Using Monodromy to Avoid High Precision

Daniel J. Bates and Matthew Niemerg

Abstract. When solving polynomial systems with homotopy continuation, the fundamental numerical linear algebra computations become inaccurate when two paths are in close proximity. The current best defense against this ill-conditioning is the use of adaptive precision. While sufficiently high precision indeed overcomes any such loss of accuracy, high precision can be very expensive. In this article, we describe a simple heuristic rooted in monodromy that can be used to avoid the use of high precision.

1. Introduction

Given a polynomial system of equations,

$$f(z) = \begin{bmatrix} f_1(z_1, z_2, \ldots, z_N) \\ f_2(z_1, z_2, \ldots, z_N) \\ \cdots \\ f_N(z_1, z_2, \ldots, z_N) \end{bmatrix} = 0,$$

in N equations and N variables, the methods of numerical algebraic geometry can be used to find numerical approximations of all isolated solutions[1]. These methods depend heavily on the computational engine of homotopy continuation, described briefly in the next section and fully in [4, 8].

The fundamental operation of homotopy continuation is the numerical tracking of a solution of a parameterized polynomial system $H(z,t) = 0$ as the parameter is varied, e.g., as t moves from 1 to 0. As t varies, the solution traces out a *homotopy path*, or simply a *path*. The green curves in Figure 1 give a simple schematic representation of such paths. The underlying numerical linear algebra methods of path-tracking can be inaccurate if the Jacobian matrix of $H(z,t)$ becomes ill-conditioned. This occurs when two paths are in close proximity, and the Jacobian becomes singular if the paths actually meet at some value of t.

With probability one, paths will never intersect, as described in the next section. However, paths often become quite close, particularly for large problems (high degree or many equations and variables). As reported in [3], for one problem of moderate size, 0.8% of paths encountered periods of ill-conditioning. While this may seem an insignificant amount, the result can be

[1]These methods apply in the non-square case as well [4], but we exclude that setting for simplicity.

missed solutions of $f(z) = 0$ or, similarly, the finding of the same solution multiple times, due to path-crossing.

The current best defense against this sort of ill-conditioning of the Jacobian matrix is the use of adaptive precision arithmetic [2]. Put simply, the number of accurate digits when solving matrix equation $Ax = b$ is approximately

$$\mathbf{P} - \log(\kappa(A)),$$

where $\mathbf{P}$ is the working precision and $\kappa(A)$ is the condition number of A. This is a result of Wilkinson [9] that has been refined in various forms (see [2] and the references therein). Thus, to maintain the desired level of accuracy, it is adequate to increase the working precision.

However, this use of higher precision can be costly. For example, in the Bertini software package [1], the cost of tracking a path in the next level of precision (19 digits) above double precision (16 digits) results in a several-fold increase in run-time. While this is a particular implementation utilizing a particular multiprecision library, the statement necessarily holds in general.

In this article, we propose a heuristic that will sometimes alleviate the need to use higher precision arithmetic when paths are in close proximity. This heuristic is illustrated via the example in §3 and stated formally in §4, but the essence of the method is as follows.

As we vary t throughout $\mathbb{C}$, we get that the solutions of $f(z) = 0$ in $\mathbb{C}^N$ form a ramified cover of $\mathbb{C}$ with finitely many ramification points r_i. In other words, these solutions form d sheets over $\mathbb{C}$ (where d is the number of solutions in a generic fiber), and these sheets will intersect in only finitely many places, called *branchpoints*, which sit above *ramification points*.

About each ramification point r_i, there is a small region in which the Jacobian is ill-conditioned. The standard method of handling ill-conditioning is to walk through these regions, increasing precision as necessary (then decreasing it as soon as possible). Instead, we propose to walk around a small circle about r_i, avoiding the ill-conditioning and taking advantage of the monodromy action in the fiber. This monodromy tracking, described in §4, is initiated at the first signal of the need for higher precision, based on the conditions in the article [2]. The radius is necessarily chosen heuristically, as described in §3.

Such a circle can be parameterized as $r_i + re^{i\theta}$, for some choice of radius r and θ ranging from 0 to 2π. Walking only halfway around the circle ($\theta = 0$ to $\theta = \pi$) could easily lead to errors (§3). We instead propose to walk around the circle until it closes up in the fiber, then track forwards (towards $t = 0$) from all points above $t = r_i - r$ and backwards (towards $t = 1$) from all points above $t = r_i + r$ except from the starting point in the fiber.

As described in §4.2, the result of this method is that we track longer paths but do not necessarily need to use higher precision to track them. In the simple example of §5, we see that this method can save computational time in some settings. However, until a careful implementation is completed, it is difficult to predict what effect this method will have on total path-tracking time for various types of problems.

In the next section, we provide some background on both homotopy continuation and monodromy. Sections 3 and 4 describe our heuristic with an example and in detail, respectively. Finally, we consider our simple example in §5.

2. Homotopy continuation and monodromy

This section introduces the background needed for the remainder of the article. We provide only brief discussions; refer to [4, 8] for more details.

2.1. Homotopy continuation

Given polynomial system $f : \mathbb{C}^N \to \mathbb{C}^N$, sometimes referred to as the *target system*, homotopy continuation is a 3-step process for approximating the isolated solutions of $f(z) = 0$:

1. Choose polynomial system $g : \mathbb{C}^N \to \mathbb{C}^N$, the *start system*, that is in some way related to $f(z)$ but is easy to solve and has only nonsingular isolated solutions.
2. Form the homotopy
$$H(z, t) = (1 - t) \cdot f(z) + \gamma \cdot t \cdot g(z)$$
so that $H(z, 1) = \gamma g(z)$, the solutions of which we can easily compute, and $H(z, 0) = f(z)$.
3. For each solution of $g(z)$, use predictor-corrector methods and endgames to track the solution as t varies discretely and adaptively from 1 to 0.

There are many variations on this idea and many details excluded from this simple description. Thanks to the choice of a random $\gamma \in \mathbb{C}$, the probability of exactly hitting a ramification point is zero. However, the probability of encountering ill-conditioning is not zero.

To combat the previously mentioned loss of accuracy in the neighborhood of a ramification point, the first author and three collaborators developed the concept of adaptive precision path tracking [2, 3]. The details of adaptive precision path tracking are not important for this article. The key is that there are three conditions developed in [2] so that a violation of any of the conditions will trigger an increase in working precision. We propose using precisely these conditions in our algorithm as the trigger to begin a monodromy loop.

2.2. Monodromy

Consider the simple parameterized polynomial system
$$f(z, t) = z^2 - t,$$
where $t \in \mathbb{C}$. Notice that there is a double root at $t = 0$. For $t = 1$, there are clearly two solutions, $z = \pm 1$. Beginning with $z = 1$ in the fiber over $t = 1$, if we walk around the unit circle once, we arrive back at $z = -1$. Another trip around the loop brings us back to $z = 1$. Of course, starting with $z = -1$, we again need to take two trips around the loop to return to $z = -1$ in the fiber over the starting point, $t = 1$.

More generally, one pass around such a circle, which we call a *monodromy loop*, results in a permutation of the points in the fiber above the starting point of the loop. By repeatedly running around the loop, the points in the fiber will eventually return to their original order, at which point we say we have "closed the loop."

It can happen that there are independent cycles with different lengths in this permutation. For example, there could be five points in the fiber that are permuted by each trip about some monodromy loop. They might all be permuted together, in a cycle of length 5, or they could be permuted in two disjoint sets, one of size 2 and one of size 3. For the method of this paper, this will not matter as our input will be a single point in the fiber above the starting point of a monodromy loop, and we will simply walk around the loop repeatedly until it closes.

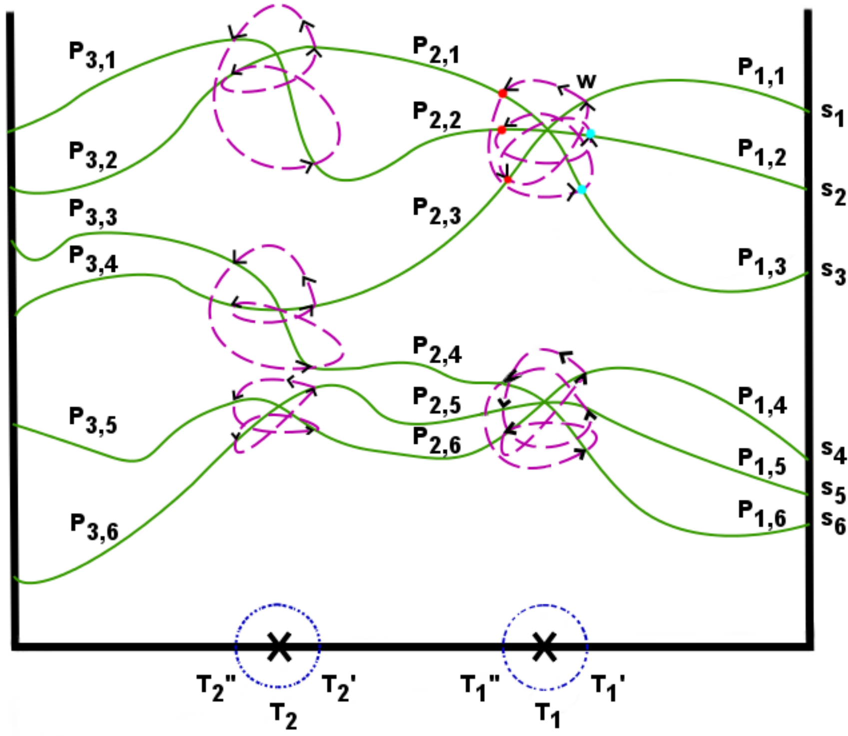

FIGURE 1. Schematic representation of the running example. The green curves are the paths to be tracked. They intersect in two triple points in the fiber above $t = T_1$ and in three double points in the fiber above $t = T_2$. The dotted blue circles are the monodromy loops about the two ramification points. The dashed magenta loops are liftings of the monodromy loops.

3. The heuristic, by example

In this section, we describe the method with an illustrative example, represented in Figure 1. In this case, there are 6 paths to track from $t = 1$ to $t = 0$, which intersect in two triple points in the fiber above $t = T_1$, then in three double points above $t = T_2$. In fact, it is easy to write down such a system:

$$\begin{aligned} f_1(x, y, t) &= (x^3 - 1) \cdot t + (x^3 + 2) \cdot (1 - t) \\ f_2(x, y, t) &= (y^2 - 1) \cdot t + (y^2 + 0.5) \cdot (1 - t) \end{aligned}$$

Of course, this system was contrived for the purposes of this article. In practice, actual path intersections are exceedingly rare and triple (or more) intersections are even more rare. However, the algorithm of this paper applies not only in the situation of actual path intersections but also , much more broadly, in the setting of near path-crossings.

For the moment, we restrict our computational resources to tracking one path segment at a time, on a single processor. The use of multiple processors is described at the end of the next section.

3.1. The first path segment, $P_{1,1}$

Tracking begins with s_1, one of the six startpoints at $t = 1$. Tracking forwards (towards $t = 0$), the point w is reached, in the fiber above $t = T_1'$. This is the first point along the path segment $P_{1,1}$ at which the Jacobian matrix is ill-conditioned. This ill-conditioning is recognized as the breaking of an adaptive precision condition [2], at which point a monodromy loop is initiated. This loop is represented in the figure as a blue dotted circle about $t = T_1$. See §3.3 for ideas on how to choose the radius of the monodromy loop.

We track around this monodromy loop (following along the magenta dashed curve in the figure) until we arrive back at w. In this case, three trips around the loop are needed. Along the way, we collect all points above T_1'', depicted as red dots in the figure, and those points above T_1' other than w, i.e., the blue dots. At this point, tracking for segment $P_{1,1}$ is complete. In particular, the tracking of segment $P_{1,1}$ consisted of:

- Beginning with s_1 and a direction;
- Tracking until the need for adaptive precision was recognized; and
- Tracking around a monodromy loop until it closed up, collecting various bits of information along the way.

The output of this procedure is a set of five new path segments to be tracked, in addition to the other five initial path segments beginning at $s_2, \ldots, s_6$. In particular, we now have three new forward path segments ($P_{2,1}$, $P_{2,2}$, and $P_{2,3}$) beginning at the red dots, and two new backwards path segments ($P_{1,2}$ and $P_{1,3}$), beginning at the blue dots. Obviously, there is no value in tracking backwards along $P_{1,1}$ from w as we have already tracked along that path.

If a path does not encounter ill-conditioning, we could simply track all the way to $t = 0$. More precisely, we could track to $t = 0.1$, at which point an endgame would take over to complete the path. The algorithm we are proposing is only intended to take the paths successfully from $t = 1$ to the beginning of the endgame, after which the endgames inherently manage ramification points [4, 8].

3.2. After the first monodromy loop

In our example, we now have five initial forward path segments, three new forward path segments, and two new backwards path segments. Technically, the processor could choose any of these segments to run next. However, choosing either $P_{1,2}$ or $P_{1,3}$ in the forward direction would result in redundant trips around the monodromy loop just handled (or one very close to it, depending on the value of t at which ill-conditioning is first encountered). Thus, the algorithm calls for handling all backwards path segments before handling forward path segments.

In following $P_{1,2}$ backwards from the blue dot, we eventually reach startpoint s_2 above $t = 1$. Once we reach $t = 1$, we can simply search our list of path segments to track forwards and eliminate s_2. After treating $P_{1,3}$ similarly, we are left with only the three red dot forward paths, $P_{1,4}$, $P_{1,5}$, and $P_{1,6}$.

If $P_{2,3}$ is tracked next, we encounter a monodromy loop about T_2, beginning at point T_2'. This will set up $P_{2,4}$ as a backwards path segment, resulting in a "backwards" trip around the monodromy loop about T_1, beginning with $t = T_1''$. It is trivial to adapt the idea of tracking forwards through a monodromy loop to this setting, so we do not provide details here. Note, though, that this backwards tracking will result in backwards tracking along $P_{1,4}$, $P_{1,5}$, and $P_{1,6}$, killing off the three remaining forward paths. While there is no clear computational benefit to following $P_{2,3}$ before following, say, $P_{1,4}$ forward, it is amusing to note that the result of chossing $P_{2,3}$ would be that we would ultimately track forward from only one of our startpoints (s_1), the other five being eliminated by backwards path segments caused by monodromy loops!

Continuing in this way, it should be clear that we will ultimately track in some direction along each path segment $P_{i,j}$ and track around five monodromy loops (three trips around each of two triple intersections, two trips around each of the other three double intersections). This is opposed to trying to track from $t = 1$ to $t = 0$ on the six green paths. A brief analysis of the cost of carrying out this algorithm is provided in §4.2.

3.3. Choosing the radius of the monodromy loop

When tracking a path segment, say $P_{1,1}$, it is easy to detect the need for a monodromy loop. However, *a priori*, we do not have any knowledge about the actual location of the nearby ramification point. For that matter, it might be that sheets come close but never actually intersect, meaning there is actually no ramification point nearby, though there is ill-conditioning.

There are any number of games that could be played to choose monodromy loops (or other shapes, e.g., diamonds), and a full analysis of these options is beyond the scope of this paper. Such an analysis will be much simpler once there is software for testing out these heuristics, i.e., when the development of Bertini 2.0 is far enough along for a careful implementation.

In the meantime, we propose the following heuristic. Since ill-conditioning has not been encountered along $P_{1,1}$ before reaching w, the step size of the path tracker should be reasonably large. Take that as the radius of the monodromy loop. In particular, if h is the current tracker step size coming in to w, the monodromy loop could be

$$t = (T_1' - h) + he^{i\theta},$$

for $\theta = 0, \ldots, 2\pi$.

Of course, if ill-conditioning was noticed because the path just happened to brush the edge of an ill-conditioning zone off to the side of the path, with a ramification point lying in a direction orthogonal to the direction of the path, this loop would immediately result in ill-conditioning along the monodromy loop, discussed in the next subsection.

3.4. Handling illl-conditioning along the monodromy loop

The assumption is that we are able to track around the monodromy loop without encountering ill-conditioning. Of course, this might not be the case, and the Jacobian might become ill-conditioning at any point along the loop. If that occurs, there are a few options, listed in order of lowest quality to highest:

1. Try to build a new monodromy loop off of the current loop in order to avoid this ill-conditioning. This clearly cascades into a debacle, since we need to track around exactly the same loop for each pass (lest the monodromy groups in the fiber get confused by

including different sets of ramification points within the loop for different passes about the loop). This is clearly a dangerous, albeit not impossible, option.
2. Start over with a different loop, e.g., with a loop of larger or smaller radius. This seems to be a reasonable option, though too many reboots of the monodromy loop will eventually eliminate any value gained by avoiding higher precision.
3. If all else fails, continue to track straight through the zone of ill-conditioning using adaptive precision as necessary. It might be the case that some ramification points are well separated from other ramification points, in which case the monodromy algorithm should work well. Others might be clustered, making adaptive precision the better option.

It seems reasonable to tighten the adaptive precision conditions (by including extra "safety digits," as in [2]) when using these conditions as a trigger for a monodromy loop. Then, by loosening these conditions to a normal level during the loop, there is a better chance that tracking around the loop will succeed. This maneuver essentially tricks the tracker into starting a monodromy loop a bit earlier than with the normal adaptive precision conditions.

As with the radius (and shape) of the monodromy loop, these options will need to be analyzed more carefully once software is readily available.

3.5. The necessity of closing the loop

A similar approach was proposed in 1991 [7], before the algebraic geometry underlying polynomial homotopy continuation was well established. In that approach, the authors suggested walking halfway around monodromy loops (called a two-phase homotopy) to avoid ill-conditioning.

While this approach sounds similar in principle, not closing the loop in the fiber is dangerous. In particular, suppose two paths intersect and that, by the vagaries of adaptive steplength, a monodromy loop is triggered on only one of the paths. Due to the monodromy action about the ramification point, the remainder of the path for which a monodromy loop was not triggered will be followed twice, with the remainder of the other path neglected. By closing the monodromy loop as in our algorithm, we never neglect a path and are certain to follow all segments of all paths.

It is worth noting that a monodromy loop can be triggered even if there is no ramification point, i.e., if the solution sheets come close but do not intersect. The result of this false positive will be a seemingly useless monodromy loop for each of the two paths exhibiting ill-conditioning, each traversed just once. While there is theoretically no problem with this scenario, it may seem a waste of computation time. However, the alternative is to increase precision and push through the ill-conditioned zone, which also results in an increase in computation time. Thus, in this setting, seemingly frivolous monodromy loops are not necessarily more costly than the alternative.

4. The heuristic, formally

In this section, we provide formal statements of our method, followed by a brief analysis of the computational cost of this method in comparison to the current standard method, and a brief discussion of how to adapt this method in the case of multiple processors.

4.1. Formal methods

Let $f : \mathbb{C}^N \to \mathbb{C}^N$ be a polynomial system, $g : \mathbb{C}^N \to \mathbb{C}^N$ a start system for $f(z)$, and $H(z,t)$ the standard homotopy of §2 from $g(z)$ to $f(z)$. A *path segment* is a pair (p, t_{target}) consisting of a point $p \in \mathbb{C}^N$ on a homotopy path and a target value of t, either 0.1 (or some other endgame starting point) or 1, providing a direction.

Our method consists of a main method (`Main`) and two subroutines (`Track` for regular path tracking and `MonoTrack` for tracking around a monodromy loop).

Algorithm 1 `Main`
Input: Homotopy $H(z,t)$ from $g(z)$ to $f(z)$; startpoints s_i for $i = 1, \ldots, k$.
Output: Set E of solutions of $H(z, 0.1) = 0$.

1: Let $P := \{(s_i, 0.1) | 1 \leq i \leq k\}$. ▷ Initialize set P of path segments to track.
2: **while** $\#P > 0$ **do** ▷ Track segments until all paths have reached $t = 0.1$.
3:     Choose a path segment (p, t_{target}) from P, with $t_{target} = 1$, if possible.
4:     `Track`(p,t_{target}). ▷ `Track` updates global list P.
5: **end while**

Algorithm 2 `Track`
Input: Path segment (p, t_{target}).
Output: No output, but list P of path segments to track is updated.

1: Track from p in the direction of t_{target}, using predictor-corrector methods with adaptive steplength, stopping when either $t = t_{target}$ or an adaptive precision condition has been broken. Let w be the current point, t_{curr} the current value of t, and h be the latest step size.
2: **if** $t = t_{target}$ and $t_{target} = 0.1$ **then**
3:     Add w to set E.
4: **else if** $t = t_{target}$ and $t_{target} = 1$ **then**
5:     Remove path segment $(w, 0.1)$ from P.
6: **else if** an adaptive precision condition has been broken **then**
7:     $its := 0$ ▷ its tracks the number of attempted monodromy loops.
8:     **while** $its < 3$ **do** ▷ 3 is arbitrary.
9:         `MonoTrack`(w,t_{curr},t_{target},h,its).
10:         **if** MonoTrack did not report an error **then**
11:             Return. ▷ Successfully tracked around loop until closure.
12:         **else**
13:             $its := its + 1$.
14:         **end if**
15:     **end while**
16:     **if** $its = 3$ **then** ▷ 3 monodromy loops failed.
17:         Continue through the ill-conditioned zone with adaptive precision.
18:     **end if**
19: **else**
20:     Report an error.
21: **end if**

Remark 4.1. The full tracking algorithm is very complicated, so we have opted to refer the reader to the flowcharts of [2] rather than record all of the details here.

Algorithm 3 `MonoTrack`
Input: Point w, t value t_{curr}, t_{target} for this path segment, and latest step size h, its.
Output: No output, but list P of path segments to track is updated.

1: Choose a monodromy loop, using t_{curr}, t_{target}, h, and its. ▷ See §3.3.
2: Let $y := w$. ▷ $y \in \mathbb{C}^N$ denotes the current point at all times.
3: Let $m := 0$. ▷ m is the number of completed monodromy loops.
4: **while** $y \neq w$ or $m = 0$ **do** ▷ $y = w$ is a numerical test.
5: Track from $\theta = 0$ to $\theta = \pi$.
6: Store (y, t_{target}) in P.
7: Track from $\theta = \pi$ to $\theta = 2\pi$.
8: If $y \neq w$, then store $(y, 1.1 - t_{target})$ in P.
9: $m := m + 1$.
10: **end while**
11: If, at any point, ill-conditioning is encountered, return an error. ▷ See §3.4.

Remark 4.2. Rather than tracking around a circle via some parameterization, the standard implementation choice is to inscribe a regular n-gon ($n = 8$ is a common choice) within the circle and follow along the n line segments ($\frac{n}{2}$ for each trip halfway around the circle). Obviously, for our purposes, n must be even.

4.2. Cost Analysis

The best case is that ill-conditioning is never encountered during `MonoTrack`. In this case, we replace each ill-conditioned zone needing adaptive precision with a low precision trip around a monodromy loop. While the loop is clearly longer (the circumference of a circle, compared to the diameter or less), the savings from using lower precision should more than compensate for this additional length. Even more compelling is the fact that the monodromy algorithm will avoid path failures due to excessive precision.[2]

The worst case is that ill-conditioning would be encountered during every call to `MonoTrack`, in which case we would resort to adaptive precision tracking on every path *and* incur the additional cost of starting each monodromy loop. This is, of course, far from ideal and serves to increase the computational time for the run with no benefit.

4.3. Handling multiple processors

Using a single processor, it is easy to see that there will be no path segments accidentally tracked multiple times, in different directions. However, with multiple processors, this could happen.

For example, suppose we attack the example above with six processors. A natural approach would be to give one of the six startpoints to each processor, so that each $P_{1,i}$ is tracked forward. Unfortunately, each of these path segments will trigger a monodromy loop around $t = T_1$, resulting in a great deal of redundancy!

While it is appealing to consider checking whether some given monodromy loop has been considered already, there are two inherent difficulties. First, as is evident in the example above,

[2]Bertini sets a maximum level of precision, lest the memory and disk space needs grow too large. Thus, paths can fail with adaptive precision not only at true path intersections, but also in very small neighborhoods about them.

there can be separate monodromy actions over the same ramification point (or ramification points in very close proximity). While this is not likely at all, it is a concern. Even worse, just because two paths could trigger monodromy loops about the same ramification point does *not* mean that they will necessarily trigger these loops from the same value of t, again due to the vagaries of adaptive step size path tracking. It is useless trying to compare points in the fiber above T_1' when T_1' is path-dependent.

There are three clear approaches for parallelizing this method:

1. If there are many paths and not too many processors, it is reasonable to accept a bit of redundancy. Since paths (and therefore path segments) are typically sent to processors in batches, not one at a time, one implementation choice is whether to send a message each time a startpoint is eliminated by backwards tracking. Depending on the parameters of the system and the processors, this may or may not be worthwhile.
2. Rather than starting all processors on initial path segments $P_{1,i}$, hold some (perhaps most) in reserve, to be used once a monodromy loop has been traversed. This would require significant hierarchical control, but it is not infeasible.

5. Example

Until this method can be fully implemented, it is very difficult to run tests of any significant size. Currently, the only option is to put together *ad hoc* runs with many calls to Bertini, passing information back and forth via hard disk writes.

We ran the example above with and without our monodromy method. Without it, all paths failed after a few seconds. This is not surprising as there are actual path intersections (a probability zero event). Each stage of the monodromy algorithm (`Track` and `MonoTrack`) either took 1-2 milliseconds or did not even register a time.

While this may appear as good news for the monodromy method, the reality is that this is a contrived example for which any path tracker would fail. It would be more compelling to test a very large problem with many variables and many ramification points.

Unfortunately, it will be many months, perhaps years, before we can reasonably expect to implement this method. Bertini is undergoing a complete rewrite, moving from C to C++, among many other improvements. While we could dig deep within the core of the current version of Bertini to implement this, there would be a significant start-up cost that would then be wasted due to the rewrite. Fortunately, the benefits clearly exist; it is just a question of scale.

6. Future work

This monodromy-based method for avoiding higher precision is still very much at the proof-of-concept level. As we re-develop Bertini and incorporate this technique, we will analyze the value of the method as a whole and also consider the variety of options for the many subroutines. Also, this method may or may not be useful for specialized types of homotopies. For example, regeneration [6] and parameter homotopies [5] are both useful techniques relying on homotopy continuation for which this method would probably be useful, though this was not considered in this article.

7. Acknowledgements

This work was partially funded by the National Science Foundation, via grant DMS–1115668.

References

[1] Bates, D.J., Hauenstein, J.D., Sommese, A.J., Wampler, C.W.: *Bertini: Software for Numerical Algebraic Geometry*. Software available at `http://www.bertini.nd.edu`, (2006).

[2] Bates, D.J., Hauenstein, J.D., Sommese, A.J., Wampler, C.W.: Adaptive multiprecision path tracking. *SIAM J. Numerical Analysis* 46, 722–746 (2008).

[3] Bates, D.J., Hauenstein, J.D., Sommese, A.J., Wampler, C.W.: Stepsize control for adaptive multiprecision path tracking. *Contemporary Mathematics* 496, 21–31, (2009).

[4] Bates, D.J., Hauenstein, J.D., Sommese, A.J., Wampler, C.W.: *Numerically Solving Polynomial Systems with Bertini*. SIAM (2013).

[5] Brake, D., Niemerg, M., Bates, D.J.: Paramotopy: parameter homotopies in parallel. Software available at `http://www.paramotopy.com/index.html` (2013).

[6] Hauenstein, J.D., Sommese, A.J., Wampler, C.W.: Regeneration homotopies for solving systems of polynomials. *Mathematics of Computation* 80, 345–377 (2011).

[7] Kalaba R.E., Tesfatsion, L.: Solving nonlinear equations by adaptive homotopy continuation. *Applied Mathematics and Computation* 41(2), 99–115 (1991).

[8] Sommese, A.J., Wampler C.W.: *The Numerical Solution to Systems of Polynomials Arising in Engineering and Science*. World Scientific (2005).

[9] Wilkinson, J.H.: *Rounding errors in algebraic processes*. Dover (1994).

Daniel J. Bates and Matthew Niemerg Department of Mathematics, Colorado State University, Fort Collins, CO USA (bates,niemerg@math.colostate.edu, `www.math.colostate.edu/~bates,~niemerg`).