- Veronese : Calcula os pontos da Veronese V^{n} _ {d} .

In[2]:=
```
Veronese[n_, d_] := Module[{permu, saida, i}, (*permu:
                    módulo de código
  todos os pontos em Z^n com valor até d em cada coordenada (importando a ordem)*)
 saida = {};
 permu = Tuples[Range[0, d], n];
         tuplas  intervalo de valores
 For[i = 1, i ≤ Length[permu], i++,
 para cada       comprimento
  If[Sum[permu[[i]][[h]], {h, 1, n}] ≤ d, saida = Append[saida, permu[[i]];];
  se  soma                                      adiciona
 ];
 saida
];
```

- SegreVeronese : Calcula os pontos da Segre - Veronese SV^{n_ 1, ..., n_r} _ {d_ 1, ..., d_r} .

In[3]:=
```
SegreVeronese[n_, d_] := Module[{saida, j},
                         módulo de código
 saida = {};
 If[Not[Length[n] == Length[d]], saida = "Falta informacao";,
 se  n···  comprimento  comprimento
  saida = Veronese[n[[1]], d[[1]]];
  For[j = 2, j ≤ Length[n], j++,
  para cada       comprimento
   saida = Flatten[Table[Table[Flatten[{saida[[p]], Veronese[n[[j]], d[[j]]][[h]]}, 1],
           achatar   tabela  tabela  achatar
       {h, 1, Length[Veronese[n[[j]], d[[j]]]]}], {p, 1, Length[saida]}], 1];
             comprimento                              comprimento
   (*Faz o "cartesiano" dá Segre-Veronese em P^{n_1,...,n_j-1}
     com a Veronese em P^{j}*)
  ];
 ];
 saida
];
```

- MaioresPts : mostra os pontos de maior peso em relação a direção "v" .

In[4]:=
```
MaioresPts[p_, v_] := Module[{i, saida, inter},
                      módulo de código
  saida = {p[[1]]};
  inter = p[[1]];
  For[i = 2, i ≤ Length[p], i++,
  para cada        comprimento
   If[inter.v > p[[i]].v, ,
   se
     If[inter.v == p[[i]].v,
     se
       saida = Append[saida, p[[i]]];,
               adiciona
       saida = {p[[i]]};
       inter = p[[i]];
        ];
    ];
  ]; saida
 ];
```

- ESimplexo : verifica se dados x pontos em Z^n formam um simplexo .

In[5]:=
```
ESimplexo[p_] := Module[{matriz, i, saida},
                 módulo de código
  saida = False;
          falso
  If[Length[p[[1]]] + 1 == Length[p],
  se  comprimento          comprimento
   matriz = {};

   For[i = 2, i ≤ Length[p], i++,
   para cada        comprimento
    matriz = Append[matriz, p[[i]] - p[[1]]];
             adiciona
   ];
   If[MatrixRank[matriz] == Length[p[[1]]],
   se  intervalo matricial          comprimento
    saida = True;
            verdadeiro
   ];
  ];
  saida
 ];
```

- Trivialmente : verifica se "v" separa trivialmente um simplexo na coleção de pontos "p" em Z^n .

In[6]:=
```
Trivialmente[p_, v_] :=
 Module[{n, saida, pord, i, j, peso, pparcial}, (*n: dimensão do espaço;
  saida: saida do programa; pparcial: guarda os pontos que não foram ordenados ainda;
  pparcial: os pontos de maior peso em "a"; pord:
   guarda os pontos "p" de forma decrescente em relação a "v"; i,j: Para o For*)

  n = Length[p[[1]]];
  pord = {};
  pparcial = p;
  saida = {};
  If[Length[p] < n + 1, saida = False,
   If[Length[p] == n + 1,
    If[ESimplexo[p], saida = p;, saida = False;];,
    For[i = 1, i ≤ Length[p], i = i + Length[peso],
     peso = MaioresPts[pparcial, v];
     For[j = 1, j ≤ Length[peso], j++,
      pord = Append[pord, peso[[j]]];
      pparcial = DeleteCases[pparcial, peso[[j]]];
      ];
     ];
    saida = Table[pord[[h]], {h, 1, n + 1}];
    If[ESimplexo[saida] && pord[[n + 2]].v < saida[[n + 1]].v, , saida = False;];
    ];];
  saida
 ];
```

- RandomSimplexo : Dado um conjunto de pontos p em Z^n, a função vai tentar (o número de "tentativa") separar trivialmente um simplexo .

In[7]:=
```
RandomSimplexo[p_, tentativa_ : Infinity] :=
                                 infinito
 Module[{saida, i, v}, (*v: Vetor aleatório em Z^{Length[p[[1]]}*)
 módulo de código                                  comprimento
  saida = False;
          falso
  For[i = 1, i ≤ tentativa, i++,
  para cada
   v = Table[RandomReal[{-1, 1}], {h, 1, Length[p[[1]]}];
       tabela  real aleatório                 comprimento
   If[Length[Trivialmente[p, v]] == Length[p[[1]] + 1,
   se  comprimento                      comprimento
    saida = Trivialmente[p, v];
    Break[];
    interrompe a execução
   ];
  ];
  saida
 ];
```

- Eindependente : Verifica se um conjunto de pontos p são independentes em Z^n .

In[8]:=
```
EIndependente[p_] := Module[{matriz, i, saida},
                     módulo de código
  saida = False;
          falso
  If[Length[p] == 1, saida = True;,
  se  comprimento          verdadeiro
   If[Length[p[[1]]] + 1 > Length[p],
   se  comprimento          comprimento
     matriz = {};

     For[i = 2, i ≤ Length[p], i++,
     para cada        comprimento
      matriz = Append[matriz, p[[i]] - p[[1]]];
               adiciona
     ];
     If[MatrixRank[matriz] == Length[p] - 1,
     se  intervalo matricial          comprimento
      saida = True;
              verdadeiro
     ];
    ];
  ];
  saida
 ];
```

- EDimensaoCheia : verifica se um conjunto p de pontos geram todo o espaço Z^n .

In[9]:=
```
EDimensaoCheia[p_] := Module[{matriz, i, saida},
                      módulo de código
  saida = False;
          falso
  matriz = {};
  If[Not[Length[p] == 0],
  se  n···  comprimento
   If[Length[p] ≥ Length[p[[1]]] + 1,
   se  comprimento   comprimento
     For[i = 2, i ≤ Length[p], i++,
     para cada         comprimento
      matriz = Append[matriz, p[[i]] - p[[1]]];
               adiciona
     ];
     If[MatrixRank[matriz] == Length[p[[1]]], saida = True;];
     se  intervalo matricial          comprimento              verdadeiro
    ];
  ];
  saida
 ];
```

- SeparaSimplexo : Dado um conjunto de pontos p em Z^n, a função vai tentar separar simplexos .

In[10]:=
```
SeparaSimplexo[p_, tentativa_ : Infinity] :=
                                 infinito
 Module[{pparcial, simplexo, sparcial, resto, saida},
 módulo de código
  resto = {};
  simplexo = {};
  pparcial = p;
  While[True,
  repet···  verdadeiro
   If[Not[EDimensaoCheia[pparcial]], resto = pparcial; Break[];,
   se  negação                                          interrompe a execução
     sparcial = RandomSimplexo[pparcial, tentativa];
     simplexo = Append[simplexo, sparcial];
                adiciona
     pparcial = Intersection[pparcial, Complement[pparcial, sparcial]];
                interseção              conjunto complementário
    ];
  ];
  If[Length[resto] == 0,
  se  comprimento
   saida = {simplexo};,
   saida = {simplexo};
   saida = Append[saida, resto];
           adiciona
  ];
  saida
 ];
```

- Defeituosa : Dado uma segre - veronese p, a função vai tentar verificar se não é defeituosa, caso não consiga, a função mostrar que não é h - defeituosa .

In[11]:=
```
Defeituosa[p_, tentativa_ : Infinity] := Module[{i, separasimplexo, VF, resto, h},
                                          infinito          módulo de código
 resto = 0;
 i = 1;
 VF = False;
      falso
 While[Not[VF] && i ≤ tentativa,
 repet··· negação
  separasimplexo = SeparaSimplexo[p];
  If[Length[separasimplexo] == 1, VF = True;,
  se  comprimento                  verdadeiro
   If[i == 1, resto = Length[separasimplexo[[2]]];
   se                 comprimento
    h = Length[separasimplexo[[1]];];
        comprimento
   If[EIndependente[separasimplexo[[2]], VF = True;,
   se                                    verdadeiro
    If[Length[separasimplexo[[2]] < resto,
    se  comprimento
      resto = Length[separasimplexo[[2]];
              comprimento
      h = Length[separasimplexo[[1]];
          comprimento
     ];
    ];
   ];
   i++;
 ];
 If[VF,
 se
  Print["Nao e defeituosa ", separasimplexo];,
  apresenta o resultado
  Print["Falhou, mas nao e ", h, "-defeituosa"];
  apresenta o resultado
 ];
];
```

- NaoEDefeituosa : Dado uma segre - veronese p, a função vai tentar verificar se não é defeituosa (True), caso não consiga, a função mostrar False .

In[12]:=
```
NaoEDefeituosa[p_, tentativa_] := Module[{i, separasimplexo, VF, resto, h, saida},
                                  módulo de código
 resto = 0;
 i = 1;
 VF = False;
      falso
 While[Not[VF] && i ≤ tentativa,
 repet··· negação
  separasimplexo = SeparaSimplexo[p];
  If[Length[separasimplexo] == 1, VF = True;,
  se  comprimento                 verdadeiro
   If[i == 1, resto = Length[separasimplexo[[2]]];
   se                 comprimento
    h = Length[separasimplexo[[1]];];
        comprimento
   If[EIndependente[separasimplexo[[2]], VF = True;,
   se                                    verdadeiro
    If[Length[separasimplexo[[2]] < resto,
    se  comprimento
      resto = Length[separasimplexo[[2]];
              comprimento
      h = Length[separasimplexo[[1]];
          comprimento
     ];
    ];
   ];
   i++;
 ];
 If[VF,
 se
  saida = True;,
          verdadeiro
  saida = False;
          falso
 ];
 saida
];
```

- QNDefeituosas : Dada uma lista L de Segre - Veronese, a função mostra apenas as que conseguiu "ver" que não é defeituosa .

In[13]:=
```
QNDefeituosas[L_, tentativa_] := Module[{saida, Lparcial, i, j, SV},
                                  módulo de código
  Lparcial = L;
  saida = {};
  For[i = 1, i ≤ tentativa, i++,
  para cada
   If[Length[Lparcial] == 0, Break[];,
   se  comprimento          interrompe a execução
    For[j = 1, j ≤ Length[Lparcial], j++,
    para cada        comprimento
      SV = SegreVeronese[Lparcial[[j]][[1]], Lparcial[[j]][[2]]];
      If[NaoEDefeituosa[SV, i],
      se
       saida = Append[saida, Lparcial[[j]]];
               adiciona
      ];
     ];
   ];
   Lparcial = Intersection[Lparcial, Complement[Lparcial, saida]];
              interseção             conjunto complementário
  ];
  saida
 ];
```