Using `feynr`

Peter Schmidt-Nielsen

January 8, 2014

Installation

You can download `feynr` at `http://github.com/petersn/feynr`. All you need are the files `feynr.sty` and `feynr_render.py`.

Best Practices Install

Mark `feynr_render.py` as executable, and place it on your path. Add `feynr.sty` to your TEXpath, and rerun `texhash`. You are now set up.

Lazy (/Non-Root) Installl

Copy both `feynr.sty` and `feynr_render.py` to the directory for your project. Instead of running `feynr_render.py source.tex` to rebuild your diagrams you'll have to run `python feynr_render.py source.tex`. C'est la vie.

Overview

Include `feynr` with the usual `\usepackage{feynr}`. Place your diagram source code in a `feynr` environment to render a diagram. A minimalish example:

```
\documentclass{article}
\usepackage{feynr}
\begin{document}
\begin{feynr}
feynr diagram specification
\end{feynr}
\end{document}
```

Run `feynr_render.py` on your LATEX file after each time you update a `feynr` diagram:

```
feynr_render.py source.tex
pdflatex source.tex
```

Basic Usage

Anywhere in your preamble you may specify comma separated global options to `feynr` in the `\feynroptions` command. For example, `\feynroptions{time-right}` will override the default of `time-up`, and switch away from Feynman's original convention. A complete list of options can be found at the end of this document.

In `feynr`, particles are explicitly tracked. You can create an input particle to your diagram with the `input` command. For example, `input a` defines an input particle called `a`. Special flags can be passed to any `feynr` command with a pair of dashes. For example, to declare an input particle `a` that's an electron we may use: `input a -- electron`. More than one particle may be declared in a single command, and the flags passed apply to all particles. For example, to declare three positrons: `input p1 p2 p3 -- positron`. Particle types may be omitted if they can be unambiguously inferred from elsewhere in the diagram.

Particles can be made to interact with each other with the `interact` command.[1] For example, `interact p1 p2` will cause the particles `p1` and `p2` to exchange a particle. Again, the interaction type may be unambiguous. For example, if `p1` and `p2` are photons, and `p1` is already known to emit a positron (from another `interact` command), then the interaction is unambiguous, and an electron will be emitted from `p1`, and propagate to interact with `p2`. However, frequently you will have to specify what form the interaction takes. The flags `electron`, `positron`, and `photon` denote that the given particle propagated from the first argument to the second argument. For example, `interact p1 p2 -- electron` causes an electron to be emitted by `p1` and absorbed by `p2`.[2] Note that as `interact` takes exactly two node arguments, the `--` separating flags from node arguments is optional. You may write either `interact a b -- flags` or `interact a b flags`, these are equivalent. Just like `input`, there is a command `output` that can be used to disambiguate diagrams. For example, if, at the end of a sequence of interactions, it's ambiguous what type of particle `a` is, we can specify with, for example: `output a -- photon`. You can identify interactions in your diagram that couldn't have their type inferred – they will appear as red sine waves.

Finally, we need to give `feynr` some hints about the layout of our diagram. If you attempt to compile a diagram without any layout hints it will fail, as it can't figure out what angles the various lines should be drawn at. There are two default layout hints, `0space` and `0time`. These hints indicate that the given interaction should be drawn perpendicular to the specified axis. The naming convention is that a `0time` interaction has zero length in time, and a `0space` interaction has zero length in space. In other words, in a time-up Feynman diagram, `0time` indicates that the interaction should be drawn horizontally. For example, we can write `interact e1 e2 photon 0time` to cause particles `e1` and `e2` to exchange a photon that is drawn perpendicular to the time axis.

The `propagate` command accumulates flags to be added on the next propagation

[1] The command `interact` can also be abbreviated as a single dash.

[2] Naturally, `interact a b -- electron` is equivalent to `interact b a -- positron`.

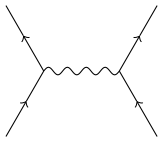
that a given particle makes. For example, if we would like the particle `a` to propagate straight along the time axis the next time it propagates, we can issue the command `propagate a -- 0space`. Like all flags in `feynr`, they apply only to a single interaction/propagation of a particle, and are not permanent. Thus, order matters. In the sequence `interact a b`, `propagate a -- flags`, `interact a c`, the extra flags given in the `propagate` command affect how particle `a` gets from the interaction with `b` to the interaction with `c`.

Putting these all pieces together, let's look at some examples:

```
input e -- electron
propagate e -- 0space
```

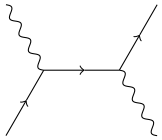

The first line defines an electron, and the second is a layout hint that it should propagate along the time axis.

```
input e1 e2 -- electron
interact e1 e2 photon 0time
```



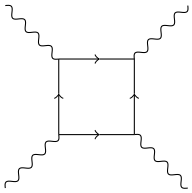
Here two electrons are defined in the first line, and made to exchange a photon along the space axis in the second.

```
input e -- electron
input p -- photon
interact e p 0time
```



Not substantially more complicated than the previous example.

```
input p1 p2 -- photon
interact p1 p2 electron 0time
propagate p1 p2 -- 0space
interact p1 p2 electron 0time
```

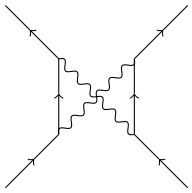


Here Delbrück scattering is concisely described in four lines. The third line (`propagate p1 p2 -- 0space`) makes sure that the electrons propagate vertically in the diagram, forming a nice square box when used with the horizontal `0time` interactions.

Advanced Usage

Let's look how we might lay out a more complicated diagram.

```
input p1 p2 -- electron
skeleton p1 p2 photon 0time
propagate p1 p2 -- 0space
skeleton p1 p2 photon 0time
draw p1~2 p2~3 photon
draw p2~2 p1~3 photon
```

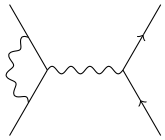


Okay, there's a lot to break down here. We're using two new commands, `skeleton` and `draw`. The command `skeleton` is exactly equivalent to `interact`, only it automatically passes the flag `hidden`, a special flag that causes the interaction to not be rendered. That is to say, `interact a b hidden` is exactly equivalent to `skeleton a b`. Crucially, the interaction is still used for type-inference, and used for structural layout – thus `skeleton`. The purpose of the `skeleton` command is to allow you to quickly build the desired shape and types of your diagram without worrying too much about how you're going to fill in the interactions. Further, it allows for a tremendous amount of uniformity and control of the generated diagrams. You can produce a skeleton interaction, then later fill in multiple different kinds of interactions between the skeleton interacting particles without changing the layout of your diagram by a single pixel. What about `draw`? Just like `skeleton`, `draw` is a version of `interact` that passes an extra flag, in this case `free`. The `free` flag says that the given interaction is purely for display purposes, and does not affect type-inference or structural layout.

In the above diagram `skeleton` is necessary, and here's why: Unfortunately, we have a dependency loop: `p1` is interacting with a future version of `p2` and vice versa. There is nothing to be done about this, so we simply produce two skeleton interactions that form the same box as before with our Delbrück scattering. Finally, we use `draw` to add photons between `p1` and `p2`, but we must specify the out-of-order interaction we want. Thus, we introduce nodes. Each particle produces a trail of nodes along its path. For example, `p1` has four nodes: the four leftmost vertices in the above diagram. These vertices are automatically internally labeled `p1~1`, `p1~2`, `p1~3`, and `p1~4`. These correspond to `p1`'s input node, two nodes connected to photons, and output node respectively. By issuing the command `draw p1~2 p2~3 photon` we render a photon from the second node on `p1`'s path to the third node on `p2`'s path. This corresponds to the photon propagating up and to the right in the above diagram. In this way, every vertex in our diagram receives a node label, and we can add extra features to our diagram without too much effort.

When drawing Feynman diagrams we frequently want to add an emitted photon without affecting the layout of our diagram. Frequently we want this photon to be reabsorbed elsewhere in our diagram. Just like before where we used `skeleton` and `draw` to separate the layout and components of our diagram, here we can use `draw` to add additional details without distoring our diagram. Let's examine the following sort of diagram:

```
input e1 e2 -- electron
propagate e1 -- no-arrow
interact e1 e2 photon 0time
propagate e1 -- no-arrow
draw e1~1:e1~2 e1~2:e1~3 photon arc
```



We've introduced two new flags, `no-arrow` and `arc`, and a new syntax which can be used in place of a node argument to `draw`. We've got almost the same diagram as in our very first example, except for the final `draw` command, which adds an arcing photon. When specifying nodes in a `draw` command you may optionally use the syntax `a:b` to anchor one end of the drawn connection to half way between node `a` and `b`. For example, in this case the first end of our arcing photon is anchored half way beteween `e1~1` and `e1~2`, namely the midpoint of the lower leftmost edge. The flag `arc` simply specifies that the drawn connection should be a segment of a circle rather than a straight connection – much nicer looking in this case. Compare to the unarced version to the right. Finally, we need to supress the arrows on the lines for `e1`, because they would overlap with the photon in an ugly way. We can do this by passing the `no-arrow` flag into `propagate` in the right two places, namely before and after the interaction.

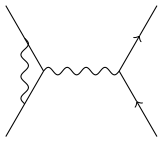
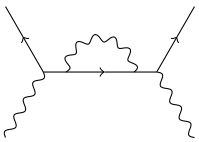


Figure 1: Ugly!

Sometimes we may wish to render a diagram like the following, with a self-interaction on an edge. Here we're using the extended syntax `proportion:a:b`, which sets an

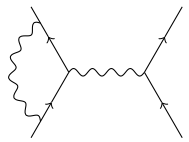
```
input p1 p2 -- photon
interact p1 p2 electron 0time long
draw 0.8:p1~2:p2~2 0.2:p1~2:p2~2 photon arc
```



anchor point part way between two nodes. Here we draw a photon between two points 20% and 80% along the middle edge. If we had wanted the arc to be below the edge instead of above it we could add the flag `flip`. Additionally, I passed the argument `long` to make the middle edge 50% longer, giving some extra room for the photon semi-circle. However, if I were laying this diagram out as a fourth order upgrade to the second order version without the extra loop, I could choose to not pass `long`, and have a diagram that beautifully differs from the lower order version *only* in the extra looped photon.

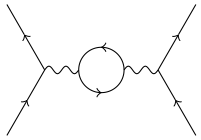
We can also use this proportion feature to make room for arrows in the previous fourth order Coulomb repulsion diagram. It's up to you to decide which you think

```
input e1 e2 -- electron
interact e1 e2 photon 0time
draw 0.25:e1~1:e1~2 0.75:e1~2:e1~3 photon arc
```



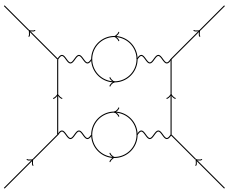
looks nicer – I really like putting the arrows back in. Using just these arc loops we can produce other useful features, such as circular electron propagation.

```
input e1 e2 -- electron
skeleton e1 e2 photon 0time long
draw e1~2 0.3:e1~2:e2~2 photon
draw 0.3:e1~2:e2~2 0.7:e1~2:e2~2 electron arc
draw 0.7:e1~2:e2~2 0.3:e1~2:e2~2 electron arc
draw 0.7:e1~2:e2~2 e2~2 photon
```



There's a lot of stuff going on here, again using our powerful combination of `skeleton` and `draw`. It's a bit of a hack, but let's go through it. The `skeleton` command gives the type, length, and angle of the interaction, laying out the structure (same as always). The first and last `draw` commands draw little lengths of photons up along the first and last 30% of the middle edge. Finally, the middle two `draw` commands draw the two halfs of the circle. Note that the order of the arguments is swapped between them, which is why they aren't just drawn on top of each other. Of course, we could also have done it by using the `positron` and `flip` flags on one, and keeping the argument order the same. However, I think this looks cleaner. For convenience, the above four draw commands can be acheived on the edge between `a` and `b` with the single macro command `photon-loop a b`. For example:

```
input e1 e2 -- electron
skeleton e1 e2 photon 0time long
propagate e1 e2 -- 0space
skeleton e1 e2 photon 0time long
photon-loop e1~2 e2~2
photon-loop e1~3 e2~3
```



Reference

Commands:

Command	Arguments
`input`	`name1 name2 ...` optionally: `-- flag1 flag2 ...`
`output`	`name1 name2 ...` optionally: `-- flag1 flag2 ...`
`propagate`	`name1 name2 ...` optionally: `-- flag1 flag2 ...`
`interact`	`node1 node2` optionally: `flag1 flag2 ...`
`-`	`node1 node2` optionally: `flag1 flag2 ...`
`skeleton`	`node1 node2` optionally: `flag1 flag2 ...`
`draw`	`location1 location2` optionally: `flag1 flag2 ...`
`photon-loop`	`node1 node2`

In the above specifications, the field `name` can be any particle name, like `xyz`. The field `node` can be a particle name or particlar node of that particle, like `xyz` or `xyz~2`. Finally, the `location` field can any of the above *plus* the special syntaxes `abc~1:xyz~2` or `proportion:abc~1:xyz~2`. Valid flags for nodes are:

`electron`, `positron`, `photon`, `input`, `output`

Valid flags for interactions are:

`electron`, `positron`, `photon`, `hidden`, `free`, `0time`, `0space`, `arc`, `flip`, `no-arrow`, `long`

Valid options to pass to `\feynroptions` are:

Option	Description
`time-up`	Same as `angle=0, flip=0`
`time-down`	Same as `angle=0, flip=1`
`time-left`	Same as `angle=270, flip=1`
`time-right`	Same as `angle=270, flip=0`
`flip=<0 or 1>`	Sets whether the y-axis is mirrored.
`angle=<degrees>`	Rotate the diagram counter-clockwise.
`photon-frequency=<num>`	Change the frequency of photons.
`photon-amplitude=<num>`	Change the amplitude of photons.

The default is:

```
\feynroptions{time-up, photon-frequency=1.0, photon-amplitude=1.0}
```