Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 539–544

ISBN 978-83-60810-51-4

Solving Systems of Polynomial Equations on a GPU

Robert A. Kłopotek
Institute of Computer Science
Polish Academy of Sciences,
ul. Jana Kazimierza 5,
01-237 Warsaw, Poland
Email: robert@klopotek.com.pl

Joanna Porter-Sobieraj
Faculty of Mathematics and
Information Science, Warsaw
University of Technology,
ul. Koszykowa 75,
00-661 Warsaw, Poland
Email: j.porter@mini.pw.edu.pl

Abstract—This paper explores the opportunities of using a GPGPU to solve systems of polynomial equations. We propose numerical real root-finding based on recursive de Casteljau subdivision over an n-dimensional rectangular domain. Two variants of parallelism - multithreading and multiprocessing - have been investigated. The speed, memory consumption and resistance for different sets of input data have also been examined.

I. Introduction

SYSTEMS of algebraic equations are widely used in many fundamental problems in computer-aided design and manufacturing, engineering, robotics, and computer graphics. They are essential in modeling, simulation, and optimization problems. Most approximation and interpolation methods employ piecewise polynomial functions over a given domain.

While finding roots for polynomials is a well-explored area, solving systems of such equations is still a serious challenge. They can have many solutions relative to the number and degrees of equations. Furthermore, in many cases only a few roots, e.g. those lying in a specific region, are the subject of interest. In such situations, a systematic search through a solution space and excluding areas not containing any roots seems to be a successful approach.

A method based on a multidimensional bisection algorithm using the multivariate Bernstein-Bezier representation, convex hull property and de Casteljau subdivision provides the approximate location of all the system roots in a given bounded domain [1]. The effectiveness of the algorithm can be significantly improved by early elimination of regions of roots that have already been isolated [2]-[4].

The rapid increase of GPU computing power means that this well-conditioned numerical algorithm, based on the analysis of Bernstein coefficients, may be useful, despite its exponential computational complexity. In recent years, many publications have proven that such an approach works well for surface intersection calculation or for the visualization of curves and surfaces [5]-[6]. They have been confirmed mostly for one or two equations and the computation of single precision floating-point values that are sufficient enough for computer graphics purposes.

Our goal is to investigate the usefulness of GPU processing in the case of larger polynomial systems and double precision floating-point arithmetic, which are required by CAD/CAM/CAE systems.

The paper is organized as follows: in Section 2 we briefly introduce the theoretical background for a multidimensional bisection algorithm. In the next sections we discuss the details of our implementation and the numerical results obtained for various sets of equations. Section 5 provides conclusions drawn from the presented analysis.

II. Problem Formulation

Consider a set of n polynomial equations in n independent variables

$$\mathbf{F}(\mathbf{x}) = \mathbf{0} \tag{1}$$

where $\mathbf{F} = (f_1, f_2, \ldots, f_n)$: $[0,1]^n \rightarrow R^n$. The problem is to calculate numerically, with a given accuracy ε, all the real roots $\{\mathbf{x_0}\}$ of the system (1).

The multidimensional bisection method discussed here is based on the Bernstein-Bezier tensor representation. Each polynomial f_k is in the form

$$f_k(\mathbf{x}) = \sum_{i_1=0}^{m_1^{(k)}} \cdots \sum_{i_n=0}^{m_n^{(k)}} b_{i_1 \ldots i_n}^{(k)} B_{i_1}^{m_1^{(k)}}(x_1) \ldots B_{i_n}^{m_n^{(k)}}(x_n) \tag{2}$$

where all polynomials $B_{i_j}^{m_j^{(k)}}(x_j)$ are Bernstein polynomials of the form

$$B_i^m(x) = \binom{m}{i} x^i (1-x)^{m-i} \tag{3}$$

and $b_{i_1 \ldots i_n}^{(k)}$ are called Bernstein coefficients [7].

The properties of the multivariate tensor-product Bernstein basis – most of all the convex-hull property over a unit box $[0,1]^n$ – are essential in the root-finding algorithm. A sufficient condition for excluding the existence of roots in a given domain $[0,1]^n$ is that all Bernstein coefficients of any

polynomial f_k are of the same sign. If all the polynomials f_k have Bernstein coefficients of both signs, the corresponding domain is suspected to contain a solution of an equations system (1) and it has to be subdivided further.

The solver routine consists of a queue formed by the areas (boxes) to be processed. If the considered area is suspected to have roots of a polynomial system, it is then subdivided along the consecutive variable x_j into two boxes. This subdivision involves the de Casteljau algorithm to calculate sets of Bernstein coefficients for both resulting subdomains. Such a division implies that both new areas are scaled into a unit box. Therefore, not only Bernstein coefficients, but also additional information about the current area's coordinates relative to an initial unit box have to be stored. A subdivision along only one variable simplifies de Casteljau algorithm implementation, and a cyclical selection of dividing coordinate x_j guarantees the decrease in diameter of subsequent subdomains.

Both new boxes together with their Bernstein coefficients are placed as one data package in a queue of regions suspected to contain a root and a consecutive data package is removed from the queue to be processed.

The root-location process is completed when the tolerance of subdivision resolution ε for all subdomains is obtained or all areas have been excluded as not containing any system solutions. This corresponds to emptying the queue.

III. Algorithm Pseudocode

The whole computation procedure was divided into 3 stages: first converting polynomial in power basis into Bernstein basis on CPU and sending converted polynomials to GPU, second making de Casteljau division on GPU, third sending back results from GPU to CPU and converting it to power basis.

First stage on CPU:

1. Scale domain from interval in R^n to multidimensional box $[0,1]^n$
2. Convert polynomials in each equation from power basis to Bernstein basis
3. Optimization step: pre-divide equation system using de Casteljau method so that each thread block have one box
4. Send converted equations to GPU
5. CPU starts a CUDA program on GPU

Second stage on GPU for each thread block (each block of threads has its own queue):

1. Initialize queue with the obtained box.
2. For all threads in block in parallel
   a. Get box from queue
   b. Divide box using de Casteljau method
   c. Check existence of roots
   d. Put into queue boxes which may contain roots
3. Check end conditions (emptiness of queue, size of boxes, number of boxes analyzed, etc.). If they are not satisfied go to 2. Else go to 4.
4. Send boxes queue from GPU to CPU

Note that due to queuing different threads at the same moment may handle boxes of different sizes.

In the literature, e.g. in [11], one can find other parallel solvers for polynomial equation systems. However, they do not rely solely on GPUs when searching through the box hierarchy, delegating hierarchy traversal to the CPU. So only one step of box processing is executed in parallel on GPUs and then the control returns to CPU. Our novel algorithm makes use of the new feature of dynamic memory allocation on CUDA that allows for processing of boxes in a loop solely on GPUs (GPUs perform the traversal) which exploits thoroughly the natural parallelism of the task itself.

In the third stage the CPU is responsible for collecting of the results and does not control search through the boxes.

Third stage on CPU:

1. Merge results from all blocks.
2. Scale boxes to initial domain.
3. As root solution take midpoint of each box.
4. Check if this point is a solution with a given accuracy epsilon.

IV. Implementation Notes

The algorithm presented in the previous section has been implemented mainly on a GPU with the use of the CUDA parallel programming model. The main concepts and an extensive description of this technology are given in [8] and [9].

Modern graphics cards contain hundreds of cores that execute tasks in parallel. The threads are logically grouped into so-called blocks, and threads from the same block are always executed on the same multiprocessor. Therefore, threads from the same block can and threads from different blocks cannot cooperate effectively. A GPU also has its own memory hierarchy. According to its speed, it looks as follows: registers, cache, shared memory, constant memory and the slowest – global memory.

The algorithm described for root isolation seems to be suitable for CUDA parallelization. Distinct boxes can be processed independently, and the exclusion tests together with the subdivision procedure are of a very simple form.

The simplest way of parallelization is using only one block of threads. The threads are then executed in parallel in groups of 32. The memory is shared for all of them and synchronization while placing data packages in, and removing them from the queue can easily be done.

The more efficient parallelization divides the threads into many blocks to execute many independent de Casteljau subdivisions on different available multiprocessors at the same time.

Our preliminary version of the solver has been written for NVIDIA GPUs with compute capability 2.x, using NVIDIA Computing SDK 3.2 for 64-bit Windows 7. To begin with, all necessary data for a given system of polynomial equations, such as coefficients and computation tolerance, are copied into the GPU's global memory. Then, a CPU initializes one or more blocks of a required number of threads, and dynamic structures, such as queues of boxes and sets of corresponding Bernstein coefficients, are allocated. Afterwards, each thread executes a de Casteljau subdivision and exclusion test for one data package which has been removed from the corresponding queue. When calculations for a block have been completed, a CPU copies the results from the GPU's global memory to RAM and then, they are joined together to get a final solution to a given set of equations.

All the calculations are done in double precision floating-point arithmetic. Single precision floating-point computing, although more efficient than the double type, does not guarantee sufficient accuracy for an arbitrary location of the roots of the system. Many times in single precision computation algorithm did not find root due to numerical errors. Due to numerical errors bisection algorithm ran differently for single and double precision. In single precision it rejected too many boxes and worse boxes with roots in it. Hence comparison between single precision and double precision is not insightful.

Secondly, the efficiency of this approach has also been decreased through applying dynamic allocation of memory for corresponding data structures. In this algorithm, the number of boxes that are suspected to contain the roots is unknown. Therefore, dynamic allocation, although it is slower than allocation of global memory of a given size, seems to be more reasonable. When a thread makes de Casteljau subdivision in GPU part of algorithm, it must allocate memory for division result. If checking box for containing a root is positive, then this thread puts box into queue and otherwise it frees memory. Memory allocated by thread is only available for threads inside a block.

The amount of memory used by a GPU is proportional to the size of the task (measured by the number and the maximum degree of equations), which influences the number of Bernstein coefficients. Table 1 shows the amount of memory for a data package with coefficients for one analyzed box for a different number of equations n and the maximum degree of equations m in each variable separately. It should be noted, that due to the tensor basis and a box's domain, the corresponding degree of the equation may even be equal to $n \cdot m$.

Note that for a given equation system m, n are fixed and therefore box descriptor size is constant.

V. Experimental Results

The numerical experiments were performed on a PC with an NVIDIA GeForce 460 GTX 1 GB graphics card with 336 CUDA cores. It is suggested to leave at least 10% of the memory for stack operations; therefore, the size of the heap for all the tests was set to 768 MB.

TABLE I.
MEMORY SIZE (IN BYTES) OF A SINGULAR BERNSTEIN COEFFICIENTS PACKAGE FOR A DIFFERENT NUMBER OF EQUATIONS *N* WITH MAXIMUM DEGREE *M* IN EACH VARIABLE SEPARATELY

n	**3**	**4**	**5**	**6**	**7**	**8**
m=1	240	576	1,360	3,168	7,280	16,512
m=2	696	2,656	9,800	35,088	122,584	420,032

The goal of our research was to compare the time and memory performance of a parallel GPU solver and its resistance for different sets of input data and their execution configuration.

The input systems of equations differed in the number n of variables and equations (ranging from 3 to 8), the maximum degree m (1 or 2) in each variable separately, and the level of their linearity.

Note that even when degree m=1 it does not mean, that an equation system is linear. If you have, say 7 variables, then the equation degree can be up to 7. The linearity control means that we restricted in a given experiment the coefficients of nonlinear terms to be by some factor smaller than the coefficients of linear terms. Anyway linearity control does not mean linear equations.

It should be stressed here that limiting the systems to only 8 quadratic equations in each variable separately, results in multi-quadratic polynomials of degree equal even to 16. Moreover, such systems are of the utmost importance not only in modeling and constraint programming, but also in robotics associated with the inverse kinematic problem. It has been proven that for a general revolute 6-DOF robot manipulator the inverse kinematic problem can be reduced to solving a system of 8 second-degree polynomial equations in 8 unknowns [10].

The speed of the calculations was ascertained by measuring the average time of processing a box, and memory consumption – by the number of subdomains that had been processed. The tolerance for the subdivision resolution ε was equal to 10^{-6}.

In this paper we present only a portion of the numerical results obtained. Fig. 1 and Fig. 2 show the time needed for processing a box depending on the number of threads.

For multilinear sets of equations (nx1) the time increases with the number of threads, and for quadratic equations in each variable (nx2) – it decreases for a number of threads less than 8 and a number of equations greater than 6. This shows that, depending on the equation system complexity (which influences the amount of data to be processed), it is worth using only a limited number of threads. An excessive number of threads generates only additional costs for managing the threads not used for calculating.

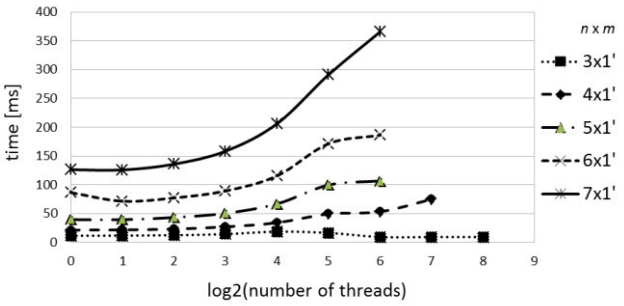

Fig. 1 The average time for processing a box for different numbers of threads in a block and systems of linear equations in each variable.

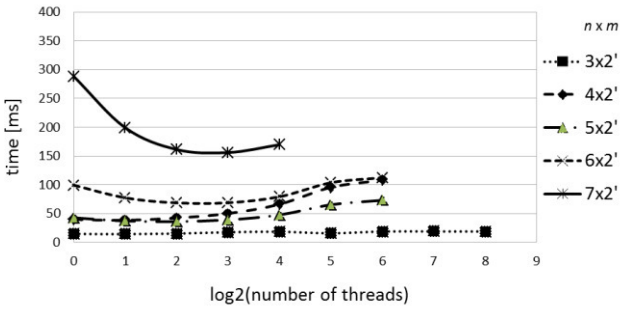

Fig. 2 The average time for processing a box for different numbers of threads in a block and systems of bilinear equations in each variable.

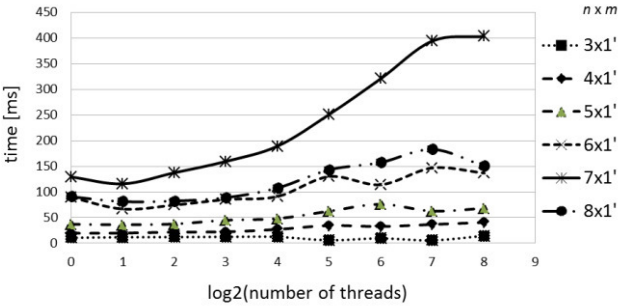

Fig. 3 The average time for processing a box for different numbers of threads in a block and systems of linear equations in each variable with dominant linear coefficients.

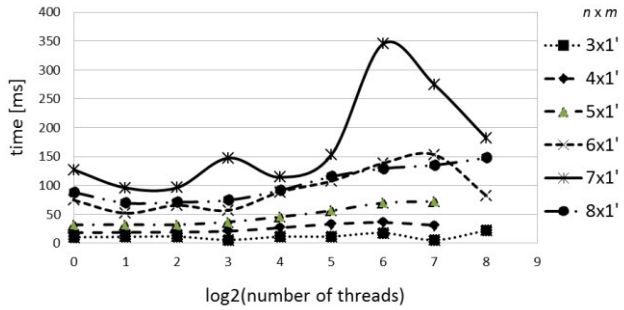

Fig. 4 The average time for processing a box for different numbers of threads in a block and systems of equations with the roots on the boundaries of subdomains.

Fig. 3 presents the results for sets of almost linear systems. The performance for one box is similar to the previous cases, but a smaller increase in time in comparison to Fig. 1 and Fig. 2 is noticeable. The number of subdivisions increased for the certain sets even by a few orders of magnitude. Thus, the roots of systems of 8 equations were also possible to locate.

Special systems of equations were investigated in Fig. 4. The roots have been placed on the boundaries of the subdomains; therefore, the adjacent areas cannot be excluded from root examination, even during later stages of computation. The time needed to process a box is still almost the same, but the number of examined boxes increases rapidly.

The next few experiments lead to an analysis of the relationship between computing performance and the number of blocks.

The initial set of equations was divided by a CPU into the required number of boxes, and then sent to a GPU and divided between the corresponding blocks. The maximum number of blocks we tested was 256.

In Fig. 5 we present the times for different numbers of blocks. The number of threads per block was always 4. Such a choice comes from the previous experiments, where the time increased for a number of threads greater than 4. Fig. 5 shows that a successive increase in the number of blocks up to 32, resulting in a decrease in the time of computation. Then, the greater the number of blocks, the slower the processing of a box. This is due to the large overhead for memory management, when the free memory for data structure allocation is used up. The lack of memory is also the reason for breaking up the calculations for larger systems and a greater number of blocks.

Additionally, when compared to the previously discussed case of a single block, in the case of multiple blocks, the similar relationships between the better performances of solving the systems with a dominant linear part have been observed.

The last three figures (Fig. 6 – Fig. 8) show the influence of the number of threads on the speed of solving similarly complex tasks.

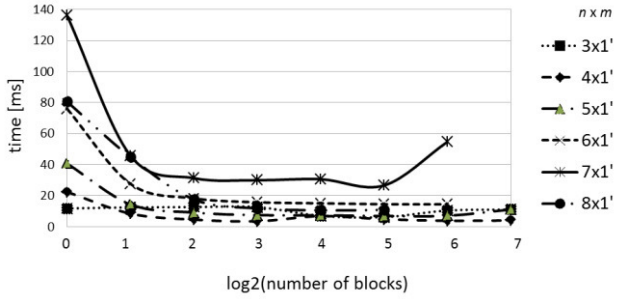

Fig. 5 The average time for processing a box for different numbers of blocks and systems of linear equations in each variable.

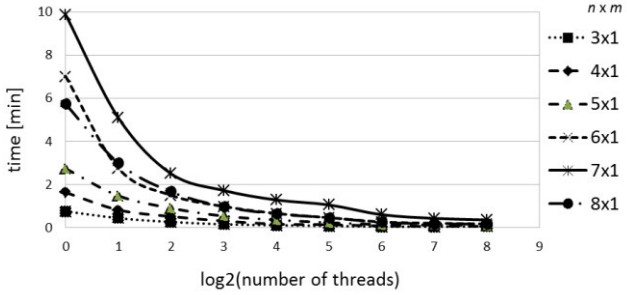

Fig. 6 The run time for processing 1000 boxes for different numbers of threads in a block and systems of linear equations in each variable.

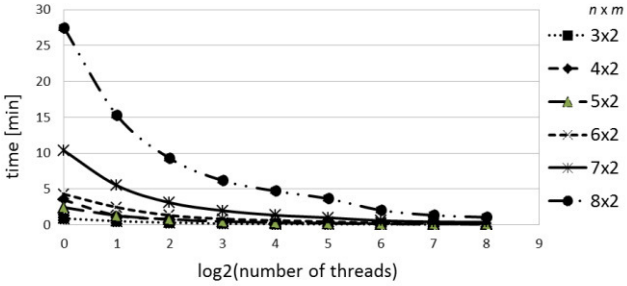

Fig. 7 The run time for processing 1000 boxes for different numbers of threads in a block and systems of bilinear equations in each variable.

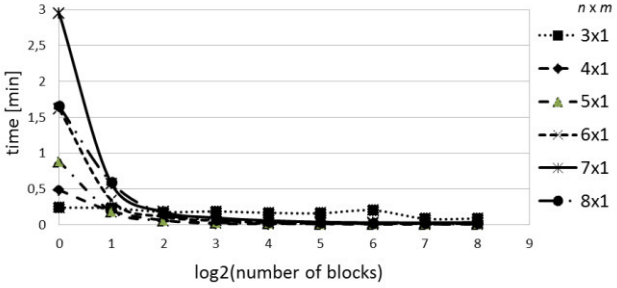

Fig. 8 The run time for processing 1000 boxes for different numbers of blocks of 4threads and systems of linear equations in each variable.

In these experiments, the root-locating procedure was ceased after just 1000 boxes had been processed. The figures present the run time for one block with a maximum number of threads equal to 256 (Fig. 6 and Fig. 7) and up to 256 blocks of only four threads (Fig. 8). Just as was expected for such a small number of processed subdomains, the greater the number of threads, the better the efficiency obtained. A comparison of these results and the previous ones allows us to suppose that the massive number of boxes that needs to be analyzed, and the corresponding memory management, decrease the effectiveness of the solver. In multibock case 4 threads were chosen from experimental results as the best speedup. When more threads are taking boxes form queue and allocating memory simultaneously the time of processing one box increases. It is caused by forced sequentiality due to synchronization of shared resources.

VI. Conclusion

In this paper the possibilities of using a GPU to isolate the roots of systems of polynomial equations have been analyzed. An algorithm based on the general multivariate Bernstein-Bezier representation and the systematic search of a given domain with the use of NVIDIA CUDA technology has been implemented.

The main objective of this research was verification of applicability of GPU in pure bisection process in such a way as not to miss any potential root. We did not aim at a fast computing method. Currently hybrid CPU/GPU solvers or pure CPU solvers based on multicore architecture turn out to be faster. They are usually exploiting Newton's method, which can converge to the solution in another simplex. This implies that some solutions may be missed and some may be found several times. We considered such a behavior disadvantageous Therefore we looked for methods finding all roots in a reasonable time.

The number of variables and equations influences the amount of memory required to store all the necessary data for a given system of equations. The memory usage increases rapidly especially according to the greater degree of a system. Due to the high memory demand and limited memory of GPUs, not all the tests were completed.

A new feature of GPU cards – dynamic memory allocation – proved to be a bottleneck. Therefore, we put a strong emphasis on minimizing such memory allocations. It allowed for a 50% increase in the speed of the solver, when compared to its initial version. The current NVIDIA Computing SDK 4.0 contains some improvements in memory management; therefore this part of the solver might turn out to be even more effective.

Increasing the number of threads in a block results in a decrease in the speed of a single box's processing. This is related to the time needed to wait for synchronization while reading the next box from, and placing two new ones in, the queue. It should be stressed that even in the case of many blocks, they use the common heap and therefore their allocation queries have to be serialized. Hence, to speed up the solver, we also reduced the need for thread synchronization.

Another problem is data transfers between global memory and the device. These can be minimized by the use of on-chip memory that speeds up the calculations. The size of the cache is equal to 128 bytes. This is not sufficient to hold even one of the smallest boxes per thread, even in the case of three trilinear equations, and to take advantage of the cache, it should store data for the whole warp of threads.

The next problem related to memory performance is the storage of results. For larger systems, the queue of boxes that have to be analyzed gets longer. Its length reaches as high as several thousand boxes. The effectiveness of the algorithm may be significantly improved by additional, more expensive tests for eliminating regions without any roots and the huge number of processed data [3].

The studies performed show that although GPU processing is a promising alternative for solving systems of algebraic equations, current software and hardware limitations strongly reduce the fields of their usage. The main problems were connected with slow double precision arithmetic, conflicts in dynamic access to the common memory and the lack of efficient synchronization of different blocks of threads.

Regardless of the technological advances of graphics cards, the use of more effective criteria for box exclusion should be considered as a future direction of research. This should reduce memory consumption and allow the solving of even more complex systems.

REFERENCES

[1] E. C. Sherbrooke and N. M. Patrikalakis, "Computation of the solutions of nonlinear polynomial systems," *Computer Aided Geometric Design*, vol. 10, no. 5, pp. 379–405, Oct. 1993.

[2] T. W. Sederberg and R. J. Meyers, "Loop detection in surface patch intersections," *Computer Aided Geometric Design*, vol. 5, no. 2, pp. 161–171, July 1988.

[3] K. Marciniak, E. Pawelec, and J. Porter-Sobieraj, "Method for finding all solutions of systems of polynomial equations," in *Proc. 7th IEEE Int. Conf. Methods and Models in Automation and Robotics*, vol. 1, pp. 155–158.

[4] G. Elber and M.-S. Kim, "Geometric constraint solver using multivariate rational spline functions," in *Proc. 6th AC M Symposium on Solid modeling and applications*, New York, 2001, pp. 1–10.

[5] C. Loop and J. Blinn, "Real-time GPU rendering of piecewise algebraic surfaces," in *ACM SIGGRAPH 2006 Papers*, New York, 2006, pp. 664–670.

[6] J. Seland and T. Dokken, "Real-time algebraic surface visualization," *Geometrical Modeling, Numerical Simulation, and Optimization*, Springer, Heidelberg, pp. 163–183, 2007.

[7] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design*. San Diego, CA: Academic Press, 1990, pp. 267–281.

[8] *CUDA C Programming Guide*, NVIDIA Corporation, 2012.

[9] *CUDA C Best Practices Guide*, NVIDIA Corporation, 2012.

[10] L. W. Tsai and A. P. Morgan, "Solving the kinematics of the most general six- and five-degree-of-freedom manipulators by continuation methods," ASME J. Mechanisms, Transmissions and Automation in Design, vol. 107, no. 2, 1985, pp. 189–200.

[11] H. Park, G. Elber et al., "A Hybrid Parallel Solver for Systems of Multivariate Polynomials using CPUs and GPUs", SIAM Conference on Geometric and Physical Modeling'11, 2011