ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΥΛΟΠΟΙΗΣΗ ΤΩΝ ΑΠΑΛΕΙΦΟΥΣΩΝ ΤΟΥ ΔΙΧΟΝ ΣΤΟ MATHICS

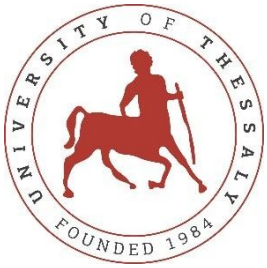
Διπλωματική Εργασία

Θεόδωρος Σαμαράς

Επιβλέπουσα καθηγήτρια:

Ασπασία Δασκαλοπούλου

Βόλος , 2020



UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MATHICS IMPLEMENTATION OF DIXON RESULTANTS

Diploma Thesis

Thodoris Samaras

Supervisor:

Aspasia Daskalopoulou

Volos , 2020

Ευχαριστώ την οικογένεια μου, τους φίλους μου, τον κύριο Ακρίτα και την Ραφ που χωρίς αυτούς η ολοκλήρωση των σπουδών μου θα ήταν αδύνατη .

I got all the time in the world

So for now, I'm just chillin'

RIP

MAC MILLER 1992-2018

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

«Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/ εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής».

Ο Δηλών

Thodoris Samaras

16/7/2020

ΠΕΡΙΛΗΨΗ

Σκοπός της παρούσας εργασίας είναι η παρουσίαση των μεθόδων του Dixon για τις απαλείφουσες. Συγκεκριμένα,ο στόχος είναι απαλοιφή αγνώστων αλλά και η εξέταση συνθηκών για την ύπαρξη ή μη κοινών ριζών σε ένα σύστημα πολυωνύμων. Γίνεται εμβάθυνση στην γενίκευση των μεθόδων αλλά και στην προσέγγιση των Kapur,Saxena και Yang. Τέλος, παρουσιάζεται το υπολογιστικό σύστημα Mathics, η ενσωμάτωση των παραπάνω μεθόδων και ορισμένα παραδείγματα εκτέλεσης σε αυτό.

ABSTRACT

The purpose of this thesis is to present Dixon's methods for the resultants. Specifically, the goal is to eliminate unknowns but also to examine conditions for the existence or not of common roots in a polynomial system. There is a deepening in the generalization of the methods but also in the approach of Kapur, Saxena and Yang. Finally, the Mathics computer system is presented, the implementation of the above methods and some examples of execution in it.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΚΕΦΑΛΑΙΟ 1

ΕΙΣΑΓΩΓΗ

Η θεωρία των απαλείφουσων έχει μια μακρά ιστορία στην επιστήμη των μαθηματικών, και ξεκίνησε με την ανάλυση των γραμμικών συστημάτων. Η πρώτη κατασκευή απαλείφουσας - ή όπως αναφέρεται στη ξένη βιβλιογραφία resultant - προτάθηκε από τους Euler και αργότερα Bezout τον 18ο αιώνα.Τον επόμενο αιώνα ακολούθησε η μέθοδος του Sylvester (1840) ενώ αργότερα μελετήθηκαν γενικεύσεις για πολυώνυμα με πολλαπλές μεταβλητές από τον ίδιο, αλλά ακολούθησε και μια σειρά μελετών τις επόμενες δεκαετίες (Salmon, 1885, Macaulay 1902, Dixon 1908, Van der Waerden 1948 και άλλοι) [1].

Οι απαλείφουσες μας προσφέρουν μία εναλλακτική οπτική στη θεωρία της απαλοιφής . Η απαλείφουσα δύο μόνο πολυωνύμων είναι γνωστή και έχει υλοποιηθεί σε αρκετά συστήματα υπολογιστικής άλγεβρας. Η γενίκευση, μάλιστα, της θεωρίας για παραπάνω από δύο πολυώνυμα αποτελεί ένα σημαντικό εργαλείο στην αναζήτηση λύσεων για εξισώσεις [2].

Οι απαλείφουσες, λοιπόν, μπορούν να χρησιμοποιηθούν για τον προσδιορισμό του κατά πόσον ένα σύστημα n πολυωνύμων με $n-1$ μεταβλητές έχουν μια κοινή ρίζα, χωρίς όμως να επιλύει αναγκαστικά για τις ρίζες. Με τη συστηματική απαλοιφή των μεταβλητών, μπορεί κανείς να αποκτήσει μια μοναδική πολυωνυμική έκφραση με τους συντελεστές των αρχικών πολυωνυμικών εξισώσεων των οποίων η λύση τους ως προς μηδέν μπορεί να μας πληροφορήσει ως προς την ύπαρξη ή όχι κοινής ρίζας. Οι απαλείφουσες συνήθως προκύπτουν συχνά ως διακρίνουσα του πίνακα του οποίου τα στοιχεία είναι από τους παράγοντες των αρχικών πολυωνύμων [3].

Στη θεωρία της απαλοιφής, υπάρχουν δύο τυποποιημένες φόρμουλες για τον σχηματισμό απαλείφουσας δύο μονομεταβλητών πολυωνύμων του βαθμού n: αυτή του Sylvester και αυτή του Bezout. Η απαλείφουσα του Sylvester είναι η ορίζουσα πίνακα τάξης $2n$, ενώ του Bezout είναι η ορίζουσα ενός πίνακα της τάξης n. Έτσι φαίνεται ότι η ορίζουσα του Bezout είναι γενικά πιο γρήγορη στον υπολογισμό. Για τρία διμερή πολυώνυμα δευτέρου βαθμού, ο Dixon παρουσίασε δύο διαφορετικές γενικεύσεις. Η πρώτη προσέγγιση είναι η απαλείφουσα του Sylvester και προκύπτει από τον πίνακα του Sylvester. Ενώ η δεύτερη τεχνική ονομάζεται και απαλείφουσα του Cayley και προκύπτει από τον πίνακα

του Bézout. Στη βιβλιογραφία, η απαλείφουσα του Cayley ονομάζεται συχνά απαλείφουσα του Dixon και αυτή η ονομασία χρησιμοποιείται στις επόμενες ενότητες [3].

Οι μέθοδοι που αναφέρθηκαν παραπάνω απαλείφουν μία μεταβλητή από δύο πολυώνυμα, έτσι για να απαλειφθούν $n-1$ μεταβλητές από n πολυώνυμα χρειάζεται να γενικευτούν με σωστό τρόπο. Για την γενίκευση αυτήν, οι τρεις κυριότερες φόρμουλες είναι οι εξής : του Macaulay, του Dixon και οι απαλείφουσες αραίου πίνακα ή όπως αναφέρεται συνήθως στην αγγλική βιβλιογραφία sparse resultant formulations [4]. Με δεδομένο ένα πλήθος πολυωνύμων, οι παραπάνω μέθοδοι μπορούν να κατασκευάσουν πίνακες από όπου προκύπτουν οι απαλείφουσες. Όπως είναι φυσικό οι πίνακες ονομάζονται Macaulay, Dixon και sparse resultant matrix αντίστοιχα. Τέλος, όπως αποδεικνύεται στο [4] η μέθοδος του Dixon είναι ο πιο γρήγορος τρόπος να υπολογίσει κανείς την ύπαρξη ή μη μιας μη τετριμμένης λύσης ενός συνόλου πολυωνύμων, διότι ο πίνακας Dixon μπορεί να είναι ακόμα και εκθετικά μικρότερος από τους αντίστοιχους πίνακες που σχηματίζουν απαλείφουσες

Για τις μεθόδους του Dixon γίνεται τόσο θεωρητική εμβάθυνση όσο και υλοποίηση στο υπολογιστικό σύστημα Mathics στις επόμενες ενότητες.

ΚΕΦΑΛΑΙΟ 2

ΕΙΣΑΓΩΓΗ ΣΤΙΣ ΑΠΑΛΕΙΦΟΥΣΕΣ ΤΟΥ DIXON

2.1 ΠΟΛΥΩΝΥΜΑ DIXON ΓΙΑ ΔΥΟ ΕΞΙΣΩΣΕΙΣ

Αρχικά ξεκινάμε με την διατύπωση του Cayley [Cayley 1865] πάνω στη μέθοδο του Bezout για την επίλυση ενός συστήματος δύο πολυωνυμικών εξισώσεων. Παρόλο που αναπτύχθηκε πρώτα από τον Cayley, το όνομα του Dixon χρησιμοποιείται περισσότερο [5,6].

‘Εστω δύο πολυώνυμα $f(x)$ και $g(x)$ και α μία βοηθητική μεταβλητή ενώ η ποσότητα degαποτελεί τον μέγιστο των βαθμών του $f(x)$ και $g(x)$.

Η ποσότητα Δ είναι η ορίζουσα του πίνακα αντικατάστασης όπου η βοηθητική μεταβλητή α αντικαθιστάται στις f και g:

$$\Delta(x,\alpha)=\begin{vmatrix} f(x) & g(x) \\ f(a) & g(a) \end{vmatrix}$$

Ονομάζουμε το δ πολυώνυμο του Dixon.

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

Είναι συμμετρικό ως προς x και α και είναι βαθμού $deg-1$. Κάθε κοινό μηδενικό των f και g αποτελεί λύση μηδενικό και στην $\Delta(x,\alpha)$ για όλες τις τιμές του α. Ως εκ τούτου σε κάθε κοινή λύση των f και g, ο συντελεστής κάθε δύναμης του α στην $\delta(x,\alpha)$ θα είναι ίσος με 0. Αυτό δίνει ένα σύνολο E πλήθους deg εξισώσεων που η κάθε μία αντιστοιχεί στους συντελεστές της α^i $(0 \le i \le deg-1)$. Με συμβολισμό πινάκων μπορούμε να δούμε ότι

$$E \equiv M\begin{bmatrix} 1 \\ \vdots \\ x^{deg-1} \end{bmatrix}=\begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

Εάν δούμε τον κάθε συντελεστή από τις δυνάμεις του x (μαζί και τονx^0) σαν μία νέα μεταβλητή z_i, μπορούμε να πάρουμε ένα σύνολο E με deg

ομογενείς γραμμικές εξισώσεις.

$$E \equiv M \begin{bmatrix} z_1 \\ \vdots \\ z_{deg} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

Εάν τελικά οι f και g έχουν κοινή λύση, τότε σημαίνει ότι υπάρχει λύση και για το M. Δηλαδή, αν τα πολυώνυμα έχουν κοινή λύση τότε και ο πίνακας έχει μη τετριμμένη λύση μας οδηγεί στο ότι η ορίζουσα του είναι 0. Η ορίζουσα του M, D λέγεται απαλείφουσα του Dixon, ενώ ο πίνακας M ονομάζεται πίνακας του Dixon. Τέλος, συμπεραίνουμε πως ο μηδενισμός της απαλείφουσας αποτελεί αναγκαία και επαρκή συνθήκη για την ύπαρξη κοινών λύσεων για τις f και g [5,6]. Παρακάτω, παρουσιάζονται μερικά παραδείγματα εφαρμογής τόσο του πολυωνύμου, του πίνακα όσο και της απαλείφουσας του Dixon.

ΠΑΡΑΔΕΙΓΜΑ 1

Έστω οι δύο συναρτήσεις f και g:

$$f(x) = x^2 - 8x + 15 = (x-5)(x-3)$$
$$g(x) = x^3 - 11x^2 + 38x - 40 = (x-5)(x-4)(x-2)$$

Μπορούμε εύκολα να καταλάβουμε ότι υπάρχει κοινή λύση ($x=5$) και θα το δείξουμε παρουσιάζοντας την απαλοιφή του Dixon.
Αρχικά το πολυώνυμο Dixon είναι:

$$P = -250 + 125a - 15a^2 + 125x - 65ax + 8a^2x - 15x^2 + 8ax^2 - a^2x^2$$

όπου το a αποτελεί βοηθητική μεταβλητή. ‘Ετσι, εξισώνοντας τους συντελεστες των δυνάμεων του a καταλήγουμε στον πίνακα του Dixon.

$$M = \begin{vmatrix} -250 & 125 & -15 \\ 125 & -65 & 8 \\ -15 & 8 & -1 \end{vmatrix}$$

Η ορίζουσα του παραπάνω πίνακα είναι 0, που αποδεικνύει την ύπαρξη κοινής ρίζας ($x=5$).

ΠΑΡΑΔΕΙΓΜΑ 2

‘Εστω οι δύο συναρτήσεις f και g:

$$f = 5 - 6x + x^2 = (x-1)(x-5)$$
$$g = -24 + 26x - 9x^2 + x^3 = (x-2)(x-3)(x-4)$$

Παραγωγίζοντας τα δύο παραπάνω πολυώνυμα φαίνεται πως δεν υπάρχει κοινή λύση και μένει να γίνει ο υπολογισμός του πολυωνύμου και του πίνακα του Dixon όπως και στο προηγούμενο παράδειγμα.

Αρχικά το πολυώνυμο Dixon είναι:

$$P = 14 + 21a - 5a^2 + 21x - 33ax + 6a^2x - 5x^2 + 6ax^2 - a^2x^2$$

όπου το a αποτελεί βοηθητική μεταβλητή. ‘Ετσι, εξισώνοντας τους συντελεστες των δυνάμεων του a καταλήγουμε στον πίνακα του Dixon.

$$M = \begin{vmatrix} 14 & 21 & -5 \\ 21 & -33 & 6 \\ -5 & 6 & -1 \end{vmatrix}$$

Η ορίζουσα το παραπάνω πίνακα δεν είναι 0 αλλά -36 και έτσι μπορούμε να επιβεβαιώσουμε την μη ύπαρξη κοινής ρίζας για τα πολυώνυμα f και g.

ΠΑΡΑΔΕΙΓΜΑ 3

Σε αυτό το παράδειγμα θα δείξουμε την εφαρμογή των τύπων του Dixon, χρησιμοποιώντας παραμετρικούς συντελεστές (A).

‘Εστω οι δύο συναρτήσεις f και g:

$$f = (x - 5 + A)(x-2) = 10 - 2A - 7x + Ax + x^2$$
$$g = (x - 2 - A)(x + A) = -2A - A^2 - 2x + x^2$$

Αρχικά το πολυώνυμο Dixon είναι:

$$P = 20 + 10A + 5A^2 - A^3 - 10a - A^2a - 10x - A^2x + 5ax - Aax$$

όπου το a αποτελεί βοηθητική μεταβλητή. ‘Ετσι, εξισώνοντας τους συντελεστες των δυνάμεων του a καταλήγουμε στον πίνακα του Dixon.

$$M = \begin{vmatrix} 20 + 10A + 5A^2 - A^3 & -10 - A^2 \\ -10 - A^2 & 5 - A \end{vmatrix}$$

Όπου η διακρίνουσα του πίνακα δηλαδή η απαλείφουσα πριν και μετά την παραγοντοποίηση της είναι η εξής:

$$D = -(-10 - A^2)^2 + (5 - A)(20 + 10A + 5A^2 - A^3) = -5A(-3 + 2A)(2 + A)$$

Έτσι θέτοντας $D = 0$ και λύνοντας ως προς A παίρνουμε τις τιμές της παραμέτρους που μηδενίζουν την απαλείφουσα.

$$A = -2,\ 0,\ \frac{3}{2}$$

Έτσι η πραγματική λύση του συστήματος $f(x, A) = 0, g(x, A) = 0$ αντικαθιστώντας τις τιμες του A στις f και g είναι:

$$(x, A) = (2, -2), (2, 0), \left(\frac{7}{2}, \frac{3}{2}\right)$$

2.2 ΓΕΝΙΚΕΥΣΗ ΠΟΛΥΩΝΥΜΩΝ DIXON ΓΙΑ ΔΥΟ ΚΑΙ ΠΕΡΙΣΣΟΤΕΡΕΣ ΜΕΤΑΒΛΗΤΕΣ

Σε αυτήν την υποενότητα, παρουσιάζεται η γενίκευση της μεθόδου του Dixon για δύο ή και περισσότερες μεταβλητές. Έστω λοιπόν ένα σύνολο $n+1$ πολυωνύμων με n μεταβλητές:

$$F = \{p_1(x_1, x_2, \cdots, x_n), \cdots, p_{n+1}(x_1, x_2, \cdots, x_n)\}$$

Όπου η ποσότητα deg_{max} αποτελεί τον μέγιστο βαθμό των παραπάνω πολυωνύμων.
Με τον ίδιο τρόπο με την προηγούμενη ενότητα σχηματίζουμε το πολυώνυμο του Dixon. Η ορίζουσα Δ του πίνακα αντικατάστασης για δύο ή παραπάνω μεταβλητές είναι η εξής:

$$\Delta\left(x_1, x_2, \cdots, x_n, a_1, a_2, \cdots, a_n\right) = \begin{vmatrix} p_1\left(x_1, x_2, \cdots, x_n\right) & \cdots & p_{n+1}\left(x_1, x_2, \cdots, x_n\right) \\ p_1\left(a_1, x_2, \cdots, x_n\right) & \cdots & p_{n+1}\left(a_1, x_2, \cdots, x_n\right) \\ p_1\left(a_1, a_2, \cdots, x_n\right) & \cdots & p_{n+1}\left(a_1, a_2, \cdots, x_n\right) \\ \cdots & \cdots & \cdots \\ p_1\left(a_1, a_2, \cdots, a_n\right) & \cdots & p_{n+1}\left(a_1, a_2, \cdots, a_n\right) \end{vmatrix}$$

Η ορίζουσα έχει μέγεθος $(n+1) \times (n+1)$ και οι μεταβλητές $\alpha_1, \cdots, \alpha_n$ παίρνουν τη θέση των x_i στα πολυώνυμα p_i όπου $1 \leq i \leq n$.
Το πολυώνυμο Dixon σχηματίζεται ως εξής:

$$\delta\left(x_1, \cdots, x_n, a_1, \cdots, a_n\right) = \frac{\Delta\left(x_1, \cdots, x_n, a_1, \cdots, a_n\right)}{\left(x_1 - a_1\right) \cdots \left(x_n - a_n\right)}$$

Το πολυώνυμο Dixon είναι βαθμού $\left((n+1-i) \times deg_{max}\right) - 1$ στο a_i και $\left(i \times deg_{max}\right) - 1$ στο x_i όπου $1 \leq i \leq n$. Όπως και πριν κάθε κοινό μηδενικό του συνόλου F μηδενίζει το πολυώνυμο του Dixon, ανεξάρτητα των τιμών του a_i.
Έτσι οδηγούμαστε πάλι σε ένα σύνολο E εξισώσεων στο x_i όπου η κάθε μία αντιστοιχεί στους συντελεστές των δυνάμεων της a_i.Το ακριβές πλήθος του συνόλου των εξισώσεων είναι s. Όπου:

$$s = \prod_{i=1}^{n} \left((n+1-i) \times deg_{max_i} \right) = n! \times \prod_{i=1}^{n} deg_{max_i}$$

Επίσης έχουμε:

$$s = = n! \times \prod_{i=1}^{n} deg_{max_i} = \prod_{i=1}^{n} i \times deg_{max_i}$$

Όπου sείναι και το σύνολο των συντελεστών των δυνάμεων των $x_1, x_2, \cdots, x_n$ στις εξισώσεις του συνόλου E.

Με συμβολισμό πινάκων μπορούμε να πούμε ότι ο πίνακας D είναι ο $s \times s$ πίνακας συντελεστών του E. Τότε:

$$Ɇ \equiv D \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \\ {x_1}^2 \\ x_1 x_2 \\ {x_2}^2 \\ \vdots \\ {x_n}^2 \\ \vdots \\ \prod_{i=1}^{n} {x_i}^{i \times deg_{max} - 1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Εάν δούμε τον κάθε συντελεστή από τις δυνάμεις του x_1 (μαζί και τον ${x_1}^0$) σαν μία νέα μεταβλητή z_i, έτσι παίρνουμε ένα σύνολο E με s ομογενείς γραμμικές εξισώσεις:

$$E \equiv D \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_s \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Όπου D είναι ο πίνακας του Dixon και η ορίζουσα του αποτελεί την απαλείφουσα του. Όμοια με την προηγούμενη υποενότητα καταλήγουμε ότι αν υπάρχει κοινό μηδενικό στο σύνολο των πολυωνύμων F τότε υπάρχει λύση για το σύνολο $Ɇ$. Δηλαδή, εάν το σύνολο των πολυωνύμων F έχει κοινό μηδενικό τότε σημαίνει ότι το Eέχει μη τετριμμένη λύση άρα και η ορίζουσα του (η απαλείφουσα του δηλαδή) μηδενίζεται [5,6]. Παρακάτω παρουσιάζεται ένα παράδειγμα για δύο ή περισσότερες μεταβλητες.

ΠΑΡΑΔΕΙΓΜΑ 4

Έστω τα τρια πολυώνυμα $f, g,$ και h όπου:

$$f = 4 - 2x - 2y + xy = (x-2)(y-2)$$
$$g = 12 - 6x - 2y + xy = (x-2)(y-6)$$
$$h = 24 - 6x - 4y + xy = = (x-4)(y-6)$$

Φαίνεται εύκολα από την παραγοντοποίηση ότι υπάρχει κοινή λύση και αυτή είναι , μένει λοιπόν να αποδείξουμε την ύπαρξη της μέσα από τις μεθόδους του Dixon. Το πολυώνυμο είναι το εξής:

$$P = -96 + 48a + 16y - 8ay$$

Όπου το a είναι βοηθητική μεταβλητή, εξισώνοντας τους συντελεστες των δυνάμεων οδηγούμαστε στον πίνακα του DIxon:

$$\begin{bmatrix} -96 & 16 \\ 48 & -8 \end{bmatrix}$$

Η ορίζουσα του παραπάνω πίνακα είναι 0, που αποδεικνύει την ύπαρξη κοινής ρίζας.

2.3 ΓΕΝΕΤΙΚΑ ΠΟΛΥΩΝΥΜΑ

Ο Dixon λοιπόν απέδειξε ότι οι μέθοδοι του μπορούν να γενικευτούν για οποιαδήποτε $n+1$ γενετικά πολυώνυμα βαθμου n με n αγνώστους. Με τον όρο αυτό εννοούμε ότι οι συντελεστές του πολυωνύμου είναι ανεξάρτητες μεταξύ τους παράμετροι. Στο [6] μας παρουσιάζεται ένας απλουστευμένος ορισμός για τα γενετικά πολυώνυμα βαθμου n:

Έστω λοιπόν ένα σύνολο από n ακεραίους $[k_1, \cdots, k_n]$ έτσι ώστε τα πολυώνυμα $p_1, \cdots, p_{n+1}$ να είναι του τύπου:

$$p_j = \sum_{i_1=1}^{k_1} \cdots \sum_{i_n=1}^{k_n} a_{j,i_1,\cdots,i_n} x_1{}_1^{j} \cdots x_{n}{}_{n}^{i}, \quad 1 \le j \le n+1$$

όπου τα a αποτελούν διακριτές απροσδιόριστες μεταβλητές.

Η εφαρμογή των μεθόδων του Dixon σε μη γενετικά πολυώνυμα μπορεί να μας οδηγήσει σε ορισμένα προβλήματα. Η διακρίνουσα λοιπόν είναι δυνατόν να μη μπορεί να δώσει αναγκαία συνθήκη για την ύπαρξη κοινών μηδενικών στο σύνολο Fδιότι ο πίνακας E μπορεί να μην έχει μη τετριμμένη λύση , ακόμα και αν υπάρχουν τα κοινά μηδενικά στο F.

Αυτό συμβαίνει γιατί ο πίνακας Dixon μπορεί να μην περιέχει την στήλη που αντιστοιχούν οι συντελεστές των μονωνύμων $\left(x^0{}_1 , \cdots , x^0{}_n \right)$, και έτσι αν το F έχει τη μοναδική κοινή λύση $x_1 = 0 , \ldots , x_n = 0$ τότε ο πίνακας E έχει μόνο την τετριμμένη λύση. Επίσης, ακόμα και αν ο πίνακας έχει την στήλη των μονωνύμων μπορεί να είναι ιδιόμορφος. Τέλος, η χειρότερη περίπτωση είναι ο πίνακας Dixon να μην είναι τετραγωνικός όπου εκεί είναι αδύνατος ο υπολογισμός της διακρίνουσας. Στην επόμενη υποενότητα θα ασχοληθούμε με την λύση των παραπάνω προβλημάτων.

2.4 Η ΠΡΟΣΕΓΓΙΣΗ ΤΩΝ KAPUR SAXENA YANG ΤΟ 1994

Οι KSY απέδειξαν και παρείχαν για τις παραπάνω περιπτώσεις που δεν μπορούμε να χρησιμοποιήσουμε τις συμβατικές μεθόδους του Dixon τόσο έναν αλγόριθμο - ο οποίος παίρνει ως είσοδο το $n+1$ πλήθους σύνολο των πολυωνύμων και τις n μεταβλητές που θέλουμε να απαλειφθούν - όσο και μια προϋπόθεση για το αν μπορούμε να τον εφαρμόσουμε. Η προϋπόθεση είναι η εξής: η στήλη του πίνακα Dixon που αντιστοιχεί στο μονώνυμο $\left[x_1{}^0, x_2{}^0, \cdots, x_n{}^0 \right]$ - όπου στη δική μας περίπτωση αποτελεί την πρώτη στήλη του πίνακα - δεν πρέπει να είναι γραμμικός συνδυασμός των υπόλοιπων στηλών [5,6]. Πριν προχωρήσουμε στα βήματα του αλγορίθμου, πρέπει να σημειωθεί ότι ο πίνακας M' είναι σε κλιμακωτή μορφή απαλείφοντας τον πίνακα DIxon M χωρίς την ιδιότητα του πολλαπλασιασμού των γραμμών του με κάποιο μη μηδενικό στοιχείο ενώ η ποσότητα D αποτελεί το γινόμενο των οδηγών του M'. Ένας πίνακας είναι σε κλιμακωτή μορφή εάν όλες οι μηδενικές του γραμμές βρίσκονται στο κάτω μέρος, το πρώτο μη μηδενικό στοιχείο μιας γραμμής βρίσκεται στα δεξιά του πρώτου μη μηδενικού στοιχείου της γραμμής από κάτω και κάτω από το πρώτο μη μηδενικό στοιχείο μιας γραμμής όλα τα στοιχεία είναι μηδέν. Ενώ οδηγοί είναι τα πρώτα μη μηδενικά στοιχεία του πίνακα σε κλιμακωτή μορφή στην κάθε γραμμή. Ο παρακάτω πίνακας είναι σε κλιμακωτή μορφή:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -2 & -3 & -4 \\ 0 & 0 & -2 & -4 \end{bmatrix}$$

Και τα στοιχεία στις θέσεις $[1,1]$,$[2,2]$ και $[3,3]$ είναι οι οδηγοί [7].

Ο αλγόριθμος λοιπόν παρουσιάζεται στα εξής απλά βήματα:

Εισοδος: Ένα σύνολο $n+1$ πολυωνύμων και τις n μεταβλητές που θέλουμε να απαλειφθούν
Βήματα:

1. Υπολογισμός του πίνακα DIxon M. Εάν η προϋπόθεση ισχύει προχωρούμε στο δεύτερο βήμα αλλιώς δεν μπορούμε να συνεχίσουμε.
2. Απαλοιφή του M χωρίς την ιδιότητα του πολλαπλασιασμού στην κλιμακωτή μορφή M'
3. Υπολογισμός του γινομένου D από τους οδηγούς του M'.

Έτσι λοιπόν εάν η προϋπόθεση ισχύει τότε το $D=0$ είναι η αναγκαία συνθήκη για την ύπαρξη κοινών μηδενικών [5,6]. Παρακάτω ακολουθούν ορισμένα παραδείγματα όπου δεν μπορούν να χρησιμοποιηθούν οι κλασικές μέθοδοι του DIxon και υπολογιζουμε την απαλείφουσα μέσω της προσσέγγισης των Kapur, Saxena και Yang.

ΠΑΡΑΔΕΙΓΜΑ 5

Έστω τα τρία πολυώνυμα $f, g,$ και h όπου:

$$f = x^2 + axy - y + b$$
$$g = xy + (a+b)\,y$$
$$h = (b+2a)\,x + y$$

Τα παραπάνω πολυώνυμα είναι μη γενετικά καθώς οι συντελεστές δεν είναι ανεξάρτητοι μεταξύ τους. Μπορούμε να δούμε π.χ. ότι ο συντελεστης του y στο g επηρεάζει το συντελεστή του x στο h. Αρχικά, μετά τον υπολογισμό του πολυωνύμου οδηγούμαστε στον πίνακα Dixon όπου μπορούμε να δούμε ότι δεν είναι τετραγωνικός και άρα δεν μπορούμε να υπολογίσουμε την ορίζουσα.

$$\begin{bmatrix} -2a^2b - 3ab^2 - b^3 & b & 0 & 0 \\ -2ab - b^2 & 2a + 2a^3 + b + 3a^2b + ab^2 & 2a^2 + 3ab + b^2 & 0 \\ 0 & 1 & 2a + b & 0 \end{bmatrix}$$

Έτσι οδηγούμαστε στην ιδιότυπη απαλοιφή χωρίς πολλαπλασιασμό που αναφέραμε πριν. Ο πίνακας σε κλιμακωτή μορφή είναι:

$$\begin{bmatrix} -b(2a^2+3ab+b^2) & b & 0 & 0 \\ 0 & \dfrac{(2a^2+2a^4-b+3ab+5a^3b+b^2+4a^2b^2+ab^3)}{(a+b)} & 2a^2+3ab+b^2 & 0 \\ 0 & 0 & \dfrac{(2a^3+4a^5-2ab+3a^2b+12a^4b-b^2+ab^2+13a^3b^2+6a^2b^3+ab^4)}{(2a^2+2a^4-b+3ab+5a^3b+b^2+4a^2b^2+ab^3)} & 0 \end{bmatrix}$$

Η προϋπόθεση ισχύει εφόσον η πρώτη στήλη δεν είναι γραμμικός συνδυασμός των υπολοίπων. Οι οδηγοί βρίσκονται στις θέσεις $[1,1]$, $[2,2]$ και $[3,3]$ και το γινόμενό τους είναι το εξής μετά από παραγοντοποίηση:

$$D = -b(a^2+2a^4-b+ab+5a^3b+4a^2b^2+ab^3)(2a+b)^2$$

Το $D=0$ αποτελεί την αναγκαία συνθήκη για την ύπαρξη κοινών μηδενικών.

Όλα τα παραδείγματα αυτης της ενότητας έγιναν με τη βοήθεια των συναρτήσεων που υλοποιήθηκαν στο υπολογιστικό σύστημα Mathics. Στην επόμενη ενότητα θα γίνει αναλυτική αναφορά σε αυτές καθώς και στο ίδιο το υπολογιστικό συστημα.

ΚΕΦΑΛΑΙΟ 3

ΥΛΟΠΟΙΗΣΗ ΤΩΝ ΣΥΝΑΡΤΗΣΕΩΝ ΤΟΥ DIXON ΣΤΟ MATHICS

3.1 ΤΟ ΣΥΣΤΗΜΑ ΥΠΟΛΟΓΙΣΤΙΚΗΣ ΑΛΓΕΒΡΑΣ MATHICS

Το mathics είναι ένα γενικής χρήσης λογισμικό υπολογιστικής άλγεβρας ανοιχτού κώδικα. Σκοπός του είναι να αποτελεί μία δωρεάν και ελεύθερη εναλλακτική στο Mathematica® της Wolfram. Ο τρόπος δόμησης και η προγραμματιστική γλώσσα του Mathics στοχεύει να μοιάζει με το Mathematica®, δίχως όμως το Mathics να σχετίζεται ή να υποστηρίζεται από τη Wolfram. Ο όρος ελεύθερο λογισμικό, δεν σημαίνει μόνο ότι διατίθεται δωρεάν αλλά και ότι ο χρήστης είναι σε θέση να δει τι συμβαίνει "μέσα" στο πρόγραμμα, το οποίο είναι δομημένο σε έναν ελεύθερο και επεκτάσιμο Python πυρήνα, ώστε να κατανοήσει καλύτερα τους υπολογισμούς του.
Μερικά από τα πιο σημαντικά χαρακτηριστικά του Mathics είναι:

• Ότι αποτελεί μια λειτουργική προγραμματιστική γλώσσα,

• ‘Ένα σύστημα που κατευθύνεται από μοτίβα και κανόνες (patterns matching - rules applying),

• Ότι μπορεί να παρουσιάσει ρητούς και μιγαδικούς αριθμούς

• Οι ρουτίνες που χειρίζονται τις λίστες και όλες τις δομές δεδομένων,

• Το διαδραστικό γραφικό περιβάλλον διεπαφής χρήστη στον browser χρησιμοποιώντας το MathML(εκτός της γραμμής εντολών),

• Η δημιουργία γραφικών και η εμφάνιση τους στον browser χρησιμοποιώντας το SVG για 2d Γραφικά και WebGL για γραφικά 3D,

• Η εξαγωγή αποτελεσμάτων στο LATEX,

• Το ολοκληρωμένο σύστημα δοκιμών (test.py)

Απαραίτητες προϋποθέσεις και εγκατάσταση

Για την εκτέλεση του Mathics χρειάζεται έκδοση της Python 3 ή

μεταγενέστερη, καθώς οι εκδόσεις του τελευταίου χρόνου έχουν σταματήσει να υποστηρίζουν την Python 2. Οι σύγχρονες εκδόσεις της Python περιλαμβάνονται στις περισσότερες διανομές των Linux καθώς και στα Mac OS X, ένω στα λειτουργικά των Windows μπορεί εύκολα κανείς να κατεβάσει την Python από το επίσημο site της (https://www.python.org/). Αξίζει όμως να σημειωθεί ότι κύριος στόχος των προγραμματιστών του Mathics είναι οι διανομές των Linux - ιδίως τα Ubuntu και τα Debian - και τα Mac OS X. Τέλος, είναι αναγκαίες οι εξής βιβλιοθήκες πακέτων της Python: libsqlite3-dev, python-dev και python-setuptools. Η εγκατάσταση του είναι πολύ απλή. κατεβάζοντας την τελευταια έκδοση του Mathics από την σελίδα του στο github (https://github.com/mathics/Mathics) και εκτελώντας στο τερματικό την εντολή:

 # python3 setup.py install

Με την εγκατάσταση του Mathics, κατεβαίνουν και εγκαθίστανται οι εξής βιβλιοθήκες: sympy, mpmath, django, και pysqlite. Η sympy μάλιστα είναι αυτή που χρησιμοποιήθηκε περισσότερο στην ανάπτυξη των συναρτήσεων για τις απαλείφουσες του Dixon. Τέλος για την εκτέλεση του Mathics χρειάζεται μία απλή μόνο εντολή στο τερματικό:

#mathics

3.2 ΥΛΟΠΟΙΗΣΗ ΣΥΝΑΡΤΗΣΕΩΝ ΣΤΟ MATHICS

Η ενσωμάτωση συναρτήσεων και κλάσεων μέσα στον πυρήνα του Mathics είναι απλή υπόθεση. Μπορούμε να γράψουμε κώδικα σε ένα νέο module στο *mathics/builtin/* και να το προσθέσουμε στην λίστα των ενεργών modules στο *mathics/builtin/__init__.py* είτε να εισάγουμε κώδικα σε ένα ήδη υπάρχων module. Πολύ σημαντικές έννοιες για τον προγραμματισμό του mathics είναι οι εξής: *attributes, rules, apply* και *evaluation*. Μία λίστα από *attributes* μπορεί να ανατεθεί σε μία κλάση, το *attribute Protected* είναι αυτό που ανατίθεται από προεπιλογή. Μερικά από τα *attributes* που το Mathics παρέχει είναι τα *HoldAll, HoldAllComplete, HoldRest, Listable* και άλλα. Με την επιλογή *rules* μπορούμε να παραθέσουμε μία σειρά από κανόνες που θα πρέπει να εφαρμοστούν. Οι συναρτήσεις της Python, μέσα στις κλάσεις, που ξεκινούν από *apply* μετατρέπονται σε ενσωματωμένους κανόνες που μεταγλωτίζονται από τον πυρήνα του Mathics. Οι *apply* συναρτήσεις παίρνουν ως όρισμα τις ίδιες μεταβλητές με τους κανόνες ή τα μοτίβα (*patterns*) μαζί με ένα πρόσθετο αντικείμενο (*object*) αυτό του

evaluation. Αυτό το *object* είναι αναγκαίο για την επιστροφή μεταβλητών στην κονσόλα ή για την εκτύπωση μηνυμάτων από τον κώδικα της Python. Παρακάτω ακολουθεί ο κώδικας του *IF* μέσα από το εγχειρίδιο του Mathics.

```
class If ( Builtin ):

attributes = ['HoldRest ']
rules = {
'If[ condition_, t_]': 'If[ condition, t,
Null ]',
}
def apply_3 ( self, condition, t, f, evaluation
):
'If[ condition_, t_, f_]'
if condition == Symbol ('True '):
return t. evaluate ( evaluation )
elif condition == Symbol ('False '):
return f. evaluate ( evaluation )
def apply_4 ( self, condition, t, f, u,
evaluation ):
'If[ condition_, t_, f_, u_]'
if condition == Symbol ('True '):
return t. evaluate ( evaluation )
elif condition == Symbol ('False '):
return f. evaluate ( evaluation )
else:
return u. evaluate ( evaluation )
```

Κλάση IF στο Mathics

Όπου μπορούμε να δούμε την χρήση των *attributes, rules, apply* και *evaluation*.

3.3 ΑΠΑΛΕΙΦΟΥΣΕΣ ΤΟΥ DIXON ΣΤΟ MATHICS

Για τις ανάγκες την εργασίας έγινε η ενσωμάτωση των παρακάτω κλάσεων:

`DixonSub [polys, xlist, alist]` είναι η λιγότερη πολύπλοκη όλων. Σχηματίζει τον πίνακα αντικατάστασης όπως ακριβώς στην προηγούμενη ενότητα.

`DixonPolynomial [polys, xlist, alist]` δημιουργεί το πολυώνυμο του DIxon χρησιμοποιώντας τον πίνακα αντικατάστασης.

`DixonMatrix [polys, xlist, alist]` δίνει ως έξοδο τον πίνακα Dixon αντιστοιχώντας τους συντελεστές του πολυωνύμου.

`ClassicalDixonResultant [polys, xlist, alist]` σχηματίζει την απαλείφουσα του Dixon με τον συμβατικό τρόπο.

`DixonResultant [polys, xlist, alist]` σχηματίζει την απαλείφουσα με την προσέγγιση των Kapur Saxena και Young

`PreconditionQ [polys, xlist, alist]` δίνει ως έξοδο αν ισχύει ή όχι η προϋπόθεση για το αν μπορεί να γίνει χρήση της φόρμουλας των Kapur Saxena και Young

`GaussElimination [matrix]` κάνει απαλοιφή του πίνακα όχι όμως με τον κλασικό τρόπο του Gauss αλλά χωρίς την ιδιότητα του πολλαπλασιασμού.

Όλες εκτός από την τελευταία παίρνουν ως είσοδο τη λίστα πολυωνύμων που θέλουμε να διαπιστώσουμε εάν υπάρχει κοινή λύση, τη λίστα των αγνώστων - δηλαδή τις μεταβλητές ως προς τις οποίες θα γίνει η απαλοιφή του συστήματος - και τη λιστα των βοηθητικών μεταβλητών που θα χρειαστούν για τον πίνακα αντικατάστασης και τον σχηματισμό του πολυωνύμου Dixon. Ενώ η *GaussElimination* παίρνει ως είσοδο έναν πίνακα.

Για την σωστή υλοποίηση των κλάσεων, χρειάστηκε να δημιουργηθούν *apply* συναρτήσεις. Αυτό συμβαίνει επειδή οι συναρτήσεις πρέπει να ανήκουν σε διαφορετικές κλάσεις ώστε για να γίνεται η σωστή κλήση από την κονσόλα. Επιπλέον, μία κλάση δε γίνεται να δημιουργηθεί μέσα

σε μία άλλη και έτσι δεν υπάρχει πρόσβαση σε συνάρτηση μιας κλάση από μία άλλη. Για αυτόν τον λόγο, η μόνη επιλογή για να χρησιμοποιήσουμε συναρτήσεις πάνω από μία φορά, είναι δημιουργούμε τις συναρτήσεις έξω από τις κλάσεις. Στο *ΠΑΡΑΡΤΗΜΑ Α* που μπορεί κανείς να δει ολόκληρο τον κώδικα, μπορεί να δει τις εξής apply συναρτήσεις: *DixonSub_apply, DixonPolynomial_apply, DixonMatrix_apply, GaussElimination_apply* όπου όπως αναφέρθηκε καλούνται σε περισσότερες από μία κλάσεις. Τέλος, κάθε κλάση επικοινωνεί με την κονσόλα με τη συνάρτηση *apply,* που πρέπει να δηλώνεται και να καλείται μέσα στην κλάση, και μαζί με τα ορίσματα που παίρνει η κλάση της, παίρνει επιπροσθέτως ως ορίσματα τα *objects self* και *evaluation.*

Πολύ σημαντικές επίσης για την υλοποίηση των παραπάνω συναρτήσεων μέσα στις κλάσεις τους, είναι ορισμένες επιλογές που δίνει το Mathics στον προγραμματιστή. Η επιλογή.*leaves* επιτρέπει την δημιουργία μιας λίστας με αντικείμενα Python από αντικείμενα του Mathics. Επιπλέον η επιλογή *to_sympy* επιτρέπει στον προγραμματιστή να μετατρέψει ένα αντικείμενο Mathics σε ένα αντικείμενο SymPy. Τέλος, η χρήση της *from_sympy* που έχει την αντίστροφη χρήση με την *to_sympy,* έτσι ώστε να είναι δυνατή η επιστροφή των αποτελεσμάτων σε μορφή Mathics αντικειμένων. Παρακάτω ακολουθεί ένα απόσπασμα της συνάρτησης *DixonPolynomial_apply* όπου γίνεται χρήση των παραπάνω επιλογών.

```
def DixonPolynomial_apply(polys, xlist,alist):

    xsym = [i.to_sympy() for i in xlist.leaves]
    asym = [i.to_sympy() for i in alist.leaves]
    res = DixonSub_apply(polys, xlist, alist)

    if not isinstance(res, Expression):####checking if the
DixonSub_apply returned error
        return res
    rsym = res.leaves
    dixon_list = [j.leaves for j in rsym] ###two level
Mathics list unpacked in 2 level python-list
    dixon_sym = [[j.to_sympy() for j in i] for i in
dixon_list]
    dix_mat = sympy.Matrix(dixon_sym) ###2-level python list
to sympy.Matrix object
```

```
    for i in range(1, len(xsym) + 1):
   #####together has poor results in sympy so simplify was
used
        dix_mat[i,:] = sympy.simplify((dix_mat[i,:] -
dix_mat[i - 1,:]) / (xsym[i - 1] - asym[i - 1]))
    # print(rsym[0])

    det_m = sympy.simplify(sympy.det(dix_mat))
    det_m = sympy.together(det_m)
    det_m = from_sympy(det_m)
    det_m = cancel(det_m)
    return det_m
```

Συνάρτηση DixonPolynomial_apply στο Mathics

Αξίζει, τέλος, να σημειωθεί ότι για την απαλοιφή Gauss υλοποιήθηκαν μερικές ακόμη υποσυναρτήσεις. Οι *find_first_pivot* και *pivot_for* χρησιμοποιούνται για την εύρεση των οδηγών στους πίνακες. Η *SwapRow* είναι η συνάρτηση που επιτρέπει την ανταλλαγή της σειράς *i* με τη σειρά *j.* Η *AddRow* επιπλέον δίνει την δυνατότητα της αντικατάστασης της σειράς *i* με την ίδια την σειρά *i* προσθέτοντας ή αφαιρώντας άλλη σειρά ή πολλαπλάσιο άλλης σειράς.

ΚΕΦΑΛΑΙΟ 4
ΠΑΡΑΔΕΙΓΜΑΤΑ ΕΚΤΕΛΕΣΗΣ ΣΤΟ ΠΕΡΙΒΑΛΛΟΝ ΤΟΥ MATHICS

Σε αυτό το κεφάλαιο παρουσιάζονται ορισμένα παραδείγματα εκτέλεσης μέσα από την κονσόλα των Linux. Πιο συγκεκριμένα, το περιβάλλον εκτέλεσης έχει την διανομή Ubuntu 18.04, την έκδοση του Mathics 1.1, της Python 3.7 και τέλος του SymPy 1.4.
Τα γενικά χαρακτηριστικά του μηχανήματος είναι τα εξής: Η κύρια του μνήμη είναι 4 Gigabyte και ο επεξεργαστής που πραγματοποιούνται τα παραδείγματα εκτέλεσης είναι i5-7200U της Intel και χρονίζεται στα 2.50GHz.

ΠΑΡΑΔΕΙΓΜΑ 1

```
In[1]:= f=x^2-6*x+8 // Factor
Out[1]= (-4 + x) (-2 + x)

In[2]:= g=x^2-10*x+24 // Factor
Out[2]= (-6 + x) (-4 + x)

In[3]:= DixonSub[{f,g},{x},{X}]
Out[3]= {{(-4 + x) (-2 + x), (-6 + x) (-4 + x)}, {(-4 + X)
(-2 + X), (-6 + X) (-4 + X)}}

In[4]:= DixonPolynomial[{f,g},{x},{X}]
Out[4]= -4 (-4 + X) (-4 + x)

In[5]:= DixonMatrix[{f,g},{x},{X}]
Out[5]= {{-64, 16}, {16, -4}}

In[6]:= ClassicalDixonResultant[{f,g},{x},{X}]
Out[6]= 0

In[7]:= PreconditionQ[{f,g},{x},{X}]
Out[7]= The precondition is FALSE.

In[8]:= DixonResultant[{f,g},{x},{X}]
Out[8]= -64
```

Παράδειγμα 1

Τα πολυώνυμα f και g φαίνεται μετά την παραγοντοποίηση τους πως έχουν κοινή λύση ($x=4$),κάτι που επιβεβαιώνεται από την *ClassicalDixonResultant* που είναι ίση με μηδέν.Επίσης, η προσέγγιση των Kapur,Saxena και Yang δε μπορεί να εφαρμοστεί και να προσφέρει μία ικανή συνθήκη διότι όπως φαίνεται εξόφθαλμα από την *DixonMatrix* η δεύτερη στήλη του πίνακα αποτελεί γραμμικό πολλαπλάσιο της πρώτης,Το ίδιο επιβεβαιώνει,μάλιστα, και η *PreconditionQ* της οποίας το αποτέλεσμα είναι *False.*

ΠΑΡΑΔΕΙΓΜΑ 2

```
In[1]:= f=x^2-5*x+6 // Factor
```

```
Out[1]= (-3 + x) (-2 + x)

In[2]:= g=x^2-5*x+4 // Factor
Out[2]= (-4 + x) (-1 + x)

In[3]:= DixonPolynomial[{f,g},{x},{X}]
Out[3]= -2 (-5 + X + x)

In[4]:= DixonMatrix[{f,g},{x},{X}]
Out[4]= {{10, -2}, {-2, 0}}

In[5]:= ClassicalDixonResultant[{f,g},{x},{X}]
Out[5]= -4
```

Παράδειγμα 2

Το συγκεκριμένα πολυώνυμα δεν έχουν κοινή λύση γεγονός που επιβεβαιωνεται τόσο από την παραγοντοποίηση τους όσο και από την *ClassicalDixonResultant*.

Το επόμενα δύο παραδείγματα που ακολουθούν περιέχουν παραμετρικές μεταβλητές οι οποίες μπορούν να δώσουν πληροφορίες για τις πραγματικές λύσεις του συστήματος μέσα από την απαλείφουσα του.

ΠΑΡΑΔΕΙΓΜΑ 3

```
In[1]:= f=(x-3+a)(x-2)
Out[1]= (-3 + a + x) (-2 + x)

In[2]:= g=(x+2-a)(x+2*a)
Out[2]= (2 - a + x) (2 a + x)

In[3]:= DixonMatrix[{f,g},{x},{X}]
Out[3]= {{-12 - 22 a + 16 a ^ 2 - 2 a ^ 3, -6 + 6 a - 2 a ^
2}, {-6 + 6 a - 2 a ^ 2, 7}}

In[4]:= DixonResultant[{f,g},{x},{X}]//Factor
Out[4]= -2 (-5 + 2 a) (-4 + a) (1 + a) (3 + a)
```

```
In[5]:= Solve[DixonResultant[{f,g},{x},{X}]==0,a]
Out[5]= {{a -> -3}, {a -> -1}, {a -> 5 / 2}, {a -> 4}}
```

Παράδειγμα 3

Σε αυτή τη περίπτωση, η απαλείφουσα του Dixon δίνει τις ικανές συνθήκες για την παραμετρική μεταβλητή ώστε το σύστημα να έχει κοινή λύση. Συγκεκριμένα βρίσκοντας τα σημεία που μηδενίζει η απαλείφουσα $a=(-3,-1,5/2,4)$ μπορούμε να πάρουμε πληροφορίες για τις λύσεις του συστήματος $f(x)=g(x)=0$ που ισχύει για $(a,x)=(-3,6),(-1,2),(5/2,1/2),(4,2)$.

ΠΑΡΑΔΕΙΓΜΑ 4

```
In[1]:= f=(x-4*a)(y-a) // Expand
Out[1]= 4 a ^ 2 - a x - 4 a y + x y

In[2]:=  g=(x-2)(y+a-2) // Expand
Out[2]= 4 - 2 a - 2 x + a x - 2 y + x y

In[3]:= h=(x-a)(y-2) // Expand
Out[3]= 2 a - a y - 2 x + x y

In[4]:= DixonMatrix[{f,g,h},{x,y},{X,Y}]
Out[4]= {{32 a - 48 a ^ 2 + 18 a ^ 3 - 4 a ^ 4, -16 a + 22
a ^ 2 - 4 a ^ 3}, {-8 + 12 a - 6 a ^ 2 + 4 a ^ 3, 4 - 4 a -
2 a ^ 2}}

In[5]:= ClassicalDixonResultant[{f,g,h},{x,y},{X,Y}] //
Factor
Out[5]= 12 a ^ 2 (-1 + a) (-1 + 2 a) (-2 + a) ^ 2

In[6]:= Solve[ClassicalDixonResultant[{f,g,h},{x,y},
{X,Y}]==0,a]
Out[6]= {{a -> 0}, {a -> 1 / 2}, {a -> 1}, {a -> 2}}
```

Παράδειγμα 4

Όμοια με το προηγούμενο παράδειγμα η απαλείφουσα του Dixon μας δίνει πληροφορίες για τις κοινές λύσεις ενός συστήματος τριών πολυωνύμων. Συγκεκριμένα, λύνοντας την απαλείφουσα ως προς μηδέν παρατηρούμε πως το $f(x,y)=g(x,y)=h(x,y)=0$ λύνεται για $(a,x,y)=(0,0,2),(1/2,2,2),(1,1,1),(2,2,2)$

Στα επόμενα παραδείγματα γίνεται χρήση της φόρμουλας των Kapur, Saxena,Yang.

ΠΑΡΑΔΕΙΓΜΑ 5

```
In[1]:= f=x*y+x*z+x-z^2-z+y^2+y
Out[1]= x + y + x y + y ^ 2 - z + x z - z ^ 2

In[2]:= g=x^2+x*z-x+x*y+y*z-y
Out[2]= -x + x ^ 2 - y + x y + x z + y z

In[3]:= h=x^2+x*y+2*x-x*z-y*z-2*z
Out[3]= 2 x + x ^ 2 + x y - 2 z - x z - y z

In[4]:= ClassicalDixonResultant[{f,g,h},{x,y},{X,Y}]
Out[4]= 0

In[5]:= PreconditionQ[{f,g,h},{x,y},{X,Y}]
Out[5]= The precondition is probabilistically TRUE.

In[6]:=res= DixonResultant[{f,g,h},{x,y},{X,Y}] // Factor
Out[6]= -8 z (-1 + z) (2 + z) (-1 + 2 z) ^ 2

In[7]:= Solve[res==0,z]
Out[7]= {{z -> -2}, {z -> 0}, {z -> 1 / 2}, {z -> 1}}
```

Παράδειγμα 5

Στο πέμπτο παράδειγμα, η κλασική φόρμουλα του Dixon δε μπορεί να δώσει επαρκείς πληροφορίες διότι η ορίζουσα του πίνακα Dixon είναι μηδέν, κάτι που είναι αναμενόμενο αφού στο σύστημα τα πολυώνυμα δεν είναι γενετικά βαθμού n. Η προϋπόθεση ισχύει, έτσι μπορούμε να χρησιμοποιήσουμε την προσέγγιση των Kapur, Saxena και Yang. Παραγωγίζοντας την απαλείφουσα και βλέποντας που μηδενίζει είναι δυνατή η λύση του συστήματος, Οι ρίζες της απαλείφουσας είναι:

$z = -2,\ 0,\ \frac{1}{2},\ 1$

και οι λύσεις του συστήματος $f(x,y,z) = g(x,y,z) = h(x,y,z) = 0$ είναι οι εξής:

$$(x,y,z) = (3,-5,-2),(0,0,0),\left(\frac{1}{2},-\frac{3}{2},\frac{1}{2}\right),\left(\frac{1}{2},0,\frac{1}{2}\right)(0,-2,1)$$

ΠΑΡΑΔΕΙΓΜΑ 6

```
In[1]:= f=(a*x)^2-x*y*a-c;
In[2]:= g=2*a*x^2-x*y-a;
In[3]:= h=3*a*x+y-c;
In[4]:= ClassicalDixonResultant[{f,g,h},{x,y},{X,Y}]
Out[4]= Dixon Polynomial is not a non-empty square matrix,
try DixonResultant

In[5]:= DixonMatrix[{f,g,h},{x,y},{X,Y}]
Out[5]= {{0, -a ^ 2 + c, a ^ 3 - 2 a c, 0}, {4 a ^ 3 - 5 a
c, 0, -a ^ 2 c, 0}, {-a ^ 2 c, a ^ 2, 3 a ^ 3, 0}

In[6]:= PreconditionQ[{f,g,h},{x,y},{X,Y}]
Out[6]= The precondition is probabilistically TRUE.

In[7]:= DixonResultant[{f,g,h},{x,y},{X,Y}]//Factor
Out[7]= a ^ 4 (16 a ^ 4 - 40 a ^ 2 c + 25 c ^ 2 - a ^ 2 c ^
2 + c ^ 3)
```

Παράδειγμα 6

Στο τελευταίο παράδειγμα αυτής της ενότητας, παρουσιάζεται άλλη μια περίπτωση που οι συμβατικές μέθοδοι του Dixon δεν μπορούν να δώσουν πληροφορίες για τη λύση του συστήματος, αυτή τη φορά επειδή ο πίνακας δεν είναι τετραγωνικός. Έτσι, χρησιμοποιείται η προσέγγιση των Kapur, Saxena και Yang που μας οδηγεί σε μία έκφραση των παραμετρικών συντελεστών, όπου μηδενίζοντας την, παίρνουμε τις τιμές για τους οποίους υπάρχει κοινή λύση του συστήματος.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] BUSÉ, Laurent; ELKADI, Mohamed; MOURRAIN, Bernard. Generalized resultants over unirational algebraic varieties. Journal of Symbolic Computation, 2000, 29.4-5: 515-526.

[2] Cox, David A., John Little, and Donal O'shea. Using algebraic geometry. Vol. 185. Springer Science & Business Media, 2006.

[3] Chionh, Eng-Wee, Ming Zhang, and Ronald N. Goldman. "Fast computation of the Bezout and Dixon resultant matrices." Journal of Symbolic Computation 33.1 (2002): 13-29.

[4] Kapur, Deepak, and Tushar Saxena. "Comparison of various multivariate resultant formulations." Proceedings of the 1995 international symposium on Symbolic and algebraic computation. 1995.

[5] Kapur, Deepak, Tushar Saxena, and Lu Yang. "Algebraic and geometric reasoning using Dixon resultants." *Proceedings of the international symposium on Symbolic and algebraic computation*. 1994.

[6] Nakos, G., and R. M. Williams. "Elimination with the Dixon resultant." *Mathematica in education and research* 6 (1997): 11-21.

[7] Margalit, Dan, and Joseph Rabinoff. "Interactive Linear Algebra." *Georgia Institute of Technology* (2018).

[8] The Mathics Team. “Mathics. A free, light-weight alternative to Mathematica” (2016).

ΠΑΡΑΡΤΗΜΑ Α

Ο ΚΩΔΙΚΑΣ ΤΩΝ ΑΠΑΛΕΙΦΟΥΣΩΝ ΤΟΥ DIXON ΕΝΣΩΜΑΤΩΜΕΝΟΣ ΣΤΟ MATHICS

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Dixon functions
"""

from mathics.builtin.base import Builtin, PostfixOperator,
SympyFunction
from mathics.builtin.algebra import cancel,Cancel,Together
from mathics.builtin.linalg import Det
from mathics.core.expression import Expression, Integer,
Number
from mathics.core.convert import (
    sympy_symbol_prefix, SympyExpression, from_sympy)
from mathics.core.rules import Pattern
from mathics.core.numbers import dps
from mathics.builtin.scoping import dynamic_scoping
from mathics.builtin.strings import String
import sympy
import itertools
import random


##########The following functions are wrapped around their
respective claases. The "apply fucntion of each class
#####calls the respective function. This is done because
all the functions must belong to different classes in order
to
###be called from the console. Also one class cannot be
created inside an other so there is no access to functions
inside a
###class from another class. This leaves us with the
options to write the functions we are using more than once
outside the classes
```

```
########ALL THE FUNCTIONS HAVE AS INPUTS AND OUTPUTS
MATHICS OBJECTS!!!!!!!!!!!!!!!

##############################################################
######################
# (* The next two procedures compute the Dixon polynomial
*)
# (* (DixonPolynomial).
*)
# (* Arguments for DixonSub and DixonPolynomial consist of
a list of n+1      *)
# (* polynomials in n variables (polys), a list of the n
original variables   *)
# (* (xlist), and a list of the n auxiliary variables
(alist).                *)
##############################################################
######################


def DixonSub_apply(polys, xlist,alist):



    xleaf = xlist.leaves ####The .leaves option creates a
python-list of mathics objects out of a mathics-list of
mathics obejcts
    aleaf = alist.leaves
    polysleaf = polys.leaves.copy()####Copy is needed to
ensure the first line of the matrix remains the same

    xsym = [i.to_sympy() for i in xleaf]#####.to_sympy is
transforming a mathics object to a sympy object
    asym = [i.to_sympy() for i in aleaf]
    res_in = polysleaf.copy()
    res_out = [res_in.copy()]

    if (len(xleaf) + 1) != len(polysleaf):####Check for
square matrix  same as
If[Length[dix]===Length[Transpose[dix]]
        # evaluation.message('DixonSub','not_eq')
        return String("The variable list is wrong.")
```

```
    for i in range(len(xleaf)): ###for every variable in
x_list
        for j in range(len(polysleaf)):###for every
polynomial
            res_in[j] =
from_sympy(res_in[j].to_sympy().subs(xsym[i], asym[i]))
###subs is substituting the variables
            #####from_sympy is transforming a Sympy object
to a mathics object
        res_out.append(Expression('List', *res_in))####the
Expression command with the option list is buildng a
mathics
    ###list out of a python iterable (list, tuple etc)
    return Expression('List', *res_out)###since the output
is a 2d matrix (list within a list) we need to asseble each
rows
###first and the put as elements of the outer list object




def DixonPolynomial_apply(polys, xlist,alist):

    xsym = [i.to_sympy() for i in xlist.leaves]
    asym = [i.to_sympy() for i in alist.leaves]
    res = DixonSub_apply(polys, xlist, alist)

    if not isinstance(res, Expression):####checking if the
DixonSub_apply returned error
        return res
    rsym = res.leaves
    dixon_list = [j.leaves for j in rsym] ###two level
Mathics list unpacked in 2 level python-list
    dixon_sym = [[j.to_sympy() for j in i] for i in
dixon_list]
    dix_mat = sympy.Matrix(dixon_sym) ###2-level python
list to sympy.Matrix object

    for i in range(1, len(xsym) + 1):
   #####together has poor results in sympy so simplify was
used
        dix_mat[i, :] = sympy.simplify((dix_mat[i, :] -
```

```
dix_mat[i - 1, :]) / (xsym[i - 1] - asym[i - 1]))
    # print(rsym[0])

    det_m = sympy.simplify(sympy.det(dix_mat))
    det_m = sympy.together(det_m)
    det_m = from_sympy(det_m)
    det_m = cancel(det_m)
    return det_m

###(* DixonMatrix computes the classical Dixon Matrix.
def DixonMatrix_apply(polys, xlist,alist):

    res = DixonPolynomial_apply(polys, xlist, alist)
    if isinstance(res, String):
        return res

    xsym = [i.to_sympy() for i in xlist.leaves]
    asym = [i.to_sympy() for i in alist.leaves]
    xasym = asym + xsym ###concatenating the a,x lists
    dix_pol = res.to_sympy()
    if dix_pol == sympy.sympify(0):###check for zero
        return from_sympy(dix_pol)
    pol = sympy.poly(dix_pol, xasym)##create sympy.poly
object
    maxdegvec = sympy.degree_list(pol, xasym)###this is
MaxDegVec function in mathematica. No function was written
since
    ###its built in
    # print(maxdegvec)
    res_in = []
    res_out = []
    mat_dim_col = 1
    mat_dim_row = 1
    for i in range(len(maxdegvec) // 2):
        mat_dim_row = mat_dim_row * (maxdegvec[i] +
1)###Number of rows in the matrix equal to the monomials
that can be
        #formed from the a list
    for i in range(len(maxdegvec) // 2,len(maxdegvec) ):
        ###Number of collums in the matrix equal to the
monomials that can be
        # formed from the x list
```

```
        mat_dim_col = mat_dim_col * (maxdegvec[i] + 1)
    A = sympy.zeros(mat_dim_row,mat_dim_col)
    #####The function lesslists is done by the command
range which give a list with all the numbers less than the
argument
    ####The itertools.products is essentially the cartesian
product for the range lists that correspongd to different
variables
    ####For example if the max degree in a,b is 1,2 we ll
get two lists [0,1],[0,1,2] and the cartesian produst will
be
    ###the couples [0,0],[0,1],[0,2],[1,0],[1,1],[1,2] this
are the half-monomials as they represent the exponets for
the
    ###a-list.
    ###The same proccess is repeated with the x list to get
another half set of exponents
    for n, i in enumerate(itertools.product(*[range(i + 1)
for i in maxdegvec[:len(maxdegvec) // 2]])): ##a-list exp
        for m,j in enumerate(itertools.product(*[range(i +
1) for i in maxdegvec[len(maxdegvec) // 2:]])):##x-list exp
        # res_in.append()
            monomial = sympy.sympify(1)
            for e1, x1 in zip(list(i)+list(j),
xasym):###this loop builds the monomials after the two half
sets are joined
                monomial = monomial * x1 ** e1
            res_out.append(pol.coeff_monomial(monomial))
            A[n,m] = pol.coeff_monomial(monomial) ##No
problem with the constant term as with mathematica
    #####This if-statement is used only for a case where
the matrix is a row vector or a col vector or a scalar
    ###In that case the "from_sympy" command will not
return a 2d matrix but MAthematica alwyas gives 2d
matrix(list of lists)
    if int(mat_dim_col)==1 | int(mat_dim_row)==1:
        res_in=from_sympy(A[0,0])
        res_out=[Expression('List', *[res_in])]
        return Expression('List', *res_out)
    return from_sympy(A)
```

```
def GaussElimination_apply(A):
    # (*Finds current pivot.First pivot in each sub-
block.*)
    def find_first_pivot(A):



        ind = 0
        if A == sympy.zeros(*A.shape):
            return -1, -1 ###zero matrixs returns -1

        while A.T[ind] == sympy.sympify(0):###if we scan
the matrix with one index it scans row-wise,so we eliminate
the
            ###need for ZeroVectorQ function
            ind += 1
            ###the result of the integer division ind //
(A.shape[0]) gives us the row of the index
            #####The remainder of the above ind %
(A.shape[0]) gives us the collumn
            ###The row and collum are interchanged in the
return statement because the index corresponds to the
TRANSPOSE
        return [ind % (A.shape[0]), ind // (A.shape[0])]
    #(* Swaps rows Rowi and Rowj .
    def SwapRow(A, i, j):

        if i == j:
            return A
        row_i = A[i, :]
        row_j = A[j, :]
        A[j, :] = -row_i
        A[i, :] = row_j
        return A

    ##(*Replace Rowj with: Rowj + scal * Rowi.*)
    def Addrow(A, j, i, scal):

        A[j, :] = sympy.expand(A[j, :] + scal * A[i, :])
```

```
        return A
    ##(* The forward pivot.
    def pivot_for(A, i, j):

        if A[i, j] == sympy.sympify(0):
            return A
        for k in range(i + 1, A.shape[0]):
            A = Addrow(A, k, i, -A[k, j] / A[i, j])
        return A

    A_l = A.leaves
    A_ll = [j.leaves for j in A_l]
    A_s = [[j.to_sympy() for j in i] for i in A_ll]

    if len(A_s) == 1:
        return A
    Am = sympy.Matrix(A_s)

    Amt = Am ####AMt is the matrix which will be reduced in
size as we procced
    top = -1 ###The variables are initialized at -1 beacuse
python is ZERO-BASED and MATHEMATICA is ONE-BASED!!!!!!!
    t1 = 0
    t2 = -1
    fp = find_first_pivot(Amt)###Python cant assign in a
condition as mathematica so the first pivot is calculated
    ###rigth before the while condition check. I
    # Am=SwapRow(Am, 0, 1)
    while Amt.shape[0] != 1 and fp[0] != -1:
        fp[0] = fp[0] + top + 1###+1 used because python is
ZERO BASED
        fp[1] = fp[1] + t2 + 1
        top += 1
        Am1 = SwapRow(Am, top, fp[0])
        Am = Am1
        Am1 = pivot_for(Am, top, fp[1])
        Am = Am1
        Amt = Am[top + 1:, fp[1] + 1:]###no need for
MinorColBlock function
        t1 = fp[0]
        t2 = fp[1]
        fp = find_first_pivot(Amt)###Python cant assign in
```

```
a condition as mathematica so the first pivot is calculated
    ###rigth before the while condition check.

    return from_sympy(sympy.simplify(Am))

class DixonSub(Builtin):
    ###every class comunicates with the console with the
function apply which must stated as below with the required
    ##self and evaluation objects and the input arguments

    ret_top='True'
    def apply(self, polys, xlist,alist, evaluation):
        'DixonSub[polys_, xlist_,alist_]'
        ###the line above is also needed for correct
calling of the class from the console
        return DixonSub_apply(polys, xlist,alist)

class DixonPolynomial(Builtin):
    def apply(self, polys, xlist,alist, evaluation):
        'DixonPolynomial[polys_, xlist_,alist_]'
        return DixonPolynomial_apply(polys, xlist, alist)


class DixonMatrix(Builtin):
    def apply(self, polys, xlist,alist, evaluation):
        'DixonMatrix[polys_, xlist_,alist_]'
        return DixonMatrix_apply(polys, xlist,alist)


# (* ClassicalDixonResultant computes the Dixon Resultant,
i.e.,             *)
# (* the determinant of the Dixon Matrix.
*)
#
# (* Note:  The number of polynomials, polys, MUST be one
more than the      *)
# (* number of variables, vars, otherwise the determinant
cannot be computed.*)

class ClassicalDixonResultant(Builtin):
    def apply(self, polys, xlist,alist, evaluation):
        'ClassicalDixonResultant[polys_, xlist_,alist_]'
```

```
        ####this is not written in a wrapper function as it
is not used anywhere else

        res=DixonMatrix_apply(polys, xlist, alist)
        if isinstance(res, String):
            return res
        if res.to_sympy() == sympy.sympify(0):
            return res
        rsym = res.leaves
        dixon_list = [j.leaves for j in rsym]

        dixon_sym = [[j.to_sympy() for j in i] for i in
dixon_list]
        dix_mat = sympy.Matrix(dixon_sym)
        if dix_mat is None or dix_mat.cols != dix_mat.rows
or dix_mat.cols == 0:
            return String("Dixon Polynomial is not a non-
empty square matrix , try DixonResultant")
        return from_sympy(sympy.det(dix_mat))



class GaussElimination(Builtin):

    def apply(self, A, evaluation):
        'GaussElimination[A_]'
        return GaussElimination_apply(A)


class DixonResultant(Builtin):

    def apply(self, polys, xlist,alist, evaluation):
        'DixonResultant[polys_, xlist_,alist_]'
        res=DixonMatrix_apply(polys, xlist, alist)
        if isinstance(res, String):
            return res
        if res.to_sympy() == sympy.sympify(0):
            return res
        rsym = res.leaves
        dixon_list = [j.leaves for j in rsym]

        dixon_sym = [[j.to_sympy() for j in i] for i in
```

```
dixon_list]
        dix_mat = sympy.Matrix(dixon_sym)
        m, n = dix_mat.shape
        rows = [i for i in range(m) if any(dix_mat[i, j] !=
0 for j in range(n))]#find non zero row indices
        cols = [j for j in range(n) if any(dix_mat[i, j] !=
0 for i in range(m))]#find non zero collumn indices
        dix_mat = dix_mat[rows, cols]###keep only the non
zero, no need for DeleteZeroRows,DeleteZeroColumns


        res =
GaussElimination_apply(from_sympy(dix_mat))###we need to
transform back to the Mathics object
        ###the function returns mathics object the we need
to unwrap and transform
        rsym = res.leaves
        dixon_list = [j.leaves for j in rsym]

        dixon_sym = [[j.to_sympy() for j in i] for i in
dixon_list]
        dix_mat = sympy.Matrix(dixon_sym)
        m, n = dix_mat.shape
        rows = [i for i in range(m) if any(dix_mat[i, j] !=
0 for j in range(n))]
        cols = [j for j in range(n) if any(dix_mat[i, j] !=
0 for i in range(m))]
        dix_mat = dix_mat[rows, cols]

        ####this is the implementaion of the
ProductLeadingEntries

        dix_list = []  ###since the rows will have
different amount of elements we need to use nested list not
matrix
        for i in range(dix_mat.shape[0]):
            dix_line=[]##the second level list
            for j in range(dix_mat.shape[1]):
                if dix_mat[i,j]!
=sympy.sympify(0):###DeleteCases[m, 0, 2] we keep only the
non zero elements of every line
                    dix_line.append(dix_mat[i,j])
```

```
            if len(dix_line)>0:###DeleteCases[mm, {}] we
keep only the non-empty lines and append them to the outer-
list
                dix_list.append(dix_line)
        prod=1
        for i in dix_list:###for every-list-line we muliply
the first element
            prod=prod* i[0]
        return from_sympy(prod)


class PreconditionQ(Builtin):

    def apply(self, polys, xlist,alist, evaluation):
        'PreconditionQ[polys_, xlist_,alist_]'
        res = DixonMatrix_apply(polys, xlist, alist)
        if isinstance(res, String):
            return res
        if res.to_sympy() == sympy.sympify(0):
            return res
        rsym = res.leaves
        dixon_list = [j.leaves for j in rsym]

        dixon_sym = [[j.to_sympy() for j in i] for i in
dixon_list]
        dix_mat = sympy.Matrix(dixon_sym)
        parlist=dix_mat.free_symbols####equivalent to
mathematica Variables

        for i in parlist:

dix_mat.subs(i,random.randint(1,100))###substite each
variable with random integer
        dix_mat=sympy.simplify(dix_mat)
        dix_mat=dix_mat.rref()[0]###compute RowReduce
        if any(dix_mat[0, i] != 0 for i in
range(1,dix_mat.shape[1])):###check the first row after the
first element
            return String("The precondition is FALSE.")
        return String("The precondition is
probabilistically TRUE.")
```