (default is `\setfont{rm}`);
- `leader`: the leader to add between the title and value (default is nothing);
- `indent`: the amount to indent the line (default is 0pt);

- `buffer`: the amount of buffer to add around the line (default is 0pt).

PDFTOOLBOX provides three types of sectioning: sections, subsections, and subsubsections. Each has a counter in its name (e.g. `section`), and a macro with the current section name (e.g. `\currsection`).

`\section (*){`⟨*title*⟩`}`: Adds a section to the document. If the asterisk is added, the section is a "pseudosection": the section counter is not incremented and not displayed, and the section is not added to the table of contents. Otherwise the section counter is incremented and displayed, and the section is added to the table of contents.

`\subsection (*){`⟨*title*⟩`}`: Adds a subsection to the document. If the asterisk is added, the subsection is a "pseudosubsection": the subsection counter is not incremented and not displayed, and the subsection is not added to the table of contents. Otherwise the subsection counter is incremented and displayed, and the subsection is added to the table of contents.

`\subsubsection (*){`⟨*title*⟩`}`: Adds a subsubsection to the document. If the asterisk is added, the subsubsection is a "pseudosubsubsection": the subsubsection counter is not incremented and not displayed, and the subsubsection is not added to the table of contents. Otherwise the subsubsection counter is incremented and displayed, but the subsubsection is still not added to the table of contents.