Algebraic Invariants of Orlik-Solomon Algebras: Some New Examples

Cahmlo Olive
Lawrence University
Appleton, WI 54912

cahmlo@hotmail.com

This paper was written as part of the 2001 REU program at NAU, under the supervision of Dr. Mike Falk.

I. Background

Matroids and their Representations

The most basic objects we deal with are matroids. It is from these that we build up the main structure of interest, the Orlik – Solomon (OS) algebra. So first we look at definitions and examples of matroids and their representations.

Definition 1.1: *Let S be a finite collection of points and g a non empty family of subsets of S satisfying*

i. If $A \subseteq B$ and $B \in g$ then $A \in g$ and

ii. If $A, B \in g$ and $|A| < |B|$ then there exists $b \in B - A$ with $A \cup \{b\} \in g$.

*The pair M = (S, g) is a **matroid**. The members of g are called **independent** and all other subsets are called **dependent*** [3, 4].

An alternative way of describing matroids is through their dependent sets. For any matroid we say $a \in S$ **depends** on $A \subseteq S$ if either $a \in A$ or $B \cup \{a\}$ is dependent for some independent subset B of A. If A is a subset of S, the set of all points dependent on A is the **closure** $\underline{A}$ of A. If $\underline{A} = A$ then we say A is a **flat** of S. If a flat F contains a flat G and for another flat H, $F \supseteq H \supseteq G$ implies F = H or H = G then say F **covers** G [3].

Theorem 1.2: Let S be a finite collection of points and F a family of subsets satisfying:

F1. $S \in F$

F2. $A, B \in F$ implies $A \cap B \in F$

F3. for any $A \in F$, $S - A$ is partitioned by members of F that cover A (i.e. every $s \notin A$ is in one and only one flat covering A)

Then F is the family of flats of the matroid M = (S, g) where $A \in g$ iff. every proper subset of A is contained in some flat not containing A [3].

Matroids can be represented pictorially in different ways. In particular we will make use of arrangements to represent matroids throughout the rest of the paper.

Definition 1.3: *Let **K** be a field and V_K a vector space of dimension l. A **hyperplane** H in V_K is a vector subspace of dimension l – 1. An **arrangement** $A_K = (A_K, V_K)$ is a finite collection of hyperplanes in V_K* [1, 2].

Example 1.4: If we consider $\mathbf{R}^3$ as our vector space then the collection $S \equiv \{x + 3z, x + z, x - z, x - 3z, x + y, y + 3z, y + z, y - z, y - 3z, x - y, z\}$, of planes through the origin in $\mathbf{R}^3$, is an arrangement. (Notice that for the rest of the paper we will restrict our attention to the projective space associated with $\mathbf{R}^3$, or the collection of all lines through the origin in $\mathbf{R}^3$. We call lines through the origin projective points, and planes through the origin projective lines.)

Two more important ideas in the theory of arrangements are the set of all intersections of elements of an arrangement and the rank function. The set of all intersections of an arrangement A is denoted L = L(A). The rank function is simply defined as r(X) = codim(X). Thus in example 1.4, r($\mathbf{R}^3$) = 0 and r({z}) = 1 and r({z} $\cap$ {x + y}) = 2.

Notice that now, given an arrangement (of planes through the origin in $\mathbf{R}^3$), we can associate a matroid with it. Let G be an arrangement of n hyperplanes. Let S ≡ {H_1, ..., H_p} be a sub-collection of these hyperplanes, and let |S| = p. S is independent if r($\cap$S) = p and dependent if r($\cap$S) < p. Then if g ≡ {S . G | S is independent}, we can show that (G, g) is a matroid. Call this matroid the matroid associated with the arrangement G.

Example 1.5: Let G ≡ {a, b, c, d, e, f, g, h, i, j, k}, whose points correspond to the hyperplanes in example 1.4, (say 'a' corresponds to x + 3z and 'k' corresponds to z), Also consider the family of subsets of G, g ≡ {all singletons, all doubles, all collections of three hyperplanes not intersecting in a projective point}. Then the matroid M = (G, g) corresponding to the arrangement in example 1.4.

We can render arrangements more simply in two ways. The first way is to consider cross-sections of a particular arrangement. As we are restricting our attention to the projective space associated with $\mathbf{R}^3$ any plane not through the origin will intersect all hyperplanes not parallel to it. This works because given one plane not through the origin, only one plane through the origin runs parallel to it. One pictorial representation of the arrangement in example 1.4 is Fig 1.a.

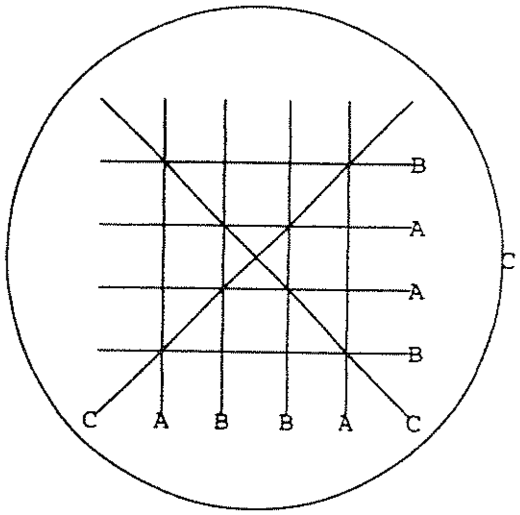

FIG 1.A(picture of the 11 line picture).

The crossection is the intersection of z = 1 with the arrangement. The circle represents the hyperplane z = 0. As z = 0 runs parallel to z = 1, and thus never punctures z = 1, it must receive special notation. Notice that points of intersection in the picture represent lines of intersection in the arrangement. This also means that as {x + 3z, x + z, x − z, x − 3z, z} all meet in a line in the arrangement, they must meet in a point in the figure. This point is on the circle or 'at infinity' in the direction of the parallel lines. In general, parallel 'lines' meet at infinity [8].

ply is to consider a cross-section of the lines normal to the hyperplanes in a given arrangement. Notice that if n projective lines intersect in a projective point their normal vectors are co-planar. and if we draw projective points in the direction of these normal vectors and take a cross-section of these projective points we get three points that are co-linear. These are the points we use.

Example 1.6: Let <a, b, c> represent a projective point in the direction of the vector <a, b, c>. The projective points normal to the hyperplanes of the arrangement in example 1.4 are {<1, 0, 3>; <1, 0, 1>; <1, 0, −1>; <1, 0, −3>; <1, 1, 0>; <0, 1, 3>; <0, 1, 1>; <0, 1, −1>; <0, 1, −3>; <1, −1, 0>; <0, 0, 1>}. The cross-section of $z = 1$ with the arrangement is the set of points {(1/3, 0, 1); (1, 0, 1); (−1, 0, 1); (−1/3, 0, 1); (1, 1, 0); (0, 1/3, 1); (0, 1, 1); (0, −1, 1); (0, −1/3, 1); (1, −1, 0); (0, 0, 1)}. We can then plot these as points in the copy of $\mathbf{R}^2$, $z = 1$.

Notice that if the 'z' coordinate of a particular point is 1 it is a regular point and if the 'z' coordinate is 0 then it is a point at infinity in the direction of the coefficient of y divided by the coefficient of x. This convention is needed because just as projective lines may not puncture a particular cross-section, projective points may not either. This is how we arrived at the picture for Fig 1.b.

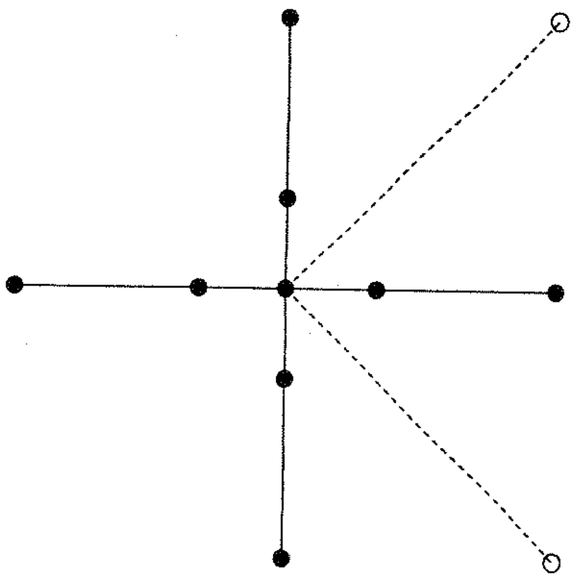

FIG 1.B(picture of the 11 point example)

The dotted lines represent points at infinity in the direction that they point. In other words a line through a point will have one extra point on it if it is parallel to a dotted line. Notice that I have not drawn in the lines that would intersect with a point at infinity.

The Orlik – Solomon algebra [5,6,9]

Here we build up the main structure of interest, the Orlik – Solomon algebra of a matroid, and discuss the main problem.

Let G be a simple matroid of with a ground set $[n] = \{1, 2, \ldots, n\}$. Let $\xi = \Lambda(e_1, \ldots, e_n)$ be the free graded exterior algebra [7] on elements e_i corresponding to the points of G (assume the ground field is $\mathbf{C}$ here). Define $\partial: \xi^p \to \xi^{p-1}$ by $\partial(e_1\wedge\ldots\wedge e_n) = \Sigma_{k=1} (-1)^{k-1} e_{i1}\wedge\ldots\wedge\underline{e_{ik}}\wedge\ldots\wedge e_{ip}$ where the underlined factor represents an omitted factor. Note that if $S = (i_1, \ldots, i_p)$ is an ordered p tuple then $e_{i1}\wedge\ldots\wedge e_{ip}$ is denoted e_S.

Definition 1.7: *The Orlik – Solomon (OS) algebra A = A(G) of G is the quotient ξ/I where I is the ideal $\{\partial e_S \mid S$ is dependent$\}$* [5, 6].

One of the main objectives regarding OS algebras is classification, up to isomorphism of graded algebras. There are several approaches. In this paper we explore specific invariants called resonance varieties $R_p(A)$. More specifically we look at the first resonance variety $R_1(A)$. The first resonance variety provides information about the combinatorial features of G; it tells us how to check whether or not a matroid has certain special characteristics. We say then that it supports resonant weight. If a matroid G' supports resonant weight and a matroid G contains G' then $R_1(A)$ must contain an element corresponding to G' and a copy of G', or another representation of the OS algebra constructed from G' must appear in any representation G. The goal is to find as many matroids supporting resonant weight as possible or to find a consistent method of generating them [5, 6, 8]. In this paper we show seven new examples contrasting the original six. We also present a potential method of finding new examples more consistently.

Resonance Varieties

Here we define the term resonance variety, discuss the main theorem related to $R_1(A)$ in connection with OS algebras and show how to use $R_1(A)$ to help distinguish between non – isomorphic OS algebras.

By cohomology [7] of A = A(G) we mean the cohomology of the complex (A, d_λ) where d_λ is the degree one mapping defined by left multiplication by a fixed element $a_\lambda = \Sigma^n{}_{i=1} \lambda_i a_i$ in A^1. As d_λ squares to zero,

$$0 \to A^0 \to A^1 \to \ldots \to A^{l-1} \to 0$$

is a co-chain complex. Note that if A is the set of hyperplanes P_i of the arrangement corresponding to G and $C = \mathbf{C}^l \setminus \cup_{H\in A} P_i$ then the cohomology $H^*(A_1, d_\lambda)$ is isomorphic to the cohomology of C. [5, 6].

Definition 1.8: *The p^{th} resonance variety $R_p(A)$ is the set $\{\lambda \in \mathbf{C}^n \mid H^p(A, a_\lambda) \neq 0\}$* [5].

$R_p(A)$ is, up to ambient linear isomorphism, an invariant of A. Moreover, $R_1(A)$ is known to contain combinatorial information about the underlying matroid G. A theorem explains how to use $R_1(A)$ to retrieve this information [5, 6, 9]. First, we need a definition.

Definition 1.9: *A partition Π of $[n]$ is a **neighborly partition** of G if $|\pi \cap X| \geq |X| - 1$ implies $X \subseteq \pi$ for each block $\pi \in \Pi$ and each flat $X \in L(G)$ of rank two* [5]. *Notice that by a neighborly partition of an arrangement we mean a neighborly partition of the matroid associated with that arrangement.*

Example 1.10: If Π consists of three blocks $\pi_A \equiv \{x + z, x - z, y + 3z, y - 3z\}$, $\pi_B \equiv \{ y + z, y - z, x + 3z, x - 3z\}$ and $\pi_C \equiv \{x + y, x - y, z\}$, then Π is a neighborly partition of the arrangement in example 1.4. To see this more clearly consider the flat $X_1 \equiv \{x + 3z, y + z\} \in L(G)$ of rank two. Notice $|\pi_B \cap X_1| \geq |X_1| - 1$ and this implies that $X_1 \subseteq \pi_B$. Also notice that if we consider the flat of rank two $X_2 \equiv \{x + 3z, x - y, y + 3z\} \in L(G)$ we get $X_2 \not\subset \pi_A$ and this implies $|\pi_A \cap X_2| < |X_2| - 1$.

There is a simple algorithm to find a neighborly partition, not necessarily non-trivial, of an arrangement of projective lines or a simple matroid. Consider arrangements. Start by coloring each hyperplane differently, i.e. Π consists of n blocks corresponding to the n hyperplanes at first. Next, starting with $m = 2$ and continuing until $m = n$, if m 'lines' intersect in a point and $m - 1$ are colored the same then color all m lines the same. This implies that if 2 lines intersect at a point then they automatically must be colored the same. Also if a color is changed from say C to B then everything colored C before must be changed to B as well.

There may be different neighborly partitions of a single arrangement or matroid. For instance consider the arrangement of 9 hyperplanes $A \equiv \{x + z, x - z, 2x + y + z, 2x + y - z, y + z, y - z, 2x - y - z, 2x + y - z, z\}$. The blocks $\pi_A \equiv \{x + z, 2x - y - z, 2x + y - z\}$, $\pi_B \equiv \{x - z, 2x + y + z, 2x - y + z\}$ $\pi_C \equiv \{y + z, y - z, z\}$ form a neighborly partition of A, which is seen in Fig 1.c. The blocks $\pi_A \equiv \{x + z, 2x - y - z, 2x + y - z\}$, $\pi_B \equiv \{x - z, 2x + y + z, 2x - y + z\}$, $\pi_C \equiv \{y + z\}$, $\pi_D \equiv \{y - z\}$ and $\pi_E \equiv \{z\}$ form a neighborly partition of A as well [8]. This partition can be seen in Fig 1.d. The algorithm will find the second of these partitions because is finds the partition with the most blocks.

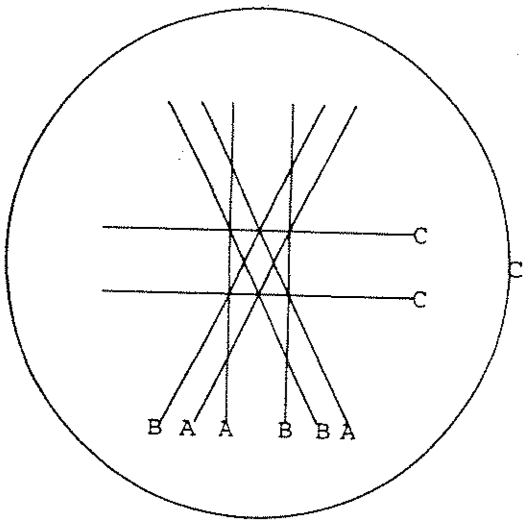

FIG 1.C(pappus arrangement line picture first coloring fewer colors)

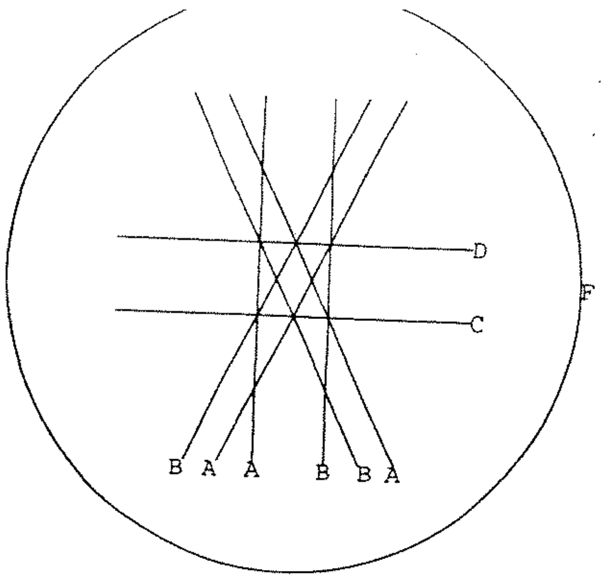

FIG 1.D (second coloring)

Theorem 1.11: $\lambda \in R_1(A(G))$ if and only if there exists a submatroid G' of G and a neighborly partition Π of G' such that:

i. $\lambda \in L_\Pi$ and

ii. there exists $\mu \in L_\Pi$ not proportional to λ such that $\langle\lambda, \mu\rangle_\Pi = 0$ [6].

Example 1.12: Let G be the matroid in example 1.5 and consider the submatroid of G that is itself, i.e. G' = G. We already have a neighborly partition of G from example 1.10. The next step is to look at L_Π. Construct a matrix L with 11 columns, a – k corresponding to the 11 points of the matroid, in the following manner. For each multicolored flat add a row, where the column gets a 1 if the corresponding point is in the flat and a 0 otherwise. For example the flat $F \equiv \{a, f, j\}$ is multicolored so the columns a, f, and j get 1's and the others get 0. There will also be a row of all 1's corresponding to whole set a through k. Then the nullspace of this matrix is L_Π. Notice that L_Π has dimension 2 and two vectors that form a basis for L_Π are $\lambda = (-1\ 0\ 0\ -1\ 1\ 0\ -1\ -1\ 0\ 1\ 2)$ and $\mu = (-1\ 1\ 1\ -1\ 0\ 1\ -1\ -1\ 1\ 0\ 0)$. These two vectors are not parallel. So we can look at the 2 x 2 determinants formed from pairs of columns in λ and μ, corresponding to points in the same block of Π, and are all 0. For instance we would look at the determinant formed by the columns a and d of each vector because a and d are both in π_A. The 2 x 2 determinants being 0 tells us that $\langle\lambda, \mu\rangle_\Pi = 0$ and therefore λ or $\mu \in R_1(A(G))$. Furthermore, if a matroid M contains a copy of G inside it, $R_1(A(M))$ will contain an element corresponding to G.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

FIG 1.E (the matrix L)

```
J -1  0  0  -1  1  0  -1  -1  0  1  2
  -1  1  1  -1  0  1  -1  -1  1  0  0 N
```

FIG 1.F (L_Π)

Theorem 1.11 suggests the following algorithm for checking whether a matroid supports resonant weight like example 1.5. Let G be the matroid under consideration.

Step 1: Find a neighborly partition of G. If the partition is non – trivial, i.e. there is not only one block in Π, then look at L_Π.
Step 2: If $\dim(L_\Pi) \geq 2$ then look for two elements in L_Π that are not parallel.
Step 3: Look at the two by two determinants formed from columns of the two non – parallel vectors in L_Π that are in the same block of Π. If these are all 0 then accept G as a new example.

Now we will look at an example of how to use this information to distinguish between non-isomorphic OS algebras. Note that Fig 1.g is an example of a matroid that supports resonant weight, it passes all three steps. The question is whether Fig 1.h and Fig 1.i have the same OS algebras. If they were the same they would have to have the same local and non-local components. The local components of each are Fig 1.j 6 times and Fig 1.k one time. But Fig 1.h has five copies of Fig 1.g and Fig 1.i has only four copies of Fig 1.g so we know that they must have non-isomorphic OS algebras.

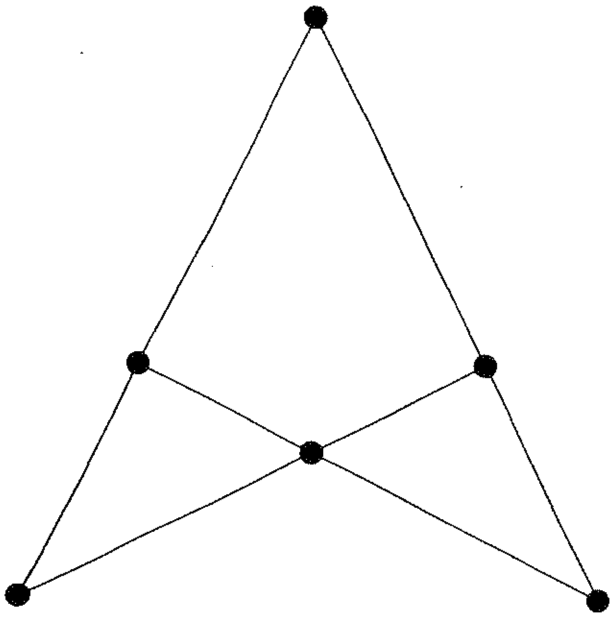
FIG 1.G(braid dot picture)

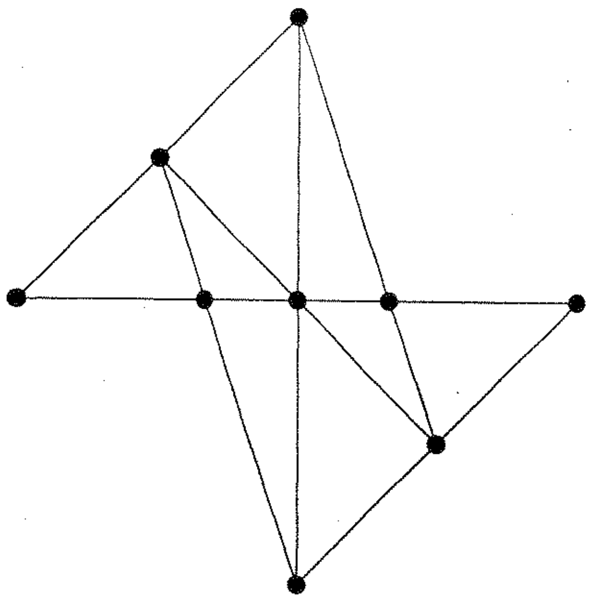
FIG 1.H(one of the pictures to be considered)

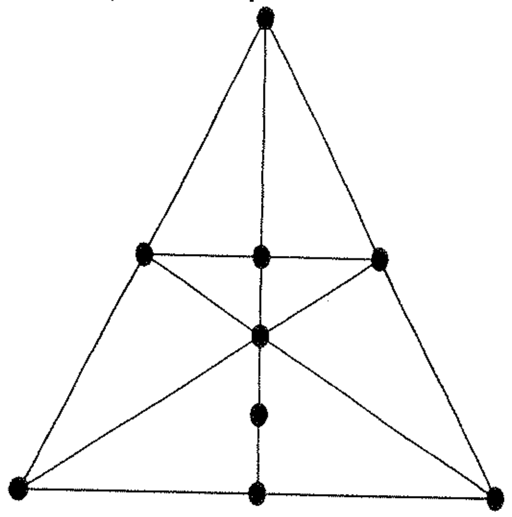
FIG 1.I(the other)

Old Examples Over **C** [5, 6, 8].

Here we quickly run through some of the original examples of matroids that support resonant weights over **C**.

Example 2.1: The smallest known example is that of the braid arrangement of rank three. In one representation the arrangement consists of the set $G \equiv \{x + z, x - z, x + y, y + z, y - z, z\}$. A neighborly partition of this arrangement is $\pi_A \equiv \{x + z, y - z\}$, $\pi_B \equiv \{x - z, y + z\}$ and $\pi_C \equiv \{x + y, z\}$

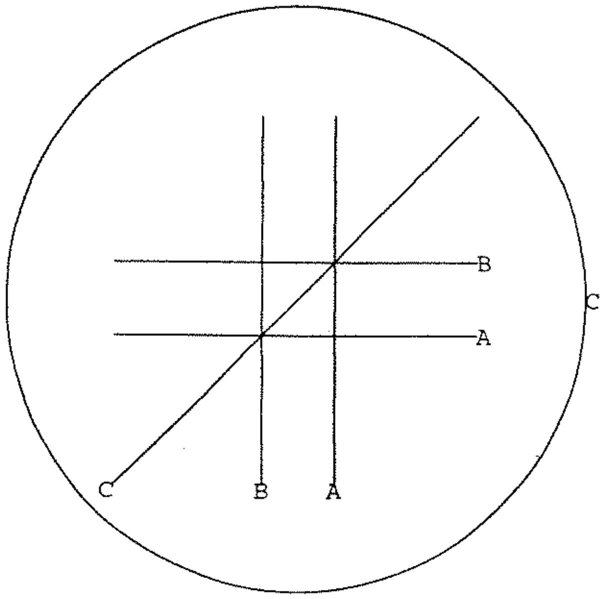

FIG 2.A (line picture of braid arrangement)

Example 2.2: A larger example is the arrangement corresponding to the Pappus matroid. In one representation its nine hyperplanes are $\{x + z, x - z, 2x + y + z, 2x + y - z, y + z, y - z, 2x - y - z, 2x - y + z, z\}$. A neighborly partition of this arrangement is $\pi_A \equiv \{x + z, 2x + y - z, 2x - y - z\}$ $\pi_B \equiv \{x - z, 2x + y + z, 2x - y + z\}$ and $\pi_C \equiv \{y + z, y - z, z\}$.

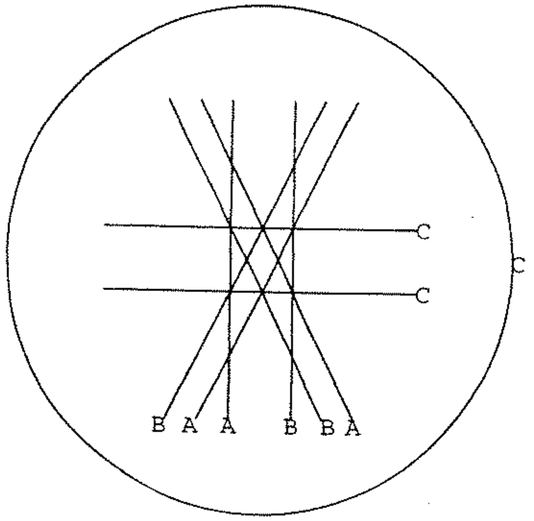

FIG 2.B(the picture with fewer colors)

Example 2.3: Another arrangement with nine hyperplanes is the arrangement corresponding to the symmetries of the cube. The arrangement is the collection $G \equiv \{x + z, x, x - z, x + y, y + z, y, y - z, x - y, z\}$. A neighborly partition is $\pi_A \equiv \{x + z, x - z, y\}$, $\pi_B \equiv \{x, y + z, y - z\}$ and $\pi_C \equiv \{x + y, x - y, z\}$.

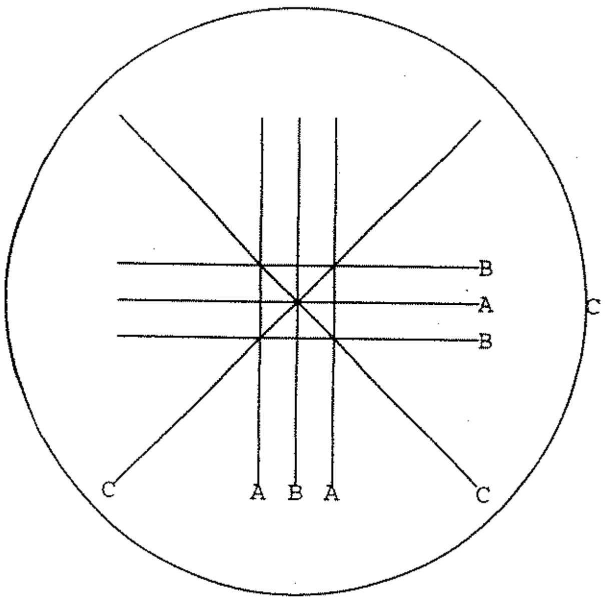

FIG 2.C

New Examples Over **C**

Here we show all six new examples of matroids that support resonant weight over **C**.

Example 2.4: The first example that we found was found by adding the projective line $x = 0$ to example 2.2 as the second point in the arrangement. It can either belong to π_A or to a new block π_D. But notice that in L_Π for this example the second column, corresponding to $x = 0$, is a column of zeros and thus is not really essential to the arrangement. This told us that we needed to modify the algorithm for checking whether a matroid supported resonant weight [8]. We have to add a step after step three that says to check for columns of all zeros, throw out the points or lines corresponding to those columns and then consider the resulting matroid. For this example the resulting arrangement is just example 2.2 again.

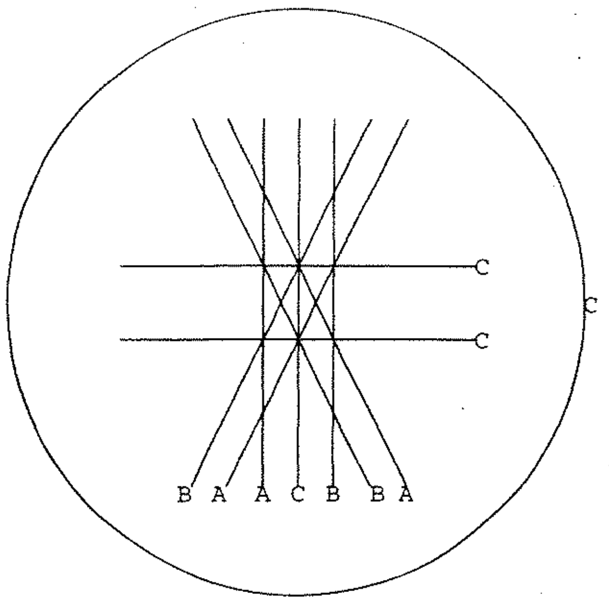

FIG 2.D

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

FIG 2.E

$$\begin{pmatrix} -1 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 1 \\ -1 & 0 & 1 & 1 & -1 & 0 & 0 & -1 & 1 & 0 \end{pmatrix}$$

FIG 2.F

Example 2.5: Another arrangement with nine projective lines is the collection G ≡ {x + z, x, x − z, 2x + y − z, x + y, y + z, y − z, x − y, 2x − y + z}. A neighborly partition for this arrangement is π_A ≡ {x + z, 2x + y − z, x − y}, π_B ≡ {x − z, x + y, 2x − y + z} and π_C ≡ {x, y − z, y + z}.

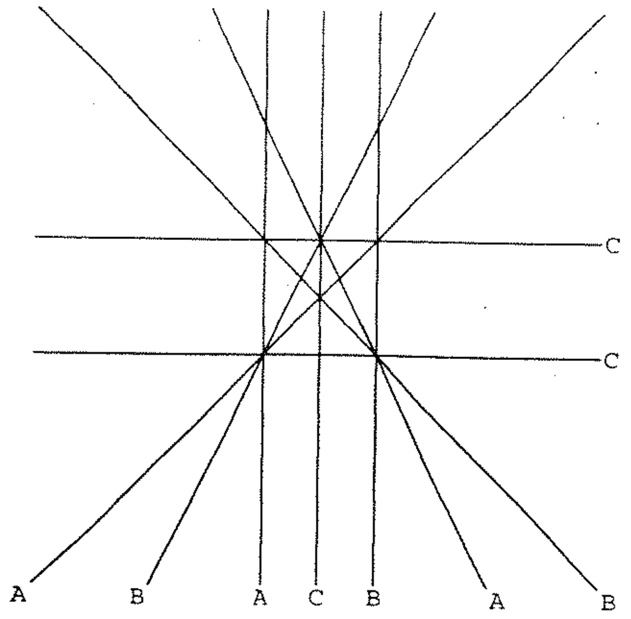

FIG 2.G

Example 2.6 An even larger example with eleven projective lines is the collection G ≡ {x + 3z, x + z, x − z, x − 3z, x + y, y − 3z, y − z, y + z, y + 3z, x − y, z}. A neighborly partition of this arrangement is π_A ≡ {x + 3z, x − 3z, y − z, y + z}, π_B ≡ {x + z, x − z, y − 3z, y + 3z} and π_C ≡ {x + y, x − y, z}. Notice that this is the same arrangement as the arrangement in example 1.4 and the same partition as the one given in example 1.10. It is interesting that in this arrangement, one of the basis vectors for L_Π has a 2 in the column representing the hyperplane z. Because the order of each block is the same in the neighborly partitions that require a hyperplane to be thrown out, it seems appropriate to say that the hyperplane z in this example represents a doubled point [8].

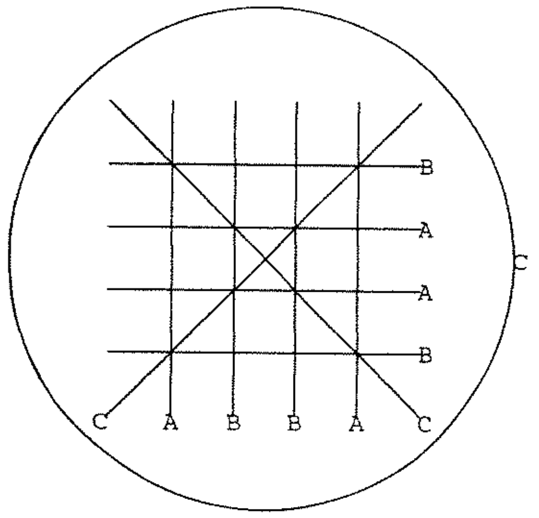

FIG 2.H

Example 2.7 An even larger arrangement with twelve projective lines is the collection G ≡ {x + z, x − z, 2x + y + z, 2x + y − z, y + 3z, y + z, y, y − z, y − 3z, 2x − y + z, 2x − y − z, z}. A neighborly partition of this arrangement is π_A ≡ {2x + y − z, y + 3z, y − z, 2x − y − z}, π_B ≡ {2x + y + z, y + z, y − 3z, 2x − y + z} and π_C ≡ {x + z, x − z, y, z}.

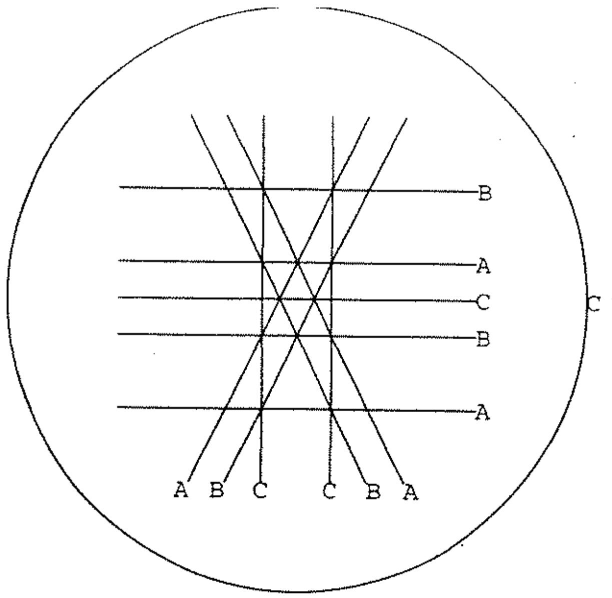

FIG 2.I

Example 2.8 Another arrangement with twelve projective lines is the collection G ≡ {x + 3z, x + z, x, x – z, x – 3z, x + y + 2z, x + y – 2z, y + z, y – z, x – y – 2z, x – y + 2z, z}. A neighborly partition of this arrangement is π_A ≡ {x + 3z, x + z, x + y – 2z, x – y – 2z}, π_B ≡ {x – z, x – 3z, x + y + 2z, x – y + 2z} and π_C ≡ {x, y + z, y – z, z

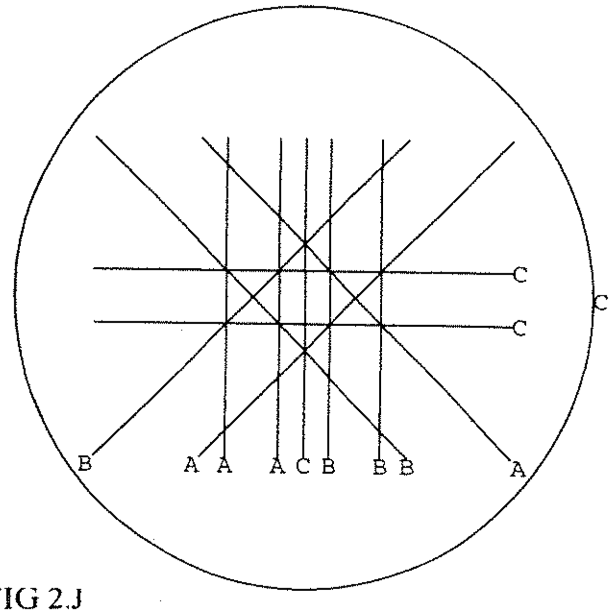

FIG 2.J

Example 2.9 The final new arrangement, with twelve projective lines, is the collection G ≡ {x + z, x, x – z, 2x + y + z, 2x + y – z, x + y + 2z, x + y – 2z, x – y – 2z, x – y + 2z, 2x – y – z, 2x – y + z, z}. A neighborly partition of this arrangement is π_A ≡ {2x + y – z, x + y + 2z, x – y + 2z, 2x – y – z}, π_B ≡ {2x + y + z, x + y – 2z, x – y – 2z, 2x – y + z} and π_C ≡ {x + z, x, x – z, z}.

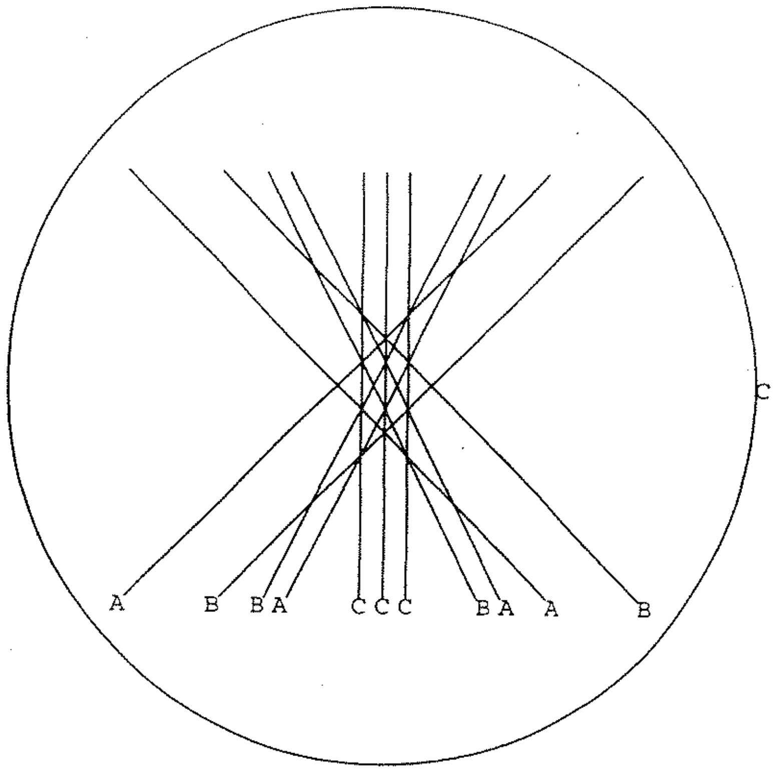

FIG 2.K

<u>Examples Over $\mathbf{Z}_p$</u>

Recall that the ground field used to construct OS algebras does not have to be **C** so it is here that we show the two known examples that work when the ground field is $\mathbf{Z}_2$.

Example 2.10 An arrangement with seven projective lines is the collection $G \equiv \{x + z, x - z, x + y, y + z, y - z, x - y, z\}$. A neighborly partition of this arrangement is $\pi_A \equiv \{x + z\}$, $\pi_B \equiv \{x - z\}$, $\pi_C \equiv \{x + y, x - y, z\}$, $\pi_D \equiv \{y + z\}$ and $\pi_E \equiv \{y - z\}$ [6].

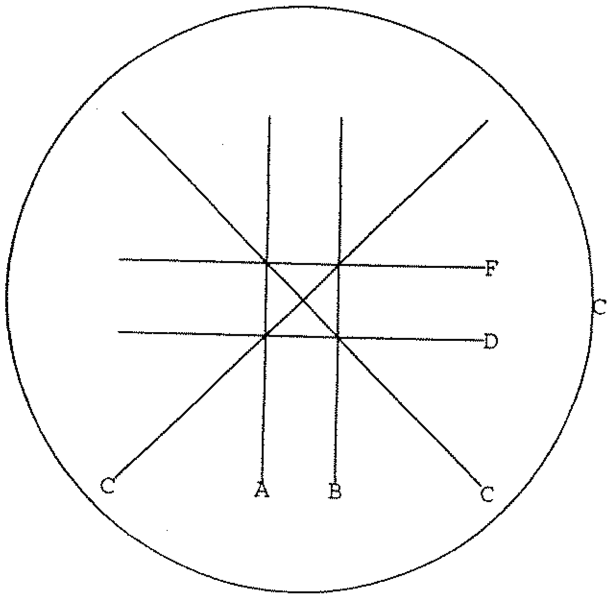

FIG 2.L

Example 2.11 An arrangement with 10 projective lines is the collection G ≡ {x + 3z, x + z, x − z, x − 3z, x + y, y − 3z, y − z, y + z, y + 3z, x − y}. A neighborly partition of this arrangement is π_A ≡ {x + 3z, x − 3z, y − z, y + z}, π_B ≡ {x + z, x − z, y − 3z, y + 3z} and π_C ≡ {x + y, x − y}. Notice that this is the same arrangement as the arrangement in example 2.6 without the line at infinity.

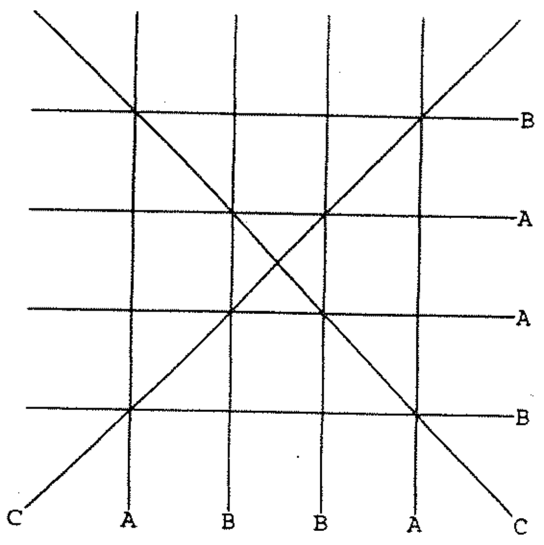

FIG 2.M

III. Line, Point Duality

The Duality

In section I we learned that the structure and representations of matroids were connected and that we could move between the two different types of representations. It turns out that this connection is a little stronger, however [8]. If we have an arrangement, construct the second way to render the arrangement and then extend the lines in this picture the resulting diagram is an arrangement. Moreover, if we start with an arrangement that supports resonant weight and follow these steps then for all the current examples the result is an arrangement that supports resonant weight. It is not known why this process works.

Example 3.1 Consider the arrangement in example 2.6. Fig 3.a is a line picture of the arrangement, Fig 3.b is a dot picture, and Fig 3.c is the arrangement resulting from extending the lines in Fig 3.b. Notice that the result is actually the same as Fig 3.a turned sideways.

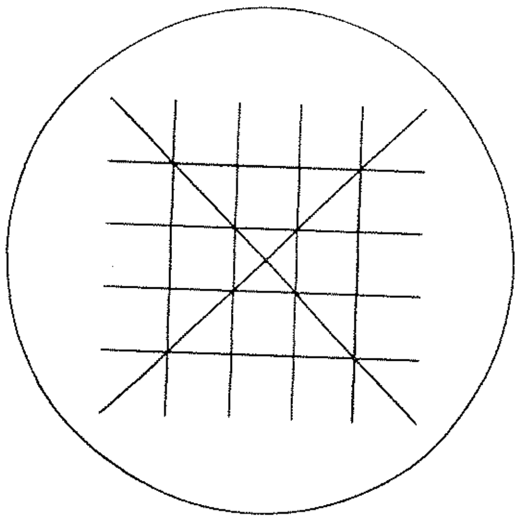
FIG 3.A

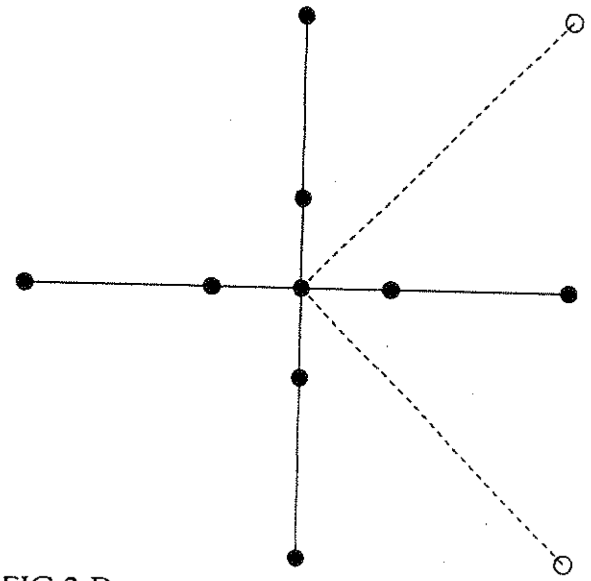
FIG 3.B

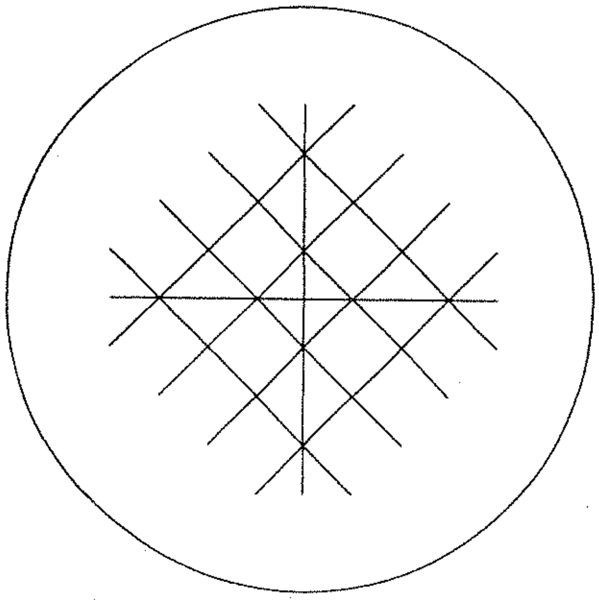
FIG 3.C

The Algorithm

The only unfortunate part about this way of looking for new examples is that it tends to give examples of arrangements that are smaller than the one being tested, and thus tends to give examples that we already have found. This is easily remedied, however. We can go the other way. This translates to starting with an arrangement, looking at the collection of rank two flats (all the projective points of intersection) and determining which of these are matroids that have neighborly partitions and that support resonant weight.

(source code for c.cpp)

```
//iscolorable.cpp will determine whether or not a given collection
//of points is colorable with a non - trivial neighborly partition, not
//all mono-colored, not all colored differently.

//there are some prerequisites however...
//the z coord can only be 1 or 0 as this program looks at the
//crossection z = 1 of the arrangement.
//also because the computer does not like large numbers the maximum
//number of projective intersection points that should be examined is
//33
//also do not use the point 1000, 1000, 1 (or 0) as this is what
//several of the arrays are initialized to

#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;

void GetInfo2(float** AP, int* Color, int* NC, int &totalnumpnts, int
&numpnts);
//this collects the total number of projective intersection points
//in the arrangement to be considered and the specific number of these
//intersection points to be sifted through. Specifically if the total
//number of intersection points is 'n' and the order of the
//subcollection is 'k' then the program will check the n choose k
//combinations of 'k' intersection points for colorability. It is also
//in this function that you must directly set up the array AP (All
//Points) with the intersection points from the arrangement you are
//considering. Finally this is where the arrays Color (which keeps
//track of what each point is colored for a specific combination of
//intersection points) and NC (Next Combination - which keeps track of
//the current combination of 'k' points to be used from AP and stored
//in the array Point) are initialized.

void ColorFlats(float** Point, int*  Color, float** POL, int* COL,
                     int numpnts, int N, int baseindex);
//*This function essentially captures the 'simple' algorithm I
described for finding a neighborly partition of a matroid. Given the
base indexed point and the 'int N', this will first examine the line
through the base index point and every other point currently in Point
using the function PointsOnLine (and LF for Little Function) to find
which points are on this line. Then this will find the colors of the
points on this line using the function ColorsOnLine. Then, if there are
N points on the line (determined by the function IsNColinear) and N - 1
```

```
are colored the same (using the functions NumOnLine and NumMax) then
this function will color the points on the line the same color using
the function ColorPoints. Finally, if a color was changed from say 2 to
1 then this will color everything originally colored 2, 1. Notice that
this function will only consider the lines going through the base
indexed point. So outside of this function you have to make sure to
change the index point.*//

void PointsOnLine(float** Point, float** POL, int numpnts, int index1,
                           int index2);
//*initializes the array POL to 1000
given two indexed points this will find which of the points currently
in point that are on the line through the indexed points using the
array POL (points on the line) to keep track.*//


int NumOnLine(float** POL, int numpnts);
//*returns the number of points on the line that POL knows.*//

void ColorsOnLine(int* Color, float** POL, int* COL, int numpnts);
//*initializes the array COL to 1000 and then records on COL which
colors are on the line that POL knows and how many of each color.*//

int MaxColor(int* COL, int numpnts);
//*returns the color that is most frequent on the line that POL
knows.*//

int NumMax(int* COL, int numpnts);
//*returns the number of points colored the max color, the most
frequent color.*//

void ColorPoints(float** POL, int* Color, int numpnts,
                          int coloringcolor);
//*colors the points POL knows the same, the coloring color*//

bool IsColorable(int* Color, int* COL, int numpnts);
//*determines whether the collection of points is colorable by
examining COL at the end of the program.*//

void LF(float** Point, float** POL, int index);
//* for Little Function this simply takes points from Point and puts
them in POL*//

bool IsNColinear(float**POL, int numpnts, int N);
//*determines whether a line has N points on it*//

void NextComb(int* NC, int totalnumpnts, int numpnts);
//*provides the next combination of points from AP to be examined for
colorability and stores this combination in the array NC which then is
used by APToPoint to move points from AP to Point*//

void APToPoint(float** AP, float ** Point, int* NC, int numpnts);
//*puts the points corresponding to the combination in NC into Point to
be checked for colorability*//

void binome(long** cofs);
```

```
//*puts the binomials coefficients up through 33 choose 33 into the
array cofs*//

long choose(int n, int k, long** cofs);
//*returns n choose k for n<34 from the array cofs*//

void main()
{
      int k;
      int numpnts;
      int totalnumpnts;

      ofstream fout;
      fout.open("output.txt");
      //*this stream is in charge of outputting those combinations of
      'k' points that are colorable*//

      float** AP;
      float** Point;
      int   Color[200];
      float** POL;
      int   COL[200];
      int   NC[200];
      long** cofs;
      cofs = new long *[34];
      for(int d=0; d<34; d++)
            cofs[d] = new long[34];

      binome(cofs);

      AP = new float *[200];
      for(k=0;k<200;k++)
            AP[k]=new float[3];

      Point=new float *[200];
      for(k=0;k<200;k++)
            Point[k]=new float[3];

      POL = new float *[200];
      for(k=0;k<200;k++)
            POL[k] = new float[3];

      GetInfo2(AP, Color, NC, totalnumpnts, numpnts);

      APToPoint(AP, Point, NC, numpnts);

//*here we just look at the first combination of points in Point and
determine if it is colorable*//
      int N,baseindex;
      for(N=2; N<numpnts; N++)
      {
            for(baseindex=0; baseindex<numpnts; baseindex++)
                  ColorFlats(Point, Color, POL, COL, numpnts, N,
                  baseindex);
      }
//*If it is colorable then we output it through the output stream
fout*//
      if(IsColorable(Color, COL, numpnts))
      {
            for(k=0; k<numpnts; k++)
            {
                  fout << k <<"th point:  " << Point[k][0] << ", " <<
                  Point[k][1] << ", " << Point[k][2]
                         << endl;
                  fout << k <<"th color:  " << Color[k] << endl;

            }
      }

//*This is where we sift through all the n choose k combination of 'k'
points and determine if they are colorable*//
      int totalchoosenumpnts = choose(totalnumpnts, numpnts, cofs);
      for(k=0; k<choose(totalnumpnts, numpnts, cofs); k++)
      {
            cout << k << <<"th step " << totalnumpnts << " choose "
                  << numpnts << " is: " <<totalchoosenumpnts<< endl;

            //*here all the points are colored different colors*//
            for(int j=0; j<numpnts; j++)
                  Color[j]=j;

            //*here we get the next combination of 'k' points*//
            NextComb(NC,totalnumpnts, numpnts);

            //*here we move those points corresponding to the correct
            combination from AP to Point*//
            APToPoint(AP, Point, NC, numpnts);

            //*here we are checking for colorability again*//
            for(N=2; N<numpnts; N++)
            {
                  for(baseindex=0; baseindex<numpnts; baseindex++)
                  {
                        ColorFlats(Point, Color, POL, COL,
                        numpnts,N,baseindex);
                  }
            }
            if(IsColorable(Color, COL, numpnts))
            {
                  for(int i=0; i<numpnts; i++)
                  {
                        fout << i << "th point:  " << Point[i][0] << ",
                        "<< Point[i][1] << ", " << Point[i][2] << endl;
                        fout << i << "th color:  " << Color[i] << endl;
                  }
                  fout.flush();
            }
      }
      fout << "hello" << endl;
}
```

```
void GetInfo2(float** AP, int* Color, int* NC, int &totalnumpnts, int
&numpnts)
{

       cout << "enter totalnumber of points: ";
       cin >> totalnumpnts;
       cout << endl;

       cout << "enter number of points to be checked: ";
       cin >> numpnts;
       cout << endl;

       AP[0][0]  = -9;
       AP[0][1]  = -15;
       AP[0][2]  = 1;

       AP[1][0]  = -9;
       AP[1][1]  = 15;
       AP[1][2]  = 1;

       AP[2][0]  = -6;
       AP[2][1]  = 0;
       AP[2][2]  = 1;

       AP[3][0]  = -3;
       AP[3][1]  = -9;
       AP[3][2]  = 1;

       AP[4][0]  = -3;
       AP[4][1]  = -3;
       AP[4][2]  = 1;

       AP[5][0]  = -3;
       AP[5][1]  = 3;
       AP[5][2]  = 1;

       AP[6][0]  = -3;
       AP[6][1]  = 9;
       AP[6][2]  = 1;

       AP[7][0]  = -1.5;
       AP[7][1]  = 0;
       AP[7][2]  = 1;

       AP[8][0]  = -1;
       AP[8][1]  = -5;
       AP[8][2]  = 1;

       AP[9][0]  = -1;
       AP[9][1]  = 5;
       AP[9][2]  = 1;

       AP[10][0]  = 0;
       AP[10][1]  = -6;
       AP[10][2]  = 1;

       AP[11][0]  = 0;
```

```
AP[11][1]  =  -3;
AP[11][2]  =  1;

AP[12][0]  =  0;
AP[12][1]  =  0;
AP[12][2]  =  1;

AP[13][0]  =  0;
AP[13][1]  =  3;
AP[13][2]  =  1;

AP[14][0]  =  0;
AP[14][1]  =  6;
AP[14][2]  =  1;

AP[15][0]  =  1;
AP[15][1]  =  -5;
AP[15][2]  =  1;

AP[16][0]  =  1;
AP[16][1]  =  5;
AP[16][2]  =  1;

AP[17][0]  =  1.5;
AP[17][1]  =  0;
AP[17][2]  =  1;

AP[18][0]  =  3;
AP[18][1]  =  -9;
AP[18][2]  =  1;

AP[19][0]  =  3;
AP[19][1]  =  -3;
AP[19][2]  =  1;

AP[20][0]  =  3;
AP[20][1]  =  3;
AP[20][2]  =  1;

AP[21][0]  =  3;
AP[21][1]  =  9;
AP[21][2]  =  1;

AP[22][0]  =  6;
AP[22][1]  =  0;
AP[22][2]  =  1;

AP[23][0]  =  9;
AP[23][1]  =  -15;
AP[23][2]  =  1;

AP[24][0]  =  9;
AP[24][1]  =  15;
AP[24][2]  =  1;

AP[25][0]  =  0;
AP[25][1]  =  1;
```

```
        AP[25][2] = 0;

        AP[26][0] = 1;
        AP[26][1] = 2;
        AP[26][2] = 0;

        AP[27][0] = 1;
        AP[27][1] = 1;
        AP[27][2] = 0;

        AP[28][0] = 1;
        AP[28][1] = 0;
        AP[28][2] = 0;

        AP[29][0] = -1;
        AP[29][1] = 1;
        AP[29][2] = 0;

        AP[30][0] = -1;
        AP[30][1] = 2;
        AP[30][2] = 0;


        for(int k = 0; k < numpnts; k++)
              Color[k] = k;

        for(int i=0; i < numpnts; i++)
              NC[i]= i;
}


bool IsNColinear(float**POL, int numpnts, int N)
{
        if(N == NumOnLine(POL, numpnts))
              return true;
        return false;
}

void ColorFlats(float** Point, int* Color, float** POL, int* COL,
                        int numpnts, int N, int baseindex)
{
        int bunny = Color[baseindex];

        for(int k=0; k<numpnts; k++)
        {
              if(k!=baseindex)
              {
                    PointsOnLine(Point,POL,numpnts,baseindex,k);
                    ColorsOnLine(Color,POL,COL,numpnts);

                    if(IsNColinear(POL,numpnts,N))
                    {
                    if(1 == (NumOnLine(POL,numpnts)-NumMax(COL,numpnts)))
                          {
                                int rabbit = Color[baseindex];
                                int bunny = Color[k];
                        ColorPoints(POL,Color,numpnts,Color[baseindex])
                        ;
                                for(int blue = 0 ; blue<numpnts; blue++)
                                {
                                        if(Color[blue] == bunny)
                                                Color[blue] = rabbit;
                                }
                        }
                }
        }
    }
}

void LF(float** Point, float** POL, int index)
{
        {
                POL[index][0] = Point[index][0];
                POL[index][1] = Point[index][1];
                POL[index][2] = Point[index][2];
        }
}


void PointsOnLine(float** Point, float** POL, int numpnts, int index1,
                                int index2)
{
        int k;

        for(k=0;k<numpnts;k++)
                {
                        POL[k][0] = 1000;
                        POL[k][1] = 1000;
                        POL[k][2] = 1000;
                }

        LF(Point, POL, index1);
        LF(Point, POL, index2);

                if(0 == Point[index1][2] && 0 == Point[index2][2])
                {
                        for(k=0; k<numpnts; k++)
                        {
                                if(0 == Point[k][2])
                                        LF(Point, POL, k);
                        }
                }

                else if(0 == Point[index1][2] && 0 == Point[index1][0])
                {
                        for(k=0; k<numpnts; k++)
                        {
                                if(Point[index2][0] == Point[k][0] && 0
                                !=Point[k][2])
                                        LF(Point, POL, k);
                        }
```

```
}

else if(0 == Point[index1][2] && 0 != Point[index1][0])
{
       double rise = (Point[index1][1]);
       double run = (Point[index1][0]);
       double brun = (run*Point[index2][1] -
       rise*Point[index2][0]);

       for(k=0; k<numpnts; k++)
       {
              if(0 != Point[k][2] && brun ==
              (run*Point[k][1]- rise*Point[k][0]))
                     LF(Point, POL, k);
       }
}


else if(0 == Point[index2][2] && 0 == Point[index2][0])
{
       for(k=0; k<numpnts; k++)
       {
              if(Point[index1][0] == Point[k][0] && 0
              !=Point[k][2])
                     LF(Point, POL, k);
       }
}

else if(0 == Point[index2][2] && 0 != Point[index2][0])
{
       double rise = (Point[index2][1]);
       double run = (Point[index2][0]);
       double brun = (run*Point[index1][1] -
       rise*Point[index1][0]);


       for(k=0; k<numpnts; k++)
       {
              if(0 != Point[k][2] && brun == (run*Point[k][1]
              - rise*Point[k][0]))
                     LF(Point, POL, k);
       }
}

else if(0 != Point[index1][2] && 0 != Point[index2][2] && 0
== (Point[index2][0] - Point[index1][0]))
{
       for(k=0; k<numpnts; k++)
       {
              if((Point[k][0] == Point[index1][0]) || (0 ==
              Point[k][2] && 0 == Point[k][0]))
                     LF(Point,POL,k);
       }
}

else if(0 != Point[index1][2] && 0 != Point[index2][2] && 0
!= (Point[index2][0] - Point[index1][0]))
```

```
            {
                    double rise = (Point[index2][1] - Point[index1][1]);
                    double run = (Point[index2][0] - Point[index1][0]);
                    double brun = (run*Point[index1][1] -
                    rise*Point[index1][0]);

                    for(k=0; k < numpnts; k++)
                    {
                            if((0 == Point[k][2] &&  (Point[k][1]*run ==
                            rise *Point[k][0])))
                                    LF(Point, POL, k);
                            else if(0 != Point[k][2] && brun ==
                            (run*Point[k][1] - rise*Point[k][0]))
                                    LF(Point, POL, k);
                    }
            }
}


int NumOnLine(float** POL, int numpnts)
{
        int numonline = 0;
        int k;

        for(k=0; k<numpnts; k++)
        {
                if(1000 != POL[k][0])
                        numonline++;
        }

        return numonline;
}


void ColorsOnLine(int* Color, float** POL, int* COL, int numpnts)
{
        int k;
        for(k=0; k<numpnts; k++)
                COL[k] = 1000;

        for(k=0; k<numpnts; k++)
        {
                if(1000 != POL[k][0])
                        COL[abs(Color[k])]++;
        }
}


int MaxColor(int* COL, int numpnts)
{
        int k;
        int maxcolor = 0;

        for(k=0; k<numpnts; k++)
        {
                if(COL[k] > COL[maxcolor])
                        maxcolor = k;
        }
        return maxcolor;
}


int NumMax(int* COL, int numpnts)
{
        return (COL[MaxColor(COL, numpnts)] - 1000);
}


void ColorPoints(float** POL, int* Color, int numpnts,
                         int coloringcolor)
{
        int k;
        for(k=0; k<numpnts; k++)
        {
                if(1000 != POL[k][0])
                        Color[k] = coloringcolor;
        }
}


bool IsColorable(int* Color, int* COL, int numpnts)
{
        int k;
        int totalnumcolors = 0;

        for(k=0; k<numpnts; k++)
                COL[k] = 1000;
        for(k=0; k<numpnts; k++)
                COL[abs(Color[k])]++;

        for(k=0; k<numpnts; k++)
        {
                if(1000 != COL[k])
                        totalnumcolors++;

        }

        if(1<totalnumcolors)
                return true;
        return false;
}


void NextComb(int* NC, int totalnumpnts, int numpnts)
{
        int i;

        for(i = (numpnts - 1); i>=0; i--)
        {
                if((totalnumpnts-numpnts+i) != NC[i])
                {
                        NC[i] = NC[i]++;
                        for(int j = i+1; j<(numpnts); j++)
                                NC[j] = (NC[i] + j - i);
                        break;
                }
        }
}


void APToPoint(float** AP, float ** Point, int* NC, int numpnts)
{
        for(int k=0; k<numpnts; k++)
        {
                Point[k][0] = AP[NC[k]][0];
                Point[k][1] = AP[NC[k]][1];
                Point[k][2] = AP[NC[k]][2];
        }
}


void binome(long** cofs)
{
        cofs[1][0] = 1;
        cofs[1][1] = 1;
        for(int k=2; k<34; k++)
        {
                for(int d=0; d<=k; d++)
                {
                        if(0 == d || k == d)
                                cofs[k][d] = 1;
                        else
                        {       long int x = cofs[k-1][d-1];
                                long int y = cofs[k-1][d];
                                long int input = (x+y);
                                cofs[k][d] = input;
                        }
                }
        }
}

long choose(int n, int k, long** cofs)
{
        return cofs[n][k];
}
```

References

[1] P. Orlik and H. Terao. *Arrangements of Hyperplanes.* Springer Verlag. Berlin Heidelberg New York, 1992.

[2] P. Orlik. *Introduction to arrangements.* Providence, R.I.: Published for the Conference Board of the Mathematical Sciences by the American Mathematical Society, c1989.

[3] Tom Brylawski and Douglas G. Kelly. Matroids and Combinatorial Geometries.

[4] D.J.A. Welsh. Matroids: Fundamental Concepts. In *Handbook of Combinatorics,* pages 481-526, 1995.

[5] M. Falk, Combinatorial and algebraic structure in Orlik – Solomon algebras

[6] Arrangements and cohomology. *Annals of Conbinatorics,* 1:135 – 157, 1997.

[7] Theodore Frankel. *The Geometry of Physics*. New York: Cambridge University Press, 2001.

[8] Private discussion with Michael Falk.