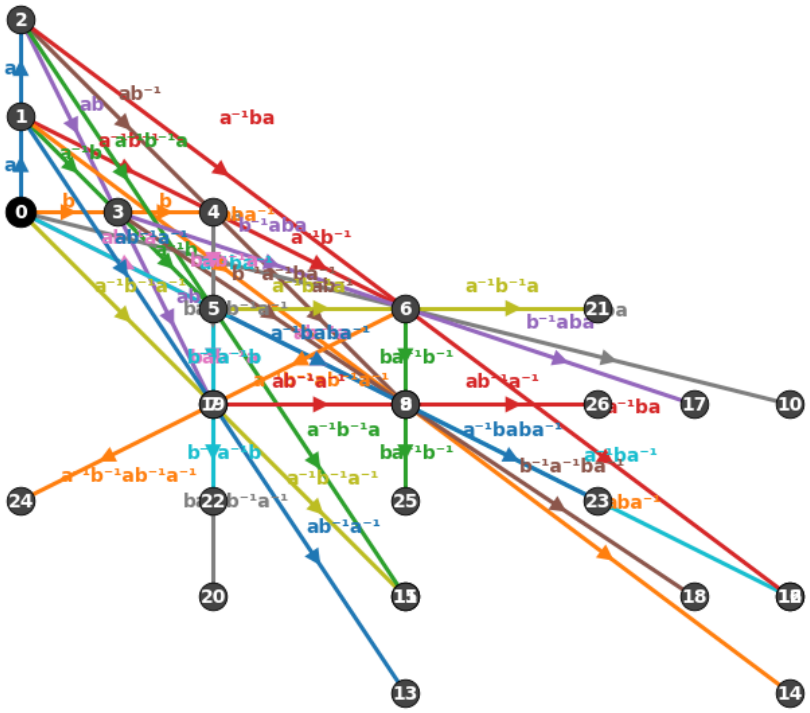


Lincidence

Free Linear Spaces, Incidence Geometry, and Symmetry

Jason Bhalla

December 2025

Abstract

Free linear spaces are incidence structures built by repeatedly adding new lines so that every pair of points lies on a unique line, while keeping the construction as "free" as possible. This project has one unified objective: to understand the symmetry of free linear spaces $\mathcal{F}_r(n)$ *and* to make that symmetry computationally visible by implementing the iterative construction and visualizing early stages. On the theoretical side, we prove a point-transitivity result: for any two points $P, Q \in \mathcal{F}_r(n)$, there exists an automorphism sending P to Q. We first control the line containing P and Q, then extend the map step-by-step so that all newly forced incidences are preserved. On the computational side, we implement the construction rule from the definition, generate finite approximations $\mathcal{F}_r^{(i)}(n)$, and visualize the resulting incidence graphs. Experiments show extremely fast growth even for small parameters (e.g. $r = 2, n = 3$), which both motivates and supports careful choices in the implementation (data structures, iteration strategy, and visualization). The combined proof and software provide a coherent picture: $\mathcal{F}_r(n)$ is not just "free" in how it is generated, but also highly symmetric in how its points relate.

Introduction

Context

Incidence geometry studies structures made of *points* and *lines* (sets of points) and the rules for when points lie on lines. A central theme is that simple axioms can produce rich global behavior, especially symmetry. In this report we focus on *linear spaces* (in the incidence-geometry sense), as presented in G. Eric Moorhouse's notes [1]. The "free" objects in many areas of math (free groups, free graphs with constraints, etc.) often have large automorphism groups, and a natural question is whether the same happens for free linear spaces.

Formal Problem Statement / Research Question

Fix parameters $r \geq 1$ (rank) and $n > 1$ (order). The free linear space $\mathcal{F}_r(n)$ is built by starting from r lines of size n through a common point and repeatedly adding new lines to connect any pair of points that is not yet on a common line, while adding new points to maintain order n. The main research question is:

> *How symmetric is $\mathcal{F}_r(n)$? In particular, can we always move any point to any other point by an automorphism?*

This report answers that question in two connected ways:

1) **Theory:** We prove that for any two points $P, Q \in \mathcal{F}_r(n)$, there exists an automorphism ϕ of $\mathcal{F}_r(n)$ such that $\phi(P) = Q$.

2) **Computation:** We implement the stage-by-stage construction $\mathcal{F}_r^{(i)}(n)$ and visualize early stages, providing concrete evidence of the "uniformity" that the proof relies on.

Literature Review

For background definitions and general incidence-geometry context, we use Moorhouse's *Incidence Geometry* notes [1].

Research Methods, Data, and Experimental Setup

Mathematical Method

The proof goal is to show point-transitivity of automorphisms of $\mathcal{F}_r(n)$. The strategy is:

- Start by controlling the *unique line* that contains the two chosen points P and Q.
- Extend the mapping outward so that whenever we commit to mapping some finite part of the space, we can extend it to include one more "forced neighborhood" without breaking the linear-space rules.
- Take a union of these finite steps to obtain a full automorphism.

Computational Method

We implemented the iterative construction in Python 3 to build finite stages $\mathcal{F}_r^{(i)}(n)$ and visualize them. Concretely:

- Points are stored with integer IDs.
- Lines are stored as ordered lists of point IDs (for visualization as cycles in the code).
- At each iteration, for each pair of points not already on a common line, the program adds a new line of cardinality n, introducing $n-2$ new points.
- Visualization uses `matplotlib` to plot points and directed edges that represent adjacency along each line cycle.

The full open-source code is available online [2].

The mathematical definition of $\mathcal{F}_r(n)$ only talks about *points* and *lines*. The code adds an extra bookkeeping layer that makes the construction and visualization easier to interpret: it attaches *generator labels* and *words* to directed edges in a way that is consistent across iterations.

- **Generators on the initial lines.** In $\mathcal{F}_r^{(1)}(n)$, we begin with r starting lines. In the program, each starting line is assigned a generator symbol (like $a, b, c, \ldots$). Intuitively, this symbol means: "move one step along that line in the chosen direction."
- **Lines as directed cycles and inverses.** Each line of order n is represented as a directed cycle
$$p_0 \to p_1 \to \cdots \to p_{n-1} \to p_0.$$
If the forward direction along that line has label w, then the reverse direction has label w^{-1}. This matches the idea that going backwards undoes going forwards.
- **Modulo-n normalization (looping).** Because a line of order n is a cycle, going forward n times returns to the start. In the code this is treated as a relation $g^n = 1$ for each generator g. Concretely: exponents are reduced modulo n (and then simplified to a shortest representative), so word strings do not grow unnecessarily.
- **Adjacency matrix with word labels.** The program stores a matrix A where $A[i][j]$ is a word if there is a directed step from point i to point j along some line, and $A[i][j] = 0$ otherwise. This makes the incidence structure searchable: a path in the directed graph corresponds to multiplying the edge-words along the path.
- **No new symbols after initialization.** When the construction adds a new line between two points i and j, the code chooses the label of that new line to be the *shortest existing word* that already takes i to j in the current structure (found by a Breadth-First Search). This is deliberate: we do not introduce a new generator each time we add a line, so the labels keep reflecting the original seed generators.

This label system is not part of the definition of $\mathcal{F}_r(n)$, but it makes the computational exploration much richer: it gives a concrete way to talk about "how to travel" between points and makes printed output (like the adjacency matrix and the running list of words) meaningful and consistent from one iteration to the next.

Experimental Parameters

In the Results section, we include a representative run with:

$$r = 2, \quad n = 3, \quad \text{iterations } i = 0, 1, 2.$$

This is small enough to compute quickly but already large enough to show the combinatorial growth.

Results

Core Main Theorem

We now state the central result of the project (the claim from the definitions section) as a theorem, and we include the full proof here.

Definitions and Claim

Definition 1. A *linear space* $(\mathcal{P}, \mathcal{L})$ is a nonempty set $\mathcal{P}$ of *points* together with a collection $\mathcal{L}$ of subsets of $\mathcal{P}$, called *lines*, such that (i) Given any two points $P, Q \in \mathcal{P}$, there exists a unique line $\ell \in \mathcal{L}$ incident with (that is, containing) P and Q and (ii) Any two lines are incident at (contain) at most one common point.

Definition 2. A *morphism* of linear spaces $(\mathcal{P}, \mathcal{L})$ and $(\mathcal{P}', \mathcal{L}')$ is a function $\phi : \mathcal{P} \to \mathcal{P}'$ such that for any line $\ell \in \mathcal{L}$, either $\phi(\ell) \in \mathcal{P}'$ or $\phi|_\ell$ (the restriction of ϕ to ℓ) is a bijection onto some line $\ell' \in \mathcal{L}'$. An *isomorphism* of linear spaces is a morphism of linear spaces that is bijective and whose inverse is a morphism of linear spaces.

Definition 3. Given numbers $r \geq 1$ (the *rank*) and $n > 1$ (the *order*), neither of which are necessarily finite, we define the *free linear space* $\mathcal{F}_r(n)$ of rank r and order n to be the linear space constructed as follows. Begin with r lines, each of cardinality n, that are coincident at a common point. Call this configuration $\mathcal{F}_r^{(1)}(n)$. For any two points P and Q in $\mathcal{F}_r^{(1)}(n)$ that do not belong to a common line, create a new line ℓ of cardinality n that contains P and Q and is such that P and Q are the only points in $\mathcal{F}_r^{(1)}(n)$ that belong to ℓ (unless $n = 2$, this will entail the creation of new points as well). Call the resulting configuration $\mathcal{F}_r^{(2)}(n)$, and repeat the process described above ad infinitum. We define

$$\mathcal{F}_r(n) \coloneqq \varinjlim \mathcal{F}_r^{(i)}(n).$$

That is, $\mathcal{F}_r(n)$ is the inductive limit (union) of the configurations $\mathcal{F}_r^{(i)}(n)$. Note that we take $\mathcal{F}_1(n)$ to consist of a single line of order n.

Claim 1. Given any two points P and Q in the free linear space $\mathcal{F}_r(n)$, there exists an automorphism ϕ of $\mathcal{F}_r(n)$ (that is, an isomorphism from $\mathcal{F}_r(n)$ to itself) such that $\phi(P) = Q$.

Theorem Statement

Theorem 1. *Given any two points P and Q in the free linear space $\mathcal{F}_r(n)$, there exists an automorphism ϕ of $\mathcal{F}_r(n)$ such that $\phi(P) = Q$.*

Full Proof (Base Case, Inductive Step, Proof of Inductive Step)

Proof. **Big picture.** We build an automorphism ϕ by first defining it on the line containing P and Q, and then extending it outward one finite step at a time. The extension step is designed so it never contradicts the incidence rules, because the free construction creates "fresh" points on new lines.

Base Case. Let ℓ be the unique line containing P and Q. Since ℓ has exactly n points, write

$$\ell = \{P, Q, R_3, \ldots, R_n\}.$$

Define a bijection $\phi_0 : \ell \to \ell$ by

$$\phi_0(P) = Q, \qquad \phi_0(Q) = P, \qquad \phi_0(R_i) = R_i \text{ for } i \geq 3.$$

This preserves incidences on ℓ (all points remain on the same line), and it forces $\phi_0(P) = Q$.

Inductive Step. Assume we have:

- a finite set $A_k \subseteq \mathcal{F}_r(n)$ that contains ℓ, and
- a bijection $\phi_k : A_k \to A_k$

such that ϕ_k preserves all line incidences among points of A_k (i.e. it is an isomorphism of the induced finite subspace on A_k).

Choose a point $x \in \mathcal{F}_r(n) \setminus A_k$. Define A_{k+1} to be the finite set consisting of:

- all points of A_k,
- the point x, and
- for every $a \in A_k$, all points on the unique line joining x and a.

This is finite because A_k is finite and each line has n points.

We will extend Φ_k to a bijection $\Phi_{k+1} : A_{k+1} \to A_{k+1}$ that still preserves all incidences among points of A_{k+1}.

Proof of the Inductive Step. The key issue is making sure the extension is consistent on every line through x and a point of A_k.

Step 1: Choose an image point y for x that avoids unwanted coincidences. Let $S = A_k$. We claim we can choose a point $y \in \mathcal{F}_r(n) \setminus S$ such that for every $s \in S$, the line $\overline{ys}$ contains no other point of S besides s.

Why such a y exists. Because S is finite, it is contained in some stage $\mathcal{F}_r^{(i)}(n)$. Choose two points $U, V \in \mathcal{F}_r^{(i)}(n) \setminus S$ that are not on a common line. In the next stage, the construction adds a new line through U and V whose only old points are U and V, and the other $n-2$ points on that line are brand new. Pick y to be one of those brand-new points.

Now fix any $s \in S$. When the construction later creates the line connecting y and s, it is created at some stage where y and s already exist, and it is created so that its only points from the previous stage are y and s. Since S is already entirely present at that earlier stage, the new line $\overline{ys}$ cannot contain any other point of S. This gives the desired property.

Finally, define

$$\phi_{k+1}(x) \coloneqq y.$$

Step 2: Extend line-by-line. For each $a \in A_k$, consider the line $\overline{xa}$. We must map it to the line $\overline{y\,\phi_k(a)}$. Because of how we chose y, the line $\overline{y\,\phi_k(a)}$ intersects A_k only at $\phi_k(a)$. This means there is no ambiguity: the points of $\overline{xa} \cap A_k$ already have images under ϕ_k, and the remaining points of $\overline{xa}$ that are not in A_k can be bijected to the remaining points of $\overline{y\,\phi_k(a)}$ that are not in A_k.

Do this for every $a \in A_k$. These choices do not conflict with each other because different lines through x map to different lines through y, and two distinct lines intersect in at most one point.

Thus ϕ_{k+1} is a well-defined bijection on A_{k+1} that preserves all incidences among points of A_{k+1}.

Finish. By induction we obtain ϕ_k on an increasing sequence of finite sets A_k whose union is all of $\mathcal{F}_r(n)$ (choose points one-by-one in some enumeration). Define

$$\phi \coloneqq \bigcup_{k \geq 0} \phi_k.$$

Because each ϕ_{k+1} extends ϕ_k, ϕ is a well-defined bijection $\mathcal{F}_r(n) \to \mathcal{F}_r(n)$. Since any line involves finitely many points, it is preserved at some finite stage, so ϕ preserves all lines. Therefore ϕ is an automorphism and $\phi(P) = Q$ by the base case. □

Computational Results: Growth and Visualizations

For $r = 2$ and $n = 3$, the number of points and lines grows quickly:

Iteration i	# Points in $\mathcal{F}_2^{(i)}(3)$	# Lines in $\mathcal{F}_2^{(i)}(3)$	New lines added in step
0	5	2	–
1	9	6	4
2	27	24	18
3	306	303	279

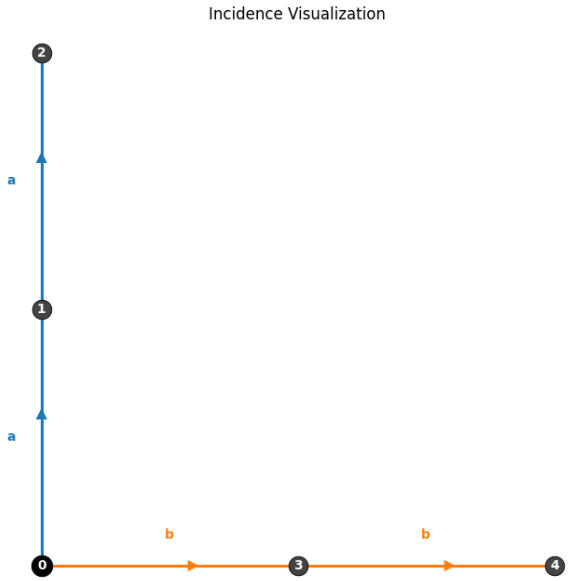
In addition to plotting the incidence graph, the program prints an adjacency matrix whose entries are word labels. Conceptually:

- $A[i][j] = 0$ means there is no directed edge from point i to point j in the current stage.
- $A[i][j] = w$ means there is a directed edge $i \to j$ and it is labeled by the word w.
- If $A[i][j] = w$, then $A[j][i] = w^{-1}$ for the reverse direction along the same line.

As iterations proceed, new lines are introduced and the adjacency matrix becomes more dense. The printed `WORDS` list (in the code) is simply the de-duplicated list of distinct words that appear in adjacency entries, where duplicates are removed even if they appear in inverse form.

The following four figures are examples of the initial configuration and first 3 iterations of generating a free linear space with rank 2 and order 3 computationally.

```
build_and_iterate(3, 2, 0) # Order 3, Rank 2, Iteration 0
```



Words: $\{a, b\}$

Lines:

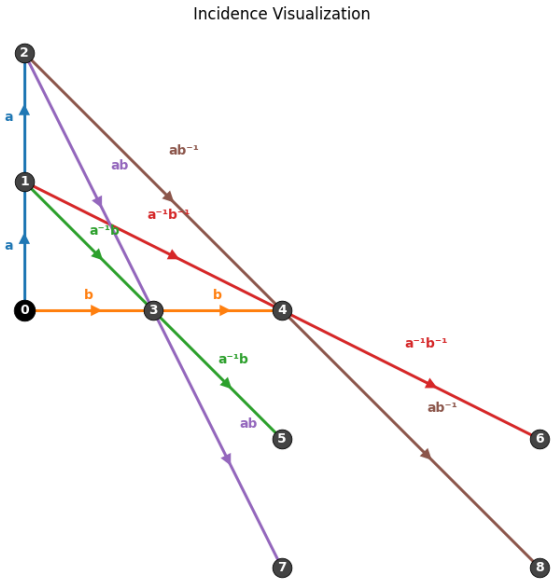
a : $[0, 1, 2]$
b : $[0, 3, 4]$

Adjacency Matrix:

	0	1	2	3	4
0	0	a	a^{-1}	b	b^{-1}
1	a^{-1}	0	a	0	0
2	a	a^{-1}	0	0	0
3	b^{-1}	0	0	0	b
4	b	0	0	b^{-1}	0

Figure 1: Visualization of $\mathcal{F}_2^{(0)}(3)$ (initial configuration).

```
build_and_iterate(3, 2, 1) # Order 3, Rank 2, Iteration 1
```



Words: $\{a,\ b,\ a^{-1}b,\ a^{-1}b^{-1},\ ab,\ ab^{-1}\}$

Lines:

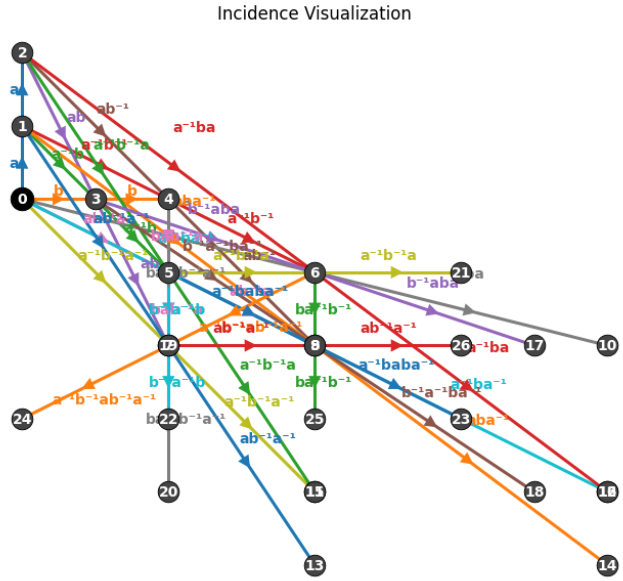
a : $[0, 1, 2]$
b : $[0, 3, 4]$
$a^{-1}b$: $[1, 3, 5]$
$a^{-1}b^{-1}$: $[1, 4, 6]$
ab : $[2, 3, 7]$
ab^{-1} : $[2, 4, 8]$

Adjacency Matrix:

	0	1	2	3	4	5	6	7	8
0	0	a	a^{-1}	b	b^{-1}	0	0	0	0
1	a^{-1}	0	a	$a^{-1}b$	$a^{-1}b^{-1}$	$b^{-1}a$	ba	0	0
2	a	a^{-1}	0	ab	ab^{-1}	0	0	$b^{-1}a^{-1}$	ba^{-1}
3	b^{-1}	$b^{-1}a$	$b^{-1}a^{-1}$	0	b	$a^{-1}b$	0	ab	0
4	b	ba	ba^{-1}	b^{-1}	0	0	$a^{-1}b^{-1}$	0	ab^{-1}
5	0	$a^{-1}b$	0	$b^{-1}a$	0	0	0	0	0
6	0	$a^{-1}b^{-1}$	0	0	ba	0	0	0	0
7	0	0	ab	$b^{-1}a^{-1}$	0	0	0	0	0
8	0	0	ab^{-1}	0	ba^{-1}	0	0	0	0

Figure 2: Visualization of $\mathcal{F}_2^{(1)}(3)$ (after one iteration).

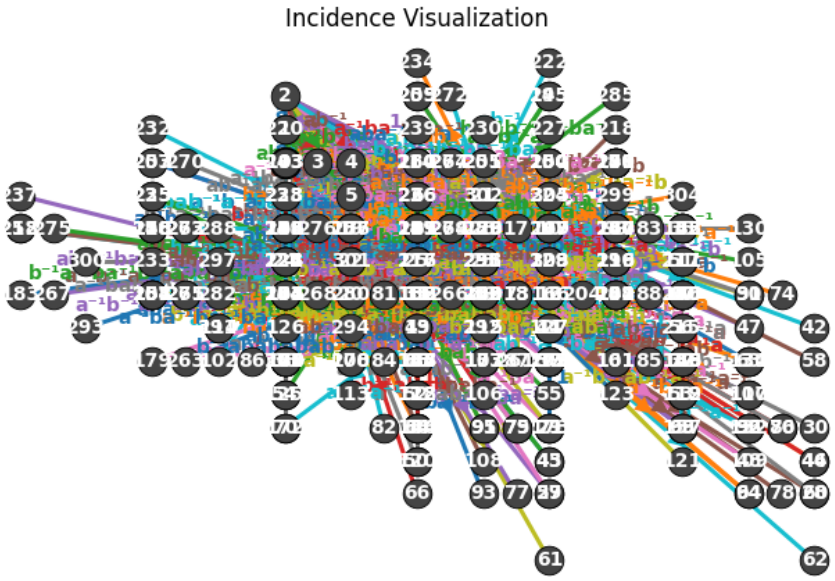
```
build_and_iterate(3, 2, 2) # Order 3, Rank 2, Iteration 2
```



Words, Lines, Adjacency Matrix: [Omitted for Brevity]

Figure 3: Visualization of $\mathcal{F}_2^{(2)}(3)$ (after two iterations).

```
build_and_iterate(3, 2, 3) # Order 3, Rank 2, Iteration 3
```

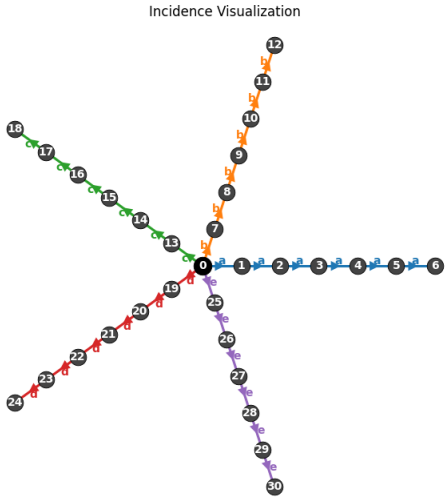


Words, Lines, Adjacency Matrix: [Omitted for Brevity]

Figure 4: Visualization of $\mathcal{F}_2^{(3)}(3)$ (after three iterations).

The next two figures illustrate the initial configuration and first iteration of generating a free linear space with rank 5 and order 7, to show that the code can generate a free linear space with *any* rank and order input (and how large the structures get quickly!).

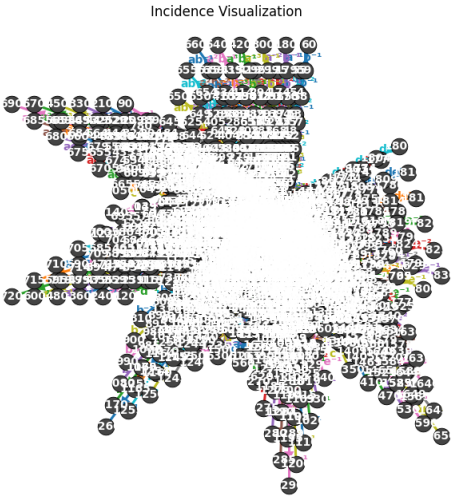
```
build_and_iterate(7, 5, 0) # Order 7, Rank 5, Iteration 0
```



Words, Lines, Adjacency Matrix: [Omitted for Brevity]

Figure 5: Visualization of $\mathcal{F}_5^{(0)}(7)$ (initial configuration).

```
build_and_iterate(7, 5, 1) # Order 7, Rank 5, Iteration 1
```



Words, Lines, Adjacency Matrix: [Omitted for Brevity]

Figure 6: Visualization of $\mathcal{F}_5^{(1)}(7)$ (after one iteration).

Discussion / Interpretation of Results / Conclusion

Interpretation

The computation shows that even small (r, n) generates very fast growth. This matches the definition: every iteration tries to connect *every* non-collinear pair, so the number of required new lines explodes.

On the theory side, Theorem 1 says that (despite the fast growth) $\mathcal{F}_r(n)$ has no "special" points: from the viewpoint of the full incidence structure, every point looks the same up to symmetry.

On the computational side, the generator/word system provides a practical lens for understanding the growth: it turns "incidence-only" data into a navigable labeled graph where paths have explicit word labels. Even though these labels are not part of the definition of $\mathcal{F}_r(n)$, they make it easier to debug, compare iterations, and reason about how new lines are being attached.

Limitations

- **Computational Limitation:** The naive all-pairs iteration becomes huge quickly; beyond a few iterations, memory/time becomes a real constraint.
- **Visualization Limitation:** Planar plotting is only a visualization device; it does not reflect actual geometry, only incidence.

Conclusion

We studied free linear spaces $\mathcal{F}_r(n)$ both theoretically and computationally. The central theoretical result is that for any two points $P, Q \in \mathcal{F}_r(n)$, there exists an automorphism ϕ of $\mathcal{F}_r(n)$ such that $\phi(P) = Q$; in other words, automorphisms of $\mathcal{F}_r(n)$ can move any point to any other point. The code complements this by generating concrete finite stages $\mathcal{F}_r^{(i)}(n)$ and visualizing early iterations, making the structure and symmetry more intuitive. The additional generator/word labeling system provides a consistent way to track movement along lines and across the growing structure, and it produces interpretable adjacency matrices that reflect the construction step-by-step.

Future Work / Open Problems

1) **Efficiency Improvements:** Replace the all-pairs scanning step with incremental bookkeeping of non-collinear pairs, and measure iteration limits for different (r, n).
2) **Free Linear Space Construction:** Expand the iterative steps to do a sort of "gluing" or merging step, to allow the successful creation of finite free linear spaces.
3) **Automorphism Experiments:** Implement explicit automorphisms suggested by the proof (as constructive algorithms) and test them on finite stages.
4) **Stronger Symmetry:** Investigate whether $\mathcal{F}_r(n)$ is transitive on ordered pairs of distinct points, or on larger configurations.
5) **Word-Label Theory Connection:** Study whether the word system is capturing a deeper algebraic structure (for example, whether certain word equivalences correspond to identifiable structural features in early iterations).

Bibliography / References

References

[1] G. Eric Moorhouse, *Incidence Geometry*, lecture notes/handout (PDF), available online at https://ericmoorhouse.org/handouts/Incidence_Geometry.pdf.

[2] Jason Bhalla, *Lincidence: Algorithmic Generation of Free Linear Spaces and Combinatorial Structures in Incidence Geometry*, GitHub repository (open source), https://github.com/jasonbhalla/Lincidence.

Addendum / Appendix

Appendix A: Selected Code Segments

The full codebase is in [2]. Below are a few key excerpts that connect directly to the construction in the definition, and to the generator/word labeling system.

A.1 Iteration rule: add a line for every pair not on a common line

```
def iterate_once(struct: IncidenceStructure, c: int) -> None:
    S = sorted(struct.P.keys())
    def same_line(i, j): return points_on_same_line(struct, i, j)
    nbrs = adjacency_as_word_neighbors(struct, c)

    shortest_word_from: Dict[int, Dict[int, Word]] = {}
    for s in S:
        shortest_word_from[s] = bfs_shortest_words(nbrs, s, c)

    pairs_to_add: List[Tuple[int, int, Word]] = []
    for i, j in itertools.combinations(S, 2):
        if same_line(i, j):
            continue
        wmap = shortest_word_from[i]
        if j not in wmap:
            continue
        w = wmap[j].reduce_mod_c(c)
        pairs_to_add.append((i, j, w))

    for i, j, w in pairs_to_add:
        add_line_connecting(struct, i, j, w, c)

    struct.adjacency = struct._build_adjacency_directed()
```

A.2 Add a new line of size n by creating $n-2$ new points (and label it)

```
def add_line_connecting(struct: IncidenceStructure, i: int, j: int, w:
    Word, c: int):
    P = struct.P
    L = struct.L
    next_line_id = (max(L.keys()) + 1) if L else 0
```

```
    next_point_id = (max(P.keys()) + 1) if P else 0

    w = w.reduce_mod_c(c)

    line_point_ids = [i, j]
    for step in range(1, c - 1):
        pid = next_point_id
        next_point_id += 1
        P[pid] = Point(id=pid, x=0.0, y=0.0)  # visual only
        line_point_ids.append(pid)

    # Entire new line shares the SAME word label w
    L[next_line_id] = Line(id=next_line_id, point_ids=line_point_ids,
        generator=str(w))
```

A.3 Generator symbols for the initial lines

```
def _gen_symbol(idx: int) -> str:
    letters = 'abcdefghijklmnopqrstuvwxyz'
    s = ''
    n = idx + 1
    while n > 0:
        n, rem = divmod(n - 1, 26)
        s = letters[rem] + s
    return s

def generate_initial(c: int, r: int, *, radius_step: float = 1.0) ->
    IncidenceStructure:
    # ...
    for k in range(r):
        gid = _gen_symbol(k)  # generator label for this starting line
        # ...
        lines[k] = Line(id=k, point_ids=line_point_ids, generator=gid)
```

A.4 Breadth-First Search to compute shortest "word paths" between points

```
def adjacency_as_word_neighbors(struct: IncidenceStructure, c: int) ->
    Dict[int, List[Tuple[int, Word]]]:
    A = struct.adjacency
    n = len(A)
    nbrs: Dict[int, List[Tuple[int, Word]]] = {i: [] for i in range(n)}
    for i in range(n):
        for j, cell in enumerate(A[i]):
            if cell != '0':
                nbrs[i].append((j, Word.parse(cell).reduce_mod_c(c)))
    return nbrs

def bfs_shortest_words(nbrs: Dict[int, List[Tuple[int, Word]]], src:
    int, c: int) -> Dict[int, Word]:
    visited = set([src])
    q: Deque[int] = deque([src])
```

```
    path_word: Dict[int, Word] = {src: Word.identity()}
    while q:
        u = q.popleft()
        for v, w_uv in nbrs[u]:
            if v not in visited:
                visited.add(v)
                composed = path_word[u].mul_plain(w_uv).reduce_mod_c(c)
                path_word[v] = composed
                q.append(v)
    return path_word
```