Aula Teos Geometria Básica - 2020

Primeiro de tudo, vamos representar tudo como vetores (tipo os pontos).

Geralmente sempre vamos representar os vetores partindo da origem, pois as coisas começam a fazer mais sentido.

Tudo o que faremos em geometria será praticamente com operações sobre esses vetores.

Para representar um vetor, usamos um par ordenado (x, y), que representará as coordenadas desses vetores.

Vamos utilizar dois vetores muito importantes:

$$\begin{cases} \hat{i} = (1, 0) \\ \hat{j} = (0, 1) \end{cases}$$

Disso, podemos decompor todos os vetores em termos desses dois vetores que chamaremos de base canônica.

$$(1, 3) = \hat{i} + 3\hat{j}$$

Se nós mudarmos os vetores da base, então fica claro que todos os vetores acabarão sendo mudados também. Chamamos isso de **transformação linear**.

Para transformar um vetor, fazemos a seguinte conta:

$$\begin{bmatrix} \hat{i} & \hat{j} \end{bmatrix} \vec{v}$$

Onde, lembramos, os vetores $\hat{i}$ e $\hat{j}$ são matrizes coluna.

Sendo assim, se mudarmos os vetores da base, vamos ter uma lei para a transformação de qualquer outro vetor do nosso plano.

Rotações 2D

(Sempre rodamos no antihorário)

Como já vimos, para descobrir a transformação, precisamos apenas ver onde os vetores da base vão cair.

Para 90^{o}, temos que a matriz de rodação é

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Pois, $\hat{i}$ cai em $(0, 1)$ e $\hat{j}$ cai em $(-1, 0)$.

Para fazer outras rotações, precisamos manter $\hat{i}$ e $\hat{j}$ perpendiculares e rotacionar $\hat{i}$ em θ. Disso, temos a matriz:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

que rotaciona um vetor ao redor da origem em θ graus.

```
// o tipo da coordenada que estamos usando
using coord = long double;
const long double PI = acos(-1L); // Pi
#define EPS 1E-8

Função para saber se é 0 (quando retorna 0, é 0)
int signal(coord x) { return (x > EPS) - (x < -EPS); }

// uma estrutura para os pontos (ou vetores)
// vamos definir algumas operações sobre eles
struct point {
    coord x, y;

    // Operações básicas
    point operator+(const &point p) { return {x + p.x, y + p.y}; }
    point operator-(const &point p) { return {x - p.x, y - p.y}; }
    point operator*(coord s) { return {x * s, y * s}; }
    point operator/(coord s) { return {x / s, y / s}; }

    // Norma (tamanho ao quadrado)
    coord norm() { return x*x + y*y; }

    // Tamanho do vetor (não é inteiro)
    coord len() { return sqrt(x*x + y*y); }

    // Rotações 2D
    point rotate90() { return {-y, x}; }
    point rotate(long double ang) {
        return { cos(ang)*x - sin(ang)*y, sin(ang)*x + cos(ang)*y };
    }

    void print() { printf("%lld %lld\n", x, y); }
}
```

Produto interno (ou escalar)

O produto intervo pode ser pensado como a fórmula:

$$v \cdot u = \|v\| \|u\| \cos \theta$$

Vejamos o significa geométrico desse produto.

Geometricamente, $v \cdot u$ é a projeção de u em v, dai vem o $\|u\| \cos \theta$. Ou seja, ele representa o tamanoh de v vezes o tamanho da projeção de u em v.

Disso, nós vemos que se eles são perpendiculares, então o produto escalar é 0. Se eles apontam "mais ou menos na mesma direção", temos que o produto é positivo e caso eles apontem "mais ou menos em direções contrárias", então temos que esse produto é negativo.

Outra coisa importante, é que apesar de geometricamente a seguinte afirmação parecer falsa, ela é verdadeira:

$$u \cdot v = v \cdot u$$

Nós poderíamos utilizar a fórmula mais geral, mas nós sabemos que o produto escalar também possui essa fórmula:

$$v \cdot u = v_x u_x + v_y u_y$$

```
coord pot(const &point p) { return x * p.x + y * p.y; }
```

Se quisermos agora o vetor da projeção em u em v, temos que

$$\frac{u \cdot v}{\|v\|} = \|u'\|$$

E depois temos que

$$\frac{v}{\|v\|} \|u'\| = u'$$

```
point project(point v) { return v * (dot(v) / v.norm()); }
```

Retas

Primeiro vamos pensar em como representamos uma reta.

Podemos achar a, b, c daquela forma tradicional:

$$ax + by + c = 0$$

Ou podemos pensar no que já estávamos trabalhando.

Suponha que p é um vetor paralelo à reta. Então, qualquer ponto da reta, ao projetarmos no vetor, será igual.

Ou seja,

$$p \cdot v = c$$

onde v são todos os pontos da reta e c é uma constante. E essa equação só vale para os pontos que estão na reta.

Temos que observar que dado um vetor a constante é única e vice-versa, mas podemos definir a mesma reta com vários possíveis vetores ou constantes.

```
struct line {
    // Temos aqui a representação da reta (por um vetor e uma constante)
    point p; coord c;
    line (point p, coord c) p(p), c(c) {};

    // construtor com dois pontos da reta
    line(point a, point b): p(rotate90(b-a)), c(p.dot(a)) {};

    bool contains(point p) { return signal(p.dot(v) - c) == 0; }
    // Retorna uma reta paralela que contenha o ponto u
    line parallel(point u) { return line(p, p.dot(v)); }
    // Retorna uma reta perpendicular qualquer
    line perpendicular() { return line(p.rotate90(), c); }
}
```

Produto vetorial

O produto vetorial tem representação gráfica muito mais clara do que o produto interno. Ele é a área formada pelo paralelogramo criado pelos vetores u e v. Se dividirmos por dois, temos a área do triângulo formado pelos dois vetores.

A fórmula mais comum é:

$$v \times u = \|v\| \|u\| \sin \theta$$

Além disso, vemos que $\sin \theta$ pode ser negativo. Se v estiver à esquerda de u, temos que o produto é negativo, caso contrário, é positivo. E se eles forem paralelos, não tem área e, logo, é 0.

Dito isso, lembramos que:

$$v \times u = -u \times v$$

Lembramos que existe uma fórmula melhor para calcular o produto cruzado utilizando o determinante dessa matriz:

$$\begin{bmatrix} u_x & v_x \\ u_y & v_y \end{bmatrix} = u_x v_y - u_y v_x$$

Pois lembramos que o determinante de uma matriz é basicamente a área do paralelogramo formado pelos vetores quando transformamos $\hat{i}$ em u e $\hat{j}$ em v.

```
coord cross(point p) { return x * p.y - y * p.x; }
```

Utilizando isso, nós podemos calcular a área de qualquer polígono utilizando um algoritmo. Tomamos u e v dois pontos adjacente desse polígono. Calculamos $u \times v$, guardamos isso numa variável de acumulação e depois movemos u e v para os próximos dois pontos do polígono em algum sentido (horário ou antiohorário).
Repetimos o processo até retornamos aos dois pontos originais.

Ao final, dividimos a variável de acumulação por 2 e tiramos o módulo. Isso dá a área.

Intersecções

Reta-reta

Suponha que tenhamos duas retas r_1 e r_2 e tenhamos seus respectivos vetores e constante p_1, c_1, p_2, c_2 e queremos calcular sua intersecção (achar o ponto v que está nas duas retas ao mesmo tempo). (Lembre-se que temos os casos em que as retas são paralelas)

Disso, precisamos que:

$$\begin{cases} p_1 \cdot v = c_1 \\ p_2 \cdot v = c_2 \end{cases}$$

Logo, queremos resolver um sistema linear:

$$\begin{bmatrix} p_{1x} & p_{1y} \\ p_{2x} & p_{2y} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

Vemos que para saber se esse sistema tem solução, basta que o determinante da minha matriz 2x2 não é 0. E isso é exatamente o produto vetorial entre p_1 e p_2 (verificar se elas não são paralelas).

Sabemos que para achar x e y, podemos fazer a regra de Cramer (colocar a matriz de coeficiente na coluna do x, calcular a determinante disso sobre a determinante da matrix 2x2 e isso dá o x, análogo para y).

```
point intersection(line l) {
    coord D = p.cross(l.p);

    assert(signal(D) != 0); // verfica se não são retas paralelas

    // Cramer
    return { point(c, p.y).cross(point(l.c, l.p.y)) / D,
             point(p.x, c).cross(point(l.p.x, l.c)) / D, }
}
```

Segmento-ponto

Primeiro vemos se o ponto está na reta que contém o segmento e depois vemos se as coordenada do ponto estão "presas" dentro do intervalo do segmento.

```
bool contains(point p) {
    return signal((p-a).cross(b-a)) == 0
    && signal((p-a).dot(b-a)) >= 0
    && signal((p-b).dot(a-b)) >= 0;
}
```

Círculo-Reta

Primeiro checamos se a distância do centro do círculo até a reta é menor ou igual que o raio. (no caso de ser igual, vamos achar dois pointos que são o mesmo)

Seja h a distância entre a reta e o círculo, r o raio e θ o ângulo que a semi-reta que parte do centro do círculo e chega na reta (formando um ângulo de $\pi/2$). Então, sabemos que

$$\cos(\theta) = \frac{h}{r}$$

Disso,

$$\theta = \arccos\left(\frac{h}{r}\right)$$

Logo, sabemos que se acharmos a direção desse vetor que sai do centro e toca a reta, mas deixarmos ele com o tamanho r, podemos rotaciona-lo em θ e $-\theta$ e os pontos que desejamos são exatamente o centro do raio somado com esses vetores rotacionados.

Precisamos então resolver esse problema de como achar um vetor que seria o vetor "distância-ponto-reta".

Vetor da distância de ponto a reta

Seja r uma reta dada por p e c, u o ponto dado pela interseção da reta paralela a p que contém $(0, 0)$ e u' um ponto de distância d até u que esteja nessa mesma reta.

Suponha que queremos saber a distância de um ponto v até a reta. Então vamos pensar nesse ponto u que esteja numa reta

Caderno

```cpp
// o tipo da coordenada que estamos usando
using coord = long double;
const long double PI = acos(-1L); // Pi
#define EPS 1E-8

// uma estrutura para os pontos (ou vetores)
// vamos definir algumas operações sobre eles
struct point {
    coord x, y;

    point(coord x, coord y) : x(x), y(y) {};

    // Operações básicas
    point operator+(const &point p) { return {x + p.x, y + p.y}; }
    point operator-(const &point p) { return {x - p.x, y - p.y}; }
    point operator*(coord s) { return {x * s, y * s}; }
    point operator/(coord s) { return {x / s, y / s}; }

    // Norma (tamanho ao quadrado)
    coord norm() { return x*x + y*y; }

    // Tamanho do vetor (não é inteiro)
    coord len() { return sqrt(x*x + y*y); }

    // Rotações 2D
    point rotate90() { return {-y, x}; }
    point rotate(long double ang) {
        return { cos(ang)*x - sin(ang)*y, sin(ang)*x + cos(ang)*y };
    }

    // Produto interno
    coord pot(const &point p) { return x * p.x + y * p.y; }
    // Projeção desse vetor em v
    point project(point v) { return v * (dot(v) / v.norm()); }

    // Produto cruzado
    coord cross(point p) { return x * p.y - y * p.x; }

    void print() { printf("%lld %lld\n", x, y); }
};

struct line {
    // Temos aqui a representação da reta (por um vetor e uma constante)
    point p; coord c;
    line (point p, coord c) p(p), c(c) {};

    // construtor com dois pontos da reta
    line(point a, point b): p(rotate90(b-a)), c(p.dot(a)) {};

    bool contains(point p) { return signal(p.dot(v) - c) == 0; }
    // Retorna uma reta paralela que contenha o ponto u
```

```
    line parallel(point u) { return line(p, p.dot(v)); }
    // Retorna uma reta perpendicular qualquer
    line perpendicular() { return line(p.rotate90(), c); }

    point intersection(line l) {
        coord D = p.cross(l.p);

        assert(signal(D) != 0); // verfica se não são retas paralelas

        // Cramer
        return { point(c, p.y).cross(point(l.c, l.p.y)) / D,
                point(p.x, c).cross(point(l.p.x, l.c)) / D, }
    }
    point intersection(segment s) {
        line l(s.a, s.b);
        point p = intersection(l);
        assert(s.contains(p));
        return p;
    }
    coord distance_sq(point v) { return ((p.dot(v)-c)*(p.dot(v)-c))/p.norm(); }
};

struct segment {
    point a; point b;
    segment(point a, point b) : a(a), b(b) {};
    bool contains(point p) {
        return signal((p-a).cross(b-a)) == 0
        && signal((p-a).dot(b-a)) >= 0
        && signal((p-b).dot(a-b)) >= 0;
    }
};

struct circle {
    point c; coord r;
    pair<point, point> intersection(line l) {
        coord h2 = l.distance_sq(c);
        assert(signal(h2-r*r) <= 0);

        point dir = l.p - (l.c-c.dot(l.p))/l.p.norm();
        dir = dir * (r / dir.len());
        coord h = sqrt(l.distance_sq(c));
        long double theta = acos(h / r);
        return { c + dir.rotate(theta), c+dir.ratate(-theta); }
    }
};
```