The Planar Enumerator

by Stephen Green

■ 1 Introduction

The Planar Enumerator is designed to enumerate objects that have a planar presentation, such as knots, links, tangles, and 2D surfaces in $\mathbb{R}^4$. It does this by first generating all the planar diagrams of a given complexity (the number of crossings for knots) and then finding out which of them are isomorphic to each other by finding relations between them (Reidemeister moves for knots). The final result is hopefully just one diagram from each isomorphism class.

At this point, however, the program is fairly slow and requires a lot of memory to run for higher complexities. It still accomplishes its task though.

■ 2 Installation

The Planar Enumerator is written primarily in C, but requires *Mathematica* as well. The front end is done in *Mathematica*.

To install, obtain the tar archive "planarenumerator.tar.gz" and extract it to some directory. You then need to put the files "mathlink.h" and "libML.a" from your *Mathematica* installation into this directory. For my setup, they can be found in the directory "/usr/local/Wolfram/Mathematica/4.2/AddOns/MathLink/DeveloperKit/Linux/CompilerAdditions", but that could certainly be different for your installation.

Once you have the files, simply type "make" at the command prompt, and the Planar Enumerator should compile. You may want to edit the Makefile first to optimize it for your system. Also, depending on the speed and memory of your computer, you might want to increase "MAXPDS", found in the file "planarenumerator.h". That variable decides how much memory to allocate the for the planar diagrams. I have it set to 2000000 diagrams by default because my computer can't handle any more.

The following command loads the Planar Enumerator into *Mathematica*, as long as the file "PlanarEnumerator.m" is in your path.

```
<< PlanarEnumerator`
```

```
Planar Enumerator loaded; Stephen Green, NSERC Undergraduate
  Student Research Award 2003 (supervised by Dror Bar-Natan).
```

Minor changes may be needed for your particular system in the Makefile or "PlanarEnumerator.m". I have it configured for Linux, but if you wanted to run it in Windows, for example, you would need to change the name of the C executable from "pe" to "pe.exe".

■ 3 "PD notation"

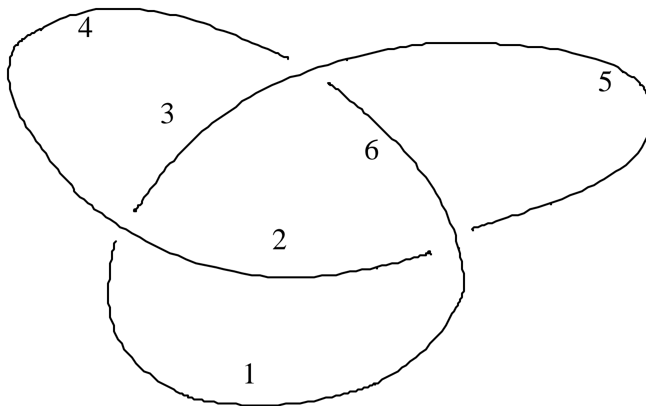
This section explains the notation used by the Planar Enumerator to represent planar diagrams. We will use the example of knots.

The following represents the trefoil:

```
PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 6, 5]];
```

Each X represents one of the vertices. The numbers number the edges. The first vertex X[1, 2, 3, 4] represents the bottom left vertex in the diagram below. For knots, the numbers go counterclockwise starting at an edge that goes 'under' at a crossing.

```
Show[Import["trefoil.eps"]];
```



Of course, you could use any type of vertex you want (except that there is a maximum of 7 characters allowed). But for this example of knots, we are using X's.

For each type of vertex you must specify three additional numbers: the complexity, the degree, and the symmetry.

Complexity

The complexity for a knot vertex is the number of crossings that it represents. So for an X vertex, the complexity is 1. Most vertices will have complexity 1. It's when you come to relations, that complexities can be higher.

Degree

The degree of a vertex is the number of edges that go out from it. It's the same as the valency. For an X vertex the degree is 4.

Symmetry

The symmetry of a vertex is a number that specifies how many times the vertex's list of numbers must be rotated before arriving at an equivalent vertex. The symmetry must divide the degree. For an X vertex, the symmetry is 2. This means that both $X[1, 2, 3, 4]$ and $X[3, 4, 1, 2]$ represent the same vertex. However, $X[2, 3, 4, 1]$ represents a different vertex.

Setting the values

To set the complexity, degree, and symmetry input the following:

```
Complexity[X] = 1; Deg[X] = 4; Symmetry[X] = 2;
```

The 'p' vertex

There is a built-in vertex type called p, for 'point'. It has complexity 0, degree 2, and symmetry 1. It is used to connect together two edges that have different edge numbers, and the Planar Enumerator will often eliminate it from your planar diagram and renumber the edges.

■ 4 Canonical Form

The function CanonicalForm[] takes a planar diagram in 'PD notation' as input and it returns a canonical form of that planar diagram, where the edges have been renumbered and the vertices have been reordered. The purpose is to be able to tell if two PDs are actually the same or not. Here are some examples:

```
CanonicalForm[PD[X[1, 4, 2, 5], X[3, 6, 4, 1], X[5, 2, 6, 3]]]
```

```
PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 6, 5]]
```

```
CanonicalForm[PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 6, 5]]]
```

```
PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 6, 5]]
```

```
CanonicalForm[PD[X[1, 5, 6, 2], X[6, 5, 4, 3], X[1, 2, 3, 4]]]
```

```
PD[X[1, 2, 3, 4], X[1, 5, 6, 2], X[4, 3, 6, 5]]
```

```
CanonicalForm[PD[X[1, 2, 3, 4], X[1, 6, 7, 2], X[7, 6, 5, 8], X[3, 4, 5, 8]]]

PD[X[1, 2, 3, 4], X[1, 5, 6, 2], X[3, 4, 7, 8], X[6, 5, 7, 8]]
```

Notice that CanonicalForm[] removes p vertices from PDs:

```
CanonicalForm[PD[X[1, 1, 2, 3], p[2, 3]]]

PD[X[1, 1, 2, 2]]
```

Lets add Y vertices to our types:

```
Complexity[Y] = 1; Deg[Y] = 3; Symmetry[Y] = 1;

CanonicalForm[PD[Y[1, 4, 100], Y[4, 2, 10], Y[115, 10, 15],
  Y[15, 14, 12], Y[1, 22, 3], Y[17, 12, 22], Y[100, 115, 17], Y[3, 14, 2]]]

PD[Y[1, 2, 3], Y[1, 4, 5], Y[2, 6, 7], Y[3, 8, 9],
 Y[4, 9, 10], Y[5, 11, 6], Y[7, 12, 8], Y[10, 12, 11]]

CanonicalForm[PD[Y[1, 2, 3], Y[1, 4, 5], Y[2, 6, 7],
  Y[3, 8, 9], Y[4, 9, 10], Y[5, 11, 6], Y[7, 12, 8], Y[10, 12, 11]]]

PD[Y[1, 2, 3], Y[1, 4, 5], Y[2, 6, 7], Y[3, 8, 9],
 Y[4, 9, 10], Y[5, 11, 6], Y[7, 12, 8], Y[10, 12, 11]]
```

The previous PD was a cube.

```
CanonicalForm[PD[Y[3, 5, 1], Y[2, 4, 3], X[5, 4, 2, 1]]]

PD[Y[1, 2, 3], X[1, 4, 5, 2], Y[3, 5, 4]]
```

■ 5 Diagrams

The function Diagrams[complexity, vertices] produces all the non-isomorphic planar diagrams of a given total complexity and with vertices of a given type(s). They are all returned in canonical form.

```
Diagrams[2, 2 X]

{PD[X[1, 2, 3, 4], X[1, 4, 3, 2]], PD[X[1, 2, 3, 4], X[2, 1, 4, 3]],
 PD[X[1, 1, 2, 3], X[2, 3, 4, 4]], PD[X[1, 1, 2, 3], X[3, 4, 4, 2]],
 PD[X[1, 2, 3, 1], X[3, 4, 4, 2]], PD[X[1, 1, 2, 3], X[2, 4, 4, 3]],
 PD[X[1, 1, 2, 3], X[3, 2, 4, 4]], PD[X[1, 2, 3, 1], X[2, 4, 4, 3]]}
```

Instead of specifying $2X$, you can write 'Several X', to allow zero or more X's, such that the total complexity is still the first argument to Diagrams[].

```
Diagrams[2, Several X]
```

```
{PD[X[1, 2, 3, 4], X[1, 4, 3, 2]], PD[X[1, 2, 3, 4], X[2, 1, 4, 3]],
 PD[X[1, 1, 2, 3], X[2, 3, 4, 4]], PD[X[1, 1, 2, 3], X[3, 4, 4, 2]],
 PD[X[1, 2, 3, 1], X[3, 4, 4, 2]], PD[X[1, 1, 2, 3], X[2, 4, 4, 3]],
 PD[X[1, 1, 2, 3], X[3, 2, 4, 4]], PD[X[1, 2, 3, 1], X[2, 4, 4, 3]]}
```

'Several' is more useful when you want to combine 2 or more types of vertices, with some total complexity.

```
Diagrams[2, Several X + Several Y]
```

```
{PD[Y[1, 2, 3], Y[1, 3, 2]], PD[Y[1, 1, 2], Y[2, 3, 3]],
 PD[X[1, 2, 3, 4], X[1, 4, 3, 2]], PD[X[1, 2, 3, 4], X[2, 1, 4, 3]],
 PD[X[1, 1, 2, 3], X[2, 3, 4, 4]], PD[X[1, 1, 2, 3], X[3, 4, 4, 2]],
 PD[X[1, 2, 3, 1], X[3, 4, 4, 2]], PD[X[1, 1, 2, 3], X[2, 4, 4, 3]],
 PD[X[1, 1, 2, 3], X[3, 2, 4, 4]], PD[X[1, 2, 3, 1], X[2, 4, 4, 3]]}
```

Loops

There is the option to turn off loops for any vertex type. Just set Loops[X] 0 to turn them off, and Loops[X] 1 to turn them back on. By default they are on. Turning them off saves a lot of time and greatly reduces the number of PDs produced.

```
Loops[X] = 0;
Diagrams[2, 2 X]
```

```
{PD[X[1, 2, 3, 4], X[1, 4, 3, 2]], PD[X[1, 2, 3, 4], X[2, 1, 4, 3]]}
```

```
Length[Diagrams[5, 5 X]]
```

```
100
```

```
Loops[X] = 1;
Length[Diagrams[5, 5 X]]
```

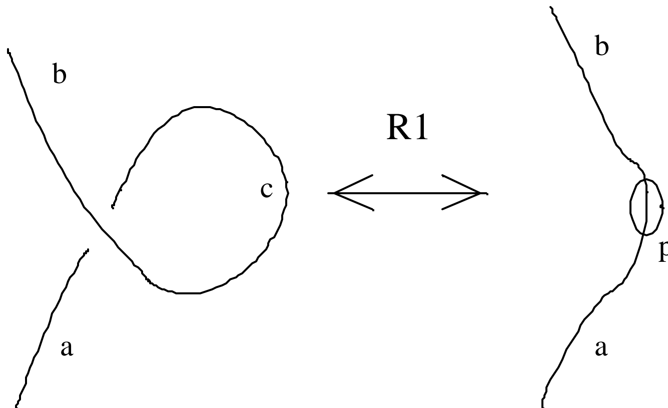
```
4740
```

■ 6 Relations

Relations are the moves that are allowed to be performed on parts of planar diagrams, and give you planar diagrams that are different from what you started with, but that represent the same object. For example, with knots and links, the relations are the Reidemeister moves: R1, R2, and R3.

Continuing with the knots and links example, here's how to specify R1, R2, and R3.

```
Show[Import["R1.eps"]];
```



We need to specify both the left hand side and the right hand side of this relation.

```
LeftHS[R1[a_, b_], c_] := X[a, c, c, b];
LeftHS[R1] = 1;
RightHS[R1[a_, b_]] := p[a, b];
RightHS[R1] = 0;
```

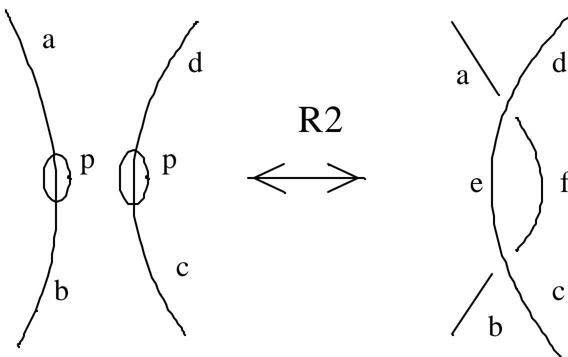
Notice that in the first line, we have LeftHS[R1[a_, b_], c_]. The 'c' is so that the program can give a number to the new edge. The second and fourth lines of the statement specify how many new edges each side has.

For each relation, you must also specify the complexity, degree, and symmetry as with ordinary vertices. The complexity should be set to the complexity of the higher complexity side.

```
Complexity[R1] = 1; Deg[R1] = 2; Symmetry[R1] = 2;
```

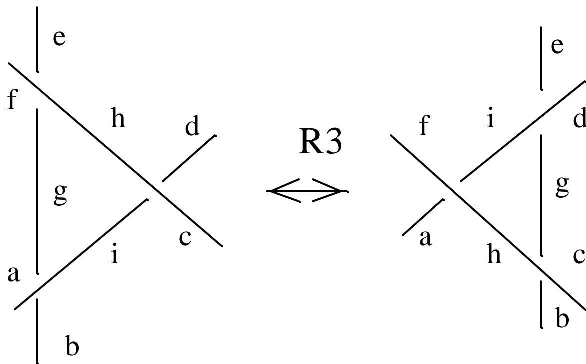
Here's the other two Reidemeister moves along with their descriptions to *Mathematica*. Notice that for sides consisting of more than one vertex, we enclose multiple vertices in a 'PD'.

```
Show[Import["R2.eps"]];
```



```
LeftHS[R2[a_, b_, c_, d_]] := PD[p[a, b], p[c, d]];
LeftHS[R2] = 0;
RightHS[R2[a_, b_, c_, d_], e_, f_] := PD[X[a, e, f, d], X[b, c, f, e]];
RightHS[R2] = 2;
Complexity[R2] = 2; Deg[R2] = 4; Symmetry[R2] = 4;
```

```
Show[Import["R3.eps"]];
```




```
LeftHS[R3[a_, b_, c_, d_, e_, f_], g_, h_, i_] :=
  PD[X[b, i, g, a], X[i, c, d, h], X[g, h, e, f]];
LeftHS[R3] = 3;
RightHS[R3[a_, b_, c_, d_, e_, f_], g_, h_, i_] :=
  PD[X[a, h, i, f], X[b, c, g, h], X[g, d, e, i]];
RightHS[R3] = 3;
Complexity[R3] = 3; Deg[R3] = 6; Symmetry[R3] = 6;
```

Mirror Images

Note that some relations have mirror images. For knots, R1 and R3 both have them. It can be advantageous to use these because they can improve the results from the Enumerate[] function discussed in the next section.

We shall call these mirror images 'R1P' and 'R3P':

```
LeftHS[R1P[a_, b_], c_] := X[a, b, c, c];
LeftHS[R1P] = 1;
RightHS[R1P[a_, b_]] := p[a, b];
RightHS[R1P] = 0;
Complexity[R1P] = 1; Deg[R1P] = 2; Symmetry[R1P] = 2;
LeftHS[R3P[a_, b_, c_, d_, e_, f_], g_, h_, i_] :=
 PD[X[b, i, g, a], X[c, d, h, i], X[g, h, e, f]];
LeftHS[R3P] = 3;
RightHS[R3P[a_, b_, c_, d_, e_, f_], g_, h_, i_] :=
  PD[X[h, i, f, a], X[b, c, g, h], X[g, d, e, i]];
RightHS[R3P] = 3;
Complexity[R3P] = 3; Deg[R3P] = 6; Symmetry[R3P] = 6;
```

■ 7 Enumerate

Enumerate[complexity, complexitybound, vertices, relations] will find all of the PDs made up of 'vertices' with complexity 'complexity' that cannot be reduced to a lower complexity by any of the 'relations'. Also, if more than one of the PDs can be linked through relations (going up to PDs with complexity 'complexitybound') to other PDs, then only one PD from the list of linked PDs is returned.

Essentially, Enumerate[] attempts to find all the PDs that represent objects that are not isomorphic to eachother. By specifying a higher 'complexitybound', results will be improved, but more computer resources will be required.

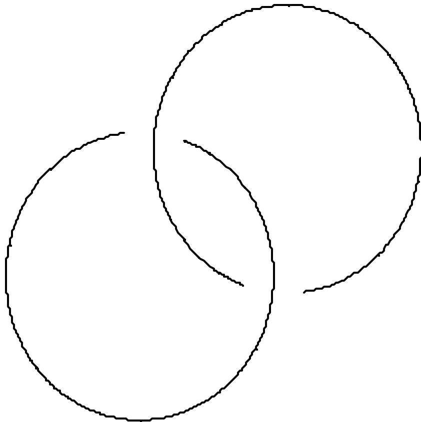
```
Enumerate[2, 2, Several X, {R1, R1P, R2, R3, R3P}]
```

```
Generating PDs

Setting up lists in C

Finding Eliminateable PDs

R1 + Several X: complexity = 2

R1P + Several X: complexity = 2

R2 + Several X: complexity = 2

Finding Relations

{PD[X[1, 2, 3, 4], X[2, 1, 4, 3]]}
```

```
<< KnotTheory`
```

```
Loading KnotTheory`...
```

```
Show[DrawPD[PD[X[1, 2, 3, 4], X[2, 1, 4, 3]]]];
```



The result given here is the Hopf Link, which is the only complexity 2 knot or link. In this case, 'complexitybound' did not need to be increased.

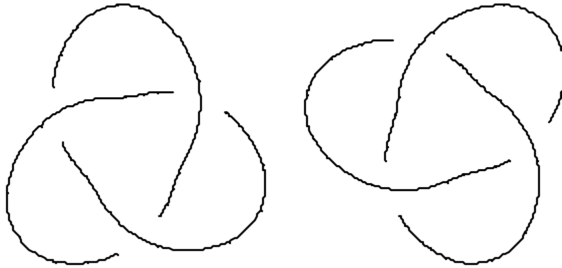
Note that the relations must be enclosed in a List for the program to work.

Enumerate[] also prints out the status of what task it is performing, while the C program prints to a stderr window information such as how many PDs are produced.

We drew the result using Emily Redelmeier's program at http://www.math.toronto.edu/~drorbn/KAtlas/Manual/DrawPD.html.

```
Enumerate[3, 3, Several X, {R1, R1P, R2, R3, R3P}]

Generating PDs

Setting up lists in C

Finding Eliminateable PDs

R1 + Several X: complexity = 3

R1P + Several X: complexity = 3

R2 + Several X: complexity = 3

Finding Relations

R3 + Several X: complexity = 3

R3P + Several X: complexity = 3

{PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 6, 5]], PD[X[2, 3, 4, 1], X[1, 5, 6, 2], X[3, 6, 5, 4]]}
```

```
Show[GraphicsArray[DrawPD /@ {PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 6, 5]],
     PD[X[2, 3, 4, 1], X[1, 5, 6, 2], X[3, 6, 5, 4]]}]];
```



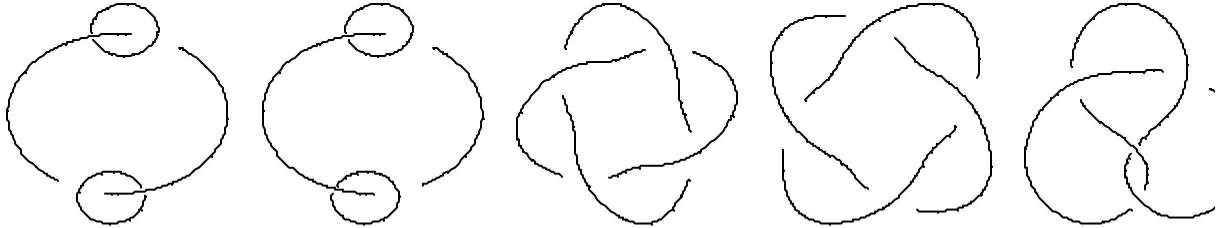
With complexity 3, we get the two mirror images of the trefoil.

Here we see the importance of using 'Several' instead of specific numbers in describing the vertices for the Enumerate[] function. As you can see from the printout, it generated PDs with 'R1 + Several X' vertices and 'R3 + Several X' vertices, both with complexity 3. R1 and R3 have different complexities themselves, so really 'R1 + Several X' was like calling 'R1 + 2X', and 'R3 + Several X' was like calling 'R3'. So if a specific number was put in place of 'Several', the program wouldn't work.

```
Enumerate[4, 4, Several X, {R1, R1P, R2, R3, R3P}]
```

```
Generating PDs

Setting up lists in C

Finding Eliminateable PDs

R1 + Several X: complexity = 4

R1P + Several X: complexity = 4

R2 + Several X: complexity = 4

Finding Relations

R3 + Several X: complexity = 4

R3P + Several X: complexity = 4

{PD[X[1, 2, 3, 4], X[2, 1, 4, 5], X[6, 7, 8, 3], X[5, 8, 7, 6]],
 PD[X[1, 2, 3, 4], X[2, 1, 4, 5], X[3, 6, 7, 8], X[6, 5, 8, 7]],
 PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 7, 8], X[6, 5, 8, 7]],
 PD[X[2, 3, 4, 1], X[1, 5, 6, 2], X[3, 7, 8, 4], X[5, 8, 7, 6]],
 PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[6, 7, 8, 3], X[4, 8, 7, 5]]}
```

```
Show[GraphicsArray[DrawPD /@ {PD[X[1, 2, 3, 4], X[2, 1, 4, 5], X[6, 7, 8, 3], X[5, 8, 7, 6]],
     PD[X[1, 2, 3, 4], X[2, 1, 4, 5], X[3, 6, 7, 8], X[6, 5, 8, 7]],
     PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 7, 8], X[6, 5, 8, 7]],
     PD[X[2, 3, 4, 1], X[1, 5, 6, 2], X[3, 7, 8, 4], X[5, 8, 7, 6]],
     PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[6, 7, 8, 3], X[4, 8, 7, 5]]}]];
```

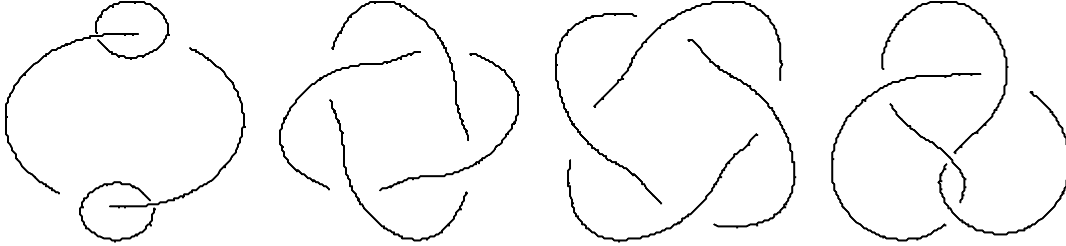


In this case, the first 2 links represent the same object. This can be fixed by increasing the 'complexitybound' to 5.

```
Enumerate[4, 5, Several X, {R1, R1P, R2, R3, R3P}]
```

```
Generating PDs

Setting up lists in C

Finding Eliminateable PDs

R1 + Several X: complexity = 4

R1P + Several X: complexity = 4

R2 + Several X: complexity = 4

R2 + Several X: complexity = 5

Finding Relations

R1 + Several X: complexity = 5

R1P + Several X: complexity = 5

R3 + Several X: complexity = 4

R3 + Several X: complexity = 5

R3P + Several X: complexity = 4

R3P + Several X: complexity = 5

{PD[X[1, 2, 3, 4], X[2, 1, 4, 5], X[6, 7, 8, 3], X[5, 8, 7, 6]],
 PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 7, 8], X[6, 5, 8, 7]],
 PD[X[2, 3, 4, 1], X[1, 5, 6, 2], X[3, 7, 8, 4], X[5, 8, 7, 6]],
 PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[6, 7, 8, 3], X[4, 8, 7, 5]]}
```

```
Show[GraphicsArray[DrawPD /@ {PD[X[1, 2, 3, 4], X[2, 1, 4, 5], X[6, 7, 8, 3], X[5, 8, 7, 6]],
    PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[4, 3, 7, 8], X[6, 5, 8, 7]],
    PD[X[2, 3, 4, 1], X[1, 5, 6, 2], X[3, 7, 8, 4], X[5, 8, 7, 6]],
    PD[X[1, 2, 3, 4], X[2, 1, 5, 6], X[6, 7, 8, 3], X[4, 8, 7, 5]]}]];
```



Now, all these 4 results represent different knots or links. However, it took much longer to run.

Loops

Something that might be worth looking into is turning off loops before calling Enumerate[]. This would mean that not as many PDs would be produced at each level, saving time. However, not as many relations would be found between PDs, requiring a higher 'complexitybound' to achieve the same results.

■ 8 How The Planar Enumerator Works

In this section, a rough outline of how the program works is given. For more details, consult the comments in the code.

Diagrams

The Diagrams[] function is written mainly in C, but the C function is called by a *Mathematica* function that formats the input a bit. Most of the code for the Diagrams[] function is found in the file "PDgenerator.c".

First, all of the adjacency matrices that could represent diagrams with some degree sequence are produced. In each entry of an adjacency matrix we find the number of edges between the vertex represented by the column and the vertex represented by the row. For example for the Hopf Link, we would get the adjacency matrix $\begin{pmatrix} 0 & 4 \\ 4 & 0 \end{pmatrix}$, since each vertex makes 4 connections with the other vertex. If you swap row i with row j and column i with column j in an adjacency matrix, you obtain a new adjacency matrix which represents the same object. A few checks are done to try to reduce the number of isomorphic adjacency matrices produced, but still, many isomorphic ones are produced at the higher complexities. Finding a canonical form for adjacency matrices would speed up the program greatly.

Turning loops on or off is implented as the adjacency matrices are generated, by allowing or disallowing non-zero entries along the diagonals of the matrices.

Next, from each adjacency matrix, we produce all the diagrams (planar and non-planar) that the matrix could represent. At this stage, we treat all vertices of the same degree as equivalent, and assign each vertex symmetry 1. We then filter out the non-planar ones by using Euler's Formula which says that:

(number of faces)   (number of edges)   (number of vertices)   2

The planar ones are put into canonical form, and the duplicates removed. These planar diagrams are called 'basic PDs' since all vertices of the same degree are equivalent, and symmetry is always 1.

For each basic PD, we then assign types to each of its vertices and rotate their edge lists according to their symmetry in order to obtain all the PDs it could represent. Each PD is put into canonical form and duplicates are again removed. When this is done for every basic PD, we have the final result.

With this technique, we generate all the planar and non-planar graphs, even though we never use the non-planar graphs. Since the non-planar graphs greatly outnumber the planar ones, it would be worthwhile to find a new method that somehow only finds the planar graphs.

Enumerate

Enumerate[] is also primarily implemented in C, but the C function makes calls to *Mathematica* through *MathLink* to access the RightHS[] and LeftHS[] functions that the user must define for the relations. Also, the *Mathematica* function Enumerate[] calls the C functions to get things started. Most of the C code is found in the file "PDhashtable.c". Also, this makes use of a hash table available at http://burtleburtle.net/bob/hash/hashtab.html.

First, all of the PDs with the desired vertices with complexities ranging from 'complexity' to 'complexitybound' are produced. Each one is then put into a hash table as the key, with an associated value which consists of a number. These numbers are just assigned sequentially starting with 1.

Then the program generates two int arrays, each of the same length as the total number of PDs already produced. One of them, called 'eliminateablePDs' contains all zeros. The other, called 'relations', contains an int in each element which is equal to the element number.

The program then finds all the PDs containing a relation which have one side with complexity within the range we're looking at and the other side with complexity below 'complexity'. The upper side of each of these can then be reduced by a single move to a PD of complexity less than we want. We then change the value of the element in 'eliminateablePDs' corresponding to the upper complexity side to 1 from 0.

We then find all the PDs containing a relation which have both sides within the range we're looking at. We take the LeftHS and RightHS of the relation to get the two PDs, which we put into canonical form. Then using the hash table we find their indices. To finish off, we change the value of the element of 'relations' which corresponds to the upper complexity PD to the index of the lower complexity PD. Also, if the element of 'eliminateablePDs' corresponding to the upper complexity side is 1, then we change the value of 'eliminateablePDs' corresponding to the lower complexity side to 1 as well. If the value of the upper complexity element in 'relations' is something other than the element number, then it already points somewhere else so we do some more changes.

Finally, we return all of the PDs of complexity 'complexity' that have 'eliminateablePDs' set to 0 and that have 'relations' point to themselves.

■ 9 Future Improvements

At this point, the program is too slow and requires too much memory. The number 7 seems to be the limit for the complexity with knots (loops turned on) on my machine (Athlon 1.2 GHz with 512 MB of RAM) due to the memory usage. The number of PDs produced increases very rapidly as the complexity increases, so storing them all becomes a problem. Solutions to the memory problem could include using some sort of disk-based database instead of that hash table stored in memory, and finding a more compact way of storing the PDs.

Although the memory is the first bottleneck I encounter, if this can be fixed, speed will become an issue. In the generation of PDs, a large improvement could be found if we either found a canonical form for the adjacency matrices, or found a way to generate only the planar graphs, as discussed above. In the enumeration part of the program, a lot of speed was gained by moving away from *Mathematica* into C (the *Mathematica* Dispatch tables were way to slow), but we still must make many calls through *MathLink* in order to access the LeftHS and RightHS functions defined by the user. If a method was found to move these functions into C and still make them easy for the user to define, then things would be greatly sped up. However, the first priority when it comes to speed is probably improving the planar diagram generation.

A new feature that would be fairly easy to implement is an option in the Enumerate function to apply a filter to all the PDs as they are generated, before they are put into the hash table. This could be used to filter out knots from all the diagrams with X vertices. Later on, functions to "relax" diagrams and tell if two diagrams represent the same object or not, with a list of moves to show how they are the same would be useful.