Computation of Knot Invariants: Determinant, Signature, and the Goeritz Matrix

Alex Klein
Winona State University, Spring 2013

I. Background and Motivation

A. Knots and Knot Invariants

We define a **knot** as a closed, non-intersecting curve in three-dimensional space. For purposes of the calculations we will be doing later, we consider the special case of the **polygonal knot**, which is a knot composed of a finite number of line segments. A pair of given knots is considered equivalent if and only if there exists a way to deform one into the other without cutting or allowing intersections in the knot. Then standard method of deforming a knot are the three "Reidemeister Moves", along with stretching/contracting the the knot itself. It's important to note that "knot popping" by compacting parts of the knot down to a single point, is not a valid deformation. Also taken into consid-

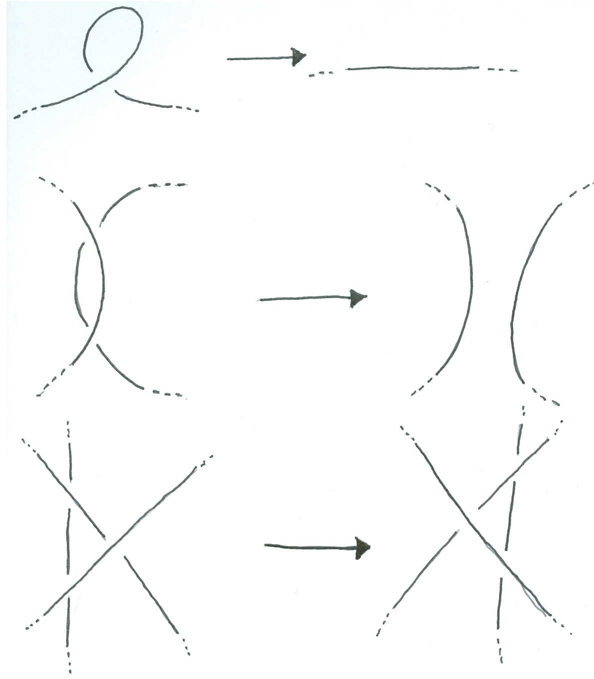


Figure 1: Top to bottom: Reidemeister moves I, II, III

eration are **knot invariants**, quantitative properties of a knot that do not change when the knot is deformed without cuts/intersections. If an invariant is different from one knot to the other, than those knots cannot be equivalent. It is important to note, however, that equal values for an invariant between two knots does not imply equivalence. The two invariants we will be considering here are the determinant and signature of a knot.

Often knots are pictured as projections onto a two dimensional plane, with "gaps" in the knot indicating crossings. Imagine it as laying a knot flat on a table and looking down it from above. For the purposes of our determinant and signature calculations, the knot is considered in the xy-plane, and all crossings will be considered with the line segments comprising then in general position. This consideration will not change the invariant, as to put a knot projection's crossing into general position it only requires a rotation of perspective about the knot as a whole with no deformations (legal or otherwise).

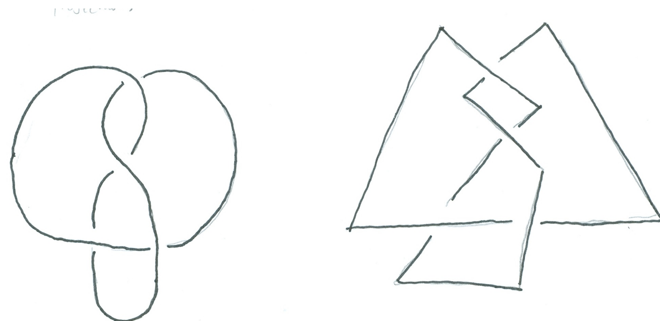
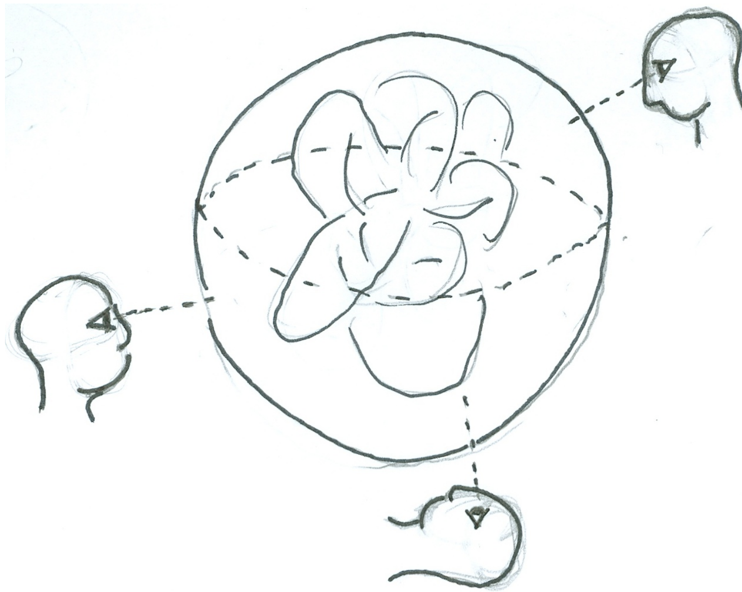


Figure 2: Left: A knot diagram. Right: A polygonal knot diagram of the same knot, composed of a finite number of line segments

Figure 3: We can freely "move about" a knot without changing it, but it does change the 2-dimensional projection.



B. **Goeritz Matrix and Determinant of a Knot**

The determinant of a knot K is given by the determinant of its projection's corresponding Goeritz matrix G. It is important to note that although the specific Goeritz Matrix is dependent on the projection, the determinant of K is invariant.

We begin by shading the projection of K in a "checkerboard-fashion", alternating coloring regions white/black such that no two regions of the same color share an edge.

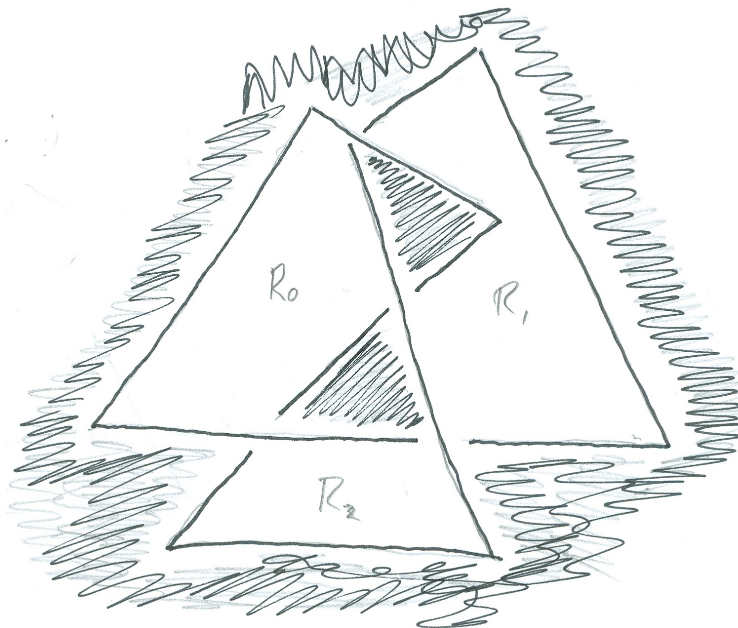


Figure 4: The "figure eight" knot colored in a checkerboard fashion.

At each crossing, C, we assign a value of $\eta(C) = \pm 1$ according to the convention pictured below.

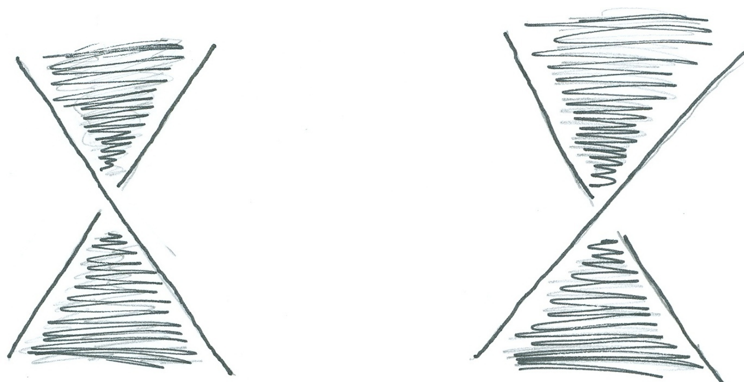


Figure 5: Left: Crossing with $\eta = -1$. Right: Crossing with $\eta = 1$.

It is from here that we can construct G', a preliminary step towards the Goeritz matrix G. To begin we denote the $n+1$ white regions of our shaded projection by $R_0, R_1, \ldots, R_n$. For each entry g_{ij} in G', we have

$$g_{ij} = \begin{cases} -\Sigma\eta(C), & \text{The sum of crossings C incident to } X_i \text{ and } X_j \text{ if } i \neq j \\ -\sum\limits_{\substack{k=0,1,\ldots,n \\ k\neq i}} g_{ik}, & \text{if } i = j. \end{cases}$$

The Goeritz matrix G is the $n \times n$ symmetric matrix given by deleting (without loss of generality) the 0th row and column of G'. The determinant of our knot K is then given simply by

$$\det(K) = |\det(G)|.$$

C. **Signature of a Knot**

Recall that the signature of a matrix is the difference between the number of positive and negative eigenvalues. It is tempting to think that since the determinant of K follows so easily from computing $\det(G)$, that the signature of K, $\sigma(K)$, is equivalent to the signature of G, $\sigma(G)$. However $\sigma(G)$ changes on a Reidemeister II move. We obtain $\sigma(K)$ then, by subtracting an adjustment factor μ from $\sigma(G)$.
To obtain adjustment factor μ, we begin by putting an orientation on the projection of K. We label each crossing C as either "Type I" or "Type II" according to the pictured convention.

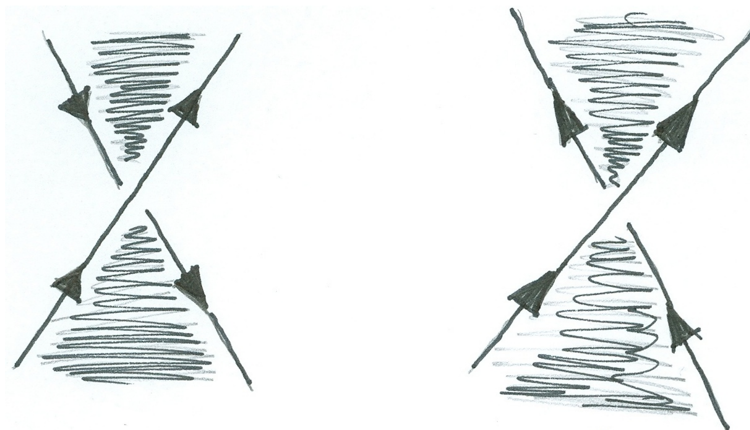


Figure 6: Left: Crossing of type I. Right: Crossing of type II.

We then define μ as the sum of the $\eta(C)$ values of Type II crossings C. The signature of K is then given by

$$\sigma(K) = \sigma(G) - \mu, \text{ where } \mu = \sum_{\text{Type II C}} \eta(C).$$

In regards to connected sums of knots, experimentations with small examples show that given two knots K and T we have,

$$\det(K\#T)$$

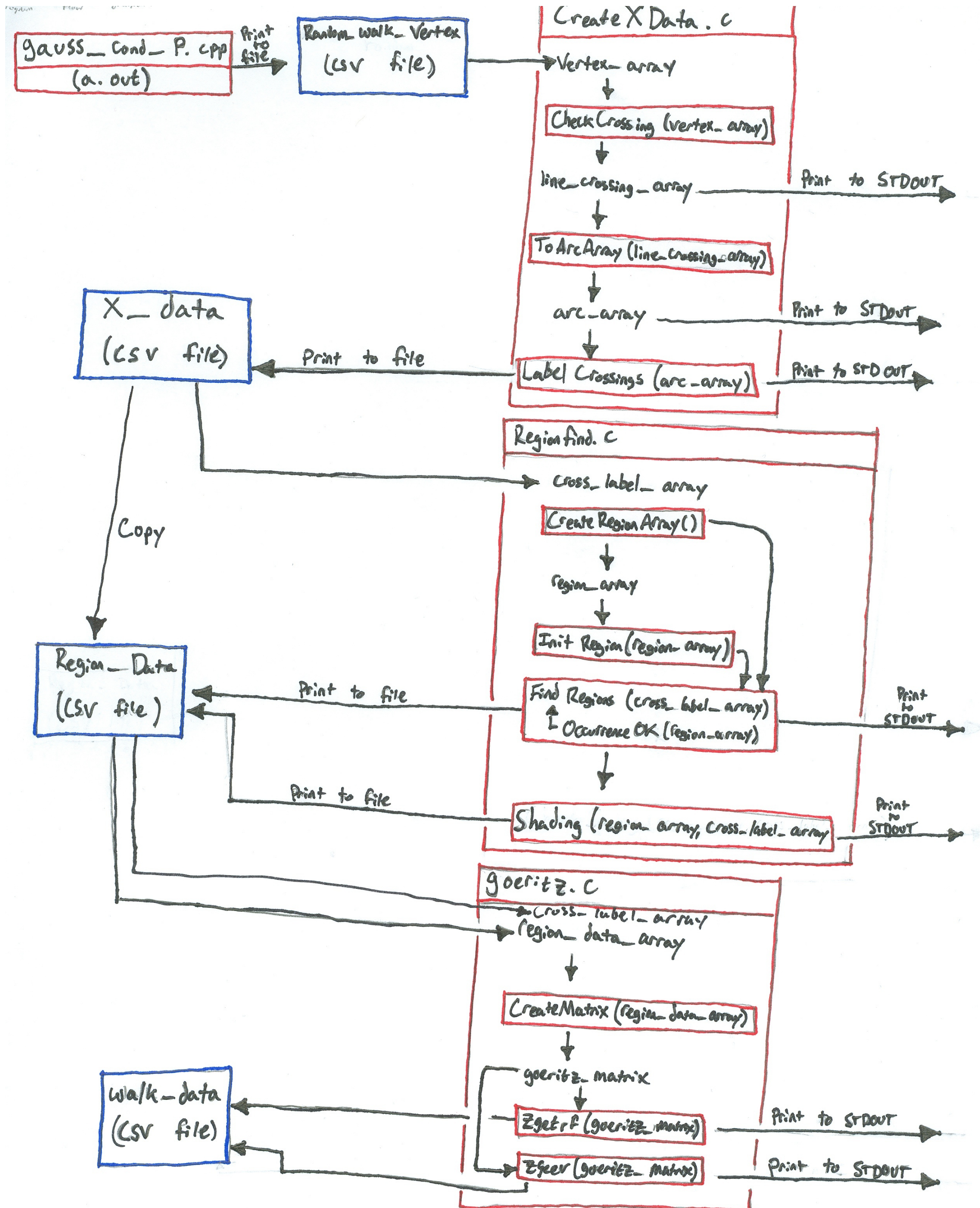
and

$$\sigma(K\#T) = \sigma(K) - \sigma(T).$$

II. Algorithmic Processes

A. Program Flow

Figure 7: Sketch of the flow of the program as data is generated and moved through different processes in separate source files



Input data is formatted as a number of ordered triples representing the coordinates in $\mathbb{R}^3$ of the vertices of the polygonal knot. Coordinates are given to the program in the order they occur while travelling along the polygonal knot in the direction of its orientation. The actual orientation itself is irrelevant, as long as the order of points is preserved for the orientation used. Each coordinate is stored as a Vertex data structure with datamembers x, y, z, refrenced as `vertexname->x`, `vertexname->y`, `vertexname->z`, respectively.

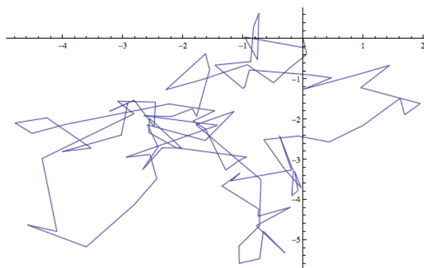


Figure 8: *Mathematica* plot of sample vertex data for a random walk.

The input data is generated as a random walk via a C++ program written by Dr. Nathan Moore, whose output is stored in a file sent to my program as a command line argument. The second command line argument to the program determines the output file, where the crossing type, η-value, and label of each crossing of the random walk will be stored as plain text. This file is sent as an argument to a second program that analyzes for regions and shading of those regions, before finally computing the determinant and signature of a Goeritz matrix.

B. **Finding Crossings**

The lines/edges of the polygonal knot are determined by parametric equations for x, y, and z coordinates of straight lines between vertices in sequence. I.e. a parametric equation describes the line/edge connecting vertex 1 to vertex 2, vertex 2 to vertex 3, and so forth. The final vertex in the sequence connects to the first in order to close the knot. The parameter for each equation is considered only on the interval $[0, 1]$,

$$(x(t), y(t), z(t)) = (x_1, y_1, z_1) + t(x_2 - x_1, y_2 - y_1, z_2 - z_1).$$

To find crossings, we consider the parametric equations of the first line/edge, parameterized by t. The equations for each line we compare it to will be parameterized by s. For purposes of finding where the lines cross each other, we flatten our knot to the xy-coordinate plane by considering only the parametric equations for the x and y coordinates of each line. To find where the lines parameterized by t and s cross, we must find where their parameterizations return equal x and y values. Let $(x_1, y_1), (x_2, y_2)$ be the xy-coordinates for the vertices defining the first line, and $(x_3, y_3), (x_4, y_4)$ be the xy-coordinates for the vertices defining the second line. Let Δ_{x1} and Δ_{y1} represent the change in x and the change in y in line 1's parametric equations for x and y, with Δ_{x2}, Δ_{y2}, defined similarly for the second line. The computations to find s and t where x and y are equal for the parametric equations for each line are given by

$$s = \frac{\Delta_{x1}(y_3 - y_1) + \Delta_{y1}(x_1 - x_3)}{(\Delta_{x2} \cdot \Delta_{y1}) - (\Delta_{y2} \cdot \Delta_{x1})},$$

$$t = \frac{x_3 + s\Delta_{x2} - x_1}{\Delta_{x1}}.$$

For this to be a "legitimate" crossing, both the s and t values must be within the interval $(0, 1)$, as that is as far as we've "drawn" each line. If a crossings is found, we put the values of s and t where the crossing occurs into the equations for the z-values of each line to determine which line (named by which vertices serve as its endpoints) crosses over the other.

To ensure that each line is checked for crossings and that each crossing is only recorded once, we perform the crossing computation on the first line compared with the third line, fourth line, etc. We then do the same for the second line compared to the fourth, fifth, etc. as we note that lines cannot possibly cross the next line in sequence in a polygonal knot. It is important to remember a definition of a line from the last vertex to the first in order to "close" the knot.

Psuedocode:

```
//Finding crossings
for(a = 1; a < (# of lines) ; a++){
        for(b = a + 2; b < (# of lines); b++){
                first_line_x = vertex_a->x + t*(vertex_b ->x - vertex_a->x);
                second_line_x = vertex_c->x + s*(vertex_d->x - vertex_c->x);
```

```
                first_line_y = vertex_a->y + t*(vertex_b ->y - vertex_a->y);
                second_line_y = vertex_c->y + s*(vertex_d->y - vertex_c->y);

        solve(first_line_x = second_line_x, first_line_y = second_line_y, computed_s,
           computed_t);

                if(s > 0 && s < 1 && t > 0 && t < 1) then
                        first_line_z = vertex_a->z + computed_t*(vertex_b ->z -
                           vertex_a->z);
                        second_line_z = vertex_c->z + computed_s*(vertex_d->z -
                           vertex_c->z);

                        if(first_line_z > second_line_z) then
                                first_line crosses over second_line
                        else
                                second_line crosses over first_line
                }
}
```

The lines on which the crossing occurs are stored as part of the `Crossing` data structure, and upon being found a negative integer is assigned to the `Crossing`. Where the crossing occurs in $\mathbb{R}^2$, and the slopes of parametric equations for x and y of each line are used in the computation of the crossing type, which immediately follows. The change in each parametric equation's respective variable - which is from here referred to as the "slope" of the parametric equaition - will also be used in the computation of η-values, which immediately follows type computations. Further more, we store in the `Crossing` data structure's1 `linedata1` and `linedata2` arrays the vertices of the lines the crossing occurs on, and the t/s-values of each line in the crossing, for use in finding knot regions. We also record the lowest numbered vertices for the over and undercrossings. As it requires at minimum four vertices to define a crossing (two crossing line segments), we allocate enough array space for four times the number of crossings for the array of `Crossing` data structures.

C. Type Checking

We compute crossing type immediately upon finding a crossing. We begin by extending a line from the point where the crossing was found on the xy-coordinate plane. As with lines representing our knot, this line is defined parametrically. As with finding crossings, let Δx_1 be the change in x for the parametric equation for x of our first line, and Δy_1 be the change in y for the parametric equation for y of the for the first line, with symbols representing the changes in x and y of the second line's equations defined similarly. The parametric equations for our line extending from the crossing point, then, is given by

$$x(t) = \texttt{crossingpoint->x} + t(\Delta x_1 + \Delta x_2),$$

$$y(t) = \texttt{crossingpoint->y} + t(\Delta y_1 + \Delta y_2).$$

We omit a parametric equation for z of the line coming from the crossing, as it is irrelevant to the type computation.

We check this line against all other lines in the knot for crossings as in part B, with the exception that instead of checking for which line is over/under the other, we merely count the total number of crossings. The number of crossings is odd, then the crossing we are checking is a Type I crossing. If the number of crossings is even, then the crossing we are checking is a Type II crossing.

Psuedocode:

```
//Check Type

crossingline_x = crossingpoint->x + t*(line1_x_slope + line2_x_slope);
crossingline_y = crossingpoint->y + t*(line1_y_slope + line2_y_slope);

for(i = 1; i <= lines_in_knot; i++){
        checkCrossing(crossingline, line[i]);
        if(crossing)
                crossingcounter+1;
}

if (crossingcounter %2 ==0)
```

```
        type = 2;

if (crossingcounter %2 == 1)
        type = 1;
```

The result of the computation is stored in each `Crossing`'s data member for `type`, which is used in the η-computation immediately following typing.

D. η Values

Computing η values of a crossing is a matter of checking various values of each `Crossing` data structure and running them through a series of conditional statements. We first check the type of crossing to find which set of conditional statements to use. We then look at the slopes of the x and y parametric equations of the overcrossing line, then the slopes of the x and y parametric equations of the undercrossing line. At the end this uniquely determines an η-value of 1 or -1 for the crossing being checked.

Pseudocode:

```
type 1:
        if(slope_x_over <= 0 && slope_y_over > 0){
                if (slope_x_under <= 0){
                        eta = -1;
                } else {
                        eta = 1;
                }
        }
        if(slope_x_over >= 0 && slope_y_over < 0){
                if(slope_x_under >= 0){
                        eta = -1;
                } else {
                        eta = 1;
                }
        }
        if(slope_x_over < 0 && slope_y_over <= 0){
                if(slope_y_under <= 0){
                        eta = -1;
                } else{
                        eta = 1;
                }
        }
        if(slope_x_over > 0 && slope_y_over >= 0){
                if(slope_y_under >= 0){
                        eta = -1;
                } else {
                        eta = 1;
                }
        }
        break;

type 2:
        if(slope_x_over <= 0 && slope_y_over > 0){
                if(slope_x_under >= 0){
                        eta = -1;
                } else {
                        eta = 1;
                }
        }
        if(slope_x_over >= 0 && slope_y_over < 0){
                if(slope_x_under <= 0){
                        eta = -1;
                } else {
                        eta = 1;
```

```
            }
        }
        if(slope_x_over < 0 && slope_y_over <= 0){
            if(slope_y_under >= 0){
                eta = -1;
            } else {
                eta = 1;
            }
        }
        if(slope_x_over > 0 && slope_y_over >= 0){
            if(slope_y_under <= 0){
                eta = -1;
            } else {
                eta = 1;
            }
        }
        break;
```

The η-values are stored in each `Crossing` data structure as the integer member `eta`. These values will be used later in computing the values of entries in the Goeritz matrix for the knot, as well as the "correction factor" for the signature of the knot.

E. **Labelling Crossings**

Crossing labelling is the preliminary step in defining the white regions of our knot projection necessary for constructing the Goeritz matrix. It begins by redefining the knot in terms of line segments between crossings, hereon referred to as "arcs". Each arc is numbered as they are encountered going around the knot according to its orientation. Crossings are then labeled as strings of 4 numbers, the order determined by the incoming arc going under, and then continuing counter-clockwise around the crossing.

We begin by creating the `line_crossing_array`. This array consists of positive integers representing the vertices of the polygonal knot (as we travel along it according to its orientation), and negative numbers representing the crossings, just the opposites of their crossing numbers. When constructing, if we are at some crossing n, we record - in the order they occur - each crossing on the line n to $n+1$, then adding $n+1$ to the array and continuing on. The `line_crossing_array` looks something like

`[..., vertex , first_crossing_on_line, second_crossing_line,..., next_vertex].`

Another function (known as `FixOrder`) fixes the order of the crossings in the `line_crossing_array` to ensure that the crossings are listed in the order they're found travelling with respect to the knot's orientation by checking the t-values of each pair of crossings against each other, and swapping them if need be. Since a crossing splits two lines into four arcs, we define the `arc_array` to be four times the number of crossings.

Psuedocode:

```
//Creating line_crossing_array

line_crossing_array_index = 0;
for(current_vert=0; current_vert<=num_of_verts; current_vert++){
        line_crossing_array[line_crossing_array_index] = current_vert % num_of_verts;
        line_crossing_array_index+1;
        for(current_cross=0; current_cross < num_of_crossings; current_cross++)){
                if(crossing_array[current_cross]->line_1_data[0]==current_vert and
                    crossing_array[current_crossing]->line_1_data[1] == current_vert+1)
                    {
                        line_crossing_array[line_crossing_array_index] =
                            crossing_array[current_crossing]->num;
                        line_crossing_array_index+1;
                        FixOrder();
                } else if (crossing_array[current_cross]->line_2_data[0]==current_vert
                     and crossing_array[current_crossing]->line_2_data[1] ==
                    current_vert+1){
```

```
                    line_crossing_array[line_crossing_array_index] =
                        crossing_array[current_crossing]->num;
                    line_crossing_array_index+1;
                    FixOrder();
            }
        }
}
```

After creating the line_crossing_array, we make an array to record what arcs a crossing occurs on, known as the arc_crossing_array. The arc_crossing_array is created by analyzing the line_crossing_array pairwise, then noting wheter the pair was both positive, both negative, or one of each. Beginning from arcnumber = 1, if the first pair we look at is both negative (that is, both crossings), we increment arcnumber and place it in between the two (seperating the crossings by the arc). If the pair we're looking at is both positive, we just increment arcnumber and add it to the array, treating multiple connected line segments with no crossings as a single arc. Should the pair be one positive, one negative, we must consider the order in which they occur. If the first in the pair is negative (a crossing), we add the crossing to arc_crossing_array, then increment the arc counter and append that to the array as well. If the first number is positive (a vertex), we merely add the negative (crossing) number from the pair to the arc_crossing_array. The final array looks like

[..., arc, crossing, next_arc, crossing, next_arc,...]

Should the arc_crossing_array begin on a positive (arc) number, the last postive(arc) number is changed to match, to account for the cyclical nature of the knot.

Psuedocode:

```
//Creating arc_crossing_array
//we want to start as close to the first crossing as possible, either directly on it
   or right next to it, so we start AFTER vertex 0
for(i =0; i < num_line_crossing_array_elements; i++){
        if(line_crossing_arra[i] !=0){
                start_index =i;
                break;
        }
}

arc_crossing_array[0] = line_crossing_array[start_index];
//if first entry in arc_crossing_array is a crossing
if(arc_array[0] < 0){
        arc_counter =0;
} else {
        arc_counter =1;
}

for(int i = start_index; i < (num_line_crossing_elements - 1); i++){
        if(line_crossing_array[i] < 0){
                if(line_crossing_array[i+1] < 0){
                        arc_counter++;
                        arc_array[arc_array_index] = arc_counter;
                        arc_array[arc_array_index+1] = line_crossing_array[i+1];
                        arc_array_index+=2;
                } else {
                        arc_counter++;
                        arc_array[arc_array_index] = arc_counter;
                        arc_array_index+=1;
                }
        }
        if(line_crossing_array[i] >= 0){
                if(line_crossing_array[i+1] < 0){
                        arc_array[arc_array_index] = line_crossing_array[i+1];
                        arc_array_index+=1;
                }
        }
```

```
} //end of algorithm loop
last_output_index = arc_array_index - 1;
//Correction factor, for when we need first/last arc to be the same
if(line_crossing_array[0] >= 0 && line_crossing_array[1] >= 0 && line_crossing_array[
    num_line_crossing_elements - 2] >= 0 && line_crossing_array[
    num_line_crossing_elements - 2] >= 0){
        arc_array[last_output_index] = arc_array[0];
}
```

Finally comes reading the `arc_crossing_array` to determine the label for each crossing. Since each crossing is touching 4 arcs, it follows that each crossing number must appear exactly twice (being surrounded by two distinct arcs each time). To begin labelling we read into each `Crossing`'s `label` array the numbers that occur on either side of its crossing number in the `arc_crossing_array` in sequential order. So, for example, for the crossing corresponding to -1, if the `arc_crossing_array` contains

`[...,1,-1,2,.....,7,-1,8,...]`.

Then that crossing's label is (for the moment) `[1278]`.

We now need to determine the incoming undercrossing arc. Consider a crossing with label $abcd$. Back when we found this crossing, we recorded the first number of the vertex of the undercrossing line, and the first number of the vertex of the overcrossing line as `lowvertex` and `highvertex`, respectively. If `lowvertext <highvertex`, then a is the incoming undercrossing arc. Otherwise, c is the incoming undercrossing. We take into consideration the crossing's `type` and `eta_value`, and shuffle the `label` we found earlier according to the following graphics to determine the actual `label`.

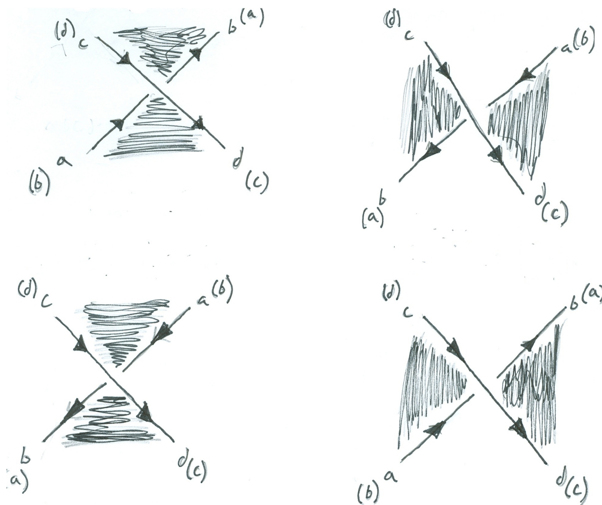


Figure 9: Crossings with a as the incoming undercrossing arc. Top Left: Type I, $\eta = -1$, Label $adbc$. Top Right: Type I, $\eta = 1$, Label $acbd$. Bottom Left: Type II, $\eta = -1$, Label $acbd$. Bottom Right: Type II, $\eta = 1$, Label $adbc$.

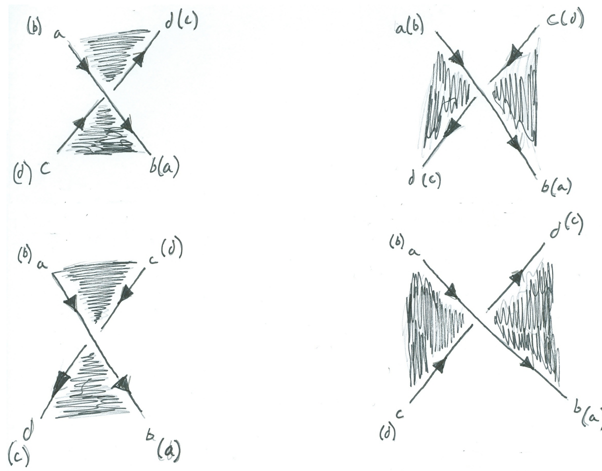


Figure 10: Crossings with c as the incoming undercrossing arc. Top Left: Type I, $\eta = -1$, Label *cbda*. Top Right: Type I, $\eta = 1$, Label *cabd*. Bottom Left: Type II, $\eta = -1$, Label *cabd*. Bottom Right: Type II, $\eta = 1$, Label *cdba*.

It is important to note that the 4 integers comprising a crossing's label are not necessarily unique. In the case of loops which can (ordinarily) be removed via a Reidemeister I move, we have crossings with repeated integers in their labels, the possible cases being pictured below.

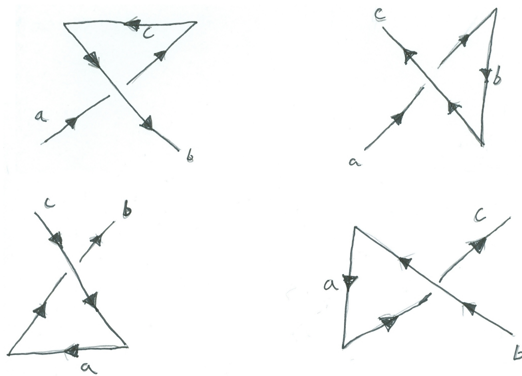


Figure 11: Top Left: Label *abcc*. Top Right: Label *abbc*. Bottom Left: Label *aabc*. Bottom Right: Label *abca*.

Psuedocode:

```
//Create Labels
//create "preliminary" label of straight arc numbers, without regard to order
for(current_crossing = 0; current_crossing < num_of_crossings; current_crossing++){
        crossing_encounter_num =0;
        first_touching_arc_index =0;
        second_touching_arc_index =0;
        while(crossing_encounter_num < 2){
                for(arc_array_index = 0; arc_array_index < last_arc_index;
                    arc_array_index++){
                if(arc_array[arc_array_index] == crossing_array[current_crossing]){
                        crossing_encounter_num++;

                        //make sure we don't "fall off" the arc_array
                        if(arc_array_index -1 >= 0 && arc_array_index + 1 <=
                            last_arc_index){
                                //set in label array the arcs on either side of the
                                    crossing
                                crossing_array[current_crossing]->label[
                                    first_touching_arc_index] = arc_array[
```

```
                                        arc_array_index - 1];
                                crossing_array[current_crossing]->label[
                                        second_touching_arc] = arc_array[arc_array_index +
                                        1];
                                first_touching_arc_index +=2;
                                second_touching_arc_index +=2;
                        } else if(arc_array_index - 1 < 0){
                                //case of crossing as first entry
                                crossing_array[current_crossing]->label[
                                        first_touching_arc_index] = arc_array[
                                        last_arc_index];
                                crossing_array[current_crossing]->label[
                                        second_touching_arc_index] = arc_array[
                                        arc_array_index+1];
                                first_touching_arc_index +=2;
                                second_toucning_arc_index +=2;
                        } else if (arc_array_index + 1 > last_arc_index){
                                //crossing is last in array, need to loop back to
                                        first arc
                                crossing_array[current_crossing]->label[
                                        first_touching_arc_index = arc_array[
                                        arc_array_index -1];
                                crossing_array[current_crossing]->label[
                                        second_touching_arc_index] = 1;
                        } //end of if-else block
                } //end of for loop in while loop
        } //end of while loop
}//end of for loop containing while loop

//"Shuffle" preliminary labels into proper crossing labels
for(current_crossing =0; current_crossing < num_of_crossings; current_crossing++){
        //put "preliminary" label into temp_label array for shuffling
        for(j = 0; j < 4; j++){
                temp_label[j] = crossing_array[current_crossing]->label[j];
        }
        //"if a incoming under"
        if(crossing_array[current_crossing]->lowest_vertex < crossing_array[
            current_crossing]->highest_vertesx){
                crossing_array[current_crossing]->label[0] = temp_label[0];
                if((crossing_array[current_crossing]->type == 1 &&
                crossing_array[current_crossing]->eta == -1) ||
                (crossing_array[current_crossing]->type == 2 &&
                crossing_array[current_crossing]->eta==1)){
                        crossing_array[current_crossing]->label[1] = temp_label[3];
                        crossing_array[current_crossing]->label[2] = temp_label[1];
                        crossing_array[current_crossing]->label[3] = temp_label[2];
                } else {
                        crossing_array[current_crossing]->label[1] = temp_label[2];
                        crossing_array[current_crossing]->label[2] = temp_label[1];
                        crossing_array[current_crossing]->label[3] = temp_label[3];
                }
        } else {//"c incoming under"
                crossing_array[current_crossing]->label[0] = temp_label[2];
                if((crossing_array[current_crossing]->type == 1 && crossing_array[
                    current_crossing]->eta==-1)
                || (crossing_array[current_crossing]->type == 2 && crossing_array[
                    current_crossing]->eta==1)){
                        crossing_array[current_crossing]->label[1] = temp_label[1];
                        crossing_array[current_crossing]->label[2] = temp_label[3];
                        crossing_array[current_crossing]->label[3] = temp_label[0];
                } else {
                        crossing_array[current_crossing]->label[1] = temp_label[0];
```

```
                    crossing_array[current_crossing]->label[2] = temp_label[3];
                    crossing_array[current_crossing]->label[3] = temp_label[1];
                }
        }
}
```

F. **Finding Regions**

A region is defined as the set of arcs surrounding a shaded or unshaded part of the colored knot diagram. We find the regions of our knot by treating labels similar (though not entirely analogous) to permutations in cycle notations. Beginning with C_{-1}, we take the first two arcs of its label as the first two arcss in the label of the first region R_0. We'll call these a and b. We search the remaining crossing labels for another occurrence of b. When we find it in some crossing C_k we look at the next arc in that label following b (if b is the fourth arc in the label then we look at the first), we'll call this c. We record this as the third arc in R_0's label. We then repeat the process looking for c in all other crossings besides C_k. Should this action ever return the first arc in the region's label, we do not record it and terminate the searching loop, completing the region's label. To start the label for the next region R_1, we use the third and fourth arc's of C_{-1}'s label, under one condition. Since regions can only exist to the "left" or "right" of an arc, it follows that any given arc number can occur only twice over the set of all region labels. Thus when beginning the search for the next region R_1, we must make sure that the arc we're starting on hasn't already occured twice among all our other region labels. Concluding finding R_1, we start the next region with the first two arcs from C_{-2}'s label (again, provided they haven't already occurred twice), and so forth.

Example 1. If we have crossings with labels

$$1726,\ 5362,\ 3841,\ 7485,$$

Then our resulting regions have labels

$$174, 725, 26, 613, 538, 84.$$

Psuedocode:

```
//Finding Regions
int current_region=0;
for(int current_crossing=0; current_crossing < total_crossings; current_crossing++){
        for(int i; i < label_size; i++){
        int region_index=0;
                if(occurrences(cross_array[current_crossing]->label[i]) < 2){
                        region_array[current_region]->label[region_index] =
                            cross_array[current_crossing]->label[i];
                        region_index++;
                        region_array[current_region]->label[region_index] =
                            cross_array[current_crossing]->label[(i+1)%label_size];

                        for(int k = 1; k <= 3; k++){
                                for(int j =0; j < label_size; j++){
                                        if(cross_array[(current_crossing + k)%CROSSNUM
                                            ]->label[j] == region_array[current_region
                                            ]->label[region_index]){
                                                if(cross_array[(current_crossing+k)%
                                                    CROSSNUM]->label[(j+1)%label_size]
                                                    != region_array[current_region]->
                                                    label[0]){
                                                region_index++;
                                                region_array[current_region]->label[
                                                    region_index] = cross_array[(
                                                    current_crossing+k)%CROSSNUM]->
                                                    label[(j+1)%label_size];
                                                break;
                                                } else {
                                                break;
                                                }
                                        }
                                }

                        }
```

```
                    current_region++;
            }
        }
}
```

Regions are stored in an array of structures `Region_Struct`, containing data for both their label and an integer indicating whether or not the region is shaded (1 for shaded, 0 for unshaded). We must allocate space for the `label` integer arrays in the `Region_Struct` structures at runtime, based on the number of crossings we have. Region memory is allocated at 4 regions to each crossing, as that is the maximum number of regions a given crossing can be incident to. Memory is allocated using the "struct hack" trick, now known formally as "flexible array member allocation". The code for which follows.

```
//dynamic allocation of array data members
typedef struct Region_Struct{
        int shade;
        int label[];
        //flexible array members must not be the only data members
        //also must be declared las
} *Region;

//allocating memory for region structs
for(int i=0; i<NUM_OF_REGIONS; i++){
        region_array[i] = (struct Region_Struct*)malloc(sizeof(struct Region_Struct) +
            LABEL_SIZE*sizeof(int));
        //where LABEL_SIZE = number of crossings
        //Adding LABEL_SIZE*sizeof(int) sets size of label[] array
}
```

G. **Region Shading**

Region shading is again determined from crossing data, primarily by the η-values and the label. Each crossing is analyzed in turn, and we find pairs of integers in it's label that correspond to shaded areas of the knot diagram as in the graphic below.

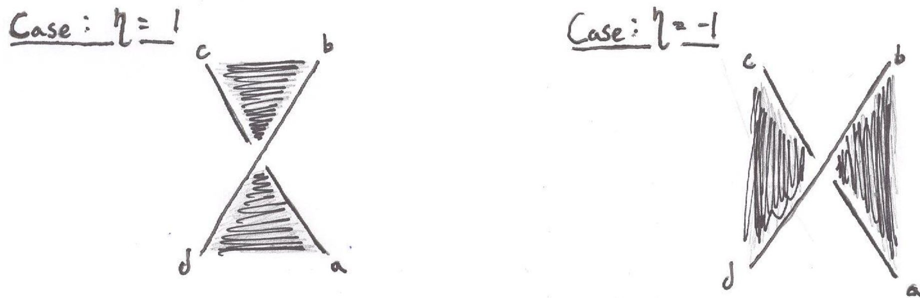


Figure 12: Left: When $\eta = -1$, we have that label integer pairs cb and ad imply any region label containing those pairs is shaded. Right: For $\eta = 1$, we have that label integer pairs ab and cd imply shading.

Having found a pair of integers that defines a shaded area, we examine the labels of each crossing for a matching pair (mindful that the last element in a label is considered adjacent to the first). In the existence of a matching pair, we set that Region's shade value to 1, and it is considered shaded. A region that has no matching pairs with any of the shaded pairs checked has its shading value left at zero.
Following checking region shading, a function searches through the region array and notes the positions of all unshaded regions, which are then stored to a seperate array, to be used in constructing the Goeritz matrix.

Psuedocode:

```
//Region Shading
for(int i=0; i<NUM_OF_CROSSINGS; i++){
        if(crossing_array[i]->eta == 1){
                shade_pair1[2]={crossing_array[i]->label[0], crossing_array->label
                    [3]};
```

```
                shade_pair2[2]={crossing_array[i]->label[1], crossing_array->label
                    [2]};
        } else{
                shade_pair1[2]={crossing_array[i]->label[0], crossing_array->label
                    [1]};
                shade_pair2[2]={crossing_array[i]->label[2], crossing_array->label
                    [3]};
        }
        for(int j=0; j<NUM_OF_REGIONS; j++){
                //search pairwise
                for(int label_index; label_index < 4; label_index++){
                        if(region_array[j]->label[label_index]==shade_pair1[0] &&
                            region_array[j]->label[(label_index+1)%4]==shadepair1[1]){
                                region_array[j]->shade=1;
                        } else if(region_array[j]->label[label_index]==shade_pair1[1]
                            && region_array[j]->label[(label_index+1)%4]==shadepair1
                            [0]){
                                region_array[j]->shade=1;
                        } else if(region_array[j]->label[label_index]==shade_pair2[0]
                            && region_array[j]->label[(label_index+1)%4]==shadepair2
                            [1]){
                                region_array[j]->shade=1;
                        } else if(region_array[j]->label[label_index]==shade_pair2[1]
                            && region_array[j]->label[(label_index+1)%4]==shadepair2
                            [0]){
                                region_array[j]->shade=1;
                        }
                }
        }
}

//assign unshaded regions to unshaded array
Region* unshaded_regions[NUM_OF_REGIONS];
int unshade_index=0;
for(int i; i<NUM_OF_REGIONS; i++){
        if(region_array[i]->shade==0){
                //point to unshaded region
                unshaded_regions[unshade_index]=&region_array[i];
                unshade_index++;
        }
}
```

H. **Matrix Operations**

The Goeritz matrix is stored in program memory as a 1 dimensional array with integral offset. Each "row" of the matrix - sequence of array positions corresponding to a row of the Goeritz matrix - is filled in according to the rule established under "Background and Motivation". The matrix is then converted into column major order (in a fashion similar to matrix transposition) for use with the LAPACK library of functions to determine the determinant and eigenvalues of the matrix, using the `zgetrf` and `zgeev` functions, respectively.

The variable storing the determinant of the matrix has it's absolute value taken to give the determinant of the knot.

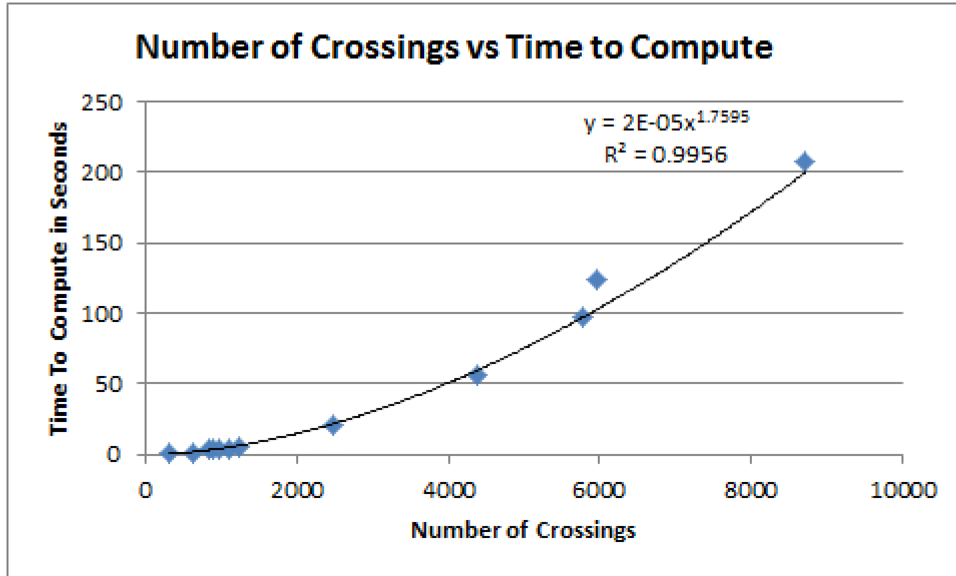
Psuedocode:

```
//find matrix dimensions
int dim_count=0;
while(region_array[dim_count]!=NULL){
        dim_count++;
}

//Assign unshaded regions eta values to matrix
for(int i=0; i<dim_count; i++){
        for(int j=0; j<NUM_OF_REGIONS; j++){
                two_d_goeritz_matrix[i][j] = -(region_array[i]->total_eta +
```

```
                        region_array[j]->total_eta)
                }
        }
        //set diagonal entries of two_d array to fit formula
        for(int i; i<dim_count; i++){
                two_d_goeritz_matrix[i][i]=0; //reinitialize
                for(int j=0; j<dim_count; j++){
                        if(j!=i){ //only use values not on diagonal
                                two_d_goeritz_matrix += (-two_d_goeritz_matrix[i][j]);
                        }
                }
        }

        //use offset to convert 2-dimensional array to 1-dimensional array, delete 0th row and
            column in the process
        //we flip i, j, for "column major" storage for LAPACK
        float goeritz[dim_count*dim_count];
        int goeritz_index=0;
        for(int i=1; i < dim_count; i++){
                for(int j=1; j<dim_count; j++){
                        goeritz[goeritz_index]=two_d_goeritz_matrix[j][i];
                }
        }

        //LAPACK functions, see LAPACK documentation for details
        float determinant = zgetrf(goeritz_matrix);
        if(determinant < 0){
                determinant = -determinant;
        }

        float eigenvalues[] = zgeev(goeritz_matrix);

        int poscount, negcount;
        for(int i=0; i<NUM_EIGENVALUES; i++){
                if(eigenvalues[i] > 0){
                        poscount++;
                } else if(eigenvalues[i]<0){
                        negcount++;
        }

        int sig = poscount - negcount;

        int mu;
        for(int i =0; i<NUM_OF_CROSSINGS; i++){
                if(crossing_array[i]->type == 2){
                        mu += crossing_array[i]->eta;
        }

        sig -= mu;
```

I. **Performance**

Testing shows that as the number of data points tested increases, the time (in seconds) to compute crossing data increases exponentially according to the following fit line.



The increase in crossings (and thus computation time) is directly correlated with the number of vertices in the vertex data input. The nature of the deeply nested `for` loops in finding crossings and evaluating their types, labels, and η-values makes them difficult to parallelize, but there is afforded many opportunities for parallelization with the actual matrix computations for determinant and signature. One option is to load the matrix off the CPU to the GPU via CUDA. CUDA has a BLAS-based library (CUBLAS) for dealing with matrices and other linear algebra operations.

Possiblities for use of the program in the future involve studying mutations of the knots generated, by the random walks, done via manipulation of the crossing data. Changing crossings from over to under, or removing them entirely (in some cases) should create drastically new knot types from the original. Utilizing this knowledge, then, we can begin analysis of physical systems such as the knotting/entangling of protein and polymer chains, or even decompose them into simpler "prime" knots by looking at connected sums.