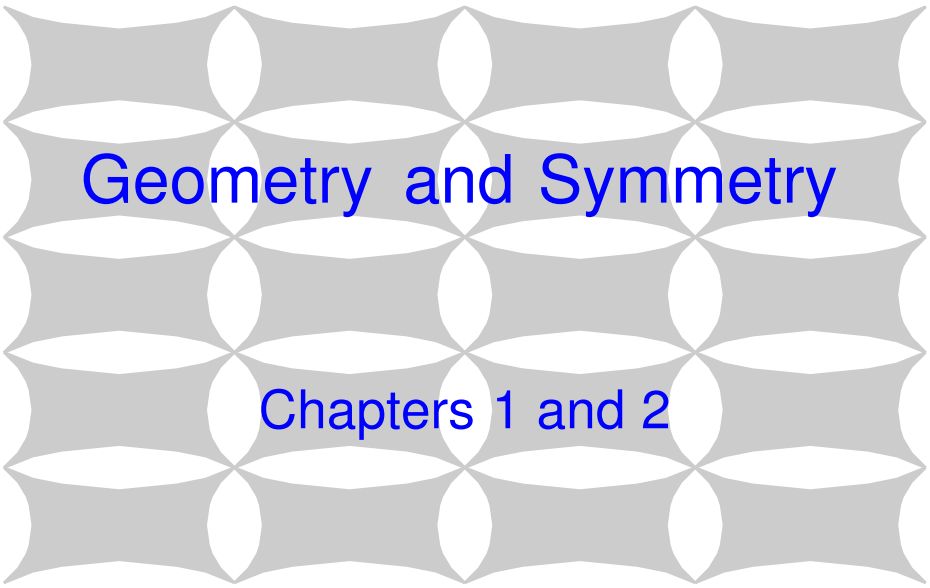


Transformation Functions and Symmetry

Barry H Dayton

https : // barryhdayton.space

In the In the 1965-66 school year I took an upper level abstract algebra course from Paul Yale, a professor at Pomona College. His motivation for teaching the course is that he was writing a book *Geometry and Symmetry* which was published by Holden-Day in 1968. Yale wanted to learn more group theory for his book so he chose the brand new textbook *Topics in Algebra* by I.N Herstein. This was a mistake for two reasons. First, it was much too hard for most of his students, a fact that Herstein later rectified by writing a later version. Second, Herstein used the functional notation xf instead of the $f(x)$ that most of us are more familiar. But Yale then used this notation in his book which makes it unreadable for modern readers.

There was another problem with Yale' s book which was unavoidable at that time, the lack of a good overall group of transformations to cover all the situations Yale studied . Mathematica has now provided this group, invertible transformation functions, and they can be easily calculated . Now it is possible to go into much more detail .

My plan is to revisit the material in Yale' s book using Mathematica and Transformation Functions . As I write material I will post it . I will keep going as long as I can given my age, hopefully I will ultimately cover much of the book . The book will be published in two formats, as text, HTML or PDF, and as a Mathematica notebook so those with Mathematica can use my functions . The functions themselves will also be offered in a separate notebook .

1. Matrix Groups

I will discuss some elementary matrix theory here mostly to show Mathematica usage for those unfamiliar with it. In Mathematica a matrix is a 2 - dimensional array . A typical 3×3 matrix look like

```
In[ ]:= A = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

We can make it display like a usual matrix

```
In[ ]:= A // MatrixForm
```

```
Out[ ]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The element in the second row, third column can be recovered by

```
In[ ]:= A[[2, 3]]
Out[ ]= 6
```

To multiply a matrix by a vector we use the dot

```
In[ ]:= A.{1, 2, 3}
Out[ ]= {14, 32, 50}
```

Note **Mathematica** automatically converts the row vector to column vector form. So this could be written

```
In[ ]:= A.{{1}, {2}, {3}}
Out[ ]= {{14}, {32}, {50}}
```

where now the output is a column.

Matrix multiplication is given by the same . function where the first matrix is multiplied by each column of the second, but as above giving columns. If B is the matrix

```
In[ ]:= B = {{1, -1, 3}, {2, 1, 2}, {3, -1, 5}};
        B // MatrixForm
```

```
Out[ ]//MatrixForm=
```

$$\begin{pmatrix} 1 & -1 & 3 \\ 2 & 1 & 2 \\ 3 & -1 & 5 \end{pmatrix}$$

Then

```
In[ ]:= A.B // MatrixForm
```

```
Out[ ]//MatrixForm=
```

$$\begin{pmatrix} 14 & -2 & 22 \\ 32 & -5 & 52 \\ 50 & -8 & 82 \end{pmatrix}$$

Note the matrix form matrices are not **Mathematica** variables but if you cut and paste **Mathematica** can still understand.

In[ ]:=
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} . \begin{pmatrix} 1 & -1 & 3 \\ 2 & 1 & 2 \\ 3 & -1 & 5 \end{pmatrix}$$

Out[ ]= {{14, -2, 22}, {32, -5, 52}, {50, -8, 82}}

or

In[ ]:=
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} . \begin{pmatrix} 1 & -1 & 3 \\ 2 & 1 & 2 \\ 3 & -1 & 5 \end{pmatrix} \text{// MatrixForm}$$

Out[ ]//MatrixForm=
$$\begin{pmatrix} 14 & -2 & 22 \\ 32 & -5 & 52 \\ 50 & -8 & 82 \end{pmatrix}$$

The identity matrix $\mathbb{I}_n$ is the n×n matrix with diagonal entries 1 and other entries 0.

In[ ]:=
```
𝕀3 = IdentityMatrix[3] // MatrixForm
```

Out[ ]//MatrixForm=
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A matrix A is *invertible* if there is a matrix B so that $A.B = B.A = \mathbb{I}_n$ for suitable n. It can be shown that for this to happen both matrices must be n×n for the same n. Although, in general, matrix multiplication is not commutative in this case given A is a square matrix it is known that one of $A.B = \mathbb{I}_n$ or $B.A = \mathbb{I}_n$ already implies they are inverses, that is *invertible.* Almost all square matrices in this book are assumed invertible.

Notice that for the matrix B above

In[ ]:=
```
B
```

Out[ ]= {{1, -1, 3}, {2, 1, 2}, {3, -1, 5}}

that

In[ ]:=
```
B^-1
```

Out[ ]= $\left\{\left\{1, -1, \frac{1}{3}\right\}, \left\{\frac{1}{2}, 1, \frac{1}{2}\right\}, \left\{\frac{1}{3}, -1, \frac{1}{5}\right\}\right\}$

not the actual

In[ ]:=
```
Inverse[B]
```

Out[ ]= $\left\{\left\{-\frac{7}{4}, -\frac{1}{2}, \frac{5}{4}\right\}, \{1, 1, -1\}, \left\{\frac{5}{4}, \frac{1}{2}, -\frac{3}{4}\right\}\right\}$

Thus we must be careful to not use the superscript -1 in the matrix context. More generally, to get the power of a matrix, say A.A.A use

```
In[ ]:= MatrixPower [A, 3]
Out[ ]= {{468, 576, 684}, {1062, 1305, 1548}, {1656, 2034, 2412}}
```

not

```
In[ ]:= A ^ 3
Out[ ]= {{1, 8, 27}, {64, 125, 216}, {343, 512, 729}}
```

So, unfortunately we must use the term Inverse [*B*] or MatrixPower [*B*, –1] for the inverse of a matrix in **Mathematica.**

Incidentally the matrix *A* above is not invertible

```
In[ ]:= MatrixPower [A, -1]
MatrixPower : Matrix {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} is singular .
Out[ ]= MatrixPower [{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}, -1]
```

an error message is given when trying to invert it , non-invertible matrices are called singular. See a linear algebra book or my Appendix 1 of my Plane Curve book for a full discussion of invertibility, particularly in the difficult case of numerical matrices.

Given this, I define a *matrix group* to be a set of invertible matrices of the same size so that if *A*, *B* are in the group so is *A*.*B*, Inverse [*A*] and Inverse [*B*].

Two trivial examples are the zero group {0} and the full set of all invertible n×n matrices. In the latter case this is because the inverse of an invertible matrix is by definition and the product of invertible matrices is invertible because Inverse [*A*.*B*] = Inverse [*B*].Inverse[A].

Matrix groups may be finite or infinite. For simple example of a finite Matrix group, let k be a fixed odd integer

```
In[ ]:= A = N[{{Cos[2 Pi / k], -Sin[2 Pi / k]}, {Sin[2 Pi / k], Cos[2 Pi / k]}}]
Out[ ]= {{Cos[6.28319/k], -1. Sin[6.28319/k]}, {Sin[6.28319/k], Cos[6.28319/k]}}
```

where k is a positive odd integer . We will show the powers of A forms a group with *k* elements.

For now let k = 5.

```
In[ ]:= A5 = A /. {k → 5}
Out[ ]= {{0.309017, -0.951057}, {0.951057, 0.309017}}
```

The powers are given by

```
In[ ]:= RecurrenceTable [{P[i + 1] == A5.P[i], P[1] == A5}, P, {i, k}]
Out[ ]= RecurrenceTable [{P[1 + i] == {{0.309017, -0.951057}, {0.951057, 0.309017}}.P[i],
          P[1] == {{0.309017, -0.951057}, {0.951057, 0.309017}}}, P, {i, k}]
```

Note the last matrix is the identity matrix so this will repeat . The reader with **Mathematica** should try some more of these.

Many important matrix groups are important . Geometrically matrices with positive determinants are orientation preserving. Historically when most math problems came with integer coefficients determinants were useful for calculation since one could avoid fractions or decimals. Nowadays we are allowed to use decimals so determinants are an inefficient way to calculate. **Mathematica** can calculate the determinant of a square matrix for you if you need it.

The important point is that determinants satisfy

A is invertible if and only if Det[A] ≠ 0 (1)

Det[A.B] = Det[A] × Det[B] (2)

Det[Inverse[A]] = 1 / Det[A] (3)

The possible problem with (1) is that with numerical matrices not being zero is a tricky concept so (1) should just be used in theory, again see my Appendix 1 in my *Plane Curve Book.*

But (1), (2), (3) together imply that the set of matrices with positive determinant form a matrix group as does the set of matrices with determinant 1.

Another important matrix group is the group of orthogonal matrices. A square matrix is *orthogonal* if either one of the following are true

Norm[A.v] = Norm[v] for every vector v (4)

Inverse[A] = Transpose[A] (5)

There actually are many more characterizations of orthogonality . In (4) Norm[v] is also known as the length of v or |v|. For (5) the transpose changes rows to columns. Transpose[A] is also given by A^T

In[ ]:=
```
Transpose[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

The reader can check using random matrices that transpose anti-commutes with multiplication and commutes with inverses so

Transpose[A.B] = Transpose[B].Transpose[A] (6)

Inverse[Transpose[A]] = Transpose[Inverse[A]] (7)

Then one can show the set of orthogonal matrices form a group using (5).

Finally one can show that all invertible matrices of the forms

In[ ]:=
```
{{a, b, c}, {e, f, g}, {0, 0, 1}} // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} a & b & c \\ e & f & g \\ 0 & 0 & 1 \end{pmatrix}$$

form a group and in fact a matrix of this form is invertible if and only if

In[ ]:=
```
{{a, b}, {e, f}} // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} a & b \\ e & f \end{pmatrix}$$

is invertible. Moreover if this latter matrix is orthogonal. A similar thing holds for 4×4 or larger matrices.

2. Transformation Functions

2.1 Definition and general properties

While matrix groups are interesting and have been thoroughly studied by mathematicians they are are not geometric transformation groups. First of all they are not transformations although given a matrix A there is an associated linear transformation given by **A.#&** in **Mathematica.** As an example

In[ ]:=
```
A = {{2, 1}, {1, 4}};
A // MatrixForm
TA = A.# &
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix}$$

Out[ ]=
```
A.#1 &
```

In[ ]:=
```
TA[{x, y}]
```

Out[ ]=
```
{2 x + y, x + 4 y}
```

This fixes one problem, but there is a bigger problem, as geometrical transformations linear transformations only give rotations about the origin and reflections in lines through the origin. We need general rotations about any point, reflections about any line, also translations. So we need to extend our set of transformations. These were originally used in Algebraic Geometry for transforming the projective line. The first general discussion that I am aware of is in Shreeram S. Abhyankar's book *Algebraic geometry for Scientists and Engineers* where he has an entire chapter titled *fractional linear Transformations.* He is particularly interested in using these for transformations of projective spaces which he defines in this chapter. In But the authors of Mathematica, and I have used them for general transformations of Euclidean space. The observation here is that a translation of the Euclidean plane or space is a linear transformation of projective space. Later chapters of this book may get to projective transformations, but I have discussed fractional linear transformations (FLT) in the context of projective transformations in my three geometry books on plane curves, space curves and surfaces.

Transformation functions are built in objects in Mathematica . They are defined by using matrices, but for the plane we use a 3×3 matrix and for space a 4×4 matrix. We get rational functions, that is the coordinates are given by fractions. The definition in the plane case is given

```
In[ ]:= A = {{a[1, 1], a[1, 2], a[1, 3]}, {a[2, 1], a[2, 2], a[2, 3]}, {a[3, 1], a[3, 2], a[3, 3]}};
        A // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} a[1,\ 1] & a[1,\ 2] & a[1,\ 3] \\ a[2,\ 1] & a[2,\ 2] & a[2,\ 3] \\ a[3,\ 1] & a[3,\ 2] & a[3,\ 3] \end{pmatrix}$$

```
In[ ]:= TransformationFunction [A][{x, y}]
```

$$\text{Out[ ]=}\ \left\{\frac{x\ a[1,\ 1] + y\ a[1,\ 2] + a[1,\ 3]}{x\ a[3,\ 1] + y\ a[3,\ 2] + a[3,\ 3]},\ \frac{x\ a[2,\ 1] + y\ a[2,\ 2] + a[2,\ 3]}{x\ a[3,\ 1] + y\ a[3,\ 2] + a[3,\ 3]}\right\}$$

There is a similar definition in the space case . Notice the two coordinates have the same denominator. An important thing to notice is that even though we used a 3×3 matrix the argument is a 2 dimensional vector. As another example

```
In[ ]:= B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

In[ ]:= TransformationFunction [B][{11, 12}]
```

$$\text{Out[ ]=}\ \left\{\frac{19}{91},\ \frac{55}{91}\right\}$$

Linear functions are a special case, for example the linear transformation *T* defined above is given by

```
In[ ]:= TFA = TransformationFunction [{{2, 1, 0}, {1, 4, 0}, {0, 0, 1}}]
```

$$\text{Out[ ]=}\ \text{TransformationFunction}\left[\left(\begin{array}{cc|c} 2 & 1 & 0 \\ 1 & 4 & 0 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$$

```
In[ ]:= TFA[{x, y}]
Out[ ]= {2 x + y, x + 4 y}
```

Note the last row and column are {0,0,1}.

Another example is

```
In[ ]:= L = {{1, 0, 5}, {0, 1, -6}, {0, 0, 1}}
Out[ ]= {{1, 0, 5}, {0, 1, -6}, {0, 0, 1}}

In[ ]:= TL = TransformationFunction [L]
```

$$\text{Out[ ]=}\ \text{TransformationFunction}\left[\left(\begin{array}{cc|c} 1 & 0 & 5 \\ 0 & 1 & -6 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$$

```
In[ ]:= Then

In[ ]:= TL[{3, 5}]
Out[ ]= {8, -1}
```

which is just a translation of the point {x,y} by the vector {5,-6}. Combining these examples

In[ ]:= **TL[TFA[{x, y}]]**

Out[ ]= {5 + 2 x + y, −6 + x + 4 y}

Note also

In[ ]:= **TLA = TransformationFunction [{{2, 1, 5}, {1, 4, −6}, {0, 0, 1}}]**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 2 & 1 & 5 \\ 1 & 4 & -6 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$

In[ ]:= **TLA[{x, y}]**

Out[ ]= {5 + 2 x + y, −6 + x + 4 y}

A very important fact about fractional linear transformations is the composition formula for matrices K, L

```
TransformationFunction[K][TransformationFunction[L]] =
 TransformationFunction[K.L]
```
(8)

In my three geometry books, *Plane Curves, Space Curves, Surfaces* I use fractional linear transformations with notation flt[p,A] , and variations, interchangeably as transformation functions. At some points TransformationFunctions are used to generate flt. Transformation functions as functions give the same values flt functions. But Mathematica treats them differently. In Mathematica Transformation Functions are a different data type than functions. This will be particularly important working with groups of TransformationFunctions where composition of functions is heavily used. While (8) is not true for TransformationFunctions it does define composition. The matrix multiplication is easy for Mathematica but composition of functions is difficult if it is wanted in the simplified form of a TransformationFunction. Mathematica will often leave function composition unevaluated unless actual numerical values are used. The correct method for composing Transformation Functions is using the operator @* Consider the example above

In[ ]:= **TL[TFA]**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 1 & 0 & 5 \\ 0 & 1 & -6 \\ \hline 0 & 0 & 1 \end{array}\right)\right]\left[\text{TransformationFunction}\left[\left(\begin{array}{cc|c} 2 & 1 & 0 \\ 1 & 4 & 0 \\ \hline 0 & 0 & 1 \end{array}\right)\right]\right]$

In[ ]:= **TL @* TFA**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 2 & 1 & 5 \\ 1 & 4 & -6 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$

In the first case we just get an unevaluated composition, in the second we get an actual TransformationFunction .

Mathematica should have a test for a function being a TransformationFunction, here is one

```
In[ ]:= TransformationFunctionQ [f_] :=
         If[Length[Dimensions[TransformationMatrix [f]]] == 2, True, False]
```

Another function that is be useful for invertible transformations

```
In[ ]:= InverseTF[α_] := TransformationFunction [Inverse[TransformationMatrix [α]]]
```

For example

```
In[ ]:= InverseTF[TL]
```

$$\text{Out[ ]= TransformationFunction}\left[\left(\begin{array}{cc|c} 1 & 0 & -5 \\ 0 & 1 & 6 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$$

Note Mathematica's **InverseFunction** will also work

```
In[ ]:= InverseFunction [TL]
```

$$\text{Out[ ]= TransformationFunction}\left[\left(\begin{array}{cc|c} 1 & 0 & -5 \\ 0 & 1 & 6 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$$

2.2 Fundamental theorems on TransformationFunctions

The fundamental theorems say that, as a function, a TransformationFunction defined by a n×n matrix is determined by its action on n+1 general position points. General position is slightly more general than linearly independent. Rather than give a definition I will give test for general position in our present context. Suppose one has n+1 points in $\mathbb{R}^n$, Euclidean n-space. Write these points as column vectors v[1],v[2],…,v[n+1], append the number 1 to the bottom of each and construct the (n+1)×(n+1) matrix with these new columns. If this matrix is invertible then the original points were in general position.

The main example in 2 space is the example {0,0}, {1,0},{0,1}, our matrix is

```
In[ ]:= gpm = {{0, 1, 0}, {0, 0, 1}, {1, 1, 1}};
        gpm // MatrixForm
```

$$\text{Out[ ]//MatrixForm=}\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

```
In[ ]:= Inverse[gpm]
Out[ ]= {{-1, -1, 1}, {1, 0, 0}, {0, 1, 0}}
```

Since A has an inverse, the set {0,0}, {1,0},{0,1} is in general position. In the algorithms below we will use the test for invertibility that Det[A] is not zero, theoretically this is right but there are possible numerical issues we will ignore. If the absolute value of the determinant is bigger than .0001 we will accept the matrix as invertible below. For this example there is no problem

In[ ]:=
```
Det[gpm]
```

Out[ ]= 1

With this understanding we give our fundamental theorems for dimensions 2, 3 as algorithms

In[ ]:=
```
Options[getTF2D] = {returnMatrix → False};
getTF2D[P_, Q_, OptionsPattern[]] := Module[{A, B, M},
  A = {{P[[1, 1]], P[[2, 1]], P[[3, 1]]}, {P[[1, 2]], P[[2, 2]], P[[3, 2]]}, {1, 1, 1}};
  B = {{Q[[1, 1]], Q[[2, 1]], Q[[3, 1]]}, {Q[[1, 2]], Q[[2, 2]], Q[[3, 2]]}, {1, 1, 1}};
  If[Abs[Det[A]] < 1.*^-4, Echo["P not in general position"]; Abort[]];
If[Abs[Det[B]] < 1.*^-4, Echo["Q not in general position"]; Abort[]];
  M = B.Inverse[A];
  If[OptionValue[returnMatrix], Return[M]];
  TransformationFunction[M]
 ]
```

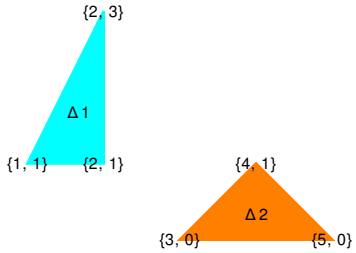
In[ ]:=
```
Options[getTF3D] = {returnMatrix → False};
getTF3D[P_, Q_, OptionsPattern[]] := Module[{A, B, M},
  A = {{P[[1, 1]], P[[2, 1]], P[[3, 1]], P[[4, 1]]}, {P[[1, 2]], P[[2, 2]], P[[3, 2]], P[[4, 2]]},
    {P[[1, 3]], P[[2, 3]], P[[3, 3]], P[[4, 3]]}, {1, 1, 1, 1}};
  B = {{Q[[1, 1]], Q[[2, 1]], Q[[3, 1]], Q[[4, 1]]}, {Q[[1, 2]], Q[[2, 2]], Q[[3, 2]], Q[[4, 2]]},
    {Q[[1, 3]], Q[[2, 3]], Q[[3, 3]], Q[[4, 3]]}, {1, 1, 1, 1}};
  If[Abs[Det[A]] < 1.*^-4, Echo["P not in general position"]; Abort[]];
If[Abs[Det[B]] < 1.*^-4, Echo["Q not in general position"]; Abort[]];
  M = B.Inverse[A];
  If[OptionValue[returnMatrix], Return[M]];
  TransformationFunction[M]
 ]
```

For example if we wish to find a Transformation Function taking the plane triangle $\Delta 1$ with vertices {{1,1}, {2,1},{2,3}} to the triangle $\Delta 2$ with vertices {{-1,0},{0,1},{1,0}} then

In[ ]:=
```
τ = getTF2D[{{1, 1}, {2, 1}, {2, 3}}, {{3, 0}, {4, 1}, {5, 0}}]
```

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 1 & \frac{1}{2} & \frac{3}{2} \\ 1 & -\frac{1}{2} & -\frac{1}{2} \\ \hline 0 & 0 & 1 \end{array}\right)\right]$

Out[ ]=



We can use these algorithms to change my fractional linear transformation is my previous books to transformationFunctions we have

In[ ]:=
```
flt2TF2D[α_] := Module[{a0, a1, a2},
  a0 = α@{0, 0};
a1 = α@{1, 0};
a2 = α@{0, 1};
  getTF2D[{{0, 0}, {1, 0}, {0, 1}}, {a0, a1, a2}]]
```

For instance if I apply this to a linear function defined by a matrix

In[ ]:=
```
Lf1 = {{1, 3}, {2, -1}}.# &
```

Out[ ]=
```
{{1, 3}, {2, -1}}.#1 &
```

In[ ]:=
```
σ = flt2TF2D[Lf1]
```

Out[ ]=
```
TransformationFunction [( 1  3 | 0 )
                        ( 2 -1 | 0 )
                        ( 0  0 | 1 )]
```

In 3 dimensions we have

In[ ]:=
```
flt2TF3D[α_] := Module[{a0, a1, a2, a3},
  a0 = α@{0, 0, 0};
a1 = α@{1, 0, 0};
a2 = α@{0, 1, 0};
  a3 = α@{0, 0, 1};
  getTF3D[{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}, {0, 0, 1}}, {a0, a1, a2, a3}]]
```

The theoretical implications of the Fundamental Theorems are perhaps more important than the practical ones. Since the action on any 3 general position points on the plane, 4 points in 3 space, the action of each point is important. Heuristically this implies that different Transformation Functions will send a random point to a different place. This is confirmed in practice, in fact the point doesn't need to be very random, a pseudo random point with decimal components of 3 digits is often random enough in a given situation. In what follows we will often call such a point a *test point* and we will infer that two Transformation Functions that send a test point to sufficiently close points, by default the allowable error is .001, are considered equal. I'm sure there will be objections from the math purists but in practice this works well and quickly.

It should be noted, however, that this single test point test works only in the general case when we are working with a large family of Transformation Functions independent of a geometrical object. Later when we work inside a group of Transformation Functions and test points, such as vertices, midpoints and centroids, come from a geometric object then it is expected that the "test points" we use will be sent to the same value by different group elements.

2.3 Types of Transformation Functions

If for no other reason Transformation Functions are important with **Mathematica** because there are built-in constructions of important transformations. For example in Chapter 3 we will consider rotations of the sphere, these are all orthogonal linear transformations, but because **Mathematica** has a general construction of rotations with different axis we will view them as Transformation Functions.

The Transformation Functions in this section are all *isometries*, also known as *rigid motions,* they preserve Euclidean distance and hence Euclidean geometry. The common feature is that the upper left square in the Transformation Matrix is an orthogonal matrix. All Transformation Matrices that give isometries will be one of these types. In 2D we will do this classification.

2.3.1 Rotation Transforms

Although **Mathematica** has many variations the most useful format for us is

```
In[ ]:= RotationTransform[θ, v]
Out[ ]= RotationTransform[θ, v]
```

where θ is an angle in radians and ax is the axis, a point, center, in two dimensions or a direction vector vector in 3 dimensions. If the axis does not go through the origin, it always will in Chapter 3, then the syntax is

```
In[ ]:= RotationTransform[θ, v, p]
Out[ ]= RotationTransform[θ, v, p]
```

where p is a point on the axis and v is a direction vector . **Mathematica** will automatically determine whether it is 2D or 3D by the number of components of v. If v is missing it will be assumed that we are in the 2 dimension case with center the origin.

A RotationTransform with $\theta = \pi$ (Pi) is called a half turn. It will be its own inverse. Although we have an inversion function note that in general the inverse of a rotation transform of angle θ is the transform with same axis and angle $-\theta$ For example

```
In[ ]:= RotationTransform[Pi / 5]@* RotationTransform[-Pi / 5]
Out[ ]= TransformationFunction[( 1  0 | 0 )
                               ( 0  1 | 0 )
                               ( 0  0 | 1 )]
```

Note that when defining a rotation transform the TransformationMatrix is returned rather than the

angle.

```
In[ ]:= RotationTransform [Pi / 5]
```

$$\text{Out[ ]= TransformationFunction}\left[\left(\begin{array}{cc|c} \frac{1}{4}\times\left(1+\sqrt{5}\right) & -\frac{1}{2}\sqrt{\frac{1}{2}\times\left(5-\sqrt{5}\right)} & 0 \\ \frac{1}{2}\sqrt{\frac{1}{2}\times\left(5-\sqrt{5}\right)} & \frac{1}{4}\times\left(1+\sqrt{5}\right) & 0 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$$

If Mathematica know an exact Transformation Matrix it will give it as in this case. I will usually bypass this with

```
In[ ]:= N[RotationTransform [Pi / 5]]
```

$$\text{Out[ ]= TransformationFunction}\left[\left(\begin{array}{cc|c} 0.809017 & -0.587785 & 0. \\ 0.587785 & 0.809017 & 0. \\ \hline 0. & 0. & 1. \end{array}\right)\right]$$

or just

```
In[ ]:= ζ = RotationTransform [2 Pi / 5.]
```

$$\text{Out[ ]= TransformationFunction}\left[\left(\begin{array}{cc|c} 0.309017 & -0.951057 & 0. \\ 0.951057 & 0.309017 & 0. \\ \hline 0. & 0. & 1. \end{array}\right)\right]$$

Notice

```
In[ ]:= Cos[2 Pi / 5.]
        Sin[2 Pi / 5.]
```

```
Out[ ]= 0.309017
```

```
Out[ ]= 0.951057
```

so one may deduce the rotation angle from this RotationTransform

```
In[ ]:= ArcCos[TransformationMatrix [ζ][[1, 1]]]
```

```
Out[ ]= 1.25664
```

where

```
In[ ]:= N[2 Pi / 5]
```

```
Out[ ]= 1.25664
```

although we have to treat the values of **ArcCos** with care .

More generally we have

```
In[ ]:= RotationTransform[θ]@*RotationTransform[ϕ] = RotationTransform[θ + ϕ]
```

but, again this is only valid up to an integer multiple of 2π.

2.3.2 TranslationTransforms

As a function a translation in v can be defined easily by

In[ ]:= **T = (# + v) &**

Out[ ]= # 1 + v &

In[ ]:= If

In[ ]:= **v = {2, 3};**

In[ ]:= **T@{1, -2}**

Out[ ]= {3, 1}

But this is not a Transformation Function. We could use **flt2TF2D** above or better use

In[ ]:= **TranslationTransform [v]**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 1 & 0 & 2 \\ 0 & 1 & 3 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$

v could be a 2 or 3 dimensional vector .

In[ ]:= **τ = TranslationTransform [{2, 3}]**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 1 & 0 & 2 \\ 0 & 1 & 3 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$

The inverse is given by vector -v

In[ ]:= **InverseTF [τ]**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 1 & 0 & -2 \\ 0 & 1 & -3 \\ \hline 0 & 0 & 1 \end{array}\right)\right]$

and composition is given by addition of vectors .

2.3.3 Reflection Transforms

Mathematica has a ReflectionTransform however I prefer my own. Here I give 2 points on the line v,w of reflection

In[ ]:=
```
reflectionTF2D [{v_,w_}]:=Module[{p,pp},
p=N[v-w];
pp={p[[2]],-p[[1]]};
getTF2D [{v,w,v+pp},{v,w,v-pp}]]
```

In 3 space I use 3 general position points on the reflecting plane

```
In[ ]:= reflectionTF3D[{p_, q_, r_}] := Module[{u, v, w, P, Q},
          u = q - p;
          v = r - p;
          w = Cross[u, v];
          P = {p, q, r, p + w};
          Q = {p, q, r, p - w};
          getTF3D[P, Q]]
```

This code shows how powerful the fundamental theorem is, normally one would have to use the linear algebra of projections to define this transformation function. But all this requires is one cross product!

Perhaps one wants to find the reflection in a plane defined by an equation. Here is an example of how to do it. Start with the plane

```
In[ ]:= x + y - z == 5.
Out[ ]= x + y - z == 5.
```

Note I make it numeric by the last decimal point . I can have **Mathematica** find my 3 points

```
In[ ]:= W = {x, y, z} /. FindInstance[x + y - z == 5., {x, y, z}, Reals, 3]
Out[ ]= {{-67/5, -11/2, -23.9}, {-19/5, -53/5, -19.4}, {62/5, -131/10, -5.7}}
```

Then

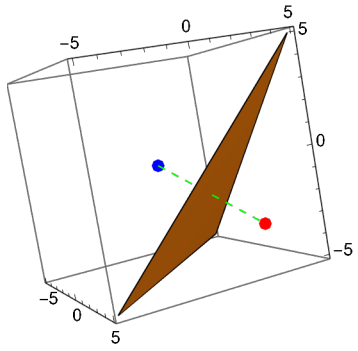
```
In[ ]:= σ = reflectionTF3D[W]
Out[ ]= TransformationFunction[(  0.333333      -0.666667      0.666667  |  3.33333
                                 -0.666667       0.333333      0.666667  |  3.33333
                                  0.666667       0.666667      0.333333  | -3.33333
                                 ---------------------------------------------------
                                 1.64799 × 10^-17  -1.12757 × 10^-17  -2.1684 × 10^-17  |  1.  )]
```

Here is a picture, note

```
In[ ]:= σ@{0, 0, 0}
Out[ ]= {3.33333, 3.33333, -3.33333}
```

In[ ]:=
```
Show[ContourPlot3D [x + y - z == 5, {x, -5, 5}, {y, -5, 5}, {z, -5, 5}, Mesh → None],
 Graphics3D [{PointSize [.04], {Blue, Point[{0, 0, 0}]}, {Red, Point[σ@{0, 0, 0}]},
   {Green, Dashed, Thickness [.005], Line[{{0, 0, 0}, 10 / 3 {1, 1, -1}}]}]}], ImageSize → Small]
```

Out[ ]=



2.3.4 Glide Reflections and Screw Displacements

A glide reflection is the composition of a translation and reflection in the direction of the translation. These are not built-in functions but we give these for convenience in classification in both 2D and 3D.

In[ ]:=
```
glideReflectionTF2D [{v_, w_}] := reflectionTF2D [{v, w}]@* TranslationTransform [w - v]
```

For 3D there are many planes containing a line in the translation so we need to also specify a point on a plane containing the line.

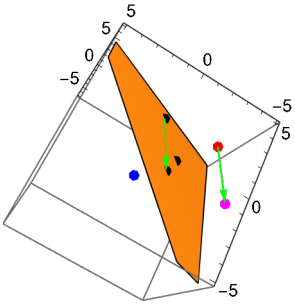
In[ ]:=
```
glideReflectionTF3D [{p_, q_}, r_] :=
 reflectionTF3D [{p, q, r}]@* TranslationTransform [q - p]
```

In[ ]:=
```
r = {5/3, 5/3, -5/3};
c = {1, 4, 0};
d = {4, 0, -1};
γ = glideReflectionTF3D [{c, d}, r];
γ@{0, 0, 0}
```

Out[ ]= $\left\{\frac{19}{3}, -\frac{2}{3}, -\frac{13}{3}\right\}$

```
In[ ]:= Show[ContourPlot3D [x + y - z == 5, {x, -7, 7}, {y, -5, 5}, {z, -5, 5}, Mesh → None],
  Graphics3D [{PointSize [.04], {Blue, Point[{0, 0, 0}]}, {Red, Point[σ@{0, 0, 0}]},
    {Magenta, Point[γ@{0, 0, 0}]}, {Black, Point[{r, c, d}]}, {Green, Thickness [.005],
     Arrow[{c, d}], Arrow[{σ@{0, 0, 0}, γ@{0, 0, 0}}]}]}, ImageSize → 150]
```

Out[ ]=



In 3 dimensions a *screw displacement* is the composition of a rotation and a translation in the axis of the rotation. We use the translation and angle to define it. In two dimensions a composition of a translation and rotation is generally a rotation. In the function p,q are points in 3 coordinates.

```
In[ ]:= screwDisplacement3DTF [θ_, {p_, q_}] :=
  TranslationTransform [q - p]@* RotationTransform [θ, q - p, p]
```

For an example, let

```
In[ ]:= ξ = N[screwDisplacement3DTF [Pi / 2, {{-.25, 0, -.25}, {.25, 0, .25}}]]
```

Out[ ]=

$$\text{TransformationFunction}\left[\left(\begin{array}{ccc|c} 0.5 & -0.707107 & 0.5 & 0.5 \\ 0.707107 & 0. & -0.707107 & 0. \\ 0.5 & 0.707107 & 0.5 & 0.5 \\ \hline 0. & 0. & 0. & 1. \end{array}\right)\right]$$

In the illustration we apply this to the point $p = \{0, -.25, 0\}$ which is the red point. The blue solid arrow is the translation which is on the black line *l*=InfiniteLine[{{-.25, 0, -.25}, {.25, 0, .25}}]. The dashed arrow is the application of this translation to *p* which is sent to the green point. The green circle through the green point with center on *l* perpendicular to the line *l*. The magenta point is the the result of ξ on p,

```
In[ ]:= ξ@{-.25, 0, .25}
```

Out[ ]= {0.5, -0.353553, 0.5}

Paul Yale, in his Chapter 2, shows that these rigid motions or isometries are the only ones possible in 2D and 3D.

I mention that rotations, translations and screw displacements have TransformationMatrices with determinant 1 while reflections and glide reflections have Transformations with determinant -1. This is a first step in classifying isometries.

In the 2D case I will prove this well known but rarely proven classification theorem by giving an algorithm that performs the classification. This will be in Chapter 3. In a later chapter I may attempt the 3D case which is much harder.

2.4 Groups of Transformation Function Isometries.

2.4 .1 Equality of transformation functions.

The set of transformation functions composes not as functions but by multiplication of Transformation Matrices. This is must faster and does not require excessive simplifications than function composition. The product is given by the operator **@*** which should always be used to insure that the product remains a transformation function. If α is a transformation function then evaluation is by **α@p** where **p** is a point in two or three space. You should never in our context use α[p] which should give the same value but if p is itself a value of a transformation function then proper evaluation of α[p] will not be done. Also never try to compose transformation functions with just @, again $\alpha@\beta$ may give the same function as $\alpha@^*\beta$ but the result is not a transformation function.

The set of all invertible 2D transformation functions and the set of all invertible 3D transformation functions are groups. Essentially as groups they are the same as the group of all 3×3, respectively 4×4 matrices. These groups are much to big for us at this point. A smaller group is the group of invertible 2D and 3D *affine transformations.* These transformation matrices are of the form, respectively, in **Mathematica's** notation

$$\left(\begin{array}{cc|c} \star & \star & \star \\ \star & \star & \star \\ \hline 0 & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{ccc|c} \star & \star & \star & \star \\ \star & \star & \star & \star \\ \star & \star & \star & \star \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

Where ★ marks a position where a number other than 0 or 1 can go .

The isometries are those transformation functions where the upper 2×2, respectively 3×3 upper left squares are orthogonal matrices. The built-in **Mathematica** function **`OrthogonalMatrixQ[]`** is the easiest way to check that a square matrix is orthogonal. As mentioned the transformation functions in section are all isometries and isometries are all one of the types considered.

In working with Transformation Function groups it is important to know when 2 transformation functions are the same . There are several ways to approach this but a central problem is that we will be working with transformation matrices constructed with machine numbers. Already deciding when two machine numbers are equal is difficult and context dependent. For this we generally use a tolerance, a small number that we feel is essentially 0 in the context. Two numbers are considered "equal" if their difference is less, in absolute value, than the tolerance. We could say two transformation functions are equal if their transformation matrices have "equal" entries. In 2D we would need to check at least the 6 entries in the top two rows, in 3D the 12 entries in the top 3 rows. Alternatively we could subtract the matrices and compare to the zero matrix. A quicker way, at least in programming, than comparing each entry with zero we can look at the **`Norm`** of a matrix, this is like looking at an absolute value of a num-

ber. If the norm of the difference is less than the tolerance then we could declare equality. Since we are thinking multiplicatively and using only invertible matrices we could decide if two transformation matrices A, B are equal if A.Inverse[B] is equal to the identity matrix using one of the methods above. I have given 3 or 4 choices, it is possible they will not always agree.

However for most of the situations I will be looking at i prefer an idea I discussed earlier that works directly with the transformation function. Pick a test point v, a 2-vector for 2D, a 3-vector in 3D. If this is sufficiently random then I will declare the the two transformation functions to be equal if the test values obtained by applying the two transformation functions to the test point if the values are equal which will be satisfied if the coordinates are numerically "equal" or the difference is numerically zero or if the Norm of the test vector is numerically zero. If nothing else, this will make coding simpler.

Here is a simple example .

In[ ]:=
```
λ = N[RotationTransform [2 Pi / 3]]
```

Out[ ]=
```
TransformationFunction [( -0.5       -0.866025 | 0.
                          0.866025    -0.5      | 0.
                          ----------------------+----
                          0.          0.        | 1. )]
```

We want to know if, given tolerance

In[ ]:=
```
ϵ = .00000001
```

Out[ ]=
```
1. × 10^-8
```

In[ ]:=
```
ω = λ@*λ
```

Out[ ]=
```
TransformationFunction [( -0.5       0.866025  | 0.
                          -0.866025  -0.5       | 0.
                          ----------------------+----
                          0.         0.        | 1. )]
```

In[ ]:=
```
μ = InverseFunction [λ]
```

Out[ ]=
```
TransformationFunction [( -0.5       0.866025  | 0.
                          -0.866025  -0.5       | 0.
                          ----------------------+----
                          0.         0.        | 1. )]
```

Visually this appears to be true . But we don't see enough digits.

In[ ]:=
```
Mω = TransformationMatrix [ω]
Mμ = TransformationMatrix [μ]
```

Out[ ]=
```
{{-0.5, 0.866025, 0.}, {-0.866025, -0.5, 0.}, {0., 0., 1.}}
```

Out[ ]=
```
{{-0.5, 0.866025, 0.}, {-0.866025, -0.5, 0.}, {0., 0., 1.}}
```

In[ ]:=
```
Mω - Mμ
```

Out[ ]=
```
{{2.22045 × 10^-16, -1.11022 × 10^-16, 0.}, {1.11022 × 10^-16, 2.22045 × 10^-16, 0.}, {0., 0., 0.}}
```

All entries are below tolerance

In[ ]:=
```
Mω.Inverse[Mμ]
```

Out[ ]= $\{\{1., -1.11022 \times 10^{-16}, 0.\}, \{1.66533 \times 10^{-16}, 1., 0.\}, \{0., 0., 1.\}\}$

All entries here differ from the Identity matrix by less than our tolerance .

In[ ]:=
```
Norm[Mω - Mμ]
```

Out[ ]= 2.48253×10^{-16}

In[ ]:=
```
Norm[IdentityMatrix[3] - Mω.Inverse[Mμ]]
```

Out[ ]= 2.89601×10^{-16}

These are also below tolerance .

The other alternative is to pick test point

In[ ]:=
```
tp = RandomReal[{-1, 1}, 2]
```

Out[ ]= {-0.359828, -0.312882}

In[ ]:=
```
q1 = ω@tp
```

Out[ ]= {-0.09105, 0.468061}

In[ ]:=
```
q2 = μ@tp
```

Out[ ]= {-0.09105, 0.468061}

In[ ]:=
```
q1 - q2
```

Out[ ]= $\{0., -1.11022 \times 10^{-16}\}$

In[ ]:=
```
Norm[q1 - q2]
```

Out[ ]= 1.11022×10^{-16}

This is easier .

2.4.2 Finite Groups of Transformation Functions

I first discuss the *order* of a Transformation Function. We have observed earlier that a rotation of angle $2\pi/n$ has the property that rasing it to the n^{th} power gives the identity transformation. More generally any transformation with that property has *order n*. Many transformations such as translations do not have a finite order, raising them to any finite power does not give the identity. Here is an easy routine to find the order.

In[ ]:=
```
Options[orderTF] = {tol → 1.*^-4};
orderTF[α_, tp_, omax_, OptionsPattern[]] := Module[{n, p},
  n = 1;
  p = α@tp;
  While[n ≤ omax && Norm[p - tp] > OptionValue[tol], p = α@p; n++];
  n]
```

This function takes a transformation function α, a test point **tp** and attempts to find a power giving the identity. There is also an input **omax** which puts a limit on the powers. Realistically if one hasn't reached the order by power 10 the order is probably infinite. The context may give some reason why not. In a finite group the order of an element is always less than the order of the group. A theorem in finite group theory says that the order of any element in the group divides the number of elements in the group. The number of elements of a finite group is also known as the *order* of the group.

An example

```
In[ ]:= κ = N[RotationTransform [Pi / 4]]
Out[ ]= TransformationFunction [( 0.707107  -0.707107 | 0. )
                                ( 0.707107   0.707107 | 0. )
                                (   0.         0.     | 1. )]
```

```
In[ ]:= orderTF[κ, {2.315 , 3.426}, 11]
Out[ ]= 8
```

another

```
In[ ]:= γ = glideReflectionTF2D [{{2, 3}, {4, -1}}]
Out[ ]= TransformationFunction [( -0.6  -0.8 |  7.6 )
                                ( -0.8   0.6 | -1.2 )
                                (  0.    0.  |  1. )]
```

```
In[ ]:= orderTF[γ, {2.315 , 3.426}, 11]
Out[ ]= 12
```

This means we have not found the order .

```
In[ ]:= orderTF[γ, {2.315 , 3.426}, 21]
Out[ ]= 22
```

We conclude that the order is probably infinite, which is true. Note that we picked a pseudo-random test point. It is random enough for our purposes.

2.4.3 Generators of transformation Groups

By definition of transformation group if α, β are in the group so is $\alpha@^*\beta$, in general so is any power of α, that is product of α with itself any number of times. If α has finite order n then the n-1 power of α is Inverse[α]. In general it is not true that if α, β have finite order that $\alpha@^*\beta$ does, for example the product of two plane reflections with parallel axis is a translation. But if we know that our group of transformations is finite then given several transformations, $\alpha_1, \alpha_2, \ldots, \alpha_k$ in the the finite group the set of products of of these transformations will be a subgroup of the group, known as *the subgroup generated by the set* $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$. Remember that transformations may not commute so we allow products in any order, so for example given α, β we need $\alpha@^*\beta$, $\beta@\alpha$, $\alpha@^*\alpha$, $\alpha@^*\alpha@^*\beta$, $\alpha@^*\beta@^*\beta$, $\alpha@^*\beta@^*\alpha$, $\beta@^*\alpha@^*\beta$,... which may all be different. So even if α,β have small order this subgroup generated by 2 transformations can be very large.

To efficiently work with all these combinations we use the association data type in Mathematica. The reader not familiar with associations may wish to read the help page on this now. Essentially an association is a generalized function where we have two sets called instead of domain and range called the *keys* and *values.* A set of arrows from a key to a value associates the key to the value. Unlike a function a key can have several values. Like a function a value can have many keys. Unlike sets, associations are demarcated by brackets <| and |>.

So discuss the subgroup generated by certain elements of a finite group we use a *tas,* a transformation association which is an association which has keys from 1,…,k with values the generators. **It is important that the keys form a consecutive sequence from 1 to k, no duplicates or missing numbers.** Then each ordered list of integers from 1 to k will denote a specific, but not necessarily unique, element of the group by taking the order product.

For example given transformations α, β then the tas is, say

```
In[ ]:= tas1 = <| 1 → α, 2 → β |>
Out[ ]= <| 1 → α, 2 → β |>
```

So now group elements are denoted by ordered lists such as

```
In[ ]:= {1}, {1, 1}, {1, 2}, {1, 2, 1}
```

where {1} is α, {2} is β, {1,1} is $\alpha @^* \alpha$, {1,2} is $\alpha @^* \beta$, {1,2,1} is $\alpha @^* \beta @^* \alpha$ and so on. These transformations need not be unique. In fact one of our goals is to find a unique set of these transformations.

The code for converting these ordered lists of integers to transformations is simply

```
In[ ]:= TasTF[S_, tas_] := Module[{len, t, i}, len = Length[S];
          t = tas[S⟦1⟧];
          i = 2;
          While[i ≤ len, t = t@* tas[S⟦i⟧]; i++];
          t]
```

Example :

```
In[ ]:= σ = RotationTransform [Pi, {0, 0}]
        ρ = reflectionTF2D [{{-1, 0}, {1, 0}}]
Out[ ]= TransformationFunction [( -1   0 | 0 )]
                               (  0  -1 | 0 )
                               (------------)
                               (  0   0 | 1 )

Out[ ]= TransformationFunction [( 1.   0. | 0. )]
                               ( 0.  -1. | 0. )
                               (-------------- )
                               ( 0.   0. | 1. )
```

```
In[ ]:= TasTF[{1, 2}, <|1 → σ, 2 → ρ|>]
```

```
Out[ ]= TransformationFunction [( -1.  0. | 0. )
                                (  0.  1. | 0. )
                                (  0.  0. | 1. )]
```

```
In[ ]:= TasTF[{1, 1}, <|1 → σ, 2 → ρ|>]
```

```
Out[ ]= TransformationFunction [( 1  0 | 0 )
                                ( 0  1 | 0 )
                                ( 0  0 | 1 )]
```

```
In[ ]:= TasTF[{2, 1, 2}, <|1 → σ, 2 → ρ|>]
```

```
Out[ ]= TransformationFunction [( -1.   0. | 0. )
                                (  0.  -1. | 0. )
                                (  0.   0. | 1. )]
```

We are now almost done finding the finite transformation group generated by a tas association. The idea is to generate all ordered, possibly repeating, lists of the integers 1 to k. We can evaluate each transformation at a fixed test point. By deleting duplicates we have a distinct list of points, each associated with a group element. We can go back searching for an appropriate key, that is, ordered list to construct that transformation. Here is the code, a subroutine first

```
In[ ]:= Options[nFindKeys] = {tol → 1.*^-5};
        nFindKeys[A_, v_, OptionsPattern []] := Module[{S},
             S = Normal[A];
             Reap[Do[If[Norm[s[[2]] - v] < OptionValue[tol], Sow[s[[1]]], {s, S}]]][[2, 1]];
```

```
In[ ]:= Options[finiteTransGroup] = {tol → .001};
        finiteTransGroup[tas_, tp_, n_, OptionsPattern []] := Module[{k, A, An, V, K},
          k = Length[tas];
          A = <|({#} → tas[#]@tp) & /@ Range[k]|> ;
          Do[An = <|(# → TasTF[#, tas]@tp) & /@ Tuples[Range[k], j]|> ;
           A = Join[A, An], {j, 2, n}];
          V = DeleteDuplicates[Values[A], Norm[#1 - #2] < OptionValue[tol] &];
          K = Table[First[nFindKeys[A, V[[i]]]], {i, Length[V]}];
          Echo[Length[V], "number of group elements calculated "];
          K]
```

To use this we need to enter a fixed pseudorandom test point and an integer n bounding the length of a key. Here are some examples, the first using the transformations ρ, σ above

```
In[ ]:= finiteTransGroup[<|1 → σ, 2 → ρ|>, {2.316, -1.347}, 3]
```

```
» number of group elements calculated  4
```

```
Out[ ]= {{1}, {2}, {1, 1}, {1, 2}}
```

We have already seen the transformations given by these .

For a more interesting group let ρ be as above and

```
In[ ]:= κ = N[RotationTransform [Pi / 3, {0, 0}]]
```

```
Out[ ]= TransformationFunction [( 0.5       -0.866025 | 0.
                                  0.866025   0.5      | 0.
                                  ----------------------+--
                                  0.         0.       | 1. )]
```

```
In[ ]:=
```

```
In[ ]:= finiteTransGroup [<| 1 → κ, 2 → ρ |> , {2.316 , -1.347}, 3]
```

» number of group elements calculated 10

```
Out[ ]= {{1}, {2}, {1, 1}, {1, 2}, {2, 1}, {2, 2}, {1, 1, 1}, {1, 1, 2}, {2, 1, 1}, {2, 1, 2}}
```

We can check that all elements have been found by running again

```
In[ ]:= G2 = finiteTransGroup [<| 1 → κ, 2 → ρ |> , {2.316 , -1.347}, 4]
```

» number of group elements calculated 12

```
Out[ ]= {{1}, {2}, {1, 1}, {1, 2}, {2, 1}, {2, 2}, {1, 1, 1},
         {1, 1, 2}, {2, 1, 1}, {2, 1, 2}, {1, 1, 1, 1}, {1, 1, 1, 2}}
```

We find two new elements, again

```
In[ ]:= finiteTransGroup [<| 1 → κ, 2 → ρ |> , {2.316 , -1.347}, 5]
```

» number of group elements calculated 12

```
Out[ ]= {{1}, {2}, {1, 1}, {1, 2}, {2, 1}, {2, 2}, {1, 1, 1},
         {1, 1, 2}, {2, 1, 1}, {2, 1, 2}, {1, 1, 1, 1}, {1, 1, 1, 2}}
```

So we probably have them all. What is interesting is that if we pick a non-random test point which is a vertex of a polygon which has this isometry transformation group then several group elements have the same test value.

```
In[ ]:= finiteTransGroup [<| 1 → κ, 2 → ρ |> , {0, 1}, 5]
```

» number of group elements calculated 6

```
Out[ ]= {{1}, {2}, {1, 1}, {1, 2}, {2, 2}, {1, 1, 2}}
```

We can recover a polygon with this transformation group using the the group association

```
In[ ]:= groupAssoc [G_, tas_, tp_] := Module[{A},
           A = <| Table[k → TasTF[k, tas]@tp, {k, G}] |> ;
           V = DeleteDuplicates [Values[A], Norm[#1 - #2] < .001 &];
           K = Table[First[nFindKeys [A, V⟦i⟧]], {i, Length[V]}];
           <| Table[k → A[k], {k, K}] |> ]
```

```
In[ ]:= G2A = groupAssoc [G2 , <| 1 → κ, 2 → ρ |> , {0, 1}]
```

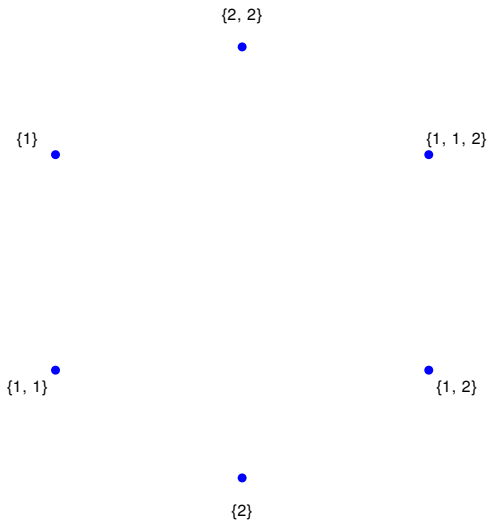
```
Out[ ]= <| {1} → {-0.866025 , 0.5}, {2} → {0. , -1.}, {1, 1} → {-0.866025 , -0.5},
           {1, 2} → {0.866025 , -0.5}, {2, 2} → {0. , 1.}, {1, 1, 2} → {0.866025 , 0.5} |>
```

We can plot them

```
In[ ]:= gpAssocGraph2D [gpAs_] :=
 Graphics[{{Blue, PointSize[.02], Point[Values[gpAs]]},
   {Black, Table[Text[k, 1.15 gpAs[k]], {k, Keys[gpAs]}]}}, ImageSize → 250]
```

```
In[ ]:= gpAssocGraph2D [G2A]
```

Out[ ]=



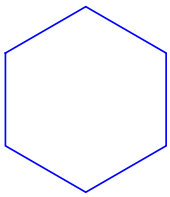
We have a regular hexagon . Using the results of the last two outputs we can draw the polygon

```
In[ ]:= P6 = {G2A[{1}], G2A[{2, 2}], G2A[{1, 1, 2}], G2A[{1, 2}], G2A[{2}], G2A[{1, 1}], G2A[{1}]}
```

```
Out[ ]= {{-0.866025 , 0.5}, {0. , 1.}, {0.866025 , 0.5},
  {0.866025 , -0.5}, {0. , -1.}, {-0.866025 , -0.5}, {-0.866025 , 0.5}}
```

```
In[ ]:= Graphics[{Blue, Thickness[.01], Line[P6]}, ImageSize → Tiny]
```

Out[ ]=



Unfortunately we need human or artificial intelligence to find the adjacencies. We will discuss these ideas further in the next two chapters .

We end this chapter by noting that even if we stick to transformations of finite order we may not get a finite group. Recall

```
In[ ]:= K
```

```
Out[ ]= TransformationFunction [( 0.5       -0.866025 | 0. )
                                ( 0.866025    0.5      | 0. )
                                ( 0.          0.       | 1. )]
```

Now let

In[ ]:= **λ = N[RotationTransform [Pi / 3, {1, 1}]]**

Out[ ]= TransformationFunction $\left[\left(\begin{array}{cc|c} 0.5 & -0.866025 & 1.36603 \\ 0.866025 & 0.5 & -0.366025 \\ \hline 0. & 0. & 1. \end{array}\right)\right]$

In[ ]:=
```
finiteTransGroup [<| 1 → κ, 2 → λ |> , {2.316 , -1.347}, 3];
finiteTransGroup [<| 1 → κ, 2 → λ |> , {2.316 , -1.347}, 4];
finiteTransGroup [<| 1 → κ, 2 → λ |> , {2.316 , -1.347}, 5];
finiteTransGroup [<| 1 → κ, 2 → λ |> , {2.316 , -1.347}, 6];
finiteTransGroup [<| 1 → κ, 2 → λ |> , {2.316 , -1.347}, 7];
```

» number of group elements calculated 13

» number of group elements calculated 23

» number of group elements calculated 37

» number of group elements calculated 56

» number of group elements calculated 78

The number of elements appears to be growing quickly, the sign of an infinite group.

One explanation of this is the well known theorem

Theorem[Yale, 3.6] A finite group of isometries has at least one point left fixed by all elements of the group. In 2D except for the group consisting of the identity and one reflection that fixed point is unique.

The proof of the first statement is easy, if there are n elements take one random test point and apply each transformation in the group to this test point. The (sum of the test values)/n is the centroid of the test values and a fixed point. Example, take G2 above.

In[ ]:= **G2A2 = groupAssoc [G2 , <| 1 → κ, 2 → ρ |> , {2.316 , -1.347}]**

Out[ ]=
```
<|{1} → {2.32454 , 1.33221}, {2} → {2.316 , 1.347}, {1, 1} → {0.00853622 , 2.67921},
 {1, 2} → {-0.00853622 , 2.67921}, {2, 1} → {2.32454 , -1.33221},
 {2, 2} → {2.316 , -1.347}, {1, 1, 1} → {-2.316 , 1.347}, {1, 1, 2} → {-2.32454 , 1.33221},
 {2, 1, 1} → {0.00853622 , -2.67921}, {2, 1, 2} → {-0.00853622 , -2.67921},
 {1, 1, 1, 1} → {-2.32454 , -1.33221}, {1, 1, 1, 2} → {-2.316 , -1.347}|>
```

In[ ]:= **Total[Values[G2A2]] / 12**

Out[ ]= $\{2.22045 \times 10^{-16}, -1.85037 \times 10^{-17}\}$

The origin is the unique fixed point .