11.2 Manipulating points

This section describes the part of this book’s twoDG package that manipulates points as in conventional plane analytic geometry, without drawing them. It’s used to compute and reason with the coordinates of points involved in a geometric figure. For more geometry, involving lines and circles, see sections 11.7–11.9.

Names of functions and constants in this package begin with lowercase letters, to distinguish them from Mathematica’s built-in functions. They often include uppercase letters to enhance readability. Names of Boolean-valued functions end with Q. This book will use upper- and lowercase Latin letters to designate points and lines as is common in geometry. Since Mathematica doesn’t allow \( C, D, E, I, N, \) or \( O \) to be global variables, its corresponding script letters \( Ĉ, Ď, Ė, Į, Ė, \) and \( Ĉ \) may be used instead.

**The Point ADT**

Mathematica graphics features and the twoDG package use objects of an abstract data type Point to represent points in geometry. A point is specified as usual by a pair of real coordinates: Point\([\{x,y\}]\) represents the point with coordinates \( x \) and \( y \). You can think of formula Point\([\{x,y\}]\) as constructing an object of this type. But there’s no function Point; the object is the (full form of the) expression itself.
The package’s first definition lets you use the slightly more convenient expression `Point[x,y]` to construct the point with coordinates `x` and `y`. This, and several other definitions in this section, require changing the built-in properties of the symbol `Point`. Such definitions are usually regarded as blunders or bad practice, so `Mathematica` protects symbols with built-in properties. You can’t change their properties without removing the protection. In this case, `Point` has almost no built-in properties, so there’s little chance of conflict. The author decided that introducing a new similarly named symbol with the necessary properties presented greater risk of confusion than augmenting the properties of `Point`.

Here’s how to do that. First, interrogate `Point` to verify that there are no built-in definitions to cause conflict:

```
in:  ??Point
out: Point[coords] is a graphics primitive that represents a point.
     Attributes[Point] = {Protected}
```

Now, remove the protection\(^1\) and define the new constructor:

```
in:  Unprotect[Point];
     Point[x_,y_] := Point[{x,y}]
```

Interrogating after the `Unprotect` statement would reveal absence of the `Attributes` list. Interrogating after the definition shows it recorded among the properties.

The next definitions use this constructor to provide a constant denoting the origin of a Cartesian coordinate system, and a constructor for the point on the unit circle with angle parameter \(\theta\):

```
origin := Point[0,0]  
cis[\theta_] := Point[Cos[\theta],Sin[\theta]]\(^2,3\)
```

Selectors are implemented for the coordinates \(c_1(P)\) and \(c_2(P)\) of a point \(P\):

---

\(^1\) Without this step the definition would merely return an error message. The final codes in the `twoDG` package include `Protect[Point]` to restore protection.

\(^2\) This notation stems from the common use of `\text{cis} \theta` as an abbreviation for the corresponding point `\cos \theta + i \sin \theta` in the complex plane.

\(^3\) You can use degree measure, too—for example, `\text{cis}[10^\circ]`. The `Mathematica` symbol `\^` merely stands for the conversion constant \(\pi/180\).
As noted in the previous section, it’s useful to define the behavior of the equality and inequality operators for \texttt{Point} objects:

\begin{align*}
\texttt{Point}[P] \ == \ & \texttt{Point}[Q] \ := \ \texttt{FullSimplify}[P \ == \ Q] \\
\texttt{Point}[P] \ != \ & \texttt{Point}[Q] \ := \ \texttt{Not}[\texttt{Point}[P] \ == \ \texttt{Point}[Q]]
\end{align*}

This works because \textit{Mathematica} judges two lists equal just when their entries coincide. \texttt{FullSimplify} is employed because without it, \textit{Mathematica} will fail to evaluate the equations when algebraic formulas involving square roots are substituted for the coordinates of \(P\) and \(Q\). The analytic geometry tasks for which \texttt{twoDG} is designed often involve such formulas. Even with \texttt{FullSimplify}, \textit{Mathematica} may fail to evaluate some equations with complicated algebraic formulas. If that situation occurs, the resulting output mess is not helpful for debugging. Don’t use \(==\) or \(\neq\) to compare \texttt{Point} objects unless you’re prepared for that.

The first nonlogical operation on \texttt{Point} objects computes distance:

\begin{verbatim}
distance[Point[{x1_,y1_}], Point[{x2_,y2_}]] ^:= \\
\phantom{distance[Point[{x1_,y1_}], Point[{x2_,y2_}]]} \texttt{Sqrt}[(x1-x2)^2 + (y1-y2)^2]
\end{verbatim}

The definition \(\text{distance}[P,Q] := \text{Sqrt}[(c1[P]-c1[Q])^2 + (c2[P]-c2[Q])^2]\) would have been possible too, but that would apply to all arguments \(P\) and \(Q\). The definition with \(^:=\) was chosen instead because another function will be introduced in section 11.7 to compute distances between points and lines.

The following example demonstrates these features and shows that no explicit definition is needed for the assignment operator:

\begin{verbatim}
in: \ P = \texttt{Point}[1,2]; \\
\{P, c1[P], P == \texttt{origin}, P \neq \texttt{origin}, \text{distance}[\texttt{origin},P]\}

out: \{\texttt{Point}[\{1,2\}], 1, \text{False}, \text{True}, \sqrt{5}\}
\end{verbatim}
Vectors

Because analytic geometry is often simplified by treating points as vectors, the *twoDG* package needs vector operations. The following definitions overload the addition and multiplication operators:

\[
\text{Point}[P_] + \text{Point}[Q_] := \text{Point}[P + Q]
\]
\[
t_*\text{Point}[P_] := \text{Point}[t*P]
\]

These work because *Mathematica* treats lists like vectors. The following examples show that subtraction and the juxtapositional form of the multiplication operator are implemented automatically:

\[
\text{in: } \quad P = \text{Point}[1,2]; \\
\{P + P, P - P\} \\
\{0*P, 1*P, (-1)P\}
\]
\[
\text{out: } \quad \{\text{Point}[\{2,4\}], \text{Point}[\{0,0\}]\} \\
\{\text{Point}[\{0,0\}], \text{Point}[\{1,2\}], \text{Point}[\{-1,-2\}]\}
\]

The previous definitions evidently overrode *Mathematica*’s rule that \(x - x = 0\) for any \(x\). But subtraction worked because its rule \(x - y = x + (-1)y\) is still valid. Negation has been implemented automatically, too.

Because *Mathematica* assumes that the \(*\) operator is commutative, the previous definition applies to \(P*t\) also. Check some examples.\(^4\)

You can now employ vector operations, as in the following familiar definition\(^5\):

\[
\text{midpoint}[P_,Q_] := (1/2)(P + Q)
\]

For example, consider

\(^4\) This may be unfortunate. The definition also yields some nonsensical results. Check \(P*P\), for example. Some problems like these could be avoided, but at considerable cost, and others would inevitably persist.

\(^5\) According to an earlier paragraph this definition with := has a disadvantage. It applies to all *Mathematica* objects \(P\) and \(Q\). But no other definition is planned, and misapplications such as \(\text{midpoint}[1,2]\) — which returns the value \(\frac{3}{2}\) — are unlikely to cause trouble.
Measuring angles

Drawing geometric figures requires a system for determining directions and angles. This is harder than you might expect. Analytic geometry texts mislead you—their exercises rely on visual cues. The twoDG package must incorporate detailed analysis instead.

It seems useful to start by observing that a measure of an angle is the difference between the directions of two lines or line segments. Analytic geometry texts use the notion of slope in place of direction. The twoDG package includes a corresponding function:

\[
\text{slope}[\text{Point}[[x_1, y_1]], \text{Point}[[x_2, y_2]]] := \text{Module}[[dx, dy],
\text{dx} = x_2 - x_1; \quad \text{dy} = y_2 - y_1;
\text{If}[\text{Simplify}[\text{dx} == 0], \text{noSlope}, \text{dy}/\text{dx}]]
\]

If the two points are vertically aligned or coincident, the function returns a special symbol noSlope, which has no associated value. That usually results from an error; the special symbol provides a signal to distinguish this from other errors.

But for graphics, slope is not a very useful concept. Each slope corresponds to two opposite directions. Moreover, although vertical line segments have no slopes, they do have directions. The following direction function returns the radian measure $\theta$ of angle $\angle POQ$ in the interval $0 \leq \theta < 2\pi$, where $P$ is a point to the right of $O$. If $O = Q$, there’s no such angle, so the function returns a special symbol noDirection, which has no associated value. The definition reflects the unexpected logical complexity of this operation:

\[
\text{direction}[[O_1, Q_1]] := \text{Module}[[dx, dy, \theta],
\text{dx} = c_1[Q] - c_1[O]; \quad \text{dy} = c_2[Q] - c_2[O];
\text{If}[\text{dx} == 0,
\text{If}[\text{dy} == 0, \text{noDirection},
\text{If}[\text{dy} < 0, 3\pi/2, \pi/2]],
\theta = \text{ArcTan}[\text{dy}/\text{dx}];
\text{If}[\text{dx} < 0, \theta + \pi,
\text{If}[\text{dy} < 0, \theta + 2\pi, \theta]]]]
\]

(* $-\pi/2 < \theta < \pi/2$ *)
The nested If’s make the code very hard to read. In most cases, direction returns an angle $\theta$ whose tangent is the slope $dy/dx$. But there’s no such angle if $dx = 0$. In that case, the first three If’s cause the function to return noDirection, $\frac{3\pi}{2}$, or $\frac{\pi}{2}$ depending on whether $dy$ is zero, negative, or positive. When $dx$ is nonzero, the arctangent provides a value of $\theta$. Since the range of the arctangent is $-\frac{\pi}{2} < \theta < \frac{\pi}{2}$, not the desired interval, $\theta$ may need adjustment. In the first quadrant, where $dx$ and $dy$ are both nonnegative, no adjustment is required, and the final two If’s ensure that direction returns $\theta$. In the second and third quadrants, it returns $\theta + \pi$, and in the fourth, $\theta + 2\pi$. Here are example direction values, from a center $Oh$ with integer coordinates, to the eight nearest points with integer coordinates; they’re followed by a ninth example to show how direction behaves with incorrect input $Oh, Oh$.

```
in: Oh = Point[1, 1];
{direction[Oh,Point[2,1]], direction[Oh,Point[2,2]],
direction[Oh,Point[1,2]], direction[Oh,Point[0,2]],
direction[Oh,Point[0,1]], direction[Oh,Point[0,0]],
direction[Oh,Point[1,0]], direction[Oh,Point[2,0]]}
direction[Oh,Oh]

out: {0, \pi/4, \pi/2, 3\pi/4, \pi, 5\pi/4, 3\pi/2, 7\pi/4}
noDirection
```

It might prove useful to apply FullSimplify in the direction function when asking whether $dx$ and $dy$ are zero, negative, or positive. The present definition doesn’t, because the function will almost always be used in drawing figures rather than reasoning about them. For drawing you generally employ floating-point coordinates, for which FullSimplify is unnecessary. Similar comments hold for other twoDG functions. If you don’t explicitly use floating-point coordinates, you may encounter some situations where Mathematica leaves a function unevaluated because it can’t determine whether an algebraic expression is zero, negative, or positive.

If $O \neq Q, R$, the radian measure $\theta$ of $\angle QOR$ in the interval $0 \leq \theta < 2\pi$, with the usual positive sense, is

$$(\text{direction}[O,R] - \text{direction}[O,Q]) \mod 2\pi.$$ 

The conventional directed angle $\delta$, though, is measured in the interval $-\pi < \delta \leq \pi$; and the conventional nondirected angle is $|\delta|$. Here are the corresponding twoDG functions:
directedAngle[Q_, O_, R_] := Module[{
θ = Mod[direction[O, R] - direction[O, Q], 2π];
If[θ > π, θ = θ - 2π, θ]
},
angle[Q_, O_, R_] := Abs[directedAngle[Q, O, R]];

Here are examples with output in degree measure:

in:  
P = Point[1,2]; Q = Point[3,4];
directedAngle[Q,P,origin]/° //N
angle[Q,P,origin]/° //N

out:  
-161.565
161.565

The built-in Mathematica constant ° used in this example has value π/180. Dividing by ° converts radian to degree measure. (Multiplying would do the opposite.) Mathematica notation //N means “apply the numerical-value function N to the value on the left.”

Exercises

1.  
Part 1. Select points O, P, Q, R with integer coordinates such that O, P, R are noncollinear, O ≠ Q, and ray OQ lies in the interior of ∠POR. Use angle to compute the measures α, β, γ of ∠POQ, ∠QOR, ∠POR and == to verify that α + β = γ.
   Part 2. Like part 1, except use floating-point coordinates. In what way does Mathematica behave differently? Which is more informative?

2.  
Part 1. Select noncollinear points P, Q, R with integer coordinates. Use angle to compute the measures α, β, γ of the angles of ΔPQR and == to verify that their sum is 180°.
   Part 2. Like part 1, except use floating-point coordinates. In what way does Mathematica behave differently? Which is more informative?

3.  
Use cis to define distinct points P, Q, R on the unit circle, angle to compute the measure θ of ∠PQR. Use ==, and N if necessary, to verify that θ is half the measure of the arc PR that doesn’t contain Q.
4. **Part 1.** Select noncollinear points \( P, Q, R \) with floating-point coordinates. Use `angle` to compute the measures of the angles of \( \Delta PQR \) and `distance` to compute the lengths of its edges. Verify the law of cosines by comparing one edge with the value given by the law. Verify the law of sines by comparing the three ratios that it says should be equal.

**Part 2.** Like part 1, except use integer coordinates. In what way does *Mathematica* behave differently? (If it happens to behave similarly, report that, then choose different integers until you can report different behavior. Now use `FullSimplify` to get a better result.)

5. The Euclidean norm of a vector is the square root of the sum of the squares of its components. Thus, when you regard a point as a vector, its norm is its distance from the origin. Define a function `norm` that returns this distance.

Select noncollinear points \( P, Q, R \) with floating-point coordinates. Use your function to verify that the norm of their sum is smaller than the sum of their norms.

6. Find three distinct points with integer coordinates such that the norm of their sum equals the sum of their norms. Confirm that with your function `norm`.