FigPut

Interactive Figures for LaTeX
`https://github.com/rsfairman/figput`
Randall Fairman
Version 0.90, July 21, 2022

The FigPut system allows LaTeX to produce interactive figures. The system produces a static PDF file, as usual; in addition, the document can be viewed with an Internet browser and the figures become interactive. See the prototype in `example/example.tex` to clarify what follows.[1] To view this example in its fully interactive form, see `rsfairman.github.io/figput-example/`.

1 Why FigPut

There are many ways to produce figures in LaTeX: MetaPost, TikZ, PSTricks, XY-pic, `graphics`, `picture`, and more. Creating technical figures is often an exercise in programming, and none of these is based on a commonly used programming language. FigPut is based on JavaScript (JS), which is probably the most widely used language today. Moreover, FigPut allows figures to be animated and/or interactive.

Aside from the fact that figures are specified with JavaScript, the system is normal LaTeX. An important feature is that the two forms of the document – static PDF and interactive – look identical, with the same pagination and formatting. The reader can move between the printed version and the interactive version without needing to reorient.

2 Using FigPut

Here's a list of all the commands available in FigPut:

```
\begin{figput} ... \end{figput}
\FigPut
\SetInnerMargin
\SetOuterMargin
\NeverSkip
\AllowSkip
\LoadFigureCode
```

---

[1] The version available on CTAN includes only what is strictly necessary to rebuild the document from scratch: `example.tex` and `exteernalcode.js`. The repository at `github` includes all the files that result from the build process, and are necessary to view the PDF in either static or interactive form.

2.1 The LaTeX Side

Interactive figures are specified with the `figput` environment or the `\FigPut` command. The full form of the environment specification is

```
\begin{figput}{<fig_name>,<fig_ht>}[optional arguments]
  <JavaScript code>
\end{figput}
```

There are two mandatory arguments: `<fig_name>` and `<fig_ht>`. Every figure must have a unique name. This name is unrelated to any `\label` or other identifier used for the figure. The `<fig_ht>` is the height of the figure, expressed like other LaTeX lengths in `pt`, `cm`, *etc.* The figure is allocated this much vertical space on the page. The remaining optional arguments are discussed below.

The `<JavaScript code>` is used to draw the figure, and the only requirement is that it be given as

```
function <fig_name>(ctx) {
  ...
}
```

The name of the JS function must match the figure name given to the `figput` environment – that's how the browser finds the correct JS function to draw a particular figure. The `ctx` argument is required; it's the "rendering context" used for drawing.[2]

All of the JS functions for the various figures share the same name-space, and it is perfectly acceptable to share functions between figures or to define helper functions within a figure's environment. If many figures use the same `drawDoodad()` function, then that function should appear with only a single figure, and it will be available to all of them.

Use `\SetInnerMargin` and `\SetOuterMargin` to set the inner and outer margins (which may differ in a book that has left and right pages). The figure will be rendered within a rectangle that is set in from the left margin by the given amount. For example, you might say

```
\SetInnerMargin{150pt}
\SetOuterMargin{150pt}
```

so that the left edge of the figure is `150pt` from the left edge of the paper. These values can be changed before every figure, if that is appropriate.

The coordinate system used within the JS code is based on big points (`bp`), with the origin at the lower-left corner of the figure. The y-coordinate increases as you move *up* the page. The y-axis is offset from the edge of the page by the value given to `\SetInnerMargin` or `\SetOuterMargin`.

---

[2] A variable of this type is commonly obtained in JavaScript by something like

```
let canvas = document.getElementById('where-drawing-happens');
let ctx = canvas.getContext('2d');
```

The full name for this type is `CanvasRenderingContext2D`.

There is no need to begin the drawing by erasing the drawing area. That's done automatically.

Optional Arguments

The arguments to `figput` in `[]`, are optional. The first two of these arguments are

```
ht_above, ht_below
```

If these arguments appear, then they must both appear, they must be the first two arguments, and they must be given in that order (above, then below); furthermore, they must include a unit specification, like `bp`. An interactive figure may need a bit of extra clearance for an animation or space for widgets that aren't part of the figure itself. The static version of the document always uses `<fig_ht>` for the figure height, while the interactive form adds this optional amount of space above and below the figure. If `ht_below` is non-zero, then the y-axis is `ht_below` above the bottom of the figure when drawn interactively.

The remaining optional arguments may appear in any order and function as boolean flags:

`nostatic`	Use this when the figure is to be shown only when the document is viewed interactively. No figure will appear in the PDF. Almost certainly, the mandatory `fig_ht` argument should be zero. The sum of `ht_above` and `ht_below` is used for the height of the interactive figure.
`done`	indicates to the browser that the `.tikz` should not be updated.
`skip`	TikZ files can be large and take a long time to process. When this is set, the TikZ file won't be loaded and the figure will appear with a default "not available" message.

When the composition of a figure is complete, it often makes sense to turn on the `done` and `skip` flags. Using `skip` saves time, and `done` prevents inadvertent overwriting of a `.tikz` file that you're happy with. However, to see the complete output, with your figures, you would have to go back and tediously remove all of the `skip` flags. Use `\NeverSkip` to avoid that tedium and disable all the `skip` flags that occur until the next `\AllowSkip`.

The `\FigPut` Command

`\FigPut` takes exactly the same arguments as the `figput` environment, but the JS code must appear in an external file. The location of this external file is indicated with `\LoadFigureCode`. If the code is found in `mydrawcode.js`, then

```
\LoadFigureCode{mydrawcode.js}
```

must appear somewhere in the `.tex` file.

It is often easier to use an external file to consolidate the drawing code in fewer files (or a single file). Not only does it make the `.tex` file shorter and easier to navigate, but it allows the use of programming tools tailored to JS. For instance, the code could be written in TypeScript and compiled to JS. On the other hand, if the code for the figures is brief, then it may be clearer and more direct to use the `figput` environment so that the code is in the `.tex` file.

Additional Remarks

There is no explicit provision in FigPut for changing the body of the text based on whether the document is to be viewed statically or interactively. This is intentional. The aim is for the text and its layout to be *identical* in the two scenarios. However, it would be relatively straightforward to set up a couple of commands like

```
\newcommand{\statictext}[1]{#1}
\newcommand{\intereactivetext}[1]{ }
```

In the above example, anything that appears in `\statictext{}` will pass through to the LaTeX document, while everything in `\interactivetext{}` will be ignored. Swap these two definitions depending on whether the document is being compiled for interactive or static reading.

2.2 Workflow

There are several phases to composing and releasing a document. The first phase is a loop through writing the LaTeX, compiling it, viewing the output in a browser and generating TikZ, until the document is done. Everything in the first phase is local to your machine. The second phase is posting the result to a public-facing website. The first phase is discussed here; see Section 3 for the second phase.

Setup

The functionality of FigPut is given by `figput.sty`. It requires the following packages: `zref`, `xsim`, `tikz` and `verbatim`.

A typical arrangement is to have a `javascript` directory with the files for the browser side of the framework, and a LaTeX directory where the document is written in the usual way. The `javascript` directory contains the JS files and the one HTML file needed to provide the browser front-end, and you'll also find `server.py`. This Python script runs a local server so that your browser can open and view the document being written.[3] In principle, it's no different than a normal web-server, but it is tailored in several ways to the task at hand. To invoke this server, use the command-line to go to the `javascript` directory and type

---

[3] This requires Python 3 – although that's probably obvious these days.

```
python server.py directory/nameof.pdf [port_num]
```

The `directory/nameof.pdf` argument is required and should be the path from the current (`javascript`) directory to the PDF being generated by LaTeX. Often, this will be something like

```
../../latex_area/num_theory/hairy_math.pdf
```

The optional `port_num` argument indicates the TCP/IP port on which the server will listen. It defaults to 8000. In principle, any integer in the range $[0, 2^{16})$ could appear here, but certain values are restricted and others may already be in use. Stick to values in the range 8000-8100, and you should be safe.[4]

By using different port numbers, it's possible to run several servers at once, so that multiple documents can be viewed and edited at the same time. Once the server is running, point your browser to `localhost:8000` (or whatever `port_num` you've chosen) to load the document.

As stated, `server.py` differs from a normal web-server in certain ways. This is done to make it easier to work with a document while it is being written. Once the document is complete, it can be served by a normal web-server.

Additional Files

There are two additional directories in the `javascript` directory: `development` and `release`. FigPut was written in TypeScript, and the `.ts` files are in `development`. Even if you don't want to modify the system, the TypeScript files are easier to follow than the JavaScript files, and are the first place to look to clarify how the system works internally. There are four files:

- `main.ts` is the main entry-point for the program. This is unlikely to interest anyone who wants to simply use FigPut.
- `layout.ts` handles page layout – also uninteresting to most people.
- `widgets.ts` has the code for the various widgets. If widgets aren't acting the way you expect, then look here.
- `tikz.ts` handles lower-level geometry, and conversion from JS to TikZ. Look here for details about the `Point2D` and `FPath` classes.

Two files found in the `javascript` directory are `pdf.worker.min.js` and `pdf.js`. These are the source files for Mozilla's pdf.js library. The complete project can be found at `github.com/mozilla/pdf.js`. The files are included here because, when developing a LaTeX document on your local machine, it would be pointless (and slow) to download these files every time an updated version of your document is loaded.

The `release` directory contains the files to be used when the document is released to a public-facing website. As noted above, `server.py` works differently than a normal web-server. The release process is discussed in more detail in Section 3.

---

[4] See `en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`

From JavaScript to TikZ

Once the server is running, a browser can view documents created with FigPut, but the figures will not be visible in the PDF document until a TikZ file is provided. The *Get TikZ* button in the browser window generates the TikZ files. The server will save these files to the same directory as the PDF file. Note that a TikZ file will only be generated if the figure has been scrolled into view (or nearly into view) since the document was loaded.

Because TikZ files can take time to load, it's often useful to compose the figures one (or a few) at a time. Apply the optional `done` argument for any figures that are complete, and the *Get TikZ* button will not generate TikZ files for them. You may want a particular frame from an animation; once you have the right frame, it would be frustrating to accidentally overwrite it in the course of working on some other figure. Using `done` prevents that from happening.

2.3 JavaScript Drawing

In most respects, any JS code can be used with FigPut, but it must be possible to translate the JS drawing commands to TikZ, and that leads to certain requirements. In a nutshell, all drawing must be done using a class that is very similar to JS's built-in `Path2D` class.

If this is too restrictive, then it is possible to side-step the requirement with a bit of extra effort. If you use JS drawing commands that are off-limits, like `drawImage()`, then the interactive version will work fine, but you'll have to find some other way to generate the figure for inclusion in the static PDF. Any of the usual methods of generating figures for LaTeX documents will work, but now the two versions, interactive and static, are generated from different source-code.

Earlier, in a footnote on p. 2, it was a stated that the `ctx` argument to your drawing function has the standard JS type, `CanvasRenderingContext2D`. That's not always true. Any drawing done *to the browser* is *precisely* an object of this type. If you only care about seeing your document within the browser framework, and you don't care about generating figures for a standalone `.pdf` file, then that's all you need to know. Draw with JS however you like.

To produce TikZ output, the built-in `CanvasRenderingContext2D` must be "spoofed." All of the drawing commands that normally go to the built-in JS browser code go to FigPut's `CTX` class instead, where they are converted to TikZ output. This class implements a sub-set of the full list of drawing commands normally available. The permitted commands are

```
ctx.fill()
ctx.stroke()
ctx.lineWidth
```

These work almost as they normally do. The `lineWidth` is the thickness of any paths, and can be thought of as being in TeX's `bp` units. The `fill()` and `stroke()` commands normally take a JS `Path2D` object, but you should pass an object of type `FPath` instead.

The methods of `FPath` listed below are exactly like those found in `Path2D`. They add a new segment to an existing `FPath`, obtained with `new`. See any JS resource for an explanation.

- `closePath()`
- `moveTo()`
- `lineTo()`
- `bezierCurveTo()`
- `ellipse()`

And these methods of `FPath` are new – they have no counterpart in `Path2D`.

- `addPath(p)` appends the `FPath`, `p`, to an existing `FPath`.
- `translate(p)` translates an `FPath` by the given `Point2D`.
- `rotate(angle)` rotates an `FPath` by the given angle, in radians.
- `scale(s)` scales an `FPath` by the given factor (relative to the origin).
- `reflectX()` reflects an `FPath` across the x-axis.
- `reflectXY()` reflects an `FPath` across the x and y-axis.
- `rotateAbout(angle,p)` rotates a given `FPath` about the given `Point2D` by the given angle.

  The methods above apply to an existing `FPath` and return a new `FPath`.
- `parametricToBezier(f,t0,t1,n)` converts a parametric function, `f(t)`, to a path consisting of `n` Bézier curves as `t` runs from `t0` to `t1`. The function must be defined to return `Point2D` objects.

  This is a static method, so it returns a new `FPath` that is unrelated to any existing `FPath` objects.

There are several additional methods in `FPath` that may be useful, but they haven't yet been tested well enough to mention here (and they may change in the future). See the source code if you're adventurous.

Several of the methods above mention the `Point2D` class, and it's what you would expect. Use

```
new Point2D(x,y);
```

to create these objects. There's a long list of methods defined in the class: `translate()`, `rotate()`, `length()`, *etc.* See the source code for a complete list.

Text is drawn in a way that is further from how it's normally done in JS. Instead of calling `ctx.fillText()`, there are three top-level functions (not part of any class) for drawing text:

```
drawText(ctx,txt,x,y,dx,dy);
drawTextBrowserOnly(ctx,txt,x,y,dx,dy);
drawTextTikZOnly(ctx,txt,x,y,dx,dy);
```

The arguments are the same in the three cases, but the first function will draw text to both the browser window and the TikZ, while the other two will draw the text to only one destination or the other. The `ctx` is the same as the argument to your drawing function, `txt` is the string to be drawn, `x,y` is the location of the text *for the browser*, relative to the lower-left corner of the bounding-box. When the text is drawn to TikZ, it is drawn at `x+dx,y+dy`. Because the fonts used in the browser and in TikZ differ slightly, sometimes it's necessary to tweak the TikZ placement. Setting

```
  ctx.font = '10px san-serif';
```

sets the browser font to something very close the default font used by TikZ, but it's not a perfect match. The `dx` and `dy` arguments are optional and default to `0`.

As a bonus, the static method, `Numerical.newton()`, can be used to solve $f(x) = y$ for x, given y. It's a naive implementation of Newton-Raphson.

2.4 JavaScript Widgets

The foregoing framework could be used an an alternative to TikZ, PSTricks, MetaPost, *etc.*, if your aim is strictly static output. To make the figures interactive, there must be some means to accept user input – widgets.

All widgets have a static `register()` method. Do not use `new` to create widgets! Registering a widget has the effect of making it available for user input, and drawing it too (with one exception). The arguments to these `register()` methods vary with the widget, but the first argument is always the `ctx` known to the drawing function, and the last argument is always a `name` (a string). One drawing could have several widgets of the same kind and `name` is used to distinguish them. So, if a particular drawing has three buttons, then they could be created/registered with

```
let b1 = ButtonWidget.register(ctx,...,"first");
let b2 = ButtonWidget.register(ctx,...,"second");
let b3 = ButtonWidget.register(ctx,...,"third");
```

These names only need to be unique within a single drawing and one type of widget. So, in the example above, two `DraggableDotWidget`s could be registered in the same drawing as the `ButtonWidget`s with

```
let dd1 = DraggableDotWidget.register(ctx,...,"first");
let dd2 = DraggableDotWidget.register(ctx,...,"second");
```

2.4.1 `ButtonWidget`

These are registered with

```
ButtonWidget.register(ctx,x,y,text,name);
```

The `x,y` argument is the location of the button, relative to the lower-left corner. The `text` is what is shown as the message within the button. Buttons work by calling back to your drawing code whenever they are clicked. Every time the button is clicked, the entire drawing function is executed.

There are two useful fields of `ButtonWidget`. If `b` is a `ButtonWidget`, then `b.clickState` toggles between `true` and `false` with every click of the button, while `b.resetState` is set to `true` whenever the button is clicked – and remains `true` until your drawing code resets it.

2.4.2 `NumberInputWidget`

This is similar to `ButtonWidget`, and is used to input single numbers. It's registered with

```
NumberInputWidget.register(ctx,x,y,v,name);
```

The `v` argument is the initial numerical value shown in the widget. Whenever the user changes this value, the widget calls back and executes the entire drawing function. To obtain this value, call `getValue()` on the relevant widget. Because this widget relies on HTML, `getValue()` returns a string. It may be necessary to call the JS `parseFloat()` or `parseInt()` functions on this value.

2.4.3 `DraggableDotWidget`

A "draggable dot" is a small dot that can be grabbed by the mouse and moved around. They're registered with

```
DraggableDotWidget.register(ctx,x,y,name);
```

To determine where the user has moved a dot, refer to the widget's `widgetX` and `widgetY` fields.

Most widgets are used "off to the side" and wouldn't normally interfere with the way the figure is drawn, but these dots are typically used as part of the drawing itself. The order in which dots are drawn may matter. Sometimes a dot should be drawn first, so that other elements of the figure may be drawn over it; and sometimes a dot should be drawn last so that nothing can obscure it. For this reason, unlike other widgets, registering the widget does not also draw the widget. An explicit call to the widget's `draw(ctx)` method must be made to draw the dot.

2.4.4 `DraggableDrawWidget`

This is similar to `DraggableDotWidget`, but it has no default drawing behavior. The user must provide functions to specify what is to be drawn. So it can also be a draggable polygon, or a draggable anything-you-can-draw. Register these with

```
DraggableDrawWidget.register(ctx,x,y,
    drawFcn,drawSelFcn,testPosFcn,
    name);
```

The `drawFcn` must be defined to take the `ctx` as the sole argument. It can draw whatever it wants, noting that the origin is shifted so that drawing should typically take place relative to `(0,0)`, not `(widgetX,widgetY)`. In most cases, this means that the drawing code is independent of where the item is on the page. `drawFcn` must return an `FPath` object to indicate the "clickable area" for the item. For example, if what is being drawn is a small square, then the function might be defined as

```
function drawSquare(ctx) {
  let p = new FPath();
  // p.lineTo, moveTo, etc., to make a square.
  ctx.fillPath(p);
  return p;
}
```

The value returned is noted, and any click in the interior of the path – as determined by the JS function `isPointInPath()` – is taken to be a valid click on the object, to select it and drag it around.

The `drawSelFcn` argument to `register` is identical to the `drawFcn` argument, but it is called to draw the item when it is "selected." Often, the only difference between the two functions will be the color used to draw the item.

The `testPosFcn` argument is used to determine whether a particular location for the item is acceptable. For example, an item could be limited to a particular region. This is called every time the mouse is moved, after the item has been selected, up until the mouse button is released. The function takes five arguments and returns a boolean. The five arguments are the proposed `x,y` position (the mouse location), followed by the width of the drawing area, then the height above and below the x-axis, where the width is equal to the text width. Return `true` if the proposed `x,y` is acceptable; `false` otherwise. At a minimum, the function should typically check that `x,y` is inside the figure rectangle.

2.4.5 `LoopAnimWidget`

This widget and the next one (`OpenAnimWidget`) are for animations, and work similarly. A `LoopAnimWidget` is used for animations that run in a repeating loop. There are many arguments to register them.

```
LoopAnimWidget.register(ctx,x,y,scale,visWidget,
    steps,start,timeStep,
    visSteps,visFastSlow,visPauseRun,visCircle,triGrab,
    name);
```

The `x,y` values are the location of the center of the circle that is the primary element of the widget. The widget can be made larger or smaller by changing the `scale` – a value of `1.0` makes the circle 40 `bp` in radius. The `visWidget` argument is a boolean; set it to `false` to run an animation without showing the user this widget.

`steps` is the integral number of steps that make up one loop of the animation. `start` is the step on which the animation starts – so that the animation starts with the `start`-th frame. `timeStep` is the number of milliseconds between frames. Unless the machine is *very* fast, anything less than `10` for `timeStep` is probably pointless, and a setting of `50` or `75` is more reasonable considering the human eye.

The next group of arguments – `visSteps` through `triGrab` – are all booleans. They determine whether certain parts of the widget are visible. `visSteps` shows or hides an area above the circle that can be used to change the number of steps taken per frame. The user can use this control to skip frames of the animation. `visFastSlow` is for some "sideways chevrons," in an area below the circle, that allow the user to change the number of milliseconds per frame. `vsPauseRun` is for a pause/run control below the circle. `visCircle` is for the entire circle. The circle has a small grabable triangle to indicate which frame is being shown; use `triGrab` to show or hide the triangle.

To draw a particular frame from your drawing function, refer to the `curStep` field of the object. This will be a value in the range from 0 to the `steps` argument to `register()`.

2.4.6 `OpenAnimWidget`

This widget is for open-ended animations that don't run in a loop, but continue "forever." Whereas `LoopAnimWidget` indicates the frame being viewed relative to a circle, `OpenAnimWidget` uses a bar, something like a scroll-bar. Registration is similar to `LoopAnimWidget`:

```
OpenAnimWidget.register(ctx,x,y,scale,width,visWidget,
    timeStep,decay,
    visSteps,visFastSlow,visPauseRun,visBar,barGrab,
    name);
```

The `ctx`, `x`, `y`, `scale` and `visWidget` arguments are as for `LoopAnimWidget`. `width` sets the width of the bar, in `bp`.

`timeStep` is in milliseconds, as for `LoopAnimWidget`. The `decay` is used to adjust the rate of travel of the "thumb" along the bar as the animation progresses. Since the animation is potentially infinite, the thumb moves quickly early in the animation, then moves more and more slowly, so that it never quite reaches the end of the bar. The position of the thumb is given by

$$1 - \frac{1}{(1 + \texttt{decay})^s},$$

expressed as a fraction of the bar's total length, where s is the frame count (*i.e.*, `curStep`). A reasonable setting for `decay` is something in the range of 10^3 to 10^6, although it will depend on the animation.

The arguments from `visSteps` to `barGrab` are all boolean and control which controls that make up the widget are visible. They're like their counterparts in `LoopAnimWidget`, except that `OpenAnimWidget` uses a bar instead of a circle.

3 Distributing a Complete Document

The PDF document is no different than any other PDF, and can be distributed in the usual ways. When moving the document in interactive form to a public-facing website, a few steps are necessary.

See the `javascript/release` directory. It has two files: `figput.html` and `figput.js`. The `.js` file is just the four files used for the local version (`main.js`, *etc.*), consolidated to a single file and minified. These two files must appear on the web-server in the same directory as the various files that make up your document.

`figput.html` is only a few lines long, and there are two important changes that must be made. See the line that reads

```
<body onload="doOpenDocument('unknown1',unknowny)" id="mainbody">
```

Change `unknown1` to your document's name, *without* the `.pdf` suffix. If the document was generated from `example.tex`, then `unknown1` should be changed to `example`. Second, change `unknowny` to `0`. When the document is opened, this is the vertical position, in `bp`, at which the document is opened. Setting `unknonwy` to `0` means that the document will open at the top of the first page. Of course, it's possible that you want the document to open at some other location, in which case use some other value for `unknowny`.

The server needs `figput.html` (after the modifications above), `figput.js`, and the files that make up your document. These files consist of the `.pdf` file, the `.fig.aux` file, any `.js` files referred to by `\LoadFigureCode`, plus any `.fjs` files generated by your document. The server does not need the `.tikz` files.

4 Code Internals

An understanding of this section isn't necessary to use FigPut; it's here to provide a bit of background for anyone who wants to improve or modify the framework. Most of FigPut is in TypeScript, which should make it easier to follow.

Throughout the various files that make up the system are `BUG` annotations. These are not bugs *per se*, but are more like infelicities, opportunities for improvement, or to draw attention to something confusing or problematic. If you want to contribute to FigPut, then these are a good place to start.

4.1 `figput.sty`

The `.sty` file is well-commented and should be self-explanatory. However, I am far from an expert in LaTeX3 programming. Here is a high-level list of ways in which it might be possible to improve things.

- `figput.sty` is written for LaTeX3. The entire thing might be simpler if it were in LuaTeX.
- The ability to explicitly specify the inner and outer margins used by FigPut (with `\SetInnerMargin` and `SetOuterMargin`) is useful, but it would often suffice to default to the margins used by LaTeX. How can those values be obtained?
- Someone who is more expert in LaTeX3 could improve the code's clarity and brevity.
- It would be nice if FigPut could create figures within a `minipage`, but that would require extensive changes to both the LaTeX and browser code.

A Quirk

LaTeX on Windows has a quirk. TeX will not write to a file whose type is executable. Thus, it won't write to a `.pl` file (Perl), `.py` (Python) file, *etc.*, depending on how the machine is set up. FigPut creates JavaScript files, which are typically saved as `.js` files, and these may be seen as executable by Windows. Fortunately, browsers doesn't care what the suffix is, so FigPut uses `.jfs` as the suffix.

This limitation on the file name suffix applies to `\write`, but not to `\write18`. TeX is limited in this way to prevent accidental errors and for security. One solution might be to have the package write files using an acceptable suffix, then change the suffix with `\write18`, which does not have this limitation. However, to use `\write18` this way, LaTeX would have to be invoked with something like

```
pdflatex -shell-escape
```

which would be annoying.

4.2 `server.py`

The server is a simple thing, so there's not much to say. It would be nice if it were written in a compiled language so that the user doesn't need to install Python.

4.3 Browser Code

This part is written in TypeScript (TS), which is essentially JS with type annotations. It must be translated to JS for the browser to run it, but TS code is easier to follow, and the type information heads off many programming errors.

Although the program is (to my knowledge!) bug-free in the usual sense, there are plenty of `BUG` annotations to draw attention to places where the code could be improved. Some of the higher-level possibilities for improvement are noted below.

- FigPut relies on the `pdf.js` library for rendering PDFs. The result is not as crisp as I would like. It's unclear whether the problem is with the library itself, or how it is being used. Someone who knows that library well may be able to improve the output. Maybe a different library would work better.
- Most of the widgets do not rely on the DOM, but `NumberInputWidget` and `ButtonWidget` are based on the built-in DOM elements. It would be nice if these didn't use the DOM.
- Many additional widgets could be created.
- Adjust the font used for figures in the browser to make it more nearly identical to what appears in the PDF.
- Color is entirely neglected on the LaTeX end of FigPut. Browser output can be colored arbitrarily (with `ctx.fillStyle` or `ctx.strokeStyle`, as usual), but the TikZ output is limited to black and white.
- Double-buffer everything. Animations seem to run fine, without any noticeable flicker, but double-buffering *all* drawing would put an end to any concerns about flicker.
- Animations should be turned off and stop generating events when they are scrolled out of view. This would save CPU cycles, which may matter for documents with many animations.

And some items that are more ambitious...

- Allow figures to be set within a `minipage` (mentioned above).
- Double-clicking a figure (or something) could enlarge the figure to give more room for interaction.
- Come up with a scheme so that the code that draws a figure could be made part of the document. For certain topics, the code that draws a figure can be as instructive as the figure itself.
- Extend the framework somehow so that it can be used within an IDE. Code that's mathematically-oriented is difficult to document within source code using simple text. It would be easier to understand if it's explained as LaTeX output that appears in the source code, rather than in a separate document.
- Extend the framework to allow inclusion of snippets of LaTeX output in standard HTML documents. Things like KaTeX and MathJax are convenient, but incomplete relative to LaTeX.

Version History

- 0.90 – July 21, 2022. I've been using this, privately, for several months. Numerous improvements and extensions are possible, but it definitely works.

Thanks to Dan Pratt for insights regarding TypeScript, and improvements to the code.

Several people on `tex.stackexchange` very patiently answered numerous questions. Thank you.