a. Implémenter l'algorithme d'orthonormalisation de Schmidt. En déduire les premiers polynômes de Legendre, qui sont les orthonormalisés de la base canonique de $\mathbb{R}[X]$ pour le produit scalaire

$$\langle f, g\rangle = \int_{-1}^{1} f(t)g(t)\,dt.$$

Que dire des racines de l'un de ces polynômes par rapport à celles du suivant ?

b. Résoudre numériquement, en la discrétisant, l'équation de Laplace $\Delta\varphi = 0$, dans le carré $[0, 1]^2$, en imposant une condition au bord. (Pour ceux qui ont des problèmes, on pourra commencer par se placer en dimension un.)

c. Implémenter l'algorithme de réduction d'une forme quadratique.

a. Pour vérifier que le résultat est exact, on utilise la fait que les polynômes en question sont donnés (à un facteur près) par

$$\frac{d^n}{dt^n}\left(t^2 - 1\right)^n.$$

Le premier essai ne semble pas marcher. Voyez-vous pourquoi ?

```
restart;
prod:=(f,g)->int(f*g,t=-1..1);
norme:=f->sqrt(prod(f,f));
normalize:=f->f/norme(f);
n:=4;
for i from 0 to n do
  u[i]:=t^i;
  e[i]:=u[i] - sum( prod(u[i],e[j])*e[j], j=0..i-1 );
  norme(e[i]);
  e[i]:=normalize(e[i]);
od;
[seq(e[j],j=0..n)];
[1,seq(factor(diff((t^2-1)^j,t$j)),j=1..n)];
seq(""[j]/"[j],j=1..n+1);
```

La problème provient de la manière dont Maple évalue les choses : lors du calcul de l'intégrale, il n'évalue pas `e[j]` au début et la considère donc comme une constante. On constate que cela marche si on remplace la somme `sum` par la commande `convert`.

```
prod:=(f,g)->int(f*g,t=-1..1);
norme:=f->sqrt(prod(f,f));
```

```
normalize:=f->f/norme(f);
n:=4;
for i from 0 to n do
  u[i]:=t^i;
  e[i]:=u[i] -
    convert([seq( prod(u[i],e[j])*e[j], j=0..i-1 )], `+`);
  norme(e[i]);
  e[i]:=normalize(e[i]);
od;
[seq(e[j],j=0..n)];
[1,seq(factor(diff((t^2-1)^j,t$j)),j=1..n)];
seq(""[j]/"[j],j=1..n+1);
map(normal,["]);
```

Quand on trace le graphe des premièrs polynômes, on constate que chaque racine de l'un est comprise entre deux racines du suivant.

```
plot([seq(e[i],i=0..n)],t=-1..1);
```

b. On commence par discrétiser l'EDP : remarquons que

$$\Delta f = \frac{f(x+h,y)+f(x-h,y)+f(x,y+h)+f(x,y-h)-4f(x,y)}{h^2}+o(1).$$

Nous n'allons pas chercher une *fonction* inconnue f mais simplement sa valeur sur les points d'un maillage du carré,

$$f_{i,j} = f\left(\frac{i}{n},\frac{j}{n}\right),$$

l'équation devient un système d'équation linéaires,

$$\begin{aligned} f_{i+1,j}+f_{i-1,j}+f_{i,j+1}+f_{i,j-1}-4f_{i,j} &= 0 \\ f_{i,j} &= g_{i,j}. \end{aligned}$$

Les équations du premier type correspondent au laplacien, *i.e.*, à ce qui se passe à l'intérieur du carré, celle du second type aux conditions au bord. On peut écrire ce système linéaire sous forme matricielle, $AX=U$, où la matrice A, de taille $n^2\times n^2$, correspond aux coefficients du système, le grand vecteur colonne X (de taille n^2) correspond aux inconnues $f_{i,j}$ et le vecteur colonne U correspond au second membre. Nous avons transformé la matrice $(f_{i,j})_{i,j}$ en un vecteur de la manière suivante : $X_{n(i-1)+j}=f_{i,j}$ (cela revient à prendre les colonnes de la matrices et à les mettre les unes au dessus des autres — sauf que j'ai dû intervertir lignes et colonnes — ce n'est pas grave). C'est ce passage d'une matrice $(f_{i,j})_{i,j}$ à un vecteur X qui posait des problèmes aux élèves : ils n'en voyaient pas la nécessité.

```
restart;
```

```
with(linalg):

N := 10;
A := array(1..(N)^2, 1..(N)^2, sparse):
U := vector((N)^2, 0);
fonction := (x,y) -> sin(4*Pi*x)*cos(4*Pi*y);
for i from 1 to N do
  for j from1 to N do
    if (i=1) or (i=N) or (j=1) or (j=N)
    then
      U[N*i+j-N] := evalf(fonction(i/N,j/N));
      A[N*i+j-N, N*i+j-N] := 1;
    else
      A[N*i+j-N, N*i+j-N] := -4;
      A[N*i+j-N, N*(i+1)+j-N] := 1;
      A[N*i+j-N, N*(i-1)+j-N] := 1;
      A[N*i+j-N, N*i+j+1-N] := 1;
      A[N*i+j-N, N*i+j-1-N] := 1;
    fi;
  od;
od;
plots[matrixplot](matrix(N,N,convert(linsolve(A,U), list)));
```

c. Étant donnée une forme quadratique, $\sum_{i,j} a_{i,j}x_ix_j$, on la transforme des deux manières suivantes. Tout d'abord, si $a_{i,j}$ et $a_{i,i}$ sont tous deux non nuls, on considère les termes en question comme le développement d'un carré.

$$\begin{aligned}
a_{i,i}x_i^2 + 2a_{i,j}x_ix_j &= a_{i,i}\left(x_i + \frac{a_{i,j}}{a_{i,i}}x_j\right)^2 - \frac{a_{i,j}^2}{a_{i,i}}x_j^2 \\
2a_{i,k}x_ix_k &= 2a_{i,k}\left(x_i + \frac{a_{i,j}}{a_{i,i}}x_j\right)x_k - 2a_{i,k}\frac{a_{i,j}}{a_{i,i}}x_jx_k
\end{aligned}$$

Ensuite, si $a_{i,i}$ et $a_{j,j}$ sont nuls mais pas $a_{i,j}$, on fait apparaitre une différence de deux carrés.

$$\begin{aligned}
2a_{i,j}x_ix_j &= 2a_{i,j}\left(\frac{x_i + x_j}{2}\right)^2 - 2a_{i,j}\left(\frac{x_i - x_j}{2}\right)^2 \\
2a_{i,k}x_ix_k &= 2a_{i,k}\left(\frac{x_i + x_j}{2}\right)x_k + 2a_{i,k}\left(\frac{x_i - x_j}{2}\right)x_k \\
2a_{j,k}x_jx_k &= 2a_{j,k}\left(\frac{x_i + x_j}{2}\right)x_k - 2a_{j,k}\left(\frac{x_i - x_j}{2}\right)x_k
\end{aligned}$$

On peut commencer par programmer séparément ces deux transformations.

```
with(linalg):
```

```
reduit1:= proc(A::matrix,i,j)
local B,k;
  B:=copy(A);
  B[j,j]:=A[j,j]-A[i,j]^2/A[i,i];
  for k from 1 to coldim(A) do
    if k<>i and k<>j
    then
      B[j,k]:=A[j,k]-A[i,k]*A[i,j]/A[i,i];
      B[k,j]:=B[j,k];
    fi;
  od;
  B[i,j]:=0;
  B[j,i]:=0;
  evalm(B);
end;
```

Je suggère vivement de regarder ce qui se passe sur un exemple.

```
B:=matrix(3,3,[1,1,1, 1,0,1, 1,1,0]);
reduit1(B,1,2);
reduit1(",1,3);
```

Voici maintenant l'autre transformation.

```
reduit2:=proc(A::matrix,i,j)
local B,k;
  B:=copy(A);
  B[i,i]:=2*A[i,j];
  B[j,j]:=-2*A[i,j];
  B[i,j]:=0; B[j,i]:=0;
  for k from 1 to coldim(A) do
    if k<>i and k<>j
    then
      B[i,k]:=A[i,k]+A[j,k];
      B[k,i]:=B[i,k];
      B[j,k]:=A[i,k]-A[j,k];
      B[k,j]:=B[j,k];
    fi;
  od;
  evalm(B);
end;
A:=matrix(3,3,[0,1,1, 1,0,1, 1,1,0]);
reduit2(A,1,2);
reduit1(",1,3);
```

On peut ensuite mettre tout cela dans une seule procédure ; on essaye tout d'abord d'utiliser les transformations du premier type.

```
reduit0:=proc(A::matrix)
local B,i,j;
  B:=copy(A);
  for i from 1 to coldim(A) do
    for j from i+1 to coldim(A) do
      if B[i,j]<>0 then
        if B[i,i]<>0
        then B:=reduit1(B,i,j);
        else if B[j,j]<>0
        then B:=reduit1(B,j,i); fi;
        fi;
      fi;
    od;
  od;
  evalm(B);
end;
evalm(B); reduit0(B);
```

S'il reste des entrées non diagonales dans la matrice, on essaye alors les transformations du second type.

```
reduit:=proc(A::matrix)
local B,i,j;
  B:=copy(A);
  B:=reduit0(B);
  for i from 1 to coldim(A) do
    for j from i+1 to coldim(A) do
      if B[i,j]<>0
      then
        B:=reduit2(B,i,j); B:=reduit0(B);
      fi;
    od;
  od;
  evalm(B);
end;
evalm(A); reduit(A);
```

Je laisse au lecteur le soin d'améliorer ces procédures, en particulier en se débarassant des inutiles et incessantes copies de matrices.