Resultant theory. A brief overview.

December 28, 2017

Recently I am working on a project to explore the memory size effect of players in the iterated prisoner's dilemma. While working on my project, I have came across a number of multivariate systems. There are two things I need to know about the systems:

- do the polynomials of each system have common roots?
- and if they do, could we identify those roots.

To answer these two questions it was decided that I was going down the line of the **resultant theory.** The resultant is a function that can help us to:

- identify whether the system has a common root
- and can also help us to find those roots.

Firstly let explain in detail what the resultant is. Assume that p and q are two polynomials the can be factored into linear factors,

$$\begin{aligned} p(x) = &\quad a_0(x-r_1)(x-r_2)\cdots(x-r_m) \\ q(x) = &\quad b_0(x-s_1)(x-s_2)\cdots(x-s_n) \end{aligned}$$

then the **resultant** R of p and q is defined as,

$$R = a_0^n b_0^m \prod_{i=1}^{m}\prod_{j=1}^{n}(r_i - s_j)$$

From the definition, it is clear that the resultant will equal zero if and only if p and q have at least one common root. Thus, identifying whether common roots exist for a system becomes trivial. Though calculating the resultant as the products of the roots of each polynomial can be computational expensive.

An explicit formula for the resultant as a determinant was given by **Sylvester** in 1840 `http://www.tandfonline.com/doi/abs/10.1080/14786444008649995`. Assume that p and q are given by,

$$\begin{aligned} p(x) = &\quad \sum_{i=0}^{m} a_i x^i \\ q(x) = &\quad \sum_{i=0}^{n} b_i x^i \end{aligned}$$

where $deg(p) = m$ and $deg(q) = n$. The Sylvester matrix is defined as the $(m+n) \times (m+n)$ matrix,

$$\begin{vmatrix} a_0 & a_1 & a_2 & \dots & a_m & 0 & \dots & 0 \\ 0 & a_0 & a_1 & \dots & a_{m-1} & a_m & \dots & 0 \\ & & \ddots & . & . & . & . & . \\ 0 & 0 & . & . & . & . & . & a_m \\ b_0 & b_1 & b_2 & \dots & b_n & 0 & \dots & 0 \\ 0 & b_0 & b_1 & \dots & b_{n-1} & b_n & \dots & 0 \\ & & \ddots & . & . & . & . & . \\ 0 & 0 & . & . & . & . & . & b_n \end{vmatrix}$$

in which there are n rows of p coefficients, m rows of q coefficients, and all elements not shown are zero. It was proven that the determinant of the Sylvester's matrix is equal to the determinant. The Sylvester's matrix is implemented in the python library Sympy. Thus a numerical example is considered using the Sympy library,

```
>>> import sympy as sym
>>> from sympy.polys import subresultants_qq_zz

>>> x = sym.symbols('x')

>>> p = x ** 2 - 5 * x + 6
>>> q = x ** 2 - 3 * x + 2

>>> matrix = subresultants_qq_zz.sylvester(p, q, x)
>>> matrix
Matrix([
[1, -5,  6, 0],
[0,  1, -5, 6],
[1, -3,  2, 0],
[0,  1, -3, 2]])
>>> matrix.det()
0
```

An alternative matrix formulation of the resultant was given by Bezout during the eighteenth century. The Bezout formulation was then reformulated by Cayley in 1865 and this is the second formulation discussed here, we will refer to it as the **Bezout Caley** formulation.

Consider again the polynomials p and q and let $d_{\max} = max(degree(p,x), degree(q,x))$. The construct the following polynomial,

$$\Delta(x,a) = \begin{vmatrix} p(x) & q(x) \\ p(a) & q(a) \end{vmatrix}$$

where α is a new variable and $p(a)$ stands for uniformly replacing x by α in p. Making $x = \alpha$ would make $\Delta = O$ which means that $x\alpha$ divides Δ. The polynomial is called the Bezout Cayley polynomial, is of degree $d_{\max} - 1$ and is defined as,

$$\delta(x,a) = \frac{\Delta(x,a)}{x-a}$$

The polynomial is symmetric in x and α. Every common zero of $p(x)$ and $q(x)$ is a zero of $\delta(x,\alpha)$ no matter what value α has; thus at a common zero of p and q, the coefficient of every power product of α in $\delta(x,\alpha)$ must be 0.

Before moving forward let us consider a numerical example.

```
>>> import imp
>>> bezout = imp.load_source('bezout', '../src/cayley_bezout.py')
>>> p(x)
x**2 - 5*x + 6
>>> q(x)
x**2 - 3*x + 2

>>> a = sym.symbols('alpha')
>>> matrix = sym.Matrix([[p(x), q(x)], [p(a), q(a)]])
>>> matrix
Matrix([
[        x**2 - 5*x + 6,         x**2 - 3*x + 2],
[alpha**2 - 5*alpha + 6, alpha**2 - 3*alpha + 2]])
>>> bezout_polynomial = (matrix.det() / (x - a)).factor().collect(a)
>>> bezout_polynomial.expand()
2*alpha*x - 4*alpha - 4*x + 8
```

Thus we have a $d_{max}-1$ equations by equating the coefficients of the power products of α to zero. Treating $x^0, x^1, \ldots, x^{d_{max}-1}$ as unknowns we retrieve $d_{max}-1$ equations in $d_{max}-1$ unknowns.

Theses will have a common root if and only if the the determinant of the coefficient matrix is equal to 0, this $d_{max} \times d_{max}$ matrix is the Bezout Cayley matrix.

```
>>> bezout.cayley_bezout_matrix(p, q, x)
Matrix([
[-4,  2],
[ 8, -4]])
>>> bezout.cayley_bezout_matrix(p, q, x).det()
0
```

Note that Bezout Cayley matrix for most cases, as shown in our example, is smaller than the Sylverster's. So computational the Bezout Cayley can be favoured over Sylverster's. Both formulation though can handle only up to a 2 variables system.

Dixon in 1908 showed how the Bezout Cayley ca be extended to $n+1$ polynomials in n variables. Thus following a similar manner to that of Bezout Cayley,

$$\Delta(x_1, \ldots, x_n, \alpha_1, \ldots, \alpha_n) = \begin{vmatrix} p_1(x_1, \ldots, x_n) & \ldots & p_n(x_1, \ldots, x_n) \\ p_1(\alpha_1, \ldots, x_n) & \ldots & p_n(\alpha_1, \ldots, x_n) \\ p_1(\alpha_1, \ldots, \alpha_n) & \ldots & p_n(\alpha_1, \ldots, \alpha_n) \end{vmatrix}$$

$$\delta = \frac{\Delta(x_1, \ldots, x_n, \alpha_1, \ldots, \alpha_n)}{((x_1 - \alpha_1) \ldots (x_n - \alpha_n))}$$

The polynomials δ is called the Dixon's polynomial and the Dixon's matrix is constructed as,

A numerical example for Dixon:

```
>>> dixon = imp.load_source('dixon', '../src/dixon.py')
>>> y = sym.symblos('y')
>>> p = sym.lambdify((x, y), x + y)
>>> q = sym.lambdify((x, y), x ** 2 + y **3)
>>> h = sym.lambdify((x, y), x ** 2 + y)
```

```
>>> dixon = dixon.DixonResultant([p, q, h], (x, y))
>>> poly = dixon.get_dixon_polynomial()
>>> matrix = dixon.get_dixon_matrix(polynomial=poly)
>>> matrix.det()
0
```