Symbolic-numeric methods to solve polynomial systems

Matías R. Bender

Technische Universität Berlin

(Supported by the ERC under the European's Horizon~2020 research and innovation programme (grant no.~787840))

Symbolic-numeric algorithms

$$\begin{matrix}\text{Polynomial} \\ \text{systems}\end{matrix} \xrightarrow[\textit{method}]{\textit{Symbolic}} \begin{matrix}\text{Linear algebra} \\ \text{problem}\end{matrix} \xrightarrow[\textit{algorithm}]{\textit{Numeric}} \begin{matrix}\text{Approximation} \\ \text{of solutions}\end{matrix}$$

In [1]:
```
using DynamicPolynomials;
using LinearAlgebra;

@polyvar x
ff =  x^3 - 2 * x^2 - 3 * x + 1
compMat = [ 2 1 0; 3 0 1; -1 0 0 ]
sols = eigvals(compMat)
[ff(x => sols[i]) for i=1:3]

# print - ignore
println("f(x) = ", ff, "\n\nCompanion matrix"); display(compMat); println("Eigen
```

```
f(x) = x^3 - 2*x^2 - 3*x + 1

Companion matrix

3×3 Matrix{Int64}:
  2  1  0
  3  0  1
 -1  0  0

Eigenvalues: [-1.1986912435159978, 0.28646206503160043, 2.9122291784843966]

Solutions:
f(-1.1986912435159978)  = -4.440892098500626e-15
f(0.28646206503160043)  = 2.220446049250313e-16
f(2.9122291784843966)   = 0.0
```

I will show examples using the Julia package `EigenvalueSolver.jl`

In [ ]:
```
## Download and install the package
import Pkg;
Pkg.add(url = "https://github.com/simontelen/JuliaEigenvalueSolver.git");

## Loads the package, might take time!
using EigenvalueSolver;
```

M. R. Bender and S. Telen. "Yet another eigenvalue algorithm for solving polynomial systems." arXiv:2105.08472 (2021).

Solve zero dimensional systems using eigenvalues

- Consider $f_1, \dots, f_r \in S := \mathbb{C}[x_1, \dots, x_n]$ and $I := \langle f_1, \dots, f_r \rangle$.
- We want $\{p \in \mathbb{C}^n : f_1(p) = \dots = f_r(p) = 0\}$. Assume finite solutions, namely $\{\zeta_1, \dots, \zeta_\delta\}$.

- The quotient $R := S/I$ describes the solution set:

$$R = \{g + I : g \in S\} \qquad \text{where } g + I = h + I \iff g - h \in I$$

.

  - Let μ_i be the multiplicity of ζ_i and define $\delta^+ := \sum_i \mu_i$.

- It is a finite dimensional $\mathbb{C}$-vector space of $R \simeq \mathbb{C}^{\delta^+}$.

Eigenvalues of the multiplication maps

Given $g \in S$, consider $M_g : R \to R$, s.t. $h + I \mapsto g\, h + I$. Then,

$$\text{CharPol}(M_g)(\lambda) = \prod_{i=1}^{\delta} (g(\zeta_i) - \lambda)^{\mu_i}.$$

- The multiplication map commutes, i.e., for $h, g \in S$,
$$M_h\, M_g = M_{h\,g} = M_g\, M_h$$

- **Solve system** by computing simultaneous Schur triangulation (diagonalization) of $M_{x_1}, \dots, M_{x_n}$.
  - From the matrix M_{x_i}, we recover all the i-th coordinates of the solutions!

How to compute multiplication maps

- Monomial basis B of $R \longrightarrow \#B = \delta^+$

  Property/definition for every $h \in S$, there are f, g such that $h = f + g$ and
  - $f \in I$, and
  - $g \in S_B := \bigoplus_{b \in B} \mathbb{C} \cdot b$ and is unique.

- Multiplication map $M_{f_0} \in \mathbb{C}^{\delta^+ \times \delta^+}$, for $f_0 \in S$.

  Property/definition for every $c \in \mathbb{C}^{\delta^+}$, there is $\mathrm{f}_c \in I$ such that
$$\left(\sum_i c_i\, b_i \right) f_0 = [b_1, \dots, b_{\delta^+}]\, M_{f_0} \begin{bmatrix} c_1 \\ \vdots \\ c_{\delta^+} \end{bmatrix} + \mathrm{f}_c$$

- **Good news**: computing $M_{f_0} \to$ finding $\mathrm{f}_c \in I$, for every c.
  The elements $\{\mathrm{f}_c\}_c$ belong to finite dim subvector space of I
  $\implies$ use linear algebra to reduce each $\{b_i\, f_0 : b_i \in B\}$.

Sylvester maps

Given monomial sets $\sigma_1, \dots, \sigma_r$, we want to study
$$\text{span}_{\mathbb{C}}(x^\beta\, f_i : x^\beta \in \sigma_i)$$

Consider Σ such that for each $x^\beta \in \sigma_i$, $x^\beta \cdot f_i$ only involves monomials in Σ.

We define the **Sylvester map**

$$Sylv_{f_1,\dots,f_r}^{\sigma_1,\dots,\sigma_r;\Sigma} : S_{\sigma_1} \times \cdots \times S_{\sigma_r} \to S_\Sigma$$

$$Sylv_{f_1,\dots,f_r}^{\sigma_1,\dots,\sigma_r;\Sigma}(g_1, \dots, g_r) := \sum_i g_i\, f_i$$

where $S_{\sigma_i} = \text{span}_{\mathbb{C}}(x^\alpha : x^\alpha \in \sigma_i)$, that is, where each g_i is a linear combinations of the elements in a monomial set σ_i.

$$\text{im}(Sylv_{f_1,\ldots,f_r}^{\sigma_1,\ldots,\sigma_r;\Sigma}) = \text{span}_{\mathbb{C}}(x^\beta f_i : x^\beta \in \sigma_i)$$

Macaulay matrices

- Matrix representation of the Sylvester map $Sylv_{f_1,\ldots,f_n}^{\sigma_1,\ldots,\sigma_n;\Sigma} : (g_1, \ldots, g_n) \mapsto \sum_i g_i f_i$

- The **Macaulay matrix** $\mathrm{M}ac(f_1, \ldots, f_n; \sigma_1, \ldots, \sigma_r; \Sigma)$ is as follows:
  - The columns are indexed by pairs (f_i, x^β), where $x^\beta \in \sigma_i$
  - The rows are indexed by mononomials $x^\alpha \in \Sigma$.
  - The entry of column (i, x^β) and row x^α corresponds to the coefficient of x^α in $x^\beta f_i$.

We use the function **EigenvalueSolver.getRes()** to compute Macaulay matrices matrix.

In [ ]:
```
@polyvar f[0:2]
@polyvar …
@polyvar ⋮
```

In [4]:
```
@polyvar x y

F = [
    x^2-3*x*y+2*y^2-2*x+5*y,
    -2*x*y+6*y^2-x+y+2
    ]

σ = [
     [x, y, 1],
     [x, y, 1]
    ]

Σ = [x^3, x^2*y, x*y^2, y^3, x^2, y^2, x*y, x, y, 1] # Σ = monomials([x,y], 0:3)

MacF = EigenvalueSolver.getRes(F,Σ,σ,[x, y])

# print - ignore
display(vcat(transpose(vcat([(…)],reduce(vcat,[f[mod(i,3)+1].*σ[i] for i=1:2]))
```

```
11×7 Matrix{Term{true, Int64}}:
 …    f₁x  f₁y  f₁   f₂x  f₂y  f₂
 x³   1    0    0    0    0    0
 x²y  -3   1    0    -2   0    0
 xy²  2    -3   0    6    -2   0
 y³   0    2    0    0    6    0
 x²   -2   0    1    -1   0    0
 y²   0    5    2    0    1    6
 xy   5    -2   -3   1    -1   -2
 x    0    0    -2   2    0    -1
 y    0    0    5    0    2    1
 1    0    0    0    0    0    2
```

The transpose of the Macaulay matrix (or multiplying by left)

Given a point $p \in \mathbb{C}^n$, let $(p)^\Sigma$ be the vector of evaluations of every monomial in Σ at p.

$$(p)^\Sigma \cdot \mathrm{M}ac(f_1, \ldots, f_r; \sigma_1, \ldots, \sigma_r; \Sigma) = \left((x^\beta f_i)(p) : x^\beta \in \sigma_i\right)$$

If p is a solution of the system, then the vector $(p)^\Sigma$ belongs to the left kernel of the matrix.

In [5]:
```
ev = (p,Σ) -> map(mon -> subs(mon,(x,y) => (p)), Σ)'

display(F); display(reduce(vcat,[f[mod(i,3)+1].*σ[i] for i=1:2])')

display(
    ev((1,1),Σ)*MacF
)
display(
    ev((2,0),Σ)*MacF
)
```

```
2-element Vector{Polynomial{true, Int64}}:
 x² - 3xy + 2y² - 2x + 5y
 -2xy + 6y² - x + y + 2

1×6 adjoint(::Vector{Term{true, Int64}}) with eltype Term{true, Int64}:
 f₁x  f₁y  f₁  f₂x  f₂y  f₂

1×6 adjoint(::Vector{Polynomial{true, Float64}}) with eltype Polynomial{true,
Float64}:
 3.0  3.0  3.0  6.0  6.0  6.0

1×6 adjoint(::Vector{Polynomial{true, Float64}}) with eltype Polynomial{true,
Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0
```

Computing multiplication map of f_0 in $\langle f_1, \ldots, f_r \rangle$

Let B be a monomial basis of R. To compute the multiplication map of f_0 in R, we need to rewrite any polynomial in

$$\mathrm{span}_{\mathbb{C}}(x^\beta f_0 : x^\beta \in B)$$

as a linear combination of polynomials involing the monomials in B and polynomials in I.

We can use again a Macaulay matrix for this. We need to compute a Schur complement.

In [6]:
```julia
f₀ = x+2*y+1
B = [x*y, x, y, 1]
Macf0 = EigenvalueSolver.getRes([f₀],Σ,[B],[x, y])

# print - ignore
pl= vcat(transpose(vcat([(…)],reduce(vcat,[f[mod(i,3)+1].*σ[i] for i=1:2]),[(⋮)
```

```
12×12 Matrix{Term{true, Int64}}:
 …    f₁x  f₁y  f₁   f₂x  f₂y  f₂   ⋮  f₀xy  f₀x  f₀y  f₀
 x³   1    0    0    0    0    0    ⋮  0     0    0    0
 x²y  -3   1    0    -2   0    0    ⋮  1     0    0    0
 xy²  2    -3   0    6    -2   0    ⋮  2     0    0    0
 y³   0    2    0    0    6    0    ⋮  0     0    0    0
 x²   -2   0    1    -1   0    0    ⋮  0     1    0    0
 y²   0    5    2    0    1    6    ⋮  0     0    2    0
 …    …    …    …    …    …    …    …  …     …    …    …
 xy   5    -2   -3   1    -1   -2   ⋮  1     2    1    0
 x    0    0    -2   2    0    -1   ⋮  0     1    0    1
 y    0    0    5    0    2    1    ⋮  0     0    1    2
 1    0    0    0    0    0    2    ⋮  0     0    0    1
```

In [7]:
```julia
f₀ = x+2*y+1
B = [x*y, x, y, 1]
Macf0 = EigenvalueSolver.getRes([f₀],Σ,[B],[x, y])

# print - ignore
pl= vcat(transpose(vcat([(…)],reduce(vcat,[f[mod(i,3)+1].*σ[i] for i=1:2]),[(⋮)
```

```
12×12 Matrix{Term{true, Int64}}:
 …    f₁x  f₁y  f₁   f₂x  f₂y  f₂   ⋮  f₀xy  f₀x  f₀y  f₀
 x³   1    0    0    0    0    0    ⋮  0     0    0    0
 x²y  -3   1    0    -2   0    0    ⋮  1     0    0    0
 xy²  2    -3   0    6    -2   0    ⋮  2     0    0    0
 y³   0    2    0    0    6    0    ⋮  0     0    0    0
 x²   -2   0    1    -1   0    0    ⋮  0     1    0    0
 y²   0    5    2    0    1    6    ⋮  0     0    2    0
 …    …    …    …    …    …    …    …  …     …    …    …
 xy   5    -2   -3   1    -1   -2   ⋮  1     2    1    0
 x    0    0    -2   2    0    -1   ⋮  0     1    0    1
 y    0    0    5    0    2    1    ⋮  0     0    1    2
 1    0    0    0    0    0    2    ⋮  0     0    0    1
```

We split our Macaulay matrix in four parts.

- We split the columns in two groups,
$$\{(x^\beta, f_i) : x^\beta \in \sigma_i\}_{i>0} \text{ and } \{(x^\beta, f_0) : x^\beta \in B\}$$
- We split the rows in the monomials in $\Sigma \setminus B$ and σ_0.

$$\left[\begin{array}{c|c} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{array}\right]$$

If B is a monomial basis of R the size of $M_{2,2}$ is $\prod_{i>0} \deg(f_i)$, i.e., the Bézout bound.

If $M_{1,1}$ invertible, the Schur complement is the multiplication map of f_0, M_{f_0}, in R.

$$\left[\begin{array}{c|c} M_{11} & M_{12} \end{array}\right] \left[\begin{array}{c|c} Id & -M^{-1} M_{12} \end{array}\right] \quad \left[\begin{array}{c|c} M_{11} & 0 \end{array}\right]$$

In [8]:
```
schurCompl_f0 = Macf0[end-3:end,:] - MacF[end-3:end,:]*inv(MacF[1:6,:])*Macf0[1:

# print - ignore
pl= vcat(transpose(vcat([(…)],reduce(vcat,[f[mod(i,3)+1].*σ[i] for i=1:2]),[(⋮)
```

```
12×12 Matrix{Term{true, Float64}}:
 …    f₁x   f₁y   f₁    f₂x   f₂y   f₂    ⋮  f₀xy    f₀x     f₀y     f₀
 x³   1.0   0.0   0.0   0.0   0.0   0.0   ⋮  0.0     0.0     0.0     0.0
 x²y  -3.0  1.0   0.0   -2.0  0.0   0.0   ⋮  0.0     0.0     0.0     0.0
 xy²  2.0   -3.0  0.0   6.0   -2.0  0.0   ⋮  0.0     0.0     0.0     0.0
 y³   0.0   2.0   0.0   0.0   6.0   0.0   ⋮  0.0     0.0     0.0     0.0
 x²   -2.0  0.0   1.0   -1.0  0.0   0.0   ⋮  0.0     0.0     0.0     0.0
 y²   0.0   5.0   2.0   0.0   1.0   6.0   ⋮  0.0     0.0     0.0     0.0
 …    …     …     …     …     …     …     …  …       …       …       …
 xy   5.0   -2.0  -3.0  1.0   -1.0  -2.0  ⋮  6.167   4.333   1.667   0.0
 x    0.0   0.0   -2.0  2.0   0.0   -1.0  ⋮  -6.917  2.667   0.333   1.0
 y    0.0   0.0   5.0   0.0   2.0   1.0   ⋮  -4.333  -4.667  0.667   2.0
 1.0  0.0   0.0   0.0   0.0   0.0   2.0   ⋮  13.833  0.667   -0.667  1.0
```

- The left eigenvectors of the multiplication map correspond to evaluation at the solutions.

In [9]:
```
eigv = eigen(transpose(schurCompl_f0))

println("Eigenvalues")
display(eigv.values)
println("\nEigenvectors")
display(hcat(B,round.(eigv.vectors.*transpose(inv.(eigv.vectors[4,:])),digits=3)
```

```
Eigenvalues

4-element Vector{ComplexF64}:
 1.7500000000000007 - 5.629165124598858im
 1.7500000000000007 + 5.629165124598858im
 3.0000000000000018 + 0.0im
  3.999999999999992 + 0.0im


Eigenvectors

4×5 Matrix{Term{true, ComplexF64}}:
 xy            (-3.875-1.083im)  (-3.875+1.083im)  (0.0+0.0im)  (1.0+0.0im)
 x             (0.25-3.031im)    (0.25+3.031im)    (2.0+0.0im)  (2.0+0.0im)
 y             (0.25-1.299im)    (0.25+1.299im)    (0.0+0.0im)  (0.5+0.0im)
 (1.0+0.0im)  (1.0+0.0im)       (1.0+0.0im)       (1.0+0.0im)  (1.0+0.0im)
```

In [10]:
```
F[1]((x,y)=>(2,0)), F[2]((x,y)=>(2,0))
```

Out[10]:
```
(0, 0)
```

- We can also solve the system using the eigenvalues. We consider a different f_0 and repeat the computation

In [11]:
```
h₀ = 3*x-y-2

Mach0 = EigenvalueSolver.getRes([h₀],Σ,[B],[x, y])

schurCompl_h0 = Mach0[end-3:end,:] - MacF[end-3:end,:]*inv(MacF[1:6,:])*Mach0[1:
```

Out[11]:
```
4×4 Matrix{Float64}:
   5.33333    6.0    2.66667    0.0
 -10.8333     3.0   -0.166667   3.0
  -3.66667  -14.0   -1.83333   -1.0
  21.6667     2.0    0.333333  -2.0
```

- As we want to match the eigenvalues corresponding to the same solutions, we reuse the eigenvectors.

In [12]:
```
matrixVsEigen = transpose(schurCompl_h0)*eigv.vectors

newEigenvalues = [(j -> matrixVsEigen[j,i]/eigv.vectors[j,i])(
        findfirst((el -> norm(el) > 1e-6),eigv.vectors[:,i])) for i=1:4]
```

Out[12]:
```
4-element Vector{ComplexF64}:
 -1.5000000000000022 - 7.794228634059945im
 -1.5000000000000022 + 7.794228634059945im
  3.9999999999999893 + 0.0im
  3.5000000000000075 + 0.0im
```

- Now, we just need to solve a linear system.

In [13]:
```
sols = [1 2; 3 -1]^(-1)*(transpose(hcat(eigv.values, newEigenvalues)) .- [1,-2])

println("$(length(sols[1,:])) solutions")
display(hcat([x;y],sols))
for i=1:4
    println("Relative error: ", norm([F[j]((x,y) => sols[:,i]) for j=1:2])/norm(
end
```

```
4 solutions

2×5 Matrix{Term{true, ComplexF64}}:
 x  (0.25-3.03109im)  (0.25+3.03109im)  (2.0+0.0im)           (2.0+0.0im)
 y  (0.25-1.29904im)  (0.25+1.29904im)  (2.29975e-15+0.0im)  (0.5+0.0im)

Relative error: 2.702118712700938e-15
Relative error: 2.702118712700938e-15
Relative error: 4.489852945960278e-15
Relative error: 7.576360059454253e-15
```

Why don't we consider the multiplication maps wrt coordinates directly?

I.e., why we don't consider $f_0 = x_i$? Numerically bad!

In [14]:
```
Macx = EigenvalueSolver.getRes([x],Σ,[B],[x, y])

schurCompl_x = Macx[end-3:end,:] - MacF[end-3:end,:]*inv(MacF[1:6,:])*Macx[1:6,:

eigv_x = eigen(transpose(schurCompl_x))
println("Eigenvalues")
display(eigv_x.values)
println("\nEigenvectors")
display(round.(eigv_x.vectors.*(transpose(inv.(eigv_x.vectors[4,:]))),digits=3))
```

```
Eigenvalues

4-element Vector{ComplexF64}:
 0.24999999999999872 - 3.031088913245533im
 0.24999999999999872 + 3.031088913245533im
  1.9999999999999996 - 1.0877919644084146e-15im
  1.9999999999999996 + 1.0877919644084146e-15im


Eigenvectors

4×4 Matrix{ComplexF64}:
 -3.875-1.083im  -3.875+1.083im  0.334+2.014im  0.334-2.014im
   0.25-3.031im    0.25+3.031im    2.0-0.0im      2.0+0.0im
   0.25-1.299im    0.25+1.299im  0.167+1.007im  0.167-1.007im
    1.0+0.0im       1.0-0.0im      1.0+0.0im      1.0+0.0im
```

Choosing basis for quotient ring

In the previous example, we assumed we know:

- Which polynomials in I to consider.
- A monomial basis for R.

What can we do in practice?

We choose a big enough degree d and, for each i, we consider all monomials of degree at most $d - d_i$, where $d_i := \deg(f_i)$, that is,

$$\sigma_i = \{x^\beta \in S_{\leq d-d_i}\}$$

- Under good assumptions, $\mathbb{C}[x_1, \ldots, x_n]/\langle f_1, \ldots, f_r \rangle$ is isomorphic to $\text{cokernel}(\text{Sylv}_{\text{f}_1,\ldots,\text{f}_\text{r}}^{\sigma_1,\ldots,\sigma_\text{r};\Sigma})$.
  - In particular, $\text{rank}(\text{coker}(\text{Sylv}_{\text{f}_1,\ldots,\text{f}_\text{r}}^{\sigma_1,\ldots,\sigma_\text{r};\Sigma})) = \delta^+$
- Moreover, the set σ_0 will contain a monomial basis B for R.

How to recover the monomial basis and the multiplication maps?

- We want to decide which monomials in σ_0 form a numerically good basis for R.
- For that, we need δ^+ monomials in σ_0 which are linear independent modulo $\text{im}(\text{Sylv}_{\text{f}_1,\ldots,\text{f}_\text{r}}^{\sigma_1,\ldots,\sigma_\text{r};\Sigma})$.
- To test, we compute the image $\text{Sylv}_1^{\sigma_0;\Sigma}$ mod $\text{coker}(\text{Sylv}_{\text{f}_1,\ldots,\text{f}_\text{r}}^{\sigma_1,\ldots,\sigma_\text{r};\Sigma})$.
- This can be done using the left kernel of the Macaulay $\text{Mac}(f_1, \ldots, f_r; \sigma_1, \ldots, \sigma_r)$.

Via the Schur complement, we have been doing this implicitly.

$$\left[\begin{array}{c|c} Id & -M_{2,1}\,M_{1,1}^{-1} \end{array}\right]\left[\begin{array}{c|c} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{array}\right] = \left[\begin{array}{c|c} 0 & M_{2,2} - M_{2,1}\,M_{1,1}^{-1}\,M_{1,2} \end{array}\right]$$

- Numerically, we do so by computing the singular value decomposition of the matrix.

In [15]:
```
Σ2 = [y^2, y*x, x^2, x, y, 1]
σ2 = [monomials([x,y],0), monomials([x,y],0:1)]

F2 = [y^2-4*x^2-2*y-8*x-3, -1e-4*x+y+1]

M2 = EigenvalueSolver.getRes(F2,Σ2,σ2,[x,y])

# print - ignore
println("New system F = (f₁,f₂)");display(F2);display(vcat(transpose(vcat([(…)],
```

```
New system F = (f₁,f₂)

2-element Vector{Polynomial{true, Float64}}:
 -4.0x² + y² - 8.0x - 2.0y - 3.0
 -0.0001x + y + 1.0

7×5 Matrix{Term{true, Float64}}:
 …    f₁    f₂x      f₂y      f₂
 y²   1.0   0.0      1.0      0.0
 xy   0.0   1.0      -0.0001  0.0
 x²   -4.0  -0.0001  0.0      0.0
 x    -8.0  1.0      0.0      -0.0001
 y    -2.0  0.0      1.0      1.0
 1.0  -3.0  0.0      0.0      1.0
```

In [16]:
```
function getCokernel(MM)
    svdobj = svd(transpose(MM), full = true)
    firstzerosingval = length(svdobj.S) + 1
    firstzeroSingval = findfirst(sv -> sv < 1e-6 * svdobj.S[1], svdobj.S)
    if !isnothing(firstzeroSingval); rk = firstzerosingval - 1; else; rk = lengt
    return transpose(svdobj.V[:, rk+1:end])
end

cokerN = getCokernel(M2)
```

Out[16]:
```
2×6 transpose(::Matrix{Float64}) with eltype Float64:
 -0.296912  -0.350188  -0.70027  0.350118   0.296877  -0.296842
  0.495153  -0.209952  -0.41984  0.20991   -0.495174   0.495195
```

- From this, we can see that in this example we shouldn't choose $[y, 1]$ as our basis for $\mathbb{C}[x, y]/\langle f_1, f_2\rangle$.

In [17]:
```
M2_1 = EigenvalueSolver.getRes([1],Σ2, [monomials([x,y],0:1)],[x,y])

N_1 = cokerN*M2_1

display(F2);display(vcat(monomials([x,y],0:1)', N_1))
```

```
2-element Vector{Polynomial{true, Float64}}:
 -4.0x² + y² - 8.0x - 2.0y - 3.0
 -0.0001x + y + 1.0

3×3 Matrix{Term{true, Float64}}:
 x         y          1.0
 0.350118  0.296877   -0.296842
 0.20991   -0.495174  0.495195
```

- In this example, we can choose $B = [x, 1]$
- In practice, we use QR with optimal pivoting to choose the set B.

- For each $f_0 \in R_{\leq d_0}$, we have a map $\mathcal{N}_{f_0} := \kappa \circ Sylv^B_{f_0}$, where

$$\bigoplus_{i \geq 1} S_{\leq d-d_i} \xrightarrow{\mathrm{Sylv}^{\sigma_1,\ldots,\sigma_r}_{f_1,\ldots,f_r}} S_{\leq d} \xrightarrow{\kappa} \mathrm{coker}(\mathrm{Sylv}^{\sigma_1,\ldots,\sigma_r}_{f_1,\ldots,f_r}) \rightarrow 0$$

  We represent $\mathcal{N}_{f_0}$ with a matrix

$$N_{f_0} := \mathrm{leftKernel}(\mathrm{Mac}(f_1, \ldots, f_r; \sigma_1, \ldots, \sigma_r)) \cdot \mathrm{Mac}(f_0; B)$$

- Morally, $\mathcal{N}_{f_0}$ represents a multiplication map of f_0 in two different basis of R:
  - the monomials B and
  - using the cokernel of Sylvester map $\mathrm{Sylv}^{\sigma_1,\ldots,\sigma_r}_{f_1,\ldots,f_r}$.
- To recover the solutions from eigenvectors, we need to write our multiplication maps in the same basis.
  - We do this by composing $\mathcal{N}_{f_0}$ with the inverse of the map $\mathcal{N}_1$.

In [18]:
```
B = [x 1]

Nf = (f0 -> (cokerN*EigenvalueSolver.getRes([f0],Σ2,[B],[x,y])))

eigv2 = eigen(Nf(x+y+1) * Nf(1)^(-1))

vcat((["Eigenvalues" "Evaluations"]),hcat(eigv2.values, [((x+y+1)((x,y) => (-2,-
```

Out[18]:
```
3×2 Matrix{Any}:
   "Eigenvalues"     "Evaluations"
 -2.0003           -2.0
 -6.10623e-16       0.0
```

- We can do the same for rational functions → the multiplication map of $\frac{f_0}{h_0}$ is $\mathcal{N}_{f_0} \circ (\mathcal{N}_{h_0})^{-1}$.

Solutions at infinity

In some circunstances, eigenvalue methods lead to big numerical errors.

- Consider the system $\{x^2 - 4\,y^2 - 2\,x - 8\,y - 3, -x + 2\,(\varepsilon + 1)\,y - 1\}$.
  - When $\varepsilon = 0$, it has only one solution: $(-1, 0)$
  - When $\varepsilon \neq 0$, it has two solutions: $(-1, 0)$, $\left(\frac{3\,\varepsilon+4}{\varepsilon}, \frac{2}{\varepsilon}\right)$

In [19]:
```
@polyvar ε

F3eps = [x^2-4*y^2-2*x-8*y-3, -x+2*(ε+1)*y-1.]

println("We solve the system \t $F3eps \nby replacing x by 2*(ε+1)*y-1\n")
println(

    subs(x^2-4*y^2-2*x-8*y-3, x => 2*(ε+1)*y-1)
)
println("\nWhen ε = 0, the system has only one solution")
```

```
We solve the system      Polynomial{true, Float64}[x² - 4.0y² - 2.0x - 8.0y -
3.0, 2.0yε - x + 2.0y - 1.0]
by replacing x by 2*(ε+1)*y-1

4*y^2*ε^2 + 8*y^2*ε - 8*y*ε - 16*y

When ε = 0, the system has only one solution
```

Solutions at infinity

In some circunstances, eigenvalue methods lead to big numerical errors.

- Consider the system $\{x^2 - 4\,y^2 - 2\,x - 8\,y - 3, -x + 2\,(\varepsilon + 1)\,y - 1\}$.
  - When $\varepsilon = 0$, it has only one solution: $(-1, 0)$
  - When $\varepsilon \neq 0$, it has two solutions: $(-1, 0)$, $\left(\frac{3\,\varepsilon+4}{\varepsilon}, \frac{2}{\varepsilon}\right)$
  - However, when $\varepsilon \to 0$, the second solutions goes to (∞, ∞).
  - **Big issue**: regardless of algorithm, some eigenvals of the mult. maps are huge $(f_0(\infty, \infty))$!
    - Somewhere, we will invert a nearly singular matrix.
    - Can affect the precision of every computed solution, not only the sols going to infinity.

In [20]:
```
Σ3 = monomials([x,y],0:2); σ3 = [monomials([x,y],0), monomials([x,y],0:1)]

err = 1.e-5
F3 = subs(F3eps,ε => err)

M3 = EigenvalueSolver.getRes(F3,Σ3,σ3,[x,y])

cokerM3 = getCokernel(M3)

M31 = EigenvalueSolver.getRes([1],Σ3,[monomials([x,y],0:1)],[x,y])
N3_1 = cokerM3*M31

f0 = rand(3)'*monomials([x,y],0:1);
M3f0 = EigenvalueSolver.getRes([f0],Σ3,[monomials([x,y],0:1)],[x,y])
N3_f0 = cokerM3*M3f0

# print - ignore
display(F3); println("We can not choose a monomial basis"); display(vcat(transpo
```

```
2-element Vector{Polynomial{true, Float64}}:
 x² - 4.0y² - 2.0x - 8.0y - 3.0
 -x + 2.00002y - 1.0

We can not choose a monomial basis

3×3 Matrix{Term{true, Float64}}:
 x            y             1.0
 -0.0163226   1.0752e-6     0.0163247
 -0.668239    -6.63055e-7   0.668238

However, the picture is different if we multiply by random f₀

3×3 Matrix{Term{true, Float64}}:
 f₀x         f₀y         f₀
 0.82497     0.407719    -0.0095228
 -0.112978   -0.251432   -0.389892
```

This problem force us to change the strategy

- We want to get coordinates to write solutions at infinity
- Our problem is that the solution space $\mathbb{C}^n$ is not compact.
- We will work in a **bigger solutions space** $\mathbb{P}(\mathbb{C}^n)$, which is **compact**.
- Algebraically, we need a system of **homogeneous** equations such that, restricted to $\mathbb{C}^n$, they define the same ideal than our system.
- We will consider the homogenization of our input equations:
  - Given $f \in \mathbb{C}[x_1, \ldots, x_n]$ of degree d, its homogenization is

$$f^h := x_0^d \cdot f\left(\frac{x_1}{x_0}, \ldots, \frac{x_1}{x_0}\right) \in \mathbb{C}[x_0, x_1, \ldots, x_n]$$

$$f := x_1^2 + x_2 + 1 \to f^h = x_1^2 + x_0\, x_2 + x_0^2$$

- Given a system $F = (f_1, \dots, f_r) \in \mathbb{C}[x_1, \dots, x_n]^r$, we will solve

$$F^h = (f_1^h, \dots, f_r^h) \in \mathbb{C}[x_0, x_1, \dots, x_n]^r$$

- We can recover every original solution: for every $(1 : p_1 : \cdots : p_n) \in \mathbb{C}^n \subset \mathbb{P}(\mathbb{C}^n)$,

$$F(p_1, \dots, p_n) = F^h(1 : p_1 : \cdots : p_n)$$

.

- Solutions at infinity of $F \subseteq$ solutions of F^h of the form $(0 : p_1 : \cdots : p_n) \in \mathbb{P}(\mathbb{C}^n)$.

- **Extremely important:** Homogenizing the input equations could lead to a non-zero dimensional system, which kills our strategy (no multiplication maps).

- In what follows, we will **assume that the solutions of F^h at infinity are finite!**
  - In this case, we have that for big enough degree d,

$$\left(\mathbb{C}[x_0, x_1, \dots, x_n]/\langle F^h \rangle\right)_d = \mathbb{C}[\text{Solutions of } F^h]$$

  .

  - Smallest of degree d satisfying this condition → **Castelnuovo-Mumford regularity**.

  - If F^h involves only n equations, geometrically we have a **complete intersection**
    - the Castelnuovo-Mumford regularity is $d_{mac} = \sum_i \text{degree}(f_i) - n + 1$.

Going back to our example, we observe that

- **Bézout bound**: our homogenized system has exactly two solutions (counting multiplicity):

$$\begin{array}{ccc} (-1, 0) & \rightarrow & (-1 : 0 : 1) \\ \left(\frac{3\,\varepsilon+4}{\varepsilon}, \frac{2}{\varepsilon}\right) & \rightarrow & (3\,\varepsilon + 4 : 2 : \varepsilon) \end{array}$$

.

- The Macaulay matrix does not change if we homogenize the equations!

In [21]:
```
@polyvar z

Σ3hom = monomials([x,y,z],2)
σ3hom = [monomials([x,y,z],0), monomials([x,y,z],1)]

homog = ((f,v=z) -> (d -> sum(mm -> mm*v^(d - degree(mm)),terms(f)))(maximum(deg

err = 1.e-5
F3hom = homog.(subs(F3eps,ε => err))

M3hom = EigenvalueSolver.getRes(F3hom,Σ3hom,σ3hom,[x,y,z]);

cokerM3hom = getCokernel(M3hom)

# print - ignore
println("Macaulay matrix for affine system"); display(vcat(transpose(vcat([(…)],
println("Macaulay matrix for homogenization"); display(vcat(transpose(vcat([(…)]
```

```
Macaulay matrix for affine system

7×5 Matrix{Term{true, Float64}}:
 …    f₁    f₂x      f₂y      f₂
 x²   1.0   -1.0     0.0      0.0
 xy   0.0   2.00002  -1.0     0.0
 y²   -4.0  0.0      2.00002  0.0
 x    -2.0  -1.0     0.0      -1.0
 y    -8.0  0.0      -1.0     2.00002
 1.0  -3.0  0.0      0.0      -1.0

Macaulay matrix for homogenization

7×5 Matrix{Term{true, Float64}}:
 …   f₁    f₂x      f₂y      f₂z
 x²  1.0   -1.0     0.0      0.0
 xy  0.0   2.00002  -1.0     0.0
 xz  -2.0  -1.0     0.0      -1.0
 y²  -4.0  0.0      2.00002  0.0
 yz  -8.0  0.0      -1.0     2.00002
 z²  -3.0  0.0      0.0      -1.0
```

- The matrix N_{x_0} is nearly singular:
  - x_0 vanishes at the solutions at infinity! $\implies$ 0 is eigenvalue
- In affine setting, this matrix IS N_1 !

In [22]:
```
N3hom_z = cokerM3hom*(EigenvalueSolver.getRes([z],Σ3hom,[[x,y,z]],[x,y,z]))

println("Try to construct Matrix Nz ->  rank(Nz) ~ 1"); display(vcat([x,y,z]', N
```

```
Try to construct Matrix Nz ->  rank(Nz) ~ 1

3×3 Matrix{Term{true, Float64}}:
 x          y           z
 0.063485   1.1466e-6   -0.0634827
 -0.665417  -5.301e-7   0.665416
```

- We will compute the multiplication maps for rational functions $\frac{f_0}{h_0}$ via $N_{f_0} \cdot N_{h_0}^{-1}$, where h_0 does not vanish at the solutions of F^h.

- If we fix h_0 and compute $n+1$ multiplication maps for $\frac{f_0^0}{h_0}, \ldots, \frac{f_0^n}{h_0}$, for random linear forms $f_0^0, \ldots, f_0^n$, we can recover the coordinates of the solutions.

In [23]:
```
fh0 = rand(3)'*monomials([x,y,z],1);
h0 = rand(3)'*monomials([x,y,z],1);

M3homf0 = EigenvalueSolver.getRes([fh0],Σ3hom,[monomials([x,y,z],1)],[x,y,z])
M3homh0 = EigenvalueSolver.getRes([h0],Σ3hom,[monomials([x,y,z],1)],[x,y,z])

multf0divg0 = (cokerM3hom*M3homf0)[:,[1,2]]*(cokerM3hom*M3homh0)[:,[1,2]]^(-1)

vcat(["Eigenvals M_{f₀/h₀}" "Evals f₀/h₀ at sols"],hcat(sort(eigen(multf0divg0).
```

Out[23]:
```
3×2 Matrix{Any}:
  "Eigenvals M_{f₀/h₀}"   "Evals f₀/h₀ at sols"
 1.09118                  1.09118
 1.83881                  1.83881
```

- The function **EigenvalueSolver.solve_CI_dense()** implements this approach.
- Using **EigenvalueSolver.get_residual()**, we can measure how good are our approximations by computing the relative backward error,

> Relative backward error of $\sum_\alpha c_\alpha x^\alpha$ at $p \in \mathbb{C}^n$ is
>
> $$\frac{|\sum_\alpha c_\alpha p^\alpha|}{1 + \sum_\alpha |c_\alpha p^\alpha|}$$

In [24]:
```
err = 1.e-15
F3 = subs(F3eps,ε => err)

@time sol = EigenvalueSolver.solve_CI_dense(F3,[x,y]; DBD = false, verbose = tru

BWEs = EigenvalueSolver.get_residual(F3,sol,[x,y])
BWE = maximum(BWEs)
```

```
constructing Sylvester matrix...
  0.000016 seconds (125 allocations: 8.875 KiB)
       Sylv is a matrix of size (6, 4)
computing cokernel...
  0.000070 seconds (8 allocations: 4.000 KiB)
       relative size of last nz singular value = 0.18269986505254374
N has size (2, 6)
finding a basis using QR-P...
  2.157991 seconds (3.63 M allocations: 188.847 MiB, 16.57% gc time, 99.89% co
mpilation time)
extracting relevant eigenvalues...
  1.437641 seconds (2.18 M allocations: 111.628 MiB, 1.82% gc time, 97.45% com
pilation time)
inverting monomial map given by A₀...
  1.071799 seconds (1.34 M allocations: 67.479 MiB, 2.43% gc time, 99.94% comp
ilation time)
found 2 solutions
 11.561615 seconds (28.62 M allocations: 1.442 GiB, 7.72% gc time, 99.27% comp
ilation time)
```

Out[24]:
```
8.236883699432725e-16
```

Solving a random system

- Let's now solve a random system given by two plane curves of degree 30.
- We can construct it using the function **EigenvalueSolver.getRandomSystem_dense()**.
- By Bézout's theorem, the number of solutions of the system is $30^2 = 900$.

In [25]:
```
planeCurves = EigenvalueSolver.getRandomSystem_dense([x,y],[30,30])

@time sol = EigenvalueSolver.solve_CI_dense(planeCurves,[x,y]; DBD = false, verb

println("\nMaximum relative backward error: ",
maximum(EigenvalueSolver.get_residual(planeCurves,sol,[x,y])))
```

```
constructing Sylvester matrix...
  0.182002 seconds (2.32 M allocations: 179.372 MiB, 18.46% gc time)
       Sylv is a matrix of size (1830, 930)
computing cokernel...
  0.882154 seconds (12 allocations: 71.642 MiB, 0.66% gc time)
       relative size of last nz singular value = 0.06392379218259907
N has size (900, 1830)
finding a basis using QR-P...
  0.642648 seconds (44.59 k allocations: 218.884 MiB, 0.88% gc time)
extracting relevant eigenvalues...
  1.871560 seconds (1.92 M allocations: 271.459 MiB, 18.67% gc time, 8.11% com
pilation time)
inverting monomial map given by A₀...
  0.000566 seconds (2.73 k allocations: 487.469 KiB)
found 900 solutions
  3.886073 seconds (4.94 M allocations: 816.262 MiB, 10.65% gc time, 3.90% com
pilation time)

Maximum relative backward error: 7.55409854978472e-12
```

Overdetermined systems

- We say that a system $(f_1, \ldots, f_r)$ is overdetermined when $r > n$.
- General overdetermined systems have no solutions,
- In practice, we deal with specific cases where they do.
- **Important**: The eigenvalue method works faster for overdetermined systems!
  - Castelnuovo-Mumford regularity tends to decrease when we add new equations
  - Hence, we solve our system using smaller Macaulay matrices.

To illustrate this, we construct an overdetermined system by fixing the solutions and the degree of the equations.

We use the function **EigenvalueSolver.getVanishingPolynomials()**.

In [26]:
```
mons = monomials([x,y],0:5)
pts = [randn(Float64,2) for i = 1:10]

sysOD = EigenvalueSolver.getVanishingPolynomials(pts, mons, [x,y];augm_prec = fa

sysOD[1]
```

Out[26]:

$$(0.021634023452737936 + 0.0im)x^5 + (-0.07401964532441385 + 0.0im)x^4y + (0.16499376924849904 + 0.0im)x^3y^2 + (-0.25587037815559993 + 0.0im)x^2y^3 + (0.2183008036686416 + 0.0im)xy^4 + (-0.3401163940674935 + 0.0im)y^5 + (-0.04480217916105163 + 0.0im)x^4 + (0.18367359907860425 + 0.0im)x^3y + (0.18936335131647283 + 0.0im)x^2y^2 + (-0.37840934105835616 + 0.0im)xy^3 + (0.6763561823418384 + 0.0im)y^4 + (-0.08572963847578774 + 0.0im)x^3 + (0.00762611765620612 + 0.0im)x^2y + (-0.04407622008154897 + 0.0im)xy^2 + (-0.07615492724602549 + 0.0im)y^3 + (0.059979998261299836 + 0.0im)x^2 + (-0.05667701783747361 + 0.0im)xy + (-0.19791448453442378 + 0.0im)y^2 + (0.04108197531544472 + 0.0im)x + (0.05105983254334641 + 0.0im)y + (-0.018659231782551203 + 0.0im)$$

First we solve a subsystem

- We take only three equations and solve the system.
- It has 10 solutions, but we need to consider the Macaulay matrix at degree 7

In [27]:
```
sol = EigenvalueSolver.solve_OD_dense(sysOD[1:3],[x,y];verbose=true)

println("\nNumber of solutions: ",length(sol))
println("\nMaximum relative backward error: ", maximum(EigenvalueSolver.get_resi
```

```
Looking for an admissible tuple
degree = 5
!!!!!!!!! criterion violated !!!!!!!!!
degree = 6
!!!!!!!!! criterion violated !!!!!!!!!
degree = 7
********* criterion satisfied ********* γ = 18
N has size (18, 36)
finding a basis using QR-P...
  0.000358 seconds (810 allocations: 287.969 KiB)
extracting relevant eigenvalues...
  0.870256 seconds (2.21 M allocations: 108.146 MiB, 2.72% gc time, 95.40% com
pilation time)
inverting monomial map given by A₀...
  0.062938 seconds (122.27 k allocations: 6.430 MiB, 99.16% compilation time)
found 10 solutions

Number of solutions: 10

Maximum relative backward error: 4.716474311075815e-15
```

Now we solve the full system

- We only need to consider the Macaulay matrix at degree 5 !

In [28]:
```
@time sol = EigenvalueSolver.solve_OD_dense(sysOD,[x,y];verbose=true)

println("\nNumber of solutions: ",length(sol))
println("\nMaximum relative backward error: ",
maximum(EigenvalueSolver.get_residual(sysOD,sol,[x,y])))
```

```
Looking for an admissible tuple
degree = 5
********* criterion satisfied ********* γ = 10
N has size (10, 21)
finding a basis using QR-P...
  0.000233 seconds (477 allocations: 180.641 KiB)
extracting relevant eigenvalues...
  0.000653 seconds (919 allocations: 286.719 KiB)
inverting monomial map given by A₀...
  0.000053 seconds (69 allocations: 7.734 KiB)
found 10 solutions
  0.524389 seconds (775.29 k allocations: 46.926 MiB, 3.85% gc time, 99.33% co
mpilation time)

Number of solutions: 10

Maximum relative backward error: 6.50721170470731e-15
```

Further considerations

- To speed up computations, it is possible to compute $\mathrm{coker}(\mathrm{Sylv}_d)$ using $\mathrm{coker}(\mathrm{Sylv}_{d-1})$. This approach leads to an iterative *degree by degree* algorithm.
  - If we run **solve_CI_dense()** with the optional parameter *DBD = true*, it will follow this strategy.

- We have a criterion to check at which degree we can solve the system by recovering (almost) multiplication maps.

In [29]:
```
sol = EigenvalueSolver.solve_CI_dense(sysOD[1:2],[x,y];verbose=true,DBD=true);
```

```
Computing cokernel degree by degree...
degree = 6
degree = 7
degree = 8
degree = 9
N has size (25, 55)
finding a basis using QR-P...
  0.000576 seconds (1.25 k allocations: 446.625 KiB)
extracting relevant eigenvalues...
  0.000718 seconds (2.10 k allocations: 280.797 KiB)
inverting monomial map given by A₀...
  0.000065 seconds (102 allocations: 15.891 KiB)
found 25 solutions
```

When we can use this method?

- The Macaulay matrix of the systems before and after homogenization are exactly the same.

- This tell us that our method works if and only if the homogenization of our input system has a finite number of solutions!

- This assumption makes sense for dense square systems (Bézout theorem).

> We say that a polynomial f is dense if the coefficient of every monomial of degree $\leq \deg(f)$ is non-zero.

- However, if the system is sparse, the homogenization most probably won't have finite solutions
  In this case, the rank of the cokernel of the Macaulay matrix will keep growing! → We won't be able to solve the system.

In [30]:
```
Fsp = rand(3,5)*[x^2, x, y, z, 1]; display(Fsp)

@polyvar w
FspAtInf = [subs(pols,w => 0) for pols in homog.(Fsp,w)]

println("Random system at infinity: \n", FspAtInf, "\n\n The projective line {[0

push!(Fsp,x+y+2*z-1.0)
for i = 2:10
    Σit = monomials([x,y,z],0:i)
    σit = [monomials([x,y,z],0:(i-2)) for j=1:3]
    push!(σit,monomials([x,y,z],0:(i-1)))
    MacIt = EigenvalueSolver.getRes(Fsp,Σit,σit,[x,y,z])
    print("Dim of coker at degree $i -> ",length(monomials([x,y,z],0:i)) - rank(
    println("\t is N_{f_0} invertble? ", length(monomials([x,y,z],0:i)) == rank(
end
```

```
3-element Vector{Polynomial{true, Float64}}:
 0.03359482428206084x² + 0.5963896418485967x + 0.7917820575571479y + 0.6100649
71288946z + 0.5443159520072154
 0.3439206907631174x² + 0.5662950099522472x + 0.15108053442923597y + 0.9871847
455022873z + 0.08038369206762774
 0.33200622618206344x² + 0.4710880268953913x + 0.7432075269762252y + 0.1576139
1688563708z + 0.16066360492711218

Random system at infinity:
Polynomial{true, Float64}[0.03359482428206084x², 0.3439206907631174x², 0.33200
622618206344x²]

 The projective line {[0 : p : q : 0] : (p,q) ∈ P^1} is always solution


Dim of coker at degree 2 -> 7    is N_{f_0} invertble? false
Dim of coker at degree 3 -> 9    is N_{f_0} invertble? false
Dim of coker at degree 4 -> 11   is N_{f_0} invertble? false
Dim of coker at degree 5 -> 13   is N_{f_0} invertble? false
Dim of coker at degree 6 -> 15   is N_{f_0} invertble? false
Dim of coker at degree 7 -> 17   is N_{f_0} invertble? false
Dim of coker at degree 8 -> 19   is N_{f_0} invertble? false
Dim of coker at degree 9 -> 21   is N_{f_0} invertble? false
Dim of coker at degree 10 -> 23  is N_{f_0} invertble? false
```

Solving sparse systems

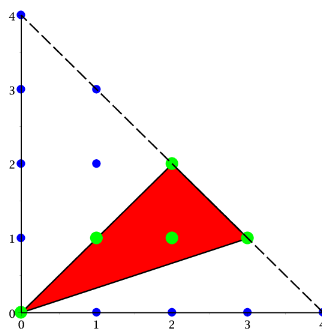
- Until now, we assumed that when we homogenize the system, it has no higher dimensional solutions at infinity.
- This assumption makes sense for dense square systems (Bézout theorem).
- However, for sparse system, we can not assume this.

What is a sparse polynomial system?

- **Newton polytope** of $f = \sum_\alpha c_\alpha \mathbf{x}^\alpha \longrightarrow$ Convex hull of $\{\alpha : c_\alpha \neq 0\}$.

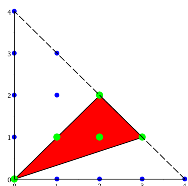
- **Sparse polynomial** $\longrightarrow$ Its Newton polytope is ``small".

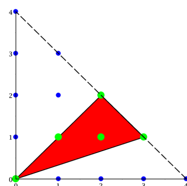
$$1 + xy + x^2y + x^2y^2 + x^3y = \quad 1 + 0 \cdot x + 0 \cdot y + 0 \cdot x^2 + xy + 0 \cdot y^2 + 0 \cdot x^3 + x^2y + 0 \cdot xy^2 + 0 \cdot y^3 + 0 \cdot x^4 + x^3y + x^2y^2 + 0 \cdot xy^3 + 0 \cdot y^4$$

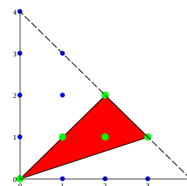


- **Unmixed sparse system** $\longrightarrow$ Polynomials with equal Newton polytope.

$f_1 = 1 + xy + x^2y + x^2y^2 + x^3y$ $\quad$ $f_2 = 1 + 2\,xy - x^2y + x^2y^2 - 2\,x^3y$ $\quad$ $f_3 = 2 + 3\,xy + x^2yx^2y^2 + x^3y$

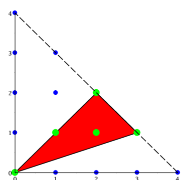


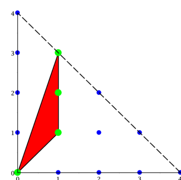


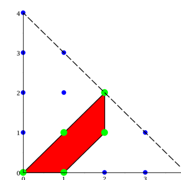


- **Mixed sparse system** $\longrightarrow$ Polynomials with different Newton polytope.

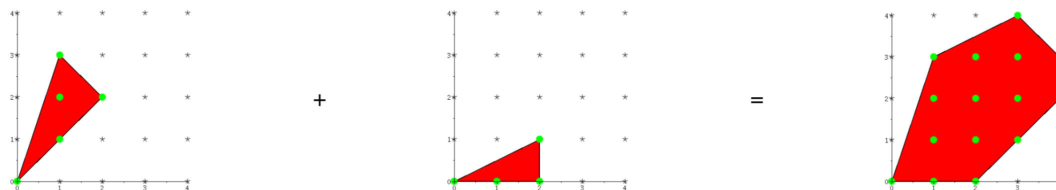
$f_1 = 1 + xy + x^2y + x^2y^2 + x^3y$ $\quad$ $f_2 = 1 + xy + xy^2 + xy^3$ $\quad$ $f_3 = 1 + x + xy + x^2y + x^2y^2$







- Given an integer polytope $A \subset \mathbb{R}^n$,
$$S_A := \text{span}_{\mathbb{C}}(x^\alpha \in S : \alpha \in A)$$
.
- Given $f \in S_A$, $g \in S_B$, we have that $f\,g \in S_{A+B}$, where $A + B$ is the Minkowski sum
$$P + Q\{\alpha + \beta : \alpha \in P, \beta \in Q\}$$



- We have $\mathbb{C}[x_1, \dots, x_n]_{\leq d} = S_{d\,\Delta_n}$, where Δ_n is the n-dimensional standard simplex.

If we have sparse polynomials $f_1, \dots, f_r$ and $r + 1$ integer polytopes $A_1, \dots, A_r, B$ such that, for each i, $NP(f_i) + A_i \subseteq B$, we have the Sylvester map as

$$Sylv_{f_1,\dots,f_r}^{A_1,\dots,A_r;B} : (g_1, \dots, g_r) \mapsto \sum_i g_i \, f_i$$

where $g_i \in S_{A_i}$ and $g_i \, f_i \in S_B$.

- In the previous algorithm (dense case)
  - We restricted to the case
    - $f_i \in S_{d_i \, \Delta_n}$,
    - $A_i = (d - d_i) \, \Delta_n$, and
    - $B = d \, \Delta_n$
  - Moreover, in the square case, we considered $d = \sum_i d_i - n + 1$

We fix n integer polytopes $P_1, \dots, P_n \subset \mathbb{R}^n$ and consider a generic sparse system

$$(f_1, \dots, f_n) \in S_{P_1} \times \dots \times S_{P_n}.$$

- **BKK theorem:** the number of solutions of the system in $(\mathbb{C} \setminus \{0\})^n$ is

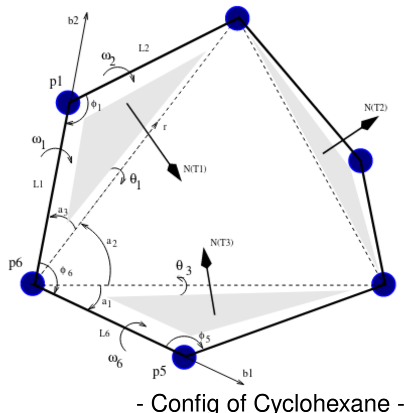
$$\text{MixedVolume}(P_1, \dots, P_n) = \sum_{I \subseteq \{1,\dots,n\}} (-1)^{n-\#I} \; \# \left(\left(\sum_{i \in I} P_i \right) \cap \mathbb{Z}^n \right)$$

- In the same spirit, we can generalize the Macaulay bound.
  - Let $P_0 = \Delta_n$ and consider a generic linear form f_0. Consider

$$A_i = \sum_{j \neq i} P_i \qquad \text{and} \qquad B = \sum_{0 \leq j \leq n} P_i.$$

  - **Theorem:** The corank of $\text{Sylv}_{f_1,\dots,f_n}^{A_1,\dots,A_n;B}$ is $\text{MixedVolume}(P_1, \dots, P_n)$.
  - **Theorem:** $\text{Sylv}_{f_0,f_1,\dots,f_n}^{A_0,A_1,\dots,A_n;B}$ is surjective
- Replacing these maps by the ones in dense ones → solve generic sparse systems!

We can solve square sparse systems using the function **EigenvalueSolver.solve_CI_mixed()**



- Config of Cyclohexane -

Square sparse mixed system, coming from molecular biology [1, Section 3.3].

[1] I. Z. Emiris and B. Mourrain. "Computer algebra methods for studying and computing molecular conformations". Algorithmica, 25(2):372–402, 1999.

In [31]:
```
@polyvar s[1:3]
β = [-13 -1 -1 24 -1; -13 -1 -1 24 -1; -13 -1 -1 24 -1]

mons1 = [1 s[2]^2 s[3]^2 s[2]*s[3] s[2]^2*s[3]^2]
mons2 = [1 s[3]^2 s[1]^2 s[3]*s[1] s[3]^2*s[1]^2]
mons3 = [1 s[1]^2 s[2]^2 s[1]*s[2] s[1]^2*s[2]^2]

Fmol = [β[1,:]'*mons1';β[2,:]'*mons2';β[3,:]'*mons3'][:]

display(Fmol)
```

```
3-element Vector{Polynomial{true, Int64}}:
 -s₂²s₃² - s₂² + 24s₂s₃ - s₃² - 13
 -s₁²s₃² - s₁² + 24s₁s₃ - s₃² - 13
 -s₁²s₂² - s₁² + 24s₁s₂ - s₂² - 13
```

In [32]:
```julia
display(Fmol)

@time sol, A₀, E, D = EigenvalueSolver.solve_CI_mixed(Fmol,s;verbose=true)

BWEs = EigenvalueSolver.get_residual(Fmol,sol,s)
BWE = maximum(BWEs)
println("Maximal backward error $(BWE)")
```

```
3-element Vector{Polynomial{true, Int64}}:
 -s₂²s₃² - s₂² + 24s₂s₃ - s₃² - 13
 -s₁²s₃² - s₁² + 24s₁s₃ - s₃² - 13
 -s₁²s₂² - s₁² + 24s₁s₂ - s₂² - 13

computed the sum of 2 out of 4 polytopes
computed the sum of 3 out of 4 polytopes
computed the sum of 4 out of 4 polytopes
computed the sum of 2 out of 3 polytopes
computed the sum of 3 out of 3 polytopes
computed the sum of 2 out of 3 polytopes
computed the sum of 3 out of 3 polytopes
computed the sum of 2 out of 3 polytopes
computed the sum of 3 out of 3 polytopes
computed the sum of 2 out of 3 polytopes
computed the sum of 3 out of 3 polytopes
constructing Sylvester matrix...
  0.014719 seconds (12.18 k allocations: 1.218 MiB, 92.08% compilation time)
        Sylv is a matrix of size (200, 252)
computing cokernel...
compressing Sylv...
  0.000957 seconds (4 allocations: 706.344 KiB)
  0.010661 seconds (11 allocations: 1.856 MiB)
        rank = 184
        relative size of last nz singular value = 0.0006745231343143405
        gap = 1.8449845631836138e12
N has size (16, 200)
Checking criterion...
********* criterion satisfied ********* γ = 16
finding a basis using QR-P...
  0.000875 seconds (4.39 k allocations: 770.250 KiB)
extracting relevant eigenvalues...
  0.001124 seconds (1.72 k allocations: 574.703 KiB)
inverting monomial map given by A₀...
  0.000063 seconds (99 allocations: 17.953 KiB)
found 16 solutions
  5.552841 seconds (14.70 M allocations: 755.746 MiB, 4.73% gc time, 87.19% co
mpilation time)
Maximal backward error 3.449499618868556e-13
```

How big are these matrices?

- Let us concentrate in dense systems again:
  - Our previous algorithm for sparse system consider $B = (\sum_{i \geq 1} \deg(f_i) + 1) \cdot \Delta_n$
  - However, our previous dense algorithm will consider $B = (\sum_{i \geq 1} \deg(f_i) - n + 1) \cdot \Delta_n$ (Macaulay bound)
- This is a great computational disadvantage, as we work with bigger matrices.

- In general, the smallest polytopes $A_0, \dots, A_n, B$ that we can consider are determined by properties of the Newton polytopes of $f_1, \dots, f_n$.

  Related to problem of computing regularity of complete intersections in a toric variety.

- In some cases, we know some better bounds, e.g., in the **unmixed case**.
  - Fix a polytope P and integers $d_1, \dots, d_n$.
  - Consider an unmixed polynomial system $(f_1, \dots, f_n)$ where $f_i \in S_{d_i \cdot P}$.
  - If the BKK bound is tight for this system, then we can consider

$$B = \left(\sum_{i \geq 1} \deg(f_i) - \text{Codegree}(P) + 2 \right) P$$

  - The codegree of P is the smallest $\lambda \in \mathbb{N}$ such that $\lambda \cdot P$ has an integer point in its relative interior.
    - For example, the codegree of Δ_n is $n+1$ $\longleftarrow$ Macaulay bound!
  - The function **EigenvalueSolver.solve_CI_unmixed()** takes into account this improvement.

In [33]:
```
A = [0 0; 1 0; 1 1; 0 1; 2 2] # monomials whose convex hull is the polytope P
d = [5;12] # equation i has support inside d[i]*P

Funm = EigenvalueSolver.getRandomSystem_unmixed([x,y],A,d)

@time sol, A₀, E, D = EigenvalueSolver.solve_CI_unmixed(Funm,[x,y],A,d; verbose

println("\nMaximal backward error $(maximum(EigenvalueSolver.get_residual(Funm,s
```

```
constructing Sylvester matrix...
  0.018139 seconds (249.36 k allocations: 22.631 MiB)
       Sylv is a matrix of size (685, 450)
computing cokernel...
  0.220440 seconds (14 allocations: 23.350 MiB)
       rank = 445
       relative size of last nz singular value = 0.026594559637965566
       gap = 1.188652641925051e14
N has size (240, 685)
Checking criterion...
********* criterion satisfied ********* γ = 240
finding a basis using QR-P...
  1.226054 seconds (2.16 M allocations: 162.328 MiB, 3.48% gc time, 91.42% com
pilation time)
extracting relevant eigenvalues...
  0.275036 seconds (156.04 k allocations: 47.099 MiB, 14.60% compilation time)
inverting monomial map given by A₀...
  0.157070 seconds (10.14 k allocations: 506.297 KiB, 99.77% compilation time)
found 240 solutions
  3.131071 seconds (5.02 M allocations: 398.984 MiB, 2.84% gc time, 72.78% com
pilation time)

Maximal backward error 2.473420163901671e-9
```

Solutions at infinity, once again

As in the dense case, sparse polynomial systems can also have solutions at infinity. Consider the following system in $\mathbb{C}[x, y]$,

$$\begin{cases} f := y^2 + y + x + 1 \\ g := -y^2 + 2\,y - x + 3 \end{cases}$$

It has only one solution, $[x = -21, y = 4]$. However, BKK bound theorem 2. Let's find it!

In [34]:
```
A = [0 0; 1 0; 0 1; 0 2]

coeffF = [1 1 1 1 ; 3 -1 -2 -1]

monsF = EigenvalueSolver.exptomon(A,[x,y])
println(monsF)

F = coeffF*monsF # F = [y^2+y+x+1,- y^2-2*y - x+3]

display(F)

@time sol, A₀, E, D = EigenvalueSolver.solve_CI_unmixed(F,[x,y],A,[1,1]; verbose

println("\nMaximal backward error $((EigenvalueSolver.get_residual(F,sol,[x,y]))
```

```
Monomial{true}[1, x, y, y²]

2-element Vector{Polynomial{true, Int64}}:
 y² + x + y + 1
 -y² - x - 2y + 3

constructing Sylvester matrix...
  0.000028 seconds (247 allocations: 18.219 KiB)
       Sylv is a matrix of size (9, 8)
computing cokernel...
  0.000083 seconds (9 allocations: 16.281 KiB)
       rank = 7
       relative size of last nz singular value = 0.2580678358329851
       gap = 3.0188125759037563e16
N has size (2, 9)
Checking criterion...
********* criterion satisfied ********* γ = 2
finding a basis using QR-P...
  0.000110 seconds (238 allocations: 286.781 KiB)
extracting relevant eigenvalues...
  0.000121 seconds (183 allocations: 49.875 KiB)
inverting monomial map given by A₀...
  0.000050 seconds (61 allocations: 7.891 KiB)
found 2 solutions
  0.168390 seconds (38.22 k allocations: 2.701 MiB, 14.70% compilation time)

Maximal backward error [2.194876584407612e-15, 0.9999999999999956]
```

In [35]:
```julia
@polyvar t

Ft = subs.(F,(x,y) => (t^2*x,t*y))

display(Ft)

for i = 1:8
    println("$t = 10^$i \t ",subs.(Ft,(x,y,t) => (x,y,10^i))/(10^i)^2)
end
println("\n")
for i = 1:8
    println("$t = 10^$i, F(-t^2, t)/t^2 = ", subs.(Ft,(x,y,t) => (-1,1,10^i))/(1
end
```

```
2-element Vector{Polynomial{true, Int64}}:
 y²t² + xt² + yt + 1
 -y²t² - xt² - 2yt + 3

t = 10^1         Polynomial{true, Float64}[y² + x + 0.1y + 0.01, -y² - x - 0.2
y + 0.03]
t = 10^2         Polynomial{true, Float64}[y² + x + 0.01y + 0.0001, -y² - x -
0.02y + 0.0003]
t = 10^3         Polynomial{true, Float64}[y² + x + 0.001y + 1.0e-6, -y² - x -
0.002y + 3.0e-6]
t = 10^4         Polynomial{true, Float64}[y² + x + 0.0001y + 1.0e-8, -y² - x
- 0.0002y + 3.0e-8]
t = 10^5         Polynomial{true, Float64}[y² + x + 1.0e-5y + 1.0e-10, -y² - x
- 2.0e-5y + 3.0e-10]
t = 10^6         Polynomial{true, Float64}[y² + x + 1.0e-6y + 1.0e-12, -y² - x
- 2.0e-6y + 3.0e-12]
t = 10^7         Polynomial{true, Float64}[y² + x + 1.0e-7y + 1.0e-14, -y² - x
- 2.0e-7y + 3.0e-14]
t = 10^8         Polynomial{true, Float64}[y² + x + 1.0e-8y + 1.0e-16, -y² - x
- 2.0e-8y + 3.0e-16]


t = 10^1, F(-t^2, t)/t^2 = Polynomial{true, Float64}[0.11, -0.17]
t = 10^2, F(-t^2, t)/t^2 = Polynomial{true, Float64}[0.0101, -0.0197]
t = 10^3, F(-t^2, t)/t^2 = Polynomial{true, Float64}[0.001001, -0.001997]
t = 10^4, F(-t^2, t)/t^2 = Polynomial{true, Float64}[0.00010001, -0.00019997]
t = 10^5, F(-t^2, t)/t^2 = Polynomial{true, Float64}[1.00001e-5, -1.99997e-5]
t = 10^6, F(-t^2, t)/t^2 = Polynomial{true, Float64}[1.000001e-6, -1.999997e-
6]
t = 10^7, F(-t^2, t)/t^2 = Polynomial{true, Float64}[1.0000001e-7, -1.9999997e
-7]
t = 10^8, F(-t^2, t)/t^2 = Polynomial{true, Float64}[1.00000001e-8, -1.9999999
7e-8]
```

```
In [36]: Σ = EigenvalueSolver.exptomon(D,[x,y])
σ = map(ee -> EigenvalueSolver.exptomon(ee,[x,y]),E[1:end-1])

MacF = EigenvalueSolver.getRes(F,Σ,σ,[x, y])

evt = ev((-1*t^2,1*t),Σ)

println("\t",evt',"* 1/t^4")
display(vcat(transpose(vcat([(…)],reduce(vcat,[f[mod(i,3)+1].*σ[i] for i=1:2])))
println("=")

display(subs.(reduce(vcat,[F[i].*σ[i]./t^4 for i=1:2]),(x,y) => (-t^2,t)))

println("→ 0, when t → ∞")

println("\n\n We conclude that, ")
display(div.(evt,t^4))
println("belongs to cokernel of Mac")
```

```
        Term{true, Int64}[1, t, t², t³, t⁴, -t², -t³, -t⁴, t⁴]* 1/t^4

10×9 Matrix{Term{true, Int64}}:
 …     f₁   f₁y   f₁y²   f₁x   f₂   f₂y   f₂y²   f₂x
 1     1    0     0      0     3    0     0      0
 y     1    1     0      0     -2   3     0      0
 y²    1    1     1      0     -1   -2    3      0
 y³    0    1     1      0     0    -1    -2     0
 y⁴    0    0     1      0     0    0     -1     0
 x     1    0     0      1     -1   0     0      3
 xy    0    1     0      1     0    -1    0      -2
 xy²   0    0     1      1     0    0     -1     -1
 x²    0    0     0      1     0    0     0      -1

=

8-element Vector{RationalPoly{Polynomial{true, Int64}, Term{true, Int64}}}:
 (t + 1) / (t⁴)
 (t² + t) / (t⁴)
 (t³ + t²) / (t⁴)
 (-t³ - t²) / (t⁴)
 (-2t + 3) / (t⁴)
 (-2t² + 3t) / (t⁴)
 (-2t³ + 3t²) / (t⁴)
 (2t³ - 3t²) / (t⁴)

→ 0, when t → ∞


 We conclude that,

1×9 Matrix{Polynomial{true, Float64}}:
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  -1.0  1.0

belongs to cokernel of Mac
```

Wrapping up

- Symbolic-numeric algorithm reduce problem to linear algebra and solve it numerically.
- Solve polynomial systems by computing eigenvalues of multiplication maps.

- Approximate these maps using Macaulay matrices → size depends on (Castelnuovo-Mumford) regularity.
- Approach works faster for overdetermined systems.
- Correctness depends on the solutions at infinity.
- You can play with these algorithms using *EigenvalueSolver.jl* [https://github.com/simontelen/JuliaEigenvalueSolver (https://github.com/simontelen/JuliaEigenvalueSolver)](https://github.com/simontelen/JuliaEigenvalueSolver)

Questions?