Counting forms: a step towards classifying sincere weakly positive forms

Danyil Bohdan

We consider the task of classifying sincere weakly positive forms over integers. We examine the means by which such forms can be obtained computationally and suggest an algorithm that uses these means. By implementing the algorithm within a Python computer program the number of sincere weakly positive forms for $n \leq 8$ is obtained. We then review the results and consider the further course of research.

Introduction

We define a *form* as a polynomial

$$\chi(X_1, \ldots, X_n) = \sum_{i=1}^{n} X_i^2 + \sum_{i<j} \chi_{ij} X_i X_j$$

over integers where $\chi_{ij} = 0$ for $j < i$ and $\chi_{ii} = 1, i = \overline{1, n}$. The form χ is said to be *weakly positive* if for all $z \in \mathbb{Z}^n$ such that $z > 0$ it holds that $\chi(z) > 0$. The form is called *sincere* if there exists $z = (z_1, \ldots, z_n) \in \mathbb{Z}^n$ such that $z_i > 0, i = \overline{1, n}$ and $\chi(z) = 1$. Such z is then called a *sincere root* of this form. We'll use the word "form" to refer to the polynomial itself, its upper triangular *incidence matrix* $(\chi_{ij})_{i,j=1}^{n}$and the well-know bigraph representation (see [2] for details) that translates variables into vertices $1, \ldots, n$ and values of χ_{ij} into the number of edges between the corresponding vertices with the edges representing χ_{ii} omitted. In case when χ_{ij} is < 0 it is represented with $-\chi_{ij}$ dotted edges while $\chi_{ij} > 0$ takes the form of χ_{ij} solid edges.

We define the *reflection* $\sigma_k(z)$ of vector $z = (z_1, \ldots, z_n)$ with respect to vertex $v_j, j \in \{1, \ldots, n\}$ of the form's graph as follows:

$$z_k = \begin{cases} -\sum_{l \neq k} (\chi_{lk} + \chi_{kl}) z_l - z_j & k = j \\ z_k & k \neq j \end{cases}$$

Per [1] we can say that the number of sincere weakly positive forms of n variables is finite for each given n and for each form's coefficients $(\chi_{ij})_{i,j=1}^{n}$ it holds that $\chi_{ij} \in \{-1, 0, 1, 2\}$

This gives rise to the problem of finding out the exact number of forms that exist for each value of n. In the current paper we will try to tackle this problem.

Algorithms

We will construct an algorithm that provides as its output the total number of forms for each $n < N$. Such algorithm can be represented as a conditional loop with several subroutines.

Main loop:

1. Initialise the queue with the form $\left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$, the only one that fits our criteria for $n = 2$.
2. Until the queue is empty: take the first form Q from the queue.
   (a) Find the roots of Q. If their number is finite, proceed; if not, go back to 2.
   (b) Derive the "progeny" of Q from it, that is to say, forms that are derived from Q by extending it in a specific way. Add the result to the queue if its size is less than the predetermined maximum size N and unconditionally to the output.

Finding a form's "progeny" is constructed as a separate method. It works as follows:

1. Initialise the list L as empty.

2. For each root $r = (r_1, \ldots, r_n)$ of a given form Q of size n, for each vector $z = (z_1, \ldots, z_n)$ from $(-1, \ldots, -1)$ to $(2, \ldots, 2)$ do:
   (a) Check if $\sum_{i=1}^{n} z_i r_i = 1$. If the equality holds then r reflects into zero with respect to z and extending the form's graph with a vertex of z incidence will maintain sincerity. In this case we proceed. If not, we go back and pick the next vector.
   (b) Create Q' by extending Q to size $n+1$ while appending the transposed vector z as a column to its right: $Q' = \binom{Q\ z^{\mathrm{T}}}{\vec{0}\ 1}$.
   (c) If Q' has a finite number of roots ($\Leftrightarrow$ it's weakly positive [1]) and doesn't have an equivalent form that's already in L, append it to L.

Determining if two forms are equivalent up to vertex permutation based upon their matrices is one of the most strenuous tasks that the program has to perform (processor time-wise). We will use cached signatures as heuristics for the initial comparison. The first signature is a vector that consist of the number of edges of each incidence value (-1, 0, 1 and 2). These vectors matching doesn't guarantee that the forms are the same but them differing is a sure way to tell that they are not. Should these match we compare the number of solid and dotted edges each vertex has. Finally, if all of above fails to produce a negative answer, we look up cached representations of both forms as sets of their edges' incidences[1] and then iterate over of all permutations $\pi_i, i = \overline{1, n!}$ to see if applying π_i to the vertices of form A makes its edge set match form B.

Above we've mentioned determining the number of roots a form has. We obtain this number using reflections. To do so we start out off with a queue initially filled with base vectors $\{e_i\}_{i=1}^{n}$ (which we know are roots no matter what). We then go on to take the root $r = (r_1, \ldots, r_n)$ from the beginning of the queue and obtain the set $P = \{\sigma_k(r)\}_{k=1}^{n}$, where

$$\sigma_k(r) = (r_1, \ldots, r_{k-1}, -\sum_{k \neq l} (\chi_{lk} + \chi_{kl}) r_l - r_k, r_{k+1}, \ldots, r_n).$$

For every $p = (p_1, \ldots, p_n) \in P$ such that p is a weakly positive root of χ that isn't already in the queue we add p to the back of the queue. This continues until the queue is exhausted. If at any point of this procedure such i is found that $|r_i - p_i| \geq 2$ then the number of roots is infinite [1]. For a from χ that has a finite number of roots, however, this procedure will necessarily terminate.

Implementation

Using the Python programming language we were able to prototype rapidly a computer program that, given the list of sincere weakly positive forms for n, obtains the forms for $n+1$. This led to our queue-based design. Initially we took an object-oriented approach with forms, vectors, etc., being represented by specific custom classes. However, later into development it was established experimentally that creating new objects within loops of the most heavily used functions (the ones responsible for comparing forms and finding a form's roots) was the main bottleneck of the program. Although objects led to better structured and more readable code we had to get rid of most of them. Thus, only forms were left as objects at the end.

Some of the conclusions worth mentioning about computation-heavy Python software that we have reached while developing this program include:

1. Using the JIT compiler Psyco removes much of the function calling overhead but not the overhead related to creating objects.
2. Automated Python-to-C translation using tools such as Cython proves only marginally helpful even with manual typing for a program that's heavily reliant upon complex built-in types.

[1] I.e., $\{(1, 1, \chi_{1,1}), (1, 2, \chi_{1,2}), \ldots, (n-1, n, \chi_{n-1,n})\}$ where $\{(1, 1, 1), (2, 2, 1)\}$ and $\{(2, 2, 1), (1, 1, 1)\}$ are considered the same.

3. Always try to replace custom code with standard Python functions written in C as these generally execute much faster. This is supported by Python's creator himself. [3]

Although the program proved initially unwieldy due to being written in an interpreted language through optimisation and establishing the the optimal interpreter empirically (CPython 2.6.6) we've been able to reduce execution time for $n = 6$ from over 20 minutes with the initial version to nearly 1 second with the latest using the same hardware.

Results

The number of sincere weakly positive forms is as follows:

n	#
2	1
3	1
4	3
5	11
6	100
7	1609
8	42816

It took 8690.98 seconds, that is, approximately 2 hours 24 minutes and 51 seconds, to obtain this result on a regular home computer.

Conclusion

We have constructed a working algorithm and a software program that allows to compute the number of sincere weakly positive forms for any given n. During the computation the forms that are obtained can be saved to provide material for identifying the patterns by which they grow and series they fall into as was done in [1] for unit forms. This program demonstrates that despite our problem differing significantly from the typical computational tasks the techniques used to simplify and optimise the code remain standard.

As the table above makes clear, the number of forms grows very fast as the number of variables n increases. If the results obtained for unit forms are to be assumed analogous we can expect this number to plateau after some $n_0 > 15$ and all the subsequent forms to fall into series. This brings us to conclusion that further study is necessary and there is a natural way to proceed with it. Simply by going from a Python prototype to a production-grade C++ program we expect to be able to reach $n = 9$ or greater without altering the flow of the program; however, further on specific algorithmic optimisations may be needed.

References

[1] P. Dräxler, Yu. A. Drozd, N. S. Golovachtchuk, S. A. Ovsienko, M. M. Zeldych. Towards the classification of sincere weakly positive unit forms. Preprint 92 - 087, Universität Bielefeld, 1992.

[2] N. S. Golovashuk. Maximal non degenerated forms. Quadratic Forms in the Representation Theory of Finite-Dimension Algebras (Surveys and Work in Progress), Workshop "Representation Theory", Universität Bielefeld, 1995.

[3] Guido van Rossum. Python Patterns - An Optimization Anecdote. Web, 2005. http://www.python.org/doc/essays/list2str.html

Authors

Danyil Bohdan — 2nd year Master's student, The Faculty of Mechanics and Mathematics, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine; E-mail: *danyil.bohdan@gmail.com*