11.07 Analytic geometry with 2DGeometry

Concepts

The lineq ADT
Constructing lineq objects
Join and intersection
Testing for concurrence
Unreliability of tests with floating-point input
Additional 2DGeometry point- and line-manipulation functions
Exercises

This section describes the part of this book’s 2DGeometry package that manipulates lines as in conventional plane analytic geometry. It’s used to compute and reason with the coordinates of the points and equations of the lines that constitute a finished drawing.

The lineq ADT

The 2DGeometry package uses objects of an abstract data type lineq to represent lines in geometry. Mathematica’s built-in Line graphics function serves some of the same purposes, but can’t be used here. It’s really intended for representing polygons, and isn’t directly related to any linear equation.

2DGeometry specifies a line as usual, by a triple of linear-equation coefficients: lineq[a,b,c] is the line with equation ax + by + c = 0. A triple a,b,c consists of valid line coefficients only when its first two entries are not both zero. To draw figures, clients will normally construct lines by using functions from this package that are guaranteed to refer to valid triples.

The following definition helps the package react intelligibly to erroneous use of expressions such as lineq[0,0,c]:
lineq[0,0,c_] := noLine

Such errors result in Mathematica’s fruitlessly attempting to handle the undefined symbol noLine as though it were a lineq object. It usually outputs the offending expression with noLine plainly visible.

Selectors are implemented for the coefficients of an equation \( ax + by + c = 0 \) of a line \( g \):

\[
c1[lineq[a_,b_,c_]] ^:= a \\
c2[lineq[a_,b_,c_]] ^:= b \\
c3[lineq[a_,b_,c_]] ^:= c
\]

The equality and inequality operators for lines are implemented next. Two coefficient triples represent the same line if they’re dependent. Here are the definitions:

\[
\begin{align*}
\text{lineq}[a1_,b1_,c1_] == \text{lineq}[a2_,b2_,c2_] & := \\
\quad \text{FullSimplify}[a1*b2 == a2*b1 && b1*c2 == b2*c1 && c1*a2 == c2*a1] \\
\text{lineq}[a1_,b1_,c1_] \neq \text{lineq}[a2_,b2_,c2_] & := \\
\quad \text{Not}[\text{lineq}[a1, b1, c1] == \text{lineq}[a2, b2, c2]]
\end{align*}
\]

noLine == noLine ^= badEquation
noLine /= noLine ^= badEquation

The following examples show these definitions in action.

\[
\begin{align*}
in: & \quad g = \text{lineq}[1,2,3]; \quad h = \text{lineq}[2,4,6]; \\
& \quad j = \text{lineq}[2,4,7]; \quad k = \text{lineq}[0,0,1]; \\
& \quad \{g == h, h == j, j \neq j\} \\
& \quad \{j == k, k == k, k \neq k\}
\end{align*}
\]

\[
\begin{align*}
out: & \quad \{\text{True, False, False}\} \\
& \quad \{\text{lineq[2,4,7] == noLine, badEquation, badEquation}\}
\end{align*}
\]

The second list of three examples consists of intentional errors. The first output resulted from the fact that \( == \) has not been defined when one of its operands is the symbol noLine. You might reason that equation \( k == k \) would produce the analogous output noLine == noLine. But Mathematica automatically gives this the value True. That’s why two special definitions were included to return the undefined symbol badEquation.
**Constructing lineq objects**

The line through point \( P = <x_0, y_0> \) with slope \( m \) has equation

\[
y - y_0 = m(x - x_0), \quad \text{i.e., } mx - y + y_0 - mx_0 = 0.
\]

Constructors are implemented for this and for the horizontal and vertical lines through \( P \):

\[
\begin{align*}
\text{pointSlope}[P_,m_] & := \text{lineq}[m,-1, c2[P] - m*c1[P]] \\
\text{horizontal}[P_] & := \text{pointSlope}[P,0] \\
\text{vertical}[P_] & := \text{lineq}[1,0,-c1[P]]
\end{align*}
\]

Examples:

\[
\begin{align*}
in: \quad & P = \text{Point}[1,2]; \\
& \{\text{pointSlope}[P, 1], \text{horizontal}[P], \text{vertical}[P]\}
\end{align*}
\]

\[
\begin{align*}
out: \quad & \{\text{lineq}[1,-1,1], \text{lineq}[0,-1,2], \text{lineq}[1,0,-1]\}
\end{align*}
\]

The outputs represent equations \( x - y + 1 = 0 \), \( -y + 2 = 0 \), and \( x - 1 = 0 \).

Structured similarly to pointSlope are two functions that represent the lines

\[
h_1: \quad a_1x + b_1y + c_1 = 0 \quad \quad h_2: \quad a_2x + b_2y + c_2 = 0
\]

through a point \( P<x_0,y_0> \) parallel and perpendicular to a given line

\[
g: \quad ax + by + c = 0.
\]

Since \( h_1 \parallel g \) and \( h_2 \perp g \), you can take \( a_1, b_1 = a, b \) and \( a_2, b_2 = -b, a \). Since \( P \) lies on \( h_1, h_2 \),

\[
\begin{align*}
0 = a_1x_0 + b_1y_0 + c_1 &= ax_0 + by_0 + c_1, \\
0 = a_2x_0 + b_2y_0 + c_2 &= -bx_0 + ay_0 + c_2.
\end{align*}
\]

You can solve those equations for \( c_1, c_2 \). Here are the definitions of the corresponding functions:
pointParallel[P_, g_] :=
  lineq[c1[g], c2[g], -c1[g]c1[P] - c2[g]c2[P]];
pointPerpendicular[P_, g_] :=
  lineq[-c2[g], c1[g], c2[g]c1[P] - c1[g]c2[P]];

You should provide examples of the behavior of these functions, as correctly used and
in error situations.

### Join and intersection

The line $P \lor Q$ through points $P = <x_1, y_1>$ and $Q = <x_2, y_2>$ that aren’t vertically
aligned has equation $y - y_1 = m(x - x_1)$, where

$$m = \frac{y_2 - y_1}{x_2 - x_1}.$$ 

Algebraic manipulation converts that to the form

$$(y_2 - y_1)x + (x_1 - x_2)y + (x_2y_1 - x_1y_2) = 0.$$ 

This equation is correct even if $P$ and $Q$ are vertically aligned, as long as $P \neq Q$. The
line $P \lor Q$ is called the *join* of $P$ and $Q$. *2DGeometry* implements this operation by
overloading the $\lor$ operator$^1$:

```mathematica
Point[x1_, y1_] \lor Point[x2_, y2_] \
^:= lineq[y2-y1, x1-x2, x2*y1 - x1*y2]
```

Examples:

```
in:  P = Point[1,2];  Q = Point[3,4];
      {P \lor Q, P \lor P}
out:  {lineq[2,-2,2], noLine}
```

Nonparallel lines $g, h$ with equations $a_1x + b_1y + c_1 = 0$, $a_2x + b_2y + c_2 = 0$
intersect when

$\quad$

---

$^1$ You can enter *Mathematica* operator $\lor$ by typing `\(\text{	extbackslash Vee}\)`. It has no preassigned meaning.
\[
\begin{align*}
    \begin{cases}
        a_1x + b_1y &= -c_1 \\
        a_2x + b_2y &= -c_2
    \end{cases}
\end{align*}
\]

\[
x = \frac{1}{d} \det \begin{bmatrix} -c_1 & b_1 \\ -c_2 & b_2 \end{bmatrix} \quad \quad y = \frac{1}{d} \det \begin{bmatrix} a_1 & -c_1 \\ a_2 & -c_2 \end{bmatrix}
\]

\[
d = \det \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}.
\]

Their intersection \( g \cap h \) consists of the single point \( <x, y> \). The following definition implements this concept via the \( \cap \) operator:

\[
\text{lineq}[a1\_,b1\_,c1\_] \cap \text{lineq}[a2\_,b2\_,c2\_] := \text{Module}\{d\},
\]

\[
d = a1*b2-a2*b1;
\]

\[
(1/d)\text{Point}[b1*c2-b2*c1, c1*a2-c2*a1]
\]

Examples (all but the first are intentional errors):

\[
in: \quad g = \text{lineq}[1,-1,0]; \quad h = \text{lineq}[1,1,-1];
\quad j = \text{lineq}[1,-1,1]; \quad k = \text{lineq}[0,0, 1];
\quad \{g \cap h, g \cap g, g \cap j, g \cap k\}
\]

\[
out^3: \quad \{\text{Point}[1/2,1/2], \text{noIntersection}, \text{noIntersection},
\quad \text{noLine} \cap \text{lineq}[1,-1,0]\}
\]

**Testing for concurrence**

Reasoning in analytic geometry, such as the demonstration in the following section, often involves deciding whether three lines are concurrent or mutually parallel.\(^4\) (For this discussion, every line is regarded as parallel to itself.) The question is whether a system

---

\(^2\) The \( \cap \) operator is commonly used for the intersection of two lists.

\(^3\) The output for \( g \cap k \) is misleading, and produces two error messages, probably due to the conventional use of \( \cap \) with lists.

\(^4\) However, drawing figures—the principal intended application of 2DGeometry—rarely calls for that decision. You usually know the answer before you start drawing.
of equations for those lines has a solution \( <x, y> \) or if the lines are mutually parallel. If there is a solution, then there’s a nontrivial solution \( <x', y', z'> \) of the related equations

\[
\begin{align*}
    a_1'x' + b_1'y' + c_1'z' &= 0 \\
    a_2'x' + b_2'y' + c_2'z' &= 0 \\
    a_3'x' + b_3'y' + c_3'z' &= 0
\end{align*}
\]  

(1)

Conversely, if this determinant is zero, then there’s a nontrivial solution \( <x', y', z'> \) of equations (2). If \( z' \neq 0 \) then \( <x, y> = <x'/z', y'/z'> \) is a solution of equations (1). Finally, equations (2) have a nontrivial solution \( <x', y', z'> \) with \( z' = 0 \) just in case there’s a nonzero vector \( <x', y'> \) perpendicular to all three lines. Such a vector exists just when the lines are mutually parallel. In summary, determinant (3), built from the coefficients of three lines, is zero if and only if the lines are concurrent or mutually parallel.

This 2DGeometry function implements some of the considerations of the previous paragraph:

```mathematica
concurrentQ[lineq[a1_,b1_,c1_],
    lineq[a2_,b2_,c2_],
    lineq[a3_,b3_,c3_]] ^:=
    FullSimplify[Det[{{a1,b1,c1},
                      {a2,b2,c2},
                      {a3,b3,c3}}] == 0]
```

It returns True just when the three lines are concurrent or mutually parallel. Here are some simple examples of its use:
in:  
g = lineq[1,2,3];  h = lineq[4,5,6];
P = g \cap h;  j = pointPerpendicular[P,g];
k = pointParallel[origin,j];
l = pointParallel[2P,j];
m = lineq[0,0,1];
{concurrentQ[g,h,j], concurrentQ[g,h,k]}
{concurrentQ[j,k,l], concurrentQ[j,k,k]}
concurrentQ[k,l,m]

out:  
{True, False}
{True, True}
concurrentQ[lineq[-2,1,0], lineq[-2,1,8], noLine]

The input arranged for point \( P \) to lie on lines \( g, h, j \), ensured that \( g, h, k \) are not concurrent, and that \( j, k, l \) are mutually parallel. The last two examples show that coincident lines are regarded as parallel, and that \( \text{concurrentQ} \) reacts reasonably to incorrect input.

In the previous examples all coordinates were integers. What if some of them are floating-point numbers? For instance, change \( g \) to \( \text{lineq}[0.9,2,3] \), recompute \( P \) and \( j \), and re-execute \( \text{concurrentQ}[g,h,j] \). You’ll get \( \text{False} \)! What has happened? The presence of floating-point input has forced \textit{Mathematica} to use floating-point arithmetic throughout, and round-off error has accumulated while computing determinant \( d \). By inserting a \texttt{Print} command in the \texttt{concurrentQ} code, you can see that \( d \approx -3.0 \times 10^{-15} \), which \textit{Mathematica} evidently regards as nonzero.

You can usually trust \( \text{concurrentQ} \) to give correct answers when its input coordinates are integers or algebraic formulas. Even then, evaluation of this function will fail if the determinant formula is so complicated that \textit{Mathematica} can’t determine whether its value is zero. With floating-point input, you simply can’t trust \( \text{concurrentQ} \), or other \textit{2DGeometry} functions that rely on testing whether some number \( x \) is zero. An alternative approach would be for such functions to ask whether \( x \) is close enough to zero. But \textit{close enough} is too hard to define to consider that in this book. For example, it would depend on the size of input values and probably on the intended use of the output.
Additional operations

Section 11.2 defined function distance to compute the distance between two points. Here, that name is overloaded to provide a function to compute the distance between a point and a line:

\[
\text{distance}[\text{Point}[x_,y_], \text{lineq}[a_,b_,c_]] := \\
\text{Module}[\{d\}, d = \text{Sqrt}[a^2 + b^2]; \\
\text{Abs}[a*x + b*y + c]/d]
\]

Another function defined in the previous section returned the slope of the segment determined by two Point objects. Here, its name is overloaded to compute the slope of a lineq object:

\[
\text{slope}[\text{lineq}[a_,b_,c_]] := \\
\text{If}[\text{FullSimplify}[b == 0], \text{noSlope}, -a/b]
\]

You can provide examples of the behavior of these functions, as correctly used and in error situations.

The 2DGeometry package includes several more Boolean functions that test relationships between points and lines:

\begin{align*}
\text{horizontalQ} & \quad \text{onQ} & \quad \text{parallelQ} & \quad \text{collinearQ} \\
\text{verticalQ} & \quad \text{throughQ} & \quad \text{perpendicularQ}
\end{align*}

Exercises 1, 3, and 5 specify their actions. You’re invited to try your hand at programming them, and to compare your code with the package’s.

Exercises

1. Part 1. 2DGeometry includes functions that you can execute as follows, with any lineq and Point objects g and P:

---

5 For a derivation of the distance formula employed in this function, see [??].

6 These seven functions, like concurrentQ, are unreliable with floating-point input.
In this chapter, \textit{between} is construed inclusively. That is, \(Q\) is regarded as lying between \(P\) and \(R\) even if \(Q = P\) or \(Q = R\) or both.

\textit{Parallel} is construed inclusively. That is, every line is regarded as parallel to itself.

They’re all Boolean functions. The first two return \texttt{True} just when \(g\) is horizontal or vertical; the second two, when it passes through \(P\). Without consulting the package, write code for these functions. Test it on known \texttt{True} and \texttt{False} cases, and on a case with invalid \(g\) input.

\textit{Part 2.} Using floating-point input and various \texttt{2DGeometry} functions, construct lines \(g, h\) and point \(Q = g \cap h\) such that \texttt{onQ[Q,h]} is \texttt{False}. (This shows that like \texttt{concurrentQ} the functions in this exercise are unreliable with floating-point input.)

\texttt{perpendicularQ[g,h]} \quad \texttt{parallelQ[g,h]}

These Boolean functions return \texttt{True} just when \(g\) and \(h\) are parallel\(^8\) or perpendicular. Without consulting the package, write code for these functions. Test it on known \texttt{True} and \texttt{False} cases, and on a cases with invalid input. \textit{Hint:} for \texttt{parallelQ} consider the code for the \texttt{==} operation. (Because these functions use the \texttt{==} operator, they’re not reliable with floating-point input.)

\textit{Part 1.} Construct a rhombus \(OPQR\), where \(O\) is the origin, as follows. Select two values of \(\theta\) (in radians) for which \texttt{Sin} and \texttt{Cos} return algebraic formulas; they should not differ by any multiple of \(\pi/2\). Use \texttt{cis} with these to construct points \(P\) and \(Q\) on the unit circle. Use \texttt{2DGeometry} functions to construct \(R\). Now use \texttt{2DGeometry} functions to construct the diagonal lines of the rhombus, and verify that they’re perpendicular.

---

\(^7\) In this chapter, \textit{between} is construed inclusively. That is, \(Q\) is regarded as lying between \(P\) and \(R\) even if \(Q = P\) or \(Q = R\) or both.

\(^8\) In this chapter, \textit{parallel} is construed inclusively. That is, every line is regarded as parallel to itself.
Part 2. What happens if you choose rational $\theta$ values for which $\sin$ and $\cos$ don’t return formulas? What if you use even one floating-point $\theta$ value? Why?

5. *2DGeometry* includes a function `collinearQ` such that `collinearQ[P,Q,R]` is True just when all of the points $P$, $Q$, $R$ fall on a single line. Without consulting the package, write code for it. Test that with integer input on known True and False cases. Find a case where it fails with floating-point input. (Like other *2DGeometry* functions that test for equality, `collinearQ` is unreliable with floating-point input.)

6. Three points are collinear just when a certain determinant $d$ involving their coordinates is zero. Function `collinearQ` computes $d$. Modify its code to output $d$. How can you predict the sign of $d$ by looking at a picture of the points?

7. Write a Boolean function `betweenQ` that applies to three (real) numbers $t, u, v$ and returns True just when $u$ lies between $t$ and $v$. Test it on known True and False cases.

8. Write and test code for a Boolean function `between2Q` with Point arguments $P, Q, R$. It should return True just when $Q$ lies between $P$ and $R$. Your code should be based on the fact that this condition holds just when

i. they’re collinear, and

ii. one of these conditions holds:

   (1) they’re horizontally aligned and their $x$-coordinates fall in order, or

   (2) they’re not, and their $y$-coordinates fall in order.

You may use the function `betweenQ` of exercise 7. Test `between2Q` on known True and False cases. Find an example where it fails. Why did it fail? (If you completed exercise 2, compare with that result.)