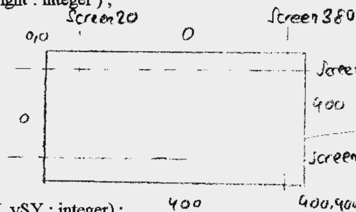
Graphs.h

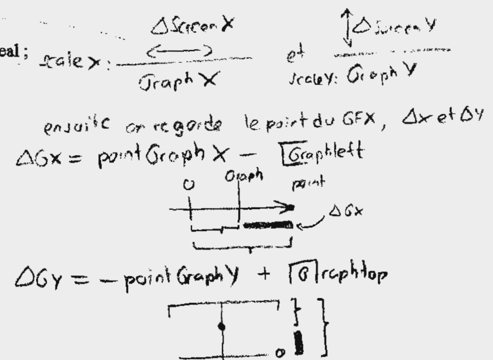
```
procedure SetGraphSize (top, left, bottom, right : real ) ; external ;
procedure SetScreenSize (top, left, bottom, right : integer ) ; external ;
procedure DrawGraphLine ( fromGX, fromGY : real ; toGX, toGY : real) ; external ;
procedure DrawGraphPoint ( hereX, hereY : real ; radius : integer ) ; external ;
procedure Draw3DLine ( fromX, fromY, fromZ : real ; toX, toY, toZ : real ) ; external ;
procedure Draw3DPoint ( hereX, hereY, hereZ : real ; radius : integer ) ; external ;
procedure Draw3DLine2 ( fromX, fromY, fromZ : real ; toX, toY, toZ : real ; DistObsX : real ) ;
external ;
procedure Draw3DPoint2 ( hereX, hereY, hereZ : real ; radius : integer ; DistObsX : real) ;
external ;
procedure CartesianToSpherical (x, y, z : real ; var a, b : real ) ; external ;
```

Graphs.p

```
module Graphs ;
#include "Graphics.h"

const
  Pi = 3.141592654 ;

var
  GRAPH_TOP : real := 1.0 ;
  GRAPH_BOTTOM : real := -1.0 ;
  GRAPH_LEFT : real := 0.0 ;
  GRAPH_RIGHT : real := 6.28 ;

  SCREEN_TOP : integer := 20 ;
  SCREEN_BOTTOM : integer := 380 ;
  SCREEN_LEFT : integer := 20 ;
  SCREEN_RIGHT : integer := 380 ;


procedure SetGraphSize (top, left, bottom, right : real ) ;
begin
  GRAPH_TOP := top ;
  GRAPH_BOTTOM := bottom ;
  GRAPH_LEFT := left ;
  GRAPH_RIGHT := right ;
end ;


procedure SetScreenSize (top, left, bottom, right : integer ) ;
begin
  SCREEN_TOP := top ;
  SCREEN_BOTTOM := bottom ;
  SCREEN_LEFT := left ;
  SCREEN_RIGHT := right ;
end ;
```




```
procedure Scale ( vGX, vGY : real ; var vSX, vSY : integer) ;
var
  distToTopG, distToLeftG : real ;
  scaleX, scaleY : real ;
begin
  scaleX := (SCREEN_RIGHT - SCREEN_LEFT) / (GRAPH_RIGHT - GRAPH_LEFT) ;
  scaleY := (SCREEN_BOTTOM - SCREEN_TOP) / (GRAPH_TOP - GRAPH_BOTTOM) ;
  distToLeftG := (vGX - GRAPH_LEFT) ;
  distToTopG := (GRAPH_TOP - vGY) ;
  vSX := SCREEN_LEFT + round (distToLeftG * scaleX) ;
  vSY := SCREEN_TOP + round (distToTopG * scaleY) ;
end ;


procedure DrawGraphLine ( fromGX, fromGY : real ; toGX, toGY : real) ;
var
  fromSX, fromSY : integer ;
  toSX, toSY : integer ;
begin
  Scale (fromGX, fromGY, fromSX, fromSY) ;
  Scale (toGX, toGY, toSX, toSY) ;
  DrawLine (fromSX, fromSY, toSX, toSY) ;
end ;


procedure DrawGraphPoint ( hereX, hereY : real ; radius : integer ) ;
var
  hereSX, hereSY : integer ;
begin
  Scale (hereX, hereY, hereSX, hereSY) ;
  FillOval ( hereSY-radius, hereSX-radius, hereSY+radius, hereSX+radius) ;
end ;


procedure Projection ( x, y, z : real ; var xp, yp : real ) ;
begin
  xp := y - 0.5 * x ;
  yp := z - 0.5 * x ;
end ;


procedure Projection2 ( x, y, z : real ; var xp, yp : real ; DOX: real ) ;
begin
  xp := y / ((x/DOX) + 1);
  yp := z / ((x/DOX) + 1);
end ;


procedure Draw3DLine ( fx, fy, fz : real ; tx, ty, tz : real ) ;
var
  fxp, fyp : real ;
  txp, typ : real ;
begin
  Projection ( fx, fy, fz, fxp, fyp ) ;
  Projection ( tx, ty, tz, txp, typ ) ;
  DrawGraphLine ( fxp, fyp, txp, typ ) ;
end ;


procedure Draw3DPoint ( fx, fy, fz : real ; radius : integer ) ;
var
  fxp, fyp : real ;
begin
  Projection ( fx, fy, fz, fxp, fyp ) ;
  DrawGraphPoint ( fxp, fyp, radius ) ;
end ;


procedure Draw3DLine2 ( fx, fy, fz : real ; tx, ty, tz : real ; DistObsX : real ) ;
var
  fxp, fyp : real ;
  txp, typ : real ;
begin
  Projection2 ( fx, fy, fz, fxp, fyp, DistObsX ) ;
  Projection2 ( tx, ty, tz, txp, typ, DistObsX ) ;
  DrawGraphLine ( fxp, fyp, txp, typ) ;
end ;


procedure Draw3DPoint2 ( fx, fy, fz : real ; radius : integer ; DistObsX : real ) ;
var
  fxp, fyp : real ;
begin
  Projection2 ( fx, fy, fz, fxp, fyp, DistObsX ) ;
  DrawGraphPoint ( fxp, fyp, radius ) ;
end ;


function ArcCos ( x : real ) : real ;
var
  y : real ;
begin
  if (x = 0) then
    begin
      ArcCos := Pi / 2 ;
    end
  else
    begin
      y := (1 - x*x)/x ;
      ArcCos := arctan (y) ;
    end ;
end ;
```




```
procedure CartesianToSpherical (x, y, z : real ; var a, b : real ) ;
var
  radius : real ;
  psi : real ;
begin
  radius := sqrt (x*x+y*y+z*z) ;
  psi := ArcCos (z/radius) ;
  b := Pi / 2 - psi ;
  if (x = 0) and (y = 0) then
    begin
      a := 0 ;
    end
  else
    begin
      if (y >= 0) then
        begin
          a := ArcCos (x/(radius*sin(psi))) ;
        end
      else
        begin
          a := 2*Pi - ArcCos (x/(radius*sin(psi))) ;
        end ;
    end ;
end ;
```

Ensuite, on est en coord. écran, donc
Ox = ScreenX20 + scaleX · ΔGX
Oy = ScreenY20 + scaleY · ΔGY

Matrices.h

```
type
 Vector3T = array[1..3] of real ;
 Matrix33T = array[1..3,1..3] of real ;

procedure InitializeMatrix(var m: array[lbm1..ubm1: integer; lbm2..ubm2: integer] of real);
external;
procedure InitializeVector(var v: array[lb..ub: integer] of real); external;
procedure VectScalarMult (v: array[lb..ub: integer] of real; s: real; var r: array[lbr..ubr:
integer] of real); external;
procedure MatScalarMult ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; s : real
; var r : array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ; external;
procedure VectVectSum ( v1 : array[lbv1..ubv1:integer] of real ; v2 : array[lbv2..ubv2:integer]
of real ; var r : array[lbr..ubr:integer] of real ) ; external;
procedure MatMatSum ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; n :
array[lbn1..ubn1:integer;lbn2..ubn2:integer] of real ; var r :
array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ; external ;
procedure VectVectMinus ( v1 : array[lbv1..ubv1:integer] of real ; v2 :
array[lbv2..ubv2:integer] of real ; var r : array[lbr..ubr:integer] of real ) ; external;
procedure MatMatMinus ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; n :
array[lbn1..ubn1:integer;lbn2..ubn2:integer] of real ; var r :
array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ; external ;
procedure ProduitVectoriel ( v1 : array[lbv1..ubv1:integer] of real ; v2 :
array[lbv2..ubv2:integer] of real ; var r : array[lbr..ubr:integer] of real ) ; external;
procedure ScalarProduct ( v1 : array[lb1..ub1:integer] of real ; v2 : array[lb2..ub2:integer] of
real ; var r : real ) ; external ;
procedure ProduitMixte ( v1 : array[lb1..ub1:integer] of real ; v2 : array[lb2..ub2:integer] of
real ; v3 : array[lb2..ub2:integer] of real ; var r : real ) ; external ;
procedure WriteVector ( var v : array[lb..ub:integer] of real ) ; external ;
procedure WriteMatrix ( var m : array[lb1..ub1:integer;lb2..ub2:integer] of real ) ; external ;
procedure MatVectMult ( v : array[lbv..ubv:integer] of real ; m :
array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; var r : array[lbr..ubr:integer] of real ) ;
external ;
procedure MatMatMult ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; n :
array[lbn1..ubn1:integer;lbn2..ubn2:integer] of real ; var r :
array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ; external ;
procedure InitXRotationMatrix (var m : Matrix33T ; angle : real ) ; external ;
procedure InitYRotationMatrix (var m : Matrix33T ; angle : real ) ; external ;
procedure InitZRotationMatrix (var m : Matrix33T ; angle : real ) ; external ;
procedure InitXYZRotationMatrix(var m : Matrix33T ; x, y, z: real ) ; external ;
procedure Rotate(v1: Vector3T; x, y, z: real;var r: Vector3T); external ;
procedure SolveSystem( s : array[lbs1..ubs1:integer; lbs2..ubs2:integer] of real ; var r :
array[lbr..ubr:integer] of real ) ; external ;
```

Matrices.p

```
module Matrices ;
type
 Vector3T = array[1..3] of real ;
 Matrix33T = array[1..3,1..3] of real ;


procedure InitializeVector(var v: array[lb..ub: integer] of real);
var
 i: integer;
 x: real;
begin
 x := seed(wallclock);
 for i := lb to ub do
  begin
   v[i] := random(x);
  end;
end;


procedure InitializeMatrix(var m: array[lbm1..ubm1: integer; lbm2..ubm2: integer] of real);

var
 i1, i2: integer;
 x: real;
begin
 x := seed(wallclock);
 for i1 := lbm1 to ubm1 do
  begin
   for i2 := lbm2 to ubm2 do
    begin
     m[i1, i2] := random(x);
    end;
  end;
end;


procedure VectScalarMult (v: array[lb..ub: integer] of real; s: real; var r: array[lbr..ubr:
integer] of real);
var
 range1, range2 : integer ;
 i : integer ;
begin
 range1 := ub - lb ;
 range2 := ubr - lbr ;
 if (range1 <> range2) then
  begin
   writeln ('VecScalarMult : range mismatch') ;
   pcexit (1) ;
  end ;
 for i := 0 to range1 do
  begin
   r[lbr+i] := 0;
   r[lbr+i] := r[lbr+i] + s * v[lb+i];
  end;
end;


procedure MatScalarMult ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; s : real
; var r : array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ;
var
 rangem1, rangem2 : integer ;
 ranger1, ranger2 : integer ;
 i, j, :integer ;
begin
 rangem1 := ubm1 - lbm1 ;
 rangem2 := ubm2 - lbm2 ;
 ranger1 := ubr1 - lbr1 ;
 ranger2 := ubr2 - lbr2 ;
  if (rangem1 <> ranger1) then
   begin
    writeln ('MatScalarMult : range mismatch') ;
    pcexit (1) ;
   end ;
  for i := 0 to ranger1 do
   begin
    for j := 0 to ranger2 do
     begin
      r[lbr1+i,lbr2+j] := 0 ;
      r[lbr1+i,lbr2+j] := r[lbr1+i,lbr2+j] + s * m[lbm1+i,lbm2+k] ;
     end ;
   end ;
end ;


procedure VectVectSum ( v1 : array[lbv1..ubv1:integer] of real ; v2 : array[lbv2..ubv2:integer]
of real ; var r : array[lbr..ubr:integer] of real ) ;
var
 rangev1, rangev2 : integer ;
 ranger : integer ;
 i, j : integer ;
begin
 rangev1 := ubv1 - lbv1 ;
 rangev2 := ubv2 - lbv2 ;
 ranger := ubr - lbr ;
 if (rangev1 <> rangev2) then
  begin
   writeln ('VectVectSum : range mismatch') ;
   pcexit (1) ;
  end ;
 if (rangev1 <> ranger) then
  begin
   writeln ('VectVectSum : range mismatch') ;
   pcexit (1) ;
  end ;
 if (rangev2 <> ranger) then
  begin
   writeln ('VectVectSum : range mismatch') ;
   pcexit (1) ;
  end ;
 for i := 0 to ranger do
  begin
   r[lbr+i] := 0 ;
   r[lbr+i] := r[lbr+i] + v1[lbv1+i] + v2[lbv2+i];
  end ;
end ;


procedure MatMatSum ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; n :
array[lbn1..ubn1:integer;lbn2..ubn2:integer] of real ; var r :
array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ;
var
 rangem1, rangem2 : integer ;
 rangen1, rangen2 : integer ;
 ranger1, ranger2 : integer ;
 i, j, :integer ;
begin
 rangem1 := ubm1 - lbm1 ;
 rangem2 := ubm2 - lbm2 ;
 rangen1 := ubn1 - lbn1 ;
 rangen2 := ubn2 - lbn2 ;
 ranger1 := ubr1 - lbr1 ;
 ranger2 := ubr2 - lbr2 ;
  if (rangem1 <> ranger1) then
   begin
    writeln ('MatMatSum : range mismatch') ;
    pcexit (1) ;
   end ;
```

```
    if (rangem2 <> rangen1) then
      begin
        writeln ('MatMatSum : range mismatcb') ;
        pcexit (1) ;
      end ;
    if (rangen2 <> ranger2) then
      begin
        writeln ('MatMatSum : range mismatcb') ;
        pcexit (1) ;
      end ;
    for i := 0 to ranger1 do
      begin
        for j := 0 to ranger2 do
          begin
            r[lbr1+i,lbr2+j] := 0 ;
            r[lbr1+i,lbr2+j] := r[lbr1+i,lbr2+j] + m[lbm1+i,lbm2+k] + n[lbn1+k,lbn2+j] ;
          end ;
      end ;
end ;


procedure VectVectMinus ( v1 : array[lbv1..ubv1:integer] of real ; v2 :
array[lbv2..ubv2:integer] of real ; var r : array[lbr..ubr:integer] of real ) ;
var
  rangev1, rangev2 : integer ;
  ranger : integer ;
  i, j : integer ;
begin
  rangev1 := ubv1 - lbv1 ;
  rangev2 := ubv2 - lbv2 ;
  ranger := ubr - lbr ;
  if (rangev1 <> rangev2) then
    begin
      writeln ('VectVectMinus : range mismatch') ;
      pcexit (1) ;
    end ;
  if (rangev1 <> ranger) then
    begin
      writeln ('VectVectMinus : range mismatch') ;
      pcexit (1) ;
    end ;
  if (rangev2 <> ranger) then
    begin
      writeln ('VectVectMinus : range mismatch') ;
      pcexit (1) ;
    end ;
  for i := 0 to ranger do
    begin
      r[lbr+i] := 0 ;
      r[lbr+i] := r[lbr+i] + v1[lbv1+i] - v2[lbv2+i];
    end ;
end ;


procedure MatMatMinus ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; n :
array[lbn1..ubn1:integer;lbn2..ubn2:integer] of real ; var r :
array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ;
var
  rangem1, rangem2 : integer ;
  rangen1, rangen2 : integer ;
  ranger1, ranger2 : integer ;
  i, j, :integer ;
begin
  rangem1 := ubm1 - lbm1 ;
  rangem2 := ubm2 - lbm2 ;
  rangen1 := ubn1 - lbn1 ;
  rangen2 := ubn2 - lbn2 ;
  ranger1 := ubr1 - lbr1 ;
  ranger2 := ubr2 - lbr2 ;
    if (rangem1 <> ranger1) then
      begin
        writeln ('MatMatMinus : range mismatch') ;
        pcexit (1) ;
      end ;
    if (rangem2 <> rangen1) then
      begin
        writeln ('MatMatMinus : range mismatch') ;
        pcexit (1) ;
      end ;
    if (rangen2 <> ranger2) then
      begin
        writeln ('MatMatMinus : range mismatcb') ;
        pcexit (1) ;
      end ;
    for i := 0 to ranger1 do
      begin
        for j := 0 to ranger2 do
          begin
            r[lbr1+i,lbr2+j] := 0 ;
            r[lbr1+i,lbr2+j] := r[lbr1+i,lbr2+j] - m[lbm1+i,lbm2+k] - n[lbn1+k,lbn2+j] ;
          end ;
      end ;
end ;


procedure ProduitVectoriel ( v1 : array[lbv1..ubv1:integer] of real ; v2 :
array[lbv2..ubv2:integer] of real ; var r : array[lbr..ubr:integer] of real ) ;
var
  rangev1, rangev2 : integer ;
  ranger : integer ;
  i, j : integer ;
begin
  rangev1 := ubv1 - lbv1 ;
  rangev2 := ubv2 - lbv2 ;
  ranger := ubr - lbr ;
  if (rangev1 <> rangev2) then
    begin
      writeln ('Cross : range mismatcb') ;
      pcexit (1) ;
    end ;
  if (rangev1 <> ranger) then
    begin
      writeln ('Cross : range mismatch') ;
      pcexit (1) ;
    end ;
  if (rangev2 <> ranger) then
    begin
      writeln ('Cross : range mismatcb') ;
      pcexit (1) ;
    end ;
  if (rangev1 <> 3) then
    begin
      writeln ('Cross : range mismatch') ;
      pcexit (1) ;
    end ;
  r[lbr] := v1[2] * v2[3] - v1[3] * v2[2];
  r[lbr+1] := v1[3] * v2[1] - v1[1] * v2[3];
  r[lbr+2] := v1[1] * v2[2] - v1[2] * v2[1];
end ;



procedure ScalarProduct ( v1 : array[lb1..ub1:integer] of real ; v2 : array[lb2..ub2:integer] of
real ; var r : real ) ;
var
  range1, range2 : integer ;
  i : integer ;
begin
  range1 := ub1 - lb1 ;
  range2 := ub2 - lb2 ;
  if (range1 <> range2) then
    begin
      writeln ('ScalarProduct : range mismatcb') ;
      pcexit (1) ;
    end ;
  r := 0 ;
  for i := 0 to range1 do
    begin
      r := r + v1[lb1+i]*v2[lb2+i] ;
    end ;
end ;


procedure ProduitMixte ( v1 : array[lb1..ub1:integer] of real ; v2 : array[lb2..ub2:integer] of
real ; v3 : array[lb2..ub2:integer] of real ; var r : real );
var
  rangev1, rangev2 : integer ;
  ranger : integer ;
  i, j : integer ;
  temp: Vector3T;
begin
  rangev1 := ubv1 - lbv1 ;
  rangev2 := ubv2 - lbv2 ;
  ranger := ubr - lbr ;
  if (rangev1 <> rangev2) then
    begin
      writeln ('ProduitMixte : range mismatch') ;
      pcexit (1) ;
    end ;
  if (rangev1 <> rangev3) then
    begin
      writeln ('ProduitMixte : range mismatch') ;
      pcexit (1) ;
    end ;
  if (rangev1 <> ranger) then
    begin
      writeln ('ProduitMixte : range mismatch') ;
      pcexit (1) ;
    end ;
  if (rangev2 <> range3) then
    begin
```

```
    writeln ('ProduitMixte : range mismatch') ;
    pcexit (1) ;
   end ;
  if (rangev2 <> ranger) then
   begin
    writeln ('ProduitMixte : range mismatch') ;
    pcexit (1) ;
   end ;
  if (rangev3 <> ranger) then
   begin
    writeln ('ProduitMixte : range mismatch') ;
    pcexit (1) ;
   end ;
  if (rangev1 <> 3) then
   begin
    writeln ('Cross : range mismatch') ;
    pcexit (1) ;
   end ;
  ProduitVectoriel(v2, v3, temp);
  ScalarProduct(v1, temp, r);
end ;


procedure WriteVector ( var v : array[lb..ub:integer] of real ) ;
var
 i : integer ;
begin
 for i := lb to ub do
  begin
   write (v[i]:8:3) ;
  end ;
 writeln ;
end ;


procedure WriteMatrix ( var m : array[lb1..ub1:integer;lb2..ub2:integer] of real ) ;
var
 i, j : integer ;
begin
 for i := lb1 to ub1 do
  begin
   for j := lb2 to ub2 do
    begin
     write (m[i,j]:8:3) ;
    end ;
   writeln ;
  end ;
 writeln ;
end ;


procedure MatVectMult ( v : array[lbv..ubv:integer] of real ; m :
array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; var r : array[lbr..ubr:integer] of real ) ;
var
 rangem1, rangem2 : integer ;
 rangev, ranger : integer ;
 i, j : integer ;
begin
 rangem1 := ubm1 - lbm1 ;
 rangem2 := ubm2 - lbm2 ;
 rangev := ubv - lbv ;
 ranger := ubr - lbr ;
 if (rangem2 <> rangev) then
  begin
   writeln ('MatVectMult : range mismatch') ;
   pcexit (1) ;
  end ;
 if (rangem1 <> ranger) then
  begin
   writeln ('MatVectMult : range mismatch') ;
   pcexit (1) ;
  end ;
 for i := 0 to ranger do
  begin
   r[lbr+i] := 0 ;
   for j := 0 to rangev do
    begin
     r[lbr+i] := r[lbr+i] + m[lbm1+i,lbm2+j]*v[lbv+j] ;
    end ;
  end ;
end ;


procedure MatMatMult ( m : array[lbm1..ubm1:integer;lbm2..ubm2:integer] of real ; n :
array[lbn1..ubn1:integer;lbn2..ubn2:integer] of real ; var r :
array[lbr1..ubr1:integer;lbr2..ubr2:integer] of real ) ;
var
 rangem1, rangem2 : integer ;
 rangen1, rangen2 : integer ;
 ranger1, ranger2 : integer ;
 i, j, k :integer ;
begin
 rangem1 := ubm1 - lbm1 ;
 rangem2 := ubm2 - lbm2 ;
 rangen1 := ubn1 - lbn1 ;
 rangen2 := ubn2 - lbn2 ;
 ranger1 := ubr1 - lbr1 ;
 ranger2 := ubr2 - lbr2 ;
  if (rangem1 <> ranger1) then
   begin
    writeln ('MatMatMult : range mismatch') ;
    pcexit (1) ;
   end ;
  if (rangem2 <> rangen1) then
   begin
    writeln ('MatMatMult : range mismatch') ;
    pcexit (1) ;
   end ;
  if (rangen2 <> ranger2) then
   begin
    writeln ('MatMatMult : range mismatch') ;
    pcexit (1) ;
   end ;
  for i := 0 to ranger1 do
   begin
    for j := 0 to ranger2 do
     begin
      r[lbr1+i,lbr2+j] := 0 ;
      for k := 0 to rangem2 do
       begin
        r[lbr1+i,lbr2+j] := r[lbr1+i,lbr2+j] + m[lbm1+i,lbm2+k]*n[lbn1+k,lbn2+j] ;
       end ;
     end ;
   end ;
end ;


procedure InitXRotationMatrix (var m : Matrix33T ; angle : real ) ;
begin
 m[1,1] := 1.0 ; m[1,2] := 0.0 ;       m[1,3] := 0.0 ;
 m[2,1] := 0.0 ; m[2,2] := cos(angle) ; m[2,3] := -sin(angle) ;
 m[3,1] := 0.0 ; m[3,2] := sin(angle) ; m[3,3] := cos(angle) ;
end ;


procedure InitYRotationMatrix (var m : Matrix33T ; angle : real ) ;
begin
 m[1,1] := cos(angle) ; m[1,2] := 0.0 ; m[1,3] := -sin(angle) ;
 m[2,1] := 0.0 ;       m[2,2] := 1.0 ; m[2,3] := 0.0 ;
 m[3,1] := sin(angle) ; m[3,2] := 0.0 ; m[3,3] := cos(angle) ;
end ;


procedure InitZRotationMatrix (var m : Matrix33T ; angle : real ) ;
begin
 m[1,1] := cos(angle) ; m[1,2] := -sin(angle) ; m[1,3] := 0.0 ;
 m[2,1] := sin(angle) ; m[2,2] := cos(angle) ;  m[2,3] := 0.0 ;
 m[3,1] := 0.0 ;       m[3,2] := 0.0 ;        m[3,3] := 1.0 ;
end ;


procedure InitXYZRotationMatrix(var m : Matrix33T ; x, y, z: real ) ;
begin
 m[1,1] := cos(z)*cos(y);
 m[1,2] := cos(x)*sin(z) - sin(y)*cos(z)*sin(x);
 m[1,3] := -sin(x)*sin(z) - sin(y)*cos(z)*cos(x);
 m[2,1] := -cos(y)*sin(z);
 m[2,2] := cos(z)*cos(x) + sin(x)*sin(z)*sin(y);
 m[2,3] := -cos(z)*sin(x) + sin(z)*sin(y)*cos(x);
 m[3,1] := sin(y);
 m[3,2] := cos(y)*sin(x);
 m[3,3] := cos(x)*cos(y);
end ;


procedure Rotate(v1: Vector3T; x, y, z: real; var r: Vector3T);
var
 m: Matrix33T;
begin
 InitXYZRotationMatrix(m, x, y, z);
 MatVectMult(v1, m, r);
end;


procedure FindPivot ( var s : array[lbs1..ubs1:integer; lbs2..ubs2:integer] of real ; column :
integer ; var row : integer ) ;
var
 i : integer ;
 max : real ;
begin
 max := 0 ;
 for i := column to ubs1 do
```

```
  begin
    if (abs(s[i,column]) > max) then
      begin
        max := abs(s[i,column]) ;
        row := i ;
      end ;
  end ;
 if (max < 0.000001) then
  begin
    writeln ('SolveSystem : pivot nul, programme interrompu') ;
    pcexit (1) ;
  end ;
end ;


procedure SwapRow( var s : array[lbs1..ubs1:integer; lbs2..ubs2:integer] of real ; row1, row2 :
integer) ;
var
 i : integer ;
 tmp : real ;
begin
 for i := lbs2 to ubs2 do
  begin
    tmp := s[row1,i] ;
    s[row1,i] := s[row2,i] ;
    s[row2,i] := tmp ;
  end ;
end ;


procedure EliminateColumn( var s : array[lbs1..ubs1:integer; lbs2..ubs2:integer] of real ; col :
integer ) ;
var
 i, j : integer ;
 factor : real ;
begin
 for i := col+1 to ubs1 do
  begin
    factor := s[i,col]/s[col,col] ;
    for j := col to ubs2 do
      begin
        s[i,j] := s[i,j] - factor * s[col,j] ;
      end ;
  end ;
end ;


procedure SolveSystem( s : array[lbs1..ubs1:integer; lbs2..ubs2:integer] of real ; var r :
array[lbr..ubr:integer] of real ) ;
var
 row : integer ;
 i, j : integer ;
begin
 (* Array bound checks *)
 if ((lbs1 <> 1) or (lbs2 <> 1) or (lbr <> 1)) then
  begin
    writeln ('SolveSystem : array bounds must be equal to 1') ;
    pcexit (1) ;
  end ;
 if (ubs1 <> ubr) then
  begin
    writeln ('SolveSystem : array bound mismatch') ;
    pcexit (1) ;
  end ;
 if (ubs1 + 1 <> ubs2) then
  begin
    writeln ('SolveSystem :  array bound mismatch') ;
  end ;

 (* triangularize system *)
 for i := 1 to ubs1 do
  begin
    FindPivot (s, i, row) ;
    SwapRow (s, i, row) ;
    EliminateColumn (s, i) ;
    writeln ('step : ', i) ;
    WriteMatrix (s) ;
  end ;

 (* solve system *)
 for i := ubs1 downto 1 do
  begin
    r[i] := s[i,ubs2] ;
    for j := i+1 to ubs1 do
      begin
        r[i] := r[i] - s[i,j]*r[j] ;
      end ;
    r[i] := r[i] / s[i,i] ;
  end ;
end ;
```

Functions. h

```
function ASIN(numero :real) :real; external;
function ACOS(numero :real) :real; external;
function ATAN(numero :real) :real; external;
function TAN(numero :real) :real;  external;
function COT(numero :real) :real;  external;
function ACOT(numero :real) :real; external;
function ASINH(numero :real) :real; external;
function ACOSH(numero :real) :real; external;
function ATANH(numero :real) :real; external;
function SINH(numero :real) :real; external;
function COSH(numero :real) :real; external;
function TANH(numero :real) :real; external;
function COTH(numero :real) :real; external;
function ACOTH(numero :real) :real; external;
function SIGN(numero :real) :real; external;
function FACT(numero :integer) :real; external;
function COMB(n, p:integer) :integer; external;
function LOG(numero :real) :real;  external;
function LN(numero :real) :real; external;
function LOGXN(numero, base :real) :real; external;
function ROOT(numero :real) :real; external;
function INV(numero :real) :real;  external;
function PUISSANCE(numero,pot :real): real; external;
```

Functions. p

```
module Functions


const
pi = 3.14159265358979;


function ATAN(numero :real) :real;
begin
 ATAN:=arctan(numero);
end;


function ASIN(numero :real) :real;
begin
 if abs(numero) = 1 then
  if numero = 1 then
   ASIN := pi/2
  else
   ASIN := -pi/2
 else
  ASIN := arctan(numero / sqrt(1 - sqr(numero)));
end;


function ACOS(numero :real) :real;
var
 y: real;
begin
 if numero = 0 then
  ACOS := pi / 2
 else
  begin
   y := (1 - numero * numero) / numero;
   ACOS := arctan(y);
  end;
end;


function TAN(numero :real) :real;
begin
 TAN := sin(numero) / cos(numero);
end;


function COT(numero :real) :real;
begin
 COT := cos(numero) / sin(numero);
end;


function ACOT(numero :real) :real;
begin
 ACOT := pi / 2-arctan(numero);
end;


function SINH(numero :real) :real;
begin
 SINH := (exp(numero) - exp(-numero)) / 2;
end;


function COSH(numero :real) :real;
begin
 COSH := (exp(numero) + exp(-numero)) / 2;
end;


function ASINH(numero :real) :real;
begin
 ASINH := ln(numero + sqrt(sqr(numero) + 1));
end;


function ACOSH(numero :real) :real;
begin
 ACOSH := ln(numero + sqrt(sqr(numero) - 1));
end;


function ATANH(numero :real) :real;
begin
 ATANH := 1 / 2 * ln((1+numero) / (1-numero));
end;


function TANH(numero :real) :real;
begin
 TANH := (exp(numero) - exp(-numero)) / (exp(numero) + exp( - numero));
end;


function COTH(numero :real) :real;
begin
 COTH := (exp(numero) + exp(-numero)) / (exp(numero) - exp(-numero));
end;


function ACOTH(numero :real) :real;
begin
 ACOTH := 1 / 2 * ln((1+numero) / (numero - 1));
end;


function SIGN(numero :real) :real;
begin
 if numero > 0.0 then
  SIGN := 1.0
 else
  if numero < 0.0 then
   SIGN := -1.0
  else SIGN := 0.0;
end;


function FACT(numero :integer) :real;
var
 k :integer;
 ans :real;
begin
 ans := 1.0;
 for k := 2 to numero do
  ans := k * ans;
 FACT := ans
end;


function COMB(n, p :integer) :integer;
var
 k :integer;
 ans :integer;
 FACTn :integer;
 FACTp :integer;
 FACTnp :integer;
begin
 if (n >= p) and (n >= 0) and (p >= 0) then
  begin
   ans := 1;
   for k := 2 to n do
```

```
      begin
        ans := k * ans;
      end;
    FACTn := ans;
    ans := 1;
    for k := 2 to p do
      begin
        ans := k * ans;
      end;
    FACTp := ans;
    ans := 1;
     for k := 2 to (n-p) do
       begin
         ans := k * ans;
       end;
    FACTnp := ans;
    COMB := FACTn div (FACTnp * FACTp);
  end;
end;


function INV(numero :real) :real;
begin
 INV := 1 / numero;
end;


function LOG(numero :real) :real;
begin
 LOG := ln(numero) / ln(10);
end;


function LN(numero :real) :real;
begin
 LN := ln(numero);
end;


function LOGXN(numero, base: real) :real;
begin
 LOGXN := ln(numero) / ln(base);
end;


function ROOT(numero :real): real;
begin
 ROOT := sqrt(numero);
end;


function PUISSANCE(numero,pot :real): real;
begin
 if numero <> 0 then
  begin
   if numero > 0 then
    numero := exp(ln(numero) * pot)
   else
    if (pot - trunc(pot))=0 then
     if trunc(pot) mod 2 = 1 then
      numero := -exp(ln(- numero) * pot)
     else
      numero := exp(ln(- numero) * pot)
    else
     if (1 / pot - trunc(1 / pot))=0 then
      if trunc(1 / pot) mod 2 = 1 then
       numero := - exp(ln(- numero) * pot)
      else
       numero := exp(ln(- numero) * pot)
    end;
   PUISSANCE := numero;
end;
```

Graphics.h

```
procedure FillRectangle (top, left, bottom, right : integer) ; external ;
procedure DrawRectangle (top, left, bottom, right : integer) ; external ;
procedure FillOval (top, left, bottom, right : integer) ; external ;
procedure DrawOval (top, left, bottom, right : integer) ; external ;
procedure DrawLine (fromX, fromY, toX, toY : integer) ; external ;
procedure PenSize (pixels : integer) ; external ;
procedure SetColor (color : integer) ; external ;
procedure Delay (millisec : integer ) ; external ;
procedure GetTime (var millisec : integer) ; external ;
procedure SuspendRefresh ; external ;
procedure ResumeRefresh ; external ;
procedure SetWindowSize (sizeX, sizeY : integer) ; external ;
procedure StartSingleCharacterMode ; external ;
procedure FinishSingleCharacterMode ; external ;
procedure GetSingleCharacter (var c : char) ; external ;
procedure CheckForSingleCharacter (var c : char) ; external ;
procedure ClearWindow ; external ;

function NewColor (red, green, blue : integer) : integer ; external ;

type StringT = varying[64] of char ;
procedure DrawString ( x, y : integer ; var string : StringT ) ; external ;
procedure IntegerToString ( i : integer ; var s : StringT ) ; external ;
procedure RealToString ( r : real ; var s : StringT ) ; external ;
```

Exercices

```
program Exercice_25;

var
 MotIn,MotOut: varying[1..255] of char ;
 i,j : integer;
begin
 MotIn:='Lausanne';
 j:=1;
 for i:= 1 to length(MotIn) do
  if not(MotIn[i]='a') then
   begin    MotOut[j]:=MotIn[i];
    j:=j+1;
   end;
  MotOut[1]:=chr(j);
  writeln(MotOut)
end.
```

```
program Exercice_24;

var
 mot: varying[255] of char ;
 i:integer;
begin
 readln(mot);
 for i:= 1 to length(mot) do
  writeln(ord(mot[i]));
end.
```

```
program Exercice_24;

var
 mot1,mot2: varying[255] of char ;
 i:integer;
 identique:boolean;

begin
 identique:=true;
 readln(mot1);
 readln(mot2);
 for i:= 1 to length(mot1) do
  begin
   if not(mot1[i]=mot2[i]) then
    begin
     identique:=false;
    end ;
  end ;
  if identique then
   begin
    writeln('les mots sont identiques') ;
   end
  else
   begin
    writeln ('les mots ne sont pas identiques') [
   end ;
end.
```

```
program repeatDeviner;

var
 nombre_lu,
 nombre_cache : integer;
 est_trouve   : boolean;

begin
 nombre_lu:= seed(wallclock);
 nombre_cache:= trunc(random(1)*5000) ;
 est_trouve := false;
 repeat
  write('Proposez un nombre: ');
  readln(nombre_lu);
  if nombre_lu < nombre_cache then
   writeln('Trop petit')
  else
   if nombre_lu > nombre_cache then
    writeln('Trop grand')
   else
    est_trouve := true;
 until est_trouve;
 writeln('Vous avez trouvé');
end.
```

```
program encoreDeviner;

var
 nombre_lu,
 nombre_cache : integer;
 est_trouve   : boolean;

begin
 nombre_lu:= seed(wallclock);
 nombre_cache:= trunc(random(1)*5000) ;
 est_trouve := false;
 while not(est_trouve) do
 begin
  write('Proposez un nombre: ');
  readln(nombre_lu);
  if nombre_lu < nombre_cache then
   writeln('Trop petit')
  else
   if nombre_lu > nombre_cache then
    writeln('Trop grand')
   else
    est_trouve := true;
 end;{ while }
 writeln('Vous avez trouvé');
end.
```

```
program devinerMot;

var
 mot_cache, mot_trouve : varying[20] of char;
 lettre: char;
 i, longueur_mot : integer;
 est_trouve : boolean;

begin
 mot_cache := 'programme';
 longueur_mot := length(mot_cache);
 for i:=1 to longueur_mot do
  mot_trouve := mot_trouve + '*';
 est_trouve := false;
 repeat
  write('Voici le mot caché : ');
  writeln(mot_trouve);
  write('Proposez une lettre: ');
  readln(lettre);
  for i:=1 to longueur_mot do
   if mot_cache[i] = lettre then
    mot_trouve[i] := lettre;
   if mot_trouve = mot_cache then
    est_trouve := true;
 until est_trouve;
 writeln('Vous avez trouvé!');
 writeln('Le mot caché est "', mot_trouve,'"');
end.
```

```
program echelle;

(* ce programme laisse le sommet de la maison fixe *)
(* plutot que le coin inferieur gauche             *)

#include "Graphics.h"

const
 x = 50;
 y = 40;
 x1 = 150;
 y1 = 130;
 x2 = 100;
 y2 = 10;

var
 echelle : real;
 new_x  : integer;
 new_x1 : integer;
 new_x2 : integer;
 new_y  : integer;
 new_y1 : integer;
 new_y2 : integer;
 begin
 writeln('donnez l''echelle:');
 readln(echelle);
```

```
 new_x:=x2+round((x-x2)*echelle);
 new_x1:=x2+round((x1-x2)*echelle);
 new_x2:=x2;
 new_y:=y2+round((y-y2)*echelle);
 new_y1:=y2+round((y1-y2)*echelle);
 new_y2:=y2;
 DrawRectangle(new_y,new_x,new_y1,new_x1);
 DrawLine(new_x,new_y,new_x2,new_y2);
 DrawLine(new_x2,new_y2,new_x1,new_y);
 DrawLine(new_x1,new_y,new_x,new_y1);
 DrawLine(new_x,new_y,new_x1,new_y1);
 readln;
end.
```

```
program sinus;
#include "Graphics.h"

var
 x      : integer;
 x_prec : integer;
 y      : integer;
 y_prec : integer;

begin
 SetWindowSize(200,200);
 x_prec:=0;
 y_prec:=100;
 for x:=1 to 200 do
  begin
   y:= trunc(100*sin(x*3.14156/50))+100;
   DrawLine(x_prec,y_prec,x,y);
   x_prec:=x;
   y_prec:=y;
  end;
 readln;
end.
```

```
program minimum;

const
 pas = 0.01;

var
 i   : real;
 min : real;
 val_inter  : real;

begin
 i:=0;
 min:=1;
 while i<=4 do
  begin
   val_inter:=sin(i);
   if val_inter<min then
    begin
     min:=val_inter;
    end;
   i:=i+pas;
  end;
 writeln('la valeur min trouvee est :',min);
 readln;
end.
```

```
program minmax;
#include "Graphics.h"

const
 pas = 0.01;

var
 i         : real;
 min       : real;
 max       : real;
 val_inter : real;
 x         : integer;
 x_prec    : integer;
 y         : integer;
 y_prec    : integer;
 offset    : real;
 facteur   : real;

begin
 i:=0;
 min:=1;
 max:=-1;
 while i<=4 do
  begin
   val_inter:=sin(i)*sin(i+1/3);
   if val_inter<min then
    begin
     min:=val_inter;
    end;
   if val_inter>max then
    begin
     max:=val_inter;
    end;
   i:=i+pas;
  end;
 i:=0;
 x_prec:=0;
 y_prec:=trunc(offset);
 facteur:=160/(max-min);
 offset:=trunc(100-(max+min)/2*facteur);
 while i<=4 do
  begin
   x:=trunc(i*100);
   y:=trunc((sin(x/100)*sin(x/100+1/3))*facteur+offset);
   DrawLine(x_prec,y_prec,x,y);
   DrawLine(0,20,400,20);
   DrawLine(0,180,400,180);
   x_prec:=x;
   y_prec:=y;
   i:=i+pas;
  end;
 writeln('la valeur min trouvee est :',min);
 writeln('la valeur maximum trouvee est :',max);
 readln;
end.
```

```
program lissajou;
#include "Graphics.h"

var
 i,pas       : real;
 nbre_points : integer;
 cst_y       : real;
 max         :  real ;
 x           : integer;
 y           : integer;
 x_prec      : integer;
 y_prec      : integer;

begin
 writeln('Donner la valeur du nombre de points voulue:');
 readln(nbre_points);
 writeln('Donner la valeur du facteur multiplicatif de y:');
 readln(cst_y);
 i:=0;
 x_prec:=100;
 y_prec:=100;
 max := 2*3.14156 /50 * nbre_points;
 while i<=max do
  begin
   x:=trunc(100*sin(i))+100;
   y:=trunc(100*sin(cst_y*i))+100;
   pas:=(2*3.14156/50);
   i:=i+pas;
   DrawLine(x_prec,y_prec,x,y);
   x_prec:=x;
   y_prec:=y;
  end;
 readln;
end.
```

```
{ex40}
program tortue;
#include "Graphics.h"

var
 x, y : integer;
 c    : char;

begin
 x := 100;
```

```
  y := 100;
  FillOval(y,x,y+4,x+4);
  writeln(' Pour deplacer le point, taper d (droite), g (gauche)');
  writeln(' h (haut), b (bas) ou q (quitter)');
  readln(c);
  while c<>'q' do
   begin
    case c of
     'd' :    x:= x+5;
     'g' :    x:= x-5;
     'h' :    y:= y-5;
     'b' :    y:= y+5;
     'q' :    writeln(' Au revoir...');
    otherwise
     writeln(' Non-valable, taper d,g,h,b ou q ');
    end; { case }
    FillOval(y,x,y+4,x+4);
    readln(c);
   end; { While }
end.
```

```
{ex42}
program triSimple;
var
  v1, v2 : array [1..6] of integer;
  i, j, minimum, indice  : integer;

begin
  v1[1] := 3;
  v1[2] := 6;
  v1[3] := 1;
  v1[4] := 9;
  v1[5] := 2;
  v1[6] := 5;
  writeln(' Vecteur non-trié: ');
  for i := 1 to 6 do
   write(v1[i]:3);
  writeln;
  for i := 1 to 6 do
   begin
    minimum := 1000;
    for j := 1 to 6 do
    begin
     if v1[j] < minimum then
     begin
      minimum := v1[j];
      indice  := j;
     end; { if }
    end; { for }
   v2[i] := v1[indice];
   v1[indice] := 1000;
  end; { for }
  writeln(' Vecteur trié    : ');
  for i := 1 to 6 do
   write(v2[i]:3);
  writeln;
end.
```

```
{ex43}
program meilleurTri;
var
  v1 : array [1..6] of integer;
  i, j, minimum, indice, temp  : integer;

begin
  v1[1] := 3;
  v1[2] := 6;
  v1[3] := 1;
  v1[4] := 9;
  v1[5] := 2;
  v1[6] := 5;
  writeln(' Vecteur non-trié: ');
  for i := 1 to 6 do
   write(v1[i]:3);
  writeln;
  for i := 1 to 6 do
   begin
    {-- calcul du minimum: -- }
    minimum := 1000;
    for j := i to 6 do
     begin
      if v1[j] < minimum then
      begin
       minimum := v1[j];
       indice  := j;
      end; { if }
     end; { for }
    {-- echange du minimum avec l'element d'indice i: -- }
    temp := v1[i];
    v1[i] := v1[indice];
    v1[indice] := temp;
   end; { for }
  writeln(' Vecteur trié    : ');
  for i := 1 to 6 do
   write(v1[i]:3);
  writeln;
end.
```

```
{ex44}
program prodMatrice;
#include "Graphics.h"

const angle = 3.14/2; { rotation de 90 degrés }
  orig_x  = 100;
  orig_y  = 100;
  bord    = 3;
  type
  T_vecteur = array [1..2] of real;
  T_matrice =  array [1..2,1..2] of real;

var
  v1, v2          : T_vecteur;
  mat             : T_matrice;
  largeur, hauteur : integer;

begin
  {-- coordonnées du vecteur original: --}
  v1[1] := 80;
  v1[2] := 20;

  {-- matrice de rotation: --}
  mat[1,1] := cos(angle);
  mat[1,2] := -sin(angle);
  mat[2,1] := sin(angle);
  mat[2,2] := cos(angle);

  {-- calcul de la rotation: --}
  v2[1] := mat[1,1]*v1[1] + mat[1,2]*v1[2];
  v2[2] := mat[2,1]*v1[1] + mat[2,2]*v1[2];

  {-- axes: --}
  largeur := 2*bord + 2*orig_x;
  hauteur := 2*bord + 2*orig_y;
  SetWindowSize(largeur , hauteur) ;
  DrawLine (bord, orig_x, largeur - bord, orig_y); { axe horizontal }
  DrawLine (orig_x, bord, orig_x, hauteur - bord); { axe vertical   }
  {-- vecteur original: --}
  DrawLine (orig_x, orig_y, orig_x + trunc(v1[1]), orig_y - trunc(v1[2]));
  {-- vecteur apres rotation: --}
  DrawLine (orig_x, orig_y, orig_x + trunc(v2[1]), orig_y - trunc(v2[2]));

  writeln(' hit return ... ');
  readln;
end.
```

```
{ex45}
program encoreMatrice;
#include "Graphics.h"

const   orig_x  = 100;
  orig_y  = 100;
  bord    = 3;

type
  T_vecteur = array [1..2] of real;
  T_matrice =  array [1..2,1..2] of real;

var
  angle  : real;
  v1, v2 : T_vecteur;
  mat    : T_matrice;
  largeur, hauteur, i, j: integer;

begin
  {-- coordonnées du vecteur original: --}
  v1[1] := 80;
  v1[2] := 20;
  largeur := 2*bord + 2*orig_x;
  hauteur := 2*bord + 2*orig_y;
  SetWindowSize(largeur , hauteur);
  {-- axes: --}
  DrawLine (bord, orig_x, largeur - bord, orig_y); { axe horizontal }
```

```
  DrawLine (orig_x, bord, orig_x, hauteur - bord); { axe vertical  }
  {-- vecteur original: --}
  DrawLine (orig_x, orig_y, orig_x + trunc(v1[1]), orig_y - trunc(v1[2]));
  SuspendRefresh;
  writeln(' Entrez l''angle de rotation (en radians): ');
  readln(angle);
  {-- matrice de rotation: --}
  mat[1,1] := cos(angle);
  mat[1,2] := -sin(angle);
  mat[2,1] := sin(angle);
  mat[2,2] := cos(angle);

  {-- calcul de la rotation: --}
  for i := 1 to 2 do
  begin
   v2[i] := 0.0;
   for j := 1 to 2 do
     v2[i] := v2[i] + mat[i,j]*v1[j];
   end; { for }
  {-- vecteur apres rotation: --}
  DrawLine (orig_x, orig_y, orig_x + trunc(v2[1]), orig_y - trunc(v2[2]));
  ResumeRefresh;
  writeln(' hit return ... ');
  readln;
end.
```

Chambres

```
program chambresExercice;

type
  ChambreT = record
          Nom: varying[32] of char;
          c1, c2, c3: integer;
        end;
var
  Chambres: array[1..4] of ChambreT;
  erreur: boolean;
  NotreChambre: integer;
  choix1, choix2, choix3: integer;
  nom1, nom2, nom3: varying[32] of char;

procedure InitialisationDeChambres;
  begin
    with Chambres[1] do
      begin
        Nom := 'salle a manger';
        c1 := 2;
        c2 := 3;
        c3 := 0;
      end;
    with Chambres[2] do
      begin
        Nom := 'cuisine';
        c1 := 1;
        c2 := 2;
        c3 := 4;
      end;
    with Chambres[3] do
      begin
        Nom := 'salle de jeux';
        c1 := 1;
        c2 := 2;
        c3 := 3;
      end;
    with Chambres[4] do
      begin
        Nom := 'bibliotheque';
        c1 := 1;
        c2 := 3;
        c3 := 4;
      end;
  end;

begin
  InitialisationDeChambres;
  erreur := false;
  NotreChambre := 1;
  repeat
    writeln ('Dans quelle chambre voulez-vous aller?');
    writeln ('Vous etes dans la chambre: ', Chambres[NotreChambre].Nom);
    writeln ('Vous pouvez aller:');
    choix1 := Chambres[NotreChambre].c1;
    choix2 := Chambres[NotreChambre].c2;
    choix3 := Chambres[NotreChambre].c3;
    nom1 := Chambres[choix1].Nom;
    nom2 := Chambres[choix2].Nom;
    writeln (choix1,' - ',nom1);
    writeln (choix2,' - ',nom2);
    if choix3 > 0 then
      begin
        nom3 := Chambres[choix3].Nom;
        writeln (choix3,' - ',nom3);
      end;
    if choix3 = 0 then
      writeln;
    readln(NotreChambre);
   if (NotreChambre <> choix1) and (NotreChambre <> choix2) and (NotreChambre <> choix3)
then
      erreur := true;
   if (NotreChambre < 1) or (NotreChambre > 4) then
      erreur := true;
  until erreur;
  writeln('Fin du programme.');
end.
```

Chambres - Pointeur

```
program chambresExercice2;

type
  ChambrePT = ^ChambreT;
  ChambreT = record
          Nom: varying[32] of char;
          c1, c2, c3: ChambrePT;
        end;
var
  Chambres: array[1..4] of ChambrePT;
  erreur: boolean;
  NotreChambre: integer;

procedure InitialisationDeChambres;
  var
    i: integer;
  begin
    for i := 1 to 4 do
      new(Chambres[i]);
    (*quand on met un ^ cela veut dire que l'on travaille avec une adresse valeur, sinon
     avec les valeurs adresses*)
    Chambres[1]^.Nom := 'salle a manger';
    Chambres[1]^.c1 := Chambres[2];
    Chambres[1]^.c2 := Chambres[3];
    Chambres[1]^.c3 := nil;

    Chambres[2]^.Nom := 'cuisine';
    Chambres[2]^.c1 := Chambres[1];
    Chambres[2]^.c2 := Chambres[2];
    Chambres[2]^.c3 := Chambres[4];

    Chambres[3]^.Nom := 'salle de jeux';
    Chambres[3]^.c1 := Chambres[1];
    Chambres[3]^.c2 := Chambres[2];
    Chambres[3]^.c3 := Chambres[3];

    Chambres[4]^.Nom := 'bibliotheque';
    Chambres[4]^.c1 := Chambres[1];
    Chambres[4]^.c2 := Chambres[3];
    Chambres[4]^.c3 := Chambres[4];
  end;

begin
  InitialisationDeChambres;
  erreur := false;
  NotreChambre := 1;
  repeat
    writeln ('Dans quelle chambre voulez-vous aller?');
    writeln ('Vous etes dans la chambre: ', Chambres[NotreChambre]^.Nom);
    writeln ('Vous pouvez aller:');
    writeln('1 - ',Chambres[NotreChambre]^.c1^.Nom);
    writeln('2 - ',Chambres[NotreChambre]^.c2^.Nom);
    writeln('3 - ',Chambres[NotreChambre]^.c3^.Nom);
    readln(NotreChambre);
    if NotreChambre = 0 then erreur := true;
  until erreur;
  writeln('Fin du programme.');
end.
```

tri – ma version

```
program tri;

const
  MAXSIZE = 10000;

type
  StringT = varying[32] of char;
  PersonT = record
          LastName: StringT;
          FirstName: StringT;
          Day: integer;
          Month: integer;
          Year: integer;
        end;

var
  Persons: array[1..MAXSIZE] of PersonT;
  NPersons: integer;

procedure ReadPersons(fichier: StringT);
 var
   f: text;
 begin
   reset(f,fichier);
   NPersons := 0;
   while not eof(f) do
     begin
       NPersons := NPersons + 1;
       readln(f, Persons[NPersons].LastName);
       readln(f, Persons[NPersons].FirstName);
       readln(f, Persons[NPersons].Day);
       readln(f, Persons[NPersons].Month);
       readln(f, Persons[NPersons].Year);
     end;
   close(f);
end;

procedure SortPersons;
 var
   i,j: integer;
   temp: PersonT;
 begin
   for i := NPersons-1 downto 1 do
     begin
       for j := 1 to i do
         begin
           if (Persons[j].LastName < Persons[j + 1].LastName) then
             begin
               temp := Persons[j];
               Persons[j] := Persons[j + 1];
               Persons[j + 1] := temp;
             end;
         end;
     end;
end;

procedure DisplayPersons;
 var
   i: integer;
 begin
   for i := 1 to NPersons do
     begin
       writeln(Persons[i].LastName);
       writeln(Persons[i].FirstName);
       writeln(Persons[i].Day);
       writeln(Persons[i].Month);
       writeln(Persons[i].Year);
       writeln;
     end;
end;

begin
  ReadPersons('noms.txt');
  SortPersons;
  DisplayPersons;
end.
```

tri : pointeurs

```
program tri2;

const
  MAXSIZE = 10000;

type
  StringT = varying[32] of char;
  PersonT = record
          LastName: StringT;
          FirstName: StringT;
          Day: integer;
          Month: integer;
          Year: integer;
        end;
  PersonPT = ^PersonT;

var
  AddrPersons: array[1..MAXSIZE] of PersonPT;
  NPersons: integer;

procedure ReadPersons(fichier: StringT);
 var
   f: text;
 begin
   reset(f,fichier);
   NPersons := 0;
   while not eof(f) do
     begin
       NPersons := NPersons + 1;
       new(AddrPersons[NPersons]);
       readln(f, AddrPersons[NPersons]^.LastName);
       readln(f, AddrPersons[NPersons]^.FirstName);
       readln(f, AddrPersons[NPersons]^.Day);
       readln(f, AddrPersons[NPersons]^.Month);
       readln(f, AddrPersons[NPersons]^.Year);
     end;
   close(f);
end;

procedure SortPersons;
 var
   i,j: integer;
   temp: PersonPT;
 begin
   for i := NPersons-1 downto 1 do
     begin
       for j := 1 to i do
         begin
           if (AddrPersons[j]^.LastName > AddrPersons[j + 1]^.LastName) then
             begin
               temp := AddrPersons[j];
               AddrPersons[j] := AddrPersons[j + 1];
               AddrPersons[j + 1] := temp;
             end;
         end;
     end;
end;

procedure DisplayPersons;
 var
   i: integer;
 begin
   for i := 1 to NPersons do
     begin
       writeln(AddrPersons[i]^.LastName);
       writeln(AddrPersons[i]^.FirstName);
       writeln(AddrPersons[i]^.Day);
       writeln(AddrPersons[i]^.Month);
       writeln(AddrPersons[i]^.Year);
       writeln;
       dispose(AddrPersons[i]);
     end;
end;

begin
  ReadPersons('noms.txt');
  SortPersons;
  DisplayPersons;
end.
```

Address Book 1

```
program AddressBook ;

const MAXSIZE = 1000 ;

type
 StringT = varying[32] of char ;
 PersonT = record
          LastName : StringT ;
          FirstName : StringT ;
          BirthDay : integer ;
          BirthMonth : integer ;
          BirthYear : integer ;
         end ;

var
 Addr : array[1..MAXSIZE] of PersonT ;
 NAddr : integer ;

procedure ReadAddresses (fn : StringT) ;
var
 af : text ;
begin
 NAddr := 0 ; reset (af, fn) ;
 while (not eof (af)) do
  begin
   NAddr := NAddr + 1 ;
   readln (af, Addr[NAddr].LastName) ;
   readln (af, Addr[NAddr].FirstName) ;
   readln (af, Addr[NAddr].BirthDay) ;
   readln (af, Addr[NAddr].BirthMonth) ;
   readln (af, Addr[NAddr].BirthYear) ;
   readln (af) ;
  end ;
end ;

procedure DisplayAddresses ;
var
 i : integer ;
begin
 for i := 1 to NAddr do
  begin
   writeln(Addr[i].LastName:16,
        Addr[i].FirstName:16, ' '
       , Addr[i].BirthDay:2, '-'
       , Addr[i].BirthMonth:2, '-'
       , Addr[i].BirthYear:4 ) ;
 end ;
end ;

procedure SortAddresses ;
var
 tmp : PersonT ;
 i, j : integer ;
begin
 for i := 1 to NAddr do
  begin
   writeln ('Before step ', i:1) ;
   DisplayAddresses ;
   for j := 1 to NAddr-i do
    begin
     if ( Addr[j].LastName > Addr[j+1].LastName ) then
      begin
       tmp := Addr[j] ;
       Addr[j] := Addr[j+1] ;
       Addr[j+1] := tmp ;
      end ;
    end ;
  end ;
end ;

begin
 ReadAddresses ('ab.txt') ;
 SortAddresses ;
 writeln ('Final') ;
 DisplayAddresses ;
end.
```

Addressbook – Pointeurs

```
program AddressBook ;

type
 StringT = varying[32] of char ;
 PersonPT = ^PersonT ;
 PersonT = record
          LastName : StringT ;
          FirstName : StringT ;
          BirthDay : integer ;
          BirthMonth : integer ;
          BirthYear : integer ;
         end ;

var
 Addr : array[1..1000] of PersonPT ;
 NAddr : integer ;

procedure ReadAddress ( var af : text) ;
var
 addressP : PersonPT ;
begin
 new (addressP) ;
 readln (af, addressP^.LastName) ;
 readln (af, addressP^.FirstName) ;
 readln (af, addressP^.BirthDay) ;
 readln (af, addressP^.BirthMonth) ;
 readln (af, addressP^.BirthYear) ;
 readln (af) ;
 NAddr := NAddr + 1 ;
 Addr[NAddr] := addressP ;
end ;

procedure ReadAddresses (fn : StringT) ;
var
 af : text ;
 i : integer ;
begin
 for i := 1 to 1000 do
  begin
   Addr[i] := nil ;
  end ;
 NAddr := 0 ;
 reset (af, fn) ;
 while not eof (af) do
  begin
   ReadAddress (af) ;
  end ;
 close (af) ;
end ;

procedure DisplayAddresses ;
var
 personP : PersonPT ;
 i : integer ;
begin
 for i := 1 to NAddr do
  begin
   personP := Addr[i] ;
   writeln ( personP^.LastName:16
        , personP^.FirstName:16, '  '
        , personP^.BirthDay:2, '-'
        , personP^.BirthMonth:2, '-'
        , personP^.BirthYear:4 ) ;
  end ;
end ;

procedure SortAddresses ;
var
 tmpP : PersonPT ;
 i, j : integer ;
begin
 for i := 1 to NAddr do
  begin
   writeln ('Before step ', i:1) ;
   DisplayAddresses ;
   for j := 1 to NAddr-i do
    begin
     if ( Addr[j]^.LastName > Addr[j+1]^.LastName ) then
      begin
       tmpP := Addr[j] ;
       Addr[j] := Addr[j+1] ;
       Addr[j+1] := tmpP ;
      end ;
    end ;
  end ;
end ;

procedure ClearAddresses ;
var
 i : integer ;
begin
 for i := 1 to NAddr do
  begin
   dispose (Addr[i]) ;
  end ;
end ;

begin
 ReadAddresses ('ab.txt') ;
 SortAddresses ;
 writeln ('Final') ;
 DisplayAddresses ;
 ClearAddresses ;
end.
```

```
program AddressBook ;

type
 StringT = varying[32] of char ;
 PersonPT = ^PersonT ;
 PersonT = record
          NextP : PersonPT ;
          LastName : StringT ;
          FirstName : StringT ;
          BirthDay : integer ;
          BirthMonth : integer ;
          BirthYear : integer ;
         end ;

var
 AddrP : PersonPT ;

procedure ReadAddress ( var af : text) ;
var
 pP : PersonPT ;
begin
 new (pP) ;
 readln (af, pP^.LastName) ;
 readln (af, pP^.FirstName) ;
 readln (af, pP^.BirthDay) ;
 readln (af, pP^.BirthMonth) ;
 readln (af, pP^.BirthYear) ;
 readln (af) ;
 pP^.NextP := AddrP^.NextP ;
 AddrP^.NextP := pP ;
end ;

procedure ReadAddresses (fn : StringT) ;
var
 af : text ;
begin
 new (AddrP) ;
 AddrP^.NextP := nil ;
 reset (af, fn) ;
 while not eof (af) do
  begin
   ReadAddress (af) ;
  end ;
 close (af) ;
end ;

procedure DisplayAddresses ;
var
 pP : PersonPT ;
begin
 pP := AddrP^.NextP ;
 while (pP <> nil) do
  begin
   writeln ( pP^.LastName:16,
        pP^.FirstName:16, ' '
       ,pP^.BirthDay:2, '-'
       ,pP^.BirthMonth:2, '-'
       ,pP^.BirthYear:4 ) ;
   flush ;
   pP := pP^.NextP ;
  end ;
end ;

procedure SortAddresses ;
var
 fromP : PersonPT ;
 i : integer ;
 personP, prevP : PersonPT ;
 maxP, maxPrevP : PersonPT ;
begin
 fromP := AddrP ; i := 0 ;
 while (fromP^.NextP <> nil) do
  begin
   writeln ('Step ', i:1) ;
   DisplayAddresses ;
   maxPrevP := fromP ;
   maxP := fromP^.NextP ;
   prevP := maxPrevP ;
   personP := maxP ;
   while (personP <> nil) do
    begin
     if ( personP^.LastName > maxP^.LastName) then
      begin
       maxP := personP ;
       maxPrevP := prevP ;
      end ;
     prevP := personP ;
     personP := personP^.NextP ;
    end ;
   if (fromP = AddrP) then
```

```
    begin
      fromP := maxP ;
    end ;
  (* remove maxP from the list *)
   maxPrevP^.NextP := maxP^.NextP ;
  (* put maxP at the beginning *)
   maxP^.NextP := AddrP^.NextP ;
   AddrP^.NextP := maxP ;
   i := i + 1 ;
 end ;
end ;

begin
 ReadAddresses ('ab.txt') ;
 SortAddresses ;
 writeln ('Final') ;
 DisplayAddresses ;
end.
```

```
program AddressBook ;

type
 StringT = varying[32] of char ;
 PersonPT = ^PersonT ;
 PersonT = record
         PrevP : PersonPT ;
         NextP : PersonPT ;
         LastName : StringT ;
         FirstName : StringT ;
         BirthDay : integer ;
         BirthMonth : integer ;
         BirthYear : integer ;
        end ;

var (* The address book *)
 AddrP : PersonPT ;

procedure InitializeAddresses ;
begin
 new (AddrP) ;
 AddrP^.NextP := AddrP ;
 AddrP^.PrevP := AddrP ;
end ;

procedure AppendPerson ( var bookP, personP : PersonPT) ;
var
 prevP : PersonPT ;
begin
 prevP := bookP^.PrevP ;
 personP^.NextP := bookP ;
 prevP^.NextP := personP ;
 personP^.PrevP := prevP ;
 bookP^.PrevP := personP ;
end ;

procedure RemovePerson ( var personP : PersonPT ) ;
var
 prevP, nextP : PersonPT ;
begin
 prevP := personP^.PrevP ;
 nextP := personP^.NextP ;
 prevP^.NextP := nextP ;
 nextP^.PrevP := prevP ;
end ;

procedure ReadAddress ( var af : text ) ;
var
 pP : PersonPT ;
begin
 new (pP) ;
 readln (af, pP^.LastName) ;
 readln (af, pP^.FirstName) ;
 readln (af, pP^.BirthDay) ;
 readln (af, pP^.BirthMonth) ;
 readln (af, pP^.BirthYear) ;
 readln (af) ;
 AppendPerson (AddrP, pP) ;
end ;

procedure ReadAddresses (fn : StringT) ;
var
 af : text ;
begin
 reset (af, fn) ;
 while not eof (af) do
  begin
   ReadAddress (af) ;
  end ;
 close (af) ;
end ;

procedure DisplayAddresses ;
var
 pP : PersonPT ;
begin
 pP := AddrP^.NextP ;
 while (pP <> AddrP) do
  begin
   writeln ( pP^.LastName:16,
        pP^.FirstName:16, '   ',
        pP^.BirthDay:2, '-',
        pP^.BirthMonth:2, '-',
        pP^.BirthYear:4) ;
   flush ;
   pP := pP^.NextP ;
  end ;
end ;

procedure SortAddresses ;
var
 minP : PersonPT ;
 i : integer ;
 personP : PersonPT ;
 firstP, lastP : PersonPT ;
begin
 i := 1 ;
 firstP := AddrP^.NextP ;
 lastP := AddrP ;
 while (lastP^.PrevP <> AddrP) do
  begin
   writeln ('Before step ', i:1) ;
   DisplayAddresses ;
   personP := AddrP^.NextP ;
   minP := personP ;
   while (personP <> lastP) do
    begin
     if ( personP^.LastName < minP^.LastName) then
      begin
       minP := personP ;
      end ;
     personP := personP^.NextP ;
    end ;
   RemovePerson (minP) ;
   AppendPerson (AddrP, minP) ;
   if (lastP = AddrP) then
    begin
     lastP := minP ;
    end ;
   i := i + 1 ;
  end ;
end ;

begin
 InitializeAddresses ;
 ReadAddresses ('ab.txt') ;
 SortAddresses ;
 writeln ('Final') ;
 DisplayAddresses ;
end.
```

Bridge

```
program Bridge ;

type
 NodeT = record
  Index : integer ;
  PositionX, PositionY : real ;
  ForceX, ForceY: integer ;
 end ;
 BarT = record
  Index : integer ;
  FromNode, ToNode : integer ;
 end ;

const
 NNODES = 5 ;
 NBARS = 2 * NNODES - 3 ;

var
 Nodes : array[1..NNODES] of NodeT ;
 Bars : array[1..NBARS] of BarT ;

begin
 writeln ('hello') ;
end.
```

```
(* compiler avec BGGraphs.p *)
program Bridge ;
#include "Graphics.h"
#include "BGGraphs.h"

type
 NodeT = record
  Index : integer ;
  PositionX, PositionY : real ;
  ForceX, ForceY : real ;
 end ;
 BarT = record
  Index : integer ;
  FromNode, ToNode : integer ;
 end ;

const
 NNODES = 5 ;
 NBARS = 2 * NNODES - 3 ;

var
 Nodes : array[1..NNODES] of NodeT ;
 Bars : array[1..NBARS] of BarT ;

procedure InitializeBridge ;
begin
 Nodes[1].Index := 1 ;
 Nodes[1].PositionX := 0.0 ;
 Nodes[1].PositionY := 0.0 ;
 Nodes[1].ForceX := 0.0 ;
 Nodes[1].ForceY := 1000.0 ;
 Nodes[2].Index := 2 ;
 Nodes[2].PositionX := 2.0 ;
 Nodes[2].PositionY := 3.0 ;
 Nodes[2].ForceX := 0.0 ;
 Nodes[2].ForceY := 0.0 ;
 Nodes[3].Index := 3 ;
 Nodes[3].PositionX := 5.0 ;
 Nodes[3].PositionY := 0.0 ;
 Nodes[3].ForceX := 0.0 ;
 Nodes[3].ForceY := -2000.0 ;
 Nodes[4].Index := 4 ;
 Nodes[4].PositionX := 8.0 ;
 Nodes[4].PositionY := 3.0 ;
 Nodes[4].ForceX := 0.0 ;
 Nodes[4].ForceY := 0.0 ;
 Nodes[5].Index := 5 ;
 Nodes[5].PositionX := 10.0 ;
 Nodes[5].PositionY := 0.0 ;
 Nodes[5].ForceX := 0.0 ;
 Nodes[5].ForceY := 1000.0 ;
 Bars[1].Index := 1 ;
 Bars[1].FromNode := 1 ;
 Bars[1].ToNode := 2 ;
 Bars[2].Index := 2 ;
 Bars[2].FromNode := 1 ;
 Bars[2].ToNode := 3 ;
 Bars[3].Index := 3 ;
 Bars[3].FromNode := 2 ;
 Bars[3].ToNode := 3 ;
 Bars[4].Index := 4 ;
 Bars[4].FromNode := 2 ;
 Bars[4].ToNode := 4 ;
 Bars[5].Index := 5 ;
 Bars[5].FromNode := 3 ;
 Bars[5].ToNode := 4 ;
 Bars[6].Index := 6 ;
 Bars[6].FromNode := 4 ;
 Bars[6].ToNode := 5 ;
 Bars[7].Index := 7 ;
 Bars[7].FromNode := 3 ;
 Bars[7].ToNode := 5 ;
end ;

procedure DisplayBridge ;
var
 i : integer ;
 fromNode, toNode : integer ;
 fromX, fromY, toX, toY : real ;
begin
 PenSize (6) ;
 for i := 1 to NBARS do
  begin
   fromNode := Bars[i].FromNode ;
   toNode := Bars[i].ToNode ;
   fromX := Nodes[fromNode].PositionX ;
   fromY := Nodes[fromNode].PositionY ;
   toX := Nodes[toNode].PositionX ;
   toY := Nodes[toNode].PositionY ;
   DrawGraphLine (fromX, fromY, toX, toY) ;
  end ;
 for i := 1 to NNODES do
  begin
   DrawGraphPoint (Nodes[i].PositionX, Nodes[i].PositionY, 10) ;
  end ;
end ;

begin
 SetGraphSize (5, 0, -5, 10) ;
 InitializeBridge ;
 DisplayBridge ;
 readln ;
end.
```

```
(* compiler avec BGGraphs.p *)
(* copier l'executable et le fichier bridge.txt    dans votre repertoire avant
d'executer *)
program Bridge ;
#include "Graphics.h"
#include "BGGraphs.h"

type
 NodeT = record
  Index : integer ;
  PositionX, PositionY : real ;
  ForceX, ForceY : real ;
 end ;
 BarT = record
  Index : integer ;
  FromNode, ToNode : integer ;
 end ;

const
 MAX_NNODES = 20 ;
 MAX_NBARS = 2 * MAX_NNODES - 3 ;

var
 nnodes, nbars : integer ;
 Nodes : array[1..MAX_NNODES] of NodeT ;
 Bars : array[1..MAX_NBARS] of BarT ;

procedure InitializeBridge ;
var
 BridgeFileName : varying[32] of char ;
 BridgeFile : text ;
 i, index : integer ;

begin
 BridgeFileName := 'bridge.txt' ;
 reset (BridgeFile, BridgeFileName) ;
 readln (BridgeFile, nnodes) ;
 nbars := 2*nnodes - 3 ;
 for i := 1 to nnodes do
  begin
   read ( BridgeFile, Nodes[i].PositionX, Nodes[i].PositionY) ;
   readln ( BridgeFile, Nodes[i].ForceX, Nodes[i].ForceY) ;
  end ;
 for i := 1 to nbars do
  begin
   readln (BridgeFile, index, Bars[i].FromNode,Bars[i].ToNode);
  end ;
 close (BridgeFile) ;
end ;
```

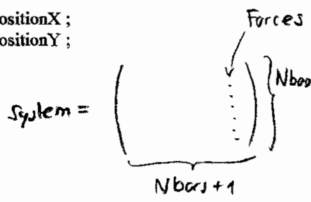
```
procedure DisplayBridge ;
var
  i : integer ;
  fromNode, toNode : integer ;
  fromX, fromY, toX, toY : real ;

begin
  PenSize (6) ;
  for i := 1 to nbars do
    begin
      fromNode := Bars[i].FromNode ;
      toNode := Bars[i].ToNode ;
      fromX := Nodes[fromNode].PositionX ;
      fromY := Nodes[fromNode].PositionY ;
      toX := Nodes[toNode].PositionX ;
      toY := Nodes[toNode].PositionY ;
      DrawGraphLine (fromX, fromY, toX, toY) ;
    end ;
  for i := 1 to nnodes do
    begin
      DrawGraphPoint (Nodes[i].PositionX, Nodes[i].PositionY, 10) ;
    end ;
end ;

begin
  SetGraphSize (5, 0, -5, 10) ;
  InitializeBridge ;
  DisplayBridge ;
  readln ;
end.


(* compiler avec BGGraphs.p and BGMatrices.p *)
(* copier l'executable et le fichier bridge.txt    dans votre repertoire avant
d'executer *)

program Bridge ;
#include "Graphics.h"
#include "BGGraphs.h"
#include "BGMatrices.h"

type
  NodeT = record
    Index : integer ;
    PositionX, PositionY : real ;
    ForceX, ForceY : real ;
  end ;
  BarT = record
    Index : integer ;
    FromNode, ToNode : integer ;
  end ;

const
  NNODES = 5 ;
  NBARS = 2 * NNODES - 3 ;

var
  nnodes, nbars : integer ;
  Nodes : array[1..NNODES] of NodeT ;
  Bars : array[1..NBARS] of BarT ;
  System : array[1..NBARS,1..NBARS+1] of real ;
  InternalForces : array[1..NBARS] of real ;

procedure InitializeBridge ;
var
  BridgeFileName : varying[32] of char ;
  BridgeFile : text ;
  i, index : integer ;
begin
  BridgeFileName := 'bridge.txt' ;
  reset (BridgeFile, BridgeFileName) ;
  readln (BridgeFile, nnodes) ;
  if (nnodes <> NNODES) then
    begin
      writeln ('program and bridge description files incompatible') ;
      pcexit (1) ;
    end ;
  nbars := 2*nnodes - 3 ;
  for i := 1 to nnodes do
    begin
      readln ( BridgeFile, Nodes[i].PositionX, Nodes[i].PositionY,
           Nodes[i].ForceX, Nodes[i].ForceY) ;
    end ;
  for i := 1 to nbars do
    begin
      readln (BridgeFile, index, Bars[i].FromNode,Bars[i].ToNode);
    end ;
  close (BridgeFile) ;
end ;
```

→ pour mettre les solutions des éq. (the "+1" in NBARS+1)

i = colonne

```
procedure InitializeSystem ;
var
  i, j : integer ;
  fromNode, toNode : integer ;
  lengthX, lengthY : real ;
  length : real ;
begin
  for i := 1 to NBARS do
    begin
      fromNode := Bars[i].FromNode ;
      toNode := Bars[i].ToNode ;
      lengthX := Nodes[fromNode].PositionX - Nodes[toNode].PositionX ;
      lengthY := Nodes[fromNode].PositionY - Nodes[toNode].PositionY ;
      length := sqrt (lengthX*lengthX + lengthY*lengthY) ;
      if (2*fromNode-1 <= NBARS) then
        System[2*fromNode-1,i] := lengthX / length ;
      if (2*fromNode <= NBARS) then
        System[2*fromNode,i] := lengthY / length ;
      if (2*toNode-1 <= NBARS) then
        System[2*toNode-1,i] := -lengthX / length ;
      if (2*toNode <= NBARS) then
        System[2*toNode,i] := -lengthY / length ;
    end ;
  j := 1 ;
  for i := 1 to NBARS do
    begin
      if (i mod 2 = 1) then
        begin
          System[i,NBARS+1] := Nodes[j].ForceX ;
        end
      else
        begin
          System[i,NBARS+1] := Nodes[j].ForceY ;
          j := j + 1 ;
        end ;
    end ;
end ;
```

→ pour chaque barre, on calcule sa longueur sur Ox, Oy et sa longueur



→ car on a des forces sur X et Y : sur la ligne 1 : force X, 2 : force Y, 3 : force X, 4 : force Y etc. — en 3D : i mod 3 car {x,y,z}

dernière colonne de la matrice : celle contenant les "solutions" : External Forces

```
procedure DisplayBridge ;
var
  i : integer ;
  fromNode, toNode : integer ;
  fromX, fromY, toX, toY : real ;
  pen : integer ;
begin
  for i := 1 to nbars do
    begin
      pen := round (abs (InternalForces[i]) / 100) ;
      if (pen < 2) then pen := 2 ;
        PenSize (pen) ;
      if (InternalForces[i] < 0) then
        begin
          SetColor (5) ;
        end
      else
        begin
          SetColor (0) ;
        end ;
      fromNode := Bars[i].FromNode ;
      toNode := Bars[i].ToNode ;
      fromX := Nodes[fromNode].PositionX ;
      fromY := Nodes[fromNode].PositionY ;
      toX := Nodes[toNode].PositionX ;
      toY := Nodes[toNode].PositionY ;
      DrawGraphLine (fromX, fromY, toX, toY) ;
    end ;
  for i := 1 to nnodes do
    begin
      DrawGraphPoint (Nodes[i].PositionX, Nodes[i].PositionY, 10) ;
    end ;
end ;

begin
  SetGraphSize (12, 0, 0, 12) ;
  InitializeBridge ;
  InitializeSystem ;
  SolveSystem2 (System, InternalForces) ;
  DisplayBridge ;
  writeln ('Internal forces') ;
  writeVector (InternalForces) ;
  readln ;
end.


(* compiler avec BGGraphs.p et BGMatrices.p *)
(* copier l'executable et le fichier bridge.txt dans votre repertoire avantt d'executer *)

program Bridge ;
```

```
#include "Graphics.h"
#include "BGGraphs.h"
#include "BGMatrices.h"

type
 NodeT = record
  Index : integer ;
  PositionX, PositionY : real ;
  ForceX, ForceY : real ;
 end ;
 BarT = record
  Index : integer ;
  FromNodeP, ToNodeP : ^NodeT ;
 end ;

const
 NNODES = 5 ;
 NBARS = 2 * NNODES - 3 ;

var
 nnodes, nbars : integer ;
 Nodes : array[1..NNODES] of NodeT ;
 Bars : array[1..NBARS] of BarT ;
 System : array[1..NBARS,1..NBARS+1] of real ;
 InternalForces : array[1..NBARS] of real ;
 procedure InitializeBridge ;
var
 BridgeFileName : varying[32] of char ;
 BridgeFile : text ;
 i, index : integer ;
 fromNode, toNode : integer ;

begin
 BridgeFileName := 'bridge.txt' ;
 reset (BridgeFile, BridgeFileName) ;
 readln (BridgeFile, nnodes) ;
 if (nnodes <> NNODES) then
begin
 writeln ('program and bridge description files incompatible') ;
 pcexit (1) ;
end ;
 nbars := 2*nnodes - 3 ;
 for i := 1 to nnodes do
begin
 readln ( BridgeFile, Nodes[i].PositionX, Nodes[i].PositionY
      , Nodes[i].ForceX, Nodes[i].ForceY) ;
 Nodes[i].Index := i ;
end ;
 for i := 1 to nbars do
  begin
   readln (BridgeFile, index, fromNode, toNode);
   Bars[i].FromNodeP := addr (Nodes[fromNode]) ;
   Bars[i].ToNodeP := addr (Nodes[toNode]) ;
   Bars[i].Index := index ;
  end ;
 close (BridgeFile) ;
end ;

procedure InitializeSystem ;
var
 i, j : integer ;
 fromNodeP, toNodeP : ^NodeT ;
 lengthX, lengthY : real ;
 length : real ;
begin
 for i := 1 to NBARS do
  begin
   fromNodeP := Bars[i].FromNodeP ;
   toNodeP := Bars[i].ToNodeP ;
   lengthX := fromNodeP^.PositionX - toNodeP^.PositionX ;
   lengthY := fromNodeP^.PositionY - toNodeP^.PositionY ;
   length := sqrt (lengthX*lengthX + lengthY*lengthY) ;
   if (2*fromNodeP^.Index-1 <= NBARS) then
    System[2*fromNodeP^.Index-1,i] := lengthX / length ;
   if (2*fromNodeP^.Index <= NBARS) then
    System[2*fromNodeP^.Index,i] := lengthY / length ;
   if (2*toNodeP^.Index-1 <= NBARS) then
    System[2*toNodeP^.Index-1,i] := -lengthX / length ;
   if (2*toNodeP^.Index <= NBARS) then
    System[2*toNodeP^.Index,i] := -lengthY / length ;
  end ;
 j := 1 ;
 for i := 1 to NBARS do
  begin
   if (i mod 2 = 1) then
    begin
     System[i,NBARS+1] := Nodes[j].ForceX ;
    end
   else
    begin
     System[i,NBARS+1] := Nodes[j].ForceY ;
     j := j + 1 ;
    end ;
  end ;
end ;

procedure DisplayBridge ;
var
 i : integer ;
 fromNodeP, toNodeP : ^NodeT ;
 fromX, fromY, toX, toY : real ;
 pen : integer ;
begin
 for i := 1 to nbars do
  begin
   pen := round (abs (InternalForces[i]) / 100) ;
   if (pen < 2) then
    pen := 2 ;
   PenSize (pen) ;
   if (InternalForces[i] < 0) then
    begin
     SetColor (5) ;
    end
   else
    begin
     SetColor (0) ;
    end ;
   fromNodeP := Bars[i].FromNodeP ;
   toNodeP := Bars[i].ToNodeP ;
   fromX := fromNodeP^.PositionX ;
   fromY := fromNodeP^.PositionY ;
   toX := toNodeP^.PositionX ;
   toY := toNodeP^.PositionY ;
   DrawGraphLine (fromX, fromY, toX, toY) ;
  end ;
 for i := 1 to nnodes do
  begin
   DrawGraphPoint (Nodes[i].PositionX, Nodes[i].PositionY, 10) ;
  end ;
end ;

begin
 SetGraphSize (12, 0, 0, 12) ;
 InitializeBridge ;
 InitializeSystem ;
 writeln ('System = ') ;
 bgWriteMatrix (System) ;
 bgSolveSystem (System, InternalForces) ;
 DisplayBridge ;
 writeln ('Internal forces = ') ;
 bgWriteVector (InternalForces) ;
 readln ;
end.
```

Affichage – Rotation 3D

{Affichage 3d}

```
(* compiler ce fichier avec le fichier BGGraphs.p *)

program FourD ;
#include "Graphics.h"
#include "BGGraphs.h"

procedure DisplayAxes ;
begin
 PenSize (2) ;
 Draw3DLine (-5.0,  0.0,  0.0, 5.0, 0.0, 0.0) ;
 Draw3DLine ( 0.0, -5.0,  0.0, 0.0, 5.0, 0.0) ;
 Draw3DLine ( 0.0,  0.0, -5.0, 0.0, 0.0, 5.0) ;
 PenSize (1) ;
end ;

procedure Draw3DLineAndTraces  ( toX, toY, toZ : real ) ;
begin
 PenSize (2) ;
 Draw3DLine (   0,   0,   0, toX, toY, toZ) ;
 Draw3DLine ( toX,   0,   0, toX, toY,   0) ;
 Draw3DLine (   0, toY,   0, toX, toY,   0) ;
 Draw3DLine ( toX, toY,   0, toX, toY, toZ) ;
 Draw3DLine (   0,   0, toZ, toX, toY, toZ) ;
 PenSize (1) ;
end ;

var
 i : integer ;

begin
 SetGraphSize (5,-5,-5,5) ;
 DisplayAxes ;
 Draw3DLineAndTraces (1, 2, 3) ;
 readln ;
end.
```

```
program FourD ;
#include "Graphics.h"
#include "BGGraphs.h"
#include "BGMatrices.h"
(* compiler ce fichier avec les fichiers BGGraphs.p et BGMatrices.p *)

procedure DisplayAxes ;
begin
 PenSize (2) ;
 Draw3DLine (-5.0,  0.0,  0.0, 5.0, 0.0, 0.0) ;
 Draw3DLine ( 0.0, -5.0,  0.0, 0.0, 5.0, 0.0) ;
 Draw3DLine ( 0.0,  0.0, -5.0, 0.0, 0.0, 5.0) ;
 PenSize (1) ;
end ;

procedure Draw3DLineAndTraces  ( toX, toY, toZ : real ) ;
begin
 PenSize (2) ;
 Draw3DLine (   0,   0,   0, toX, toY, toZ) ;
 Draw3DLine ( toX,   0,   0, toX, toY,   0) ;
 Draw3DLine (   0, toY,   0, toX, toY,   0) ;
 Draw3DLine ( toX, toY,   0, toX, toY, toZ) ;
 Draw3DLine (   0,   0, toZ, toX, toY, toZ) ;
 PenSize (1) ;
end ;

var
 i : integer ;
 x : Vector3T ;              (* vecteur a faire tourner *)
 r : Vector3T ;              (* axe de rotation *)
 x1, x2, x3, x4, x5 : Vector3T ; (* valeurs intermediaires *)
 alpha, beta, gamma : real ;
 mrx, mry1, mry2, mrz1, mrz2 : Matrix33T ;   --> type défini dans matrices.p

begin
 SetGraphSize (5,-5,-5,5) ;

 (* initializer le vecteur a faire tourner *)
 x[1] := 1.0 ; x[2] := 2.0 ; x[3] := 3.0 ;

 (* initializer l'axe de rotation *)
 r[1] := 1.0 ; r[2] := 2.0 ; r[3] := 0.0 ;
 CartesianToSpherical (r[1], r[2], r[3], alpha, beta) ;

 (* initializer les matrices de rotation *)
 InitZRotationMatrix (mrz1, -alpha) ;
 InitYRotationMatrix (mry1, -beta) ;
 InitYRotationMatrix (mry2, beta) ;
 InitZRotationMatrix (mrz2, alpha) ;

 for i := 0 to 628 do
  begin
   Delay (40) ;
   SuspendRefresh ;
   SetColor (8) ;
   FillRectangle (0,0,400,400) ;
   SetColor (0) ;
   gamma := i/100 ;
   InitXRotationMatrix (mrx, gamma) ;
   MatVectMult (x,  mrz1, x1) ;
   MatVectMult (x1, mry1, x2) ;
   MatVectMult (x2, mrx,  x3) ;
   MatVectMult (x3, mry2, x4) ;
   MatVectMult (x4, mrz2, x5) ;
   DisplayAxes ;
   Draw3DLine (0, 0, 0, r[1], r[2], r[3]) ;
   Draw3DLineAndTraces (x5[1], x5[2], x5[3]) ;
   ResumeRefresh ;
  end ;
 readln ;
end.
```

```
program Life ;
#include "Graphics.h"
  const
    CELLSIZE = 5 ; (* pixels *)
    GRIDSIZE = 80 ; (* cells *)
  type
    GridT = array [0..GRIDSIZE-1,0..GRIDSIZE-1] of boolean ;
  var
    Grid : array[0..1] of GridT ;
    Prev, Next : integer ;
    Index : integer ;
    Time1, Time2, Time3 : integer ;

  procedure InitializeGrids ;
    var
      x : real ;
      i, j : integer ;
    begin
      x := 2.0 ;
      for i := 0 to GRIDSIZE - 1 do
        begin
               for j := 0 to GRIDSIZE - 1 do
                 begin
                   Grid[0][i,j] := false ;
                   if (random (x) >= 0.5) then
                     begin
                            Grid[1][i,j] := true ;
                     end
                   else
                     begin
                            Grid[1][i,j] := false ;
                     end ;
                 end ;
        end ;
    end ;

  procedure NextCell ( var prev : GridT
                     ; var next : GridT
                     ; i, j : integer
                     ) ;
    var
      top, left, bottom, right : integer ;
      sum : integer ;
    begin
      top := j-1 ;
      if (top < 0) then
        begin
               top := GRIDSIZE - 1 ;
        end ;
      left := i-1 ;
      if (left < 0) then
        begin
               left := GRIDSIZE - 1 ;
        end ;
      bottom := j+1 ;
      if (bottom >= GRIDSIZE) then
        begin
               bottom := 0 ;
        end ;
```

```
  right := i+1 ;
  if (right >= GRIDSIZE) then
   begin
        right := 0 ;
   end ;
  sum := 0 ;
  if (prev[left,top] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[i,top] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[right,top] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[left,j] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[i,j] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[right,j] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[left,bottom] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[i,bottom] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if (prev[right,bottom] = true) then
   begin
        sum := sum + 1 ;
   end ;
  if ((sum = 4) or (sum = 3)) then
   begin
        next[i,j] := true ;
   end
  else
   begin
        next[i,j] := false ;
   end ;
 end ;

procedure NextGrid ( var prev : GridT ; var next : GridT) ;
 var
  i, j : integer ;
 begin
  for i := 0 to GRIDSIZE-1 do
   begin
        for j := 0 to GRIDSIZE-1 do
         begin
```

```
            NextCell (prev, next, i, j) ;
          end ;
      end ;
  end ;

procedure DrawCell (i, j : integer) ;
  var
    top, left, bottom, right : integer ;
  begin
    top := j * CELLSIZE + 1 ;
    left := i * CELLSIZE + 1 ;
    bottom := (j+1) * CELLSIZE - 1 ;
    right := (i+1) * CELLSIZE - 1 ;
    FillRectangle (top, left, bottom, right) ;
  end ;

procedure DrawGrid ( var prev, next : GridT) ;
  var
    i, j : integer ;
  begin
    for i := 0 to GRIDSIZE - 1 do
      begin
          for j := 0 to GRIDSIZE - 1 do
            begin
              if (prev[i,j] = true) then
                begin
                    if (next[i,j] = false) then
                      begin
                        SetColor (0) ;
                        DrawCell (i, j) ;
                      end ;
                  end
              else
                begin
                    if (next[i,j] = false) then
                      begin
                        SetColor (8) ;
                        DrawCell (i, j) ;
                      end ;
                end ;
            end ;
      end ;
  end ;

begin
  InitializeGrids ;
  Index := 2 ;
  while (true) do
    begin
      Prev := (Index - 1) mod 2 ;
      Next := (Index) mod 2 ;
          GetTime (Time1) ;
          NextGrid (Grid[Prev], Grid[Next]) ;
          GetTime (Time2) ;
          SuspendRefresh ;
          DrawGrid (Grid[Prev], Grid[Next]) ;
          ResumeRefresh ;
          GetTime (Time3) ;
          writeln ('Step ', Index:1) ;
          writeln ('   compute time : ', Time2 - Time1, 'msec') ;
```

```
        writeln ('   drawing time : ', Time3 - Time2, 'msec') ;
        flush ;
        Delay (100) ;
        Index := Index + 1 ;
    end ;
end.
```

```
program mandel2;
#include "Graphics.h"

var
        reelle, imaginaire: real;
        xmin, xmax,imin,imax :real;
        totx,toti: integer;
        dx, di : real;
        iteration: integer;
        i, j: integer;
        controle: boolean;

procedure calcul (var un, deux: real; itera: integer);
var
        k : integer;
        temp: real;
        test: boolean;
        re,im: real;
begin
        re := un;
        im := deux;
        controle := false;
        k := 0;
        repeat
                k := k+1;
                temp := sqr(un) -sqr(deux) + re;
                deux := 2*un*deux+im;
                un := temp;
                test := (sqr(un) + sqr(deux)) <2;
        until (k>=itera) or not (test);
        if test then
                controle := true
        else
                controle := false;
end;

procedure dessine;
begin
        if controle = true then
                SetColor(0)
        else
                SetColor(6);
        DrawLine(i,j,i,j);
end;

begin
        writeln ('Donnez: xmin, xmax, imin, imax');
        readln (xmin, xmax, imin, imax);
        writeln ('Donnez les dimensions de l''écran');
        readln (totx, toti);
        writeln ('Donnez le nombre d''itération maximum');
        readln (iteration);
        dx := (xmax-xmin) / totx;
        di := (imax-imin) / toti;
        SetWindowSize (totx,toti);
        for j := 1 to toti do
                begin
                        for i := 1 to totx do
                        begin
                                reelle := xmin+i*dx;
```

```
                    imaginaire := imin+j*di;
                    calcul(reelle, imaginaire, iteration);
                    dessine;
               end;
          end;
writeln ('Appuyez sur RETURN');
flush;
readln;
end.
```

Programmation I : Examen Final

Mécaniciens

Comment s'y prendre

Il y a 3 exercices pour lesquels vous devez écrire un programme PASCAL. Remettez le fichier source et l'exécutable dans le répertoire ~gennart/examen-final/meca/login_name. N'oubliez pas de mettre votre nom en commentaire dans chacun de vos programmes.

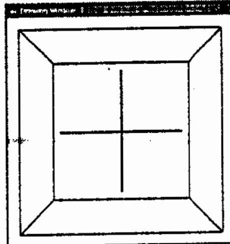
1 Projection 3-D 2-D améliorée.

La routine de projection que l'on a utilisée pour le cours est assez peu satisfaisante. Il existe de meilleures routines de projection utilisant la notion de point de fuite. La formule à utiliser pour la projection est :

$$\text{DistObsX} = 10.0$$

$$x_p = \frac{y}{\left(\frac{x}{\text{DistObsX}} + 1\right)} \qquad y_p = \frac{z}{\left(\frac{x}{\text{DistObsX}} + 1\right)}$$

On demande de remplacer dans votre module de dessin en coordonnées de graphe (normalement constitué de deux fichiers Graphs.h et Graphs.p) la routine de projection que je vous ai donné au cours par une nouvelle routine effectuant le calcul des formules ci-dessus. Pour tester votre programme, montrer les 12 arêtes d'un cube dont les 8 sommets sont : s0 = (-4,-4,-4) ; s1 = (-4,-4,4) ; s2 = (-4,4,-4) ; s3 = (-4,4,4) ; s4 = (4,-4,-4) ; s5 = (4,-4,4) ; s6 = (4,4,-4) ; s7 = (4,4,4).. Vous devriez obtenir un dessin qui a l'allure suivante. La croix au centre est composée de deux droites : (0,-3,0) jusqu'à (0,3,0), et (0,0,-3) jusqu'à (0,0,3). **(e1.p, 20 points)**



2 Rotation 3D

Avec la nouvelle routine de projection, faites tourner le cube autour de l'axe y (l'axe horizontal sur la figure). Pour cela on reprend l'exercice de rotation en 3D, et l'on fait tourner chacun des 8 sommets

du cube. On retrace alors les droites à partir des sommets tournés. Ecrivez un programme qui fait tourner le cube de 360 degrés par pas de 1 degrés. **(e2.p, 20 points)**

3 Problème du treillis en 3D

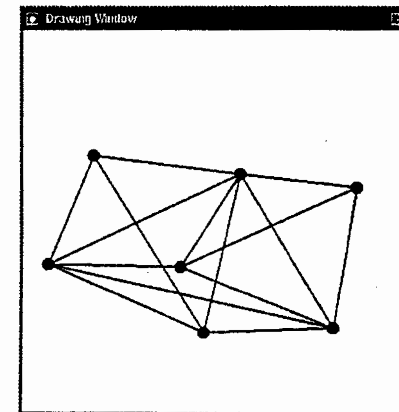
Dans le cadre du problème du treillis, on considère maintenant un treillis à 3 dimensions. Le treillis comporte toujours des noeuds et des barres, mais les noeuds ont maintenant 3 champs de position : PositionX, PositionY, et PositionZ. La relation entre le nombre de noeuds et de barres dans le treillis est : nbars = 3*nnodes - 6.

Le format du fichier de description du treillis comporte le nombre de noeuds dans le treillis, une liste de 3 réels (PositionX, positionY, PositionZ) pour chaque noeud, et une liste de trois entiers pour chaque barre : l'indice de la barre, le noeud d'origine, et le noeud d'arrivée. Par exemple, le fichier suivant décrit un treillis simple (fichier ~gennart/examen-final/br3d1.txt).

```
 7

-0.812    0.812    2.772
 0.108    2.721   -0.765
 2.721    0.108    0.765
 2.017    3.640    2.772
 2.936    5.549   -0.765
 5.549    2.936    0.765
 4.845    6.469    2.772

 1    1    2
 2    1    3
```

```
 3   2   3
 4   1   4
 5   2   4
 6   3   4
 7   2   5
 8   3   5
 9   4   5
10   3   6
11   4   6
12   5   6
13   4   7
14   5   7
15   6   7
```

On demande d'écrire une routine d'initialisation du treillis à partir d'un fichier au format expliqué ci-dessus, d'initialiser un treillis 3D avec la routine d'initialisation suivante, et de l'afficher à l'écran. Ceci devrait donner les résultats suivants : **(e3.p, 20 points)**



Cubeexam (mécaniciens 1996) (n°2) +(n°1)

```
program Cubexam ;
#include "Graphics.h"
#include "Graphs.h"
#include "Matrices.h"

const
 Dist = 8;
 largeur = 400;
 hauteur = 400;
 vitesse = 20;
 Pi = 3.1415;


var
 points: array[1..8] of Vector3T;
 pointsr: array[1..8] of Vector3T;
 angle: real ;
 matrice: Matrix33T ;
 z : integer;


procedure rot;
var
 i : integer;
begin
 for i := 1 to 8 do
  begin
   MatVectMult(points[i], matrice, pointsr[i]);
  end;
end;

procedure valeurs;
 begin
  points[1][1] := -4;
  points[1][2] := -4;
  points[1][3] := -4;
  points[2][1] := -4;
  points[2][2] := -4;
  points[2][3] :=  4;
  points[3][1] := -4;
  points[3][2] :=  4;
  points[3][3] := -4;
  points[4][1] := -4;
  points[4][2] :=  4;
  points[4][3] :=  4;
  points[5][1] :=  4;
  points[5][2] := -4;
  points[5][3] := -4;
  points[6][1] :=  4;
  points[6][2] := -4;
  points[6][3] :=  4;
  points[7][1] :=  4;
  points[7][2] :=  4;
  points[7][3] := -4;
  points[8][1] :=  4;
  points[8][2] :=  4;
  points[8][3] :=  4;
 end;

procedure dessin;
 begin
  Draw3DLine2(0, -3, 0, 0, 3, 0, Dist);
  Draw3DLine2(0, 0, -3, 0, 0, 3, Dist);
  Draw3DLine2(pointsr[6][1], pointsr[6][2], pointsr[6][3], pointsr[8][1],
        pointsr[8][2], pointsr[8][3], Dist);
  Draw3DLine2(pointsr[6][1], pointsr[6][2], pointsr[6][3], pointsr[5][1],
        pointsr[5][2], pointsr[5][3], Dist);
  Draw3DLine2(pointsr[6][1], pointsr[6][2], pointsr[6][3], pointsr[2][1],
        pointsr[2][2], pointsr[2][3], Dist);
  Draw3DLine2(pointsr[4][1], pointsr[4][2], pointsr[4][3], pointsr[2][1],
        pointsr[2][2], pointsr[2][3], Dist);
  Draw3DLine2(pointsr[4][1], pointsr[4][2], pointsr[4][3], pointsr[8][1],
        pointsr[8][2], pointsr[8][3], Dist);
  Draw3DLine2(pointsr[4][1], pointsr[4][2], pointsr[4][3], pointsr[3][1],
        pointsr[3][2], pointsr[3][3], Dist);
  Draw3DLine2(pointsr[1][1], pointsr[1][2], pointsr[1][3], pointsr[2][1],
        pointsr[2][2], pointsr[2][3], Dist);
  Draw3DLine2(pointsr[1][1], pointsr[1][2], pointsr[1][3], pointsr[5][1],
        pointsr[5][2], pointsr[5][3], Dist);
  Draw3DLine2(pointsr[1][1], pointsr[1][2], pointsr[1][3], pointsr[3][1],
        pointsr[3][2], pointsr[3][3], Dist);
  Draw3DLine2(pointsr[7][1], pointsr[7][2], pointsr[7][3], pointsr[5][1],
        pointsr[5][2], pointsr[5][3], Dist);
  Draw3DLine2(pointsr[7][1], pointsr[7][2], pointsr[7][3], pointsr[8][1],
        pointsr[8][2], pointsr[8][3], Dist);
  Draw3DLine2(pointsr[7][1], pointsr[7][2], pointsr[7][3], pointsr[3][1],
        pointsr[3][2], pointsr[3][3], Dist);
 end;

begin
 SetWindowSize(largeur, hauteur);
 SetGraphSize (12,-12,-12,12) ;
 valeurs;
 for z := 1 to 8 do
  InitializeVector(pointsr[z]);
 for z := 0 to 1000000 do
  begin
   SuspendRefresh ;
   SetColor (8) ;
   FillRectangle (0,0,largeur,hauteur) ;
   SetColor (0) ;
   angle := z * (Pi/360);
   InitYRotationMatrix(matrice, angle);
   rot;
   dessin;
   ResumeRefresh;
  end ;
 readln ;
end.
```

bridge3D 1 (n°3)

```
program bridge3D1;
#include "Graphics.h";
#include "Graphs.h";

const
 Perspective = 10;
 MAXNnodes = 7;
 MAXNbars = 3 * MAXNnodes -6;

type
 NodeT = record
         Index: integer;
         PositionX, PositionY, PositionZ: real;
         ForceX, ForceY, ForceZ: real;
        end;
 BarT = record
         Index: integer;
         FromNode, ToNode: integer;
         InternalForce: real;
        end;

var
 NNodes: integer;
 Nbars: integer;
 Nodes: array[1..MAXNnodes] of NodeT;
 Bars: array[1..MAXNbars] of BarT;
 i: integer;
 x: real;

procedure InitializeBridge;
 var
  fichier: text;
 begin
  reset(fichier, 'bridge3D1.txt');
  readln(fichier, NNodes);
  if (NNodes <> MAXNnodes) then
   writeln('Type mismatch: incorrect number of nodes.');
  Nbars := 3 * NNodes - 6;
  for i := 1 to NNodes do
   begin
    with Nodes[i] do
     begin
      readln(fichier, PositionX, PositionY, PositionZ);
      Index := i;
     end;
   end;
  for i := 1 to Nbars do
   begin
    with Bars[i] do
     begin
      readln(fichier, Index, FromNode, ToNode);
     end;
   end;
  close(fichier);
 end;

procedure DisplayBridge;
 var
  fromX, fromY, fromZ, toX, toY, toZ: real;
  FromNode, ToNode: integer;
  PointX, PointY, PointZ: real;
 begin
  PenSize(6);
  for i := 1 to MAXNbars do
   begin
    FromNode := Bars[i].FromNode;
    ToNode := Bars[i].ToNode;
    fromX := Nodes[FromNode].PositionX;
    fromY := Nodes[FromNode].PositionY;
    fromZ := Nodes[FromNode].PositionZ;
    toX := Nodes[ToNode].PositionX;
    toY := Nodes[ToNode].PositionY;
    toZ := Nodes[ToNode].PositionZ;
    PenSize(4);
    Draw3DLine2(fromX, fromY, fromZ, toX, toY, toZ,Perspective);
   end;
  for i := 1 to MAXNnodes do
   begin
    PointX := Nodes[i].PositionX;
    PointY := Nodes[i].PositionY;
    PointZ := Nodes[i].PositionZ;
    Draw3DPoint2(PointX, PointY, PointZ, 10, Perspective);
   end;
 end;


begin
 SetGraphSize(5,-1,-2, 6);
 InitializeBridge;
 DisplayBridge;
 readln;
end.
```

Programmation I : Examen Final

Physiciens

Comment s'y prendre

Il y a 4 exercices (e1.p, e2.p, e3.p, e4.p) pour lesquels vous devez écrire un programme PASCAL. Remettez le fichier source et l'exécutable dans le répertoire ~gennart/examen-final/physique/login_name. N'oubliez pas de mettre votre nom en commentaire dans chacun de vos programmes.

1 Utilisation des routines de calcul matriciel

On demande d'écrire un programme qui détermine les coefficients c_i d'un polynôme de degré n qui passe par n+1 points donnés (x_0, y_0), (x_1,y_1), ..., (x_n,y_n). Pour cela, il faut initialiser un système de la façon suivante :

en général		n = 2
taille du système = (n+1) lignes, (n+2) colonnes		
$s[i,j] = x_{i-1}^{j-1}$	avec $1 \le i \le n+1$, $1 \le j \le n+1$	$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$
$s[i,n+2] = y_{i-1}$	avec $1 \le i \le n+1$	

(EQ 1)

On demande d'écrire un programme qui (1) initialise un vecteur de n+1 valeurs (x_0,y_0), (x_1,y_1), ...(x_n,y_n), (2) initialise la matrice du système à partir des x_i, y_i (équation 1), et (3) appelle la routine SolveSystem. La routine SolveSystem calcule les coefficients c_0, c_1,..., c_n du polynôme de degré n. Pour vérifier votre programme, si les 3 valeurs (x_i,y_i) sont (-1,3), (0,0), (1,-1), les coefficients c_i sont 0, -2, 1 **(e1.p, 15 points)**

Deux remarques :

- Pour calculer les x_i^j, une boucle qui fait la multiplication de façon répétée convient le mieux.
- vous pouvez utiliser la routine SolveSystem2 qui est dans le fichier ~gennart/exercices-corriges/BGMatrices.p

2 Affichage d'un polynôme

La formule du polynôme est :

$$y = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1} + c_n x^n$$

Pour calculer efficacement le polynôme, on factorise la formule de la façon suivante :

$$y = ((\ldots(((c_n \cdot x + c_{n-1}) \cdot x + c_{n-2}) \cdot x + c_{n-3}) \ldots) \cdot x + c_1) \cdot x + c_0$$

Pour évaluer y, on procède donc de la façon suivante :

$y_0 = c_n$	...
$y_1 = y_0 \cdot x + c_{n-1}$	$y_{n-1} = y_{n-2} \cdot x + c_1$
$y_2 = y_1 \cdot x + c_{n-2}$	$y = y_{n-1} \cdot x + c_0$

On demande de créer une routine qui, étant donné un vecteur de coefficients c(0..n), et une valeur x, calcule la valeur du polynôme représenté par le vecteur c, en utilisant la méthode ci-dessus. On demande ensuite de créer

une routine qui, étant donné le vecteur de coefficients c(0..n), affiche graphiquement à l'écran le polynôme entre deux bornes spécifiées par l'utilisateur. **(e2.p, 15 points)**

3 Entrées/Sorties

On souhaite initialiser un vecteur de points (x_i, y_i) à partir de chiffres contenus dans un fichier. Le format du fichier est : un entier n indiquant la taille du vecteur, suivi de n+1 paires de nombres. On demande d'écrire une routine d'initialisation d'un vecteur à partir d'un fichier. La routine doit vérifier que la taille de la (des) variable(s) correspond au nombre de points dans le fichier. Vous devez inclure la routine dans un programme qui teste la routine. **(e3.p, 15 points)**. Voici un exemple de fichier "points2.txt" à lire :

```
    2
-1.0   3.0
 0.0   0.0
 1.0  -1.0
```

4 Intégration des exercices précédents

On demande d'intégrer les deux exercices précédents. Pour cela, il vous faut écrire un programme qui :

- initialise un vecteur de n+1 points (x_0, y_0), (x_1,y_1), ..., (x_n,y_n), avec x et y compris entre 5 et -5. Les points doivent être lu à partir d'un fichier.
- initialise le système linéaire permettant de trouver les coefficients du polynome passant par les n+1 points ;
- résout le système en utilisant la routine de résolution de système linéaire et trouve les coefficients du polynôme ;
- affiche (1) les n+1 points, et (2) le polynôme entre 5 et -5.

Pour bien faire, le polynôme devrait passer par les points. **(e4.p, 15 points)**

Annexe : explication de l'exercice 1

Formule du polynôme :

$$y = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1} + c_n x^n \qquad \text{(EQ 2)}$$

En particulier, on a, pour des équations du deuxième et $n^{ième}$ degré et les points (x_0, y_0), (x_1,y_1), ..., (x_n,y_n):

$$\begin{array}{l} y_0 = c_0 + c_1 x_0 + c_2 x_0^2 \\ y_1 = c_0 + c_1 x_1 + c_2 x_1^2 \\ y_2 = c_0 + c_1 x_2 + c_2 x_2^2 \end{array} \quad \left| \quad \begin{array}{l} y_0 = c_0 + c_1 x_0 + \ldots + c_{n-1} x_0^{n-1} + c_n x_0^n \\ y_1 = c_0 + c_1 x_1 + \ldots + c_{n-1} x_1^{n-1} + c_n x_1^n \\ \ldots \\ y_n = c_0 + c_1 x_n + \ldots + c_{n-1} x_n^{n-1} + c_n x_n^n \end{array} \right. \qquad \text{(EQ 3)}$$

Ce qui peut s'écrire sous forme matricielle, pour des équations du deuxième et $n^{ième}$ degré :

$$\begin{bmatrix} 1 & x_0^1 & x_0^2 \\ 1 & x_1^1 & x_1^2 \\ 1 & x_2^1 & x_2^2 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} \quad \left| \quad \begin{bmatrix} 1 & x_0^1 & \ldots & x_0^{n-1} & x_0^n \\ 1 & x_1^1 & \ldots & x_1^{n-1} & x_1^n \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_{n-1}^1 & \ldots & x_{n-1}^{n-1} & x_{n-1}^n \\ 1 & x_n^0 & \ldots & x_n^{n-1} & x_n^n \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \ldots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \ldots \\ y_{n-1} \\ y_n \end{bmatrix} \right. \qquad \text{(EQ 4)}$$

Examen Physicien 1996

```
program phys1;
#include "Matrices.h"
#include "Functions.h"

const
  degre = 5;

type
  pointT = record
          x: real;
          y: real;
        end;

var
  points: array[1..degre+1] of pointT;
  matrice: array[1..degre+1, 1..degre+2] of real;
  solutions: array[1..degre+1] of real;

procedure valeursAuHasard;
 var
   i: integer;
   x: integer;
 begin
   x := seed(wallclock);
   for i := 1 to degre+1 do
     begin
      points[i].x := random(x);
      points[i].y := random(x);
     end;
 end;


procedure initialiseMatrice;
 var
   i: integer;
   j: integer;
   z: integer;
 begin
   InitializeMatrix(matrice);
   for i := 1 to degre+1 do
     begin
      matrice[i, degre+2] := points[i].y;
     end;
   for i := 1 to degre + 1 do
     begin
      z := 0;
      for j := 1 to degre + 1 do
        begin
          matrice[i,j] := PUISSANCE(points[i].x, z);
          z := z + 1;
        end;
     end;
   for i := 1 to degre + 1 do
     matrice[i, 1] := 1;
 end;

begin
  InitializeVector(solutions);
  valeursAuHasard;
  initialiseMatrice;
  SolveSystem(matrice, solutions);
  WriteVector(solutions);
  readln;
end.


program phys2;
#include "Graphics.h";
#include "Matrices.h";
#include "Graphs.h";

const
  degre = 8;

type
  polynomeT = array[1..degre+1] of real;

var
  solutions: polynomeT;
  igrec: polynomeT;
  i: integer;
  x: real;
  xmin: real;
  xmax: real;
  ymin: real;
  ymax: real;

procedure bornes(xmi, xma: real);
  var
   i: integer;
   x: real;
 begin
  x := xmi;
  while x < xma do
   begin
    for i := 2 to degre+1 do
     begin
      igrec[i] := igrec[i-1]*x + solutions[degre+1 - i];
     end;
     x := x + 0.01;
     if igrec[degre+1] > ymax then ymax := igrec[degre+1];
     if igrec[degre+1] < ymin then ymin := igrec[degre+1];
   end;
 end;

function calcul(x: real): real;
 var
  i: integer;
 begin
  for i := 2 to degre+1 do
  begin
   igrec[i] := igrec[i-1]*x + solutions[degre+1 - i];
  end;
  calcul := igrec[degre+1];
 end;

procedure dessin;
 var
  z: real;
  fromx, fromy, tox, toy: real;
 begin
  z := xmin;
  SuspendRefresh;
  while z < xmax do
   begin
    fromx := z;
    fromy := calcul(z);
    z := z + 0.01;
    tox := z;
    toy := calcul(z);
    DrawGraphLine(fromx, fromy, tox, toy);
   end;
  ResumeRefresh;
 end;

begin
 writeln('Valeur de x?');
 readln(x);
 writeln('Xmin?');
 readln(xmin);
 writeln('Xmax?');
 readln(xmax);

 InitializeVector(solutions);
 igrec[1] := solutions[degre + 1];
 writeln('Valeur du polynome pour x = ',x,' : ',calcul(x));
 ymax := 0;
 ymin := 0;
 bornes(xmin,xmax);

 SetGraphSize(ymax,xmin,ymin,xmax);
 dessin;
 readln;
end.

{pc -I$GINC -g -o phys2 phys2.p -I$GLIB Matrices.p Graphs.p}


2
-1 3
0 0
1 -1

program phys3;

const
 degre = 2;

type
 pointT = record
         x: real;
         y: real;
        end;

var
 points: array[1..degre+1] of pointT;
 i: integer;

procedure readFile;
 var
  fichier: text;
  degredufichier: integer;
 begin
```

```
  reset(fichier, 'phys3.txt');
  readln(fichier, degredufichier);
  if degredufichier <> degre then
   begin
    writeln('dimension incorrecte');
    pcexit(1);
   end;
  i := 1;
  while not eof(fichier) do
   begin
    readln(fichier, points[i].x, points[i].y);
    i := i + 1;
   end;
  close(fichier);
 end;

procedure affiche;
 var
  i: integer;
 begin
  for i := 1 to degre+1 do
   writeln(points[i].x:6 :2, ' ', points[i].y:6 :2);
 end;

begin
 readFile;
 affiche;
 readln;
end.
```

```
3
-1 3
0 0
1 -1
-1.4 3.5
```

phys 4

```
program aeffacer;
#include "Graphics.h"
#include "Graphs.h"
#include "Matrices.h"
#include "Functions.h"

const
 degre = 3;

type
 pointT = record
        x: real;
        y: real;
      end;
 stringT = varying[32] of char;

var
 points : array[1..degre + 1] of pointT;
 matrice : array[1..degre + 1, 1..degre + 2] of real;
 solutions : array[1..degre + 1] of real;
 x: real;
 xmin, xmax: real;
 ymin, ymax: real;
 igrecs: array[1..degre + 1] of real;

procedure readValeurs;
 var
  fichier: text;
  filename: stringT;
  degredufichier: integer;
  i: integer;
 begin
  filename := 'phys4.txt';
  reset(fichier, filename);
  readln(fichier, degredufichier);
  if degredufichier <> degre then
   begin
    writeln ('Degre incorrect: fichier ne correspond pas au programme');
    pcexit(1);
   end;
  i := 1;
  while not eof(fichier) do
   begin
    readln(fichier, points[i].x, points[i].y);
    i := i + 1;
   end;
  close(fichier);
 end;

procedure InitialiseLaMatriceDuSysteme;
 var
  i,j: integer;
 begin
  for i := 1 to degre + 1 do
   begin
    matrice[i, degre + 2] := points[i].y;
   end;
  for i := 1 to degre + 1 do
   begin
    for j := 1 to degre + 1 do
     begin
      matrice[i,j] := PUISSANCE(points[i].x,j-1);
     end;
   end;
  for i := 1 to degre + 1 do
   matrice[i,1] := 1;
 end;

function calcule(x: real): real;
 var
  i: integer;
 begin
  igrecs[1] := solutions[degre + 1];
  for i := 2 to degre + 1 do
   begin
    igrecs[i] := igrecs[i - 1] * x + solutions[degre + 2 - i]
   end;
  calcule := igrecs[degre + 1];
 end;

procedure TrouveY;
 var
  z: real;
  temp: real;
 begin
  z := xmin;
  while z < xmax do
   begin
    temp := calcule(z);
    if z < ymin then ymin := z;
    if z > ymax then ymax := z;
    z := z + 0.01;
   end;
 end;

procedure DessineLesAxes;
 begin
  DrawGraphLine(xmin,0,xmax,0);
  DrawGraphLine(0,ymin,0,ymax);
 end;

procedure DessineLesPoints;
 var
  i: integer;
 begin
  for i := 1 to degre + 1 do
   begin
    DrawGraphPoint(points[i].x, points[i].y, 4);
   end;
 end;

procedure DessineLePolynome;
 var
  z: real;
  fromx, fromy, tox, toy: real;
 begin
  z:= xmin;
  while z < xmax do
   begin
    fromx := z;
    fromy := calcule(z);
    z := z + 0.01;
    tox := z;
    toy := calcule(z);
    DrawGraphLine(fromx, fromy, tox, toy);
   end;
 end;

begin
 readValeurs;
 writeln('Valeur de x?');
 readln(x);
 writeln('Xmin?');
 readln(xmin);
 writeln('Xmax');
 readln(xmax);
 InitialiseLaMatriceDuSysteme;
 SolveSystem(matrice, solutions);
 ymin := 0;
 ymax := 0;
 TrouveY;
 SetGraphSize(ymax, xmin, ymin, xmax);
 DessineLesAxes;
 DessineLesPoints;
 DessineLePolynome;
 readln;
end.
```