Application, in Python/Sympy, of Van Vleck's method for the computation of Sturm sequences with matrix triangulation

Diploma Thesis
by
Angelou Areti-Christina

University Of Thessaly
Department of Computer and Electrical Engineering

Supervisor: Akritas Alkiviadis
Co-supervisor: Stamoulis Georgios

Volos, 2014

Abstract.

Van Vleck proposed a very effcient method to compute the Sturm sequence of a polynomial $p(x) \epsilon \mathbb{Z}[x]$ by triangularizing one of Sylvester's matrices of $p(x)$ and its derivative $p'(x)$. That method works fine only for the case of complete sequences provided no pivots take place. Later, A. J. Pell and R. L. Gordon pointed out this "weaknes" in Van Vleck's theorem, rectified it but did not extend this method, so that it also works in the cases of complete Sturm sequences with pivot, and incomplete Sturm sequences. In 2014 Akritas, Malaschonok,Viglas, modified slightly the formula of the Pell-Gordon Theorem and presented a generalized triangularization method, the VanVleck-Pell-Gordon method, which correctly computes in $\mathbb{Z}[x]$ polynomial Sturm sequences,both complete and incomplete.

In this thesis both methods, Van Vleck's and the generalized triangularization method are explained through detailed examples, where TeXmacs has been used as interface so as all the matrice's triangularizations and the computations needed for the computaion of Sturm sequences to be done with the use of Python. In this thesis is also presented the implementation of both of these methods in Python version(2.0.7) and are available for use.

Acknowledgments.

Foremost I would like to express my sincere gratitude to my supervisor, professor Alkiviadis Akritas, for his overall guidance, support and motivation in all the time of implementing this thesis and especially for his confidence in me. I would like to express my deeply appreciation to him for the time he disposed.

Moreover I would also like to sincerely thank my co-supervisor, professor Georgios Stamoulis for his willingness to supervise my thesis and his help on this and other projects through my studies.

Last but not least, I would like to express my special thanks to my family and Marios for their endless support and love that showed to me all these years. Their encouragement and trust led me this far.

Table of contents

1. Introduction to Python

Python is a widely used general-purpose, high level programming language. Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI(Centrum Wiskunde & Informatica) in the Netherlands. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. Python has a large and comprehensive standard library, commonly cited as one of Python's greatest strength, providing tools suited to many tasks. This is deliberate and has been described as a "batteries included" Python philosophy. For Internet-facing applications, a large number of standard formats and protocols (such as MIME and HTTP) are supported. Modules for creating graphical user interfaces, connecting to relational databases, pseudorandom number generators, arithmetic with arbitrary precision decimals, manipulating regular expressions, and doing unit testing are also included.

Python interpreters are available for installation on many operating systems, allowing Python code execution on a wide variety of systems. Most Python implementations (including CPython) can function as a command line interpreter, for which the user enters statements sequentially and receives the results immediately. In short, Python acts as a shell. Other shells add capabilities beyond those in the basic interpreter, including IDLE and IPython. While generally following the visual style of the Python shell, they implement features like auto-completion, retention of session state, and syntax highlighting. In addition to standard desktop Python IDEs (integrated development environments), there are also browser-based IDEs, Sage (intended for developing science and math-related Python programs), and a browser-based IDE and hosting environment, PythonAnywhere.

Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution(late binding), which binds method and variable names during program execution.

Python is intended to be a highly readable language. It is designed to have an uncluttered visual layout, frequently using English keywords where other languages use punctuation. Furthermore, Python has a smaller number of syntactic exceptions and special cases than C or Pascal.[16]

1.2 The Python library Sympy

Sympy is a Python library for symbolic computation. It provides computer algebra capabilities either as a standalone application, as a library to other applications, or live on the web as Sympy Live or Sympy Gamma. Sympy is trivial to install and to inspect because is written entirely in Python and because it does not depend on any additional libraries. This ease of access combined with a simple and extensible code base in a well known language make Sympy a computer algebra system with a relatively low barrier to entry.

The Sympy library is split into a core with many optional modules. Currently, the core of Sympy has around 260,000 lines of code (also includes a comprehensive set of self-tests) and its capabilities include features ranging from basic symbolic arithmetic to calculus, algebra, discrete mathematics and quantum physics. It is capable of formatting the result of the computations as LaTeX code.

Sympy is free software and is licensed under New BSD license. The lead developers are Ondřej Čertík and Aaron Meurer.[17]

1.3 Installation of Sympy

The Sympy CAS can be installed on virtually any computer with Python 2.6 or above. Sympy does not require any special Python modules. The recommended method of installation is directly from the source files. Alternatively, executables are available for Windows, and some Linux distributions have Sympy packages available. In the following link there are clear instructions for the installation of Sympy: http://docs.sympy.org/latest/install.html

Although Sympy does not have any hard dependencies, many nice features are only enabled when certain libraries are installed. For example, without Matplotlib, only simple text-based plotting is enabled. With the IPython notebook or qtconsole, we can get nicer LATEX printing by running `init_printing()`. An easy way to get all these libraries in addition to Sympy is to install Anaconda, which is a free Python distribution from Continuum Analytics that includes SymPy, Matplotlib, IPython, NumPy, and many more useful Python packages for scientific computing.In the following link the reader can download and install Anaconda: http://continuum.io/downloads

Alternatively, TeXmacs can also be used as an interface in order to run Pyhton and Sympy. TeXmacs is a free editing platform with special features for scientists. In the following link there are intructions for the installation of TeXmacs http://www.texmacs.org/tmweb/download/windows.en.html.

In this thesis I wrote my report with the TeXmacs editing platform and I used Anaconda with Python 2.7 for the implemantation of the two methods of computing Sturm sequences.

2. Introduction to the theoretical background of computing Sturm sequences

The Sturm sequence of a polynomial $p(x) \epsilon \mathbb{Z}[x]$ or $p(x) \epsilon \mathbb{Q}[x]$, of degree $n \geqslant 2$, is the sequence of functions $f_0(x), f_1(x), \ldots, f_k(x), k \leqslant n$, where $f_0(x) = p(x)$, $f_1(x) = p'(x)$, and, for $2 \leqslant j \leqslant k$, $f_j(x)$ is the negative remainder obtained on dividing $f_{j-2}(x)$ by $f_{j-1}(x)$.

In other words, the Sturm sequence of $p(x)$ is obtained by negating the remainders obtained in the process of finding the greatest common divisor of $p(x)$ and $p'(x)$ using the Euclidean algorithm.

If $k = n$, the Sturm sequence is called complete, whereas if $k < n$, it is called incomplete.

We see that obtaining polynomial remainders is the major operation in computing Sturm sequences. The most widely known and commonly used methods to compute these remainders is to use either polynomial pseudo-divisions (explained below) in $\mathbb{Z}[x]$ or regular polynomial divisions in $\mathbb{Q}[x]$.

For example, the Sturm sequence in $\mathbb{Z}[x]$ of $p(x) = x^3 + 3x^2 - 7x + 7$ is obtained by the function sturm of Xcas(another freely available computer algebra system)

```
> sturm( x^3 + 3x^2 - 7x + 7 )[1]
                    [[1,3,-7,7], [3,6,-7], [60,-84], -2912]
```

where to obtain the first remainder, $60x - 84$, we had to premultiply the divident times 3^2, that is, times the leading coefficient of the divisor raised to the power $\deg(p) - \deg(p') + 1$.

In $\mathbb{Q}[x]$ the sequence is obtained using the function sturm of Sympy

```
Python] from sympy import *
Python] x= var('x')
Python] sturm( x**3 + 3*x**2 -7*x + 7 )
    [x**3 + 3*x**2 - 7*x + 7, 3*x**2 + 6*x - 7, 20*x/3 - 28/3, -182/25]
```

In 1840 Sylvester discovered sylvester1, the most widely known and used form of the two matrices that bear his name, and used it to compute in $\mathbb{Z}[x]$ the resultant of two polynomials $p(x)$, $q(x)$ along with the coefficients of the polynomial remainders obtained by applying Euclid's algorithm on $p(x), q(x)$ [12]. The coefficients of the polynomial remainders were obtained as determinants of submatrices, subresultants, of `sylvester1` are the smallest possible without introducing rationals and without computing (integer) greatest common divisors.

In 1853 Sylvester discovered the little known matrix `sylvester2` and used it to compute in $\mathbb{Z}[x]$ the coefficients of the polynomial remainders obtained by applying Sturm's algorithm on $p(x), p'(x)$ [13],[4]. Again, the coefficients of the modifed "Euclidean" polynomial remainders were obtained as determinants of submatrices, modified subresultants, of `sylvester2` are the smallest possible without introducing rationals and without computing (integer) greatest common divisors.

Sylvester's result of 1853 is valid only for complete Sturm sequences, while the case of incomplete Sturm sequences remained open, since the signs of the coefficients could not be correctly computed. Analogous observation was also made by Van Vleck in 1900 and is stated as Theorem 1 in this thesis.

Additionally, for complete Sturm sequences in $\mathbb{Z}[x]$, Van Vleck presented in 1900 a theorem, Theorem 2 in this thesis, and a computational method for computing the coefficients of the polynomial remainders by triangularizing Sylvester's matrix `sylvester2` of $p(x)$ and $p'(x)$. In his method VanVleck cleverly takes advantage of the special form of `sylvester2` and successively triangularizes matrices of only 3 rows, thus making his method extremely fast and suitable even for hand computations [14].

However, Van Vleck's method computes the correct sign of the coefficients only for complete Sturm sequences, when no pivot occurs in the triangularization process. In all other cases the sign of the coefficients may not be correct. This was observed by Pell and Gordon ([11], p. 193) and they presented a theorem, Theorem 3 in this thesis, to correctly compute the sign of the coefficients of the Sturm remainders in all cases.

In their work, Pell and Gordon compute in $\mathbb{Q}[x]$ the coeffcients of the polynomials in complete or incomplete Sturm sequences as modified subresultants of `sylvester2` divided by appropriate powers of the leading coefficients of the remainders, a complete example demonstrating their theorem can be found elsewhere [7]. However, they did not generalize Van Vleck's triangularization method to work for incomplete sequences.

In 1988, not being aware of the 1917 paper by Pell and Gordon, Akritas extended Van Vleck's method for generalized polynomial remainder sequences [2]. He used the Dodgson-Bareiss integerpreserving triangularization method but was not able to compute the exact signs of the polynomials in incomplete sequences [25, 24]. An attempt to resolve this issue was undertaken, in 1994, by Akritas, E. K. Akritas and Malaschonok but the "sign problem" remained elusive [5], despite the fact that improvements were made regarding the computational implementation [6].

Recently in 2014, in their paper Akritas, Malaschonok, Viglas[1], solved the "sign problem" and present a generalized triangularization method, the VanVleck-Pell-Gordon method, which exactly computes the sign of the polynomials in Sturm sequences, both complete and incomplete. Their breakthrough is due to the theorem by Pell and Gordon ([11], pp. 190, 193), and it came after Vigklas discovered their work in the scientific databases.

This thesis depends on the theory background stated on paper of Akritas, Malaschonok,Viglas[1], which the reader can consult for more details and examples.

The rest of the paper is organised as follows:

In Section 3 there is a review of the theoretical background of Van Vleck's method, a presentation of various aspects of the triangularization method along with a detailed example of the Van Vleck's triangularization method for computing complete Sturm sequences, where TeXmacs has been used as interface to run Python throughout the example, and the example follows the implementation of this method in Python with the usage of Sympy library.

In Section 4 there is a review of the Pell-Gordon Theorem ([11], pp. 190, 193) along with the modification of it, which is incorporated into Van Vleck's triangularization method. Also in this section I present a detailed example of the generalized triangularization method for computing Sturm sequences of any kind, where also TeXmacs has been used as interface to run Python, followed with the implementation in Python with the usage of Sympy library.

3. Van Vleck's Theorem and the Triangularization Method for Complete Sturm Sequences in $\mathbb{Z}[x]$

In this section, we need to introduce the notion of the resultant (and subresultants) of two polynomials, these polynomials will be $p(x)$ and its derivative $q(x)=p'(x)$, both in $\mathbb{Z}[x]$.

3.1 Polynomial Remainders with Matrix Triangularization

Van Vleck's method is based on the fact that polynomial remainders can be computed by triangularizing a special matrix. If the divident is $p(x)=a_nx^n+\ldots+a_0$, of degree n, and the divisor is $q(x)=A_mx^m+\ldots+A_0$, of degree m, $m<n$, then the dimension of the matrix M to be triangularized is $(n-m+2)\times(n+1)$ and its rows are listed bellow:

$$M=[[A_m,\ldots,A_0,0,\ldots,0],[0,A_m,\ldots,0,\ldots,0],\ldots,[0,\ldots,0,A_m,\ldots,A_0],[a_n,\ldots,a_0]].$$

In M, the first $n-m+1$ rows consist of the coefficients of $q(x)$ – shifted sequentially to the right – and the last row consists of the coefficients of $p(x)$. After triangularization, the last row, $[a_n,\ldots,a_0]$, is transformed to the row $[0,\ldots,0,r_k,\ldots,r_0]$ containing the coefficients of the remainder.

In the Sturm sequences we are interested in computing the polynomial remainders negated. To obtain the negated remainders we can either negate the remainder computed above or we can triangularize the matrix M after swapping its last two rows; that is the negated remainder is obtained by triangularizing the following matrix:

$$M=[[A_m,\ldots,A_0,0,\ldots,0],[0,A_m,\ldots,0,\ldots,0],\ldots,[a_n,\ldots,a_0],[0,\ldots,0,A_m,\ldots,A_0]].$$

After triangularization of the above matrix its last row contains the coefficients of the remainder negated. This last approach of computing negated remainders is used by VanVleck in his triangularization method.

Example 1. Let $p(x)=x^3+3x^2-7x+7$ and $q(x)=p'(x)=3x^2+6x-7$. To compute, in $\mathbb{Z}[x]$, the remainder on dividing $p(x)$ by $q(x)$ we use pseudo-division, the process by which – in order to force the quotient and remainder to be in $\mathbb{Z}[x]$ – we premultiply $p(x)$, times the leading coefficient of $q(x)$ raised to the power degree(p) – degree$(q)+1$. Sympy has the function prem that does this for us:

```
Python] print(prem( x**3 + 3*x**2 - 7*x + 7, 3*x**2 + 6*x - 7))
   -60*x + 84
Python]
```

To compute the remainder negated, which is,as we see, $-r(x)=60x-84$, it is also obtained with polynomial pseudodivision.

```
Python] print(-prem( x**3 + 3*x**2 - 7*x + 7, 3*x**2 + 6*x - 7))
   60*x - 84
Python]
```

3.2 Polynomial Resultants and Sylvester's Matrices

The resultant, res(p, q), of two polynomials $p(x)$ and $q(x)$, is defined as the product of all the differences between the roots of the polynomials. That is, if $p(x)=a_0(x-r_1)\cdot(x-r_2)\cdots(x-r_n)$ and $q(x)=b_0(x-s_1)\cdot(x-s_2)\cdots(x-s_m)$ then

$$\operatorname{res}(p,q)=a_0^m\cdot b_0^n\prod_{j=1}^{n}\prod_{k=1}^{m}(r_j-s_k)$$

By grouping together factors, we may also rewrite the resultant as

$$\operatorname{res}(p,q)=a_0^m\prod_{j=1}^{n}q(r_j)$$

or

$$\operatorname{res}(p,q)=(-1)^{m\cdot n}\,b_0^n\prod_{k=1}^{m}p(s_k).$$

A well known result states that the vanishing of the resultant of two polynomials is the necessary and suffcient condition for the two polynomials to have a common root.

3.3 Computation of Complete Sturm Sequences in $\mathbb{Z}[x]$ using Sylvester's Matrix

Sylvester presented a way to exactly compute the coefficients of the polynomial remainders in complete Sturm sequences as modified subresultants of `sylvester2`. This was reiterated by Van Vleck in the following Theorem:

Theorem 1. ***(Van Vleck, 1900)*** *Consider the polynomials* $p(x) = c_n x^n + ... + c_0$ *and* $q(x) = d_m x^m + ... + d_0$, *in* $\mathbb{Z}[x]$, *with* $c_n \neq 0, d_m \neq 0, n \geqslant m$. *Then the successive polynomials that are formed from the first* $2j$ *rows,* $j = 2, ..., n$, *of Sylvester's matrix (*`sylvester2`*) for* $p(x), q(x)$, *constitute a Sturm sequence.*

The proof of this theorem can be found in Van Vleck's paper [14] and elsewhere ([3], p. 263).

Notice that the theorem makes no reference to the Sturm sequence being complete, but clearly this is what Van Vleck had in mind. Sylvester himself was aware of incomplete sequences, but did not attempt to compute the correct sign of their polynomials. As stated in the next Section, this problem was solved by Pell and Gordon in 1917 [11].

For a given j, $2 \leqslant j \leqslant n$, the $n - j + 1$ coeffcients of the polynomial remainder are computed as determinants of $n - j + 1$ submatrices of the matrix s formed by the first $2j$ rows of Sylvester's matrix $S =$ `sylvester2(p,q)`. All of these submatrices have the same first $2j - 1$ columns, whereas the $2j$ − th column is successively the $(2j - 1 + k)$ − th column of s, where $k = 1, ..., n - j + 1$. The determinants of these $2j \times 2j$ submatrices are the modified subresultants.

3.4 Van Vleck's Triangularization Method for Computing in $\mathbb{Z}[x]$ Complete Sturm Sequences

Computing the coefficients of the polynomial remainders in a Sturm sequence by evaluating modified subresultants is quite a tedious process if carried out by hand and quite time consuming if carried out by computer.

Van Vleck realized that one does not have to compute modified subresultants of Sylvester's matrix `sylvester2` in order to find the coefficients of the polynomial remainders in the Sturm sequence. It suffices to simply triangularize `sylvester2` using integer preserving transformations, in which case the modified subresultants (the coefficients) can be read off the triangularized matrix.

We have the following ([14], p. 8):

Theorem 2. ***(Van Vleck, 1899)*** *Let* $p(x)$ *and* $q(x) = p'(x)$ *be two polynomials of degree* n *and* $n - 1$ *respectively and let* S *be their Sylvester matrix* `sylvester2(p,q)`. *If, using integer preserving transformations, we bring* S *into its upper triangular form,* $T(S)$, *then the even rows of* $T(S)$ *furnish the coefficients of the successive polynomial remainders of the Sturm sequence. The coefficients taken from a given row are multiplied times* $(-1)^k$, *where* k *is the number of negative elements on the principle diagonal above the row under consideration.*

Van Vleck takes advantage of the special form of Sylvester's matrix and computes $T(S)$ by updating only two rows at a time; to update these two rows he triangularizes a matrix of only three rows, a fact that makes his procedure extremely efficient. To keep the coefficients small he removes at each step the greatest common divisor (content) of the elements in both updated rows, and uses those reduced coefficients in the next three-row matrix.

Van Vleck's computation is justified by the fact that in Sylvester's matrix the elements (entries) of any two consecutive rows are the same as those of the two preceding rows. Therefore, if in any row the values of the elements are changed by adding a multiple of the preceding row, exactly the same change can be made in the elements of each alternate row thereafter, without altering the value of any modied subresultant that appears as a coefficient in one of the polynomials of the Sturm sequence.

In conclusion, Van Vleck presented a very efficient procedure for computing Sturm sequences in $\mathbb{Z}[x]$, and below it is demonstrated with the same example used by him ([14], pp. 8-9).

Warning 1. Although it is not stated in his paper, Van Vleck also applies the sign rule (mentioned in his theorem) to the triangularized smaller matrices of three rows. Namely, the coefficients taken from a given row are multiplied times $(-1)^k$, where k is the number of negative elements on the principle diagonal above the row under consideration.

Example 2. To compute the Sturm sequence of $p(x) = x^6 + x^5 - x^4 - x^3 + x^2 - x + 1$ we form the matrix $S = \text{sylvester2}(p, p')$:

```
Python] def sylvester2(f, g, x):
            # Trivial case f = g = 0
            if (f == 0 or g == 0):
                return Matrix([0])
            if (f == g and g == 0):
                return Matrix([0])
            fp = Poly(f, x).all_coeffs()
            gp = Poly(g, x).all_coeffs()
            m, n = degree( Poly(f, x), x), degree( Poly(g, x), x)
            # Trivial case f, g are constants
            if (m == n and n == 0):
                return Matrix([[1]])
            # order polys according to degree
            if (m < n):
                fp, gp = gp, fp
                m, n = n, m
            dim = 2*m
            M = zeros( dim )
            k = 0
            for i in range( m ):
                j = k
                for coeff in fp:
                    if (i == 0):
                        M[i, j] = coeff
                    else:
                        M[2*i, j] = coeff
                    j = j + 1
                j = m - n + k
                for coeff in gp:
                    if (i == 0):
                        M[i+1, j] = coeff
                    else:
                        M[2*i + 1, j] = coeff
                    j = j + 1
                k = k + 1
            return M
Python] S=Matrix(sylvester2(x**6+x**5-x**4-x**3+x**2-x+1, 6*x**5+5*x**4-4*x**3-
       3*x**2+2*x-1, x))
Python] pprint (Matrix(S))
   [1  1  -1  -1  1   -1  1   0   0   0   0   0 ]
   [                                            ]
```

```
[0  6  5   -4  -3  2   -1  0   0   0   0   0 ]
[                                              ]
[0  1  1   -1  -1  1   -1  1   0   0   0   0 ]
[                                              ]
[0  0  6   5   -4  -3  2   -1  0   0   0   0 ]
[                                              ]
[0  0  1   1   -1  -1  1   -1  1   0   0   0 ]
[                                              ]
[0  0  0   6   5   -4  -3  2   -1  0   0   0 ]
[                                              ]
[0  0  0   1   1   -1  -1  1   -1  1   0   0 ]
[                                              ]
[0  0  0   0   6   5   -4  -3  2   -1  0   0 ]
[                                              ]
[0  0  0   0   1   1   -1  -1  1   -1  1   0 ]
[                                              ]
[0  0  0   0   0   6   5   -4  -3  2   -1  0 ]
[                                              ]
[0  0  0   0   0   1   1   -1  -1  1   -1  1 ]
[                                              ]
[0  0  0   0   0   0   6   5   -4  -3  2   -1]
```

The first two rows stay the same, since there is no element to be eliminated in the first column.

In the second column there is one element to be eliminated. So, we form the 3×12 matrix M consisting of the second, third and fourth rows of S.

```
Python] M=Matrix(S[1:4,0:12])
Python] pprint(M)
```

```
[0  6  5  -4  -3  2   -1  0   0  0  0  0]
[                                       ]
[0  1  1  -1  -1  1   -1  1   0  0  0  0]
[                                       ]
[0  0  6  5   -4  -3  2   -1  0  0  0  0]
```

Using the function pivot we obtain, in two steps, matrix $M2$, the triangularized version of M.

```
Python] def pivot(M, i, j):
            # M is a matrix, and M[i, j] specifies the pivot point.
            # All elements below M[i, j] in the j-th column will
            # be zeroed, if they are not already, according to
            # Bareiss' integer preserving transormation.
            Ma = M[:, :] # copy of matrix M
            rs = Ma.rows  # No. of rows
            cs = Ma.cols  # No. of cols
            for r in range(i+1, rs):
                if Ma[r, j] != 0:
                    for c in range(j+1, cs):
                        Ma[r, c] = Ma[i, j]*Ma[r, c] - Ma[i, c]*Ma[r, j]
                    Ma[r, j] = 0
            return Ma
Python] M1=pivot(M,0,1)
```

```
Python] pprint(M1)
   [0  6  5  -4  -3  2   -1  0   0  0  0  0]
   [                                        ]
   [0  0  1  -2  -3  4   -5  6   0  0  0  0]
   [                                        ]
   [0  0  6  5   -4  -3  2   -1  0  0  0  0]
Python] M2=pivot(M1,1,2)
Python] pprint(M2)
   [0  6  5  -4  -3   2   -1   0   0  0  0  0]
   [                                         ]
   [0  0  1  -2  -3   4   -5   6   0  0  0  0]
   [                                         ]
   [0  0  0  17  14  -27  32  -37  0  0  0  0]
```

The coefficients in the second and third row of $M2$ cannot be further reduced because the gcd of the elements in each row is 1. Moreover, since the signs of the diagonal elements are all positive nothing will change. Hence, the last two rows of $M2$ will replace the third and fourth rows of S.

```
Python] S[2,:]=M2[1,:]
Python] S[3,:]=M2[2,:]
Python] pprint(S)
   [1  1  -1  -1  1   -1   1    0   0   0   0   0 ]
   [                                              ]
   [0  6  5   -4  -3   2   -1   0   0   0   0   0 ]
   [                                              ]
   [0  0  1   -2  -3   4   -5   6   0   0   0   0 ]
   [                                              ]
   [0  0  0   17  14  -27  32  -37  0   0   0   0 ]
   [                                              ]
   [0  0  1   1   -1  -1   1   -1   1   0   0   0 ]
   [                                              ]
   [0  0  0   6   5   -4   -3   2   -1  0   0   0 ]
   [                                              ]
   [0  0  0   1   1   -1   -1   1   -1  1   0   0 ]
   [                                              ]
   [0  0  0   0   6    5   -4   -3  2   -1  0   0 ]
   [                                              ]
   [0  0  0   0   1    1   -1   -1  1   -1  1   0 ]
   [                                              ]
   [0  0  0   0   0    6   5   -4   -3  2   -1  0 ]
   [                                              ]
   [0  0  0   0   0    1   1   -1   -1  1   -1  1 ]
   [                                              ]
   [0  0  0   0   0    0   6    5   -4  -3  2   -1]
```

The next matrix with three rows is formed by the two newly inserted rows in S, rotated appropriately when needed, and we remove the content from each one of the second and third rows of $M2$ and we repeat the pivoting procedure described above:

```
Python] def rotate(l,n):
            return l[-n:] + l[:-n]
```

```
Python] M=Matrix([S[3,:],rotate(S[2,:],1),rotate(S[3,:],1)])
Python] pprint(M)
   [0  0  0  17  14  -27  32   -37   0   0  0  0]
   [                                              ]
   [0  0  0  1   -2  -3    4   -5    6   0  0  0]
   [                                              ]
   [0  0  0  0   17  14   -27  32   -37  0  0  0]
Python] M1=pivot(M,0,3)
Python] pprint(M1)
   [0  0  0  17  14   -27  32   -37   0   0  0  0]
   [                                              ]
   [0  0  0  0   -48  -24  36   -48  102  0  0  0]
   [                                              ]
   [0  0  0  0   17   14   -27  32   -37  0  0  0]
Python] M1[1,:]=M1[1,:]/abs(S[1,1])
Python] pprint(M1)
   [0  0  0  17  14  -27  32   -37   0   0  0  0]
   [                                              ]
   [0  0  0  0   -8  -4    6   -8   17   0  0  0]
   [                                              ]
   [0  0  0  0   17  14   -27  32   -37  0  0  0]
Python] M2=pivot(M1,1,4)
Python] pprint(M2)
   [0  0  0  17  14  -27  32   -37   0   0  0  0]
   [                                              ]
   [0  0  0  0   -8  -4    6    -8   17  0  0  0]
   [                                              ]
   [0  0  0  0   0   -44  114  -120  7   0  0  0]
Python] M2[2,:]=M2[2,:]/abs(S[2,2])
Python] pprint(M2)
   [0  0  0  17  14  -27  32   -37   0   0  0  0]
   [                                              ]
   [0  0  0  0   -8  -4    6    -8   17  0  0  0]
   [                                              ]
   [0  0  0  0   0   -44  114  -120  7   0  0  0]
```

Next we take care of the signs: The second row of $M2$ will replace the fifth row of S as is, since the diagonal element above -8 is positive, however, the third row of $M2$ will change sign since there is one negative element in the second row, on the diagonal.

```
Python] S[4,:]=M2[1,:]
Python] S[5,:]=-M2[2,:]
Python] pprint(S)
   [1  1  -1  -1  1   -1    1     0   0   0   0   0 ]
   [                                               ]
   [0  6  5   -4  -3   2    -1    0   0   0   0   0 ]
   [                                               ]
   [0  0  1   -2  -3   4    -5    6   0   0   0   0 ]
   [                                               ]
```

```
[0  0  0   17  14  -27   32   -37  0   0   0   0 ]
[                                                 ]
[0  0  0   0   -8  -4    6    -8   17  0   0   0 ]
[                                                 ]
[0  0  0   0   0   44   -114  120  -7  0   0   0 ]
[                                                 ]
[0  0  0   1   1   -1    -1    1   -1  1   0   0 ]
[                                                 ]
[0  0  0   0   6    5    -4   -3   2   -1  0   0 ]
[                                                 ]
[0  0  0   0   1    1    -1   -1   1   -1  1   0 ]
[                                                 ]
[0  0  0   0   0    6    5    -4   -3  2   -1  0 ]
[                                                 ]
[0  0  0   0   0    1    1    -1   -1  1   -1  1 ]
[                                                 ]
[0  0  0   0   0    0    6     5   -4  -3  2   -1]
```

Again, the next matrix with three rows is formed by the two newly inserted rows in S, rotated appropriately when needed:

```
Python] M=Matrix([S[5,:],rotate(S[4,:],1),rotate(S[5,:],1)])
Python] pprint(M)
   [0  0  0  0  0  44  -114  120   -7   0   0  0]
   [                                             ]
   [0  0  0  0  0  -8   -4    6    -8   17  0  0]
   [                                             ]
   [0  0  0  0  0  0    44   -114  120  -7  0  0]

Python] M1=pivot(M,0,5)
Python] pprint(M1)
   [0  0  0  0  0  44  -114   120    -7    0   0  0]
   [                                              ]
   [0  0  0  0  0  0   -1088  1224  -408  748  0  0]
   [                                              ]
   [0  0  0  0  0  0    44    -114  120   -7   0  0]

Python] M1[1,:]=M1[1,:]/abs(S[3,3])
Python] pprint(M1)
   [0  0  0  0  0  44  -114  120   -7   0   0  0]
   [                                             ]
   [0  0  0  0  0  0   -64    72   -24  44  0  0]
   [                                             ]
   [0  0  0  0  0  0    44   -114  120  -7  0  0]

Python] M2=pivot(M1,1,6)
Python] pprint(M2)
   [0  0  0  0  0  44  -114  120    -7      0    0  0]
   [                                                 ]
   [0  0  0  0  0  0   -64    72    -24    44    0  0]
   [                                                 ]
   [0  0  0  0  0  0    0    4128  -6624  -1488  0  0]

Python] M2[2,:]=M2[2,:]/abs(S[4,4])
Python] pprint(M2)
```

```
   [0  0  0  0  0  44  -114  120   -7    0    0  0]
   [                                                ]
   [0  0  0  0  0  0   -64   72   -24    44   0  0]
   [                                                ]
   [0  0  0  0  0  0    0    516  -828  -186  0  0]
```

```
Python] S[6,:]=M2[1,:]
Python] S[7,:]=-M2[2,:]
Python] pprint(S)
```

```
   [1  1  -1  -1  1   -1    1     0     0    0    0   0 ]
   [                                                    ]
   [0  6  5   -4  -3   2    -1    0     0    0    0   0 ]
   [                                                    ]
   [0  0  1   -2  -3   4    -5    6     0    0    0   0 ]
   [                                                    ]
   [0  0  0   17  14  -27   32   -37    0    0    0   0 ]
   [                                                    ]
   [0  0  0   0   -8  -4    6    -8    17   0    0   0 ]
   [                                                    ]
   [0  0  0   0   0   44   -114  120   -7    0    0   0 ]
   [                                                    ]
   [0  0  0   0   0    0   -64    72   -24  44    0   0 ]
   [                                                    ]
   [0  0  0   0   0    0    0    -516  828  186   0   0 ]
   [                                                    ]
   [0  0  0   0   1    1    -1    -1    1   -1    1   0 ]
   [                                                    ]
   [0  0  0   0   0    6    5     -4   -3    2   -1   0 ]
   [                                                    ]
   [0  0  0   0   0    1    1     -1   -1    1   -1   1 ]
   [                                                    ]
   [0  0  0   0   0    0    6     5    -4   -3   2   -1]
```

```
Python] M=Matrix([S[7,:],rotate(S[6,:],1),rotate(S[7,:],1)])
Python] pprint(M)
```

```
   [0  0  0  0  0  0  0  -516  828   186   0   0]
   [                                              ]
   [0  0  0  0  0  0  0  -64    72   -24  44   0]
   [                                              ]
   [0  0  0  0  0  0  0   0    -516  828  186  0]
```

```
Python] M1=pivot(M,0,7)
Python] pprint(M1)
```

```
   [0  0  0  0  0  0  0  -516   828    186     0     0]
   [                                                  ]
   [0  0  0  0  0  0  0   0    15840  24288  -22704  0]
   [                                                  ]
   [0  0  0  0  0  0  0   0    -516    828    186    0]
```

```
Python] M1[1,:]=M1[1,:]/abs(S[5,5])
Python] pprint(M1)
```

```
   [0  0  0  0  0  0  0  -516  828   186   0    0]
```

```
   [                                                    ]
   [0  0  0  0  0  0  0   0    360   552  -516  0]
   [                                                    ]
   [0  0  0  0  0  0  0   0   -516  828  186   0]
```

```
Python] M2=pivot(M1,1,8)
Python] pprint(M2)
   [0  0  0  0  0  0  0  -516  828   186       0     0]
   [                                                   ]
   [0  0  0  0  0  0  0   0    360   552     -516    0]
   [                                                   ]
   [0  0  0  0  0  0  0   0     0   582912  -199296  0]
```

```
Python] M2[2,:]=M2[2,:]/abs(S[6,6])
Python] pprint(M2)
   [0  0  0  0  0  0  0  -516  828  186     0    0]
   [                                              ]
   [0  0  0  0  0  0  0   0    360  552   -516   0]
   [                                              ]
   [0  0  0  0  0  0  0   0     0   9108  -3114  0]
```

```
Python] S[8,:]=M2[1,:]
Python] S[9,:]=M2[2,:]
Python] pprint(S)
   [1  1  -1  -1  1   -1   1    0    0    0      0    0 ]
   [                                                    ]
   [0  6  5   -4  -3  2    -1   0    0    0      0    0 ]
   [                                                    ]
   [0  0  1   -2  -3  4    -5   6    0    0      0    0 ]
   [                                                    ]
   [0  0  0   17  14  -27  32   -37  0    0      0    0 ]
   [                                                    ]
   [0  0  0   0   -8  -4   6    -8   17   0      0    0 ]
   [                                                    ]
   [0  0  0   0   0   44   -114 120  -7   0      0    0 ]
   [                                                    ]
   [0  0  0   0   0   0    -64  72   -24  44     0    0 ]
   [                                                    ]
   [0  0  0   0   0   0    0    -516 828  186    0    0 ]
   [                                                    ]
   [0  0  0   0   0   0    0    0    360  552   -516  0 ]
   [                                                    ]
   [0  0  0   0   0   0    0    0    0    9108  -3114 0 ]
   [                                                    ]
   [0  0  0   0   0   1    1    -1   -1   1     -1    1 ]
   [                                                    ]
   [0  0  0   0   0   0    6    5    -4   -3    2     -1]
```

```
Python] M=Matrix([S[9,:],rotate(S[8,:],1),rotate(S[9,:],1)])
Python] pprint(M)
   [0  0  0  0  0  0  0  0  0  9108  -3114    0  ]
```

```
   [                                                          ]
   [0  0  0  0  0  0  0  0  0  360    552   -516 ]
   [                                                          ]
   [0  0  0  0  0  0  0  0  0   0    9108   -3114]

Python] M1=pivot(M,0,9)
Python] pprint(M1)
   [0  0  0  0  0  0  0  0  0  9108   -3114      0    ]
   [                                                   ]
   [0  0  0  0  0  0  0  0  0   0    6148656  -4699728]
   [                                                   ]
   [0  0  0  0  0  0  0  0  0   0     9108     -3114  ]

Python] M1[1,:]=M1[1,:]/abs(S[7,7])
Python] pprint(M1)
   [0  0  0  0  0  0  0  0  0  9108  -3114    0  ]
   [                                             ]
   [0  0  0  0  0  0  0  0  0   0    11916  -9108]
   [                                             ]
   [0  0  0  0  0  0  0  0  0   0    9108   -3114]

Python] M2=pivot(M1,1,10)
Python] pprint(M2)
   [0  0  0  0  0  0  0  0  0  9108  -3114     0    ]
   [                                                ]
   [0  0  0  0  0  0  0  0  0   0    11916   -9108  ]
   [                                                ]
   [0  0  0  0  0  0  0  0  0   0      0    45849240]

Python] M2[2,:]=M2[2,:]/abs(S[8,8])
Python] pprint(M2)
   [0  0  0  0  0  0  0  0  0  9108  -3114    0   ]
   [                                              ]
   [0  0  0  0  0  0  0  0  0   0    11916  -9108 ]
   [                                              ]
   [0  0  0  0  0  0  0  0  0   0      0    127359]

Python] S[10,:]=-M2[1,:]
Python] S[11,:]=M2[2,:]
Python] pprint(S)
   [1  1  -1  -1  1   -1    1     0     0    0      0       0   ]
   [                                                            ]
   [0  6  5   -4  -3   2    -1    0     0    0      0       0   ]
   [                                                            ]
   [0  0  1   -2  -3   4    -5    6     0    0      0       0   ]
   [                                                            ]
   [0  0  0   17  14  -27   32   -37    0    0      0       0   ]
   [                                                            ]
   [0  0  0   0   -8  -4    6     -8   17    0      0       0   ]
   [                                                            ]
   [0  0  0   0   0   44   -114  120   -7    0      0       0   ]
```

```
[                                                                    ]
[0  0  0  0  0  0  -64   72   -24   44    0       0    ]
[                                                                    ]
[0  0  0  0  0  0   0   -516  828  186    0       0    ]
[                                                                    ]
[0  0  0  0  0  0   0    0    360  552   -516     0    ]
[                                                                    ]
[0  0  0  0  0  0   0    0     0   9108  -3114    0    ]
[                                                                    ]
[0  0  0  0  0  0   0    0     0    0    -11916  9108  ]
[                                                                    ]
[0  0  0  0  0  0   0    0     0    0      0     127359]
```

We finally obtain the triangularized Sylvester matrix $T(S)$, shown above.

From the even rows of the triangularized matrix we extract the coefficients of the polynomials in the Sturm sequence, correcting the sign as indicated in Theorem 2.

Namely, the coefficients of the Sturm sequence are:

- from row 1: [1, 1, -1, -1, 1, -1, 1]
- from row 2: [6, 5, -4, -3, 2, -1]
- from row 4: [17, 14, -27, 32, -37]
- from row 6: [44, -144, 120, -7] => [-44, 144, -120, 7] sign changed
- from row 8: [-516, 828, 186]
- from row 10: [9108, -3114]
- from row 12: [127359] => [-127359] sign changed

Indeed, using the function sturm of Xcas we see that the results are the same. Whereas, sometimes the coefficients may differ in value, as expected, since Van Vleck removes the content from each polynomial, their signs are identical.

```
> sturm( x^6 + x^5 - x^4 - x^3 + x^2 - x + 1 )[1]
```

```
[[1, 1, -1, -1, 1, -1, 1], [6, 5, -4, -3, 2, -1], [17, 14, -27, 32, -37], [ -44,
114, -120, 7], [-516, 828,186],[9108, -3114], -127359]
```

3.5 Implementation of Van Vleck's method for complete Sturm Sequences in Python

In this way I have implemented Van Vleck's procedure in `Python`, with the use of `Sympy`, in the function `sturmSeqVanVleck`, which is quite straightforward and uses the following additional functions:

- `Sq_free`, which converts a polynomial into a product of square free factors (a necessary condition for computing its Sturm sequence),
- `sylvester2`, which constructs the appropriate Sylvester matrix of two polynomials,
- `row2poly`, which converts a matrix row to a polynomial of a specified degree,
- `pivot`, which converts a matrix to its upper triangular form,
- `makelist`, which takes as argument the bounds a,b of an index variable and a step s and makes the list of a function
- `rotate`, which rotates a list by a given number of items to the left or right,
- `smallMatrixVanVleckRule`, which applies Van Vleck's sign rule to correctly compute the signs of the coefficients in the triangularized 3-rows matrices, and
- `sturmSeqVanVleckRule`, which applies Van Vleck's sign rule to correctly compute the signs of the coefficients in the final triangularized matrix.

However, instead of removing the content of each polynomial, in `sturmSeqVanVleck` we follow Sylvester's practice for complete sequences and reduce the coefficients by dividing out the diagonal element "three" rows up [15]. This way the coefficients computed with Van Vleck's method are modified subresultants and they are the same as those obtained with the sturm function of Xcas.

The whole program for the Van Vleck's procedure of computing complete Sturm sequences with matrix triangularization written in Python (version 2.7.5) is as follows:

```
import sympy as sp
import giacpy as gp    # an interface to be able to use from Python the Giac
features
                              # Giac is a Computer algebra system
from sympy.abc import x


def smallMatrixVanVleckRule(M):
    # M is a diagonal matrix with 3 rows
    # This is Van Vleck's rule for 3-row matrices
    # Check to make sure it is diagonal matrix
    if ((M[1,0]!=0) or (M[2,0]!=0) or  (M[2,1]!=0) or (M[0,0]==0) or
(M[1,1]==0) or (M[2,2]==0)):
        print ("Not a diagonal matrix")
        return (0)
```

```
    # Check negative diagonals
    # Van Vleck's rule for the 3-rows matrix
    if (sp.sign(M[0,0])==-1):
        M[1,:]=-M[1,:]
    if ((sp.sign(M[0,0])==-1) and (sp.sign(M[1,1])==1)):
        M[2,:]=-M[2,:]
    if ((sp.sign(M[0,0])==1) and (sp.sign(M[1,1])==-1)):
        M[2,:]=-M[2,:]
    return((M))


def sylvester2(f, g, x):
    # Trivial case f = g = 0
    if (f == 0 or g == 0):
        return sp.Matrix([0])
    if (f == g and g == 0):
        return sp.Matrix([0])
    fp = sp.Poly(f, x).all_coeffs()
    gp = sp.Poly(g, x).all_coeffs()
    m, n = sp.degree( sp.Poly(f, x), x), sp.degree( sp.Poly(g, x), x)
    # Trivial case f, g are constants
    if (m == n and n == 0):
        return sp.Matrix([[1]])
    # order polys according to degree
    if (m < n):
        fp, gp = gp, fp
        m, n = n, m
    dim = 2*m
    M = sp.zeros( dim )
    k = 0
    for i in range( m ):
        j = k
        for coeff in fp:
            if (i == 0):
                M[i, j] = coeff
            else:
                M[2*i, j] = coeff
            j = j + 1
        j = m - n + k
        for coeff in gp:
            if (i == 0):
                M[i+1, j] = coeff
            else:
                M[2*i + 1, j] = coeff
            j = j + 1
        k = k + 1
    return M


def sturmSeqVanVleckRule(M):
    SL=[M[0,:],M[1,:]]
    # Van Vleck's rule for the FINAL triangularized matrix
    for j in range (3, M.shape[0], 2):
        pr=1
```

```
        for k in range (0,j):
            pr=pr*(sp.sign(M[k,k]))
        pr=pr*M[j,:]
        SL.append(pr)
    return (sp.Matrix(SL))


def row2poly(row_E,deg):
    #a row of a matrix E where NOT all elements are coeffs of the poly
    # some entries at the begining and/or at the end are zero
    poly=[]
    length=len(row_E)
    # find the beginning of the poly ; i.e. the first non-zero element of the
row
    k=0
    while row_E[k]==0:
        k=k+1
    # Once we have the beginning, append the next deg+1 elements to poly
    for j in range(0, deg + 1):
      if (k+j <= length):
          poly.append(row_E[k+j])
    return (poly)


def Sq_free(f):
    f=sp.expand(f)
    lst,q = sp.sqf_list(f)[1],1
    for i in range(len(lst)):
        q=q * lst[i][0]
    return(q)


def pivot(M, i, j):
    # M is a matrix, and M[i, j] specifies the pivot point.
    # All elements below M[i, j] in the j-th column will
    # be zeroed, if they are not already, according to
    # Bareiss' integer preserving transormation.
    Ma = M[:, :] # copy of matrix M
    rs = Ma.rows  # No. of rows
    cs = Ma.cols  # No. of cols
    for r in range(i+1, rs):
        if Ma[r, j] != 0:
            for c in range(j+1, cs):
                Ma[r, c] = Ma[i, j]*Ma[r, c] - Ma[i, c]*Ma[r, j]
            Ma[r, j] = 0
    return Ma


def rotate(l,n):
    return l[-n:] + l[:-n]
```

```
def sturmSeqVanVleck_poly(pp):
    # It is assumed pp is squarefree and that 0 is NOT a root
    # It works ONLY for polys with POSITIVE leading coefficient
    # if the lead. coef. of the poly is negative, i.e. coeffs(p)[0] <0  make p
:= (-1)*p,
    # work with it and at the end multiply the result times (-1)

    # initialize
    p=pp
    FLAG=0
    pv=0
    E=0
    polyList=[]

    # Zero degree polynomial
    if (type(p)==int):
       print("Maybe zero has been divided out. Poly is of degree 0.")
       return (p)

    # get variable name && check sign of poly
    v=sp.Poly(pp).free_symbols

    if(sp.Poly(pp).LC()<0):
        p=(-1)*p
        FLAG=1
    # make sure p is square free
    spp=sp.sqf(pp)
    if (spp != sp.simplify(pp)):
        p=Sq_free(pp)

    # Decrease the degree of the poly if 0 is a root
    while (p).subs(x,0)==0 :
        p=sp.simplify(p/x)

    # Divide by the content
    p=sp.simplify(p/sp.content(p))

    n=sp.degree(p,x)
    p2=sp.diff(p,x)
    polyList.append(sp.Poly(p,x).all_coeffs()) #convert(symb2poly(p),list))
    polyList.append(sp.Poly(p2,x).all_coeffs()) #convert(symb2poly(p2),list))
    nextDeg=n-2

    # first degree polynomial
    if (n==1):
        if (FLAG==1):
            for j in range (0 , len(polyList)):
                for i in range(0,len(polyList[j])):
                    polyList[j][i]=(-1)*polyList[j][i]
            return polyList
        else: return (polyList)

    # form the Sylvester matrix E
    E=sylvester2(p,p2,v)
```

```
    # main loop
    for j in range (0,(E.shape[0]) - 3 + 1, 2):
    #  form and triangularize the 3xc matrix
     #  The way Van Vleck sets up the 3-rows matrix he gets the remainder
negated in the last row.
    #  However, if the remainder appears in the second row it is NOT negated,
    #  that is why negate it when swap rows

        M=0

M=sp.Matrix([rotate(E[j+1,:],-j-1),rotate(E[j,:],-j),rotate(E[j+1,:],-j)])

        M1=0
        # Van Vleck removes the content at each step
        # However, we reduce the coefficients by dividing out the diagonal
        # element 3 rows up --- when j > 0
        M1=pivot(M,0,0)

        if  ((j > 0) and (pv!=1)):
            M1[1,:]=M1[1,:]/abs(E[j-1,j-1])

        # check if pivot needs to be done as in p:=x^3-7x+7
        if (M1[1,1]==0):
            M1.row_swap(1,2)

            M1[2,:]=-M1[2,:]
            pv=1
            print("Pivot; terminate")

        M2=0
        # see comment for M1 above
        M2=pivot(M1,1,1)
        if ((j > 0) and (pv!=1)):
            M2[2,:]=M2[2,:]/abs(E[j,j])
        #  change the signs when necessary

        M2=smallMatrixVanVleckRule(M2)
        #  update the rows of E
        E[j+2,:]= sp.Matrix([(rotate((M2[1,:]),j+1))])
        E[j+3,:]= sp.Matrix([(rotate((M2[2,:]),j+1))])
        pDet=row2poly(E[j+3,:], nextDeg)
        polyList.append(pDet)
        nextDeg=nextDeg-1

    #  return the Sturm sequence according to Van Vleck
    res=sturmSeqVanVleckRule(E)

    if FLAG == 1:
        res=(-1)*res
        #polyList=(-1)*polyList
        for j in range (0 , len(polyList)):
            for i in range(0,len(polyList[j])):
                polyList[j][i]=(-1)*polyList[j][i]

    for j in range (2 , len(polyList)):
            if (sp.sign(polyList[j][0]) != sp.sign(res[j,2*j-1])):
```

```
                for i in range(0,len(polyList[j])):
                    polyList[j][i]=(-1)*polyList[j][i]

    return polyList


p = x**6+x**5-x**4-x**3+x**2-x+1

print sturmSeqVanVleck_poly(p)

print gp.sturm(p)[1]      #check the results with the function sturm of giacpy
```

4. The Generalized Triangularization Method for Computing in $\mathbb{Z}[x]$ Sturm Sequences of Any Kind

As we saw before, for incomplete Sturm sequences we cannot compute the exact signs of the polynomial coefficients using modified subresultants. Therefore, we cannot easily extend Van Vleck's triangularization method for complete Sturm sequences to general Sturm sequences, i.e. sequences that can be either complete or incomplete. The reason is that, in trying to compute general Sturm sequences by triangularizing `sylvester2` matrices, we are faced with the following major problems:

- In general, the coefficients computed by the matrix triangularization process are not modified subresultants and their signs may not be correct.
- We cannot use Theorem 2, Van Vleck's "sign rule", since it computes the correct signs of the polynomial coefficients of only complete Sturm sequences.

To wit, to correctly compute the signs of the polynomial coefficients of a general Sturm sequence, we clearly have to use another "sign rule", one that is valid for both complete and incomplete Sturm sequences. This new rule is provided by the following theorem by Pell and Gordon [11], which also makes use of the same matrix `sylvester2`, used by Van Vleck:

Theorem 3. ***(Pell-Gordon, 1917)*** *Let*

$$A = a_0 x^n + a_1 x^{n-1} + \cdots + a_n$$

and

$$B = b_0 x^n + b_1 x^{n-1} + \cdots + b_n$$

be two polynomials of the n-th degree. Modify the process of finding the highest common factor of A and B by taking at each stage the negative of the remainder. Let the i-th modified remainder be

$$R^{(i)} = r_0^{(i)} x^{m_i} + r_1^{(i)} x^{m_i - 1} + \cdots + r_{m_i}^{(i)}$$

where $(m_i + 1)$ *is the degree of the preceeding remainder, and where the first* $(p_i - 1)$ *coefficients of* $R^{(i)}$ *are zero, and the* p_i*-th coefficient* $\varrho_i = r_{p_i - 1}^{(i)}$ *is different from zero. Then for* $k = 0, 1, \ldots, m_i$ *the coefficients* $r_k^{(i)}$ *are given by*

$$r_k^{(i)} = \frac{(-1)^{u_i - 1}(-1)^{u_i - 2} \cdots (-1)^{u_1}(-1)^{v_i - 1}}{\varrho_{i-1}^{p_{i-1}+1} \varrho_{i-2}^{p_{i-2}+p_{i-1}} \cdots \varrho_1^{p_1+p_2} \varrho_0^{p_1}} \cdot \mathrm{Det}(i,k), \qquad (1)$$

where $u_{i-1} = 1 + 2 + \ldots + p_{i-1}$, $v_{i-1} = p_1 + p_2 + \ldots + p_{i-1}$ *and*

$$\mathrm{Det}(i,k) = \begin{vmatrix}
a_0 & a_1 & a_2 & \ldots & . & . & \ldots & a_{2v_i-1} & a_{2v_i-1+1+k} \\
b_0 & b_1 & b_2 & \ldots & . & . & \ldots & b_{2v_i-1} & b_{2v_i-1+1+k} \\
0 & a_0 & a_1 & \ldots & . & . & \ldots & a_{2v_i-1-1} & a_{2v_i-1+k} \\
0 & b_0 & b_1 & \ldots & . & . & \ldots & b_{2v_i-1-1} & b_{2v_i-1+k} \\
. & . & . & \ldots & . & . & \ldots & . & . \\
0 & 0 & 0 & \ldots & a_0 & a_1 & \ldots & a_{v_i-1} & a_{v_i-1+1+k} \\
0 & 0 & 0 & \ldots & b_0 & b_1 & \ldots & b_{v_i-1} & b_{v_i-1+1+k}
\end{vmatrix}$$

Proof. The proof by induction of this theorem depends on two Lemmas and can be found in the original paper of Pell and Gordon.

As indicated elsewhere [7], we use a modification of formula (1) to compute the coefficients of the Sturm sequence. In our case $p_0 = \deg(A) - \deg(B) = 1$, since B is the derivative of A and, hence, the modified formula is shown below with the changes appearing in bold:

$$r_k^{(i)} = \frac{(-1)^{u_i - 1}(-1)^{u_i - 2} \cdots (-1)^{u_1}(-1)^{u_0}(-1)^{v_i - 1}}{\varrho_{i-1}^{p_{i-1}+p_i\text{- degDiffer}} \varrho_{i-2}^{p_{i-2}+p_{i-1}} \cdots \varrho_1^{p_1+p_2} \varrho_0^{p_0+p_1}} \cdot \frac{\mathrm{Det}(i,k)}{\varrho_{\text{-}1}}, \qquad (2)$$

where $\varrho_{\text{-}1} = a_0$, the leading coefficient of A and degDiffer is the difference between the expected degree m_i and the actual degree of the remainder.

It should be noted that in the general case $p_0 = \deg(A) - \deg(B)$ and that the division $\dfrac{\mathrm{Det}(i,k)}{\varrho_{\text{-}1}}$ is possible if the leading coeffcient of A is the only element in the first column of `sylvester2`. Moreover, if the leading coeffcient of A is negative we work with the polynomial negated and atthe end we reverse the signs of all polys in the sequence. □

To see how equation (2) of Theorem 3 is used in the general triangularization process, suppose that we have computed with the latter the $i-$th polynomial remainder

$$s^{(i)} = s_0^{(i)} x^{m_i} + s_1^{(i)} x^{m_i - 1} + \cdots + s_{m_i}^{(i)}, \qquad (3)$$

where $s_k^{(i)} \epsilon \mathbb{Z}$, $0 \leqslant k \leqslant m_i$, in general, the coefficients $s_k^{(i)}$ are not modified subresultants and we are not sure about the correctness of their signs.

To compute the correct sign of $s^{(i)}$ we evaluate equation (2) only for the leading coefficient $r_j^{(i)}$, that is, for $k = j$ where j is the smallest integer for which $\frac{\mathrm{Det}(i,j)}{\varrho_{-1}} \neq 0$. Then if $\mathrm{sgn}(s_j^{(i)}) \neq \mathrm{sgn}(r_j^{(i)})$ we set $s^{(i)} = -s^{(i)}$.

Having computed the correct sign of $s^{(i)}$ we can force its coefficients to equal the corresponding modified subresultants by multiplying $s^{(i)}$ times $\frac{|\mathrm{Det}(i,j)|}{|s_j^{(i)} \varrho_{-1}|}$. In other words, by computing just one determinant, $\mathrm{Det}(i,j)$, and forming the product

$$\frac{|\mathrm{Det}(i,j)|}{|s_j^{(i)} \varrho_{-1}|} \cdot s^{(i)} \qquad (4)$$

we obtain the $i-$ th Sturmian polynomial remainder, whose $k-$ th coefficient is the modified subresultant shown below:

$$\frac{(-1)^{u_i - 1}(-1)^{u_i - 2} \cdots (-1)^{u_1}(-1)^{u_0}(-1)^{v_i - 1}}{\mathrm{sgn}(\varrho_{i-1})^{p_{i-1} + p_i - \mathrm{degDiffer}} \mathrm{sgn}(\varrho_{i-2})^{p_{i-2} + p_{i-1}} \cdots \mathrm{sgn}(\varrho_1)^{p_1 + p_2} \mathrm{sgn}(\varrho_0)^{p_0 + p_1}} \cdot \frac{\mathrm{Det}(i,k)}{\varrho_{-1}} \qquad (5)$$

This is so, because of the existing ratio equality

$$\frac{|\mathrm{Det}(i,j)|}{|s_j^{(i)} \varrho_{-1}|} = \frac{|\mathrm{Det}(i,j)|}{|s_k^{(i)} \varrho_{-1}|} \qquad (6)$$

that holds for $j < k \leqslant m_i$, recall that $s_j^{(i)}$ is the leading coefficient of the polynomial remainder $s^{(i)}$, in equation (3), computed by the generalized matrix triangularization method.

From the above it is obvious that for our purposes we only need expression (5). The overhead in this generalized method is that we have to keep track of all the variables in Theorem 3 and compute one determinant for each polynomial remainder.

Example 3. Using the generalized matrix triangularization method we will compute the incomplete Sturm sequence of the polynomial $p(x) = 2x^5 - 3x^4 - 3$.

We begin by constructing the `sylvester2` matrix S, which will remain unchanged so that we can compute the various modified subresultants, we also make a copy of it SS, which will be triangularized.

As in Van Vleck' procedure we form the 3-rows matrices M, $M1$, $M2$ and from the last one we obtain a candidate for the first remainder, namely $72x^3+300$, moreover, the second and third rows of $M2$ will replace, respectively, the $3-$rd and $4-$th rows of SS.

```
Python] S=sylvester2(2*x**5-3*x**4-3,10*x**4-12*x**3,x)
Python] pprint(S)
   [2  -3   0    0    0   -3    0   0   0   0 ]
   [                                           ]
   [0  10  -12   0    0    0    0   0   0   0 ]
   [                                           ]
   [0  2   -3    0    0    0   -3   0   0   0 ]
   [                                           ]
   [0  0   10   -12   0    0    0   0   0   0 ]
   [                                           ]
   [0  0    2   -3    0    0    0  -3   0   0 ]
   [                                           ]
   [0  0    0   10   -12   0    0   0   0   0 ]
   [                                           ]
   [0  0    0    2   -3    0    0   0  -3   0 ]
   [                                           ]
   [0  0    0    0   10   -12   0   0   0   0 ]
   [                                           ]
   [0  0    0    0    2   -3    0   0   0  -3 ]
   [                                           ]
   [0  0    0    0    0   10   -12  0   0   0 ]
Python] M=Matrix(S[1:4,0:12])
Python] pprint(M)
   [0  10  -12   0   0  0  0   0  0  0]
   [                                  ]
   [0  2   -3    0   0  0  -3  0  0  0]
   [                                  ]
   [0  0   10   -12  0  0  0   0  0  0]
Python] M1=pivot(M,0,1)
Python] pprint(M1)
   [0  10  -12   0   0  0   0   0  0  0]
   [                                   ]
   [0  0   -6    0   0  0  -30  0  0  0]
   [                                   ]
   [0  0   10   -12  0  0   0   0  0  0]
Python] M2=pivot(M1,1,2)
Python] pprint(M2)
   [0  10  -12  0   0  0   0   0  0  0]
   [                                  ]
   [0  0   -6   0   0  0  -30  0  0  0]
   [                                  ]
   [0  0    0   72  0  0  300  0  0  0]
Python] SS=Matrix(S)
Python] SS[2,:]=M2[1,:]
Python] SS[3,:]=M2[2,:]
Python] pprint(SS)
   [2  -3   0   0    0   -3    0   0   0   0 ]
```

```
[                                                  ]
[0  10  -12  0    0    0    0    0    0    0 ]
[                                                  ]
[0  0   -6   0    0    0   -30   0    0    0 ]
[                                                  ]
[0  0    0   72   0    0   300   0    0    0 ]
[                                                  ]
[0  0    2   -3   0    0    0   -3    0    0 ]
[                                                  ]
[0  0    0   10  -12   0    0    0    0    0 ]
[                                                  ]
[0  0    0   2   -3    0    0    0   -3    0 ]
[                                                  ]
[0  0    0   0   10   -12   0    0    0    0 ]
[                                                  ]
[0  0    0   0    2   -3    0    0    0   -3]
[                                                  ]
[0  0    0   0    0   10   -12   0    0    0 ]
```

Now we have to compute the modified subresultants which will be used below for evaluation of expression (5).

Following VanVleck's theorem, for $j=2$ we take the submatrix s of S, consisting of the first $2j=4$ rows of S and, since $n=5$, we compute the coefficients of the polynomial remainder of degree $n-j=3$.

```
Python] s=Matrix(S[0:4,0:10])
Python] pprint(s)
   [2  -3   0    0   0  -3  0   0  0  0]
   [                                    ]
   [0  10  -12   0   0  0   0   0  0  0]
   [                                    ]
   [0  2   -3    0   0  0  -3  0  0  0]
   [                                    ]
   [0  0   10   -12  0  0   0   0  0  0]
```

The $n-j+1=4$ coefficients we are after will be computed as the determinants of four $2j \times 2j$ submatrices of s. All of these submatrices have the same first $2j-1$ columns, whereas the $2j$-th column is successively the $(2j-1+k)$-th column of s, where $k=1,...,n-j+1$.

Since $n=5$, $j=2$, all four rows of s have at least $2n-(n+1)-j+1=3$ trailing zeros. The first coefficient is 72:

```
Python] Det=int(det(s[0:4,0:4]) / 2)
Python] print(Det)
   72
Python] s.col_swap(3,4)
Python] Det=int(det(s[0:4,0:4]) / 2)
Python] print(Det)
   0
Python] s.col_swap(3,5)
Python] Det=int(det(s[0:4,0:4]) / 2)
```

```
Python] print(Det)
   0
Python] s.col_swap(3,6)
Python] Det=int(det(s[0:4,0:4]) / 2)
Python] print(Det)
   300
```

The second, third and fourth coefficients are, respectively, 0, 0 and 300. Therefore, the first remainder of the Sturm sequence is $r_1(x) = 72x^3 + 300$.

To see if we have to correct the sign of the polynomial or to change its coefficients to modified subresultants we have to evaluate expression (5) which now becomes

$$\frac{(-1)^{u_0}(-1)^{v_0}}{\text{sgn}(\varrho_0)^{p_0+p_1\text{- degDiffer}_1}} \cdot \frac{\text{Det}(i,k)}{\varrho_{\text{-}1}} \qquad (7)$$

The variables are as follows:

- $i = 1$, for the first remainder,
- $\varrho_{\text{-}1} = 2$, for the leading coefficient of $p(x)$,
- $\frac{\text{Det}(1,0)}{\varrho_{\text{-}1}} = 72$, as computed before, this implies that the degree of the first remainder is 3, that is $d_{r_1} = 3$,
- $\text{degDiffer}_1 = 0$, where in the terminology of Theorem 3, $\text{degDiffer}_1 = m_1 - d_{r_1} = 3 - 3 = 0$,
- $p_0 = 1$, the number of leading zeros in the derivative of $p(x)$ if the former is also considered of degree 5, this is equivalent to $p_0 = \deg(p) - \deg(p') = 1$, the difference in degrees between $p(x)$ and its derivative $p'(x) = 10x^4 - 12x^3$,
- $p_1 = 1$, since $p_1 = \deg(p') - d_{r_1} = 1$,
- $p\_\text{List} = [p_0] = [1]$, we need this list so that we can compute variable v,
- $u_0 = 1$, since $u_0 = 1 + 2 + \ldots + p_0 = 1$ as stated in Theorem 3,
- $u\_\text{List} = [u_0] = [1]$, each member of the list will be used as an exponent to -1 as shown in expression 5,
- $v_0 = 1$, since $v_0 = p_0 + p_1 + \ldots + p_0 = 1$ as stated in Theorem 3, and
- $\text{sgn}(\varrho_0) = \text{sgn}(10) = 1$, for the sign of the leading coeffcient of $p'(x)$.[16]

Replacing the variables in expression (7) with their values we obtain 72, which exactly matches, both in sign and value, the leading coeffcient of the remainder we computed by matrix triangularization. Hence, the first Sturmian remainder is $r^{(1)} = 72x^3 + 300$.

To compute the second Sturmian remainder we use rows 3 and 4 of SS to form the 3-rows matrices M, $M1$, $M2$ and from the last one we obtain a candidate for the second remainder, namely $1800x-2160$. Obviously, the second row of $M2$ will replace the 5-th row of SS but the 3-rd row of $M2$ will now replace the 7-th row of SS(Note that if we start enumeration with 0, the number of the row in SS that will be replaced is indicated by the number of leading zeros in the 3-rd row of M2). Row 6 in SS is the reduntant row and so can be replaced again by the second row of $M2$ rotated by one.

```
Python] M=Matrix([SS[3,:],rotate(SS[2,:],1),rotate(SS[3,:],1)])
Python] pprint(M)
   [0  0  0  72  0   0  300   0   0  0]
   [                                  ]
   [0  0  0  -6  0   0   0   -30  0  0]
   [                                  ]
   [0  0  0  0   72  0   0   300  0  0]
Python] M1=pivot(M,0,3)
Python] pprint(M1)
   [0  0  0  72  0   0  300     0     0  0]
   [                                      ]
   [0  0  0  0   0   0  1800  -2160  0  0]
   [                                      ]
   [0  0  0  0   72  0   0     300    0  0]
Python] M1.row_swap(1,2)
Python] pprint(M1)
   [0  0  0  72  0   0  300     0     0  0]
   [                                      ]
   [0  0  0  0   72  0   0     300    0  0]
   [                                      ]
   [0  0  0  0   0   0  1800  -2160  0  0]
Python] SS[4,:]=M1[1,:]
Python] SS[5,:]=Matrix([rotate(M1[1,:],1)])
Python] SS[6,:]=M1[2,:]
Python] pprint(SS)
   [2  -3   0   0   0   -3    0      0      0    0 ]
   [                                              ]
   [0  10  -12  0   0    0    0      0      0    0 ]
   [                                              ]
   [0  0   -6   0   0    0   -30     0      0    0 ]
   [                                              ]
   [0  0    0   72  0    0   300     0      0    0 ]
   [                                              ]
   [0  0    0   0   72   0    0     300     0    0 ]
   [                                              ]
   [0  0    0   0   0   72    0      0     300   0 ]
   [                                              ]
   [0  0    0   0   0    0   1800  -2160    0    0 ]
   [                                              ]
   [0  0    0   0   10  -12   0      0      0    0 ]
   [                                              ]
```

```
[0  0    0   0   2   -3    0       0     0   -3]
[                                               ]
[0  0    0   0   0   10   -12      0     0   0 ]
```

Again we compute the modified subresultants which will be used below for evaluation of expression (5).Now,we set $j=3$, we take the submatrix s of S, consisting of the first $2j=6$ rows of S and, since $n=5$, we will try to compute the coefficients of the polynomial remainder of degree $n-j=2$. All six rows of s have at least $2n-(n+1)-j+1=2$ trailing zeros. The first coefficient is 0 and the second, and third coefficients are, respectively, 2160 and -2592:

```
Python] s=Matrix(S[0:6,0:10])
Python] pprint(s)
```

```
[2  -3   0    0    0   -3  0   0   0  0]
[                                       ]
[0  10  -12   0    0   0   0   0   0  0]
[                                       ]
[0  2   -3    0    0   0   -3  0   0  0]
[                                       ]
[0  0   10   -12   0   0   0   0   0  0]
[                                       ]
[0  0    2   -3    0   0   0   -3  0  0]
[                                       ]
[0  0    0   10   -12  0   0   0   0  0]
```

```
Python] Det=int(det(s[0:6,0:6]) / 2)
Python] print(Det)
```

```
0
```

```
Python] s.col_swap(5,6)
Python] Det=int(det(s[0:6,0:6]) / 2)
Python] print(Det)
```

```
2160
```

```
Python] s.col_swap(5,7)
Python] Det=int(det(s[0:6,0:6]) / 2)
Python] print(Det)
```

```
-2592
```

Here, instead of a polynomial remainder of degree 2 we obtained one of degree 1, namely $r_2(x)=2160x-2592$.

If we next set j = 4, n=5 and take the submatrix and make the computations as before we will find another first degree remainder $r_3(x)=-64800x+77760$.

The appearance of multiple remainders of the same degree is quite normal, however, of those remainders with the same degree only the first one is used. The same phenomenon appears as well in the form of redundant rows as we show before.

To see if we have to correct the sign of the polynomial or to change its coeffcients to modified subresultants we have to evaluate again expression (5) which now becomes

$$\frac{(-1)^{u_1}(-1)^{u_0}(-1)^{v_1}}{\mathrm{sgn}(\varrho_1)^{p_1+p_2\text{- degDiffer}_2}\mathrm{sgn}(\varrho_0)^{p_0+p_1}}\cdot\frac{\mathrm{Det}(i,k)}{\varrho_{\text{-}1}} \qquad (8)$$

Below are the new variables and the ones that changed:

- $i=2$, for the second remainder,
- $\frac{\text{Det}(2,1)}{\varrho\text{-}1}=2160$, as computed before, this implies that the degree of the second remainder is 1, that is $d_{r_2}=1$,
- $\text{degDiffer}_2=1$, where in the terminology of Theorem 3, $\text{degDiffer}_2=m_2-d_{r_2}=2-1=1$,
- $p_2=2$, since $p_2=d_{r_1}-d_{r_2}=3-1=2$,
- $p\_\text{List}=[p_0,p_1]=[1,1]$, we need this list so that we can compute variable v,
- $u_1=1$, since $u_1=1+2+\ldots+p_1=1$ as stated in Theorem 3,
- $u\_\text{List}=[u_0,u_1]=[1,1]$, each member of the list will be used as an exponent to -1 as shown in expression 5,
- $v_1=2$, since $v_1=p_0+p_1+\ldots+p_1=1+1=2$ as stated in Theorem 3, and
- $\text{sgn}(\varrho_1)=\text{sgn}(72)=1$, for the sign of the leading coeffcient of the first remainder $r^{(1)}(x)$.

Replacing the variables in expression (8) with their values we obtain 2160, which means that the sign of the polynomial $1800x-2160$ was correctly computed by matrix triangularization. However, the values of the coeffcients are not modified subresultants. This is easily rectified by multiplying $1800x-2160$ times $2160/1800$. Indeed, we have

$$2160x-2592$$

and checking back with the computations above we found that -2592 is a modified subresultant. Therefore, the second Sturmian remainder is $r^{(2)}=2160x-2592$.

To compute the third, and final, remainder in the Sturm sequence we form matrix M, which now has 4 rows(that is, M becomes a matrix with (3+ degDiffer) rows). This is one of the new features in this extended method. The rows of M can be formed either from rows of SS or from the correctly computed remainders. Let's follow the first approach and form matrices M, $M1$, $M2$, $M3$. From $M3$, the triangularized form of matrix M we obtain as candidate remainder the constant 2475194112000, which replaces the last row of SS. The 2-nd and 3-rd rows of $M3$ replace, respectively, the 8-th and 9-th rows of SS.

```
Python] M=Matrix([SS[6,:],rotate(SS[6,:],1),rotate(SS[6,:],2),rotate(SS[5,:],1)])
Python] pprint(M)
   [0  0  0  0  0  0  1800  -2160    0       0  ]
   [                                            ]
   [0  0  0  0  0  0    0    1800   -2160    0  ]
   [                                            ]
   [0  0  0  0  0  0    0      0    1800   -2160]
   [                                            ]
   [0  0  0  0  0  0   72      0       0     300 ]
Python] M1=pivot(M,0,6)
```

```
Python] pprint(M1)
   [0  0  0  0  0  0  1800  -2160     0       0   ]
   [                                               ]
   [0  0  0  0  0  0   0     1800   -2160     0   ]
   [                                               ]
   [0  0  0  0  0  0   0      0     1800   -2160 ]
   [                                               ]
   [0  0  0  0  0  0   0    155520    0    540000]
Python] M2=pivot(M1,1,7)
Python] pprint(M2)
   [0  0  0  0  0  0  1800  -2160      0          0    ]
   [                                                   ]
   [0  0  0  0  0  0   0    1800     -2160        0    ]
   [                                                   ]
   [0  0  0  0  0  0   0      0       1800       -2160 ]
   [                                                   ]
   [0  0  0  0  0  0   0      0     335923200  972000000]

Python] M3=pivot(M2,2,8)
Python] pprint(M3)
   [0  0  0  0  0  0  1800  -2160    0          0       ]
   [                                                    ]
   [0  0  0  0  0  0   0    1800   -2160        0       ]
   [                                                    ]
   [0  0  0  0  0  0   0      0    1800       -2160     ]
   [                                                    ]
   [0  0  0  0  0  0   0      0      0    2475194112000]

Python] SS[6,:]=M3[0,:]
Python] SS[7,:]=M3[1,:]
Python] SS[8,:]=M3[2,:]
Python] SS[9,:]=M3[3,:]
Python] pprint(SS)
   [2  -3   0   0   0   -3   0      0      0          0       ]
   [                                                          ]
   [0  10  -12  0   0   0    0      0      0          0       ]
   [                                                          ]
   [0  0   -6   0   0   0   -30     0      0          0       ]
   [                                                          ]
   [0  0    0   72  0   0   300     0      0          0       ]
   [                                                          ]
   [0  0    0   0   72  0    0     300     0          0       ]
   [                                                          ]
   [0  0    0   0   0   72   0      0     300         0       ]
   [                                                          ]
   [0  0    0   0   0   0   1800  -2160    0          0       ]
   [                                                          ]
   [0  0    0   0   0   0    0    1800   -2160        0       ]
   [                                                          ]
   [0  0    0   0   0   0    0      0    1800       -2160     ]
   [                                                          ]
   [0  0    0   0   0   0    0      0      0    2475194112000]
```

Finally, for $j=5$ we evaluate the determinant of the whole matrix S to compute the constant term of the sequence. Its value is 11459232:

```
Python] print(int(det(S)/2))
   11459232
```

To see if we have to correct the sign of the constant polynomial 2475194112000 or to change it to a modified subresultant we have to evaluate again expression (5) which now becomes

$$\frac{(-1)^{u_2}(-1)^{u_1}(-1)^{u_0}(-1)^{v_2}}{\mathrm{sgn}(\varrho_2)^{p_2+p_3\cdot \mathrm{degDiffer}_3}\,\mathrm{sgn}(\varrho_1)^{p_1+p_2}\,\mathrm{sgn}(\varrho_0)^{p_0+p_1}}\cdot\frac{\mathrm{Det}(i,k)}{\varrho_{\text{-}1}} \tag{9}$$

Below are the new variables and the ones that changed:

- $i=3$, for the third remainder,
- $\dfrac{\mathrm{Det}(3,0)}{\varrho_{\text{-}1}}=11459232$, as computed before, this implies that the degree of the third remainder is 0, that is $d_{r_3}=0$,
- $\mathrm{degDiffer}_3=0$, where in the terminology of Theorem 3, $\mathrm{degDiffer}_3=m_3-d_{r_3}=0-0=0$,
- $p_3=1$, since $p_3=d_{r_2}-d_{r_3}=1-0=1$,
- $p\_\mathrm{List}=[p_0,p_1,p_2]=[1,1,2]$, we need this list so that we can compute variable v,
- $u_{i-1}=u_2=3$, since $u_2=1+2+\ldots+p_2=3$ as stated in Theorem 3,
- $u\_\mathrm{List}=[u_0,u_1,u_2]=[1,1,3]$, each member of the list will be used as an exponent to -1 as shown in expression 5,
- $v_{i-1}=v_2=4$, since $v_2=p_0+p_1+\ldots+p_2=1+1+2=4$ as stated in Theorem 3, and
- $\mathrm{sgn}(\varrho_2)=\mathrm{sgn}(2160)=1$, for the sign of the leading coeffcient of the second remainder $r^{(2)}(x)$.

Replacing the variables in expression (9) with their values we obtain -11459232, which means that our constant with correct sign is -2475194112000. Obviously, this value – computed by matrix triangularization – is not a modified subresultant. This is easily rectified by multiplying -2475194112000 times $11459232/2475194112000$. Indeed, we have

$$\frac{11459232}{2475194112000}\cdot(-2475194112000)=-11459232$$

and the third member of the Sturm sequence is $r^{(3)}=-11459232$.

Therefore the Sturm sequence of $p(x)=2x^5-3x^4-3$ is

$$[2x^5-3x^4-3,\,10x^4-12x^3,\,72x^3+300,\,2160x-2592,\,-11459232]$$

which agrees with the result obtained with the function sturm of Xcas.

```
> sturm( 2x^5 - 3x^4 - 3 )[1]

  [[2,-3,0,0,0,-3],[10,-12,0,0,0], [72,0, 0, 300], [2160,-2592],-1782139760640]
```

Note that the last term of the Sturm sequence in Xcas is not a modified subresultant, so it may differ in value but not in sign.

4.1 Implementation of Generalized Triangularization method for Sturm Sequences in Python

In this way I have implemented version 1 of the generalized matrix triangularization (VanVleck-Pell- Gordon) procedure in `Python`, with use of `Sympy`, in the function `sturmSeqVanVleckPellGordon`. This function also uses `sylvester2`, `Sq_free`, `row2poly`, `pivot`, `makelist` and `rotate` as does `sturmSeqVanVleck`. However, instead of Van Vleck's "sign rule", it uses the functions:

- `gaps`, which is activated when a pivot takes place in the process of triangularizing the 3-rows matrix M. It uses the theorem by Pell and Gordon to compute the correct sign of the remainder and also to force its coefficients to become modified subresultants,
- `compute_correct_sign`, used by gaps only when a pivot took place in a complete sequences. It also uses the Pell-Gordon theorem to determine the correct sign of the remainder.
- `compute_modified_subresultant`, used by gaps to compute the modified subresultant for the leading coefficient of the determinant.

The whole program for the generalized matrix triangularization (VanVleck-Pell-Gordon) procedure written in Python (version 2.7.5) is as follows:

```
import sympy as sp
import giacpy as gp    # an interface to be able to use from Python the Giac
                       # features, Giac is a Computer algebra system

from sympy.abc import x


def sylvester2(f, g, x):
    # Trivial case f = g = 0
    if (f == 0 or g == 0):
        return sp.Matrix([0])
    if (f == g and g == 0):
        return sp.Matrix([0])
```

```
    fp = sp.Poly(f, x).all_coeffs()
    gp = sp.Poly(g, x).all_coeffs()
    m, n = sp.degree( sp.Poly(f, x), x), sp.degree( sp.Poly(g, x), x)
    # Trivial case f, g are constants
    if (m == n and n == 0):
        return sp.Matrix([[1]])
    # order polys according to degree
    if (m < n):
        fp, gp = gp, fp
        m, n = n, m
    dim = 2*m
    M = sp.zeros( dim )
    k = 0
    for i in range( m ):
        j = k
        for coeff in fp:
            if (i == 0):
                M[i, j] = coeff
            else:
                M[2*i, j] = coeff
            j = j + 1
        j = m - n + k
        for coeff in gp:
            if (i == 0):
                M[i+1, j] = coeff
            else:
                M[2*i + 1, j] = coeff
            j = j + 1
        k = k + 1

    return M



def row2poly(row_E,deg):
    # a row of a matrix E where NOT all elements are coeffs of the poly
    # some entries at the begining and/or at the end are zero
    poly=[]
    length=len(row_E)
    # find the beginning of the poly,i.e. the 1-st non-zero element of the row
    k=0
    while row_E[k]==0:
        k=k+1
    # Once we have the beginning, append the next deg+1 elements to poly
    for j in range(0, deg + 1):
      if (k+j <= length):
         poly.append(row_E[k+j])
    return (poly)



def Sq_free(f):
    f=sp.expand(f)
    lst,q = sp.sqf_list(f)[1],1
    for i in range(len(lst)):
```

```
        q=q * lst[i][0]
    return(q)


def pivot(M, i, j):
    # M is a matrix, and M[i, j] specifies the pivot point.
    # All elements below M[i, j] in the j-th column will
    # be zeroed, if they are not already, according to
    # Bareiss' integer preserving transormation.
    Ma = M[:, :] # copy of matrix M
    rs = Ma.rows  # No. of rows
    cs = Ma.cols  # No. of cols

    for r in range(i+1, rs):
        if Ma[r, j] != 0:
            for c in range(j+1, cs):
                Ma[r, c] = Ma[i, j]*Ma[r, c] - Ma[i, c]*Ma[r, j]
            Ma[r, j] = 0

    return Ma



def compute_correct_sign(j):
    #  Only for complete sequences
    #  Determine correct sign for the even row E[j+3] using Pell-Gordon
    #   equation
    #  the sign of the first coefficient must be as computed by formula (1)

    global p_List,u_List,rho_List,v,EE,rho_List_Minus_1
    num=1
    den=1
    detEE=0
    #  update the p_List
    p_List.append(1)

    # compute the sign
    for n in range(0,len(u_List)):
        num=num*(-1)**u_List[n]
    num=num*(-1)**v
    for n in range(0,len(rho_List)):
        den=den*rho_List[n]**(p_List[n]+p_List[n+1])
    detEE=int(sp.det(EE[0:j+1,0:j+1]) / rho_List_Minus_1)
    rho=(num/den)*detEE
    sg=sp.sign(rho)
    return detEE,sg



def compute_modified_subresultant():
    # compute modified subresultant for lead. coeff of pDet
    global EE, degDiffer, expectedDeg, rho_List_Minus_1
```

```
    d=(EE.shape[0])-2*expectedDeg-1
    M3=EE[0:d+1,0:(EE.shape[0])]
    # swap column d with d+actualDeg because dets are 0 in between
    M3.col_swap(d,d+degDiffer)
    valDet=int(sp.det(M3[0:d+1,0:d+1]) / rho_List_Minus_1)

    return valDet


def gaps():
    # for zc >= 2 we insert in the to-be-triangularized matrix E
    # zc-1 additional copies of row M1[1] and zc copies of row M1[2],
    # all appropriately rotated and as long as dim(E) allow

    global j, E, EE, M1, polyList, zc, v
    global actualDeg, expectedDeg, currDeg, degDiffer
    global p_List, u_List, rho_List, rho_List_Minus_1

    #  update the rows of E
    # insert the appropriate rotated rows of M1 in matrix E until k=zc-2
    for k in range (0,zc-1):
        E[j+2+k,:]=sp.Matrix([(rotate(M1[1,:],(1+k)))])

    jj=j+zc      #at the end of this loop we are at row j+zc

    M1copy=sp.Matrix(M1)  # we create a copy of M1 matrix

    # matrix E is now updated with the copies of rows M1[1] and M2[2]
    # now we have to compute the correct sign

    if zc==2 :
        valDet,sg=compute_correct_sign(jj+1)

    if zc>2 :
         #  update the p_List
         p_List.append(1+degDiffer)
         num1=1
         for n in range(0,len(u_List)):
             num1=num1*(-1)**u_List[n]
         num=num1*(-1)**v
         den1=1
         for n in range(0,len(rho_List)):
             den1=den1*rho_List[n]**(p_List[n]+p_List[n+1])
         den= den1* rho_List[len(rho_List)-1]**(p_List[len(rho_List)-1] +
p_List[len(rho_List)]-degDiffer)

         # compute modified subresultant for lead. coeff of pDet
         valDet=compute_modified_subresultant()
         rho=(num/den)*valDet
         sg=sp.sign(rho)

    if sp.sign(M1copy[2,jj]) != sg :
        M1copy[2,:]=(-1)*M1copy[2,:]

    # Force coeffs of pDet to become modified subresultants
```

```
    pDet=row2poly(M1copy[2,:],actualDeg)
    q=(abs(valDet / pDet[0]))
    pDet2=[]
    for i in range(0,len(pDet)):
        pDet2.append(q*pDet[i])
    polyList.append(pDet2)

    #  update the rho_List, u_List and v
    rho_List.append(sg)
    u_List.append(sum(makelist(1,p_List[len(p_List)-1],1)))
    v=sum(p_List)

    #compute the final negated remainder
    expectedDeg=actualDeg-1
    M=[]
    for k in range(0,zc):
        E[j+zc+1+k,:]=sp.Matrix([(rotate(M1copy[2,:],(1+k)))])
        if (expectedDeg< 0) and (j+zc+1+k==(E.shape)[0]-1):
            DONE=1
            return DONE
        M.append(E[j+zc+1+k,:])

    j=j+2*zc+1
    # last row of M
    M.append(rotate(E[jj,:],1))
    M=sp.Matrix(M)

    # triangularize M
    M=pivot(M,0,jj+1)
    for k in range (1,zc):
       M=pivot(M,k,jj+1+k)

    # insert last row in matrix E
    E[j,:]=(-1)*sp.Matrix([(M[zc,:])])

    # determine actual degree of E[j,:]
    # count how many leading zeros are in the last row of E
    zcc=0
    n=j
    while(E[j,n]==0):
                n=n+1
                zcc=zcc+1
                if n>j: break
    actualDeg=expectedDeg-zcc
    degDiffer=expectedDeg-actualDeg

    # determine correct sign of this remainder (E[j,:]) using the Pell-Gordon
formula

    #  update the p_List
    p_List.append(1+degDiffer)
    num1=1
    for n in range(0,len(u_List)):
        num1=num1*(-1)**u_List[n]
    num=num1*(-1)**v
    den=1
```

```
    for n in range(0,len(rho_List)):
        den=den*rho_List[n]**(p_List[n]+p_List[n+1])

    if zc == 2 :
        valDet=int(sp.det(EE[0:j+1,0:j+1]) / rho_List_Minus_1)
    if zc > 2 :
        # compute modified subresultant for lead. coeff of pDet
        valDet=compute_modified_subresultant()

    rho=(num/den)*valDet
    sg=sp.sign(rho)

    if sp.sign(E[j,j]) != sg :
        E[j,:]=(-1)*E[j,:]

    # Force coeffs of pDet to become modified subresultants
    pDet=row2poly(E[j,:],actualDeg)
    q=(abs(valDet / pDet[0]))
    pDet2=[]
    for i in range(0,len(pDet)):
        pDet2.append(q*pDet[i])
    polyList.append(pDet2)

    #  update the rho_List, u_List and v
    rho_List.append(sg)
    u_List.append(sum(makelist(1,p_List[len(p_List)-1],1)))
    v=sum(p_List)

    if j >= (E.shape[0])-2 :
        DONE=1
        return DONE



def makelist(start,stop,inc):
    value=start
    result=[]
    while value <= stop:
        result.append(value)
        value=value + inc
    return result



def rotate(l,n):
    return l[-n:] + l[:-n]



def sturmSeqVanVleckPellGordon(pp):
    # Zero will be removed as root of the the polynomial. Moreover,
```

```
# if the lead. coef. LC of the poly is negative, i.e. coeffs(p)[0] < 0
# make p := (-1)*p, work with it and at the end multiply each member

# of the Sturm sequence times -1.

# FLAG is set if sign(LC) < 0, pv is set if a pivot took place
# M is the matrix to be triangularized


# initialize
global j, E, EE, M1, polyList, zc, v
global actualDeg, expectedDeg, currDeg, degDiffer
global p_List, u_List, rho_List, rho_List_Minus_1
p=pp
FLAG=0
pv=0
DONE=0
E=0
polyList=[]

# Zero degree polynomial?
if (type(p)==int):
    print("Maybe zero has been divided out. Poly is of degree 0.")
    return (p)

# the variable of the poly
var=sp.Poly(pp).free_symbols
# leading coefficient of polynomial
if(sp.Poly(pp).LC()<0):
    p=(-1)*p
    FLAG=1
# make sure p is square free
spp=sp.sqf(pp)
if (spp != sp.simplify(pp)):
    p=Sq_free(pp)

# Decrease the degree of the poly if 0 is a root
while (p).subs(x,0)==0 :
    p=sp.simplify(p/x)

# Divide by the content
p=sp.simplify(p/sp.content(p))

n=sp.degree(p,x)
p2=sp.diff(p,x)
polyList.append(sp.Poly(p,x).all_coeffs()) #convert(symb2poly(p),list))
polyList.append(sp.Poly(p2,x).all_coeffs()) #convert(symb2poly(p2),list))

# First degree polynomial?
if (n==1):
    if (FLAG==1):
        # polyList= (-1)*polyList
        for j in range (0 , len(polyList)):
            for i in range(0,len(polyList[j])):
                polyList[j][i]=(-1)*polyList[j][i]
        return polyList
```

```
        else: return (polyList)

    # form the Sylvester matrices E and EE
    E=sylvester2(p,p2,var)
    # EE does not change, to compute modified subresultants
    EE=sylvester2(p,p2,var)

    # Initialize Pell-Gordon variables
    rho_List_Minus_1=sp.Poly(p,var).LC()
    rho_List=[sp.sign(sp.Poly(p2,var).LC())]   # Only the signs are needed
    p_List=[1]
    u_List=[sum(makelist(1,p_List[0],1))]
    v=sum(p_List)
    currDeg=n-1     # current degree
    expectedDeg=n-2

    j=0
    while True:
        # form and triangularize the 3-rows matrix

        M=0
        M=sp.Matrix([rotate(E[j+1,:],-1),E[j,:],E[j+1,:]])
        M1=0
        # Van Vleck removes the content at each step

        # However, as seen below, we force the coefficients to become modified
subresultant
        M1=pivot(M,0,j)

        if (M1[1,j+1]==0):
            M1.row_swap(1,2)
            pv=1

        if pv:
            # A pivot ocurred

            # count how many leading terms are missing, that is count how many
leading
            # zeros are in the LAST row of M1
            # counting STARTS from k=j

            zc=0
            k=j
            while(M1[2,k]==0):
                k=k+1
                zc=zc+1
            #Compute actual degree of poly in the row
            if zc == 2 :
                 actualDeg=expectedDeg
            if zc > 2 :
                 actualDeg=expectedDeg-(zc-2)

            degDiffer=expectedDeg-actualDeg
            # check if pivot was done as in p=x^3-7x+7 (zc==2) OR not (zc > 2)
            if zc >= 2 :
                gaps()
```

```
                if DONE==1:
                    break

        if pv!=1 :
            #  In this case there was no pivot and hence the actual degree is
as expected

            actualDeg=expectedDeg
            degDiffer=expectedDeg-actualDeg
            M2=0
            # see comment for M1 above
            M2=pivot(M1,1,j+1)
            # update the rows of E
            E[j+2,:]= sp.Matrix([(rotate((M2[1,:]),1))])
            E[j+3,:]= sp.Matrix([(rotate((M2[2,:]),1))])

            #  change the signs when necessary
            valDet,sg=compute_correct_sign(j+3)

            if sp.sign(E[j+3,j+3]) != sg :
                E[j+3,:]=(-1)*E[j+3,:]

            pDet=row2poly(E[j+3,:],actualDeg)
            q=(abs(valDet / pDet[0]))
            pDet2=[]
            for i in range(0,len(pDet)):
                pDet2.append(q*pDet[i])
            polyList.append(pDet2)

            #  update the rho_List, u_List and v
            #  sg & p_List have been set in compute_correct_sign() or gaps()
            rho_List.append(sg)
            u_List.append(sum(makelist(1,p_List[len(p_List)-1],1)))
            v=sum(p_List)
            j=j+2
        if pv :
            j=j-1
        pv=0
        currDeg=actualDeg
        expectedDeg=expectedDeg -1

        if j >= (E.shape[0])-2 or expectedDeg <0:
            DONE=1

        if DONE==1: break

    res=E

    if FLAG == 1:
        res=(-1)*res
        for j in range (0 , len(polyList)):
            for i in range(0,len(polyList[j])):
                polyList[j][i]=(-1)*polyList[j][i]

    return polyList
```

```
p = 2*x**5 - 3*x**4 -3

print sturmSeqVanVleckPellGordon(p)

print gp.sturm(p)[1]      #check the results with the fuction sturm of giacpy
```

5. References

[1] Akritas, A.G., Malaschonok, G.I., Vigklas, P.S.: "On a Theorem by Van Vleck Regarding Sturm Sequences" (2014)

[2] Akritas, A.G.: "A New Method for Computing Polynomial Greatest Common Divisors and Polynomial Remainder Sequences". Numerische Mathematik, 52, (1988), 119-127.

[3] Akritas, A.G.: Elements of Computer Algebra with Applications. John Wiley Interscience, New York, (1989).

[4] Akritas, A.G.: "Sylvester's Forgotten Form of the Resultant." Fibonacci Quarterly, 31, (1993),325-332.

[5] Akritas, A.G., Akritas, E.K, Malaschonok, G.I.: "Matrix Computations of Subresultant Polynomial Remainder Sequences in Integral Domains." Reliable Computing, 1, (1988), 375-381.

[6] Akritas, A.G., Malaschonok, G.I.: "Fast Matrix Computation of Subresultant Polynomial Remainder Sequences." Computer Algebra in Scientific Computing - CASC 2000, Springer Verlag, Heidelberg, (2000), 1-11.

[7] Akritas, A.G., Malaschonok, G.I., Vigklas, P.S.: "Sturm Sequences and Modified Subresultant Polynomial Remainder Sequences". Submitted for publication.

[8] http://www.sympy.org/en/index.html

[9] http://docs.sympy.org/latest/index.html

[10] http://docs.sympy.org/0.7.2/modules/matrices/expressions.html

[11] Pell, A.J., Gordon, R.L.: "The Modified Remainders Obtained in Finding the Highest Common Factor of Two Polynomials". Annals of Mathematics, Second Series, 18(4), (Jun., 1917), 188-193.

[12] Sylvester, J.J.: "A method of determining by mere inspection the derivatives from two equations of any degree". Philosophical Magazine, 16, (1840), 132-135.

[13] Sylvester, J.J.: "On the Theory of Syzygetic Relations of Two Rational Integral Functions,

Comprising an Application to the Theory of Sturm's Functions, and that of the Greatest Algebraical Common Measure". Philosophical Transactions, 143, (1853), 407-548.

[14] Van Vleck, E.B.: "On the Determination of a Series of Sturm's Functions by the Calculation of a Single Determinant". Annals of Mathematics, Second Series, 1(1/4), (1899 - 1900), 1-13.

[15] Akritas, A.G.: "A Simple Proof of the Validity of the Reduced PRS Algorithm". Computing, 38, (1987), 369-372.

[16] http://en.wikipedia.org/wiki/Python_(programming_language)

[17] http://en.wikipedia.org/wiki/SymPy

[18] http://www.codecademy.com/en/tracks/python

[19] http://www.python-course.eu/index.php

[20] http://www.tutorialspoint.com/python/python_functions.html

[21] http://docs.sympy.org/0.7.2/modules/matrices/expressions.html

[22] http://www.i-programmer.info/programming/python/3942-arrays-in-python.html

[23] http://stackoverflow.com/questions/tagged/python

[24] Bareiss, E.H.: "Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination". Mathematics of Computation, 22, (1968), 565-578.

[25] Dodgson, C.L.: "Condensation of Determinants". Proceedings of the Royal Society of London, 15, (1866), 150-155.

[26] von zur Gathen, J., Lucking, T.: "Subresultants Revisited". Theoretical Computer Science, 297, (2003), 199-239.