# ChordDiagram Programs

A set of generally useful chord diagram tools was created as part of this study, and is available for use by others. (not yet uploaded to any open source platform)

While the .py version of these programs is documented below, compiled versions can be easily created using any of the compiled python tools. “nuitka” was used to for tests used toward this dissertation.

## Program: gen\_diagrams.py

The gen\_diagrams tool generates all possible diagrams with the specified number of chords. Various filtering options are provided to remove diagrams that contain one, two, or three Reidemeister moves, chords of a specific length, or diagrams that are rotationally equivalent to an already generated diagram.

Usage:

gen\_diagrams.py –n <n-chords> [-s <n\_diagrams>] [-1] [-2] [-3] [–l <chord\_length>] [-r] [-k]

Options:

-n <n-chords>: (Required) All diagrams with the specified number of chords will be generated.

-s <n-diagrams>: (optional, default=100,000) When n-diagrams is greater than 0, a status message is written to stderr.

Note 1: Generated diagrams that pass all filtering are written to stdout. This allows output data to be redirected to a file, while still monitoring generation progress.

Note 2: The status count expresses the number of generated diagrams, not the number of filtered diagrams written to stdout.

-1: Do not generate diagrams containing Reidemeister one moves. Note: As an optimization, chords with adjacent nodes are removed from the list of all possible chords prior to diagram generation. This option reduces the total number of generated diagrams, it does not merely filter diagrams, but prevents generation of non-qualifying diagrams.

-2: Do not print diagrams containing Reidemeister two moves

-3: Do not print diagrams containing Reidemeister three moves

-l <chord-length>: Do not generate diagrams containing chords with the specified length. Note: As an optimization, chords with the specified length are removed from the list of all possible chords prior to diagram generation. This option reduces the total number of generated diagrams. It does not merely filter diagrams, but prevents generation of non-qualifying diagrams.

-r: This option prevents the printing of diagrams that are rotationally equivalent to previously printed diagrams. An in-memory dictionary is maintained of all generated diagrams. This can create extreme memory pressure depending on the size diagram requested and the memory available on the host.

-k: Specifies that diagram output is in “key” form, a more compressed representation that can be read by other ChordDiagram programs, but requires less storage space.

## find\_planarable.py

find\_planarable.py reads diagrams from stdin and writes only the planarable diagrams to stdout.

Usage:

find\_planarable.py [-c <n-cpus>] [-s <n-diagrams>] [-k]

-c <n-cpus>: (optional, default=all CPUs or cores) Determining planarability of a diagram is a CPU intensive operation. find\_planarable.py manages a process pool to allow multi-cpu/multi-core systems to process multiple diagrams concurrently. On most systems the default value provides optimal performance. The number of CPUs chosen is displayed to stderr at startup.

-s <n-diagrams>: (optional, default=100,000) When n-diagrams is greater than 0, a status message is written to stderr.

-k: Specifies that diagram output is in “key” form, a more compressed representation that can be read by other ChordDiagram programs, but requires less storage space.

Examples:

1. gen\_diagrams.py –n 8 -1 -2 -3 –s 0 | find\_planarable.py –s 100 >out.diags

This unix pipeline generates all 8-chord diagrams with no one, two, or three moves and checks them for planarability, writing all planarable diagrams to a file. A status outline is written to stderr every 100 diagrams checked.

2. gen\_diagrams.py –n 8 –k > 8\_chord.diags

Write all 8-chord diagrams to a file in the space-optimized key format for later use by other programs.

## show\_invariant\_chords.py

TODO: full writeup

Displays invariant chords for each input diagram

## filter\_moves.py

TODO: full writeup

1. Displays each diagram with info on whether it contains 1, 2, or 3 moves

Or

1. Filters input diagrams, only displaying those without the specified moves

## cat\_diagrams.py

TODO: Full writeup

Converts diagrams between the space-optimized key format and human-readable string format.

# The ChordDiagram Library

The ChordDiagram python package, as its name implies, deals with chord diagrams, which are known internally, simply as diagrams.

## Python Classes

### Chord

A Chord represents an intersection on the planar version of the chord diagram. Each chord is a pair of nodes. Internally, a chord is represented as a 2-element python list, where the lowest numbered node is always the 0th element in the list.

A chord’s length is the absolute value of the different between the two nodes. The term “complement” or “node complement” is used to mean the node at the other end of the chord.

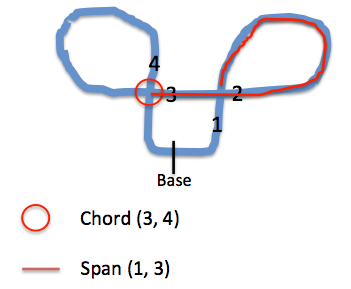
### Diagram

A diagram is a list of chords, maintained in sorted order by the low-node of each chord pair. The diagram size is the number of chords in the diagram.

Typically, the nodes in a diagram are consecutive integers from 1 to twice the diagram’s size. When extending the diagram with additional chords, it’s convenient to scale the diagram’s node values to allow for additional chords to be added between existing chords. This implementation was chosen so that the original diagram plus any chord extensions can be printed separately and still make sense. The other implementation considered was to insert new nodes only where needed, but this makes it difficult to compare the original diagram with and without any extension chords. Convenience methods are available to scale and compress the diagram.

## Span

Spans represent one or more segments of a planar diagram. Consider the diagram [(1,4), (2, 3)]:



Chords and Spans are both stored as a node pair, but while a chord is an intersection, a span is a length along the diagram. (Nodes are not classes or objects, just integers. Is more of a description of a node needed?)

## Algorithms

### Generating Chords

Programs using the ChordDiagram library may wish to test all possible diagrams searching for diagrams with certain characteristics. The first step in generating all possible diagrams is to generate the set of possible chords. This list is created as all combinations of the set of nodes for the diagram, taken two at a time.

The generated chords are stored in a list-of-lists structure that helps ensure uniqueness when generating diagrams:

Example: Chords for a 3-chord diagram

[ [(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)],

[ (2, 3), (2, 4), (2, 5), (2, 6)],

[ (3, 4), (3, 5), (3, 6)],

[ (4, 5), (4, 6)],

[ (5, 6)] ]

### Generating Diagrams

Basically, the algorithm accumulates unique diagrams by selecting one chord from each level in the possible-chords structure above, unless a node in the chord already exists in the set of accumulated chords.

More specifically, it’s a recursive algorithm that processes each level in the diagram as follows:

1. Consider each chord in the current level, one at a time.

2. Skip the chord if either node is already contained in the accumulated diagram.

3. If no chords qualify in the current level, start at step 1 on the next deeper level using the current accumulated diagram.

4. if a qualifying chord is found:

5. If the current accumulated diagram plus the new chord form a complete diagram then save that diagram and move back up a level.

6. Start again at step 1 against the next deeper level, but holding the current accumulated diagram plus the new chord.

Stepping through the process: Accumulated Diagram

1st level: Hold (1, 2), move down a level (1, 2)

2nd level: No unique chords, skip to next level down

3rd level: Hold (3, 4), move down a level (1, 2), (3, 4)

4th level: No chords with unused nodes. Skip down a level

5th level: Hold (5, 6). Diagram complete - save it. **(1, 2), (3, 4), (5, 6)**

No deeper levels. Exhausted chords at this level. Move up a level

4th level: Exhausted chords at this level, Move up a level

3rd level: Hold next qualifying chord: (3, 5), move down (1, 2), (3, 5)

4th level: Hold (4, 6). Diagram complete – save it. **(1, 2), (3, 5) (4, 6)**

Move up a level.

3rd level: Hold (3, 6). Move down a level. (1, 2), (3, 6)

4th level: Hold (4, 6). Complete diagram – save it. **(1, 2), (3, 6), (4, 5)**

Move up a level.

3rd level: Exhausted all chords, move up a level

2nd level: No qualifying chords, move up a level

1st level: Hold (1, 3) (1, 3)

Continue in this way until all chords in the 1st level have been processed against all deeper levels.

#### Calculating the Number of Possible n-chord Diagrams

This formula becomes apparent considering all combinations of chords from the possible chords structure in section (INSERT SECTION NUMBER/name/link). For example, the possible chords for a 3-chord diagram:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Count of number of choices possible from each chord row when creating selected diagrams | | | |
| Available chords | (1,2)  (3,4)  (5,6) | (1,2)  (3,5)  (4,6) | (1,2)  (3,6)  (4,5) | (1,3)  (2,4)  (5,6) |
| (1, 2)(1, 3), (1, 4), (1, 5) (1, 6) | 5 | 5 | 5 | 5 |
| (2, 3), (2, 4), (2, 5), (2, 6) | - | - | - | 3 |
| (3, 4), (3, 5), (3, 6) | 3 | 3 | 3 | - |
| (4, 5), (4, 6) | - | 1 | 1 | - |
| (5, 6) | 1 | - | - | 1 |

In each case, before any chords have been selected from the top row, 5 chords are available for selection from the top row. Each time a chord is selected from a row, it narrows the available options in lower rows in two ways:

Left node invalidation: The selected chord contains a value found in each chord of a lower row, invalidating the entire row.

Right node invalidation: The selected chord contains a value found in each of the remaining valid lower rows.

Thus, when choosing the first chord 5 options are always available. When choosing the second chord, 3 options are always available, and when choosing the final chord, only a single chord is ever available. Hence, the total number of possible 3-chord diagrams is 5 \* 3 \*1, or the product of the first 3 odd positive integers.

It turns out that the number of unique n-chord diagrams is:

1. The product of the first n odd positive integers, or
2. (2i+1) as i=1..n, or
3. (2n)!/2nn!

### Checking Diagrams for Planarability

The algorithm for determining planarability is very much like the manual process of drawing the planar diagram given a chord diagram, except that the algorithm is unaware of direction or relative position.

The diagram is aware of regions, and their boundaries with other regions. Initially, the base point is in the “outside” region, internally named “@”. Every span added to create a new intersection, also creates a new region. Additional regions are named “A”...”Z”. This introduces an artificial limit of 26 regions, which has been sufficient so far. This limit can be quite easily extended, if or when needed.

#### BoundaryTracker Class

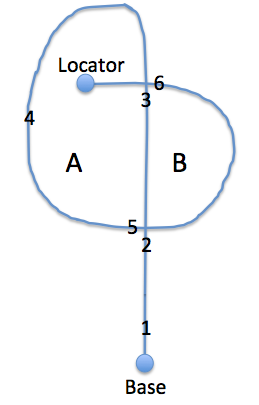
Internally, a BoundaryTracker class is used to manage information about regions and boundaries. For each boundary, it maintains a list of spans sharing that boundary.

Boundaries are named simply in the format: <region1>\_<region2>. For example, the boundary between regions A and C is named A\_C. To ensure a canonical name, region names are listed in ascending lexical order; therefore, C\_A is an invalid boundary name.

Note that @ (outside) sorts before A, so a boundary between region A and Outside is labeled “@\_A”. The specifics of boundary naming are an internal-only construct, unnecessary to understand unless one is working in the code. This notation will be used in examples though to demonstrate progress while drawing actual diagrams.

In addition to the boundary name and list of spans, the BoundaryTracker also tracks the node for the last intersection formed (high end of the chord), the list of nodes that currently participate in an intersection, the region containing the starting BASE point, and the region containing the nodes that have not yet been considered.

Example 1:



The BoundaryTracker for the diagram above is:

Base region: @ (outside)

Locator region: A

Last intersected node: 6

Intersections: 2, 3, 5, 6

Boundaries:

@\_A: (3, 5)

@\_B: (5, 6)

A\_B: (2, 3)

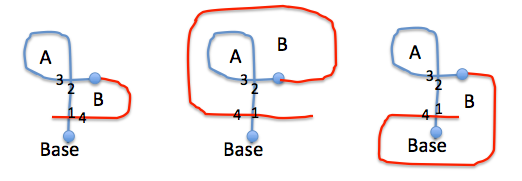
#### The Process

At a very high level the process is:

1. Add the next span that creates an intersection. This will also create a new region.
2. Decide the boundaries of the new region.

When drawing a span manually on paper, up to three possible positions can be drawn. For example, consider adding the (3, 4) span that forms the (1, 4) intersection / chord. The code is only aware that that the span (3, 4) has been added. The difference between the solutions below is in how the boundaries are decided. Unfortunately, no single choice is always right. Therefore, the code effectively tries each solution, continues adding spans, and if that solution does not lead to a planarable diagram, the code backtracks to then try the next option.

Example 2:



|  |  |  |
| --- | --- | --- |
| Base region: @  Locator region: @  Last intersected node: 4  Intersections: 1, 2, 3, 4  Boundaries:  @\_A: (2, 3)  @\_B: (1, 2), (3, 4) | Base region: @  Locator region: @  Last intersected node: 4  Intersections: 1, 2, 3, 4  Boundaries:  @\_B: (1, 2), (3, 4)  A\_B: (2, 3) | Base region: B  Locator region: B  Last intersected node: 4  Intersections: 1, 2, 3, 4  Boundaries:  @\_A: (2, 3)  @\_B: (1, 2), (3, 4) |

Notice that the first and third solutions differ only in their base and locator regions, even though they look quite different to the human eye.

**Step 1: Loop over nodes from lowest to highest node in the diagram.**

The node list is not necessarily integers from 1 to (2 \* number-of-chords) as space may be left open between some or all nodes to accommodate future additional chord extensions.

**Step 2: Inspect the chord containing the current node.**

If the current node is the low-end of the chord, then no work can be done at this point for this chord. No boundaries or regions are affected, so the state of the BoundaryTracker is unchanged. No other work can be completed yet for this chord, and the process is started again with the chord for the next node.

**Step 3: A new intersection and new region**

When the current node is the high-end of the chord, it forms an intersection with the low-end node that was previously passed in Step 2. This also forms a new region.

The intersection is only valid though, if the current node is on the border of the region containing the locator. In Example 1, a current node of 4 is a valid next intersection given that 4 lies on the boundary of region A, which holds the locator, i.e. the locator can “get there” without crossing an existing span. On the other hand, a next node of 1, is not valid and renders the diagram un-planarable, given that node 1 is outside and the locator is stuck within region A.

At this point, we know a new intersection and region will be formed. Two basic types of regions: loops, and slicing. In Example 1, when adding the (3, 5) span, it formed a loop that carved out an area from within the outside region. Examples of slicing are the (5, 6) span that creates region B, and the region created if the next chord is (7, 4); these both split an existing region or sliced off part of an existing region.

The reason for the distinction between region formation types, is that loop boundaries are trivial to determine, but boundaries for regions created via slicing require a much more involved process.

The code determines the region formation type by comparing the current node with the last intersected node (held by the BoundaryTracker). If the last intersected node is less than the current node then this is a loop region; otherwise, it is a sliced region. Note that the BoundaryTracker initializes the last intersected node to zero.

**Step 4: New region type: loop region**

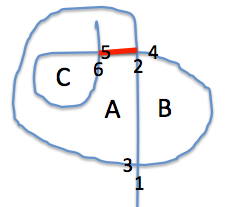
The next region identifier is requested from the RegionFactory object. This object is wholly responsible for region assignments, and ensures that region identifiers can be compared (to ensure the canonical form of a boundary name).

A loop region has a simple boundary, formed by the span whose end points are the same as those in the current chord. In Example 1, chord (2, 5) represents the intersection that formed region A. Span (2, 5) is the boundary stretching from node 2 to node 5. The BoundaryTracker is updated, adding the span (2,5), between the regions outside and A, with a flag denoting that this is due to a new intersection.

The locator and base regions are never changed via creation of a loop region.

An additional bounding span must be added if this loop connects back to an existing diagram. The end points for the span are the last intersected node to the low-end of the current chord. The regions on both sides of this new connector span are the same -- the current locator region.

Example 3

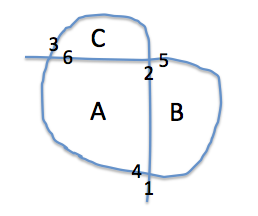


Example 3 shows loop region C, created by the chord (5, 6) and the connector span (4, 5), shown in red, that attaches the loop to the existing diagram.

**Step 5: New region type: slicing region**

If the new region is not formed by a loop, it is instead a slice or bifurcation of an existing region.

Example 4



The (5, 6) span split region A, slicing off a portion that becomes region C.