Algorithmes géométriques utilisés dans ElCanari

Pierre Molinaro

21 juillet 2019

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Points

Dans ce chapitre, un point P est défini par ses coordonnées $(P.x, P.y)$. Les fonctions sont définies par une extension de `CGPoint` dans le fichier `extension-CGPoint.swift`.

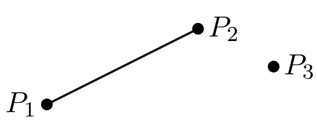
1.1 Distance entre deux points

Élementaire !

```
extension CGPoint {
  static func distance (_ p1 : CGPoint, _ p2 : CGPoint) -> CGFloat {
    let dx = p1.x - p2.x
    let dy = p1.y - p2.y
    return sqrt (dx * dx + dy * dy)
  }
}
```

1.2 Fonction `product`

Cette fonction permet de savoir si un point P_3 est situé *à droite* (comme dans la figure ci-dessous) ou *à gauche* d'un segment P_1P_2. Cette fonction est fondamentale pour de nombreux calculs, comme par exemple l'intersection de deux rectangles.



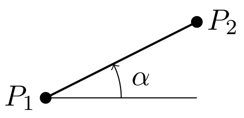
Pour cela, on calcule la composante verticale du produit vectoriel $\overrightarrow{P_1P_2} \wedge \overrightarrow{P_1P_3}$:

— positive, le point P_3 est à gauche du segment P_1P_2 ;
— négative, le point P_3 est à droite du segment P_1P_2 ;
— nulle, le point P_3 est aligné avec le segment P_1P_2.

```
extension CGPoint {
  static func product (_ p1 : CGPoint, _ p2 : CGPoint, _ p3 : CGPoint) -> CGFloat {
    let dx2 = p2.x - p1.x
    let dy2 = p2.y - p1.y
    let dx3 = p3.x - p1.x
    let dy3 = p3.y - p1.y
    return dx2 * dy3 - dx3 * dy2
  }
}
```

1.3 Angle d’un vecteur avec l’horizontal

Étant donnés deux points P_1 et P_2, il s’agit de déterminer l’angle α que fait le vecteur $\overrightarrow{P_1P_2}$ avec l’horizontale :



La fonction `atan2` renvoie l’angle en radian, et gère tous les cas particuliers (voir le *man* de cette fonction) :
— `atan2(+-0, -0)` retourne $\pm\pi$;
— `atan2(+-0, +0)` retourne ± 0;
— `atan2(+-0, x)` retourne $\pm\pi$ pour $x < 0$;
— `atan2(+-0, x)` retourne ± 0 pour $x > 0$;
— `atan2(y, +-0)` retourne $+\pi/2$ pour $y > 0$;
— `atan2(y, +-0)` retourne $-\pi/2$ pour $y < 0$.

```
extension CGPoint {
  static func angleInRadian (_ p1 : CGPoint, _ p2 : CGPoint) -> CGFloat {
    let width = p2.x - p1.x
    let height = p2.y - p1.y
    return atan2 (height, width) // Result in radian
  }
}
```

Chapitre 2

Rectangle horizontal

Par rectangle « horizontal », on entend un rectangle dont les côtés sont parallèles aux axes. Un tel rectangle est décrit par le type `CGRect`.

Dans ce chapitre, des additions à ce type sont présentées. Elles sont définies comme des extensions du type `CGRect`, et implémentées dans `extension-CGRect.swift`.

2.1 Construction d'un rectangle à partir de deux points

Cet initialiseur permet de construire un rectangle à partir de deux points ; si les points sont confondus, la taille du rectangle est nulle.

```
extension CGRect {
  init (point p1: CGPoint, point p2: CGPoint) {
    origin = CGPoint (x: min (p1.x, p2.x), y: min (p1.y, p2.y))
    size = CGSize (width: abs (p1.x - p2.x), height: abs (p1.y - p2.y))
  }
}
```

Chapitre 3

Cercle

Un cercle est caractérisé par son centre et son rayon. C'est un type qui n'existe pas en Cocoa, il est défini dans `ElCanari` par une structure non mutable. Ce type est implémenté dans `GeometricCircle.swift`. Il est simplement construit à partir d'un point et d'un rayon.

```
struct GeometricCircle {
  let center : CGPoint
  let radius : CGFloat

  init (center : CGPoint, radius : CGFloat) {
    self.center = center
    self.radius = radius
  }
}
```

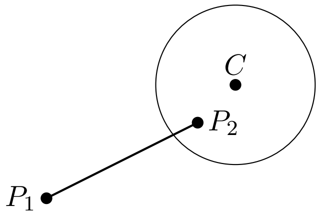
3.1 Intersection entre deux cercles

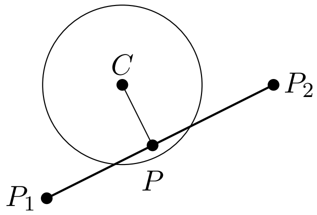
Élémentaire, il y a intersection si la distance entre les centres est inférieure ou égale à la somme des rayons.

```
  func intersects (circle : GeometricCircle) -> Bool {
    let d = CGPoint.distance (self.center, circle.center)
    return d <= (self.radius + circle.radius)
  }
```

3.2 Intersection avec un segment

Voici quelques illustrations des différents cas :





On va calculer l'abscisse du point P, projection de C sur P_1P_2 :

— pour simplifier, on translate les points de façon à amener P_1 en $(0, 0)$;
— l'abscisse curviligne de P est repérée par μ ($\mu = 0$, P est en P_1, $\mu = 1$, P est en P_2).

Les coordonnées de P_2 sont donc : $(p2x, p2y)$, celles de P $(\mu{\cdot}p2x, \mu{\cdot}p2y)$, et celles de C (Cx, Cy).

Le point P est défini par $\overrightarrow{P_1P_2} \,.\, \overrightarrow{PC} = 0$, donc :

$$p2x \cdot (\mu \cdot p2x - Cx) + p2y \cdot (\mu \cdot p2y - Cy) = 0$$

D'où :

$$\mu = \frac{p2x \cdot Cx + p2y \cdot Cy}{p2x^2 + p2y^2}$$

Si $\mu < 0$ ou $\mu > 1$, le point P est en dehors du segment, il n'y a pas d'intersection. Sinon, on calcule la distance entre P et C :

$$d_{PC} = \sqrt{(\mu \cdot p2x - Cx)^2 + (\mu \cdot p2y - Cy)^2}$$

Et il y a intersection si $d_{PC} \leqslant radius$.

```
func intersects (segmentFrom inP1 : NSPoint, to inP2 : NSPoint) -> Bool {
//--- We translate P1, P2, C (center of circle) so that P1 is at (0, 0)
  let p2x = inP2.x - inP1.x
  let p2y = inP2.y - inP1.y
  let Cx = self.center.x - inP1.x
  let Cy = self.center.y - inP1.y
//--- Then we compute the relative abscisse \textmu of P, the projection of C on P1P2
  let \textmu = (p2x * Cx + p2y * Cy) / (p2x * p2x + p2y * p2y)
  if \textmu < 0.0 { // Outside
    return false
  }else if \textmu > 1.0 { // Outside
    return false
  }else{ // Inside: we compute the distance between P and C
    let dx = \textmu * p2x - Cx
    let dy = \textmu * p2y - Cy
    let d = (dx * dx + dy * dy).squareRoot ()
    return d <= self.radius
```

```
    }
  }
```

3.3 Intersection avec un segment (ancien)

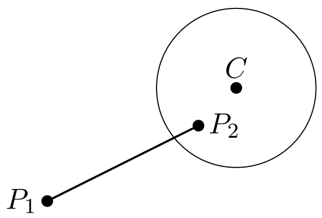
Ancien algorithm. L'intersection entre un cercle et un segment est plus compliquée, il y a plusieurs tests à faire :

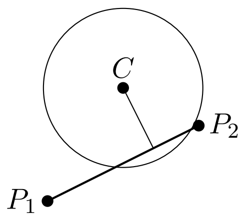
— d'abord, on teste si la distance entre le centre du cercle et les extrémités du segment; si l'une de ces distances est inférieure au rayon du cercle, il y a intersection;

— ensuite, on calcule les coordonnées du centre du cercle dans le repère dont l'origine est le centre du segment, et l'axe des abscisses la direction du segment; il y a intersection si et seulement si :

  — la valeur absolue de l'ordonnée du centre est inférieure au rayon;

  — la valeur absolue de l'abscisse du centre est inférieure à la moitié de la distance entre les deux points.






```
func intersects (segmentFrom p1 : CGPoint, to p2 : CGPoint) -> Bool {
  var intersects = CGPoint.distance (p1, self.center) <= self.radius
  if !intersects {
    intersects = CGPoint.distance (p2, self.center) <= self.radius
  }
  if !intersects {
    let segmentAngle = CGPoint.angleInRadian (p1, p2)
    let segmentCenter = CGPoint (x: (p1.x + p2.x) / 2.0, y: (p1.y + p2.y) / 2.0)
    let tr = CGAffineTransform (rotationAngle: -segmentAngle)
            .translatedBy (x:-segmentCenter.x, y:-segmentCenter.y)
    let point = self.center.applying (tr)
    intersects = abs (point.y) <= self.radius
    if intersects {
      let segmentLength = CGPoint.distance (p1, p2)
      intersects = abs (point.x) <= (segmentLength * 0.5)
    }
  }
```

```
    return intersects
  }
```

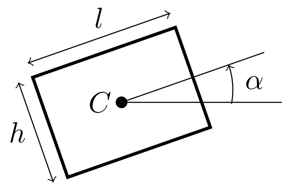
Chapitre 4

Rectangle

Il s’agit de rectangles d’orientation quelconque, ce qui généralise les rectangles « horizontaux » du chapitre 2 page 8.

Un rectangle est caractérisé par :

— son centre C ;
— son angle α avec l’horizontal ;
— sa largeur l ;
— sa hauteur h.



C’est un type qui n’existe pas en Cocoa, il est défini dans `ElCanari` par une structure non mutable. Ce type est implémenté dans `CanariRect.swift`.

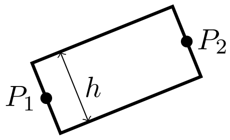
```
struct CanariRect {
  let center : CGPoint
  let angle : CGFloat // In radians
  let size : CGSize

  ...
}
```

4.1 Construction à partir d’un rectangle horizontal (`CGRect`)

```
init (cgrect : CGRect) {
  center = CGPoint (x: NSMidX (cgrect), y: NSMidY (cgrect))
  size = cgrect.size
  angle = 0.0
}
```

4.2 Construction à partir de deux points et d'une hauteur

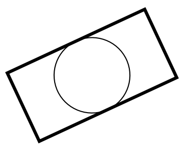



```
init (from p1 : CGPoint, to p2 : CGPoint, height : CGFloat) {
  center = CGPoint (x: (p1.x + p2.x) * 0.5, y: (p1.y + p2.y) * 0.5)
  size = CGSize (width: CGPoint.distance (p1, p2), height: height)
  angle = CGPoint.angleInRadian (p1, p2)
}
```

4.3 Construction à partir du centre, angle et taille

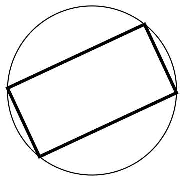
```
init (center : CGPoint, size : CGSize, angle : CGFloat) {
  self.center = center
  self.size = size
  self.angle = angle
}
```

4.4 Cercle inscrit



```
func inscribedCircle () -> CanariCircle {
  let radius = min (size.width, size.height) / 2.0
  return CanariCircle (center: self.center, radius: radius)
}
```

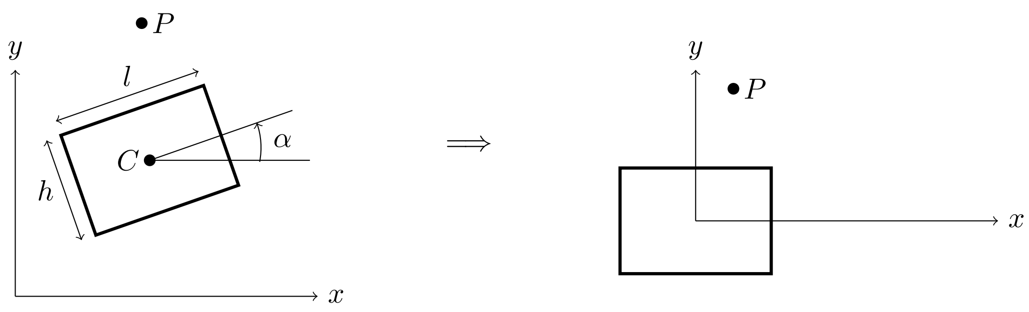
4.5 Cercle circonscrit



```
func circumCircle () -> CanariCircle {
  let radius = sqrt (size.width * size.width + size.height * size.height) / 2.0
  return CanariCircle (center: self.center, radius: radius)
}
```

4.6 Inclusion d'un point

Le test d'inclusion d'un point est plus délicat à mettre au point, bien que le code résultant soit court. Le principe est d'effectuer un changement de repère, de façon à obtenir les coordonnées du point testé dans le repère lié au rectangle, qui devient alors un rectangle horizontal. Tester l'appartenance du point devient élémentaire dans ce nouveau repère.




```
func contains (point p : CGPoint) -> Bool {
  let tr = CGAffineTransform (rotationAngle: -angle)
        .translatedBy (x:-center.x, y:-center.y)
  let point = p.applying (tr)
  return (abs (point.x) <= (size.width * 0.5))
      && (abs (point.y) <= (size.height * 0.5))
}
```

4.7 Coordonnées des sommets

La fonction suivante retourne dans un tableau les quatre sommets du rectangle. Le tableau est ordonné, on parcourt les sommets dans le sens trigonométrique.

```
func vertices () -> [CGPoint] { // Returns the four vertices in counterclock order
  let cosSlash2 = cos (angle) / 2.0
  let sinSlash2 = sin (angle) / 2.0
  let widthCos  = size.width  * cosSlash2
  let widthSin  = size.width  * sinSlash2
  let heightCos = size.height * cosSlash2
  let heightSin = size.height * sinSlash2
  return [
    CGPoint (x: center.x + widthCos - heightSin, y: center.y + widthSin + heightCos),
    CGPoint (x: center.x - widthCos - heightSin, y: center.y - widthSin + heightCos),
    CGPoint (x: center.x - widthCos + heightSin, y: center.y - widthSin - heightCos),
    CGPoint (x: center.x + widthCos + heightSin, y: center.y + widthSin - heightCos)
  ]
}
```

4.8 Intersection avec un cercle

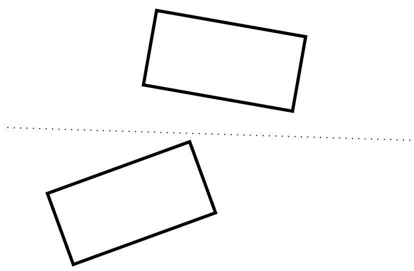
Il y a intersection si :
- si le cercle contient le centre du rectangle;
- ou si le rectangle contient le centre du cercle;
- ou, à défaut, si le cercle présente une intersection avec l'un des quatre côtés du rectangle.

```
func intersects (circle : CanariCircle) -> Bool {
//--- Intersection if circle contains rectangle center
  var intersects = CGPoint.distance (self.center, circle.center) <= circle.radius
//--- Intersection if rectangle contains circle center
  if !intersects {
    intersects = self.contains (point: circle.center)
  }
//--- Test intersection between circle and rectangle edge
  if !intersects {
    let vertices = self.vertices ()
    var i = 0
    while !intersects && (i < vertices.count) {
      intersects = circle.intersects (segmentFrom: vertices [i],
                                      to: vertices [(i+1) % vertices.count])
      i += 1
```

```
        }
    }
    return intersects
}
```

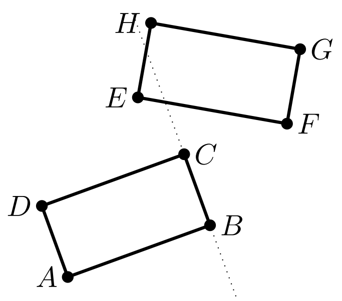
4.9 Intersection avec un autre rectangle

La méthode qui fait autorité est la méthode dite de *séparation d'axes*. Elle est illustrée dans la video https://www.youtube.com/watch?v=WBy6AveIRRs. En résumé, il n'y a pas intersection si il existe une droite pour laquelle un rectangle est complètement contenu dans un demi-plan, et l'autre rectangle complètement contenu dans l'autre demi-plan.



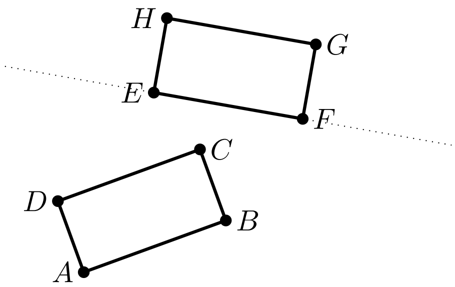
L'intérêt de la méthode est que l'on n'a pas besoin de construire une telle droite, qui d'ailleurs n'est pas unique dans le cas général.

Il faut commencer par construire les sommets des rectangles.



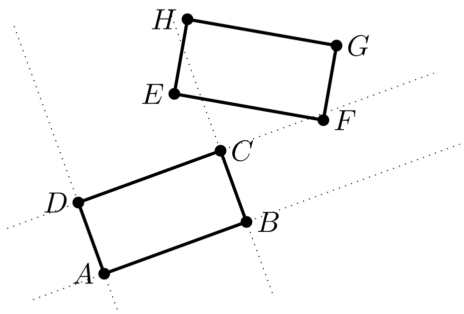
Ensuite, pour chaque côté du premier rectangle, on effectue le *test de séparation* : il est positif si les quatre sommets de l'autre rectangle sont « de l'autre côté ». Par exemple, on considère le coté BC, qui définit une droite qui partage le plan en deux. Par la fonction `CGPoint.product`, on calcule la composante verticale de $\overrightarrow{BC} \wedge \overrightarrow{CD}$: son signe caractérise le demi-plan du premier rectangle. On calcule ensuite la composante verticale de $\overrightarrow{BC} \wedge \overrightarrow{CP}$, pour les quatre sommets P du second rectangle. On obtient un signe contraire pour les sommets F, G, H, mais le même signe pour le sommet E, ce qui fait échouer le test de séparation.

Le test de séparation réussit en considérant le segment EF :



Il faut donc effectuer le test de séparation pour chaque côté du premier rectangle **et** chaque côté du second rectangle. Il y a intersection si **tous** les tests de séparation échouent, donc pas d'intersection si un des tests réussit.

Il est indispensable de faire les tests pour les deux rectangles : en effet, ils peuvent échouer pour chaque côté du premier rectangle, alors qu'il n'y a pas intersection. Par exemple dans la figure suivante, les quatre tests d'isolation échouent pour les quatre côtés du rectangle $ABCD$.



Voici le code de la fonction.

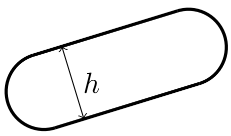
```
func intersects (rect : CanariRect) -> Bool {
//--- Method of separating axes (https://www.youtube.com/watch?v=WBy6AveIRRs)
  var intersects = true
  let vertices1 = self.vertices ()
  let vertices2 = rect.vertices ()
  do{
    var i = 0
    while intersects && (i < vertices1.count) {
      let ref = CGPoint.product (vertices1 [i],
                                 vertices1 [(i+1) % vertices1.count],
                                 vertices1 [(i+2) % vertices1.count])
      var outside = true
      var j = 0
      while outside && (j < vertices2.count) {
```

```
        let test = CGPoint.product (vertices1 [i],
                                    vertices1 [(i+1) % vertices1.count],
                                    vertices2 [j])
        outside = (ref * test) < 0.0
        j += 1
      }
      intersects = !outside
      i += 1
    }
  }
//---
  if intersects {
    var i = 0
    while intersects && (i < vertices2.count) {
      let ref = CGPoint.product (vertices2 [i],
                                 vertices2 [(i+1) % vertices2.count],
                                 vertices2 [(i+2) % vertices2.count])
      var outside = true
      var j = 0
      while outside && (j < vertices1.count) {
        let test = CGPoint.product (vertices2 [i],
                                    vertices2 [(i+1) % vertices2.count],
                                    vertices1 [j])
        outside = (ref * test) < 0.0
        j += 1
      }
      intersects = !outside
      i += 1
    }
  }
//---
  return intersects
}
```

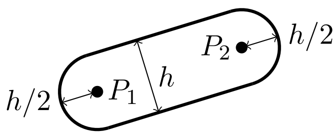
Chapitre 5

Oblong

Un oblong est un rectangle terminé par deux demi-cercles :



Un oblong est défini par deux points et la hauteur de la partie centrale, qui est aussi le diamètre des cercles d'extrémité :




```
struct CanariOblong {
  let p1 : CGPoint
  let p2 : CGPoint
  let height : CGFloat

  init (from p1 : CGPoint, to p2 : CGPoint, height : CGFloat) {
    self.p1 = p1
    self.p2 = p2
    self.height = height
  }
  ...
}
```

5.1 Point dans un oblong

Il suffit de tester successivement si :

— le point est dans le cercle de centre P_1 ;

— le point est dans le cercle de centre P_2 ;

— le point est dans le rectangle formé par la partie centrale ;

```
func contains (point p : CGPoint) -> Bool {
//--- p inside P1 circle
  var inside = CGPoint.distance (self.p1, p) <= (height / 2.0)
//--- p inside P2 circle
  if !inside {
    inside = CGPoint.distance (self.p2, p) <= (height / 2.0)
  }
//--- p inside rectangle
  if !inside {
    let r = CanariRect (from: self.p1, to: self.p2, height: self.height)
    inside = r.contains (point: p)
  }
  return inside
}
```

5.2 Intersection avec un autre rectangle

Il suffit de tester successivement si :

— l'autre rectangle intersecte le cercle de centre P_1 ;

— l'autre rectangle intersecte le cercle de centre P_2 ;

— l'autre rectangle intersecte le rectangle formé par la partie centrale ;

```
func intersects (rect : CanariRect) -> Bool {
//--- rect intersects P1 circle
  let c1 = CanariCircle (center: self.p1, radius: self.height / 2.0)
  var intersects = rect.intersects (circle: c1)
//--- rect intersects P2 circle
  if !intersects {
    let c2 = CanariCircle (center: self.p2, radius: self.height / 2.0)
    intersects = rect.intersects (circle: c2)
  }
//--- rect intersects rectangle
  if !intersects {
    let r = CanariRect (from: self.p1, to: self.p2, height: self.height)
    intersects = rect.intersects (rect: r)
```

```
  }
  return intersects
}
```

5.3 Dessiner

Pour dessiner un oblong, le plus simple est de tracer le segment P_1P_2, avec l'épaisseur `height`. En précisant la terminaison `kCALineCapRound`, les deux demi-disques sont ajoutés au tracé. Si les points sont confondus, le tracé résulte en un disque.

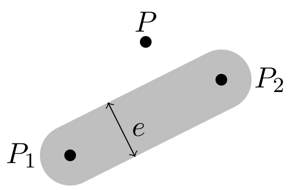
```
func shape () -> CAShapeLayer {
  let mutablePath = CGMutablePath ()
  mutablePath.move (to: self.p1)
  mutablePath.addLine (to: self.p2)
  let newLayer = CAShapeLayer ()
  newLayer.path = mutablePath
  newLayer.lineWidth = self.height
  newLayer.lineCap = kCALineCapRound
  return newLayer
}
```

Pour le tracé effectif, il faut préciser sa couleur, par exemple :

```
  let oblong = CanariOblong (...)
  let shapeLayer = oblong.shape ()
  shapeLayer.strokeColor = NSColor.black.cgColor
```

5.4 Point dans un oblong, autre méthode

Cette technique n'est pas implémentée dans ElCanari.

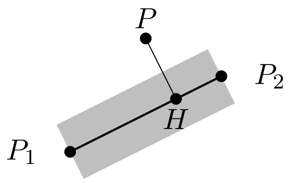


Il y a un cas particulier si les deux points sont confondus : il suffit alors de calculer la distance entre P et le point P_1 (ou P_2), et de la comparer avec $e/2$; ce n'est pas vraiment un cas particulier cas le cas général commence par tester si point P dans le disque autour des extrémités.

Pour le cas général, on effectue trois tests :

— point P dans le disque autour de P_1 : calcul de la distance entre P et P_1, et comparaison avec $e/2$;
— point P dans le disque autour de P_2 : calcul de la distance entre P et P_2, et comparaison avec $e/2$;
— point P dans la partie centrale : c'est le plus compliqué, et est présenté ci-après.

On considère le point $H(H.x, H.y)$, projection de P sur P_1P_2.



Le point P est dans la partie centrale si et seulement si :
— le point H est entre P_1 et P_2 ;
— la distance PH est inférieure à $e/2$.

Nous allons calculer les coordonnées de H, que l'on écrit sous la forme :

$$h.x = \frac{p_1.x + p_2.x}{2} - \lambda\,\frac{p_1.x - p_2.x}{2} \text{ et } h.y = \frac{p_1.y + p_2.y}{2} - \lambda\,\frac{p_1.y - p_2.y}{2}$$

Ceci assure que H est sur la droite P_1P_2 ; si $|\lambda| \leqslant 1$, H est entre P_1 et P_2. Pour calculer λ, on va écrire que $\overrightarrow{PH}$ et $\overrightarrow{P_1P_2}$ sont orthogonaux.

Ainsi :

$$\overrightarrow{HP} \left| \begin{array}{l} \dfrac{p_1.x + p_2.x}{2} - \lambda\,\dfrac{p_1.x - p_2.x}{2} - p.x \\ \dfrac{p_1.y + p_2.y}{2} - \lambda\,\dfrac{p_1.y - p_2.y}{2} - p.y \end{array} \right.$$

$$\overrightarrow{P_2P_1} \left| \begin{array}{l} p_1.x - p_2.x \\ p_1.y - p_2.y \end{array} \right.$$

Pour que ces deux vecteurs soient orthogonaux :

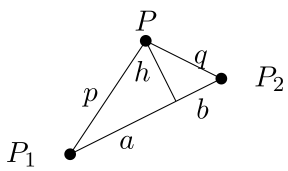
$$(\frac{p_1.x + p_2.x}{2} - \lambda\,\frac{p_1.x - p_2.x}{2} - p.x)\,(p_1.x - p_2.x) = (\frac{p_1.y + p_2.y}{2} - \lambda\,\frac{p_1.y - p_2.y}{2} - p.y)\,(p_1.y - p_2.y)$$

D'où :

$$\lambda = \frac{(p_1.x + p_2.x - 2\,p.x)(p_1.x - p_2.x) + (p_1.y + p_2.y - 2\,p.y)(p_1.y - p_2.y)}{(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2} = \frac{N}{D}$$

```
func segment (from p1 : CGPoint,
              to p2 : CGPoint,
              halfWidth : CGFloat,
              contains p : CGPoint) -> Bool {
//--- Near First point ?
  var contains = p.distanceTo (point: CGPoint (x: p1.x, y: p1.y)) < halfWidth
//--- Near Second point ?
  if !contains {
    contains = p.distanceTo (point: CGPoint (x: p2.x, y: p2.y)) < halfWidth
  }
//--- In segment ?
  if !contains && (( p1.x != p2.x) || (p1.y != p2.y)) {
    let dx = p1.x - p2.x
    let dy = p1.y - p2.y
    let N = (p1.x + p2.x - 2.0 * p.x) * dx + (p1.y + p2.y - 2.0 * p.y) * dy
    let D = dx * dx + dy * dy
    let lambda = N / D
    contains = abs (lambda) < 1.0
    if contains {
      let hx = (p1.x + p2.x) * 0.5 - lambda * dx * 0.5
      let hy = (p1.y + p2.y) * 0.5 - lambda * dy * 0.5
      contains = p.distanceTo (point: CGPoint (x: hx, y: hy)) < halfWidth
    }
  }
  return contains
}
```

5.5 Distance d'un point à une droite, 3e méthode



Problème : connaissant les trois points P_1, P_2 et P, calculer h. Plus présicement, on connait :

— p, distance entre P_1 et P ;

— q, distance entre P_2 et P ;

— $d = a + b$, distance entre P_1 et P_2.

Le système est donc (où les inconnues sont a, b et h, on cherche h) :

$$\begin{aligned} d &= a + b \\ p^2 &= a^2 + h^2 \\ q^2 &= b^2 + h^2 \end{aligned}$$

Il vient :

$$p^2 - q^2 = a^2 - b^2 = (a+b)(a-b) = d(a-b)$$

Donc (ce qui exige que les deux points P_1 et P_2 soient distincts) :

$$a - b = \frac{p^2 - q^2}{d}$$

D'autre part :

$$2h^2 = p^2 + q^2 - a^2 - b^2 = p^2 + q^2 - \frac{1}{2}(a+b)^2 - \frac{1}{2}(a-b)^2$$

Finalement :

$$4h^2 = 2p^2 + 2q^2 - d^2 - (\frac{p^2 - q^2}{d})^2$$

Chapitre 6

Tests d'isolation

Ces algorithmes sont utilisés pour tester l'isolation entre deux les pistes, les pads, les vias...

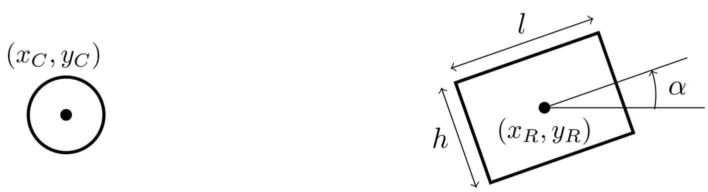
Il s'agit de rectangles d'orientation quelconque, ce qui généralise les rectangles « horizontaux » du chapitre 2 page 8.

6.1 Isolation entre un disque et un rectangle

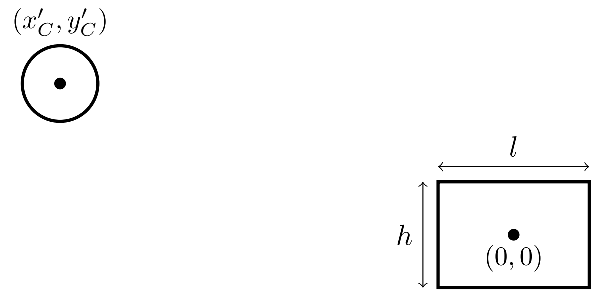
Le rectangle est défini par :
- son centre (x_R, y_R) ;
- son angle α avec l'horizontal ;
- sa largeur l ;
- sa hauteur h.

Le disque est défini par :
- son centre (x_C, y_C) ;
- son rayon r.

La distance d'isolation est iso, c'est-à-dire que la disque entre deux points quelqonques du disque et du rectangle doit être supérieure ou égale à iso.



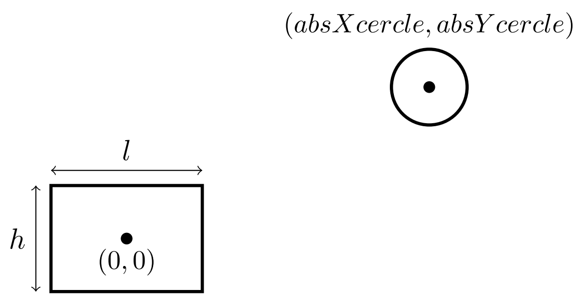
Nous allons effectuer un changement de repère : l'origine du nouveau repère est le centre du rectangle est son orientation celle de la largeur du rectangle (c'est-à-dire une rotation de $-\alpha$). Le centre du cercle a pour coordonnées (x'_C, y'_C) dans ce nouveau repère.



Le calcul de (x'_C, y'_C) (en fait $(Xcercle, Ycercle)$) est le suivant :

```
let rectHalfWidth  = inRectangleSize.width  / 2.0
let rectHalfHeight = inRectangleSize.height / 2.0
let Xrelative = inCircleCenter.x - inRectangleCenter.x
let Yrelative = inCircleCenter.y - inRectangleCenter.y
let cosAngleRectangle = cos (inRectangleAngleInRadians)
let sinAngleRectangle = sin (inRectangleAngleInRadians)
let Xcercle =   Xrelative * cosAngleRectangle + Yrelative * sinAngleRectangle
let Ycercle = - Xrelative * sinAngleRectangle + Yrelative * cosAngleRectangle
```

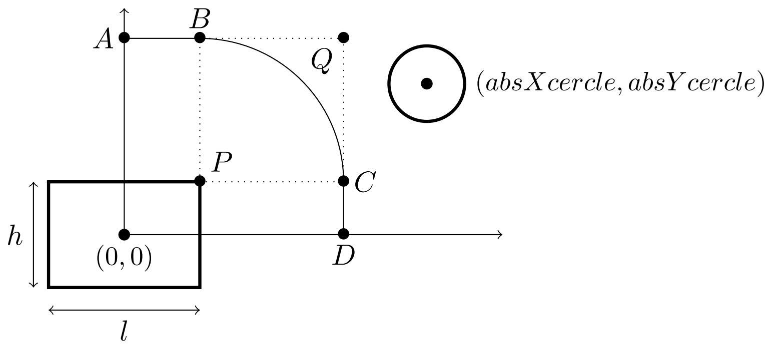
Comme on s'intéresse uniquement à vérifier que la distance cercle / rectangle est suffisante, on prend la valeur absolue de chaque coordonnée du centre du cercle : on se ramène uniquement au premier quadrant.



Ainsi :

```
let absXcercle = fabs (Xcercle)
let absYcercle = fabs (Ycercle)
```

La distance est suffisante si le centre du cercle se trouve *au dessus* ou *à droite* de la ligne $ABCD$.



L'ordonnée des points A et B est $h/2+r+iso$. L'abscisse des points C et D est $l/2+r+iso$. BC est un quart de cercle de centre le sommet supérieur droit du rectangle P, et de rayon $r+iso$.

L'isolation est respectée si :

— le centre du cercle est *suffisament* à droite;

— ou *suffisament* haut;

— ou si il est dans la surface délimitée par BQC.

Le centre du cercle est *suffisament* à droite :

```
var ok = absXcercle >= (l / 2.0 + r + iso)
```

Le centre du cercle est *suffisament* haut :

```
if !ok {
  ok = absYcercle >= (h / 2.0 + r + iso)
}
```

Le centre du cercle est est dans le rectangle $BPCQ$ et la surface délimitée par BQC

```
if !ok && (absXcercle >= (l / 2.0)) && (absYcercle >= (h / 2.0)) {
  let dx = absXcercle - l / 2.0
  let dy = absYcercle - h / 2.0
  let distance = sqrt (dx * dx + dy * dy)
  ok = distance >= (r + iso)
}
```