Operaciones con Vectores

Rodrigo García Herrera

2016-04-27

Abstract

Reflexión didáctica acerca de algunos conceptos de Programación Orientada a Objetos

Contents

Operaciones con Vectores

Reflexión didáctica acerca de algunos conceptos de Programación Orientada a Objetos.

El ejercicio consiste de la recapitulación de algunas técnicas y mejores prácticas de programación. El programa bajo estudio hace operaciones con vectores, las diferentes implementaciones permiten contrastar la forma en que diferentes lenguajes y usos de lenguajes dan lugar a formas de expresar la solución a un problema.

Se provee también un PDF generado con pandoc a partir de este archivo.

Operaciones de vectores

Las explicaciones de las operaciones están en otros lados, el tema es cómo convertirlas en software.

No implementamos todas las operaciones, unas cuántas nomás de ejemplo. Para tenerlas claras a continuación las escribimos en Python que es bastante legible para cualquiera.

Representación polar, como magnitud y ángulo a componentes

Llegada la hora de escribir el programa en Python seguramente habrá una clase Vector, tendrá un constructor polimórfico que reciba como argumentos componentes x, y o una magnitud y un ángulo.

```
    v = vector(magnitud=10, angulo=0)
    u = polar_a_componentes( v )

    def polar_a_componentes( v ):
        return vector(x = v.magnitud * cos( v.angulo ),
                      y = v.magnitud * sin( v.angulo ) )
```

Vector por un número

Acá se muestra el constructor por componentes.

```
    a = vector(x=3, y=4)
    n = 3

    def vector_x_num( vector, num ):
        return vector( vector.x * num, vector.y * num )
```

Suma de Vectores

```
    vector_suma( a, b ):
        return vector( a.x + b.x,
                       a.y + b.y )
```

Producto escalar, producto punto

```
    a = vector(x=3, y=4)
    b = vector(x=3, y=4)

    def producto_escalar( a, b ):
        return (a.x * b.x) + (a.y * b.y)
```

Módulo del Producto cruz

```
    a = vector(x=3, y=4)
    b = vector(x=5, y=6)

    def modulo_producto_cruz( a, b ):
        return sqrt(((a.x*.y)-(a.y*b.x))**2)
```

Código spaghetti

Así llamado por que seguir el hilo de ejecución es parecido a seguir una hebra de pasta en un plato de espaghetti: entra por acá, sale por allá, regresa más arriba, etc.

En Basic esta es la forma natural de programar, usando la expresión GOTO para controlar el flujo de un programa.

El código fuente está en el repositorio original.

```
10 REM ================================================
20 REM Programa que opera Vectores
30 REM ================================================
40 CLS

41 REM En aquellos días este tipo de interfaz era buena,
42 REM quizá por lo mal shell que era command.com
43 REM
44 REM A continuación un bucle que pide valores para armar
45 REM dos vectores, hace varias operaciones con ellos e
46 REM imprime los resultados.

50 PRINT "Programa que opera Vectores"
60 PRINT
70 INPUT "vector U, componente x"; U_X
80 INPUT "vector U, componente y"; U_Y
90 PRINT
100 INPUT "vector V, magnitud"; V_magnitud
110 INPUT "vector V, ángulo"; V_angulo
120 PRINT
130 INPUT "escalar"; E
140 PRINT
150 PRINT
160 PRINT


170 REM Esta técnica es precursora del paso de argumentos
171 a una subrutina. NX y NY son variables transitorias que
172 la subrutina en 666 usará de argumentos.
180 NX = U_X : NY = U_Y
190 PRINT "Vector U"
200 GOSUB 666
210 PRINT

220 REM convertir de representación polar a por-componentes
230 NM = V_magnitud : NA = V_angulo : GOSUB 888
231 De manera análoga la función devuelve sus valores en
232 las variables NX y NY.
240 V_X = NX : V_Y = NY
250 PRINT "Vector V"
260 GOSUB 666
```

```
270 PRINT

280 REM vector U por escalar E
290 NX = U_X : NY = U_Y : NE = E : GOSUB 777
300 PRINT "Vector U por escalar E"
310 GOSUB 666
320 PRINT
330 REM vector U por escalar E
340 NX = V_X : NY = V_Y : NE = E : GOSUB 777
350 PRINT "Vector V por escalar E"
360 GOSUB 666
370 PRINT


380 REM Producto Punto
390 MX = U_X : MY = U_Y
400 NX = V_X : NY = V_Y
410 GOSUB 1111
420 PRINT "Producto punto U·V="; E
430 PRINT


440 REM Suma de vectores
450 MX = U_X : MY = U_Y
460 NX = V_X : NY = V_Y
470 PRINT "Suma de vectores U+V"
480 GOSUB 999
490 PRINT


500 REM Producto cruz
510 MX = U_X : MY = U_Y
520 NX = V_X : NY = V_Y
530 GOSUB 2222
540 PRINT "modulo de producto cruz |UxV|="; E
550 PRINT



560 INPUT "Desea continuar (S/N)"; C$
570 IF C$ = "S" OR C$ = "s" THEN GOTO 40
580 END
```

```
666 REM ========================================
676 REM Impresion de Vectores por Componentes
686 REM ========================================
696 X = NX : Y = NY
706 PRINT "("; X; "x, "; Y; "y)"
716 RETURN




777 REM ========================================
787 REM Vector por un escalar
797 REM ========================================
807 vX = NX : vY = NY : nE = NE
817 NX = vX * nE
827 NY = vY * nE
837 RETURN




888 REM ========================================================
898 REM Conversión de representación polar a por-componentes
908 REM ========================================================
918 M=NM : A = NA
928 NX = M * cos( A )
938 NY = M * sin( A )
948 RETURN




999 REM ====================
1009 REM Suma de vectores
1019 REM ====================
1029 A_X = NX : A_Y = NY
1039 B_X = MX : B_Y = MY
1049 X = A_X + B_X
1059 Y = A_Y + B_Y
1069 NX = X : NY = Y
1079 GOSUB 666
1089 RETURN
```

```
1111 REM ==================
1121 REM  Producto punto
1131 REM ==================
1141 A_X = NX : A_Y = NY
1151 B_X = MX : B_Y = MY
1152 REM Esta subrutina devuelve E
1161 E = (A_X * B_X) + (A_Y * B_Y)
1171 RETURN



2222 REM =================
2232 REM Producto cruz
2242 REM =================
2252 A_X = NX : A_Y = NY
2262 B_X = MX : B_Y = MY
2263 REM Esta subrutina devuelve E
2272 E = SQR( ((A_X*B_Y)-(A_Y*B_X))^2 )
2282 RETURN
```

Programación Procedural

Este ejemplo está escrito en SmallBasic. Basic evolucionó: ahora ya no es necesario numerar las líneas de código y hay sintaxis que sirve para denotar subrutinas de manera explícita, con genuinos argumentos y nada de variables transitorias.

El código fuente está en el repositorio original.

```
const X = 1
const Y = 2
REM ================================================
REM Programa que opera Vectores
REM ================================================
CLS
DIM U(2)
DIM V(2)
DIM A(2)
DIM B(2)
DIM VECTOR(2)
REPEAT
  PRINT "Programa que opera vectores"
  PRINT
```

```
  INPUT "Vector U, componente x"; UX
  INPUT "Vector U, componente y"; UY
  U = VectorCrear(UX, UY)
  PRINT
  INPUT "Vector V, magnitud"; magnitud
  INPUT "Vector V, angulo"; angulo
  V = VectorCrearPolar(magnitud, angulo)
  PRINT
  INPUT "Escalar"; E
  PRINT

  PRINT "Vector U"
  VectorImprimir U

  PRINT "Vector V"
  VectorImprimir V

  PRINT "Escalar E"
  PRINT E

  PRINT "suma de vectores U+V"
  A = VectorSumar(U, V)
  VectorImprimir A

  PRINT "vector por escalar U*E"
  A = VectorPorEscalar(U, E)
  VectorImprimir A

  PRINT "vector por escalar V*E"
  A = VectorPorEscalar(V, E)
  VectorImprimir A

  PRINT "producto punto U.V"
  E = ProductoPunto(U, V)
  PRINT E

  PRINT "modulo de producto cruz |UxV|"
  E = ModuloDeProductoCruz(U, V)
  PRINT E
  PRINT
  PRINT

  INPUT "Desea continuar (S/N)"; C$
UNTIL C$ <> "S" AND C$ <> "s"
END
```

```
REM ========================================
REM Crear Vectores por componentes
REM ========================================
FUNC VectorCrear(XX, YY)
  DIM vtmp(2)
  vtmp(X) = XX
  vtmp(Y) = YY
  VectorCrear = vtmp
END FUNC


REM ========================================
REM Crear Vectores Polar
REM ========================================
FUNC VectorCrearPolar(M, ANG)
  VectorCrearPolar = VectorCrear(M * COS(ANG), M * SIN(ANG))
END FUNC



REM ========================================
REM Suma de Vectores
REM ========================================
FUNC VectorSumar(A, B)
  VectorSumar = VectorCrear(A(X)+B(X), A(Y)+B(Y))
END FUNC

REM ==========================
REM Vector por un escalar
REM ==========================
FUNC VectorPorEscalar(VECTOR, E)
    VectorPorEscalar = VectorCrear( VECTOR(X)*E, VECTOR(Y) * E )
END FUNC

REM ======================================================
REM Producto escalar, producto punto
REM ======================================================
FUNC ProductoPunto(A, B)
    ProductoPunto = (A(X) * B(X)) + (A(Y) * B(Y))
END FUNC

REM ======================================================
REM Módulo del producto cruz
REM ======================================================
FUNC ModuloDeProductoCruz(A, B)
```

```
    ModuloDeProductoCruz = SQR(((A(X)*B(Y))-(A(Y)*B(X)))^2)
END FUNC


REM ========================================
REM Impresion de Vector
REM ========================================
SUB VectorImprimir(VECTOR)
  PRINT "("; VECTOR(X); "x, "; VECTOR(Y); "y)"
END SUB
```

Tipos de datos

Algunos lenguajes soportan la especificación de los tipos de datos a usarse.

Para el programador mejora la legibilidad y facilita la depuración, para el compilador la información de tipos de datos permite optimizar la creación de código de máquina.

Además de tipos de dato nativos, como “integer”, “real” o “string”, Algunos lenguajes permiten la creación de tipos de dato definidos por el usuario. Se construyen a partir de los tipos nativos del lenguaje pero permiten al programador expresar relaciones nombradas y fácilmente asequibles entre datos más elementales.

Versión en C

Esta versión en C muestra el uso de tipos de datos a través del uso de “struct”.

Acá el código fuente.

Se compila con gcc así:

```
$ gcc -lm tipos_de_datos.c
```

```
#include <stdio.h>
#include <math.h>

typedef struct _TVector_ {
  float X;
  float Y;
```

```
} TVector;

//Prototipos de las funciones
TVector VectorCrear(float, float);
TVector VectorCrear_MagnitudAngulo(float, float);
TVector Vector_x_escalar(TVector, float);
TVector VectorSumar(TVector, TVector);
float VectorProductoPunto(TVector, TVector);
float VectorModulo_ProductoCruz(TVector, TVector);
void VectorComoCadena(TVector);

int main() {
  TVector U, V;
  float E, x, y, m, a;

  printf("Vector U componente X: ");
  scanf("%f", &x);
  printf("Vector U componente Y: ");
  scanf("%f", &y);
  U = VectorCrear(x, y);
  printf("Vector V magnitud: ");
  scanf("%f", &m);
  printf("Vector V angulo: ");
  scanf("%f", &a);
  V = VectorCrear_MagnitudAngulo(m, a);
  printf("Escalar E: ");
  scanf("%f", &E);


  printf("\n\n\nVector U=");
  VectorComoCadena(U);
  printf("Vector V=");
  VectorComoCadena(V);
  printf("E=%2.2f\n\n\n", E);

  printf("Vector U x E=");
  VectorComoCadena(Vector_x_escalar(U, E));
  printf("Vector V x E=");
  VectorComoCadena(Vector_x_escalar(V, E));
  printf("U+V=");
  VectorComoCadena(VectorSumar(U,V));
  printf("U.V=%2.2f\n", VectorProductoPunto(U, V));
  printf("|UxV|=%2.2f\n", VectorModulo_ProductoCruz(U, V));

  return 0;
```

```
}

TVector VectorCrear(float X, float Y) {
  TVector temp;
  temp.X = X;
  temp.Y = Y;
  return temp;
}

TVector VectorCrear_MagnitudAngulo( float m, float a) {
  return VectorCrear( m * cos(a), m * sin(a) );
}


// Vector por un escalar
TVector Vector_x_escalar(TVector vector, float escalar) {
   return VectorCrear( vector.X * escalar, vector.Y * escalar);
}

// Suma de Vectores
TVector VectorSumar(TVector a, TVector b) {
  return VectorCrear( a.X + b.X, a.Y + b.Y );
}


// Producto escalar o punto
float VectorProductoPunto(TVector a, TVector b) {
   return (a.X * b.X) + (a.Y * b.Y);
}

// Módulo del Producto cruz
float VectorModulo_ProductoCruz(TVector a, TVector b) {
  return sqrt(pow( (a.X * b.Y)-(a.Y*b.X), 2));
}


void VectorComoCadena( TVector v) {
  printf("(%2.2fx, %2.2fy)\n", v.X, v.Y);
}
```

Versión en Pascal

Pascal tiene soporte para tipos de datos definidos por el usuario a través del tipo “record”.

Este código fuente compila en Free Pascal, con el siguiente comando:

```
$ fpc tipos_de_datos.pas
```

```
program vectores_td;

type
  {Anteponer a los tipos una "T" es una buena costumbre.
   Se conoce como "notación húngara"}
  TVector = record
    X : real;
    Y : real;
  end;

var
  U, V     : TVector;
   x,y,m,a : real;
  E        : real;


function VectorCrear(x, y : real): TVector;
var
  temp : TVector;
begin
  temp.X := x;
  temp.Y := y;
  VectorCrear := temp;
end;


{Crear un vector con magnitud y angulo}
function VectorCrear_MagnitudAngulo(m, a : real): TVector;
begin
   VectorCrear_MagnitudAngulo := VectorCrear( m * cos( a ), m * sin( a ) );
end;


{Vector por un escalar}
function Vector_x_escalar(vector : TVector; escalar : real): TVector;
begin
   Vector_x_escalar := VectorCrear( vector.X * escalar, vector.Y * escalar);
end;
```

```
{Suma de Vectores}
function VectorSumar(a, b : TVector): TVector;
begin
   VectorSumar := VectorCrear( a.X + b.X, a.Y + b.Y );
end;


{Producto escalar o punto}
function VectorProductoPunto(a, b :  TVector): real;
begin
   VectorProductoPunto := (a.X * b.X) + (a.Y * b.Y);
end;

{Módulo del Producto cruz}
function VectorModulo_ProductoCruz(a, b : TVector ): real;
begin
   VectorModulo_ProductoCruz:= sqrt(sqr( (a.X * b.Y)-(a.Y*b.X)))
end;

{imprimir un vector}
procedure VectorImprimir(V : TVector);
begin
   writeln('(',V.X, 'x, ', V.Y, ')');
end;



begin
   write('vector U componente X= ');
   readln(x);
   write('vector U componente Y= ');
   readln(y);
   U := VectorCrear(x,y);

   write('vector V magnitud= ');
   readln(m);
   write('vector V angulo= ');
   readln(a);
   V:= VectorCrear_MagnitudAngulo(m, a);

   write('Escalar= ');
   readln(E);

   write('U= ');
   VectorImprimir(U);
```

```
   write('V= ');
   VectorImprimir(V);

   write('UxE= ');
   VectorImprimir(Vector_x_escalar(U, E));
   write('VxE= ');
   VectorImprimir(Vector_x_escalar(V, E));

   write('U+V= ');
   VectorImprimir(VectorSumar(U, V));

   write('U.V= ');
   writeln(VectorProductoPunto(U, V));
   write('UxV= ');
   writeln(VectorModulo_ProductoCruz( U, V));

end.
```

Orientación a Objetos

La transición de tipos de datos a orientación a objetos es notoria pero suave. A la distancia pareciera que el único cambio consiste de acuerpar funciones en la misma estructura.

Los tipos de datos de la sección anterior se convierten en clases, lo que simplifica la definición de las funciones a través de la autorreferencia con el identificador "self", se nota en la omisión de un argumento que ya no es necesario.

A continuación se muestra el mismo programa pero evolucionado.

Implementación en Pascal

Free Pascal tiene soporte para sintaxis orientada a objetos.

Esta versión muestra la técnica para crear una "Unit" que es la forma en que Pascal empaqueta y distribuye bibliotecas.

El código fuente está dividido en dos archivos: uvectores_oo.pas donde se definen las clases y vectores_oo.pas donde se instancían y se usan.

Compila en Free Pascal, con el siguiente comando:

```
$ fpc -Mobjfpc vectores_oo.pas
```

```
unit uvectores_oo;


interface

uses
sysutils;

type
   TVector = class
   private
   X : real;
   Y : real;

public

  constructor Crear(a,b: real);
  constructor Crear_MagnitudAngulo(m, a : real);
  function x_escalar( escalar : real): TVector;
  function Suma(b : TVector): TVector;
  function ProductoPunto( b :  TVector): real;
  function Modulo_ProductoCruz( b : TVector ): real;
  function ComoCadena: string;
  destructor Free;
end;



implementation

{Crear un vector con sus componentes x y y}
constructor TVector.Crear(a,b : real);
begin
   inherited Create;
   self.X := a;
   self.Y := b;
end;


{Crear un vector con magnitud y angulo}
constructor TVector.Crear_MagnitudAngulo(m, a : real);
begin
   self.X := m * cos( a );
   self.Y := m * sin( a );
end;
```

```
destructor TVector.Free;
begin
   inherited Destroy;
end;




{Vector por un escalar}
function TVector.x_escalar(escalar : real): TVector;
begin
   Result := TVector.Crear( self.X * escalar, self.Y * escalar);
end;

{Suma de Vectores}
function TVector.Suma(b : TVector): TVector;
begin
   Result := TVector.Crear( self.X + b.X, self.Y + b.Y );
end;


{Producto escalar o punto}
function TVector.ProductoPunto(b :  TVector): real;
begin
   Result := (self.X * b.X) + (self.Y * b.Y);
end;

{Módulo del Producto cruz}
function TVector.Modulo_ProductoCruz(b : TVector ): real;
begin
   Result:= sqrt(sqr( (self.X * b.Y)-(self.Y*b.X)))
end;

{autorrepresentacion imprimible}
function TVector.ComoCadena: string;
begin
   Result := '('+ FloatToStr(self.X) + 'x, ' + FloatToStr(self.Y) + 'y)';
end;

end.
```

```
program vectoresOO;
uses
```

```
  uvectores_oo,
  sysutils;

var
  U, V     : TVector;
  x,y,m,a  : real;
  E        : real;

begin

   write('vector U componente X= ');
   readln(x);
   write('vector U componente Y= ');
   readln(y);
   U := TVector.Crear(x,y);

   write('vector V magnitud= ');
   readln(m);
   write('vector V angulo= ');
   readln(a);
   V:= TVector.Crear_MagnitudAngulo(m, a);

   write('Escalar= ');
   readln(E);

   writeln('U= ', U.ComoCadena);
   writeln('V= ', V.ComoCadena);


   writeln('UxE= ', U.x_escalar(E).ComoCadena);
   writeln('VxE= ', V.x_escalar(E).ComoCadena);

   writeln('U+V= ', U.Suma(V).ComoCadena);

   writeln('U.V= ', U.ProductoPunto(V));
   writeln('UxV= ', U.Modulo_ProductoCruz(V));


   U.Free;
   V.Free;

end.
```

Implementación en C++

El lenguaje C++ agrega programación orientada a objetos al lenguaje C.

En esta versión también se muestra el uso de un archivo de encabezado (.h) que es una forma de crear bibliotecas cuya funcionalidad puede incluirse en otros programas.

El archivo vectores_oo.h declara la clase que se instancía en vectores_oo.cpp.

Se muestran dos formas de crear una función miembro, también conocida como método: una al declarar la clase, cual es el caso del método x_escalar, otra creando un prototipo al declarar la clase y detallándolo posteriormente, como en el caso del método ProductoPunto.

```
#include <math.h>

class TVector {
 private:
  float X;
  float Y;

 public:

  TVector() {
    X = 0;
    Y = 0;
  }

  TVector(float x, float y) {
    X = x;
    Y = y;
  }

  TVector MagnitudAngulo( float m, float a) {
    X = m * cos(a);
    Y = m * sin(a);
  }


  // Vector por un escalar
  TVector x_escalar(float escalar) {
    return TVector( X * escalar, Y * escalar);
  }


  // Suma de Vectores
```

```
  TVector Sumar(TVector);

  // Producto escalar o punto
  float ProductoPunto(TVector);

  // Módulo del Producto cruz
  float Modulo_ProductoCruz(TVector);

  // representacion imprimible
  char* ComoCadena() {
    printf("(%2.2fx, %2.2fy)\n", X, Y);
  }


};

// Suma de Vectores
TVector TVector::Sumar(TVector b) {
  return TVector( X + b.X, Y + b.Y );
}


// Producto escalar o punto
float TVector::ProductoPunto(TVector b) {
   return (X * b.X) + (Y * b.Y);
}

// Módulo del Producto cruz
float TVector::Modulo_ProductoCruz(TVector b) {
  return sqrt(pow( (X * b.Y)-(Y*b.X), 2));
}
```

```
#include <stdio.h>
#include "vectores_oo.h"

int main() {
  float x, y, m, a, E;

  printf("Vector U componente X: ");
  scanf("%f", &x);
  printf("Vector U componente Y: ");
  scanf("%f", &y);
  TVector U = TVector(x, y);
```

```
  printf("Vector V magnitud: ");
  scanf("%f", &m);
  printf("Vector V angulo: ");
  scanf("%f", &a);

  TVector V = TVector();
  V.MagnitudAngulo(m, a);


  printf("Escalar E: ");
  scanf("%f", &E);


  printf("\n\n\nVector U=");
  U.ComoCadena();
  printf("Vector V=");
  V.ComoCadena();
  printf("E=%2.2f\n\n\n", E);

  printf("Vector U x E=");
  U.x_escalar(E).ComoCadena();
  printf("Vector V x E=");
  V.x_escalar(E).ComoCadena();
  printf("U+V=");
  U.Sumar(V).ComoCadena();
  printf("U.V=%2.2f\n", V.ProductoPunto(U));
  printf("|UxV|=%2.2f\n", V.Modulo_ProductoCruz(U));

  return 0;
}
```

Implementación en Python

Python es un lenguaje de programación con excelente soporte para programación orientada objetos: la implementación es notoriamente breve y clara.

Nótese cómo el método ___repr___ de las clases que heredan de "object" es una forma integral al lenguaje del ComoCadena que hemos visto en otras implementaciones.

Además la capacidad de declarar argumentos nombrados y valores default en la definición de una función hacen innecesario el polimorfismo de constructores.

En Python basta con que dos archivos con extensión ".py" estén en el mismo directorio para que uno pueda incluir de otro.

El archivo vectores_oo.py hace de biblioteca y en él se declara la clase que se usa en vectores.py.

```
# coding: utf-8
from math import sin, cos, sqrt

class Vector(object):

    # el constructor puede usar x,y o magnitud y ángulo
    def __init__(self, x=0, y=0, m=None, a=None):

        if m != None and a != None:
            self.X = m * cos(a)
            self.Y = m * sin(a)
        else:
            self.X = x
            self.Y = y

    # vector por escalar
    def x_escalar(self, escalar):
        return Vector(self.X * escalar, self.Y * escalar)

    # Suma de Vectores
    def Suma( self, v ):
        return Vector( self.X + v.X, self.Y + v.Y )

    # Producto escalar o punto
    def ProductoPunto( self, v ):
        return (self.X * v.X) + (self.Y * v.Y)

    # Módulo del Producto cruz
    def Modulo_ProductoCruz( self, v ):
        return sqrt(pow( (self.X * v.Y)-(self.Y * v.X), 2))

    # representación imprimible
    def __repr__(self):
        return "(%2.2fx, %2.2fy)" % (self.X, self.Y)
```

```
from vectores_oo import Vector

x = input('vector U componente X= ')
y = input('vector U componente X= ')
U = Vector(x,y)

m = input('vector V magnitud= ')
```

```
a = input('vector V angulo= ')
V = Vector(m=m, a=a)

E = input('Escalar= ')

print "U=%s" % U
print "V=%s" % V

print 'UxE=%s' % U.x_escalar(E)
print 'VxE=%s' % V.x_escalar(E)

print 'U+V=%s' % U.Suma(V)
print 'U.V=%s' % U.ProductoPunto(V)
print '|UxV|=%s' % U.Modulo_ProductoCruz(V)
```