A SURVEY OF GRID DIAGRAMS AND A PROOF OF ALEXANDER'S THEOREM

NANCY SCHERICH

Abstract. Grid diagrams are a representation of knot projections that are particularly useful as a format for algorithmic implementation on a computer. This paper gives an introduction to grid diagrams and demonstrates their programmable viability in an algorithmic proof of Alexander's Theorem. Throughout, there are detailed comments on how to program a computer to encode the diagrams and algorithms.

1. Introduction

Grid diagrams have gained popularity since the use of grids to give a combinatorial definition of knot Floer homology [8]. They have also proved useful in determining the mosaic number of knot mosaics [6] and understanding arc index [2]. Additionally, grid diagrams can further simplify knot invariance arguments as there are only two required grid moves in contrast to the three required Reidemeister moves [2, 3]. The purpose of this paper is to provide an introduction to grid diagrams and grid moves with a focus on computer implementation. The main result of this paper is a computer implemented, grid diagrammatic proof of Alexander's Theorem, based on the work of Kauffman and Lambropoulou [5].

1.1. **Grid diagrams.** A **grid diagram** is a square grid such that each square within the grid is decorated with an x, o or is left blank. This is done in a manner such that every column and every row has exactly one x and one o. The **grid number** (or index) of a grid diagram is the number of columns (or rows) in the grid. This paper follows the grid notation used by [8] which uses matrix notation with the convention that the rows and columns are numbered top to bottom and left to right.

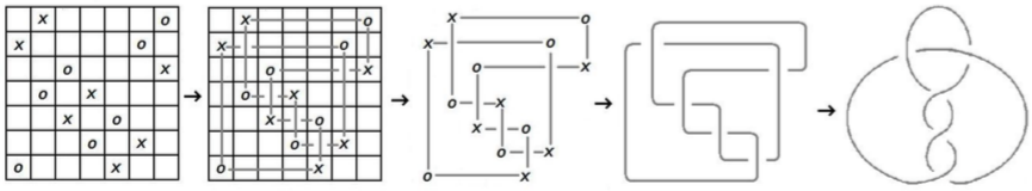


Figure 1

A grid diagram is associated with a knot by connecting the x and o decorations in each column and row by a straight line *with the convention that vertical lines cross over horizontal lines.* These lines form strands of the knot, and removing the grid

leaves a projection of the knot. As a result, grid diagrams represent particular planar projections of knots.

Here we use the word *knot* synonymously for a knot or link. The *knot type of a grid* is the knot type of the knot associated with the grid. The *strand of the grid* or *strand of the knot* refers to a portion of the arc of the knot that is represented in a single column or row of the grid. There are some instances where the distinction between the x and o decorations is important. However, for the results in this paper, it is only the location of the decorations, and not the actual decoration that is needed.

1.1.1. *Computer implementation.* Each x and o decoration has a location (row,column) which will be called a **node** of the grid. We can define an object *grid* as a set of $2n$ nodes.

$$grid = \{(r_1, c_1), \cdots, (r_{2n}, c_{2n})\}$$

For the example grid in Figure 1, we get the grid

$$\{(1,2),(1,7),(2,7),(2,6),(3,3),(3,7),(4,2),(4,4),(5,3)(5,5),(6,4,),(6,6),(7,1),(7,5)\}.$$

The notation *node.row* and *node.column* will be used to collect either the row or column number of the node. Here are some helper functions that can be defined on a grid diagram. To locate a strand of the knot in a row or column, one needs to find the corresponding nodes in the same row or same column.

---

find-row-neighbor

Input: node
for node in grid **do**
  if node.row == input.row and node.column $\neq$ input.column **then**
    return node
  end if
end for

Drawing a projection of the knot represented by the grid is as easy as plotting lines between the nodes in each row and column. This will not encode the crossing information, but it is understood that vertical lines cross over horizontal lines. Be aware that the nodes in the grid are of the form (row,column) which is not in Euclidean (x, y) coordinates.

Euc

Input: node
return (n+node.row+1, node.column)

---

graph

Input: grid
for node in grid **do**
  plot line from Euc(node) to Euc(find-row-neighbor(node))

plot line from Euc(node) to Euc(find-column-neighbor(node))
end for

1.2. **Grid moves.** There are two grid moves used to relate grid diagrams: (de-) stabilization and commutation. These play a role analogous to the Reidemeister moves for knot diagrams [10]. Following the notation from [8] and [11], these grid moves are are defined below.

(de-)Stabilization: Stabilization is the addition of a kink while destabilization is the removal. The term (de-)stabilization is used to describe this move without specifying whether a kink is being added or removed. It is important to note that (de-)stabilization does not preserve the grid number, but simply corresponds to an isotopy of the underlying knot. A kink may be added to the right or left of a column, above or below a row, and at an point along the strand of the knot in any row or column. For one example, Figure 2 shows a kink addition to row c. To do this, insert an empty column between the x and o markers of row c. Then insert an empty row above or below row c. Move either the x or o decoration in row c into the adjacent grid square in the added row. Complete the added row and column with x and o decorations appropriately.

To add a kink to a column, switch the notions of column and row. To remove a kink, follow these instructions in reverse order. As shown, stabilization increases the grid number by 1 while de-stabilization reduces the grid number by 1.

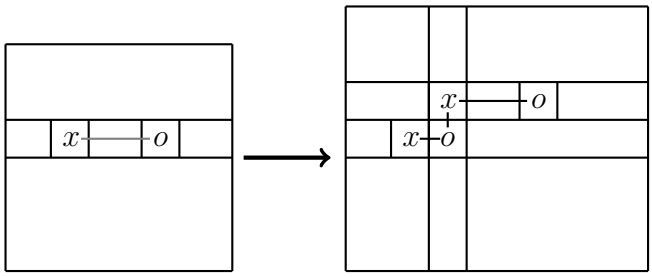


FIGURE 2. De-stabilization, or kink addition

Commutation: Commutation interchanges two consecutive rows or columns of a grid diagram. This move preserves the grid number, but does not always preserve the knot type.

When a commutation preserves the knot type of the grid, it will be called an *admissible commutation*, which is explained in more detail in Section 1.3. The following theorem, due to Cromwell [2] and Dynnikov [3], see also [11], explicates the relationship between grid diagrams, knots and the two grid moves.

Theorem 1.1. *Two grid diagrams have the same knot type if an only if there exists a finite sequence of admissible commutation and (de-)stabilization grid moves to relate one grid to the other.*

One way to see this theorem is to accomplish all of the Reidemeister moves using the two grid moves. For example, the Reidemeister I move can by done by adding a kink and then commuting the new column, shown in Figure 3.

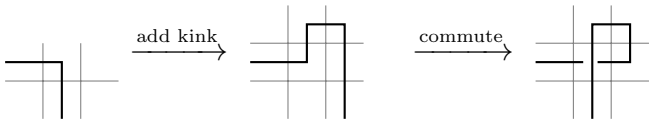


FIGURE 3. Reidemeister I move via grid moves

1.2.1. *Computer implementation.* While the commutation grid move has subtleties on when admissible, it is very straightforward to implement. Either two rows or two columns can be commuted.

---

column-commutation

Input: grid, column numbers i and $i+1$
for node in grid with node.column i or $i+1$ **do**
  if node.column $== i$ **then**
    increase node.column to $i+1$
  end if
  if node.column $== i+1$ **then**
    decrease node.column to i
  end if
end for
return grid

On the other hand, (de)-stabilization is always admissible, but much more complicated to implement. There are many different ways a kink can be added or removed, each of which requires a separate implementation. The following algorithm will implement the specific kink addition depicted in Figure 2 which adds a kink below an indicated row. This function can be gently altered to accomplish the other types of kink addition.

---

specific-kink-addition

Input: grid, nodes $(i,j),(i,k)$ with $j<k$
for node in grid **do**
  if node.column $> j$ **then**
    increase node.column by 1
  end if
  if node.row $> i$ **then**
    increase node.row by 1
  end if
end for
replace (i,j) by $(i+1,j)$

add nodes to grid: $(i, j+1), (i+1, j+1)$
return grid

1.3. **Commutation; A closer look.** The commutation grid move is defined to interchange *any* two consecutive rows or columns in a grid, which could change the knot type of the grid. This section will establish some conditions under which commutation is admissible. The results of this section focus on column commutation, but conditions for row commutations are analogous. For a more detailed discussion, see [11].

Figure 4 shows the four possible relative positions of two consecutive columns, up to different x and o labeling and exact spacing. Denote these possibilities as *disjoint*, *nested*, *point-shared* and *interlocked*.

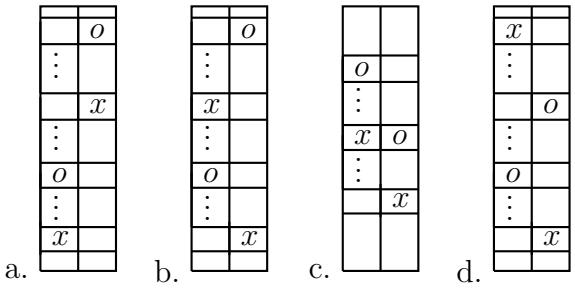


FIGURE 4. Consecutive columns that are: a. disjoint, b. nested, c. point-shared, d. interlocked

Proposition 1.2. *Commutation of columns that are disjoint, nested or point-shared are admissible commutations.*

This can be quickly proven by considering the different strand configurations of the underlying knot, all of which commutations correspond to an isotopy, Reidemeister I or II move.

Depending on the strand configuration of the underlying knot, commutation of interlocked columns can change the knot type of the grid. So as long as the two columns are not interlocked, then commutation is admissible.

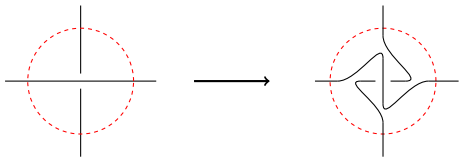
Corollary 1.3. *A column that has the x and o in adjacent grid squares or the outermost grid squares can be commuted with any other column.*

Proof. This column can never be interlocked with another column. □

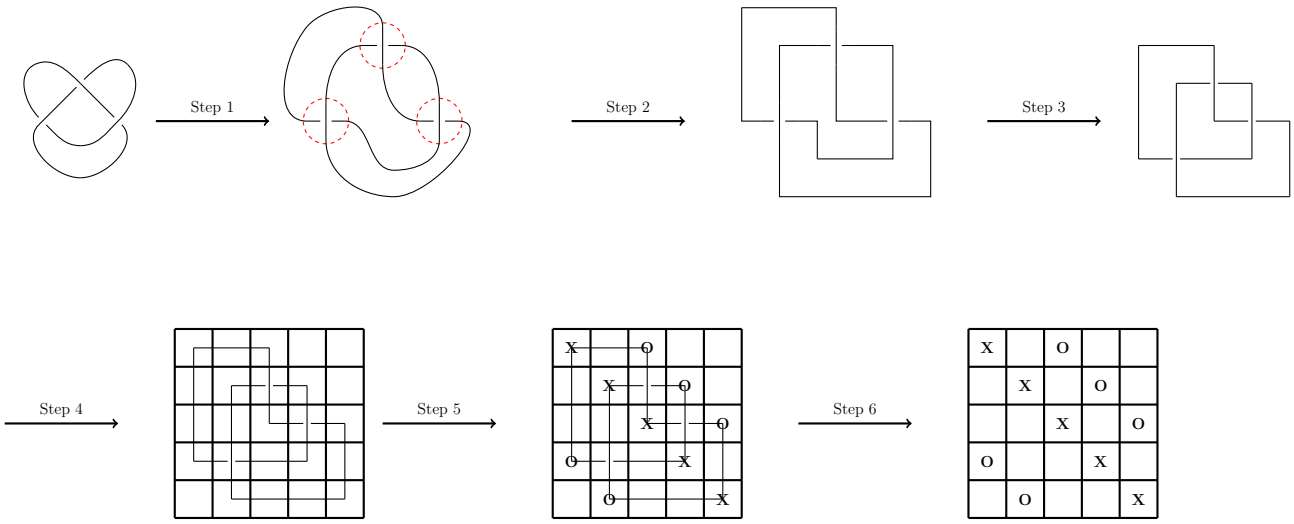
2. From knot diagram to grid diagram; an algorithm

Cromwell proved in [2] that every knot can be represented by a grid diagram. Below is a more detailed algorithm of the process to create a grid diagram.

Step 1: Start with a projection of the knot in general position. Locally isotope each crossing by a rotation so that the vertical strand crosses over the horizontal strand.



Step 2: Rectalinearize or polygonalize the knot.
Step 3: Isotope the rectangular knot so that no two strands are colinear.
Step 4: Superimpose a grid on top of the knot so that there is one column for every vertical strand and one row for every horizontal strand.
Step 5: Place an x or an o on the corners of the knot in an alternating fashion so that every row and every column of the grid has exactly one x and one o.
Step 6: Remove the strands of the knot leaving only the grid diagram behind.



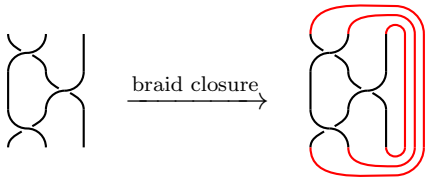
3. Braids and Alexander's Theorem; A review

Geometrically, a **braid** on n strands is a vertical stack of pictures of the form:

σ_i $\qquad\qquad$ σ_i^{-1}

Here, σ_i has the strand in the i'th position cross downwards behind the strand in the $i+1$ position, and σ_i^{-1} has the strand in the i'th position cross downwards in front of the strand in the $i+1$'th position. A braid can be described by listing the σ_i's that occur in order from top to bottom. Conversely, given a word in σ_i's, one can recreate a braid by stacking the diagrams in order from top to bottom. Importantly, what distinguishes a braid from a more general tangle is the monotonicity of the strands and

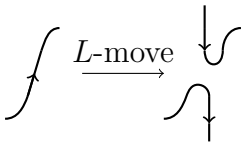
the crossings occur at distinct heights in the braid. This is best seen by orienting the strands with a downwards flow. Braids are only considered up to isotopy of the strands relative to the endpoints and which preserves the monotonicity of the strands. To get a knot from a braid, one takes the **braid closure** by connecting the top endpoints of the braids to the bottom endpoints of the braid without adding additional crossings. By convention, these new strands wrap around the right side of the braid.



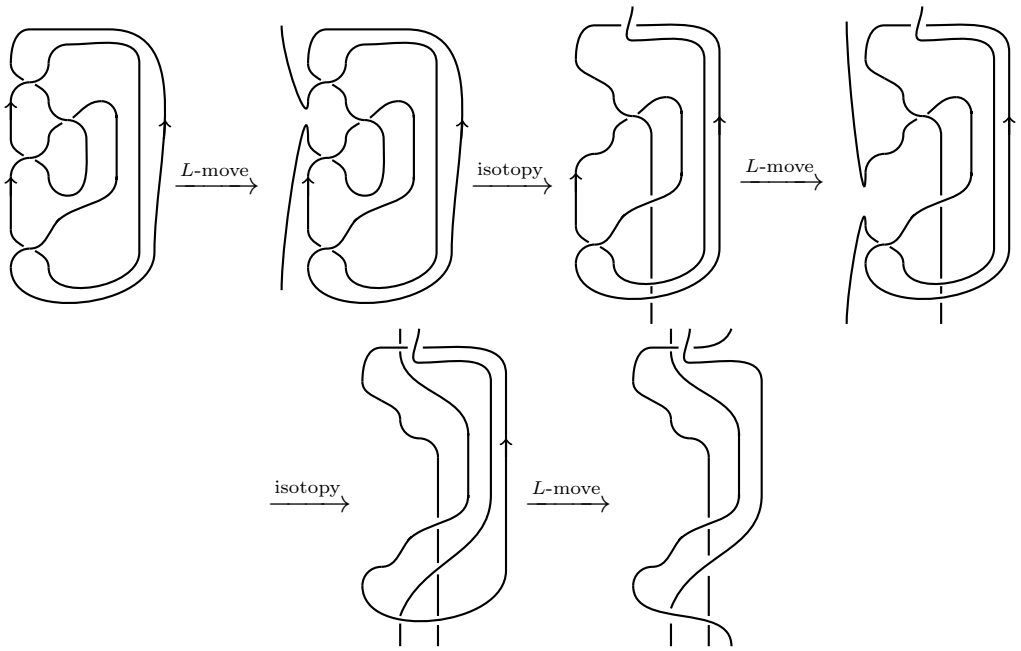
It is important to understand exactly what a braid closure looks like. In the figure above, the knot on the right has all of its crossings on the left side, and the rest of the strands wrap around on the right. The crossings on the left occur as a braid, namely the crossings occur at distinct heights and the strands flow monotonically downward. It is clear that some knots can be represented as the closure of a braid, but Alexander's Theorem gives the full result.

Alexander's Theorem: [1] *Every knot diagram is isotopic to a braid closure.*

There are many proofs of Alexander's theorem, including Yamada and Vogel's algorithm in [12], Morton's algorithm by threading [9] and Jones' more casual algorithm in [4] by "throwing the bad parts over one's shoulder". At first glance, one might naively try to prove Alexander's theorem by isotoping all the crossing of a knot to one side in a way that resembles a braid closure. It is easy to force the crossings to occur at distinct heights, but the problem that quickly arises is the strands may not be monotone. Now Kauffman and Lambropoulou in [5] offer a solution to fix this monotonicity by using L-moves. The idea behind the L-move is to break a strand with the wrong orientation and replace it by two strands with correct orientations.



The two new vertical strands either both lie above the entire knot, or both below. This is determined visually by the crossings of the strand on either side of where the move is applied. The algorithm suggested by Kauffman and Lambropoulou is to choose an orientation of the knot and apply L-moves to the upwards oriented arcs. This leaves a resulting diagram isotopic to a braid, and the closure of this braid is isotopic the original knot. One downfall of this procedure is the amount of visual decisions and isotopy to choose where to apply the L-moves, and then to finally adjust the diagram to look like a braid. Here is an example this algorithm to find a braid for the figure 8 knot.

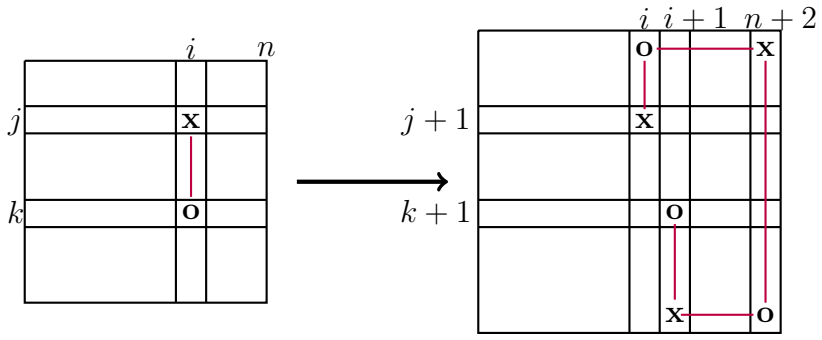


4. Mid-grid Moves and Main Result

In the closure of the braid, all of the strands in the braid flow downward, while the strands in the closure flow monotonically upward. These upward strands lie on the outside righthand side of the braid. The idea of the mid-grid move is to move strands of the knot represented in the middle of a grid to the outside of the grid. This move, when applied to an oriented knot will lead to a shape similar to the braid closure. If applied to a strand with an upwards orientitaion, this move changes the orientation of the strand to have two downwards oriented strands in the middle of grid, and one upward oriented strand on the outside right of the grid.

This move is the grid analogue of the L-move. There are several benefits of interpreting this move in the grid environment. The mid-grid move can be concretely described by a sequence of admissible commutation and (de)-stabilization grid moves giving a clear proof that the knot type is preserved. Additionally, there is no ambiguity on whether the new vertical strands will lie above or below the entire knot. Since within the constraints of a grid, all vertical strands cross over horizontal strands, the new vertical strands can be seen as lying entirely above without question or need of visual determination.

4.1. **Mid-Grid Move on Column i.** Starting with a grid diagram for the knot, keep the strands of the knot pictured in the grid. The mid-grid move will increase the grid size by 2 and is shown below (up to x and o labeling).

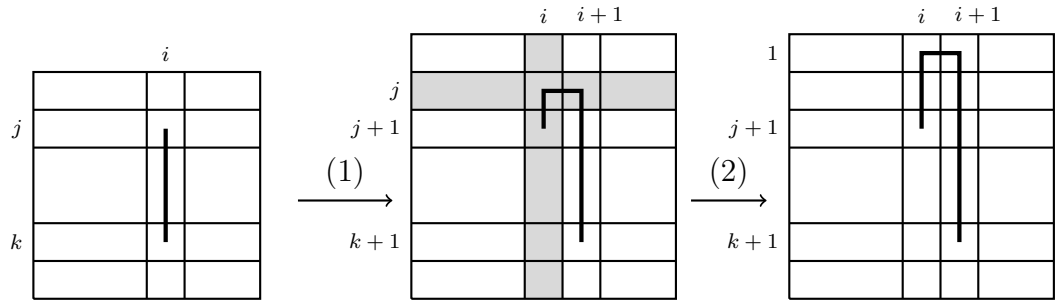


Proposition 4.1. *The mid-grid move preserves the knot type of the grid.*

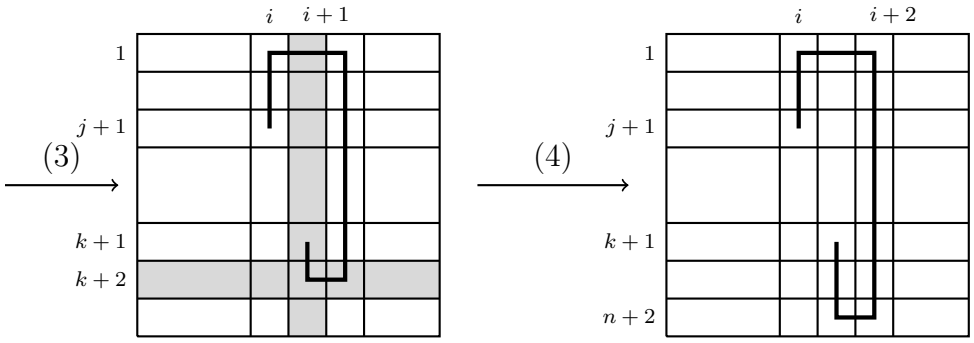
Proof. It suffices to show that the mid-grid move can be accomplished by a sequence of admissible commutation and (de)-stabilization grid moves.

Let j be the row of the top decoration and k be the row of the bottom decoration of the strand in column i. In the pictures to follow, the specific decorations have been omitted for clarity.

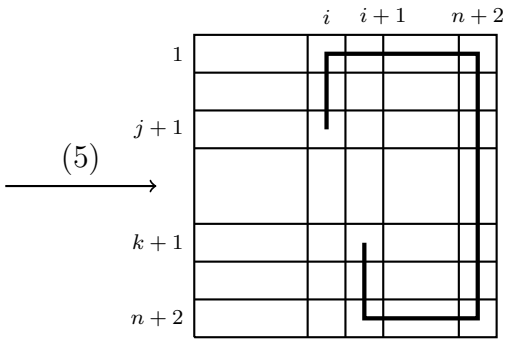
(1) Add a kink above and to the left of row j. The new row and column are shaded.
(2) The horizontal portion lying in the new row j has its endpoints in adjacent grid squares. By Corollary 1.3 this row can be admissibly commuted with any other row. So commute row j upwards $j-1$ times until it is the new top row. The vertical strand that was in column i has been shifted to column $i+1$ and the bottom entry has been shifted to row $k+1$.



(3) Add a kink below and to the left of the bottom endpoint of the strand in column $i+1$.
(4) The horizontal portion of the new row is now in row $k+2$. Since the horizontal endpoints are in adjacent grid squares, again Corollary 1.3 gives that this row admissibly commutes with any other row. So commute row $k+2$ downwards $n-k$ times until it is the new bottom row.



(5) The vertical strand that was originally in column i is now in column $i+2$ and extends the entire height of the grid. By Corollary 1.3 column $i+2$ can be admissibly commuted to the right $n-i$ until it is the rightmost column.



□

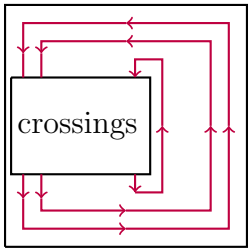
4.1.1. *Computer implementation.* The midgrid move, while a complicated list of commutation and (de)-stabilizations, can be described very succinctly.

---

midgrid

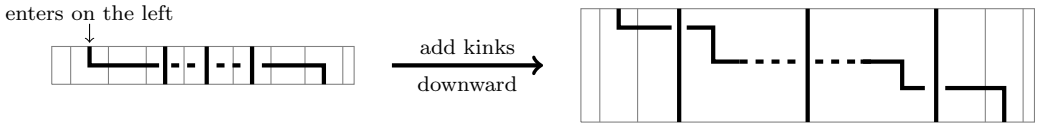
Input: grid, column number i
for node in grid **do**
  increase node.row by 1
  if node.column $> i$ **then**
    increase node.column by 1
  else
    if node.column $= i$ and node.row $>$ (find-column-neighbor(node)).row **then**
      increase node.column by 1
    end if
  end if
end for
add nodes to grid: $(1,i),(1,n+2),(n+2,i+1),(n+2,n+2)$
increase n by 2
return grid

4.2. **A Proof of Alexander's Theorem.**
Start with the an oriented diagram of the knot. Follow the algorithm in Section 2 to get a grid diagram for the knot, but leave the strands of the knot on the grid. Locate the columns where the orientation is upwards. Perform mid-grid moves on those columns in successive order starting with the furthest right column and work one by one to the left. This will leave a grid diagram so that all the crossings are on the left side with orientations flowing downwards, pictured below.

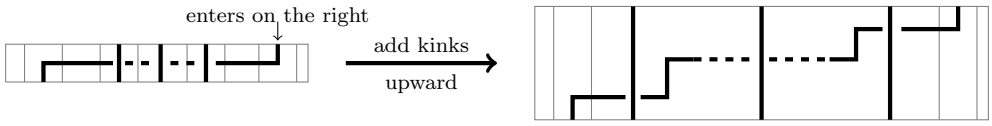


Now that all of the crossings have been grouped together with the appropriate downward orientation, the last hurdle is to create monotonicity of the crossings; each row needs to have only one crossing. To achieve this, working from top to bottom, identify the first row that has more than one crossing. Because of the downward orientation, there are only two possibilities for how the horizontal strand in the row enters and exists the row.

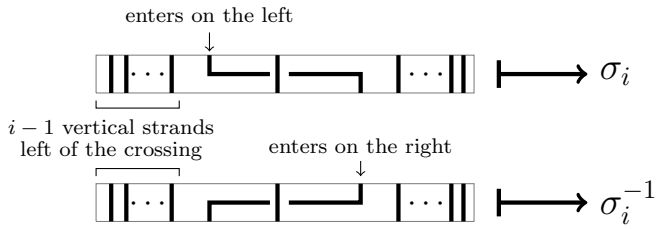
One possibility is that the strand enters the row on the left of the crossings and exists on the right. Working from left to right, add a kink downward in between each vertical strand passing through the row. This forces each crossing to happen in a separate row.



The other possibility is that the horizontal portion of the strands enters the row on the right and exists on the left. Working form left to right, add a kink upwards between each vertical stand, again forcing each crossing to happen in a separate row.

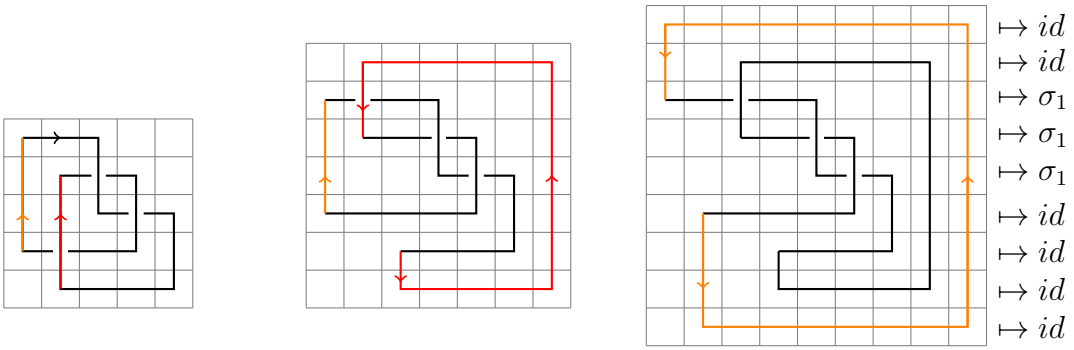


At this point, you can read off the desired braid word by assigning a braid element to each row and listing the elements from top to bottom. Rows with out a crossing get assigned the identity braid element. Each row with a crossing gets assigned either a σ_i or σ_i^{-1} using the following rules:



The index i depends only on the number of vertical strands passing through the row *to the left of the crossing*. Notice that it is not enough to just consider the column number of the crossing. There may be many columns to the left of the crossing without a vertical strand and these columns do not count towards the index of the σ. Whether you assign a generator σ_i or its inverse depends only on where (to the left or right of the crossing) the end of the horizontal strand enters the row. □

Example 4.2. Starting with the grid diagram of a trefoil knot described earilier, this algorithm finds the braid σ_1^3.



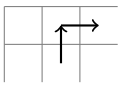
4.2.1. *Computer Implementation.* A working implementation in Python can be found in [7].

The first step in this algorithm is to fix an orientation of the knot within the grid. Because this implementation utilizes object oriented design, we define a new object type OrientedGrid which adds two attributes to the nodes:

(row, column, in-orientation, out-orientation).

Here the in/out-orientations are $above, below, left$ or $right$. These correspond to the in-orientation flowing in from left, in from above, or the out-orientation is flowing to the left, to the box above etc.

For example, the oriented node $(1, 2, below, right)$ shows:



Every grid has many different associated OrientedGrids. However, there are only two OrientedGrids that give rise to an orientation of the associated knot. The in and out orientations must agree on every row, column and node. The following algorithm systematically traverses the grid to produce an OrientedGrid that does gives rise to one choice of orientation for the associated knot.

```
orient
  Input: grid, start= first node in grid
  if find-row-neighbor(start).column > start.column then
      set out-orientation of start to "right"
  else
      set out-orientation of start to "left"
  end if
  previous-node = start
  current-node = find-row-neighbor(start)
  while True do
      if previous-node.row == current-node.row then
          if previous-node.column > current-node.column then
              set in-orientation of current-node to "left"
          else:
              set in-orientation of current-node to "right"
          end if
          next = find-col-neighbor(current-node)
          if  next.row > current-node.row then
              set out-orientation of current-node to "below"
          else
              set out-orientation of current-node to "above"
          end if
      else
          if previous-node.col == current-node.col then
              if previous-node.row > current-node.row then
                  set in-orientation of current-node to "above"
              else
                  set in-orientation of current-node to "below"
              end if
              next = find-row-neighbor(current-node)
              if  next.column > current-node.column then
                  set out-orientation of current-node to "right"
              else
                  set out-orientation of current-node to "left"
              end if
              break if current-node=start
          end if
      end if
  end while
       return OrientedGrid
```

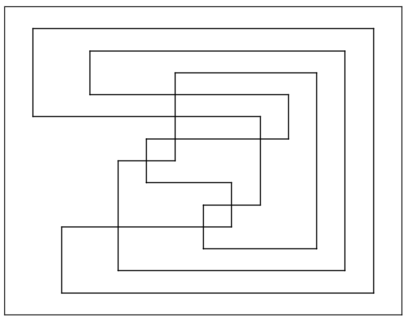
Now, all of the machinery is in place to implement the algorithm described in the proof of Alexander's theorem, which we will call grid-to-braid. As written, the grid-to-braid algorithm will not output a braid word, but will output a projection of the knot represented by the grid after the appropriate midgrid moves are applied. The user then would have to manually make the crossings monotone and assign braid elements to each row, which is a simple visual process.

---

grid-to-braid

```
Input: grid
OrientedGrid = orient(grid)
for  node in OrientedGrid do
    mark node if out-orientation is "above"
end for
for marked nodes in OrientedGrid do
    if node.column is largest column of marked nodes then
        OrientedGrid = midgrid(node, node.column)
    end if
end for
return graph(OrientedGrid)
```

Example 4.3. The code in [7] when applied to the grid in Figure 1 outputs the following diagram.



This diagram requires two adjustments because rows 6 and 10 each have two crossings. After adjusting, this diagram gives the braid word $\sigma_2\sigma_1\sigma_2^{-1}\sigma_1^{-1}\sigma_1^{-1}\sigma_2^{-1}\sigma_2^{-1}\sigma_1^{-1}$.

Acknowledgement

Many of the results in this paper were written in an unpublished VIGRE funded REU paper advised by Liam Watson in 2010. After attending Lambropoulou's talk at the Knots in Hellas 2016 conference, I was inspired to re-write this paper to give a rounded perspective of the results. I would like to especially thank Andrew Malta and Kevin Malta for their great help with the computer implementation.

References

[1] James Alexander. A lemma on a system of knotted curves. *Proc. Nat. Acad. Sci. USA*, 9:93–93, 1923.

[2] Peter R. Cromwell. Embedding knots and links in an open book. I. Basic properties. *Topology Appl.*, 64(1):37–58, 1995.

[3] I. A. Dynnikov. Arc-presentations of links: monotonic simplification. *Fund. Math.*, 190:29–76, 2006.

[4] Vaughan Jones. The Jones Polynomial for dummies. Lecture notes, 2014.

[5] L. H. Kauffman and S. Lambropoulou. Virtual Braids. *ArXiv Mathematics e-prints*, July 2004.

[6] Hwa Jeong Lee, Kyungpyo Hong, Ho Lee, and Seungsang Oh. Mosiac number of knots.

[7] Andrew Malta, Kevin Malta, and Nancy Scherich. https://github.com/nscherich/grid-algorithm.

[8] Ciprian Manolescu, Peter Ozsváth, Zoltán Szabó, and Dylan Thurston. On combinatorial link Floer homology. *Geom. Topol.*, 11:2339–2412, 2007.

[9] H. R. Morton. Threading knot diagrams. *Math. Proc. Cambridge Philos. Soc.*, 99:247–260, 1986.

[10] K. Reidemeister. *Knotentheorie*. Springer-Verlag, Berlin, 1974. Reprint.

[11] Nancy Scherich. A simplification of grid equivalence. *Involve*, 8(25):721–734, 2015.

[12] Pierre Vogel. Representation of links by braids: A new algorithm. *Commentarii mathematici Helvetici*, 65(1):104–113, 1990.