En el siguiente notebook procedemos a aplicar paso a paso en código lo realizado en el ejemplo de la memoria, Sección 5.4. Recordemos el contexto del ejemplo:

- Un cierto usuario *Alice* dispone de su clave asimétrica. Recordemos $\mathcal{P} : \mathbb{F}^n \mapsto \mathbb{F}^m$ su clave pública y $\mathcal{P}^{-1} = \mathcal{T}^{-1}(\mathcal{F}^{-1}(\mathcal{S}^{-1}))$ su clave privada.
- Una segunda parte, *Bob*, cifra para Alice un mensaje $\mathbf{z} \in \mathbb{F}^n$ utilizando su clave pública generando $\mathbf{w} = \mathcal{P}(\mathbf{z})$.
- Por último, entramos en este ejemplo como la parte maliciosa *Eve*. Suponiendo haber interceptado la comunicación entre Alice y Bob, por nuestra parte conocemos únicamente el mensaje cifrado $\mathbf{w}$ y la clave pública de Alice $\mathcal{P}$.

Con ello, nuestro objetivo consistirá en resolver el sistema $\mathcal{P} = \mathbf{w}$. Recordemos la "defensa" inerte de los cuerpos finitos utilizados, que nos dan lugar a tener que repetir el proceso entero cada vez que se quiera hallar un nuevo $\mathbf{w}$. Vamos a elegir nuestros parámetros:

- n, dimensión de partida de $\mathcal{P}$.
- m, dimensión del espacio de llegada de $\mathcal{P}$.
- q, cardinal del cuerpo finito sobre el que se construyen los anillos polinomiales $\mathbb{F}_q[x_1, \cdots, x_n]$.

[4]:
```
n = 4
m = 4
q = 4
```

A continuación pasamos a declarar en Sage los tres grupos sobre los que vamos a trabajar: El cuerpo finito $\mathbb{K}$ y los anillos de polinomios $\mathbb{F}_q[x_1, \cdots, x_n]$, $\mathbb{F}_q[x_1, \cdots, x_m]$. Destacamos el establecimiento del orden monomial lexicográfico en ambos anillos, necesario para más adelante proceder a resolver el sistema por eliminación.

[5]:
```
K.<alpha> = FiniteField(q)
F_n = PolynomialRing(K, 'x_', n, order="lex")
F_m = PolynomialRing(K, 'x_', m,order="lex")

F_n.inject_variables()
F_m.inject_variables()
```

```
Defining x_0, x_1, x_2, x_3
Defining x_0, x_1, x_2, x_3
```

Generamos aleatoriamente un sistema polinomial $\mathcal{P}$, que consistirá en m elementos aleatorios de $\mathbb{F}[x_1, \cdots, x_n]$. Dejamos a su vez una línea donde se declara $\mathcal{P}$ como en el ejemplo de la memoria.

[6]:
```
P = [F_n.random_element(degree=2) for _ in range(m)]
P = [(alpha + 1) * x_0 * x_2 + alpha * x_2 * x_3 + alpha * x_2, x_2 * x_3,␣
 ↪x_1**2 + x_1 * x_3 + alpha * x_1 + alpha * x_2 * x_3, x_1 + alpha * x_3**2 + 1]

print("POLINOMIO P:")
print(latex(P))
```

```
POLINOMIO P:
\left[\left(\alpha + 1\right) x_{0} x_{2} + \alpha x_{2} x_{3} + \alpha x_{2},
x_{2} x_{3}, x_{1}^{2} + x_{1} x_{3} + \alpha x_{1} + \alpha x_{2} x_{3}, x_{1}
+ \alpha x_{3}^{2} + 1\right]
```

De manera análoga, generamos el mensaje secreto $\mathbf{z}$ y lo ciframos como $\mathbf{w} = \mathcal{P}(\mathbf{z})$. Recordemos que *Eve* carece de información alguna sobre $\mathbf{z}$ (aunque lo mostramos para cerciorarnos de haber descifrado correctamente), si bien ha interceptado en la comunicación entre *Alice* y *Bob* el texto cifrado $\mathbf{w}$. Dejamos una línea con la asignación de $\mathbf{z}$ como en el ejemplo.

[9]:
```
z = [K.random_element() for _ in range(n)]
z = [alpha + 1, 0, alpha + 1, alpha]
print("TEXTO PLANO z:")
print(latex(z))

w = [p(z) for p in P]
print("TEXTO CIFRADO w:")
print(latex(w))
```

```
TEXTO PLANO z:
\left[\alpha + 1, 0, \alpha + 1, \alpha\right]
TEXTO CIFRADO w:
\left[\alpha, 1, \alpha, 0\right]
```

A continuación, creamos el sistema de polinomios (simplemente restar $\mathbf{w}$) y obtenemos del mismo tanto su ideal como su base de Gröbner según el orden lexicográfico. El paso de declaración del ideal I es simple puesto que ya conocemos los polinomios de $\mathcal{P}'$; sin embargo, ya hemos visto a lo largo de la memoria varios algoritmos de creación de bases de Gröbner y su dificultad. Por simplicidad de código, usaremos el método ya implementado por Sage.

[10]:
```
P_sistema = [P[i] - w[i] for i in range(m)]

I = ideal(P_sistema)
G = I.groebner_basis()

print("BASE DE GRÖBNER G:")
print(latex(G))
```

```
BASE DE GRÖBNER G:
```

```
\left[x_{0} + \alpha + 1, x_{1} + \alpha x_{3}^{2} + 1, x_{2} + x_{3}^{3} +
\left(\alpha + 1\right) x_{3}^{2} + x_{3} + \alpha, x_{3}^{4} + \left(\alpha +
1\right) x_{3}^{3} + x_{3}^{2} + \alpha x_{3} + 1\right]
```

A continuación se define una función que ilustra la resolución por eliminación de un sistema multivariable dada su base de Gröbner lexicográfica. Se implementa el método de forma recursiva para dar más luz al hecho de cómo cada problema de eliminación se va reduciendo a cada ideal I_j, $j = 0, \cdots, n$. Se deja un `print` para comprender a cada paso cómo se forma un árbol de relaciones entre las variables, sus posibles raíces y, con ello, de los posibles valores de $\mathbf{z}$, elementos pertenecientes a la variedad afín generada por el ideal de $\mathcal{P}'$.

[16]:
```
def eliminacion_grobner(G,roots=None,var_order=None,l=0):
    # Preparar diccionario de posibles valores
    if not roots:
        roots = { "x_"+str(i) : [] for i in range(G.nvariables())}
    if not var_order:
        var_order = []

    # Sustituir esos valores en g y extraer los de una variable
    g_elim = [g for g in G if g.nvariables() == 1]
    if not g_elim:
        return roots

    # Copiar la base de Gröbner, de la cual retiramos los polinomios univariable
    G_nueva = list(G)
    G_nueva = [g for g in G_nueva if g not in g_elim]

    # Para cada polinomio univariable
    for g in g_elim:
        # "Apuntamos" la variable de la que depende
        var = g.variable()
        var_order.append(var)
        # Obtenemos sus raíces
        g_roots = [ r[0] for r in g.univariate_polynomial().roots()]
        # Las añadimos al árbol de raíces
        roots[str(var)].append(g_roots)

        # Para cada posible raíz
        for r in g_roots:
            print("\t"*l + "{} ~~> PROBANDO CON {} = {}".
↪format(str(l),str(var),str(r)))
            # Generamos la nueva base de Gröbner en la que se evalúa el x_i de g
            G_eval = [g(**{str(var): r}) for g in G_nueva]
            # Volvemos a resolver por eliminación la base remanente
            eliminacion_grobner(G_eval,roots,var_order,l+1)

    return roots, var_order
```

Se devuelven dos objetos a la salida de `eliminacion_grobner`:

- `roots` es un diccionario de par variable/posibles raíces. Estas posibles raíces se almacenan como listas de listas, donde elementos de una misma lista indican ráices múltiples en una misma G, mientras que raíces en distintas listas indican que dichos valores dependerán del valor múltiple de otra variable (de esta forma se pasa a formar un árbol de raíces de los polinomios.
- `var_order` hace referencia al orden de exploración realizado. Se utiliza para poder construir e interpretar el árbol de soluciones dado el diccionario `roots`.

[17]:
```
eliminacion_grobner(G)
```

```
0 ~~> PROBANDO CON x_0 = alpha + 1
0 ~~> PROBANDO CON x_3 = 1
        1 ~~> PROBANDO CON x_1 = alpha + 1
        1 ~~> PROBANDO CON x_2 = 1
0 ~~> PROBANDO CON x_3 = alpha
        1 ~~> PROBANDO CON x_1 = 0
        1 ~~> PROBANDO CON x_2 = alpha + 1
```

[17]:
```
({'x_0': [[alpha + 1]],
  'x_1': [[alpha + 1], [0]],
  'x_2': [[1], [alpha + 1]],
  'x_3': [[1, alpha]]},
 [x_0, x_3, x_1, x_2, x_1, x_2])
```

Interpretamos de la salida un árbol de soluciones como el siguiente (nos remitimos al ejemplo de la memoria):

```
x_0         x_3        x_1      x_2
alpha+1 ---  1 ------ alpha+1 --  1
        |
        \-- alpha -----  0 ---- alpha+1
```

Luego los dos elementos de la variedad afín son los dos posibles recorridos del nodo raíz a los nodos hoja:

1. $\mathbf{z} = (\alpha + 1, \alpha + 1, 1, 1)$
2. $\mathbf{z} = (\alpha + 1, 0, \alpha + 1, \alpha)$

De entre los cuales sabemos que el texto plano es la segunda opción.

Se adjunta por último una función para generar ejemplos variando los parámetros iniciales q, m, n.

[20]:
```
def generar_ejemplo_q(n,m,q,verbose=False):
    K.<alpha> = FiniteField(q)
    F_n = PolynomialRing(K, 'x_', n, order="lex")
    F_m = PolynomialRing(K, 'x_', m,order="lex")
    F.<x_0> = PolynomialRing(K,'x_0',1)

    F_n.inject_variables()
```

```
    F_m.inject_variables()

    P = [F_n.random_element(degree=2) for _ in range(m)]

    if verbose:
        print("POLINOMIO P:")
        print(latex(P))

    z = [K.random_element() for _ in range(n)]
    if verbose:
        print("MENSAJE PRIVADO z:")
        print(latex(z))

    w = [p(z) for p in P]
    if verbose:
        print("MENSAJE CIFRADO w:")
        print(latex(w))

    P_sistema = [P[i] - w[i] for i in range(m)]
    I = ideal(P_sistema)
    G = I.groebner_basis()
    if verbose:
        print(latex(G))

    return eliminacion_grobner(G)[0]
```

Mediante el comando `%time` de Jupyter podemos medir los tiempos de generación y resolución de los ejemplos para cercionarnos de cómo escala exponencialmente la complejidad del problema (atención a `Wall time` en la salida).

[26]:
```
%time generar_ejemplo_q(8,8,16)
```

```
Defining x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7
Defining x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7
0 ~~> PROBANDO CON x_7 = alpha^3
        1 ~~> PROBANDO CON x_0 = alpha^3 + alpha^2
        1 ~~> PROBANDO CON x_1 = alpha + 1
        1 ~~> PROBANDO CON x_2 = alpha^3 + alpha + 1
        1 ~~> PROBANDO CON x_3 = alpha^3
        1 ~~> PROBANDO CON x_4 = alpha^3 + alpha + 1
        1 ~~> PROBANDO CON x_5 = alpha^3
        1 ~~> PROBANDO CON x_6 = alpha^2 + 1
CPU times: user 32 s, sys: 24 ms, total: 32 s
Wall time: 32 s
```