Toward the best algorithm for approximate GCD of univariate polynomials

Nagasaka, Kosaku

(Citation)
Journal of Symbolic Computation,105:4-27

(Issue Date)
2021-08

(Resource Type)
journal article

(Version)
Accepted Manuscript

(URL)
https://hdl.handle.net/20.500.14094/90007781

Toward the Best Algorithm for Approximate GCD of Univariate Polynomials[1]

Kosaku Nagasaka

Kobe University, Japan

Abstract

Approximate polynomial GCD (greatest common divisor) of polynomials with a priori errors on their coefficients, is one of interesting problems in Symbolic-Numeric Computations. In fact, there are many known algorithms: QRGCD, UVGCD, STLN based methods, Fastgcd and so on. The fundamental question of this paper is "which is the best?" from the practical point of view, and subsequently "is there any better way?" by any small extension, any effect by pivoting, and any combination of sub-routines along the algorithms. In this paper, we consider a framework that covers those algorithms and their sub-routines, and makes their sub-routines being interchangeable between the algorithms (i.e. disassembling the algorithms and reassembling their parts). By this framework along with/without small new extensions and a newly adapted refinement sub-routine, we have done many performance tests and found the current answer. In summary, 1) UVGCD is the best way to get smaller tolerance, 2) modified Fastgcd is better for GCD that has one or more clusters of zeros with large multiplicity, and 3) modified ExQRGCD is better for GCD that has no cluster of zeros.

Keywords: Symbolic-Numeric Algorithm for Polynomials, Approximate Polynomial GCD, Subresultant Matrix, Sylvester Matrix, QR Factorization, Rank Structured Matrix

1. Introduction

Finding the greatest common divisor (GCD) of the given two polynomials is one of essential computations in computer algebra. In fact, most of other algebraic computations (e.g. reduction of fraction, factorization and so on) rely on the polynomial GCD. In general, finding the GCD can be done by the Euclidean algorithm or other fast algorithms (e.g. halfGCD and its variants (Roy and Sedjelmaci, 2013)). However, in some empirical or practical situations, the coefficients of the given polynomials may have a priori errors, and any conventional algorithm can not compute their expected GCD since their perturbations arisen from the errors make the given polynomials being coprime even if they had some non-trivial common factor in nature. For example, for $f(x) = 0.999x^2 + 1.999x + 1.001 \approx (x + 1)^2$ and $g(x) = 1.001x^2 - 0.999 \approx (x + 1)(x - 1)$, the

[1]This work was supported by JSPS KAKENHI Grant Number 15K00016.

Email address: nagasaka@main.h.kobe-u.ac.jp (Kosaku Nagasaka)

URL: https://wwwmain.h.kobe-u.ac.jp/~nagasaka/ (Kosaku Nagasaka)

Euclidean algorithm can not compute any non-trivial GCD, even if we would like to have their GCD like $d(x) = 1.00063x + 0.999375$.

To overcome this problem, for the couple of decades, several algorithms have been studies and the resulting quasi GCD is called "approximate polynomial GCD". In the early stage of literatures, the first sight of the related problem is appeared in Dunaway (1974) where the Euclidean algorithm with numerical computations is proposed. In this stage, the other algorithms (Schönhage, 1985; Noda and Sasaki, 1991; Sasaki and Noda, 1989) follows the same approach hence their results can be considered as numerical variants of exact GCD. The turning point tending toward the modern definition (e.g. a non-trivial GCD of polynomials in the neighborhoods of the given polynomials over a number field without any numerical fashion) is given by Emiris et al. (1997, 1996), and this definition or similar definitions are widely used after that.

In the present day, there are many algorithms for approximate GCD of univariate polynomials: QRGCD (Corless et al., 2004), UVGCD (Zeng, 2011), STLN based methods (Kaltofen et al., 2006, 2007), Fastgcd (Bini and Boito, 2007), ExQRGCD (Nagasaka and Masui, 2013) and GPGCD (Terui, 2013) that are focused in this paper, and the others (Fazzi et al. (2018), Usevich and Markovsky (2017), Bourne et al. (2017), Christou et al. (2010) and so on). Therefore, one may have the very fundamental question "which is the best?" especially from the practical point of view. One of our contributions is answering to this question. One may think that each research paper has some comparison with known methods hence the question must exist in the papers. This is partially correct. However, in general, those comparisons are based on different programming languages, computer environments, implementors and code optimization levels, hence any well-considered comparison under an uniform treatment is required in addition to those comparisons.

Moreover, it is notable and important that most of these algorithms rely on matrix factorizations of some structured matrices and optimization techniques (for theoretical fundamentals and most of literatures can be found in Boito (2011)). This means that we can consider a framework that covers these algorithms and their sub-routines and makes their sub-routines being interchangeable between the algorithms. By this framework, we can disassemble the algorithms and reassemble their parts as new algorithms. Therefore, we have some answers to the subsequent open question "is there any better way?" along with/without small new extensions (including any effect by pivoting) and a newly adapted sub-routine (based on the approach in Giesbrecht et al. (2017)). This is also one of contributions of this paper and our result is given in Sections 4 and 5. In summary, 1) UVGCD is the best way to get smaller tolerance (a kind of radius of neighborhoods), 2) modified Fastgcd is better for GCD that has one or more clusters of zeros with large multiplicity, and 3) modified ExQRGCD is better for GCD that has no cluster of zeros.

After the preliminary subsection including some definitions and notations below, we start with the several brief reviews of known algorithms, and introduce some small new extensions and a newly adapted sub-routine in Section 2. Our framework is given in Section 3, and our result of some hard performance test is given in Section 4. Finally, in Section 5, we give our recommendation among the algorithms.

1.1. Definitions and Notations

We assume that polynomials $f(x) = \sum_{i=0}^{m} f_i x^i \in K[x]$ and $g(x) = \sum_{i=0}^{n} g_i x^i \in K[x]$ are given with the assumption $m \geq n$. The domain $K[x]$ that $f(x)$ and $g(x)$ belong to is $\mathbb{R}[x]$ or $\mathbb{C}[x]$ depending on each problem setting. Each of these given polynomials has the unit Euclidean norm (2-norm) (i.e. $\|f(x)\|_2 = \|g(x)\|_2 = 1$). We scale the polynomials if they do not have the unit

norm at the input. Although there are several variants of definitions, we use the following two definitions of approximate GCD.

Definition 1 (approximate polynomial GCD with the given tolerance)**.**
For the tolerance $\varepsilon \in \mathbb{R}_{\geq 0}$, we compute the polynomial $d(x) \in K[x]$ called "approximate polynomial GCD" of tolerance ε, which has the maximum degree and satisfies

$$f(x) + \Delta_f(x) = f_1(x)d(x),\ g(x) + \Delta_g(x) = g_1(x)d(x)$$

for some polynomials $\Delta_f(x), \Delta_g(x), f_1(x), g_1(x) \in K[x]$ such that

$$\deg(\Delta_f) \leq \deg(f),\ \deg(\Delta_g) \leq \deg(g),\ \|\Delta_f\|_2 < \varepsilon \|f\|_2,\ \|\Delta_g\|_2 < \varepsilon \|g\|_2\ .$$

Definition 2 (approximate polynomial GCD with the given degree)**.**
For the degree $k \in \mathbb{N}$, we compute the polynomial $d(x) \in K[x]$ called "approximate polynomial GCD" of degree k, which minimizes $\|\Delta_f\|_2^2 + \|\Delta_g\|_2^2$ (called perturbation) and satisfies

$$f(x) + \Delta_f(x) = f_1(x)d(x),\ g(x) + \Delta_g(x) = g_1(x)d(x),\ \deg(d) = k$$

for some polynomials $\Delta_f(x), \Delta_g(x), f_1(x), g_1(x) \in K[x]$ such that

$$\deg(\Delta_f) \leq \deg(f),\ \deg(\Delta_g) \leq \deg(g).$$

Remark 1. *Most of algorithms for Definition 1 do not guarantee the maximum degree but try to find a factor of nearly maximum degree by floating-point calculations (see Emiris et al. (1997) for a maximum degree condition). Also for Definition 2, most of algorithms do not guarantee the minimum perturbation but try to find a factor with a nearly minimum perturbation by floating-point calculations, and there are similar but different objective functions (e.g.* $\max(\| \Delta_f \|_2, \| \Delta_g \|_2)$*). Moreover, there are algorithms for polynomials in $\mathbb{Z}[x]$ (hence by exact calculations), proposed by von zur Gathen et al. (2010) and Nagasaka (2011) for example.*

Remark 2. *In this paper, we seek the best algorithm among the algorithms we are interested in. The criteria of our evaluation is very simple. A larger detecting degree is better (and a smaller resulting tolerance is better if the same degree) for approximate polynomial GCD with the given tolerance (Definition 1), and a smaller resulting perturbation is better for approximate polynomial GCD with the given degree (Definition 2). However, too much elapsed time may not be a good behavior from the practical point of view. Therefore, we consider their elapsed time also.*

For vectors and matrices, $\|\cdot\|_2$ and $\|\cdot\|_F$ denote the Euclidean norm (2-norm) and the Frobenius norm, respectively. A^T, A^H and A^{-1} are the transpose, the conjugate transpose, and the inverse of matrix A, respectively. The coefficient vector (in the descending term order) of $p(x)$ of degree k is denoted by $\vec{p} \in K^{k+1}$. The Sylvester matrix of $f(x)$ and $g(x)$ is denoted by $\mathrm{Syl}(f, g)$ defined as

$$\mathrm{Syl}(f,g) = \begin{pmatrix} f_m & f_{m-1} & \cdots & f_1 & f_0 & & & \\ & f_m & f_{m-1} & \cdots & f_1 & f_0 & & \\ & & \ddots & \ddots & \cdots & \ddots & \ddots & \\ & & & f_m & f_{m-1} & \cdots & f_1 & f_0 \\ g_n & g_{n-1} & \cdots & g_1 & g_0 & & & \\ & g_n & g_{n-1} & \cdots & g_1 & g_0 & & \\ & & \ddots & \ddots & \cdots & \ddots & \ddots & \\ & & & g_n & g_{n-1} & \cdots & g_1 & g_0 \end{pmatrix} \in K^{(m+n)\times(m+n)}.$$

The subresultant matrix of order k of $f(x)$ and $g(x)$ ($n-k$ columns for $\vec{f}$ and $m-k$ columns for $\vec{g}$) is denoted by $\mathrm{S}_k(f,g)$ defined as

$$\mathrm{S}_k(f,g) = \begin{pmatrix} f_m & & & & g_n & & & \\ f_{m-1} & f_m & & & g_{n-1} & g_n & & \\ \vdots & f_{m-1} & \ddots & & \vdots & g_{n-1} & \ddots & \\ f_1 & \vdots & \ddots & & g_1 & \vdots & \ddots & \\ f_0 & f_1 & \vdots & & g_0 & g_1 & \vdots & \\ & f_0 & \ddots & \vdots & & g_0 & \ddots & \vdots \\ & & \ddots & f_1 & & & \ddots & g_1 \\ & & & f_0 & & & & g_0 \end{pmatrix} \in K^{(m+n-k)\times(m+n-2k)}.$$

Throughout this paper, the capital letters D, L, U, P, Q and R denote diagonal, lower triangular, upper triangular, permutation, unitary (orthogonal) and upper triangular matrices, respectively.

2. Known Algorithms and Our Extensions

In this paper, we are interested in the algorithms: QRGCD, ExQRGCD, UVGCD, Fastgcd, STLN based methods and GPGCD, hence we give some brief reviews for these algorithms. Especially for the first 4 algorithms, we show some their backgrounds and give our extensions (e.g. pivoting). Moreover, we adapt a new refinement method based on the method for structured polynomial matrices.

2.1. QRGCD and ExQRGCD

QRGCD is the most famous algorithm for approximate polynomial GCD in Definition 1 since it was studied in the early stage and its implementation has been distributed with Maple. ExQRGCD is a complementary algorithm of QRGCD with more theoretical considerations known after QRGCD. Algorithms 1 and 2 are their brief versions, and they are based on the following well known lemma (see e.g. Laidacker (1969)).

Lemma 1. *Let R be the upper triangular matrix of the QR factorization without pivoting of $Syl(f,g)$. The last non-zero row vector of R gives the coefficients of the polynomial GCD of $f(x)$ and $g(x)$.*

2.1.1. Pivoting Capability for QRGCD and ExQRGCD

In general, the Householder QR factorization is norm-wise backward stable (i.e. $\|A - \hat{Q}\hat{R}\|$ is very small where $\hat{Q}\hat{R}$ is the QR factorization of A by numerical computation). However, complete pivoting (column pivoting and row pivoting) may improve the row-wise backward error in some cases (Cox and Higham, 1998; Powell and Reid, 1969). Therefore, there may be a chance to improve the algorithms by the QR factorization with complete pivoting.

The proof of Lemma 1 is usually based on the fact that the linear space spanned by the rows of the Sylvester matrix and the image of the Sylvester mapping are isomorphic. In general, the QR factorization with complete pivoting breaks this isomorphism hence Lemma 1 requires it without pivoting. However, the maximum degree of GCD is bounded and less than the number

Algorithm 1: QRGCD (brief description)

Input: $f(x), g(x) \in K[x]$, and tolerance $\varepsilon \in \mathbb{R}_{\geq 0}$.
Output: $d(x), f_1(x), g_1(x) \in K[x]$ (as in Definition 1)

$f_1(x) := f(x),\ g_1(x) := g(x),\ i := 1;$
while $i \leq 2$ **do**
  compute the QR factorization of Sylvester matrix: $QR = \mathrm{Syl}(f_1, g_1)$, and let $R^{(k)}$ be the bottom-rightmost $(k+1) \times (k+1)$ submatrix of R such that $\|R^{(k)}\|_2 > \varepsilon$ and $\|R^{(k-1)}\|_2 < \varepsilon$;
  switch do
    case $\|R^{(0)}\|_2 > \varepsilon$ **do** `// approximately coprime`
      $d_i(x) := 1;$
    case $\|R^{(k-1)}\|_2 < 10\varepsilon\, \|R^{(k)}\|_2$ **do** `// big gap found`
      $d_i(x) :=$ the polynomial formed by the last k-th row vector of R;
    case $\exists k_1 (biggest),\ \|R^{(k_1-1)}\|_2 < 10\varepsilon\, \|R^{(k_1)}\|_2$ **do** `// gap found`
      $d_i(x) :=$ the polynomial formed by the last k_1-th row vector of R;
    otherwise do `// difficult case`
      some special treatment (see the original article);
    end
  end
  let $f_1(x), g_1(x)$ be the reversal of cofactors of $f_1(x)$ and $g_1(x)$ by $d_i(x)$, and $i := i + 1$;
end
let $d_2(x)$ be the reversal of $d_2(x)$, and $d(x) := d_1(x)d_2(x)$;
return $d(x), f_1(x), g_1(x)$.

of columns of the Sylvester matrix hence we can slightly update the lemma with almost complete pivoting (row pre-sorting and column pivoting) in part as follows.

Let $\mathrm{Syl}_\alpha(f,g)$ and $\mathrm{Syl}_\beta(f,g)$ be the first α columns and the last β columns of $\mathrm{Syl}(f,g)$, respectively, as $\mathrm{Syl}(f,g) = \left(\mathrm{Syl}_\alpha(f,g)\ \mathrm{Syl}_\beta(f,g)\right)$ where $\alpha = m + n - \min(m,n)$ and $\beta = \min(m,n)$. We compute the QR decomposition with complete pivoting of $\mathrm{Syl}_\alpha(f,g)$ as

$$\mathrm{Syl}_\alpha(f,g) = P_r Q_\alpha \begin{pmatrix} R_\alpha \\ O_{\beta,\alpha} \end{pmatrix} P_c$$

where $P_r \in \mathbb{Z}^{(m+n)\times(m+n)}$ and $P_c \in \mathbb{Z}^{\alpha\times\alpha}$ are permutation matrices, and $O_{\beta,\alpha} \in \mathbb{C}^{\beta\times\alpha}$ is the zero matrix. Let $\mathrm{Syl}_{\beta\uparrow}(f,g)$ and $\mathrm{Syl}_{\beta\downarrow}(f,g)$ be the first α rows and the last β rows of $Q_\alpha^H P_r^{-1} \mathrm{Syl}_\beta(f,g)$, respectively. We have

$$\mathrm{Syl}(f,g) = P_r Q_\alpha \begin{pmatrix} R_\alpha & \mathrm{Syl}_{\beta\uparrow}(f,g) \\ O_{\beta,\alpha} & \mathrm{Syl}_{\beta\downarrow}(f,g) \end{pmatrix} \begin{pmatrix} P_c & O_{\alpha,\beta} \\ O_{\beta,\alpha} & I_\beta \end{pmatrix}$$

where $I_\beta \in \mathbb{C}^{\beta\times\beta}$ denotes the identity matrix.

Let $\vec{v}$ be any vector in the linear space spanned by the rows of $\mathrm{Syl}(f,g)$ whose first α elements are zeros, and $\vec{w} \in \mathbb{C}^\beta$ be the last β elements of $\vec{v}$. By the construction process of $\mathrm{Syl}_{\beta\downarrow}(f,g)$, $\vec{w}$ must be in the linear space spanned by the rows of $\mathrm{Syl}_{\beta\downarrow}(f,g)$. Therefore Lemma 1 is extended to the following lemma.

Algorithm 2: ExQRGCD (brief description)

Input: $f(x), g(x) \in K[x]$, and tolerance $\varepsilon \in \mathbb{R}_{\geq 0}$.
Output: $d(x), f_1(x), g_1(x) \in K[x]$ (as in Definition 1)

$f_1(x) := f(x),\ g_1(x) := g(x),\ d_0(x) := 0,\ d(x) := 1,\ i := 1;$
repeat
  compute the QR factorization of Sylvester matrix:$QR = \mathrm{Syl}(f_1, g_1)$, and let $R^{(k)}$ be the bottom-rightmost $(k+1)\times(k+1)$ submatrix of R and $\vec{r}_k$ be the top row vector of $R^{(k)}$;
  switch do
    case $\|R^{(0)}\|_F > \varepsilon\sqrt{m+n}$ **do** `// approximately coprime`
      $d_i(x) := 1;$
    case $\|R^{(0)}\|_F \leq \varepsilon\sqrt{m+n}$ **do** `// trial divisions`
      $d_i(x :=$ the polynomial formed by $\vec{r}_k$ that minimizes $\|R^{(k-1)}\|_F \,/\, \|\vec{r}_k\|_2;$
      $d(x) := \prod_{j=1}^{i} d_j(x);$
      if $\nexists f_2(x), g_2(x)$, $\|f(x) - d(x)f_2(x)\|_2 < \varepsilon$ *and* $\|g(x) - d(x)g_2(x)\|_2 < \varepsilon$ **then**
        some special treatment (see the original article);
      end
  end
  let $f_1(x), g_1(x)$ be the cofactors of $f(x)$ and $g(x)$ by $d(x)$, and $i := i + 1;$
  let $f(x), g(x), f_1(x), g_1(x), d_1(x), \ldots, d_{i-1}(x), d(x)$ be their reversal polynomials;
until $d_{i-2}(x) \neq 1$ *or* $d_{i-1}(x) \neq 1;$
if *i is even* **then** let $d(x), f_1(x), g_1(x)$ be their reversal polynomials;
return $d(x), f_1(x), g_1(x)$.

Lemma 2. *Let R be the upper triangular matrix of the QR factorization without pivoting of* $Syl_{\beta\downarrow}(f, g)$. *The last non-zero row vector of R gives the coefficients of the polynomial GCD of* $f(x)$ *and* $g(x)$.

2.2. UVGCD

UVGCD is the algorithm for approximate GCD in Definition 1 with $\sqrt{\|\Delta_f\|_2^2 + \|\Delta_g\|_2^2} < \varepsilon$ instead of $\|\Delta_f\|_2 < \varepsilon \|f\|_2$ and $\|\Delta_g\|_2 < \varepsilon \|g\|_2$, and computes a pair of candidate cofactors from the singular vector of $\mathrm{S}_{k-1}(f, g)$ where k is the largest integer such that the smallest singular value of $\mathrm{S}_{k-1}(f, g)$ is less than the certain value. In general, k is not known in advance hence we have to compute the smallest singular value of $\mathrm{S}_{k-1}(f, g)$ for $k = n, n-1, \ldots, 1$ sequentially though this computation is time-consuming.

To overcome this problem, UVGCD computes the QR factorization of $\mathrm{S}_{k-1}(f, g)P_k$ from $\mathrm{S}_k(f, g)P_k$, instead of $\mathrm{S}_{k-1}(f, g)$ directly as follows. Let P_{k-1} $(n \geq k \geq 1)$ be the column permutation matrix satisfying $P_{n-1} = I_{m-n+2}$ and

$$\mathrm{S}_{k-1}(f, g)P_{k-1} = \begin{pmatrix} & & f_m & \\ & & f_{m-1} & g_n \\ & \mathrm{S}_k(f, g)P_k & \vdots & g_{n-1} \\ & & \vdots & \vdots \\ & & f_1 & g_1 \\ 0 & \cdots\ 0 & f_0 & g_0 \end{pmatrix}.$$

Algorithm 3: UVGCD (brief description)

Input: $f(x), g(x) \in K[x]$, and tolerance $\varepsilon \in \mathbb{R}_{\geq 0}$.
Output: $d(x), f_1(x), g_1(x) \in K[x]$ (as in Definition 1)

let $P_{n-1} := I$, and compute the QR factorization: $Q_{n-1}R_{n-1} = \mathrm{S}_{n-1}(f, g)P_{n-1}$;
for $j = n, n-1, \ldots, 1$ **do**
  compute the smallest singular value σ_{-1} and its vector $\vec{y}$ as
  $R_j^H \vec{z} = \vec{y}_i,\ R_j \vec{z'} = \vec{z},\ \vec{y}_{i+1} = \vec{z'} / \|\vec{z'}\|$;
  if $\sigma_{-1} < \varepsilon\sqrt{m-j+1}$ **then**
    construct cofactors $f_1(x)$ and $g_1(x)$ from $\vec{y}$;
    compute initial $d(x)$ by the least square;
    refine $d(x), f_1(x), g_1(x)$ by the Gauss-Newton method;
    if *the perturbation is less than* ε **then return** $d(x), f_1(x), g_1(x)$;
  end
  update R_{j-1} as $\mathrm{S}_{j-1}(f, g)P_{j-1} = Q_{j-1}R_{j-1}$;
end
return $1, f(x), g(x)$.

By this sequential construction, computing the QR factorization of $\mathrm{S}_{k-1}(f, g)P_{k-1}$ requires only two orthogonal transformations for each k. Computing the smallest singular value $\sigma_{-1}(\mathrm{S}_{k-1}(f, g))$ and its singular vector $\vec{z}$ can be done by the following iterations for a randomly generated initial vector $\vec{z}_0$ where QR is the QR factorization of $\mathrm{S}_{k-1}(f, g)$.

1. Solve $R^H \vec{y}_i = \vec{z}_{i-1}$ by forward substitution.
2. Solve $R\vec{z}_i = \vec{y}_i$ by backward substitution.
3. Normalize $\vec{z}_i$.

2.2.1. Pivoting Capability for UVGCD

The stage of finding the candidate degree of approximate GCD in UVGCD uses its special permutation matrices only, hence the other pivoting strategies easily can be integrated as follows. At first, we compute the QR factorization with almost complete pivoting (row pre-sorting and column pivoting) of $\mathrm{S}_{n-1}(f, g)$ and let it be $\mathrm{S}_{n-1}(f, g) = \bar{P}'_{n-1}Q_{n-1}R_{n-1}\bar{P}_{n-1}$. At each update step of $\mathrm{S}_k(f, g)$ to $\mathrm{S}_{k-1}(f, g)$, we apply the row pre-sorting $\bar{P}'_{n-1}$ to the newly added two columns ($\vec{f}$ and $\vec{g}$) and compute the QR factorization with column pivoting w.r.t. these columns though the number of applicable selections of columns is only 2. Finally, we take into the account of those permutations when we construct the candidate cofactors $u(x)$ and $v(x)$ from the computed singular vector.

Moreover, at the refinement stage, the Gauss-Newton method of UVGCD uses the linear least squares solver to find each correction vector hence we can use both of the QR factorization of column pivoting and the complete orthogonal factorization on our demand.

2.3. Fastgcd

Fastgcd is the algorithm for approximate GCD in Definition 1, and its approach is very similar to that of UVGCD. It computes a candidate degree of approximate GCD, and refines a candidate approximate GCD and its cofactors. In principal, such an approach requires some matrix

factorization (QR, LU and so on) of the computational complexity $O(m^3)$ to detect the candidate degree, compute the candidate factors and refine the factors. In Fastgcd, to overcome this problem (sub-routines of high complexity are required), they replace the sub-routines with a structured matrix factorization of the computational complexity $O(m^2)$ as follows.

In general, we say that the matrix $S \in \mathbb{C}^{m\times n}$ has a displacement structure (of displacement rank r)[2] if there exist $V \in \mathbb{C}^{m\times r}$ and $H \in \mathbb{C}^{r\times n}$ such that $O_L S - S O_R = VH$ for some $r \in \mathbb{Z}_{>0}$. $O_L \in \mathbb{C}^{m\times m}$ and $O_R \in \mathbb{C}^{n\times n}$ are called "displacement operators", and V and H are called "generators". Especially, the matrix S is called "Toeplitz-like" if it has the displacement structure with operators $O_L = T_m^1$ and $O_R = T_n^\delta$ for some $\delta \in \mathbb{C}$ where

$$T_\ell^\delta = \begin{pmatrix} & & \delta \\ & I_{\ell-1} & \end{pmatrix} \in \mathbb{C}^{\ell\times\ell}.$$

In fact, the Sylvester matrix, the subresultant matrix and the Jacobian matrix used in Fastgcd are Toeplitz-like matrices. The displacement ranks of Sylvester, subresultant and Jacobian matrices are 2, 2 and 5, respectively.

There is another displacement structure called "Cauchy-like" when O_L and O_R are diagonal matrices, and any Toeplitz-like matrix S with operators T_m^1 and T_n^δ can be converted into Cauchy-like matrix $C = F_m S D^{-1} F_n^H$ satisfying $\Omega C - C\Lambda = (F_m V)(HD^{-1}F_n^H)$ where $F_\ell = (f_{ij})$, $f_{ij} = \sqrt{1/\ell}\,(e^{\frac{2\pi\mathrm{i}}{\ell}(i-1)(j-1)})$, $D = \mathrm{diag}(d_{ii})$, $d_{ii} = \delta^{\frac{i-1}{n}}$, $\Omega = \mathrm{diag}(\omega_{ii})$, $\omega_{ii} = e^{\frac{2\pi\mathrm{i}}{m}(i-1)}$, $\Lambda = \mathrm{diag}(\lambda_{ii})$, $\lambda_{ii} = \delta^{\frac{1}{n}} e^{\frac{2\pi\mathrm{i}}{n}(i-1)}$. For Cauchy-like matrices, there is a fast LU factoring method (Gu, 1998) called "modified GKO method" of complexity $O(m^2)$. Note that the construction of C is not required to compute its LU factorization since only generators have to be converted by FFT, whose complexity $O(rm \log m)$. In the below, we assume $\delta = -1$ since Fastgcd does.

Remark 3. *The LU or QR factorizations of Cauchy-like matrix converted from a Toeplitz-like matrix can not be reversely converted to the factorizations of the original Toeplitz-like matrix since the upper triangular matrix must be converted by FFT hence its triangular property will be lost. However, finding a candidate degree of approximate GCD, computing candidate factors and solving linear equations can be done without computing any actual factor of the LU and QR factorizations of the original matrix.*

2.3.1. Real Polynomial Capability for Fastgcd

In Bini and Boito (2007), it is noted that the algorithm generally outputs an approximate GCD with complex coefficients, even if $f(x)$ and $g(x)$ are real polynomials, and in Boito (2011), it is noted that it would not be possible to compute an approximate GCD with real coefficients, even if it was required. The reason is that the algorithm computes the candidate degree, candidate factors and refined factors by converting Toeplitz-like matrices with real elements into Cauchy-like matrices with complex elements hence the resulting properties must be over $\mathbb{C}$. However, we can compute an approximate GCD over $\mathbb{R}$ for polynomials $f(x), g(x) \in \mathbb{R}[x]$ as follows.

At first, before the while loop, the algorithm determines the candidate degree of approximate GCD by computing the approximate rank of $\mathrm{Syl}(f, g)$ by computing its LU factorization with almost complete pivoting by the modified GKO method. The magnitudes of singular values

[2]Most of literatures including the original paper of Fastgcd introduce the displacement structure for a square matrix. However, in this paper, we give it with general rectangular matrix since subresultant matrix is not square in general.

Algorithm 4: Fastgcd (brief description)

Input: $f(x), g(x) \in K[x]$, and tolerance $\varepsilon \in \mathbb{R}_{\geq 0}$.
Output: $d(x), f_1(x), g_1(x) \in K[x]$ (as in Definition 1)

convert $\mathrm{Syl}(f, g)$ into the Cauchy-like matrix C and compute the LU factorization with almost complete pivoting: $C = P_r L U P_c$ by the modified GKO method;
determine a candidate degree k of approximate GCD;
while $k \geq 0$ **do**
  convert $\mathrm{S}_{k-1}(f, g)$ into the Cauchy-like matrix C and compute the LU factorization with almost complete pivoting: $C = P_r L U P_c$ by the modified GKO method;
  construct cofactors $f_k(x), g_k(x)$ from $\mathrm{Ker}(U)$;
  compute initial $d_k(x)$ by the DFT division (FFT), and refine $d_k(x), f_k(x), g_k(x)$ by the Gauss-Newton method with inversion of Jacobian by the modified GKO method;
  if $d_k(x)$ *satisfies the given tolerance condition* **then**
    increment the candidate degree as $k := k + 1$;
  else
    if *any lower non-trivial approximate GCD has not been found* **then**
      decrement the candidate degree as $k := k - 1$;
    else
      $d(x) :=$ the latest non-trivial approximate GCD among $d_1(x), \ldots, d_k(x)$, and let $f_1(x), g_1(x)$ be the corresponding cofactors;
      return $d(x), f_1(x), g_1(x)$.
    end
  end
end
return $1, f(x), g(x)$.

of real matrix are not changed even if we compute them over $\mathbb{C}$. Therefore, we can treat the computed candidate degree as a candidate degree over $\mathbb{R}$.

Within the while loop, the algorithm determines the candidate cofactors from $\mathrm{Ker}(U)$ over $\mathbb{C}$, where U is the upper triangular matrix of the LU factorization of $\mathrm{S}_{k-1}(f, g)$ over $\mathbb{C}$. This means that Fastgcd finds $\vec{z} \in \mathbb{C}^{m+n-2(k-1)}$ satisfying $U\vec{z} = \vec{0}$ hence we have $\mathrm{S}_{k-1}(f, g)\vec{z} = \vec{0}$. By $\mathcal{R}e(\cdot)$, we denote the element-wise mapping from $\mathbb{C}$ to $\mathbb{R}$ such that $\mathcal{R}e(a + b\mathrm{i}) = a$ for any $a, b \in \mathbb{R}$. Since $\mathrm{S}_{k-1}(f, g)$ and $\vec{0}$ are over $\mathbb{R}$, $\mathcal{R}e(\vec{z})$ must be in the null space of $\mathrm{S}_{k-1}(f, g)$. Then, Fastgcd refines the candidate factors. The assumption $f(x), g(x), d(x), f_1(x), g_1(x) \in \mathbb{R}[x]$ implies that the objective function to be minimized by the Gauss-Newton method is over $\mathbb{R}$ and its Jacobian is also over $\mathbb{R}$. Therefore, for each iteration from $\vec{z}_i$ to $\vec{z}_{i+1}$, even if we compute each correction vector $\vec{z'}_{i+1}$ by the modified GKO method over $\mathbb{C}$, $\mathcal{R}e(\vec{z'}_{i+1})$ must be a correction vector over $\mathbb{R}$. As a consequence, we have the following lemma.

Lemma 3. *Fastgcd can find an approximate GCD over $\mathbb{R}$ for the given real polynomials $f(x)$ and $g(x)$ if we take the real projections of computed candidate factors and correction vectors within the while loop, respectively.*

2.4. STLN based methods

Though there are several algorithms based on STLN (structured total least norm), we give brief reviews of "STLN-GCD1" and "STLN-GCD2" only where we call the algorithm in Kaltofen

	polynomials over $\mathbb{R}$	polynomials over $\mathbb{C}$
STLN-GCD1	$\mathbb{R}^{(2m+2n-k+2)\times(2m+2n-2k+1)}$	$\mathbb{C}^{(2m+2n-k+2)\times(2m+2n-2k+1)}$
STLN-GCD2	$\mathbb{R}^{(2m+2n-k+2)\times(2m+2n-2k+1)}$	$\mathbb{R}^{(4m+4n-2k+4)\times(4m+4n-4k+2)}$

Table 1: Sizes of coefficient matrices in STLN-GCD1 and STLN-GCD2

et al. (2007) "STLN-GCD1" since this approach is already appeared earlier (Zhi and Yang, 2004), and call the other one in Kaltofen et al. (2006) "STLN-GCD2". Note that they are the algorithms for approximate GCD in Definition 2.

The most remarkable point of both the algorithms against other algorithms is that they do not compute any approximate GCD directly. They just find a nearby subresultant matrix that is rank deficient and preserving its displacement structure. At first they divide the subresultant matrix $\mathrm{S}_{k-1}(f,g)$ into two part a matrix $A_{k-1}(f,g)$ and a vector $\vec{b}_{k-1}(f,g)$ where $\vec{b}_{k-1}(f,g)$ is a column vector of $\mathrm{S}_{k-1}(f,g)$ and $A_{k-1}(f,g)$ is the remainder columns of $\mathrm{S}_{k-1}(f,g)$. Note that STLN-GCD1 always chooses the first column vector as $\vec{b}_{k-1}(f,g)$ while it is depending on the given polynomials in STLN-GCD2. Then, they solve the following overdetermined system w.r.t. the unknown vector $\vec{x}$ using STLN.

$$A_{k-1}(f,g)\vec{x} \approx \vec{b}_{k-1}(f,g).$$

Its solution can be represented as $A_{k-1}(f+\Delta_f, g+\Delta_g)\vec{x} = \vec{b}_{k-1}(f+\Delta_f, g+\Delta_g)$ hence the resulting $\mathrm{S}_{k-1}(f+\Delta_f, g+\Delta_g)$ is rank deficient (i.e. non-trivial approximate GCD exists). Once the rank deficient subresultant matrix (hence the perturbation polynomials) is computed, computing its approximate GCD is not difficult and they leave the choice of the method to find $d(x)$ to the readers' discretion.

The above STLN problem is to be solved by the least squares with penalty in the algorithms, and we show two different parts between STLN-GCD1 and STLN-GCD2 though we do not describe their details here. The first different part is the method to find the initial value. STLN-GCD1 computes the un-structured solution of the above overdetermined system and uses the solution as the initial value. STLN-GCD2 computes the initial value by another method based on Lagrangian multipliers. The second different part is the way to handle the problem over $\mathbb{C}$ (complex numbers). STLN-GCD1 solves the problem directly over $\mathbb{C}$, but STLN-GCD2 solves the problem over $\mathbb{R}$. Therefore, the size of coefficient matrix of the least squares problem with penalty are different as in Table 1.

2.5. GPGCD

GPGCD is the algorithm for approximate GCD in Definition 2. The name of GPGCD comes from that it uses the gradient-projection method (or the modified Newton method as the generalization of gradient-projection method) to minimize $\|\Delta_f\|_2^2 + \|\Delta_g\|_2^2$ subject to

$$\mathrm{S}_{k-1}(f+\Delta_f, g+\Delta_g)\begin{pmatrix} \vec{u} \\ \vec{v} \end{pmatrix} = \vec{0}$$

and $\|u\|_2^2 + \|v\|_2^2 = 1$ w.r.t. the coefficients of $\Delta_f(x)$, $\Delta_g(x)$, $u(x)$ and $v(x)$. This can be considered as the most straightforward way to convert the approximate GCD problem into some optimization problem. Moreover, this method solves the problem over $\mathbb{R}$ even if the given polynomials are over $\mathbb{C}$ (the size of matrix becomes larger).

Algorithm 5: Framework of approximate GCD

```
begin – Loop Structure –
    // searching for the degree of approximate GCD, if not given.
    begin – Candidate Degree –
        // finding a candidate degree of approximate GCD.
    end
    begin – Candidate Factors –
        // computing candidate approximate GCD and cofactors.
    end
    begin – Refinement Step –
        // getting better approximate GCD and cofactors.
    end
end
```

2.6. GHLGCD (new refinement method)

In Giesbrecht et al. (2017), a new method to compute the nearest rank-deficient matrix polynomial is proposed. Their algorithm embeds the given matrix polynomial into a structured matrix over $\mathbb{R}$, and tries to find the nearest rank-deficient structured matrix. Especially their embedded structured matrix is a block Toeplitz matrix. Since the Sylvester and subresultant matrices are also block Toeplitz we can adapt it to our purpose that we minimize

$$\|\mathrm{S}_{k-1}(\Delta_f, \Delta_g)\|_F^2 + \|\vec{w}\|_F^2 - 1$$

subject to $\mathrm{S}_{k-1}(f + \Delta_f, g + \Delta_g)\vec{w} = \vec{0}$ and $\vec{w}^T\vec{w} = 1$ w.r.t. $\vec{x} = \left(\vec{\Delta_f}^T,\ \vec{\Delta_g}^T,\ \vec{w}^T\right)^T$, by the Newton method. The actual Newton iteration is as follows.

$$\begin{pmatrix} \vec{x}_{i+1} \\ \vec{\lambda}_{i+1} \end{pmatrix} = \begin{pmatrix} \vec{x}_i + \vec{\Delta}_{xi} \\ \vec{\lambda}_i + \vec{\Delta}_{\lambda i} \end{pmatrix},\quad \nabla^2 L_i \begin{pmatrix} \vec{\Delta}_{xi} \\ \vec{\Delta}_{\lambda i} \end{pmatrix} = -\nabla L_i$$

where $L = \|\mathrm{S}_{k-1}(\Delta_f, \Delta_g)\|_F^2 + \|\vec{w}\|_F^2 - 1 + \left((\mathrm{S}_{k-1}(f + \Delta_f, g + \Delta_g)\vec{w})^T,\ \vec{w}^T\vec{w} - 1\right)^T \vec{\lambda}$ and the Lagrangian multipliers $\vec{\lambda} = (\lambda_0\ \ldots\ \lambda_{m+n-k+1})^T$. This can be considered as the algorithm for approximate GCD in Definition 2 with a little bit different objective function. Once the rank deficient subresultant matrix is computed, as in STLN-GCD1 and STLN-GCD2, computing its approximate GCD is not difficult. We call this algorithm "GHLGCD".

2.7. Theoretical Overview of Performance

All the algorithms above basically follow the steps as in Algorithm 5. We give some brief theoretical overviews in the time complexities and size of perturbations (residues) for the loop structure and the refinement step.

2.7.1. Performance of Loop Structure

The loop structure is only for approximate GCD in Definition 1 hence we focus on QRGCD, ExQRGCD, UVGCD and Fastgcd. At first, their approaches are characterized as the two stage factor peeling, the multi stage factor peeling, the unidirectional degree search, and bidirectional degree search, respectively.

two stage factor peeling At first, this tries to detect an approximate common divisor whose zeros lie inside the unit circle in the complex plane, and then outside the unit circle. The number of searching stages is only two hence this is very fast. However, its detecting accuracy is very rough hence its resulting perturbation easily becomes large. Moreover, each stage requires the QR decomposition of the Sylvester matrix hence the time complexity of this part is $O(m^3)$.

multi stage factor peeling This follows the two stage peeling but with a more accurate method, and repeats this peeling sequentially. Therefore, this approach is slower than the two stage peeling. However, its detecting accuracy is more precise. Moreover, each stage requires the QR decomposition of the Sylvester matrix hence the time complexity of this part is $O(km^3)$ where k denotes the degree of approximate GCD.

unidirectional degree search Since the degree of approximate GCD must be bounded by n, searching for the degree k from n to 1 by the subresultant matrix of order k may detect the correct one. While this takes $O(nm^3)$ operations in general, UVGCD reduces this time complexity to $O(m^3)$. The accuracy of detected degree and overall time complexity depends on its refinement method.

bidirectional degree search The degree of approximate GCD can be estimated by the QR decomposition of the Sylvester matrix with time complexity $O(m^3)$. Once a candidate degree detected we can start to find the correct one from this degree. This may reduce the number of searching stages. Moreover, by the modified GKO method Fastgcd reduces this time complexity to $O(m^2)$. The accuracy of detected degree and overall time complexity depends on its refinement method.

Consequently, the theoretical background above says that the unidirectional degree search is the best from the view point of maximizing the degree, and the bidirectional degree search is the best from the view point of time complexity.

2.7.2. Performance of Refinement Step

The refinement step is not only for approximate GCD in Definition 2 but also in Definition 1 hence we focus on all the algorithms except for QRGCD and ExQRGCD since they have no refinement step. Especially we focus on their matrix sizes and convergent properties.

UVGCD This uses the Gauss-Newton method and its size of the Jacobian matrix is $(m+n+3)\times(m+n-k+3)$ for both cases of real and complex polynomials. The Gauss-Newton method has a kind of quadratic convergence property in some limited cases, however, this property is not guaranteed. In its iteration, $O(m^3)$ operations are required for solving a least squares problem.

Fastgcd This also uses the Gauss-Newton method and its size of the Jacobian matrix is $(m + n + 3) \times (m + n - k + 3)$ for both cases of real and complex polynomials. However, in its iteration, only $O(m^2)$ operations are required for solving a least squares problem since Fastgcd uses the modified GKO method (fast LU factorization).

STLN-GCD1 By its iteration, the STLN problem is to be solved by a least squares with penalty and its size of coefficient matrix is $(2m+2n-k+2)\times(2m+2n-2k+1)$ and $(2m+2n-k+2)\times(2m+2n-2k+1)$ for real and complex polynomials, respectively. Therefore, $O(m^3)$

operations are required for solving each least squares problem. The rate of convergence of this method can approach quadratic as same as of the Gauss-Newton method.

STLN-GCD2 This follows STLN-GCD1 except for the size of coefficient matrix for complex polynomials that becomes $(4m + 4n - 2k + 4) \times (4m + 4n - 4k + 2)$. Moreover, the initial guess may be better than that of STLN-GCD1 hence it may convergent faster.

GPGCD This uses the modified Newton method and its size of the Jacobian matrix is $(m + n - k + 2) \times (2m + 2n - 2k + 4)$ and $(2m + 2n - 2k + 3) \times (4m + 4n - 4k + 8)$ for real and complex polynomials, respectively. In its iteration, $O(m^3)$ operations are required for solving a linear equation.

GHLGCD This uses the Newton method and its size of the Hessian matrix is $(3m + 3n - 3k + 6) \times (3m + 3n - 3k + 6)$ for real polynomials. This method has the quadratic convergence property and $O(m^3)$ operations are required for solving a least squares problem in each iteration.

Consequently, the theoretical background above says that the method of Fastgcd is the best from the view point of time complexity in each iteration, and the method of GHLGCD is the best from the view point of convergent ratio.

3. Interchangeable Framework and Implementation

As described in the previous subsections, all the algorithms basically follow the steps as in Algorithm 5, and Table 2 shows their sub-routines used in each step. We give some short notes for "n/a" in the table. 1) QRGCD and ExQRGCD do not have any original refinement step, 2) the STLN based methods, GPGCD and GHLGCD are designed for Definition 2 and they do not need to find the degree of approximate GCD, and 3) the STLN based methods and GHLGCD do not have any original method to find the approximate GCD since they only compute perturbed polynomials $f(x) + \Delta_f(x)$ and $g(x) + \Delta_g(x)$, and leave the choice of the method to find $d(x)$ to the readers' discretion.

By understanding Algorithm 5 and Table 2 as an unified framework of approximate GCD, we can disassemble the algorithms and reassemble their components as new algorithms. Therefore, from the practical point of view, we are interested in the most effective combination of components (cf. Remark 2) since unfortunately at least in the published literatures, any trial that interchanges sub-routines between the known algorithms has not been shown.

3.1. Arrangements of Sub-routines

We consider the following combinations represented by the regular expression where the symbols `q`, `e`, `u`, `f` and `g` denote the sub-routines of QRGCD, ExQRGCD, UVGCD, Fastgcd and GPGCD, respectively. Moreover, the program is linked to the single thread BLAS/LAPACK libraries if the prefix is `s`, and the threaded (parallel threads) BLAS/LAPACK libraries if it is `t`. Note that we run the programs on the box with 12 cores (24 hyper-threading logical cores).

`(s|t)-qrgcd-(q|u|f)`
: QRGCD with the refinement method (for $d_i(x)$ in the inner most step) specified by `q`,`u`,`f` hence `s-qrgcd-q` is the original version (no refinement).

Loop Structure	Candidate Degree	Candidate Factors		Refinement Step
		approx.GCD	cofactors	
QRGCD 2 stages w.r.t. zeros	QR of Syl(f,g)	row vector of R	least squares (from GCD)	n/a
ExQRGCD multi stages w.r.t. zeros	QR of Syl(f,g)	row vector of R	least squares (from GCD)	n/a
UVGCD unidirectional degree search	smallest singular value of $S_k(f,g)$	least squares (from cofacts)	right singular vector of $S_k(f,g)$	Gauss-Newton by QR
Fastgcd bidirectional degree search	LU of Syl(f,g)	FFT (from cofacts)	Ker(U) of $S_k(f,g)$	Gauss-Newton by GKO
STLN-GCD1 n/a	n/a	n/a	least squares	STLN
STLN-GCD2 n/a	n/a	n/a	Lagrangian multipliers	STLN or STLS
GPGCD n/a	n/a	least squares (from cofacts)	SVD of $S_k(f,g)$	modified Newton
GHLGCD n/a	n/a	n/a	SVD and least squares	Newton

Table 2: Framework (with mainly used approaches)

`(s|t)-exqrgcd-(e|u|f|g)`
: ExQRGCD with the refinement method (for $d(x)$ in the inner most step) specified by `q`,`u`,`f`,`g` hence `s-exqrgcd-e` is the original version (no refinement).

`(s|t)-fastgcd-(f|r)(f|u|s|t)(f|u)(f|u)`
: The 1st grouping defines the method of candidate degree where `r` denotes the method of PivQR[3](Boito, 2011). The 2nd grouping defines the method of initial cofactors where `u` denotes UVGCD without any pivoting, `s` denotes UVGCD with column pivoting, and `t` denotes UVGCD with UVGCD's pivoting strategy. Please note that `u`,`s`,`t` compute the right singular vector by the method used in UVGCD but based on the conventional QR factorization (not the one in UVGCD). The 3rd grouping defines the method of initial approximate GCD. The last grouping defines the method of refinement. `s-fastgcd-ffff` is the original version.

`(s|t)-uvgcd-(u|g)`
: UVGCD with the refinement method specified by `u` and `g` hence `s-uvgcd-u` is the original version.

[3]Finding the lowest diagonal element of R (computed with column pivoting) whose absolute value is less than the certain value, proposed in Boito's thesis. Our implementation uses the F-norm instead of the 2-norm.

others for tolerance given

`(s|t)-qrgcd-p`, `(s|t)-exqrgcd-p` and `(s|t)-uvgcd-p` denote QRGCD, ExQRGCD and UVGCD with pivoting, described in "Pivoting Capability" subsections, respectively.

others for degree given

`(s|t)-stlngcd1`, `(s|t)-stlngcd2-(a|y)`, `(s|t)-gpgcd` and `(s|t)-ghlgcd` denote STLN-GCD1, STLN-GCD2, GPGCD and GHLGCD, respectively, where `a` and `y` denote ATLAS's `gels` sub-routine and the reference LAPACK's `gelsy` sub-routine for computing the QR decomposition. Moreover, we implemented the degree given version of UVGCD and denote it by `(s|t)-uvgcd-(a|y)`.

In total, the number of combinations is 102. Moreover, as described in the previous section, all the method can find an approximate GCD over $\mathbb{R}$ if $f(x), g(x) \in \mathbb{R}[x]$, and an approximate GCD over $\mathbb{C}$ if $f(x), g(x) \in \mathbb{C}[x]$ hence the number becomes 203 (102 for over $\mathbb{R}$ and 101 for over $\mathbb{C}$ [4]). Their performance result is shown in the next section.

3.2. Implementation

We implemented the combinations[5] proposed in the previous subsections, with GNU C Compiler 5.4.0 (optimized with -O2 -march=native), ATLAS 3.11.39 (as BLAS), reference LAPACK 3.7.0 (through LAPACKE) and FFTW 3.3.4 (Fastgcd requires FFT) on Ubuntu 16.04 LTS (x86_64, kernel 4.4.0) with Intel Xeon E5-2687W v4 and 256GB memory.

We implemented QRGCD as proposed in the original paper and followed several schemes and thresholds used in the implementation of QRGCD in Maple, as possible as we can. As for ExQRGCD, we also implemented it exactly as proposed in the original paper. ExQRGCD is an extended version of QRGCD hence they share the implementation of certain sub-routines. We also tried to implement the other algorithms (UVGCD and so on) under the same coding rule (write as in the original paper and do not optimize the code explicitly) as for QRGCD and ExQRGCD. For vector and matrix computations, we try to use corresponding BLAS and LAPACK sub-routines as possible as we can. For STLN-GCD1, STLN-GCD2 and GHLGCD, we use the sub-routine of UVGCD to compute an approximate GCD $d(x)$ of computed perturbed polynomials. Moreover, we give the following notes on each algorithm.

UVGCD This computes the smallest singular value and vector by the iterative method hence it requires the threshold (stop condition) but not given clearly in the original paper. We stop the iteration if the 2-norm of the correction is less than or equal to 10^{-12} or the number of iterations is greater than 3 (elapsed time heavily depends these values). Moreover, the definition of approximate GCD is a bit different from ours hence we modified the output criterion to being compatible.

Fastgcd This calls the modified GKO method hence we implemented the algorithm 3 in Gu (1998). Since the other algorithms use the well-implemented sub-routines of ATLAS or the reference LAPACK for their matrix factorizations, our implementation of the modified GKO method may not provide good performance against ATLAS/LAPACK though we use the sub-routines of BLAS as possible as we can. For most of selectable schemes and thresholds in Fastgcd, we follows Boito's implementation of Fastgcd for Matlab. For

[4] We've not yet implemented the complex version of GHLGCD.

[5] The whole implementation is available online at the URL: https://wwwmain.h.kobe-u.ac.jp/~nagasaka/research/snap/

example, we use $K = 1$ that the integer controls the number of small QR factorization in the modified GKO method, and we use the threshold 10^{-9} to find the null space. As for the line search method in the Gauss-Newton method in Fastgcd, we use the Newton-Raphson method (max iteration is only 3) for the polynomial in the step size variable of degree 4.

GPGCD The modified Newton method of GPGCD requires the pseudo inverse of the Jacobian matrix. However, the required computation can be done by solving a linear system (`getrf` + `getri`) hence we do not use any SVD here.

STLN based methods STLN-GCD1 and STLN-GCD2 solve the STLN problem by the least squares with penalty. In our implementation, we follow the value of penalty $w = 10^9$ as in the original paper.

Least Squares solvers In the reference LAPACK, there are several sub-routines to solve the least squares: `gels`, `gelsd`, `gelsy` and so on. In general, `gels` is the fastest and `gelsd` is the most reliable. However, `gelsy` is as fast as `gels` and more reliable, and ATLAS has its original implementation of `gels` that are much faster than `gelsy`. In our implementation, if not specified, for the problem in Definition 1 we use the ATLAS's `gels`, and for the problem in Definition 2 we use `gelsy`.

4. Performance Test and Result

We have prepared the following tolerance sensitive problems where the 2-norm of perturbed polynomials (as noisy error) in each pair is 10^{-8}. This performance test with the input tolerance $\varepsilon = 10^{-8}$ is not easy task since for each pair the expected tolerance of the expected degree of approximate GCD is mostly in $[10^{-8}, 10^{-7}]$.

half For $\ell \in \mathbb{Z}_{>0}$, we have generated 100 pairs of polynomials of degree 10ℓ, of unit 2-norm. Each pair has the GCD of degree 5ℓ and each factor has integer coefficients randomly chosen from $[-99, 99]$ before normalization. We added the same degree polynomials whose norm is 10^{-8}, and made them re-normalized again.

low Similar to the half degree except that each pair has the GCD of degree ℓ instead of 5ℓ.

asym Similar to the low degree except that degrees of each pair of polynomials are 2ℓ and 18ℓ.

Remark 4. *As noted in the previous section, we have not done any optimization on the code and our implementation is non-official except ExQRGCD since ExQRGCD is proposed by the present author. Therefore, timing data should be considered as just a reference (including ExQRGCD).*

4.1. Result of Tolerance Given Problem

Table 3 shows the whole result (but only for $\ell = 10$) of **half**, **low** and **asym** for the tolerance ($\varepsilon = 1.0^{-8}$) given problem of Definition 1, by the single thread version binaries (e.g. `s-*`) since threaded version binaries (e.g. `t-*`) are not faster for this size of problems according to our results. Each cell has 2 numbers that are the averages of detected degree of approximate GCD and the elapsed time (sec.) (not CPU time), respectively. Bold styled numbers denote the best 3 results (elapsed time) in each column among the results whose detected degree is the expected degree (i.e. 50, 50, 10, 10, 10 and 10, respectively). For those bold styled combinations, Figures

1, 2 and 3 show the elapsed time for $\ell = 1, 2, \ldots, 10$. Note that we ran the programs once for each pair $f(x)$ and $g(x)$ in this order and also once for the reverse order (i.e. $g(x)$ and $f(x)$) to avoid any effect of input ordering (all the tables and figures show the averages).

Unfortunately there is not so significant difference between "with pivoting" and "without pivoting" though the elapsed time are slightly different. Therefore, any pivoting strategy for QR factorization is useless at least for our tests. As for the extension in Subsection 2.3.1, Fastgcd can find approximate GCDs over $\mathbb{R}$ as well as over $\mathbb{C}$.

The result shows `uvgcd` is the best to find the degree of approximate GCD, and `exqrgcd-u` and `fastgcd-***u` are faster than `uvgcd` in some cases among the programs which found nearly the best degree. This indicates that the refinement method of UVGCD is well designed and it can refine candidate factors even if they are far from the optimum. However, UVGCD is weak for polynomial pairs with GCD of low degree as shown in Figure 2 and also reported by Boito (2011). The total computational complexities of `exqrgcd-u`, `uvgcd` and `fastgcd-***u` are not so different if we ignore the number of iterations required by each iterative method. However, UVGCD needs to compute the smallest singular value n times at most (i.e. $k = n, \ldots, 1$), and the worst complexity of each iteration is $O(m^2)$. This unidirectional degree search is too time-consuming if the expected degree is very small though this feature maximizes the degree of approximate GCD found. In contrast, the loop structure of ExQRGCD repeatedly seeks an approximate common divisor and combines them hence this is faster than UVGCD for **low** polynomial pairs since the number of this iteration is bounded by the degree of GCD at most.

4.1.1. Additional Test

We also use the following known polynomials for further performance test.

Boito 8.1.1 (Zeng's Test 2) We normalize the following polynomials.

$$f(x) = \prod_{j=1}^{10}(x - \omega_j),\ g(x) = \prod_{i=j}^{10}(x - \omega_j + 10^{-j}),\ \omega_j = (-1)^j(j/2).$$

Boito 8.1.2 We normalize the following polynomials.

$$f(x) = \prod_{j=1}^{16}(x - x_j),\ g(x) = \prod_{j=1}^{16}(x - y_j),$$

$$\{x_1, \ldots, x_{16}\} = \{a + b\mathrm{i} \mid a, b \in \{\pm 0.35, \pm 1.05\}\},\ y_j = x_j + 10^{-j}(1 + \mathrm{i}).$$

Boito 8.2.1 We normalize the following polynomials.

$$f(x) = x^{20} + (x - 1/5)^7,\ g(x) = f'(x).$$

Boito 8.2.2 We normalize the following polynomials.

$$f(x) = x^{100} + (x - 1/2)^{17},\ g(x) = f'(x).$$

Boito 8.4.1 (Zeng's Test 1) We normalize the following polynomials.

$$f(x) = f_1(x)d(x),\ g(x) = g_1(x)d(x),\ \ell = n/2$$

$$d(x) = \prod_{j=1}^{\ell}((x - r_1\alpha_j)^2 + r_1^2\beta_j^2),\ r_1 = 0.5,\ r_2 = 1.5,$$
$$f_1(x) = \prod_{j=1}^{\ell}((x - r_2\alpha_j)^2 + r_2^2\beta_j^2),\ \alpha_j = \cos(j\pi/n),$$
$$g_1(x) = \prod_{j=\ell+1}^{n}((x - r_1\alpha_j)^2 + r_1^2\beta_j^2),\ \beta_j = \sin(j\pi/n).$$

Boito 8.6.1 We normalize the following polynomials.

$$f(x) = (x^3 + 3x - 1)(x - 1)^n,\ g(x) = f'(x).$$

Boito 8.9.2(1) Let $n_1 = 25\ell$, $n_2 = 15\ell$, $n_3 = 10\ell$ and define $f(x) = f_1(x)d(x)$ and $g(x) = g_1(x)d(x)$, where $f_1(x) = (x^{n_1} - 1)(x^{n_2} - 2)(x^{n_3} - 3)$, $g_1(x) = (x^{n_1} + 1)(x^{n_2} + 5)(x^{n_3} + \mathrm{i})$ and $d(x) = x^4 + 10x^3 + x - 1$. Then, we normalize these polynomials.

Tables 4, 5, 6, 7, 8 and 9 show the results of Boito 8.1.1, 8.1.2, 8.2.1, 8.2.2, 8.4.1 and 8.6.1 for the tolerance ($\varepsilon = 1.0^{-5}$ if not specified) given problem in Definition 1, by the single thread version binaries (e.g. `s-*`). Each cell has 2 numbers that are the averages of detected degree of approximate GCD and the sizes of perturbations (residues), respectively. Figure 4 shows the elapsed time of Boito 8.9.2(1) for $\ell = 1, 2, \ldots, 20$ (i.e. $m = 54, 104, \ldots, 1004$). Note that we show only a subset of results selected according to the results of **half**, **low** and **asym**.

The results indicate that the multi stage factor peeling works well though it is not better than the unidirectional degree search, and the bidirectional degree search is not so better than others. The latter fact is not expected before since the bidirectional search enlarges the candidate degree as mush as found. However, this is based on the fact already reported (i.e. Zeng (2011), Terui (2013) and so on), that the systems to be solved in the algorithms are ill-conditioned if there exists any nearby other approximate GCD of the same degree or more. Therefore, for example, it is easily happened that approximate GCDs of degree $k = \alpha$ and $k = \alpha + 2$ can be found but any GCD of degree $k = \alpha + 1$ can not be found. This makes the multi stage peeling and unidirectional search being advantageous though the unidirectional search is weak for polynomial pairs with GCD of low degree as shown in Figure 4.

4.2. Result of Degree Given Problem

Tables 10 and 11 show the whole result (but only for $\ell = 10$) of **half**, **low** and **asym** for the degree ($k = 50$, 10 and 10, respectively) given problem in Definition 2, by the single thread version binaries (e.g. `s-*`) since threaded version binaries (e.g. `t-*`) are not faster for this size of problems according to our results. Each cell has 2 numbers that are the averages of detected perturbations ($\sqrt{\|\Delta_f\|_2^2 + \|\Delta_g\|_2^2}$) and the elapsed time (sec.) (not CPU time), respectively. For the best 4 combinations (`(s|t)-stlngcd2-(y|a)` and `(s|t)-uvgcd-(a|y)`), Figures 5, 6 and 7 show the elapsed time for $\ell = 20, 30, \ldots, 100$ (i.e. $m + n = 400, 600, \ldots, 2000$, respectively), and Table 12 shows the average numbers of iterations used in their refinement steps. Note that we ran the programs once for each pair $f(x)$ and $g(x)$ in this order and also once for the reverse order (i.e. $g(x)$ and $f(x)$) to avoid any effect of input ordering (all the tables and figures show the averages).

Unfortunately GHLGCD is not better in the elapsed time than UVGCD and STLN-GCD2 though its perturbations are enough small and nearly indistinguishable from UVGCD and STLN-GCD2. Since the size of matrix in GHLGCD is smaller than that in STLN-GCD2, this means that the computation of the Hessian matrix is much time-consuming.

The result shows that `uvgcd-*` achieved consistent performance and is the best to minimize the perturbation. The resulting perturbations of `stlngcd2-y` are nearly indistinguishable from

Strategy 6: choosing Loop Structure

if *the expected degree of approximate GCD (k/n) is not small* **then**
  use the UVGCD's unidirectional degree search;
else if *the expected approximate GCD has any cluster of large multiplicity zeros* **then**
  use the Fastgcd's bidirectional degree search;
else
  use the ExQRGCD's multi stage factor peeling;
end

`uvgcd-*`, however `stlngcd2-y` is much slower than `uvgcd-*`. For polynomials of higher degree, the multi threads binaries of `stlngcd2-*` are much faster than the single thread binaries while the multi threads binaries of `uvgcd-*` are not so different from the single thread binaries. This is because of the following two reasons. The size of matrix of `stlngcd2-*` is much larger than that of `uvgcd-*` hence the parallelization of `stlngcd2-*` has larger effect. The number of iterations of `uvgcd-*` is much larger than that of `stlngcd2-*` hence the elapsed time of its sequential part of `uvgcd-*` is not negligible.

4.2.1. Additional Test

We also use the following known polynomials for further performance test.

Boito 8.3.1 (Zeng's Test 3) For $k \in \mathbb{N}$, let $d(x)$ be a polynomial of degree k whose coefficients are random integers in $[-5, 5]$ and define $f(x) = f_1(x)d(x)$ and $g(x) = g_1(x)d(x)$, where $f_1(x) = \sum_{j=0}^{3} x^j$ and $g_1(x) = \sum_{j=0}^{4}(-x)^j$. Then, we normalize these polynomials.

Table 13 shows the result of Boito 8.3.1 for the degree given problem in Definition 2. Each cell has 2 numbers that are the averages of detected perturbations ($\sqrt{\|\Delta_f\|_2^2 + \|\Delta_g\|_2^2}$) and the elapsed time (sec.) (not CPU time), respectively. Note that `stlngcd1` ($k = 500$) with the input polynomials $g(x)$ and $f(x)$ in this order did not converge, and its result with $f(x)$ and $g(x)$ in this order is 0.608`e-`15 (0.560) and 0.557`e-`15 (1.007) by `s-` and `t-`, respectively. The result shows that `uvgcd-*` achieved consistent performance again and is the best to minimize the perturbation.

5. Recommendation and Concluding Remarks

According to our experiments and theoretical backgrounds, we recommend our algorithm selection strategies for approximate GCD: Strategies 6 and 7. For choosing Loop Structure, the unidirectional degree search is the unique answer except for the case that the expected degree of approximate GCD is small. If it is small, we have two choices: the bidirectional degree search and the multi stage factor peeling. The result of Boito 8.6.1 indicates the multi stage factor peeling is weak for polynomials with a cluster of large multiplicity zeros hence in this case we recommend the bidirectional degree search. For choosing Refinement Step, the UVGCD's method is also the unique answer from the view points of computational time and perturbation. However, the numerical results indicate that `t-stlngcd2-a` is much faster than `*-uvgcd-*` for large degree polynomials if any fast multi-threaded (parallel) BLAS/LAPACK and multi-cores hardware are available.

Strategy 7: choosing Refinement Step

if *the degree of given polynomials is not large* **then**
  use the UVGCD's refinement method;
else if *fast multi-threaded BLAS/LAPACK and multi-cores hardware are available* **then**
  use the STLN-GCD2's refinement method;
else
  use the UVGCD's refinement method;
end

In this paper, we are interested in implementing modern algorithms for approximate GCD that has various possibilities, pivoting, finding over $\mathbb{R}$ or $\mathbb{C}$, selecting the sub-routines and matrix factoring methods. Our numerical experiments indicate the followings. QRGCD (including ExQRGCD) and UVGCD are compatible with the QR factorization with pivoting however pivoting does not improve their performance. Fastgcd with the small modification is able to find approximate GCDs over $\mathbb{R}$ and it works as well as other algorithms, which is originally designed for over $\mathbb{C}$ though. In practise, the Gauss-Newton refinement of UVGCD is much better than that of Fastgcd, and ExQRGCD with the refinement method of UVGCD works very well though it was not expected before. In theory, the computational complexity of Fastgcd is much better than others. However, the performance is not well. This may be because of the following reason. Our implementation of the modified GKO method does not provide good performance, and the LU factorization does not preserve the row-wise norms hence it is worse than the QR factorization of ATLAS/LAPACK. STLN-GCD2 is not so time-consuming and is competitive with UVGCD if we use any fast parallel BLAS/LAPACK library.

In summary for the future research, replacing the LU factorization with the QR factorization may make Fastgcd much better since the algorithms based on the QR factorization work well. Therefore, the QR factorization of complexity $O(m^2)$ will make all the algorithms better further in the both of theory and practise. However, though we tried to use one of such QR factoring method (Delvaux et al., 2010), it is not numerical stable for the Cauchy-like matrix converted from a Toeplitz-like matrix. To improve all the algorithms for approximate GCD, we need any BLAS/LAPACK compatible (i.e. practical fast) and numerical stable QR factoring algorithm of complexity $O(m^2)$.

Moreover, we are interested in disassembling the recent algorithms (Usevich and Markovsky (2017) and Fazzi et al. (2018)) and reassembling their components in our framework, since according to their papers, their performance seem to be nearly equal to UVGCD. We note that in the numerical examples of Fazzi et al. (2018), UVGCD is much worse than their algorithm, however, for their pairs of polynomials, our implementation of UVGCD computes smaller resulting perturbations than their algorithm.

Acknowledgements

The author wishes to thank Prof. Erich Kaltofen for his wide-range contributions to Symbolic-Numeric computations, including various constructive comments to younger researchers (including the author).

References

Bini, D. A., Boito, P., 2007. Structured matrix-based methods for polynomial ϵ-gcd: analysis and comparisons. In: ISSAC 2007. ACM, New York, pp. 9–16.

Boito, P., 2011. Structured matrix based methods for approximate polynomial GCD. Vol. 15 of Tesi. Scuola Normale Superiore di Pisa (Nuova Series) [Theses of Scuola Normale Superiore di Pisa (New Series)]. Edizioni della Normale, Pisa.

Bourne, M., Winkler, J. R., Su, Y., 2017. A non-linear structure-preserving matrix method for the computation of the coefficients of an approximate greatest common divisor of two Bernstein polynomials. J. Comput. Appl. Math. 320, 221–241.

Christou, D., Karcanias, N., Mitrouli, M., 2010. The ERES method for computing the approximate GCD of several polynomials. Appl. Numer. Math. 60 (1-2), 94–114.

Corless, R. M., Watt, S. M., Zhi, L., 2004. *QR* factoring to compute the GCD of univariate approximate polynomials. IEEE Trans. Signal Process. 52 (12), 3394–3402.

Cox, A. J., Higham, N. J., 1998. Stability of Householder *QR* factorization for weighted least squares problems. In: Numerical analysis 1997 (Dundee). Vol. 380 of Pitman Res. Notes Math. Ser. Longman, Harlow, pp. 57–73.

Delvaux, S., Gemignani, L., Van Barel, M., 2010. *QR*-factorization of displacement structured matrices using a rank structured matrix approach. In: Numerical methods for structured matrices and applications. Vol. 199 of Oper. Theory Adv. Appl. Birkhäuser Verlag, Basel, pp. 229–254.

Dunaway, D. K., 1974. Calculation of zeros of a real polynomial through factorization using Euclid's algorithm. SIAM J. Numer. Anal. 11, 1087–1104.

Emiris, I. Z., Galligo, A., Lombardi, H., 1996. Numerical univariate polynomial GCD. In: The mathematics of numerical analysis (Park City, UT, 1995). Vol. 32 of Lectures in Appl. Math. Amer. Math. Soc., Providence, RI, pp. 323–343.

Emiris, I. Z., Galligo, A., Lombardi, H., 1997. Certified approximate univariate GCDs. J. Pure Appl. Algebra 117/118, 229–251, algorithms for algebra (Eindhoven, 1996).

Fazzi, A., Guglielmi, N., Markovsky, I., 2018. An ODE-based method for computing the approximate greatest common divisor of polynomials. Numer. Algor.

Giesbrecht, M., Haraldson, J., Labahn, G., 2017. Computing the nearest rank-deficient matrix polynomial. In: ISSAC'17—Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation. ACM, New York, pp. 181–188.

Gu, M., 1998. Stable and efficient algorithms for structured systems of linear equations. SIAM J. Matrix Anal. Appl. 19 (2), 279–306.

Kaltofen, E., Yang, Z., Zhi, L., 2006. Approximate greatest common divisors of several polynomials with linearly constrained coefficients and singular polynomials. In: ISSAC 2006. ACM, New York, pp. 169–176.

Kaltofen, E., Yang, Z., Zhi, L., 2007. Structured low rank approximation of a Sylvester matrix. In: Symbolic-numeric computation. Trends Math. Birkhäuser, Basel, pp. 69–83.

Laidacker, M. A., 1969. Another theorem relating Sylvester's matrix and the greatest common divisor. Math. Mag. 42, 126–128.

Nagasaka, K., 2011. Approximate polynomial GCD over integers. J. Symbolic Comput. 46 (12), 1306–1317.

Nagasaka, K., Masui, T., 2013. Extended qrgcd algorithm. In: Proceedings of the 15th International Workshop on Computer Algebra in Scientific Computing - Volume 8136. CASC 2013. Springer-Verlag New York, Inc., New York, NY, USA, pp. 257–272.

Noda, M.-T., Sasaki, T., 1991. Approximate GCD and its application to ill-conditioned algebraic equations. In: Proceedings of the International Symposium on Computational Mathematics (Matsuyama, 1990). Vol. 38(1-3). pp. 335–351.

Powell, M. J. D., Reid, J. K., 1969. On applying Householder transformations to linear least squares problems. In: Information Processing 68 (Proc. IFIP Congress, Edinburgh, 1968), Vol. 1: Mathematics, Software. North-Holland, Amsterdam, pp. 122–126.

Roy, M.-F., Sedjelmaci, S. M., 2013. New fast Euclidean algorithms. J. Symbolic Comput. 50, 208–226.

Sasaki, T., Noda, M.-T., 1989. Approximate square-free decomposition and root-finding of ill-conditioned algebraic equations. J. Inform. Process. 12 (2), 159–168.

Schönhage, A., 1985. Quasi-gcd computations. J. Complexity 1 (1), 118–137.

Terui, A., 2013. GPGCD: an iterative method for calculating approximate GCD of univariate polynomials. Theoret. Comput. Sci. 479, 127–149.

Usevich, K., Markovsky, I., 2017. Variable projection methods for approximate (greatest) common divisor computations. Theoret. Comput. Sci. 681, 176–198.

von zur Gathen, J., Mignotte, M., Shparlinski, I. E., 2010. Approximate polynomial gcd: small degree and small height perturbations. J. Symbolic Comput. 45 (8), 879–886.

Zeng, Z., 2011. The numerical greatest common divisor of univariate polynomials. In: Randomization, relaxation, and

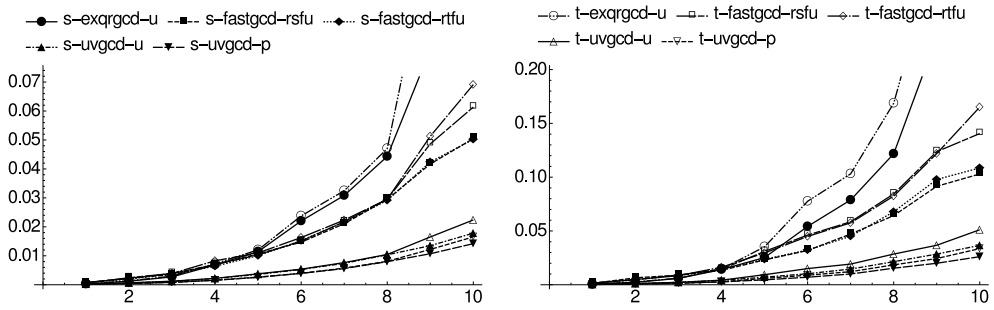

Figure 1: Elapsed time of **half** tolerance given (left: over $\mathbb{R}$, right: over $\mathbb{C}$)

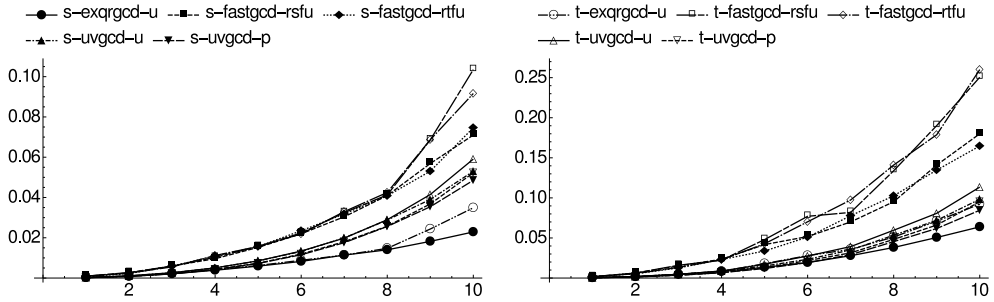

Figure 2: Elapsed time of **low** tolerance given (left: over $\mathbb{R}$, right: over $\mathbb{C}$)

complexity in polynomial equation solving. Vol. 556 of Contemp. Math. Amer. Math. Soc., Providence, RI, pp. 187–217.

Zhi, L., Yang, Z., 2004. Computing approximate gcd of univariate polynomials by structure total least norm. MM Research Preprints 24, 375–387.

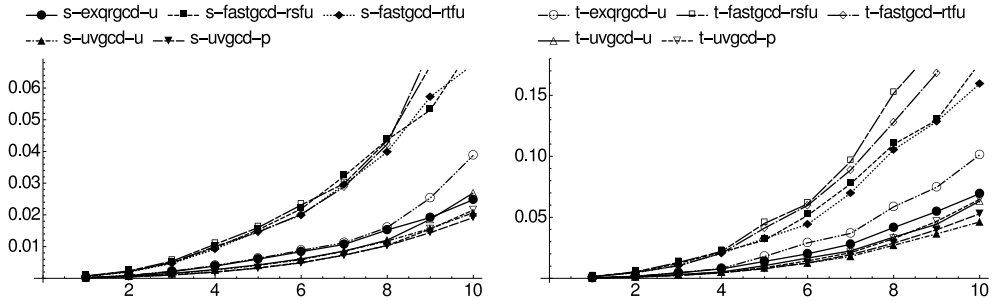

Figure 3: Elapsed time of **asym** tolerance given (left: over $\mathbb{R}$, right: over $\mathbb{C}$)

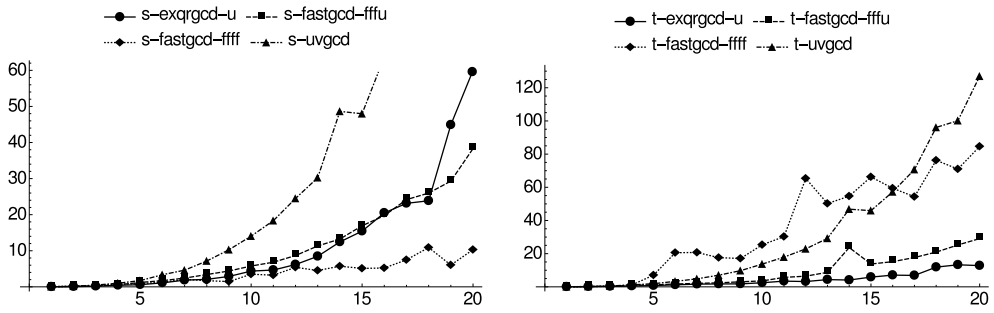

Figure 4: Elapsed time of Boito 8.9.2.1 (left: single thread, right: multi threads)

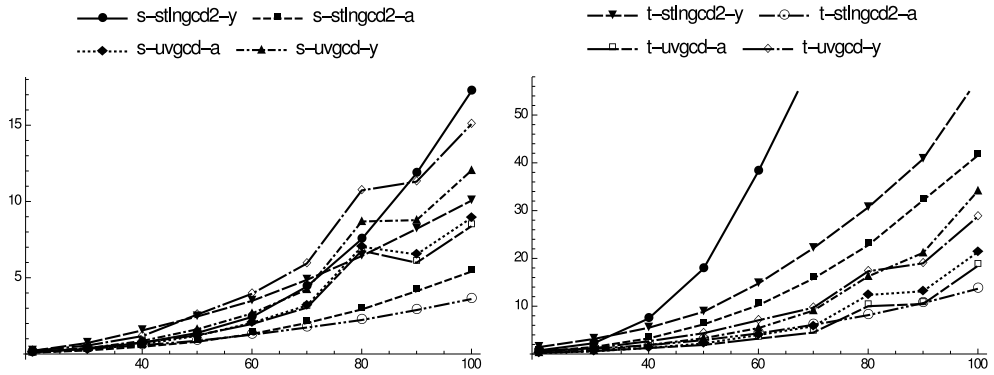

Figure 5: Elapsed time of **half** degree given (left: over $\mathbb{R}$, right: over $\mathbb{C}$)

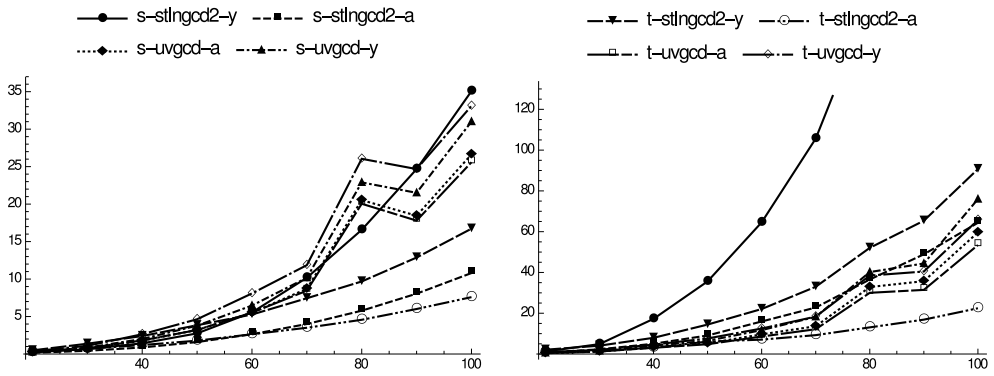

Figure 6: Elapsed time of **low** degree given (left: over $\mathbb{R}$, right: over $\mathbb{C}$)

	half : degree (time in sec.)		**low** : degree (time in sec.)		**asym** : degree (time in sec.)	
	over $\mathbb{R}$	over $\mathbb{C}$	over $\mathbb{R}$	over $\mathbb{C}$	over $\mathbb{R}$	over $\mathbb{C}$
`s-qrgcd-q`	0.0 (0.0102)	0.0 (0.0259)	1.2 (0.0041)	1.2 (0.0097)	1.5 (0.0040)	1.5 (0.0098)
`s-qrgcd-p`	0.0 (0.0164)	0.0 (0.0415)	1.2 (0.0079)	1.2 (0.0187)	1.5 (0.0079)	1.5 (0.0177)
`s-qrgcd-u`	0.0 (0.0102)	0.0 (0.0258)	1.2 (0.0059)	1.2 (0.0151)	1.6 (0.0066)	1.6 (0.0174)
`s-qrgcd-f`	0.0 (0.0101)	0.0 (0.0258)	1.1 (0.0152)	1.1 (0.0155)	1.5 (0.0339)	1.5 (0.0184)
`s-exqrgcd-e`	0.0 (0.0359)	0.0 (0.1004)	4.1 (0.0246)	4.1 (0.0704)	2.3 (0.0125)	2.3 (0.0332)
`s-exqrgcd-p`	0.0 (0.0578)	0.0 (0.1636)	4.1 (0.0416)	4.1 (0.1146)	2.3 (0.0169)	2.3 (0.0429)
`s-exqrgcd-u`	47.7 (0.1061)	47.7 (0.3028)	**10.0 (0.0229)**	**10.0 (0.0638)**	**10.0 (0.0248)**	**10.0 (0.0694)**
`s-exqrgcd-f`	43.7 (0.4963)	35.2 (0.4774)	9.7 (0.2824)	9.7 (0.1347)	8.5 (0.4928)	8.4 (0.2658)
`s-exqrgcd-g`	0.0 (6.9534)	0.0 (42.6231)	1.1 (6.4077)	1.2 (39.5741)	0.4 (6.5046)	0.6 (39.2894)
`s-uvgcd-u`	**50.0 (0.0178)**	**50.0 (0.0366)**	**10.0 (0.0530)**	**10.0 (0.0983)**	**10.0 (0.0207)**	**10.0 (0.0462)**
`s-uvgcd-g`	50.0 (0.0907)	50.0 (0.8938)	10.0 (0.1525)	10.0 (0.8846)	10.0 (0.0573)	9.6 (38.3071)
`s-uvgcd-p`	**50.0 (0.0142)**	**50.0 (0.0261)**	**10.0 (0.0485)**	**10.0 (0.0846)**	**10.0 (0.0191)**	**10.0 (0.0529)**
`s-fastgcd-ffff`	43.3 (0.3198)	48.3 (0.2343)	9.4 (0.2584)	9.7 (0.1583)	6.1 (0.3353)	7.9 (0.2610)
`s-fastgcd-fffu`	49.7 (0.0378)	50.0 (0.0900)	9.9 (0.0556)	9.9 (0.1178)	9.0 (0.0644)	9.9 (0.1562)
`s-fastgcd-ffuf`	40.4 (0.3355)	47.3 (0.2215)	9.5 (0.2608)	9.6 (0.1540)	6.3 (0.3159)	7.8 (0.2322)
`s-fastgcd-ffuu`	49.5 (0.0408)	50.0 (0.0873)	9.9 (0.0516)	10.0 (0.1169)	9.1 (0.0619)	9.9 (0.1551)
`s-fastgcd-fuff`	36.0 (0.3820)	20.4 (1.3548)	7.0 (0.2446)	5.6 (0.2700)	3.3 (0.2730)	3.5 (0.4446)
`s-fastgcd-fufu`	47.5 (0.0557)	49.6 (0.0629)	9.8 (0.0429)	9.8 (0.1254)	7.4 (0.0709)	8.3 (0.1731)
`s-fastgcd-fuuf`	35.3 (0.3234)	22.7 (0.6116)	7.3 (0.1959)	6.7 (0.1656)	3.2 (0.1913)	3.4 (0.2330)
`s-fastgcd-fuuu`	48.1 (0.0426)	49.7 (0.0596)	9.7 (0.0387)	9.7 (0.1081)	7.2 (0.0679)	8.6 (0.1407)
`s-fastgcd-fsff`	35.4 (0.3968)	22.1 (1.1877)	6.7 (0.2371)	5.9 (0.2744)	3.1 (0.2876)	3.6 (0.4432)
`s-fastgcd-fsfu`	47.9 (0.0616)	48.5 (0.1076)	9.7 (0.0428)	9.8 (0.1277)	7.3 (0.0740)	8.4 (0.1656)
`s-fastgcd-fsuf`	34.8 (0.3378)	24.9 (0.5995)	7.3 (0.1936)	6.9 (0.1610)	3.0 (0.1873)	3.6 (0.2509)
`s-fastgcd-fsuu`	47.1 (0.0548)	49.7 (0.0632)	9.7 (0.0406)	9.8 (0.1126)	7.1 (0.0720)	8.4 (0.1441)
`s-fastgcd-ftff`	35.3 (0.3671)	23.0 (1.0284)	6.9 (0.2393)	6.0 (0.2328)	3.4 (0.2395)	3.9 (0.4130)
`s-fastgcd-ftfu`	48.1 (0.0526)	49.5 (0.0716)	9.8 (0.0393)	9.8 (0.1151)	7.6 (0.0672)	8.6 (0.1476)
`s-fastgcd-ftuf`	35.0 (0.3130)	22.9 (0.4713)	7.3 (0.1941)	6.8 (0.1238)	2.9 (0.1846)	3.8 (0.2293)
`s-fastgcd-ftuu`	47.4 (0.0517)	49.5 (0.0712)	9.8 (0.0342)	9.8 (0.0933)	7.3 (0.0643)	8.8 (0.1198)
`s-fastgcd-rfff`	44.8 (0.3176)	48.4 (0.2827)	9.7 (0.2717)	9.7 (0.2048)	8.6 (0.3411)	8.3 (0.2976)
`s-fastgcd-rffu`	50.0 (0.0634)	50.0 (0.1391)	10.0 (0.0846)	10.0 (0.1725)	10.0 (0.0789)	10.0 (0.1778)
`s-fastgcd-rfuf`	42.1 (0.3344)	47.3 (0.2690)	9.7 (0.2747)	9.7 (0.1999)	8.6 (0.3277)	8.3 (0.2727)
`s-fastgcd-rfuu`	50.0 (0.0647)	50.0 (0.1360)	10.0 (0.0801)	10.0 (0.1687)	10.0 (0.0773)	10.0 (0.1766)
`s-fastgcd-ruff`	37.9 (0.3949)	22.0 (1.3422)	7.2 (0.2759)	6.5 (0.2983)	5.8 (0.3103)	5.0 (0.4204)
`s-fastgcd-rufu`	50.0 (0.0505)	50.0 (0.1054)	9.9 (0.0689)	9.9 (0.1788)	10.0 (0.0663)	10.0 (0.1557)
`s-fastgcd-ruuf`	38.3 (0.3289)	24.8 (0.6289)	7.5 (0.2303)	7.0 (0.2023)	5.9 (0.2282)	5.1 (0.2443)
`s-fastgcd-ruuu`	50.0 (0.0543)	50.0 (0.1037)	9.9 (0.0628)	9.9 (0.1438)	10.0 (0.0667)	10.0 (0.1545)
`s-fastgcd-rsff`	37.7 (0.3992)	22.3 (1.2124)	7.1 (0.2691)	6.2 (0.3125)	5.7 (0.3188)	4.6 (0.4353)
`s-fastgcd-rsfu`	50.0 (0.0505)	**50.0 (0.1030)**	9.9 (0.0710)	9.9 (0.1800)	10.0 (0.0735)	10.0 (0.1765)
`s-fastgcd-rsuf`	38.3 (0.3361)	27.1 (0.5930)	7.6 (0.2387)	7.4 (0.1932)	5.9 (0.2412)	5.0 (0.2654)
`s-fastgcd-rsuu`	50.0 (0.0507)	50.0 (0.1065)	9.9 (0.0698)	9.9 (0.1490)	10.0 (0.0864)	10.0 (0.1501)
`s-fastgcd-rtff`	38.0 (0.3762)	21.7 (1.2164)	7.1 (0.2735)	6.2 (0.2930)	5.8 (0.2986)	4.9 (0.4280)
`s-fastgcd-rtfu`	**50.0 (0.0502)**	50.0 (0.1086)	9.9 (0.0746)	9.9 (0.1647)	10.0 (0.0675)	10.0 (0.1595)
`s-fastgcd-rtuf`	38.3 (0.3173)	23.7 (0.5209)	7.5 (0.2248)	6.7 (0.1808)	5.9 (0.2253)	5.2 (0.2413)
`s-fastgcd-rtuu`	50.0 (0.0518)	50.0 (0.1097)	9.9 (0.0655)	9.9 (0.1556)	10.0 (0.0672)	10.0 (0.1486)

Table 3: Results of tolerance given by single thread binaries for $\ell = 10$

tol.	`uvgcd-u`	`exqrgcd-u`	`fastgcd-ffff`	`fastgcd-fffu`	`fastgcd-rsfu`	`fastgcd-rtfu`
10^{-2}	9.0 (0.3996e-2)	9.0 (0.3996e-2)	9.0 (0.4256e-2)	9.0 (0.3996e-2)	7.0 (0.0136e-2)	9.0 (0.3996e-2)
10^{-3}	8.0 (0.1729e-3)	8.0 (0.1729e-3)	8.0 (0.2049e-3)	8.0 (0.1729e-3)	5.5 (0.2773e-3)	8.0 (0.1729e-3)
10^{-4}	7.0 (0.0709e-4)	7.0 (0.0709e-4)	7.0 (0.1016e-4)	7.0 (0.0709e-4)	2.5 (0.0001e-4)	5.0 (0.1531e-4)
10^{-5}	7.0 (0.7089e-5)	7.0 (0.7089e-5)	7.0 (1.0161e-5)	7.0 (0.7089e-5)	2.5 (0.0011e-5)	3.0 (0.2232e-5)
10^{-6}	6.0 (0.1829e-6)	6.0 (0.1829e-6)	4.5 (0.1414e-6)	6.0 (0.1829e-6)	2.5 (0.0112e-6)	0.5 (0.0005e-6)
10^{-7}	5.0 (0.0449e-7)	5.0 (0.0449e-7)	4.0 (0.0257e-7)	5.0 (0.0449e-7)	0.0 (0.0000e-7)	0.5 (0.2200e-7)
10^{-8}	5.0 (0.4487e-8)	5.0 (0.4487e-8)	4.0 (0.2574e-8)	5.0 (0.4487e-8)	0.0 (0.0000e-8)	0.0 (0.0000e-8)
10^{-9}	4.0 (0.0840e-9)	4.0 (0.0840e-9)	1.5 (0.0012e-9)	4.0 (0.2105e-9)	0.0 (0.0000e-9)	0.0 (0.0000e-9)

Table 4: Results of Boito 8.1.1 (by single thread version over $\mathbb{R}$)

tol.	uvgcd-u	exqrgcd-u	fastgcd-ffff	fastgcd-fffu	fastgcd-rsfu	fastgcd-rtfu
10^{-2}	15.0 (0.723e-2)	15.0 (0.723e-2)	14.5 (0.720e-2)	15.0 (0.723e-2)	14.0 (0.720e-2)	15.0 (0.723e-2)
10^{-3}	13.0 (0.146e-3)	13.0 (0.146e-3)	13.0 (0.212e-3)	13.0 (0.146e-3)	4.5 (0.568e-3)	8.5 (0.071e-3)
10^{-4}	12.0 (0.058e-4)	12.0 (0.058e-4)	12.0 (0.090e-4)	12.0 (0.058e-4)	3.0 (0.019e-4)	7.0 (0.521e-4)
10^{-5}	12.0 (0.576e-5)	12.0 (0.576e-5)	12.0 (0.898e-5)	12.0 (0.576e-5)	3.0 (0.019e-5)	3.0 (0.005e-5)
10^{-6}	11.0 (0.421e-6)	11.0 (0.421e-6)	11.0 (0.741e-6)	11.0 (0.421e-6)	3.0 (0.193e-6)	3.0 (0.054e-6)
10^{-7}	9.0 (0.135e-7)	9.0 (0.135e-7)	9.0 (0.176e-7)	9.0 (0.135e-7)	0.5 (0.14e-12)	5.5 (1.044e-7)
10^{-8}	8.0 (0.151e-8)	8.0 (0.151e-8)	8.0 (0.273e-8)	8.0 (0.151e-8)	0.5 (0.14e-12)	2.0 (0.501e-8)
10^{-9}	7.0 (0.135e-9)	7.0 (0.135e-9)	7.0 (0.249e-9)	7.0 (0.135e-9)	0.5 (0.14e-12)	1.0 (0.13e-12)

Table 5: Results of Boito 8.1.2 (by single thread version over $\mathbb{C}$)

tol.	uvgcd-u	exqrgcd-u	fastgcd-ffff	fastgcd-fffu	fastgcd-rsfu	fastgcd-rtfu
10^{-7}	6.0 (0.2760e-7)	6.0 (0.2760e-7)	6.0 (0.6480e-7)	6.0 (0.2760e-7)	6.0 (0.2760e-7)	6.0 (0.2760e-7)
10^{-8}	5.0 (0.1961e-8)	5.0 (0.1961e-8)	0.0 (0.0000e-8)	4.0 (0.0104e-8)	0.0 (0.0000e-8)	3.5 (0.0076e-8)
10^{-9}	4.0 (0.1036e-9)	4.0 (0.1036e-9)	0.0 (0.0000e-9)	4.0 (0.1036e-9)	0.0 (0.0000e-9)	3.5 (0.0756e-9)
10^{-10}	4.0 (1.036e-10)	4.0 (1.036e-10)	0.0 (0.000e-10)	4.0 (1.036e-10)	0.0 (0.000e-10)	0.0 (0.000e-10)
10^{-11}	3.0 (0.461e-11)	2.0 (0.012e-11)	0.0 (0.000e-11)	2.0 (0.231e-11)	0.0 (0.000e-11)	0.0 (0.000e-11)

Table 6: Results of Boito 8.2.1 (by single thread version over $\mathbb{R}$)

tol.	uvgcd-u	exqrgcd-u	fastgcd-ffff	fastgcd-fffu	fastgcd-rsfu	fastgcd-rtfu
10^{-0}	99.0 (0.012e-0)	99.0 (0.012e-0)	99.0 (0.015e-0)	99.0 (0.012e-0)	99.0 (0.012e-0)	99.0 (0.012e-0)
10^{-1}	99.0 (0.118e-1)	99.0 (0.118e-1)	99.0 (0.153e-1)	99.0 (0.118e-1)	99.0 (0.118e-1)	99.0 (0.118e-1)
10^{-2}	99.0 (1.179e-2)	99.0 (1.179e-2)	58.0 (0.600e-2)	99.0 (1.179e-2)	99.0 (1.179e-2)	99.0 (1.179e-2)
10^{-3}	16.0 (0.014e-9)	16.5 (0.220e-3)	17.0 (0.093e-3)	16.0 (0.230e-3)	4.0 (0.211e-3)	8.0 (0.440e-3)
10^{-4}	16.0 (0.020e-9)	16.5 (0.034e-5)	16.5 (0.487e-4)	15.5 (0.247e-4)	3.0 (0.241e-4)	3.0 (0.495e-4)
10^{-5}	18.0 (0.043e-5)	16.0 (0.013e-9)	8.0 (0.360e-5)	14.5 (0.143e-5)	2.0 (0.085e-7)	0.5 (0.494e-5)
10^{-6}	16.0 (0.013e-9)	16.0 (0.014e-9)	0.0 (0.000e-6)	14.0 (0.350e-6)	0.0 (0.000e-6)	0.0 (0.000e-6)
10^{-7}	16.0 (0.015e-9)	16.0 (0.013e-9)	0.0 (0.000e-7)	0.0 (0.000e-7)	0.0 (0.000e-7)	0.0 (0.000e-7)
10^{-8}	16.0 (0.012e-9)	16.0 (0.013e-9)	0.0 (0.000e-8)	0.0 (0.000e-8)	3.0 (0.280e-8)	0.0 (0.000e-8)
10^{-9}	16.0 (0.014e-9)	16.0 (0.013e-9)	0.0 (0.000e-9)	0.0 (0.000e-9)	0.0 (0.000e-9)	0.0 (0.000e-9)
10^{-10}	16.0 (0.19e-10)	16.0 (0.13e-10)	0.0 (0.00e-10)	0.0 (0.00e-10)	0.0 (0.00e-10)	0.0 (0.00e-10)
10^{-11}	0.0 (0.00e-11)	8.0 (0.53e-11)	0.0 (0.00e-11)	0.0 (0.00e-11)	0.0 (0.00e-11)	0.0 (0.00e-11)

Table 7: Results of Boito 8.2.2 (by single thread version over $\mathbb{R}$)

k	uvgcd-u	exqrgcd-u	fastgcd-ffff (over $\mathbb{R}$, over $\mathbb{C}$)		fastgcd-fffu (over $\mathbb{R}$, over $\mathbb{C}$)	
12	12.0 (2.265e-14)	12.0 (2.218e-14)	12.0 (9.759e-15)	12.0 (2.256e-14)	13.0 (1.112e-5)	12.0 (3.22e-14)
14	14.0 (5.247e-14)	14.0 (6.727e-14)	14.0 (6.479e-14)	14.0 (8.111e-14)	15.0 (4.196e-6)	15.0 (4.196e-6)
16	16.0 (2.648e-13)	16.0 (1.909e-13)	16.0 (1.083e-13)	16.0 (2.215e-13)	17.0 (4.813e-7)	17.5 (2.871e-6)
18	18.0 (1.101e-12)	19.0 (8.7944e-9)	17.0 (3.014e-12)	18.0 (4.377e-12)	21.0 (1.596e-6)	20.0 (1.549e-6)
20	20.0 (3.762e-12)	22.0 (1.0249e-8)	0.0 (0.000e-00)	0.0 (0.000e-00)	13.5 (9.122e-7)	25.0 (4.055e-6)

Table 8: Results of Boito 8.4.1 (by single thread version)

k	uvgcd-u	exqrgcd-u	fastgcd-ffff (over $\mathbb{R}$, over $\mathbb{C}$)		fastgcd-fffu (over $\mathbb{R}$, over $\mathbb{C}$)	
15	15.0 (7.616e-5)	14.0 (8.802e-17)	14.5 (5.791e-5)	14.0 (1.835e-16)	15.0 (7.616e-5)	15.0 (7.616e-5)
25	25.0 (0.787e-5)	24.0 (1.666e-16)	25.0 (3.629e-5)	25.0 (1.4836e-5)	25.0 (0.787e-5)	25.0 (0.787e-5)
35	36.0 (6.177e-5)	20.5 (3.4547e-5)	35.5 (5.134e-5)	35.5 (5.1662e-5)	36.0 (6.177e-5)	36.0 (6.177e-5)
45	46.0 (2.985e-5)	16.5 (2.3327e-5)	46.0 (7.218e-5)	46.0 (5.5176e-5)	46.0 (2.985e-5)	46.0 (2.985e-5)

Table 9: Results of Boito 8.6.1 (by single thread version)

	half over $\mathbb{R}$ residue (time in sec.)	**low** over $\mathbb{R}$ residue (time in sec.)	**asym** over $\mathbb{R}$ residue (time in sec.)
s-gpgcd	1.90889e-8 (0.0107)	1.51085e-8 (0.0188)	1.49530e-8 (0.0237)
s-stlngcd1	5.79764e-9 (0.0187)	2.00984e-9 (0.0322)	7.82311e-8 (0.0878)
s-stlngcd2-y	5.77291e-9 (0.0228)	1.99538e-9 (0.0430)	2.46829e-9 (0.0425)
s-stlngcd2-a	5.77363e-9 (0.0164)	2.02477e-9 (0.0301)	2.61354e-9 (0.0300)
s-uvgcd-a	5.77291e-9 (0.0139)	1.99538e-9 (0.0336)	2.46829e-9 (0.0145)
s-uvgcd-y	5.77291e-9 (0.0198)	1.99538e-9 (0.0412)	2.46829e-9 (0.0225)
s-ghlgcd	5.77291e-9 (0.0452)	1.99538e-9 (0.0887)	2.85186e-9 (0.0894)

Table 10: Results of degree given by single thread binaries for $\ell = 10$ over $\mathbb{R}$

	half over $\mathbb{C}$ residue (time in sec.)	**low** over $\mathbb{C}$ residue (time in sec.)	**asym** over $\mathbb{C}$ residue (time in sec.)
`s-gpgcd`	1.90889e-8 (0.0526)	1.51085e-8 (0.0920)	1.49530e-8 (0.1229)
`s-stlngcd1`	6.99438e-9 (0.0833)	2.81449e-9 (0.1465)	1.13585e-6 (0.3793)
`s-stlngcd2-y`	5.77291e-9 (0.1095)	1.99538e-9 (0.2071)	2.46829e-9 (0.1920)
`s-stlngcd2-a`	5.77291e-9 (0.0979)	1.99538e-9 (0.1529)	2.46830e-9 (0.1484)
`s-uvgcd-a`	5.77291e-9 (0.0297)	1.99538e-9 (0.0567)	2.46829e-9 (0.0340)
`s-uvgcd-y`	5.77291e-9 (0.0554)	1.99538e-9 (0.0921)	2.46829e-9 (0.0675)

Table 11: Results of degree given by single thread binaries for $\ell = 10$ over $\mathbb{C}$

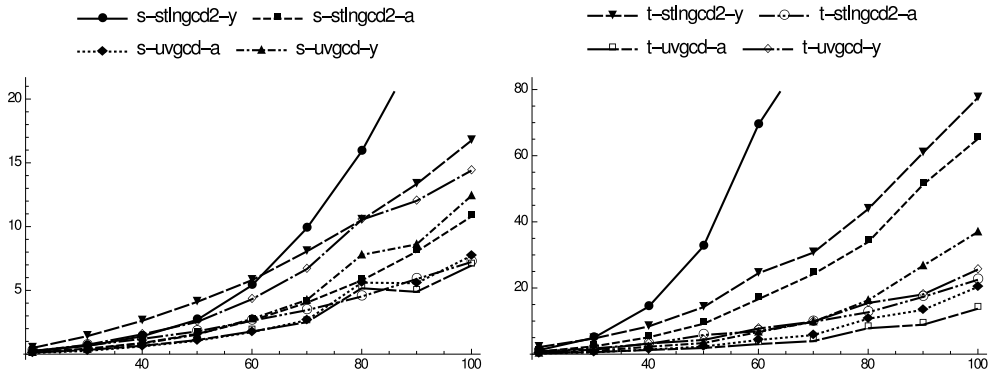

Figure 7: Elapsed time of **asym** degree given (left: over $\mathbb{R}$, right: over $\mathbb{C}$)

	half (`s-`, `t-`)		**low** (`s-`, `t-`)		**asym** (`s-`, `t-`)	
`stlngcd2-y`	2.000	2.000	2.000	2.000	2.000	2.006
`stlngcd2-a`	3.017	3.022	3.000	3.122	3.006	3.039
`uvgcd-a`	7.839	8.022	7.811	7.850	7.933	7.772
`uvgcd-y`	8.117	8.033	7.694	7.756	7.789	7.983

Table 12: The average numbers of iterations in the refinement steps ($\ell = 20, 30, \ldots, 100$)

	$k = 50$	$k = 100$	$k = 200$	$k = 500$	$k = 1000$
`s-gpgcd`	0.372e-14 (0.001)	0.309e-14 (0.003)	0.491e-14 (0.009)	1.657e-14 (0.067)	1.706e-14 (0.398)
`t-gpgcd`	0.252e-14 (0.001)	0.504e-14 (0.004)	0.628e-14 (0.012)	1.485e-14 (0.251)	1.701e-14 (0.493)
`s-stlngcd1`	0.722e-15 (0.003)	0.633e-15 (0.006)	0.528e-15 (0.086)	0.1599024 (26.08)	0.688e-15 (2.915)
`t-stlngcd1`	0.733e-15 (0.003)	0.687e-15 (0.012)	0.568e-15 (0.159)	0.1599024 (56.20)	0.650e-15 (1.954)
`s-stlngcd2-y`	0.691e-15 (0.003)	0.663e-15 (0.014)	0.631e-15 (0.055)	0.613e-15 (0.594)	0.638e-15 (5.784)
`t-stlngcd2-y`	0.647e-15 (0.003)	0.554e-15 (0.020)	0.588e-15 (0.124)	0.605e-15 (0.977)	0.645e-15 (4.253)
`s-stlngcd2-a`	0.0033662 (0.007)	0.687e-15 (0.009)	0.599e-15 (0.039)	0.666e-15 (0.376)	0.663e-15 (2.133)
`t-stlngcd2-a`	0.0026874 (0.007)	0.638e-15 (0.021)	0.571e-15 (0.053)	0.540e-15 (0.362)	0.611e-15 (1.175)
`s-uvgcd-a`	0.206e-15 (0.002)	0.196e-15 (0.008)	0.172e-15 (0.028)	0.193e-15 (0.208)	0.205e-15 (1.063)
`t-uvgcd-a`	0.220e-15 (0.002)	0.194e-15 (0.013)	0.178e-15 (0.150)	0.177e-15 (0.345)	0.185e-15 (0.854)
`s-uvgcd-y`	0.205e-15 (0.003)	0.179e-15 (0.010)	0.191e-15 (0.053)	0.186e-15 (0.521)	0.195e-15 (2.353)
`t-uvgcd-y`	0.245e-15 (0.002)	0.194e-15 (0.010)	0.180e-15 (0.138)	0.183e-15 (0.837)	0.198e-15 (5.396)

Table 13: Results of Boito 8.3.1 as Problem 2 (degree given, over $\mathbb{R}$)