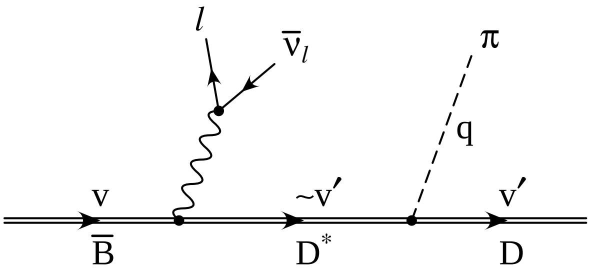
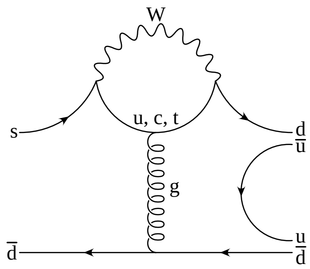




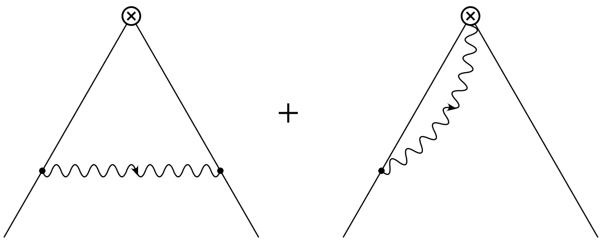
FeynDiagram

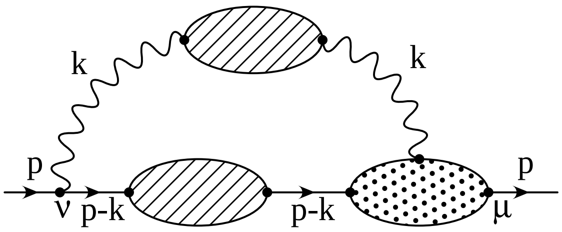
Version 2.5

Tutorial

by Bill Dimm

bdimm@hotneuron.com





This software may be obtained from http://www.erols.com/bdimm

Contents

To users of previous versions of FeynDiagram: There is an incompatibility between version 2.1 and the versions that came before it. As of version 2.1, all `true()` functions have been changed to `settrue()` and `false()` has been changed to `setfalse()`. This was done because `true` and `false` have become reserved words in the ANSI C++ standard.

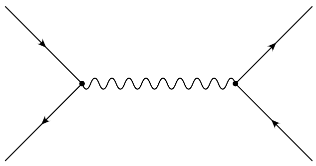
Introduction

FeynDiagram (pronounced "Fine Diagram" of course) is a set of C++ objects which allow you to easily produce high quality Feynman diagrams by writing a short C++ program to describe the diagram. Output is in PostScript* which can be printed on a laser printer or viewed with a program such as ghostscript(gs)†. This manual gives a brief description of **FeynDiagram** with examples. Prior knowledge of C++ is not needed, but familiarity with C is assumed. Even if you do not know C, you ought to be able to achieve a reasonable understanding of **FeynDiagram** by following the examples. All of the example programs in this manual were included with the software.

The "largest" object is a *page*, which describes one page of output. Each page may have several Feynman diagrams on it, which are represented by a class called *FeynDiagram*. Each *FeynDiagram* contains lines, vertices, text, etc. To specify the positions of these objects, each *FeynDiagram* has its own coordinate system with points being denoted by objects called *xy*. An example will help:

Program 1

```
1  #include <FeynDiagram/fd.h>

2  main()
3     {
4     page pg;
5     FeynDiagram fd(pg);

6     xy e1(-10,5), e2(-10,-5), e3(10,5), e4(10,-5);
7     vertex_dot v1(fd,-5,0), v2(fd,5,0);

8     line_plain f1(fd,e1,v1);
9     line_plain f2(fd,v1,e2);
10    line_wiggle photon(fd,v1,v2);
11    line_plain f3(fd,v2,e3), f4(fd,e4,v2);

12    pg.output();
13    return 0;
14    }
```



On line 1 (these line numbers are **not** part of the C++ program), we include "FeynDiagram/fd.h" which defines all of the classes for the **FeynDiagram** package. On line 4 we define a *page* called `pg`. Line 5 then attaches a *FeynDiagram* onto `pg` – we will only put this one Feynman diagram on the page. On line 6 we define four objects, `e1`...`e4`, of type *xy*. An *xy* is simply a two component vector (the rather brief name *xy* was chosen because these objects occur all over the place, as you will see). The *xy*'s defined on line 6 will be the starting and ending points of the external lines. The arguments in parenthesis are simply

* PostScript is a registered trademark of Adobe Systems Inc.

† by Aladdin Enterprises

cartesian coordinates. On line 7 we define the two vertices. These are attached to `fd` and have the cartesian coordinates given by their second and third arguments. Notice that all objects that make up a Feynman diagram have a *FeynDiagram* as their first argument so that they are automatically attached to the right diagram. Other objects such as *xy* and *page* do not have a *FeynDiagram* as their first argument.

On line 8 we define an incoming fermion line. It is an incoming "fermion" because it starts at `e1`, an external point, and ends at `v1` which is a vertex. In contrast, on line 9 we define an incoming "anti-fermion" – the second argument (where it comes from) is a vertex. You might be thinking that lines 8 and 9 are inconsistent since their second and third arguments are of different types – one is an *xy* and the other is a *vertex_dot*. This is okay because **FeynDiagram** has rules for converting a *vertex_dot* (or any other type of vertex) into an *xy*. So, at any place where the coordinates of a point are needed, a vertex may be substituted (a vertex, however, is more than just a point since it produces output on the page). On line 10 we define a photon line which connects the two vertices. On line 11 we define the outgoing fermion and anti-fermion. Then line 12 tells `pg` to output itself (actually, this will output all of the pages in the program – you won't be able to use your output in other programs like TEX* if you use more than one page). Since we didn't tell it where the output should go (we didn't give an argument to `pg.output()`), it is sent to stdout. If we don't redirect the output of this program when we run it, stdout will be the screen – quite a mess! If we called the compiled program "fig1", we might execute it as "fig1 | lpr" to send it to the printer (or perhaps "fig1 | lp -T PS" depending on your Unix system), or "fig1 >fig1.ps" to create a PostScript file which we could view on the screen. If you are on a Unix computer, you can get instructions on how to compile your program and other information about your local version of **FeynDiagram** by typing "man feyndiagram".

If you haven't used C++ before, some of this may seem a little strange. Lines 4 through 11 are definitions. In C you define things like "`double x;`" and "`int i;`". In C++ you can create your own types (in our case classes), which is what *page*, *FeynDiagram*, *xy*, *vertex_dot*, *line_plain* and *line_wiggle* are. When you define objects, you give them initial values by putting arguments in parenthesis after the variable name (in C you do something like "`int i=1;`"). C++ classes are similar to structures in C. One of the differences from C is that in addition to having members which are variables, you can also have members which are functions. An example of this is shown on line 12 where we invoke the `output()` function which is a member of the *page* class. Note that some C++ compilers require the filename for your source to end in ".C" (upper case).

Throughout, all angles are in degrees. Also, unless stated otherwise, all lengths, locations, etc. are measured in the coordinate system of the *FeynDiagram* involved.

Vertices

There are several different vertex types supported by **FeynDiagram**. Samples of the output from each are shown below.

•	vertex_dot	⊗	vertex_circlecross
×	vertex_cross	■	vertex_box

Any of these may be defined like:

```
vertex_dot v(fd,xy);
vertex_dot v(fd,x,y);
```

Where `fd` is a *FeynDiagram*, `xy` is an *xy*, `x` and `y` are *double*, and `v` is the name of the variable you are defining. `xy` or `x` and `y` specify the position of the vertex in the coordinate system of `fd`. C++, like ANSI C with prototypes, knows the type of the arguments, so if you give an integer for `x` or `y`, it will automatically be converted to a double. Additionally, *vertex_cross*, *vertex_circlecross*, and *vertex_box* have an optional argument which gives the orientation of the vertex. For example:

```
vertex_circlecross v(fd,x,y,angle);
```

Where `angle` is a *double* which gives the angle in degrees. The default value is 0°. There is another type of vertex not listed above. It is *vertex*, and it produces no output on the printed page (generally, an *xy* can be used in place of a *vertex* since a *vertex* does nothing more than keep track of a location).

The appearance of the vertices may be adjusted by changing their "parameters". This may be done for a single vertex or all the vertices in a diagram simultaneously. See the section titled "Adjusting Parameters"

---

* TEX is a trademark of the American Mathematical Society

for more information. The parameters for all vertices are:

radius (d) - the radius of the vertex
thickness (d) - the thickness of lines used to construct the vertex

Lines

The various lines (propagators) supported by **FeynDiagram** are shown below.

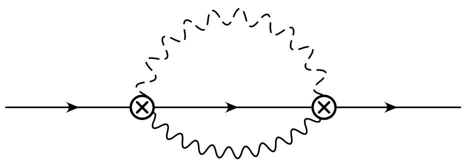
All of these lines may have an arrow attached to them, but only *line_plain* and *line_doubleplain* have arrows by default. All lines are defined like:

```
line_plain lin(fd,begin_point,end_point);
```

Where `fd` is a *FeynDiagram*, and `begin_point` and `end_point` are of type *xy*. A line follows some specified curve. By default, this curve is a straight line connecting the given points. You can change the curve to an arc by using the `arcthru` member function to specify a third point. This is done in the example below on line 10 where the first photon line is made to follow an arc through the point `arcpt` which is defined on line 6. On line 12 `arcthru` is used with x and y values passed as arguments rather than an *xy*. The curve associated with a line is parameterized by a variable which takes on values between 0 and 1. You can find a point on the curve or the tangent vector at a point on the curve. An example of this is given in the section titled "Manipulations with xy."

Program 2

```
1  #include <FeynDiagram/fd.h>

2  main()
3     {
4     page pg;
5     FeynDiagram fd(pg);

6     xy e1(-10,0), e2(10,0), arcpt(0,4);
7     vertex_circlecross v1(fd,-4,0), v2(fd,4,0);
8     line_plain f1(fd,e1,v1), f2(fd,v1,v2), f3(fd,v2,e2);
9     line_wiggle photon1(fd,v1,v2), photon2(fd,v1,v2);

10    photon1.arcthru(arcpt);
11    photon1.dashon.settrue();
12    photon2.arcthru(0,-2);
13    photon2.width.scale(0.7);
14    pg.output();
15    return 0;
16    }
```



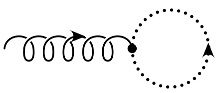
On line 11 in the example above, the *dashon* parameter (described below) is used to turn dashing on for `photon1`. The *width* parameter is used on `photon2` on line 13 to make the wiggles smaller than the default by a factor of 0.7.

The dashes in the dashed lines have rounded ends. When describing the length of dashes, the endcap used to give the rounded edge is not included in the length of the dash. So, to get a dotted line you can set the *dsratio* (described below) to zero, which tells it that the dashes should have zero length. But, the endcaps will still be drawn, giving a dotted line.

To draw a tadpole loop, your line will have the same beginning and ending point. But, this isn't enough information – **FeynDiagram** doesn't know what direction the line goes in (ie. which way should the arrow point if there is one). By default, it assumes that the tadpole goes in the clockwise direction. To make it go counterclockwise, pass an additional argument of `1` to `arcthru()` as shown below on line 10. Notice that on line 13 we make the the ghost line 1.8 times thicker than a normal line. This is generally a good idea for dotted lines so that they aren't too light. On line 7, the `xy(-5,0)` constructs a temporary object of type *xy*.

Program 3

```
1  #include <FeynDiagram/fd.h>

2  main()
3     {
4     page pg;
5     FeynDiagram fd(pg);

6     vertex_dot vtx(fd,0,0);
7     line_spring gluon(fd,xy(-5,0),vtx);
8     gluon.arrowon.settrue();
9     line_plain ghost(fd,vtx,vtx);
10    ghost.arcthru(3,0,1);
11    ghost.dashon.settrue();
12    ghost.dsratio.set(0);
13    ghost.thickness.scale(1.8);
14    pg.output();
15    return 0;
16    }
```



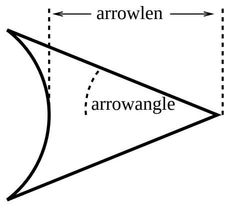
Lines which have a *width* parameter can be flipped by making the *width* negative. This is shown in the example below. Also, it is sometimes desirable to make a line look as if it passes over another line. The `ontop` member function handles this as shown in the example below. The argument to `ontop` is an integer in the range [1,9] which tells **FeynDiagram** how to order the lines (ie. which lines go on top of which other lines). For example, if there were three lines which crossed, you would use `ontop(1)` for one of them and `ontop(2)` for another. You wouldn't call `ontop` for the third line (which goes on the bottom).

Program 4

```
#include <FeynDiagram/fd.h>

main()
   {
   page pg;
   FeynDiagram fd(pg);

   line_spring gluon(fd,xy(-10,0),xy(-2,0));
   gluon.arcthru(-6,3);
   line_spring gluonflip(fd,xy(2,0),xy(10,0));
   gluonflip.arcthru(6,3);
   gluonflip.width.scale(-1);
```

```
    line_plain fermion(fd,xy(-6,0),xy(-2,3));
    fermion.ontop(1);
    pg.output();
    return 0;
    }
```

All lines have the following parameters:

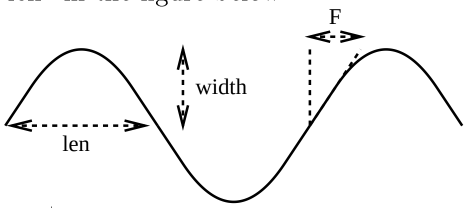


thickness (d) - the thickness of the line
dashon (b) - whether or not to make the line dashed
spacelen (d) - length of the space between dashes (if there are any)
dsratio (d) - ratio of length of dash to the length of space between dashes
arrowon (b) - whether or not to put an arrow on the line
arrowlen (d) - length of the arrow as shown
arrowangle (d) - angle shown, in degrees
arrowposition (d) - location along curve for arrow, between 0 and 1

line_doubleplain has the additional parameter:
lineseparation (d) - distance between centers of two lines

line_zigzag, *line_wiggle*, and *line_spring* have the additional parameters:
width (d) - proportional to the width from center of the curve as shown below
lwratio (d) - ratio of "len" in figure below (this will stretch) to *width*

line_wiggle has the additional parameter:
fract (d) - ratio of "F" to "len" in the figure below



line_spring has the additional parameters:
threeD (b) - whether or not to make it look three dimensional
threeDgap (d) - size of gap used to make it look 3-D
fract1 (d) - ratio of length of little arc to length of big arc

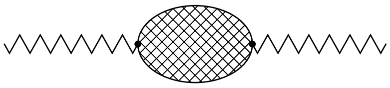
Blobs

A *blob* is an ellipse which may be shaded. To create a blob, do:
`blob bl(fd,center,rx,ry);`
Where `fd` is the *FeynDiagram* which it is attached to, `center` is an *xy* which specifies the center of the ellipse, and `rx` and `ry` are *double*'s which give the radii in the x and y directions respectively. The blob will be empty unless you add shading. Let's look at an example:

Program 5

```
#include <FeynDiagram/fd.h>

main()
    {
    page pg;
    FeynDiagram fd(pg);

    xy e1(-10,0), e2(10,0);
    blob b(fd,xy(0,0),3,2);
    b.addshading_lines(45);
    b.addshading_lines(-45);
```

```
10    vertex_dot v1(fd,b.location(180)), v2(fd,b.location(0));
11    line_zigzag l1(fd,e1,v1), l2(fd,v2,e2);

12    pg.output();
13    return 0;
14    }
```



A blob is defined on line 7. On lines 8 and 9 the blob is shaded with lines along 45 and -45 degrees respectively. If we call the `addshading_lines()` or `addshading_dots()` member function without telling it what angle to use, it will use the default value. Line 10 defines two vertices which lie on the ellipse of the *blob* `b`. This is done by using the *blob* member function `location()`, which returns an *xy* representing a point on the ellipse at an angle given by the argument to `location()`.

blob's have the parameter:

thickness (d) - the thickness of the outline

Shading works by attaching a shading object to the blob each time there is a call to a *blob*'s member function to add shading. These shading objects also have parameters.

All shading objects have the parameter:

angle - angle for the shading. Normally set by `addshading_lines()` etc.

Parameters for *shading_lines*:

thickness (d) - thickness of lines

spacing (d) - distance between lines

Parameters for *shading_dots*:

radius (d) - radius of dots

spacing (d) - distance between centers of dots

Text

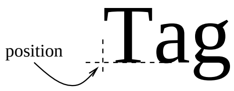
FeynDiagram takes text strings in a format similar to TeX. Unlike TeX, you do not need to enclose mathematical objects in $'s, and spaces are translated without any modification. To produce a subscript, preceed the character with an "_" (underscore). To produce a superscript, preceed the character with a "^". To superscript or subscript multiple characters, group them in braces "{}". To override the meaning of special characters such as braces and underscores, preceed them with a "\" (backslash). Note that the C++ compiler interprets backslashes which occur in constant text strings, so you must replace all of the backslashes you would normally type by two backslashes in your C++ program. To put a line over some text do `\overline`{text}; for a tilde do `\widetilde`{text}; for a tilde with a bar under it do `\tildebar`{text}; and for an arrow do `\overrightarrow`{text}. Also note that the character `'` produces an apostrophe, while `\prime` produces a prime symbol. You can change fonts in the middle of a string of text by doing `\font`{fontname}{text}. **FeynDiagram** understands the following control sequences:

\alpha	α	\varphi	φ	\approx	≈
\beta	β	\chi	χ	\conj	≅
\gamma	γ	\psi	ψ	\propto	∝
\delta	δ	\omega	ω	\perp	⊥
\varepsilon	ε	\Gamma	Γ	\cdot	·
\zeta	ζ	\Delta	Δ	\times	×
\eta	η	\Theta	Θ	\ast	∗
\theta	θ	\Lambda	Λ	\bullet	•
\vartheta	ϑ	\Xi	Ξ	\leftarrow	←
\iota	ι	\Pi	Π	\Leftarrow	⇐
\kappa	κ	\Sigma	Σ	\rightarrow	→
\lambda	λ	\Upsilon	ϒ	\Rightarrow	⇒
\mu	μ	\Phi	Φ	\leftrightarrow	↔
\nu	ν	\Psi	Ψ	\Leftrightarrow	⇔
\xi	ξ	\Omega	Ω	\uparrow	↑
\pi	π	\Re	ℜ	\Uparrow	⇑
\rho	ρ	\Im	ℑ	\downarrow	↓
\sigma	σ	\partial	∂	\Downarrow	⇓
\varsigma	ς	\infty	∞	\langle	⟨
\tau	τ	\nabla	∇	\rangle	⟩
\upsilon	υ	\equiv	≡	\vert	|
\phi	ϕ	\sim	∼		

Objects of type *text* are defined:

```
text tx(fd,str,position,angle);
```

Where `fd` is the *FeynDiagram* which it is attached to, and `str` is a text string of type *(char *)*. `position` is an *xy* giving the position of the point shown below, and `angle` is a *double*. `angle` is optional, and gives the orientation of the text. It defaults to 0°.



There is another syntax for the *text* definition that allows special positioning:

```
text tx(fd,str,position,xfract,yfract,angle);
```

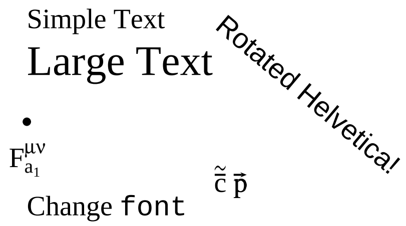
This allows the `position` argument to refer to a different point in the text than the one shown in the diagram above. **FeynDiagram** will find the boundary of a box around the text, and then use `xfract` and `yfract` to determine which point `position` refers to. For example, with `xfract` and `yfract` both set to 0.5, `position` will refer to the center of the box which encloses the text. On line 11 in the example below, `xfract` and `yfract` are set to 0.5 and 1.0 respectively. In that case, `position` refers to the location of the center of the top of the text.

Now an example:

Program 6

```
1  #include <FeynDiagram/fd.h>

2  main()
3     {
4     page pg;
5     FeynDiagram fd(pg);

6     xy pos1(-5,2);
7     text t1(fd,"Simple Text",pos1);

8     text t2(fd,"Large Text",xy(-5,0));
9     t2.fontsize.scale(1.5);


10    vertex_dot v(fd,-5,-2);
```

```
11    text t3(fd,"F^{\\mu\\nu}_{a_1}",v + xy(0,-.5),0.5,1);

12    text t4(fd,"Rotated Helvetica!",xy(3,2),-40);
13    t4.fontname.set("Helvetica");

14    text t5(fd,"Change \\font{Courier}{font}",xy(-5,-6));

15    text t6(fd,"\\tildebar{c} \\overrightarrow{p}",xy(3,-5));

16    pg.output();
17    return 0;
18    }
```



Line 6 defines `pos1` which is used to locate the *text* object defined on line 7. On line 8 another *text* object is created. This time, its position is given by "`xy(-5,0)`" which constructs a temporary object of type *xy*. The *fontsize* parameter is used on line 9 to make the text 1.5 times its normal size. On line 10 a *vertex_dot* is created named `v`. We then position text relative to this object on line 11. The meaning of "`v + xy(0,-.5)`" on this line is the following. `v` is converted to type *xy* which specifies the position of the vertex. Then, this position has a vector of length 0.5, pointing in the -y direction, added to it. This gives the position of the center of the top of the text. On line 12 we create text using an angle of $-40°$ instead of the default. On line 13 the *fontname* parameter is changed so that the "Helvetica" font is used instead of the default font which is "Times-Roman".

Parameters for *text*:

fontname (s) - name of the PostScript font to use
fontsize (d) - size of the font

Some common PostScript fonts:

Times-Roman	AvantGarde-Book	Helvetica	Bookman-Light
Courier	**Helvetica-Bold**	Palatino-Roman	NewCenturySchlbk-Roman

Manipulations with xy

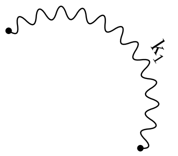
An *xy* is a representation of a two dimensional vector. As you saw in the example for the section titled "Text," it is possible to add objects of type *xy* just as you add vectors. You can also multiply them by scalars (*double*s). The class *xy* has the following member functions:

double	`x()` - return x component
double	`y()` - return y component
double	`len()` - length of the vector
double	`angle()` - polar angle of the vector
xy	`perp()` - return a unit vector which is perpendicular to the *xy*
xy&	`rotate(angle)` - rotate the *xy* by angle and return a reference to itself

There are two unit vectors perpendicular to any (non-null) vector in two dimensions. The `perp()` member function returns the vector which is to the left as we look in the direction of the original vector. So, the cross product: `vect` $\times$ `vect.perp()` is always out of the page. There is an additional function (not a **member** function of *xy*) called `polar(r,theta)` which takes *double*'s for `r` and `theta`, and returns an *xy*. Example:

Program 7

```
1  #include <FeynDiagram/fd.h>

2  main()
3     {
4     page pg;
5     FeynDiagram fd(pg);

6     vertex_dot v1(fd,-3,4), v2(fd,polar(5,-30));
7     line_wiggle L(fd,v1,v2);

8     xy midpt, arcpt;
9     midpt = 0.5 * (v1 + v2);
10    arcpt = midpt + 4.0 * (v2-v1).perp();
11    L.arcthru(arcpt);

12    xy refpt, tanvect;
13    refpt = L.curve->f(0.6);
14    tanvect = L.curve->tang(0.6);
15    refpt += 0.8 * tanvect.perp();
16    text txt(fd,"k_1",refpt,tanvect.angle());

17    pg.output();
18    return 0;
19    }
```



The `polar()` function is used on line 6 to specify the position of the second vertex in polar coordinates. On line 9 we set `midpt` to the midpoint of the two vertices. We then compute the point to draw the arc through on line 10 by adding a vector which is perpendicular to the line connecting the two vertices. On line 13 we use the curve associated with the photon line to find a point which is 60% of the way along the curve. Actually, the argument to `L.curve->f()` is the value of the curve parameter, but for straight lines and arcs this is proportional to the distance along the curve. On line 14 we find the unit tangent vector to the curve at the same point. Our goal is to label the photon line, so we need the location of a point which is slightly displaced from the line. Thus, we add a vector to `refpt` on line 15 which is perpendicular to the tangent to the curve. Note that in lines 8 and 12 we define *xy*'s without giving them values. If we tried to use them in a calculation before giving them values, an error message would be generated.

Coordinate Systems

So far, when we have created a *FeynDiagram*, we have always done so using "`FeynDiagram fd(pg);`". This gives us a Feynman diagram of the default size with the default coordinate system. The default for the x coordinate is to have -10 be one inch from the left side of the page, and +10 be one inch from the right side. The default y coordinate has its zero at the center of the page. We can also define a *FeynDiagram* by doing:

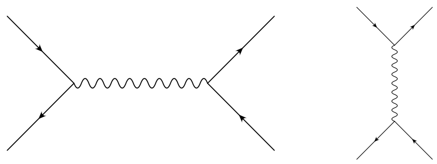
```
FeynDiagram fd(pg,xleft_in,xright_in,yref_in);
```

Where `pg` is a *page*, and `xleft_in`, `xright_in`, and `yref_in` are *double*'s. `xleft_in` and `xright_in` tell where on the page (in inches relative to the left edge) the x coordinate has values -10 and 10 respectively. `yref_in` is the height from the bottom of the page (in inches) where the y coordinate is zero.

Note that **FeynDiagram** does not limit where you can put vertices, lines, etc. The x coordinates -10 and 10 are reference points only. In the example below, we will construct two Feynman diagrams on the same page.

Program 8

```
1  #include <FeynDiagram/fd.h>

2  main()
3     {
4     page pg;
5     xy pt[6];

6     pt[0] = xy(-10,5);
7     pt[1] = xy(-10,-5);
8     pt[2] = -pt[1];
9     pt[3] = -pt[0];

10    pt[4] = xy(-5,0);
11    pt[5] = -pt[4];

12    FeynDiagram fdA(pg,1,4,4);
13    line_plain fA1(fdA,pt[0],pt[4]), fA2(fdA,pt[4],pt[1]);
14    line_plain fA3(fdA,pt[5],pt[2]), fA4(fdA,pt[3],pt[5]);
15    line_wiggle pA(fdA,pt[4],pt[5]);

16    int i;
17    for (i = 0; i < 6; ++i)
18       {
19       pt[i].rotate(90);
20       pt[i] = xy(pt[i].x(),-pt[i].y());
21       }

22    FeynDiagram fdB(pg,4.5,6.2,4);
23    line_plain fB1(fdB,pt[0],pt[4]), fB2(fdB,pt[4],pt[1]);
24    line_plain fB3(fdB,pt[5],pt[2]), fB4(fdB,pt[3],pt[5]);
25    line_wiggle pB(fdB,pt[4],pt[5]);

26    pg.output();
27    return 0;
28    }
```



First, the diagram on the left is drawn. On lines 6 through 9 we give the location of the endpoints of the lines which are stored in an array of *xy*'s. On lines 10 and 11 we set the location of the vertices (although no actual vertex dots are drawn in this diagram). On line 12 we define the *FeynDiagram* called `fdA`. We have chosen the left edge to be at 1 inch, and the right edge at 4 inches. The contents of the diagram are constructed on lines 13 through 15.

To draw the diagram on the right, we rotate all of the points in the first diagram by 90° (line 19), and then flip about the x-axis (line 20). On line 22 we define the second *FeynDiagram*. This one has its coordinates mapped onto a smaller region of the page. Notice that everything in the diagram has been scaled down appropriately. Lines 23 through 25 are identical to 13 through 15 except that the variable names have been changed.

When you want to draw multiple diagrams on a page, it is highly recommended that you use one *FeynDiagram* for each of the diagrams. That way, each has its own coordinate system, so it can be moved or resized by changing only one line in your program. Also, the default settings for the parameters assume that your (single) diagram goes from roughly -10 to +10 in both the x and y directions in the *FeynDiagram*'s coordinate system. So, if you cram several diagrams into this space, they will look funny. For example, the wiggles in the photon lines will look too big (ie. have too few wiggles) because **FeynDiagram** assumed that the lines would typically be longer than they are.

When debugging your Feynman diagram, you may find it useful to have a grid showing your coordinate system with the diagram on it. If your *FeynDiagram* is called `fd`, you can do this by putting `fd.gridon.settrue()` in your program. To turn grids on for all of the diagrams in your program, pick any one diagram, and do `fd.global_gridon.settrue()`, and this will turn the grid on for all of the diagrams (unless you explicitly turn them off). The grid will be just a little larger than your diagram. If you want to change the range of the grid, use `grid_xmin`, `grid_xmax`, `grid_ymin`, and `grid_ymax`. For example, `fd.grid_xmin.set(-5.5)`.

Sometimes, you will want to specify the size of something in terms of the actual size that it will be printed. For example, if you had several diagrams on one page, you might want to set the size of the text labels to be 0.2 inches regardless of the size of the individual diagram. Each *FeynDiagram* has a member function `inch2lencoord(len)` which takes a length in inches, `len`, and returns the corresponding length in terms of the systems of units for the *FeynDiagram*.

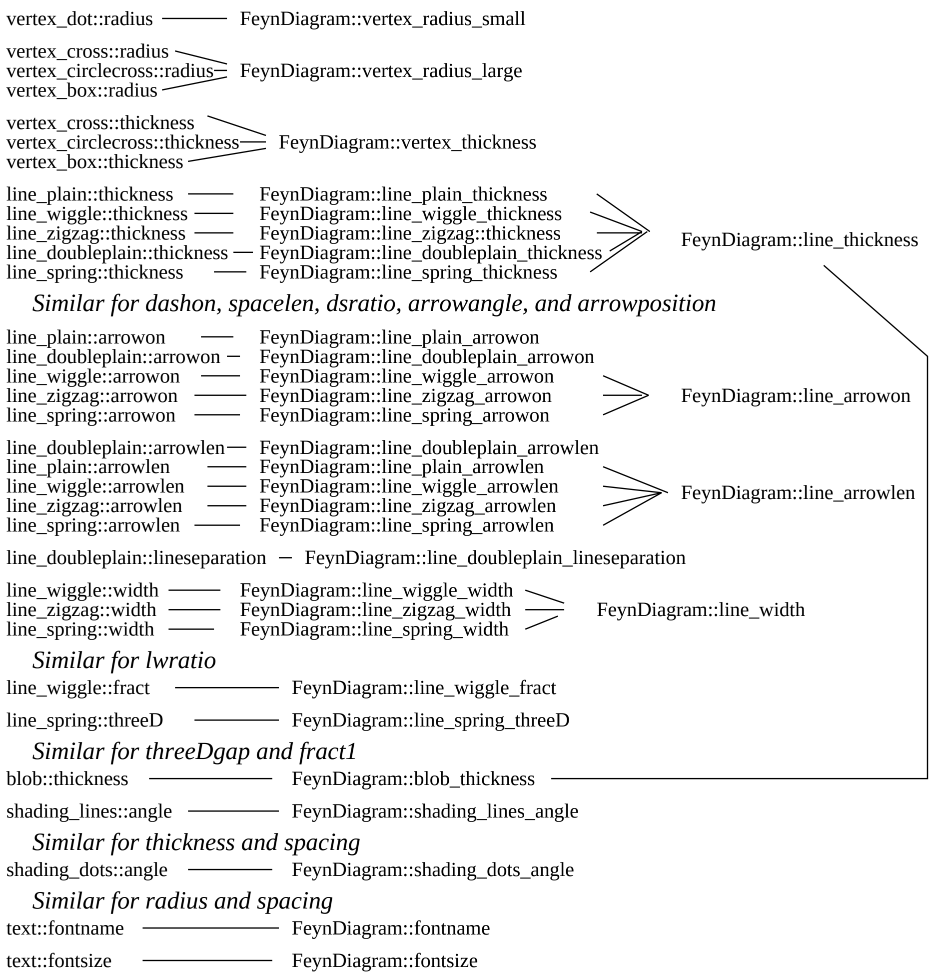
Adjusting Parameters

In previous sections, lists of parameters were given. These parameters have some default value which you can modify. A parameter may have a parent, so if its value isn't set it gets a value from the parent. The letter in parenthesis following the parameter name tells what type the parameter is. (s) means that the parameter is a string. You can use the `set()` member function to change the value of the string. (d) means that the parameter is a *double*, which can be set using `set()` or `scale()`. The `scale()` member function tells the parameter to multiply its parent's value by the argument of `scale()` when it is asked for its value. (b) means the parameter is a boolean. Rather than using `set()`, you use `settrue()` or `setfalse()` to set its value.

When a diagram is drawn, **FeynDiagram** checks the values of the parameters for the vertices, lines, etc. For example, when drawing a *line_wiggle* defined by:

```
line_wiggle ph(fd,startpt,endpt);
```

it checks the *width* parameter `ph.width`. If the *width* parameter has been set, it uses this value. If it hasn't been, it looks to the parent which is `fd.line_wiggle_width`. If this hasn't been set either, it proceeds to `fd.line_width` which is set automatically by **FeynDiagram**. `fd.line_width` controls the default width of all the *line_wiggle*'s, *line_zigzag*'s, and *line_spring*'s in the diagram as we see from the chart below. So, to change the width of just `ph`, use `ph.width.set()` or `ph.width.scale()`. If you want to change the width of all the *line_wiggle*'s in the diagram whose widths aren't explicitly set, use `fd.line_wiggle_width.set()` or `fd.line_wiggle_width.scale()`. To change the width of all *line_wiggle*'s, *line_zigzag*'s, and *line_spring*'s in the diagram (whose widths aren't explicitly set), use `fd.line_width.set()`. Note that `fd.line_width.scale()` cannot be used since `fd.line_width` has no parent. When trying to find the value of a parameter, **FeynDiagram** searches the chart below from left to right until it finds a value that has been set. Parameters and their parents:

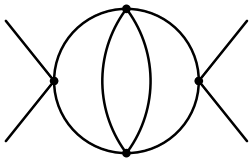


Now lets look at an example where we change the values of the parents of some of the parameters (we have changed parameters for individual lines in previous examples):

Program 9

```
#include <FeynDiagram/fd.h>

main()
   {
   page pg;
   FeynDiagram fd(pg);
   fd.line_plain_arrowon.setfalse();
   fd.line_plain_thickness.scale(3.0);
   fd.vertex_radius_small.set(2*fd.vertex_radius_small.get());

   xy e1(-10,5), e2(-10,-5), e3(10,5), e4(10,-5);
   vertex_dot v1(fd,-6,0), v2(fd,6,0), v3(fd,0,6), v4(fd,0,-6);
   line_plain L1(fd,e1,v1), L2(fd,e2,v1), L3(fd,e3,v2), L4(fd,e4,v2);
   line_plain L5(fd,v1,v2), L6(fd,v2,v1);
   L5.arcthru(v3);
   L6.arcthru(v4);
   line_plain L7(fd,v3,v4), L8(fd,v4,v3);
```

```
16   L7.arcthru(-2,0);
17   L8.arcthru(2,0);

18   pg.output();
19   return 0;
20   }
```



On line 6 we change the default so that all our *line_plain* objects won't have arrows. On line 7 we triple the default thickness for the same objects. On line 8 we change the default radius used by our vertices by a factor of 2. Notice that we had to use `set()` rather than `scale()` since `fd.vertex_radius_small` doesn't have a parent (look at the top line of the chart of parameters and their parents).

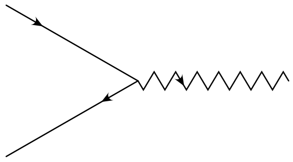
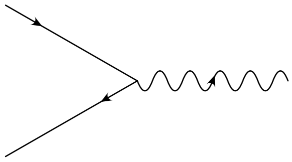
There is another parameter which is a member of *FeynDiagram*. This is *scalefactor*. Its parent is *FeynDiagram::defaultscalefactor*, which is hidden so that you cannot modify it. *scalefactor* is used to scale the default values for all quantities which measure length (eg. *line_thickness*, *line_width*, **not** *line_arrowangle*). You can use `scalefactor.scale()` to shrink or enlarge all such quantities for a *FeynDiagram*. This is useful if you are drawing very complicated diagrams where you might want all features to be somewhat smaller than normal. After changing *FeynDiagram::scalefactor*, you must call the member function `updatedefaults()` for the *FeynDiagram*. This will cause the values of default parameters such as *FeynDiagram::line_width* to be recomputed using the new value of *FeynDiagram::scalefactor*.

You may want to change the default value of some parameters for all of the diagrams in your program. To do this, you give **FeynDiagram** a function which it executes for each *FeynDiagram* which it creates. This function is called after the default values of the parameters are set, so it can modify them. Note that you must tell **FeynDiagram** about this function before a *FeynDiagram* is created for it to work. An example:

Program 10

```
#include <FeynDiagram/fd.h>

void adjust(FeynDiagram &fd, double scalefactor)
   {
   fd.line_arrowon.settrue();
   fd.line_arrowposition.set(.25);
   fd.line_wiggle_arrowposition.set(.5);
   fd.line_wiggle_width.set(.7*scalefactor);
   }

main()
   {
   page pg;
   pg.adjustparams = adjust;
   xy pt1(-7,4), pt2(0,0), pt3(-7,-4), pt4(8,0);
   FeynDiagram fdA(pg,1,4,5.5);
   line_plain L1A(fdA,pt1,pt2), L2A(fdA,pt2,pt3);
   line_zigzag L3A(fdA,pt2,pt4);

   FeynDiagram fdB(pg,4.5,7.5,5.5);
   line_plain L1B(fdB,pt1,pt2), L2B(fdB,pt2,pt3);
   line_wiggle L3B(fdB,pt2,pt4);
```

```
21    pg.output();
22    return 0;
23    }
```

On lines 2-8 we create a function which will adjust the parameters for an individual *FeynDiagram*. The first argument for our function `adjust` is the diagram to adjust. This is passed by reference, so our modifications will remain in tact when the function exits (normally, C passes by value, so your function modifies a copy of the object – the `&` in `&fd` tells it to pass by reference). The second argument, `scalefactor` should multiply any parameter which measures a length, as in line 7 (this argument is produced by the parameter *scalefactor* mentioned earlier). On line 4 we force all lines to have an arrow, and on line 5 we say that the arrow should be positioned 25% of the way along the line. But, on line 6 we override this for all *line_wiggle*'s. This is because *line_arrowposition* is the parent of *line_wiggle_arrowposition*, and when **FeynDiagram** accesses the *arrowposition* for a *line_wiggle* the value is obtained from *line_wiggle_arrowposition* (since it is set) without going to its parent. On line 12 we tell **FeynDiagram** to use our `adjust` function on all of the diagrams (this will affect all diagrams on all pages).

Normally, when you make a *FeynDiagram* fit onto a smaller space on the page, all features are scaled down. But, when **FeynDiagram** determines the default values for the parameters, it has minimum values beyond which it won't shrink things anymore. Typically, during photoreduction the lines in a figure will get thinner by more than the amount you are reducing by. The default parameters have been chosen so that diagrams should be able to survive a 50% reduction. If you want to be able to reduce things even more, you can do `pg.adjustparams = params4reduction`. The function `params4reduction` will make the lines and some other features somewhat thicker. It will also change the default font to a thicker font.

Index