A Parallel Adaptive Method for Pseudo-arclength Continuation

by

Alexander Dubitski

A thesis submitted in conformity with the requirements
for the degree of Masters of Science
Faculty of Graduate Studies (Modelling And Computational Science)
University of Ontario Institute of Technology

Supervisor(s): Dr. Lennaert van Veen and Dr. Dhavide Aruliah

Abstract

A Parallel Adaptive Method for Pseudo-arclength Continuation

Alexander Dubitski

Masters of Science

Faculty of Graduate Studies

University of Ontario Institute of Technology

2011

We parallelize the pseudo-arclength continuation method for solving nonlinear systems of equations. Pseudo-arclength continuation is a predictor-corrector method where the correction step consists of solving a linear system of algebraic equations. Our algorithm parallelizes adaptive step-length selection and inexact prediction. Prior attempts to parallelize pseudo-arclength continuation are typically based on parallelization of the linear solver which leads to completely solver-dependent software. In contrast, our method is completely independent of the internal solver and therefore applicable to a large domain of problems. Our software is easy to use and does not require the user to have extensive prior experience with High Performance Computing; all the user needs to provide is the implementation of the corrector step. When corrector steps are costly or continuation curves are complicated, we observe up to sixty percent speed up with moderate numbers of processors. We present results for a synthetic problem and a problem in turbulence.

Dedication

This thesis is dedicated to my fiancee and my parents.

Contents

Chapter 1

Introduction

Given a set of equations, under determined by a single one, the solutions generically lie on curves. We can approximate such curves with a discrete set of points. Pseudo-arclength continuation is a numerical method that, given an initial point, returns a set of points that lie on such curves or extremely close to them. It was originally introduced by Keller [1] in the late 1970s. Pseudo-arclength continuation is a modified natural continuation method. The natural continuation method takes fixed steps in one of the unknowns, where pseudo-arclength continuation takes a step in the arc-length along the curve. Pseudo-arclength continuation is a predictor-corrector method. In the prediction step we extrapolate along the tangent to the curve. In the corrector step a new point on the curve is computed, usually by means of the Newton method (also called a Newton-Raphson method). Newton's method requires solving a system of linear equations, which can be highly time consuming for large sets of equations.

Currently the most common software that implements pseudo-arclength continuation is AUTO [2]. AUTO is vastly used in applied mathematics and according to Google Scholar it has over 672 citations at the time of writing. The latest version of AUTO parallelizes pseudo-arclength continuation by parallelizing the linear solver. Unfortunately AUTO cannot be used for problems in which the number of equations exceeds a few hundred. We focus on the case when pseudo-arclength continuation is used for larger problems. Even when we use the most time efficient methods (typically based on a Krylov subspace, see section 2.1.3) to solve matrix-vector problems, the computation of one Newton iteration may require days to complete, which leads to months of computational time. Such problems are often encountered in the study of equilibrium and periodic solutions to the Navier-Stokes equations (see, e. g., [3]).

Pseudo-arclength continuation is inherently sequential, i.e. each step largely depends on a previous step. Problems where each step largely depends on a previous step are often considered non-parallelizable or require clever and non-trivial parallelization techniques. Barney [4] uses the example of the Fibonacci sequence:

"...the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the F(n) value uses those of both F(n-1) and F(n-2). These three terms cannot be calculated independently and therefore, not in parallel."

Even though pseudo-arclength continuation falls under the category of inherently sequential problems such as the Fibonacci sequence, we can take advantage of the fact that pseudo-arclength continuation involves a step-size selection. The step-size selection is a non-trivial optimization problem. With a small step size we need fewer Newton iterations but must compute more points on the curve, while with a large step we need to compute fewer points on the curve, but we need more Newton iterations and some Newton iterations might diverge. Such failing steps are computationally expensive so we want to eliminate most of them.

We present in this thesis a new, high level algorithm. It does not depend on the choice of the linear solver for the Newton corrector steps and the user needs only a minimal knowledge of High Performance Computing to parallelize his problem with the software that we provide. In our approach, we try many step sizes in parallel and extrapolate on intermediate results to compute a few continuation points simultaneously on different CPUs. We have performed numerical experiments that cover many possible curve structures where the performance improvement is approximately 60%. In studying certain large-scale problems, e.g. in fluid mechanics [3], computing the continuation curve for a given problem could take a year to complete. For such a problem, our proposed algorithm could allow users to get their results seven months earlier. Our method is independent of the internal solver and can be combined with different methods to perform a correction step which often can also be parallelized to achieve even greater speed-ups. For instance we can use methods such as Newton-Raphson, fixed-point iteration or optimization based methods such as Simulated annealing and Levenberg-Marquardt. The only change required to our algorithm is to determine conditions for the coloring scheme that we explain in chapter 3.

1.1 Thesis Outline

To increase readability of our work, we describe it in three main parts. In Chapter 2 we provide a mathematical formulation of pseudo-arclength continuation and show briefly its derivation from the multivariate Taylor series. We also explain the use of direct vs. inexact linear solvers for pseudo-arclength continuation. In the Chapter 3, we focus entirely on the parallelization algorithm. Lastly we show results from numerical experiments that we have performed to provide a performance estimate of our algorithm and software in Chapter 4.

1.2 Notations

Based on our parallelization method we need to run many corrector steps in parallel. Each of those corrector steps we represent by a node, as explained in Chapter 3. For clarity, we introduce the following notation:

- $z_j \in \mathbb{R}^N$ represents a point on the continuation curve Γ, where j is the index of a particular point, most often referred to the last known point on Γ.
- T represents a tangent direction used in pseudo-arclength continuation .
- $\alpha_n^{(\nu)}$ corresponds to a node in the tree path or a point on a pipeline. Here the lower index represents the depth of the point from z_j and ν represents the number of Newton iterations that have been performed on that node.
- h_{α_n} represents the step size with which α_n was spawned.
- k represents the number of spawned child nodes.
- t_i represents a factor by which to multiply $h_{\alpha_n}, i \in [1, k]$.
- τ is an actual wall-clock time to compute each Newton step.
- $w_n^{(\nu)}$ represents a child node of α_n or a forked point from z_j.
- U represents a set of nodes that form a path in the tree
- W_n represents the set of at most k child nodes of α_n
- $U_{\scriptscriptstyle G}$ represents a path U that consists entirely of green nodes.
- $U_{\scriptscriptstyle Y}$ represents a path U that consists entirely of green or yellow nodes.
- $L_{\scriptscriptstyle G} = \sum\limits_{i=m+1}^{n} h_{\alpha_i}$, where $\alpha_i \in U_{\scriptscriptstyle G}$, is the length of the green path.
- $L_{\scriptscriptstyle Y} = \sum\limits_{i=m+1}^{n} h_{\alpha_i}$, where $\alpha_i \in U_{\scriptscriptstyle Y}$, is the length of the yellow path.
- $G_j(x) = 0, j \in [1, N], x \in \mathbb{R}^N$, is the set of non-linear equations.
- $J = \dfrac{\partial G}{\partial x}$, is the Jacobian matrix.
- $G(x) = O(g(x))$ at $x_0 \Leftrightarrow \|G(x)\| < C|g(x)|$ for some $C \in \mathbb{R}^+$ on an open neighborhood Ω of x_0.

Chapter 2

Mathematical Background

2.1 Solving Nonlinear Systems of Equations

A system of equations is linear if it satisfies the superposition principle, which states that, if x and y are solutions, then so is any linear combination $ax + by$. Any system of equations that does not have this property is called nonlinear.

2.1.1 Newton's method

Let a nonlinear system of equations be given by

$$G_j(x) = 0, j \in [1, N], x \in \mathbb{R}^N \tag{2.1}$$

A common way to solve this kind of problem is to choose a predicted point $x^{(0)}$ and approximate solution with Newton's method:

$$x^{(\nu+1)} = x^{(\nu)} - [J(x^{(\nu)})]^{-1} G(x^{(\nu)}) \quad (\nu = 0, 1, 2, \dots) \tag{2.2}$$

where J is the Jacobian matrix associated with the vector field G as defined in section 1.2. Newton's method can be derived using a truncated Taylor series expansion. That is, to determine a zero x of the nonlinear system of equations $G(x) = 0$, assume that we have a putative zero, say, $x^{(0)} \in \mathbb{R}^N$. Then, a multivariate Taylor series expansion centred at $x^{(0)}$ would be written as

$$G(x) = G(x^{(0)}) + J(x^{(0)})(x - x^{(0)}) + O(\|x - x^{(0)}\|_2^2), \tag{2.3}$$

where the truncation error has the asymptotic behavior as described in the last term (see section 1.2). Newton's method, then, consists of ignoring the higher order terms in (2.3). That is, we consider the linear model

$$G(x) \approx G(x^{(0)}) + J(x^{(0)})(x - x^{(0)}). \tag{2.4}$$

If x is in fact a zero of G, then $G(x) = 0$. Assuming that (2.4) models G well near the zero x, we solve for x in (2.4), obtaining

$$x = x^{(0)} - [J(x^{(0)})]^{-1} G(x^{(0)}). \tag{2.5}$$

The computation in (2.5) suggests an iteration we can apply to yield a sequence that may converge to an actual zero of the vector field G, given by 2.2.

Theory predicts that if the sequence $x^{(\nu)}$ converges to x, then

$$\| G(x^{(\nu)}) \|_2 = C \| G(x^{(\nu-1)}) \|_2^2 \text{ for sufficiently large } \nu \text{ and some } C \in \mathbb{R}^+ \tag{2.6}$$

The obvious case when this method fails to work is when the Jacobian $J(x^{(\nu)})$ is singular.

2.1.2 Direct Newton solvers

Newton solvers that rely on the direct solution of the linear system of equations for each Newton step are referred to as direct Newton solvers. Direct solvers for linear systems of equations are solvers that are guaranteed to terminate in a predetermined number of steps. The most costly part of the computation in Newton's method is the computation of the inverse of the Jacobian matrix $[J(x^{(\nu)})]^{-1}$. In practice, the inverse of the Jacobian is usually not computed explicitly. Instead, the LU factors L, U, P are constructed such that:

$$P^{(\nu)}J(x^{(\nu)}) = L^{(\nu)}U^{(\nu)} \tag{2.7}$$

where $P^{(\nu)} \in \mathbb{R}^{N\times N}$ is a permutation matrix, $L^{(\nu)} \in \mathbb{R}^{N\times N}$ is a unit lower triangular matrix, and $U^{(\nu)} \in \mathbb{R}^{N\times N}$ is an upper triangular matrix. It is well-known that construction of the LU decomposition of an invertible matrix is equivalent to Gaussian elimination.

Elementary counting arguments show that the asymptotic complexity of LU decomposition as measured in flops ("floating-point operations") is $O(N^3)$, where the size of the matrix is $N \times N$. If $N = 10^5$ then the number of flops needed to perform an LU is at least 10^{15}. Assume that each elementary floating-point operation can be performed on a modern processor in a single CPU clock cycle (this is a simplification as most arithmetic operations require a few clock cycles). Assume further that a modern CPU has roughly $10GHz$. Under these circumstances, given a matrix of dimensions $10^5 \times 10^5$, the computation of the LU decomposition would take on the order of 10^5 seconds, which is over 27 hours.

2.1.3 Inexact Newton methods

The most time consuming part of solving nonlinear systems of equations using Newton's method is the computation of the Newton step $\Delta x^{(\nu)}$ at each iteration, i.e., the solution of the linear system of equations

$$J(x^{(\nu)})\Delta x^{(\nu)} = -G(x^{(\nu)}).$$

As we have seen earlier, direct solution of an $N \times N$ linear system of equations is not feasible for direct Newton solver when N is too large2.1.2. This problem was recognized even before the era of digital computers when computations were done by hand.

As an alternative to direct solvers, *iterative linear solvers* can be used when direct methods are infeasible. The literature on iterative solvers for linear systems of equations extends back earlier than Gauss. Classical examples of iterative solvers for linear systems of equations include Jacobi's method, the Gauss-Seidel method, and SOR (Successive Over-Relaxation) [6]. In the mid-20th century, modern iterative methods for linear systems of equations were derived using Krylov subspaces [7], including CG (Conjugate Gradients) for Hermitian positive definite coefficient matrices and GMRES (Generalized Minimal RESidual) for general non Hermitian problems. For a comprehensive survey of both classical stationary iterative methods and Krylov subspace methods for solving linear systems of equations by iteration, see [7].

Inexact Newton methods or *Newton iterative methods*, then, are methods that rely on an inner iteration to solve (2.9) for the Newton step $\Delta x^{(\nu)}$ within an outer Newton iteration. The outer iteration uses the computed Newton step to carry out the update

$$x^{(\nu+1)} = x^{(\nu)} + \Delta x^{(\nu)}$$

after computing the Newton step using an iterative linear solver. Inexact Newton methods sometimes bear the name of the inner iterative solver, e.g., Newton-SOR, Newton-CG, Newton-GMRES, etc. Inexact Newton methods based on modern Krylov subspace methods are collectively called Newton-Krylov methods. Kelley provides a good overview of theoretical and practical aspects of inexact Newton methods in [5]. The principal advantage of using inexact Newton solvers rather than direct Newton solvers is that it is generally possible to apply inexact methods to much larger systems of nonlinear equations than can be solved using direct Newton solvers. It turns out that many large, sparse linear systems of equations that arise in practice can be solved efficiently by iterative methods which in turn implies to the efficacy of iterative Newton methods.

2.2 Numerical continuation problems

2.2.1 Families of nonlinear equations with a single parameter

To approximate a curve $\Gamma = \{x, \lambda \in \mathbb{R}^{N-1} \times \mathbb{R} \mid F(x, \lambda) = 0\}$ given an initial point $z_0 \in \Gamma$, we choose a predicted point $z_1^{(0)} = (x_1^{(0)}, \lambda_1^{(0)}), z_1^{(0)} \in \mathbb{R}^N$ and approximate $F(z_1^{(0)})$ with a multivariate Taylor series at that point with a first order approximation.

$$F(z_1^{(1)}) = F(z_1^{(0)}) + DF(z_1^{(1)} - z_1^{(0)}) \tag{2.8}$$

The system 2.8 is under-determined since $F \in \mathbb{R}^{N-1}$. To create a full rank system we fix a hyperplane $\Pi(z) = 0$ through $z_1^{(0)}$ orthogonal to the tangent at z_0 and expand it up to the first order with Taylor series.

$$\Pi(z_1^{(1)}) = \Pi(z_1^{(0)}) + D\Pi(z_1^{(0)})(z_1^{(1)} - z_1^{(0)}) \tag{2.9}$$

Then we make the first iteration by looking for $z = z_1^{(1)}$ where the function $F(z) = 0$ and the hyperplane $\Pi(z) = 0$ intersect as shown in Figure 2.1:

$$\begin{bmatrix} F(z_1^{(0)}) \\ \Pi(z_1^{(0)}) \end{bmatrix} + \begin{bmatrix} DF(z_1^{(0)}) \\ D\Pi(z_1^{(0)})) \end{bmatrix} (z_1^{(1)} - z_1^{(0)}) = 0 \tag{2.10}$$

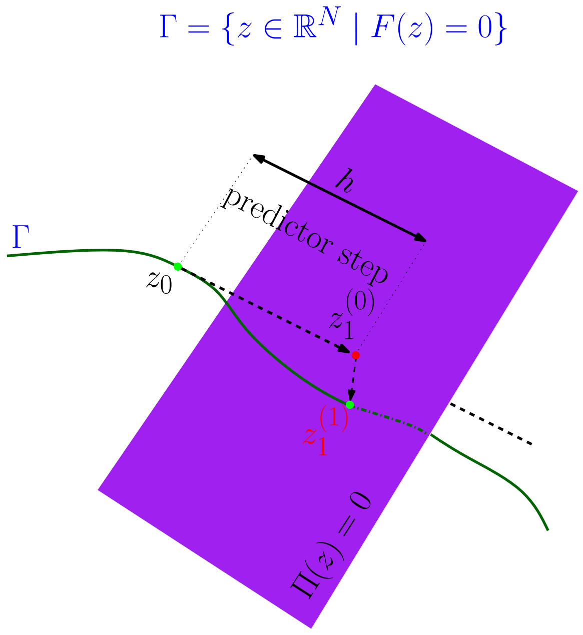

Figure 2.1: The geometry of a pseudo-arclength continuation.

Let $G(z) = \left[F^T(z), \Pi(z)\right]^T$ and $J(z) = DG(z)$ be the Jacobian of $G(z)$.
Then we get a typical Newton method

$$G(z_1^{(0)}) + J(z_1^{(0)})(z_1^{(1)} - z_1^{(0)}) = 0 \tag{2.11}$$

Two obvious cases when this method fails to work are:

1. When $J(z_1^{(0)})$ is singular, meaning that we fixed our hyperplane $\Pi(z_1^{(0)})$ parallel to the tangent direction, which can be found by computing the null space of $DF(z_1^{(0)})$. Typically, this happens when the radius of curvature is small, so even though we extrapolate in the tangent direction from z_0, the hyperplane $\Pi(z_1^{(0)})$ ends up in parallel to the tangent at z_1.

2. The hyperplane $\Pi(z)$ that fixed through $z_1^{(0)}$ does not intersect the curve $F(z)$. To be more precise $\nexists z_1^{(1)} \in \mathbb{R}^n$ such that $G(z_1^{(1)}) = 0$. This usually happens when we take a too large step, for instance when $J(z_1^{(0)})$ is singular.

2.2.2 Methods for numerical continuation

Two common methods for numerical continuation are natural continuation and pseudo-arclength continuation. They differ in the choice of the hyperplane specified by $\Pi(z)$. Below we provide a pseudo code for each of these methods.

Natural continuation

In the natural continuation method, given an initial point $(x_0, \lambda_0) \in \Gamma$, we take a prediction $(\tilde{x}, \tilde{\lambda}) = (x_0, \lambda_0 + h)$. Then we fix a hyperplane Π through the point $(\tilde{x}, \tilde{\lambda})$ such that the normal vector of the hyperplane is $(0, 1)$. Then we perform a number of Newton corrector steps and converge to the intersection of the curve and the hyperplane, (x_1, λ_1). Then we make a new prediction $(\tilde{x}, \tilde{\lambda}) = (x_1, \lambda_1 + h)$ and fix a new hyperplane through the point $(\tilde{x}, \tilde{\lambda})$. Again we perform a number of Newton steps to converge to the intersection of the curve and the hyperplane. We repeat predictor-corrector steps unless a stopping criteria is reached. One example of stopping a criteria is to loop until $(\lambda > \lambda_{end})$.

The weakness of the natural continuation that it fails to work at turning points, i.e. points on the curve where $dx/d\lambda = \infty$.

We can summarize it as follows:

```
**NaturalContinuation** $\{(x_0, \lambda_0) \in \Gamma, \lambda_{end}, h\}$
    $(x_0, \lambda_0) \to (x, \lambda)$
    **while** $(\lambda < \lambda_{end})$ **do**
        $(x, \lambda + h) \to (\tilde{x}, \tilde{\lambda})$
        *Solve* $G(\tilde{x}, \tilde{\lambda}) = 0 \to (x, \lambda) \in \Gamma$
        Adjust $h$ as explained in 2.2.3
    **end loop**
**end**
```

Pseudo-Arclength continuation

The pseudo-arclength continuation method is very similar to the natural continuation method except that we also approximate x to extrapolate in the direction tangent to the solution curve. Given an initial point $(x_0, \lambda_0) \in \Gamma$, we take a prediction $(\tilde{x}, \tilde{\lambda}) = (x_0, \lambda_0) + Th$, where T is the unit tangent to Γ at (x_0, λ_0) and fix a hyperplane $\Pi(\tilde{x}, \tilde{\lambda})$ such that T is the normal vector to this hyperplane. Then we perform a number of Newton corrector steps to converge to the intersection of the curve and the hyperplane. We repeat this procedure for example until $\lambda > \lambda_{end}$.

We can summarize it as follows:

```
**PseudoArclengthContinuation** $\{(x_0, \lambda_0) \in \Gamma, \lambda_{end}, h\}$
    $(x_0, \lambda_0) \to (x, \lambda)$
    **while** $(\lambda < \lambda_{end})$ **do**
      $T_{(x,\lambda)}\Gamma \to T$
      $(x,\lambda) + Th \to (\tilde{x},\tilde{\lambda})$
      *Solve* $G(\tilde{x},\tilde{\lambda}) = 0 \to (x,\lambda) \in \Gamma$
      Adjust $h$ as explained in 2.2.3
    **end loop**
  **end**
```

Here, $T_{(x,\lambda)}\Gamma$ denotes the unit tangent to Γ at (x,λ).

2.2.3 Adaptivity and step-length selection

Both pseudo-arclength continuation and natural continuation sometimes lead to inconsistent nonlinear systems and consequently to non-converging iterations. Usually this is because the Newton method is supposed to converge to the intersection of the hyperplane $\Pi(z)$ and the parametric curve Γ. However, if we fix the hyperplane such that it does not intersect with the curve then the Newton iteration diverges. Another reason for the Newton method to fail is if $J(z_i)$ is singular. One situation for $J(z_i)$ to be singular is if the normal vector of the hyperplane $\Pi(z_i)$ is orthogonal to the tangent T at $\Gamma(z_i)$. This situation is very uncommon to pseudo-arclength continuation because we aim to extrapolate along the curve, however can happen if the natural continuation method is used as we always extrapolate in λ and so it may fail if the radius of curvature is small. We also need to remember that Newton's method works only if the initial prediction is close enough to the curve Γ and if we take a prediction that is too far from Γ then the Newton method fails.

This typically happens when we choose the step size h too large. The choice of the step size is heuristic and it is usually based on the previous step size and the estimation of local curvature of the curve. If the radius of curvature is large then we can take larger prediction as the curve more resembles a straight line, while if the radius of convergence is small, we reduce the step size.

Chapter 3

Parallelization Algorithm

3.1 Introduction to parallelization

For large problems as ours each Newton step is very time consuming and each continuation step must be very small leading to months of computational time. One could parallelize the computation of each Newton-Raphson iteration however this would lead to completely solver dependent software.

In contrast, our approach is completely independent of the method that we use to perform a corrector step and it does not require extensive experience with High Performance Computing . Also, if someone parallelizes the corrector step she can still use our method and software to achieve an even a better speed up.

As mention in Chapter 2, pseudo-arclength continuation consists of an outer iterator and corrector steps. If we do not parallelize the corrector step then we can only parallelize the outer iterator of the pseudo-arclength continuation . From the first glance the outer iterator of the pseudo-arclength continuation may seem inherently sequential, however, when we look closer we can use two techniques that were not obvious at first. Basically we can still execute multiple corrector steps concurrently. Let us agree that when we say concurrent corrector/Newton step we refer to execution of multiple corrector steps in parallel that are synchronized before and after each execution.

3.2 Minimizing the number of concurrent Newton steps

When Newton steps are time consuming we aim to minimize their number. This is done by:

- Extrapolating through intermediate corrector steps to obtain new predictions which we can execute simultaneously.
- Attempting predictor steps with varying step-lengths simultaneously.
- Monitoring concurrent corrector steps, ignoring diverged Newton steps and choosing predictors that converge in an optimal time.

3.3 Concurrent Computation (Pipeline Technique)

Let z_j be a point on the curve. Then make a prediction step $\alpha_1^{(0)} = z_j + th_{z_j} T_0$, see Figure 3.1. Here, $T_0 = \frac{z_j - z_{j-1}}{\|z_j - z_{j-1}\|}$ is an estimation of the unit tangent to Γ.

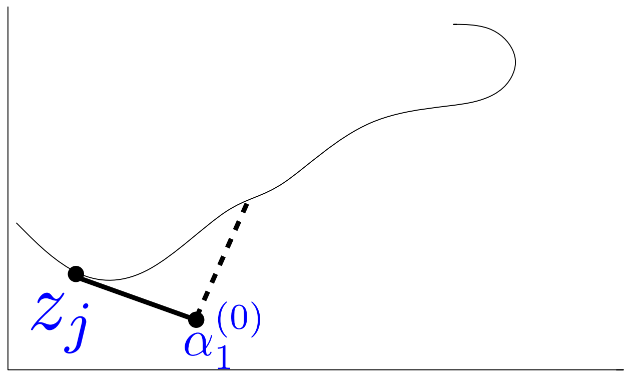

Figure 3.1: Pseudo-arclength continuation Prediction.

Then we perform one Newton step: $\alpha_1^{(\nu+1)} = \alpha_1^{(\nu)} + \Delta\alpha_1^{(\nu)}$, see Figure 3.2.

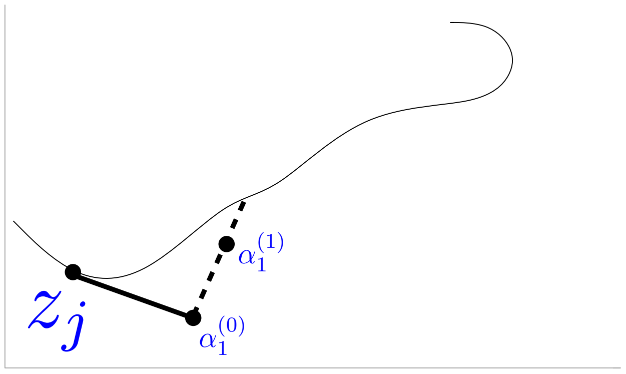

Figure 3.2: After the first iteration.

In serial pseudo-arclength continuation we wait for the sequence $\alpha_1^{(i)}$ to converge to z_{j+1} before we compute z_{j+2}. However, when a point is close to Γ, Newton steps tend to converge quadratically. In practice this holds very well with little adjustments.

Therefore even after the first Newton step (when $\nu = 1$) $\alpha_1^{(\nu)}$ is sufficiently close to the curve to extrapolate again even though it has not converged ($\| G(\alpha_1^{(\nu)}) \| > Tol$), see Figure 3.3.

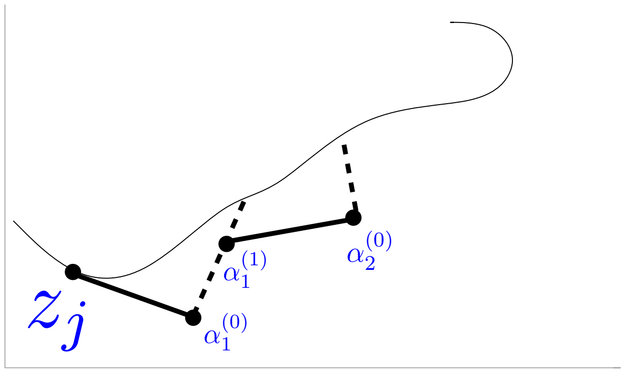

Figure 3.3: Extrapolation through intermediate results.

We extrapolate to $\alpha_2^{(0)}$ on intermediate results with a new tangent parallel to $T_1 = \alpha_1^{(1)} - z_j$. Then we compute the Newton corrector step for $\alpha_1^{(1)}$ and $\alpha_2^{(0)}$ in parallel such that the former is assigned to process p_1 and the latter to process p_2, see Figure 3.4.

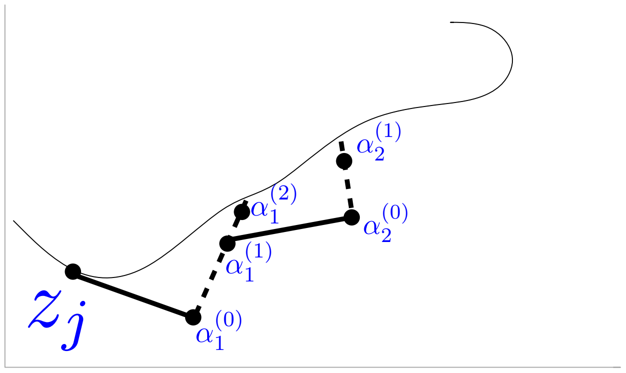

Figure 3.4: Pseudo-arclength continuation -Parallel corrector step on points $\alpha_1^{(1)}$ and $\alpha_2^{(0)}$.

The obvious question is: how many times can we extrapolate on intermediate results? The answer is that there is no limit as long as none of the Newton corrector iterations diverge. In case one of the intermediate iterations fails to converge, then we neglect points that we have extrapolated from that point.

In actuality we have only a limited number of processes available so we cannot assign a new process to each predicted point. Thus our algorithm must reassign processes that have completed their jobs and make sure that each predicted point has a unique process assigned to it before execution of each concurrent Newton step.

This requires to coordinate the mapping between processes and predicted points so we need to keep track of every point and the process attached to it, and then move that process to a newly extrapolated point. However, a new questions arises: what do we do if there is no point yet available or one of the points diverges? When a sequence diverges we also need to detach and erase all subsequent predictions and reuse those processors for different points. Coordination of such a mapping between processors and predicted points leads to a complex algorithm that is not easy to implement.

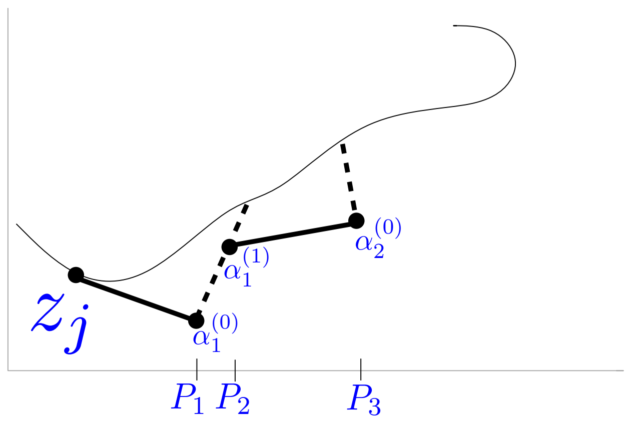

Figure 3.5: Pseudo-arclength continuation -Processes $1..n$ are mapped to points $\alpha_1^{(\nu_1)}...\alpha_n^{(\nu_n)}$.

Fortunately in practice each sequence converges in five or fewer Newton steps or quickly diverges (see Chapter 2); so we monitor the decrease of the residual and if it does not meet the minimum progress then we omit that point and extrapolate again with a smaller step. This means that our sequence is of at most five concurrent points and therefore we need at most five processes.

Then instead of moving processes along the curve, we came up with a notion of conveyor. Imagine that we fix the processes in Figure 3.5 and move the curve along the processes as on a production line. So instead of letting one process execute a sequence of Newton steps until it converges, now the process p_i may pass the predicted point to p_{i-1} after a certain number of Newton steps. For example when a sequence $\alpha_1^{(\nu_1)}$ converges to z_{j+1}, we detach it from p_1 and attach $\alpha_2^{(\nu_2)}$ to p_1 (which was previously attached to p_2). Then we attach $\alpha_3^{(\nu_3)}$ to p_2 and so forth.

In other words when the first point converges the first process starts to work on the second point, then it switches to the third point and then to the fourth point and so on. However, the point that was assigned to p_1 has often performed a number of corrector steps so instead of working on $\alpha_i^{(0)}$ it works on $\alpha_i^{(\nu_i)}$. Each process is idle unless assigned work except for the first process. Obviously the first process is always busy, then the second process only works when data is extrapolated from the first process while other processes are rarely executed.

We can summarize it as follows:

(A) Perform a parallel corrector step on points $\alpha_i^{(\nu_i)}, i \in [1, n]$ that have not converged.

(B) Extrapolate from $\alpha_n^{(\nu_n)}$ (where n is the last active processor in the pipeline) using the secant direction $T_n = (\alpha_{n-1}^{(\nu_{n-1})}, \alpha_n^{(\nu_n)})$ to obtain a new prediction $\alpha_{n+1}^{(0)}$ in the hyperplane Π (see Chapter 2) .

Again, corrector steps are computed in the new hyperplanes on each processor.

(C) Check for convergence of $\alpha_1^{(\nu_1)}$ and if converged then map points $\alpha_i^{(\nu_i)}, i \in [2, n+1]$ to $p_i, i \in [1, n]$.

(D) Perform a parallel corrector step (A).

3.4 Parallel step-size selection (Fork Technique)

Another technique to minimize the number of concurrent Newton corrector steps is to try a number of Newton steps with different steps sizes $t_i hT$ in parallel and choose an optimal step size, see Figure 3.6.

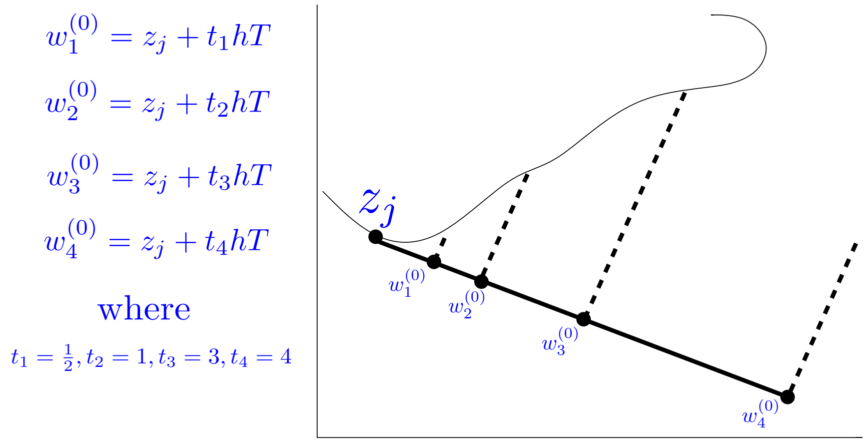
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.6: Example distribution of step sizes to try in parallel.

The advantage would be to save a significant amount of time on Newton steps that fail to converge and to try a bigger $t_i h$. Failures usually arise from large curvature in combination with large step-size so the predicted point appears too far from the curve. At the same time we wish to choose $t_i h$ as big as possible in order to minimize number of concurrent Newton corrector steps. The conventional procedure of choosing the step size is heuristic and is mainly based on the previous step size and the estimate of the local curvature (see Chapter 2.2.3). Thus usually step sizes are not as large as they can be or lead to diverged Newton-Raphson iterations as they are too big.

Let h be the step size at z_j so we make predictions with step sizes $t_i h, i \in [1, k]$. Typically the step size $t_i h$ corresponds to the distance that we "travel" on the curve from z_j to z_{j+1}. So the larger a distance we travel, the fewer number of continuation points it takes to arrive to the destination. However, different points take different number of Newton steps to converge to Γ.

Let us assume the following:

- Let $\nu^{(i)}$ be the number of Newton steps after which the sequence with initial point $w_i^{(0)}$ converges to Γ.
- Each Newton corrector step takes the same time τ.
- We neglect communication and synchronization times.

Based on the above assumptions; $\nu^{(i)}\tau$ is the wall-clock time required to $w_i^{(0)}$ to converge to Γ

Then we introduce the speed $S_i = \dfrac{t_i h}{\nu^{(i)}\tau}$

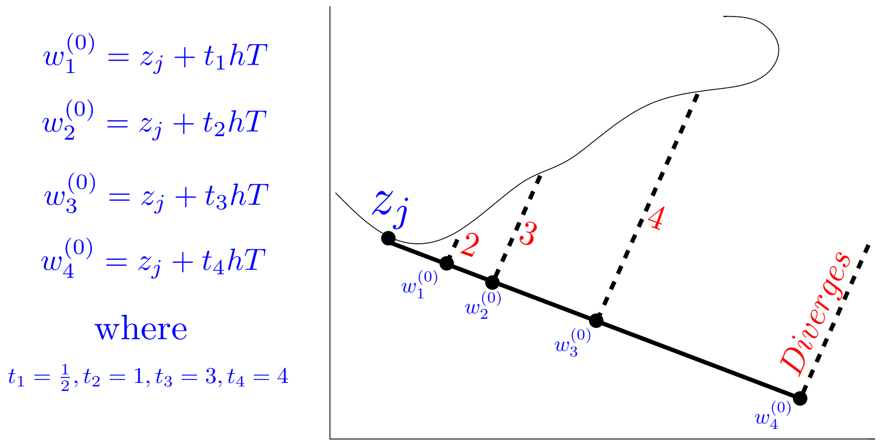
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
2
3
4
Diverges
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.7: The picture shows number of corrector steps for each prediction ratio t_i.

Figure 3.7 shows an example with predictions $w_1^{(0)}, w_2^{(0)}, w_3^{(0)}, w_4^{(0)}$ with associated numbers of Newton corrector steps within which predicted points converge. We can see that $w_1^{(0)}$ converges after $\nu^{(1)} = 2$ iterations, $w_2^{(0)}$ with $\nu^{(2)} = 3$, $w_3^{(0)}$ with $\nu^{(3)} = 4$ and $w_4^{(0)}$ diverges.

The question is: which of those points should we choose? As we are only concerned about the wall time to compute the entire curve, we wish to move as fast as possible so we would like to choose w_i with $\max(S_i), i \in [1, k]$ to maximize the speed (obviously we neglect all diverged points), where k is the number of predictors with different step lengths $t_i h$.

So we compute: $S_1 = \frac{\frac{1}{2}h}{2\tau}, S_2 = \frac{h}{3\tau}, S_3 = \frac{2h}{4\tau}$. Since $S_1 < S_2 < S_3$ we would like to choose w_3, however we do not know ahead in how many corrector steps each point is going to converge, see Figure 3.8.

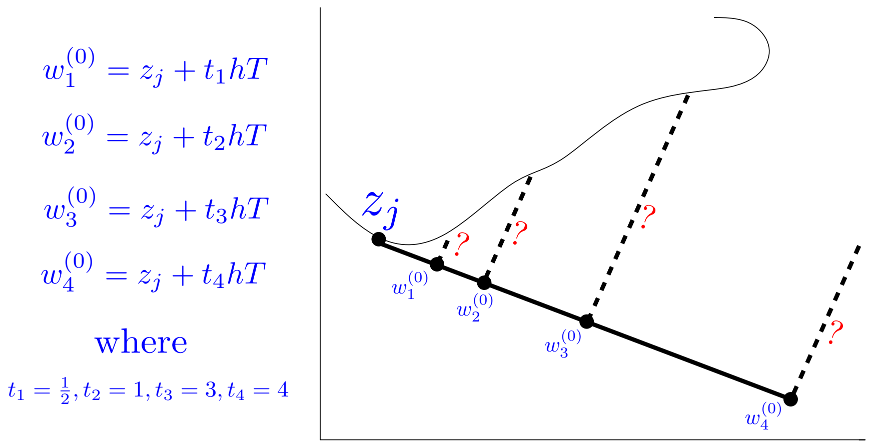
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
?
?
?
?
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.8: In reality we do not know in how many Newton corrector steps w_i^0 will converge.

Fortunately we know that Newton steps converge approximately quadratically when the point is close to the curve Γ. So we can determine whether it converges on the next iteration by checking the residual:

$$\text{if } \log(Res(w_i^{(\nu)})) < \frac{\log(Res_{Tol})}{2} \text{ then } z_{j+1} = w_i^{(\nu+1)}$$

Based on the residual criteria we can divide all sequences into four types:

1. Sequences that have converged.
2. Sequences that are going to converge on the next iteration.
3. Sequences that are still in progress and their future is unknown.

4. Sequences that have diverged.

Instead of referring to the residual of a point and corresponding criteria let us assign colors to each point type:

- ● Green The point has converged which means it is on the curve.

  $Res(w_i^{(\nu)}) < Res_{Tol}$

- ● Yellow The point close to the curve and should converge on the next iteration

  $Res(w_i^{(\nu)}) < \dfrac{\log(Res_{Tol})}{2}$

  Note that there is no guarantee that the Yellow point will converge on the next iteration. However, this almost always holds in the numerical experiments described below.

- ● Red The point is in progress and there is no information about its future.

- ● Black The point has diverged or did not show a significant progress.

  This completely depends on the way a significant progress is defined.

As an example to illustrate our coloring scheme (Figure 3.9) consider the situation where $t_1 = \frac{1}{2}, t_2 = 1, t_3 = 2, t_4 = 4$. We start from the first point on the curve (i.e. it is green by our coloring mechanism). Then we extrapolate $w_1^{(0)}, w_2^{(0)}, w_3^{(0)}, w_4^{(0)}$ with step sizes $t_i h, i \in [1, 4]$. Upper indices of newly extrapolated points $w_i^{(0)}, i \in [1, 4]$, are zero as they have not performed any corrector steps and they are all red since there is no information about their future.

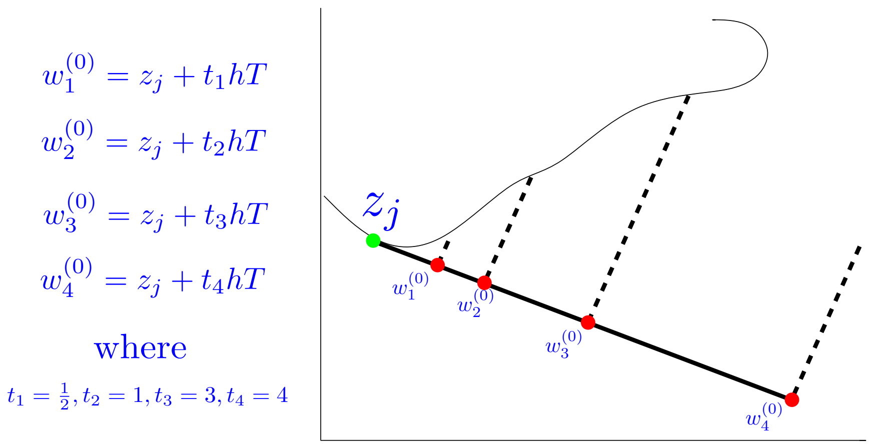
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.9: Extrapolated points are Red since there is no information regarding their future.

Then we perform one concurrent corrector step based on each prediction (see Figure 3.10).

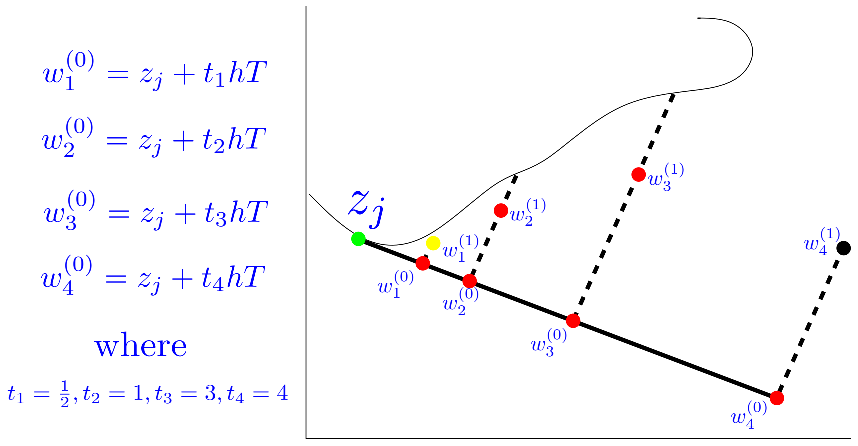
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
$w_1^{(1)}$
$w_2^{(1)}$
$w_3^{(1)}$
$w_4^{(1)}$
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.10: After one concurrent Newton step.

After performing one concurrent Newton step, we compute the residuals and color the iterates appropriately. The first predicted point $w_1^{(1)}$ becomes yellow which means that it is going to converge on the next iteration and $w_4^{(1)}$ has diverged so we paint it black. Since there is still no information about $w_2^{(1)}$ and $w_3^{(1)}$ they are both red. First we remove black point $w_4^{(1)}$ and ignore data that comes from it and then perform a second concurrent Newton step on points $w_1^{(1)}, w_2^{(1)}, w_3^{(1)}$ and again wait for the green point to appear.

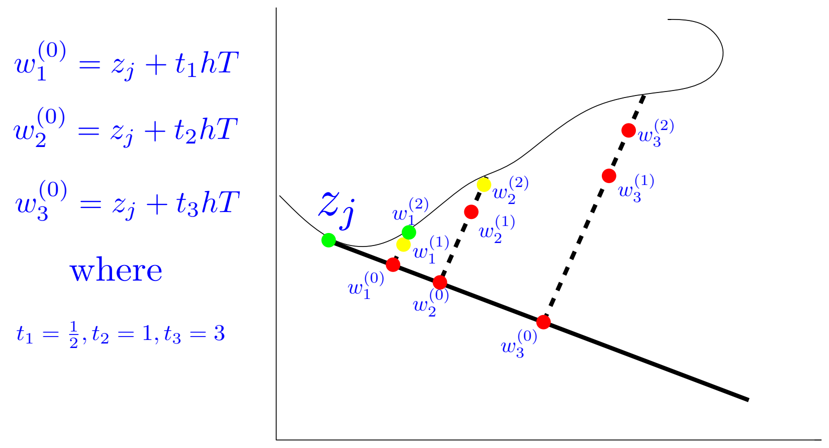
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3$
z_j
$w_1^{(0)}$
$w_1^{(1)}$
$w_1^{(2)}$
$w_2^{(0)}$
$w_2^{(1)}$
$w_2^{(2)}$
$w_3^{(0)}$
$w_3^{(1)}$
$w_3^{(2)}$

Figure 3.11: After two concurrent Newton steps.

After two concurrent Newton steps the sequence $\{w_1^{(0)}, w_1^{(1)}, w_1^{(2)}\}$ has converged and therefore the point $w_1^{(2)}$ is colored green (Figure 3.11) while $w_2^{(2)}$ becomes yellow and $w_3^{(2)}$ remains red.

Now the question is whether to stop and consider $z_{j+1} = w_1^{(2)}$ or to wait for other points to converge. If we had only green and red points then we would have chosen $w_1^{(2)}$ as we do not know anything about red points. However, in this case we have also a yellow point which we expect to converge on the next iteration. So we perform the following comparison:

- ν : number of iterations that green and yellow points have performed.
- $\nu + 1$: is the number of iterations after which we expect yellow points to converge.
- Let $w_G^{(\nu)}, w_Y^{(\nu)}$ be green and yellow points with largest t_i, where t_G is associated with $w_G^{(\nu)}$ and t_Y with $w_Y^{(\nu)}$
- if $\dfrac{t_G h}{\nu} < \dfrac{t_Y h}{\nu + 1}$ then continue

Since $\nu = 2$, $\frac{t_G h}{\nu} = \frac{1}{2}h$, $\frac{t_Y h}{\nu} = h$, we perform another concurrent Newton step on $w_2^{(2)}$ and $w_3^{(2)}$. The reason we do not perform any actions on $w_1^{(2)}$ is that it is already green (green means converged).

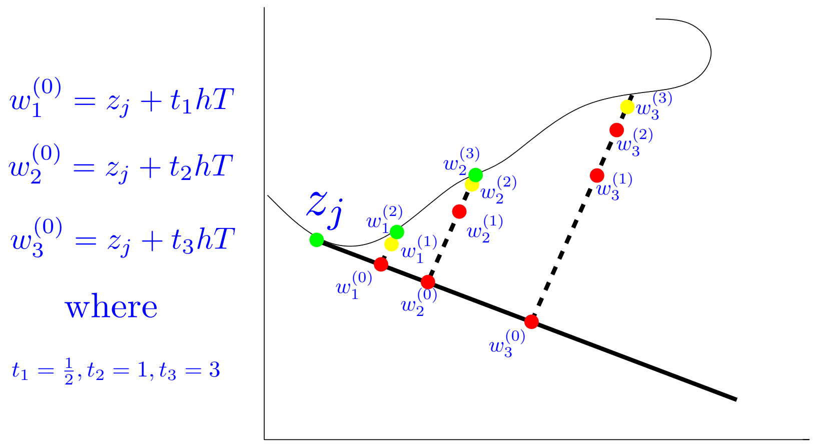

Figure 3.12: After three concurrent Newton steps ($\nu = 3$).

After three concurrent Newton steps (Figure 3.12) points $w_1^{(2)}, w_2^{(3)}$ are green and $w_3^{(3)}$ is yellow. If $w_3^{(3)}$ remained red and did not turn into yellow then we would choose $z_{j+1} = w_2^{(3)}$ since $t_2 h$ is the largest step size that corresponds to a converged point. However, since $w_3^{(3)}$ turned out yellow we perform the following comparison: $\frac{t_G h}{\nu} < \frac{t_Y h}{\nu + 1}$, where $\nu = 3, t_G = 1, t_Y = 2$ and we get $\frac{h}{3} < \frac{2h}{4}$. Therefore we perform a Newton step on $w_3^{(3)}$. There is no reason to perform more Newton steps on points $w_2^{(2)}, w_3^{(3)}$ as they are already green, see Figure 3.13.

After four concurrent Newton steps (Figure 3.13) all three points are green so there is no need to make a comparison and thus we simply choose $z_{j+1} = w_3^{(4)}$ since $t_1 < t_2 < t_3$

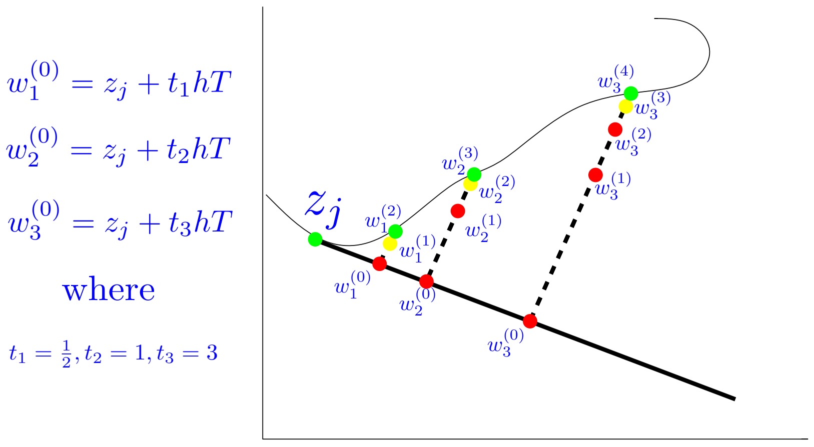

Figure 3.13: After four concurrent Newton steps all three points are green.

3.5 Pipeline Tree - Combination of Pipeline and fork techniques

Since we execute simultaneously all Newton steps in both Pipeline and Fork techniques and all Newton steps complete in approximately the same amount of time, the wall-clock time will be approximately the same as if we perform a Newton iteration just on one point as long as we are able to assign a unique process to each point. Therefore, we wish to use all available processors to minimize the number of concurrent Newton steps and consequently the wall-clock time. The purpose of the Pipeline and Fork techniques is to minimize the number of concurrent Newton steps. So we combine them both into what we call a "Pipeline Tree".

An illustrative example is where we extrapolate from the green point on the curve as we did for the fork technique, see Figure 3.14.

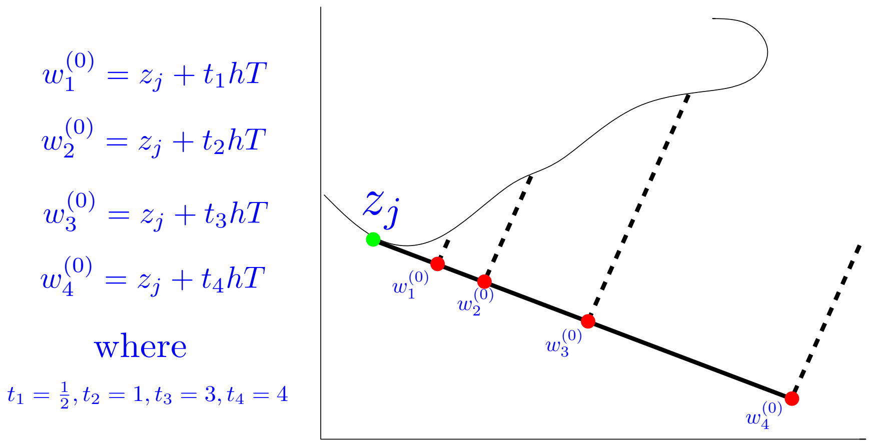
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.14: Extrapolate points with different step sizes $t_i h$.

Then we perform one concurrent corrector step, see Figure 3.15.

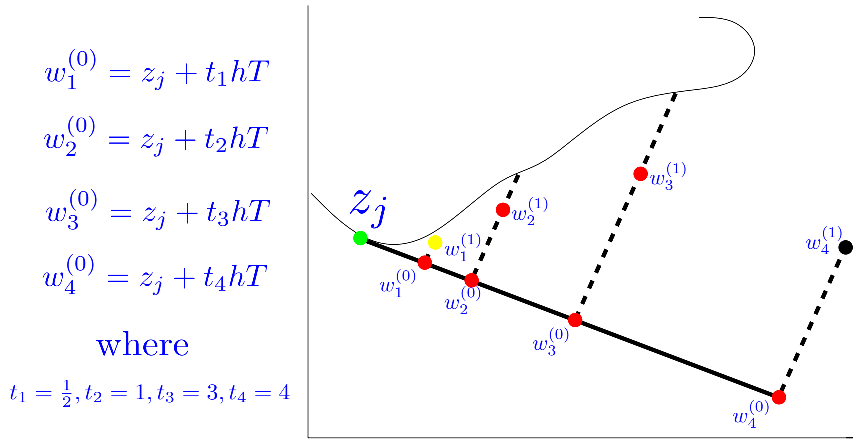
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
$w_4^{(0)} = z_j + t_4 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3, t_4 = 4$
z_j
$w_1^{(1)}$
$w_2^{(1)}$
$w_3^{(1)}$
$w_4^{(1)}$
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$
$w_4^{(0)}$

Figure 3.15: After one concurrent Newton step.

After computing residuals, we find that sequence 4 has diverged. As the result we color $w_4^{(1)}$ black as in Figure 3.15 and discard this process (Figure 3.16).

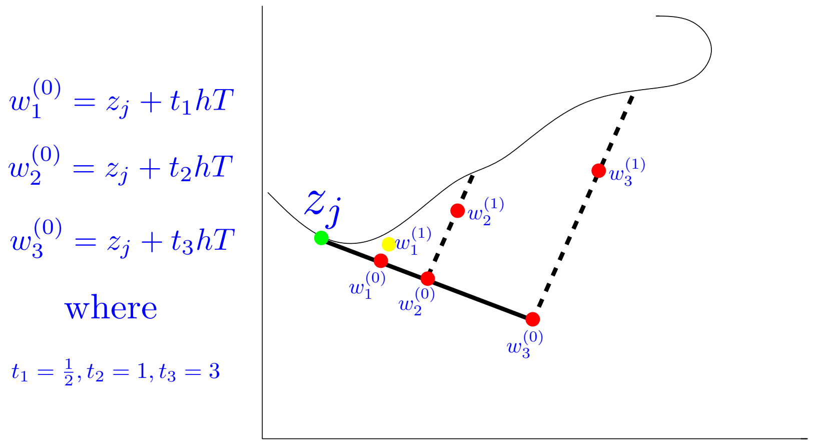
$w_1^{(0)} = z_j + t_1 hT$
$w_2^{(0)} = z_j + t_2 hT$
$w_3^{(0)} = z_j + t_3 hT$
where
$t_1 = \frac{1}{2}, t_2 = 1, t_3 = 3$
z_j
$w_1^{(1)}$
$w_2^{(1)}$
$w_3^{(1)}$
$w_1^{(0)}$
$w_2^{(0)}$
$w_3^{(0)}$

Figure 3.16: After removing black points.

At this stage none of the sequences have converged. However, we can extrapolate on intermediate results similar to the pipeline techniques (see Figure 3.17).

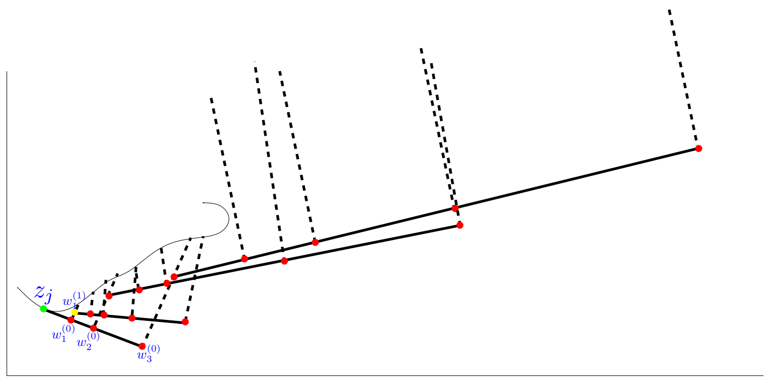

Figure 3.17: Extrapolate on the intermediate results.

Then we perform second concurrent Newton step (see Figure 3.18).

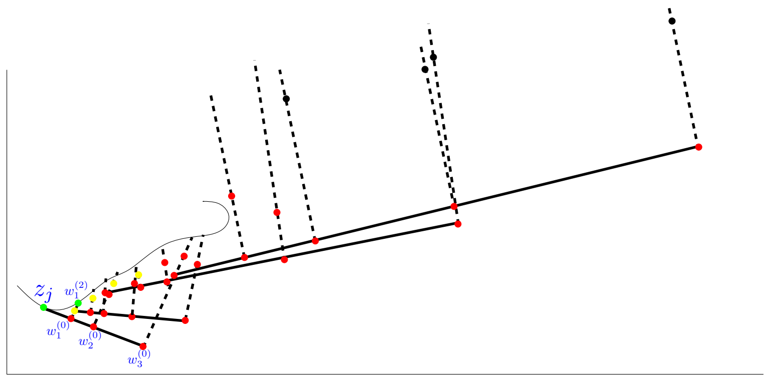

Figure 3.18: After second Newton step.

And remove black points, see Figure 3.19.

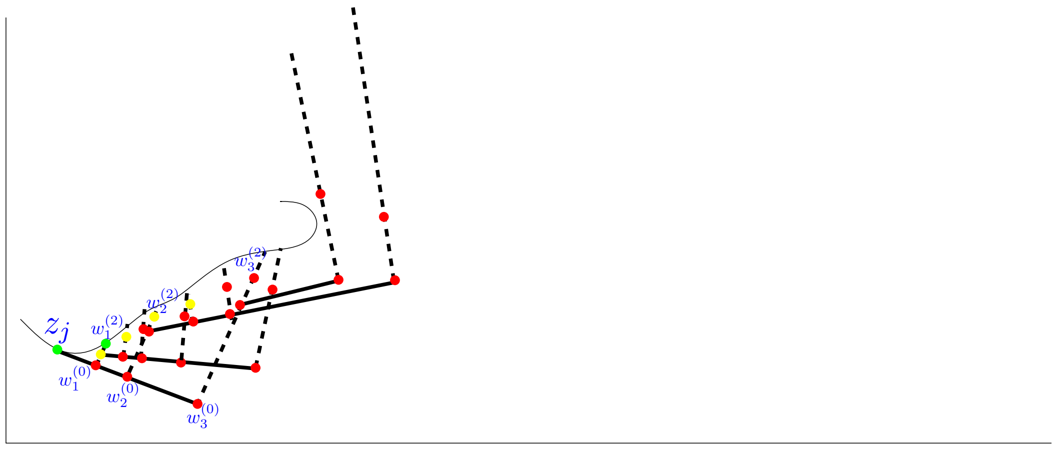

Figure 3.19: Removing black points.

Then we extrapolate a second time on intermediate results, see Figure 3.20.

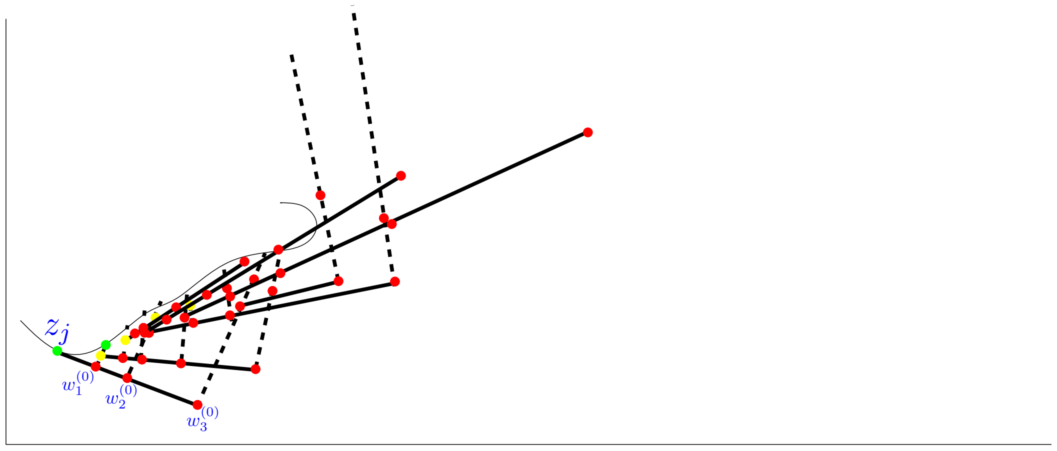

Figure 3.20: Extrapolate second time on the intermediate results.

It is immediately apparent that this scheme gets complicated quickly. After four or five extrapolations, it becomes extremely hard if not impossible to picture at all. Another question that arises is how to manage it and based on which criteria to perform the selection. To cope with this issue we borrow a data structure from graph theory, namely a rooted tree (see Figure 3.21).

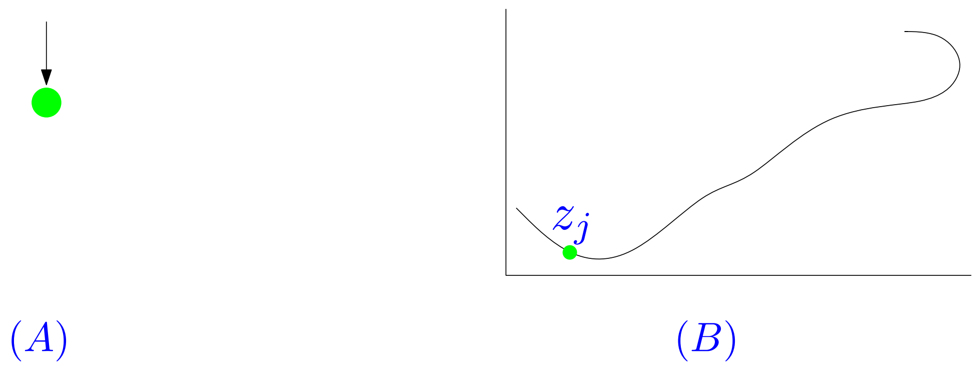

Figure 3.21: Point z_j on the curve corresponds to the root of the tree.

In the fork technique, given a point on the curve we used to extrapolate points with different t_i. Here, we instead spawn child nodes with different t_i from the root of the tree. In this structure each node contains all information about the associated point so we can interchangeably think of points and nodes as they essentially represent the same information (see Figure 3.22).

Each node in the tree corresponds to a point, either it is a green point that has converged and it corresponds to a green node or a yellow/red point that corresponds to a yellow/red node and it is still in progress. Black nodes correspond to black points that have diverged and we remove them before we spawn child nodes. When we remove black nodes, we also remove the entire subtrees associated with them.

As mentioned above, the process is synchronized before and after each Newton step. Therefore, all yellow/red nodes start and finish execution at the same time. Let us call a Tree Execution a concurrent execution of all yellow/red nodes in the Tree.

Under the following conditions each Tree execution completes in exactly the same

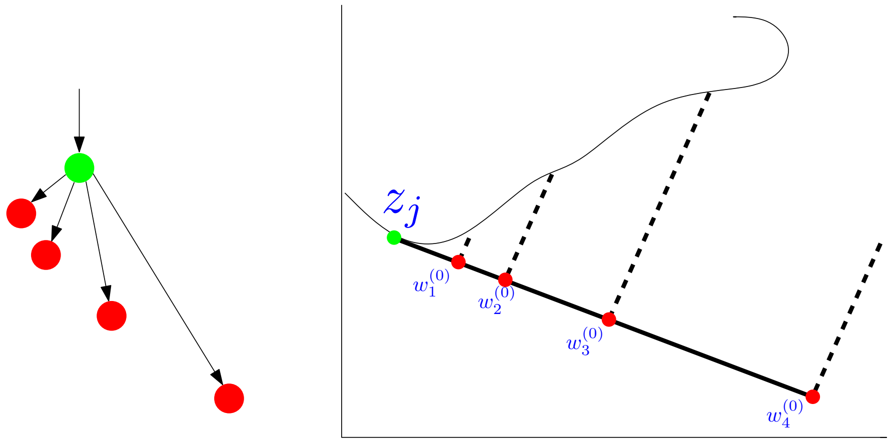

Figure 3.22: Fork from the root and spawn child nodes.

time regardless of its size and internal structure:

- Assume there are enough processors available to attach a unique process to each node.
- Let us neglect the communication and synchronization times.
- Assume all nodes in the tree take exactly the same time to complete.
- Assume there is at least one yellow/red node in the tree.

Then we perform one Tree Execution, see Figure 3.23.

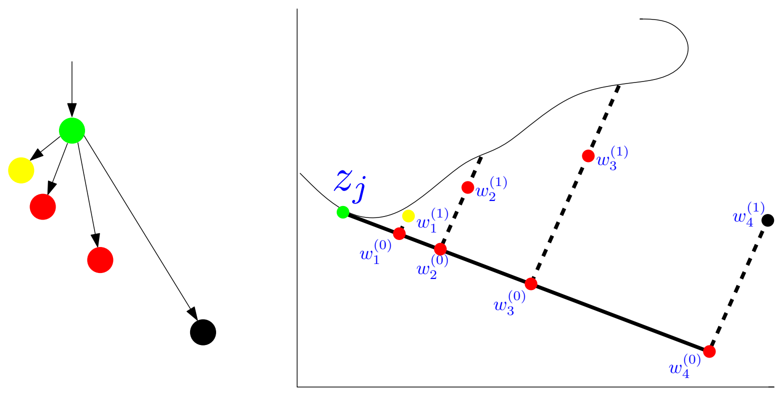

Figure 3.23: After one Tree Execution, $\nu = 1$.

After one Tree Execution one node becomes yellow, two nodes remain red, and one becomes black. Then we remove the black nodes (see Figure 3.24).

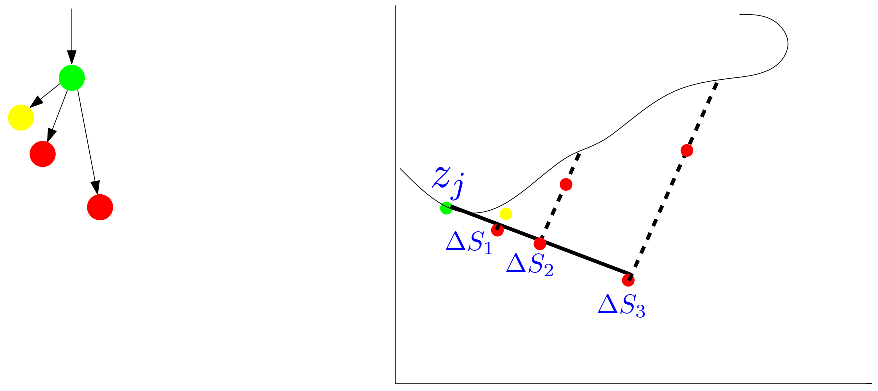

Figure 3.24: After removing black nodes.

Then we extrapolate for the second time on intermediate results or equivalently spawn child nodes from each leaf, which increase tree depth to $(d = 2)$ as in Figure 3.25.

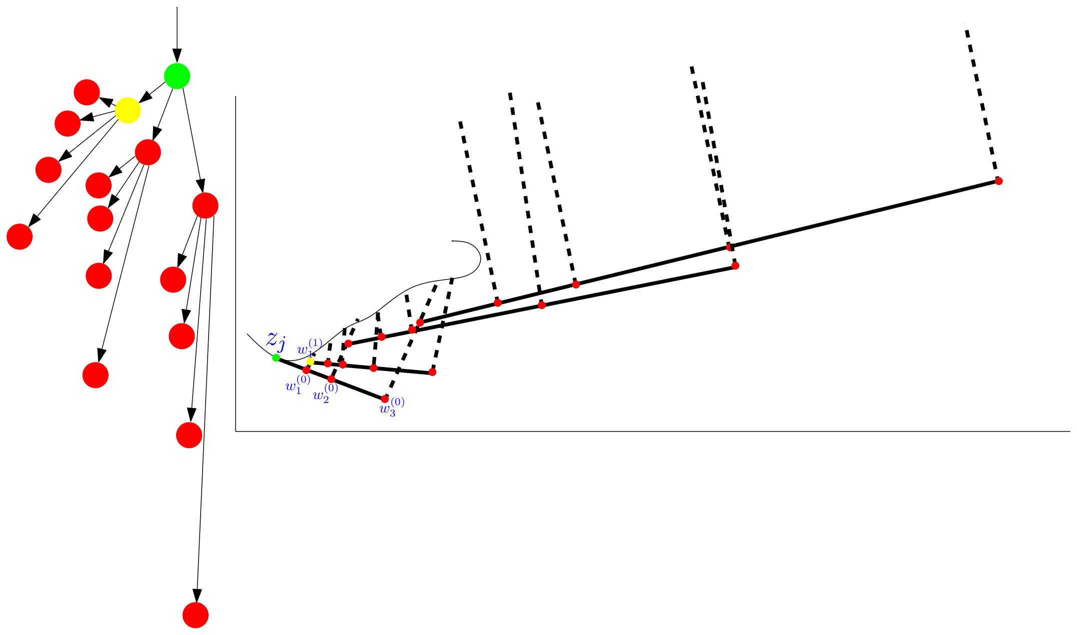

Figure 3.25: Spawn child nodes.

Then Execute Tree for the second time, see Figure 3.26.

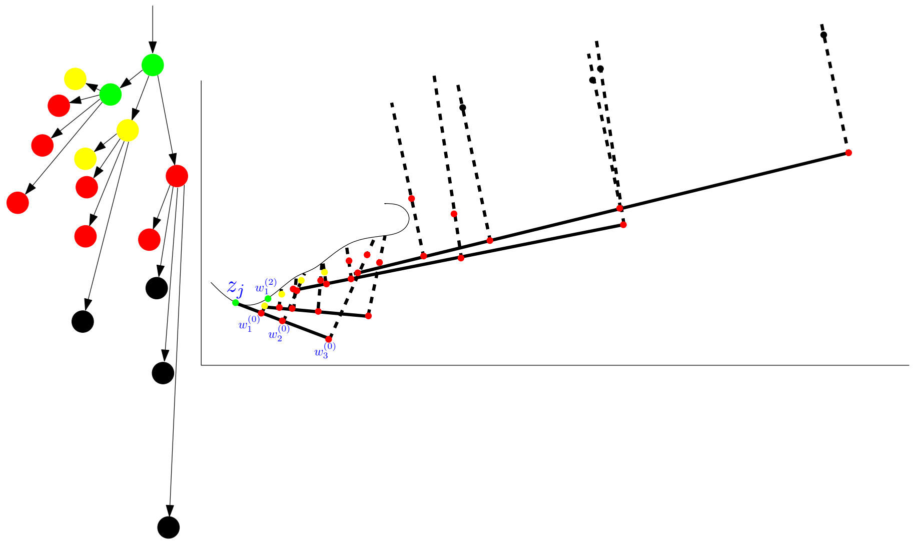

Figure 3.26: After second Tree Execution, $\nu = 2$.

Then we remove black nodes, see Figure 3.27.

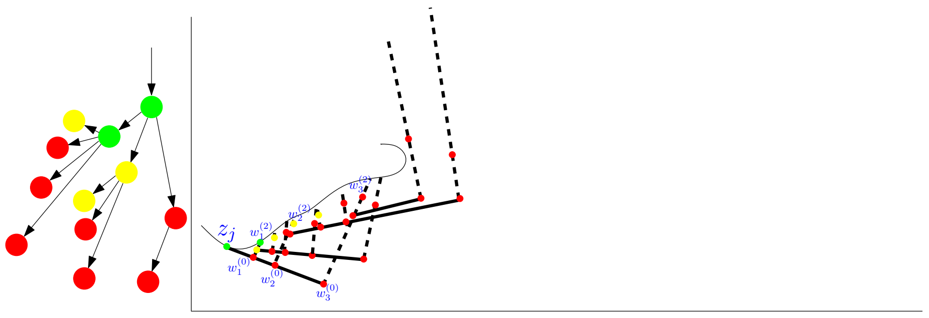

Figure 3.27: After removing black subtrees.

And spawn for the third time, see Figure 3.28.

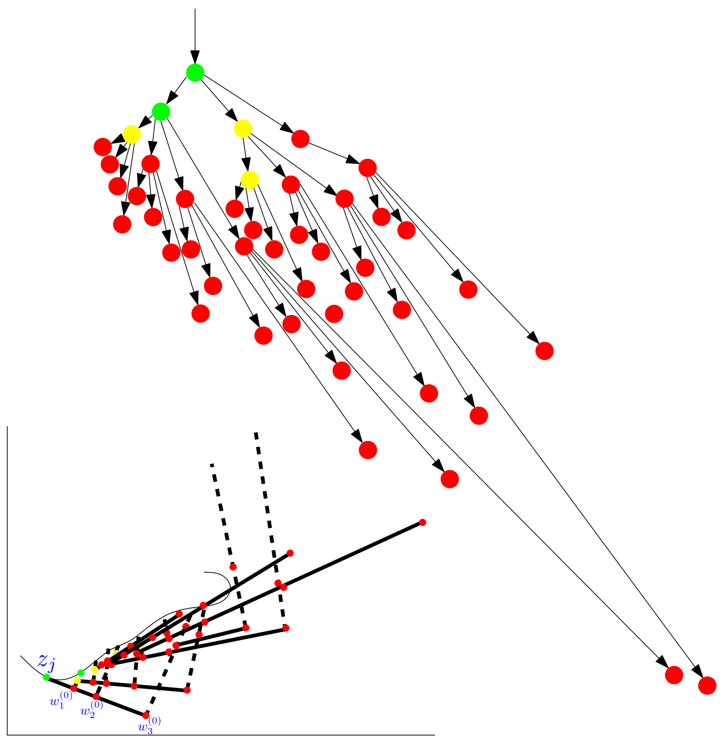

Figure 3.28: After spawning for the third time.

The problem that arises in the Pipeline Tree is that the tree grows exponentially and each yellow/red node has a physical processor attached to it. However, we have only a limited number of processors available so we need to truncate the tree and to choose only one set of nodes that leads to the final solution.

Therefore we came up with a Tree management algorithm that monitors the progress and decides which brunches of the tree to truncate. The truncation and decision criteria in this algorithm are similar to the one in the Fork technique and later we provide a detailed algorithm. Since the fork technique does not contain extrapolation on intermediate results, it directly corresponds to a tree with depth one whereas the pipeline tree can be of any depth. Remember, we based the fork technique criteria exclusively on green and yellow nodes and neglected the red nodes.

In the pipeline tree however, we consider yellow and green nodes also at other levels.

For this purpose we recursively traverse the tree from the bottom to the top and decide at every node whether to truncate its subtrees.

Before we continue to explain the details of the truncation criteria let us introduce a path U in the pipeline tree.

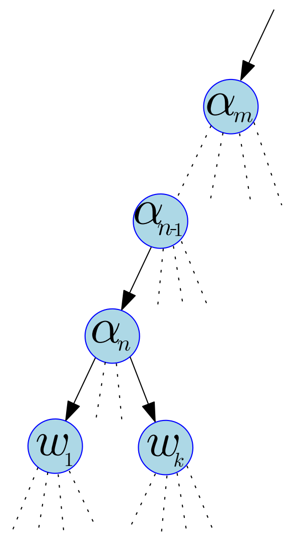

Figure 3.29: Path $U = \{\alpha_m \rightarrow \alpha_{m+1} \rightarrow ... \rightarrow \alpha_{n-1} \rightarrow \alpha_n\}$. W_n is the set of child nodes of α_n, $W_n = \{w_{_{n,1}}, w_{_{n,2}}, ..., w_{_{n,k}}\}$.

Let α_n be any node in the tree at depth $d = n$ (where $n > 0$). Then the parent of α_n is α_{n-1} and the grandparent of α_n is α_{n-2}. Then define a path $U = \{\alpha_m, \alpha_{m+1}, ..., \alpha_{n-1}, \alpha_n\}$ where m corresponds to the top of the path and n to the bottom of the path. The special case when $m = 0$ is when α_m is the root of the pipeline tree. W_n is the set of child nodes of α_n, $W_n = \{w_{_{n,1}}, w_{_{n,2}}, ..., w_{_{n,k}}\}$. We then define:

- Green Path: $U_{_G} = \{\alpha_m, \alpha_{m+1}, ..., \alpha_{n-1}, \alpha_n\}$ where all nodes $\alpha_i, i \in [m, n]$ are green and none of child nodes $W_n = w_{_{n,1}}, w_{_{n,2}}, ...w_{_{n,k}}$ of α_n are green. In addition all nodes on the path $\{\alpha_0, \alpha_1, ..., \alpha_{m-1}\}$ are green or yellow.

- Yellow Path: $U_{_Y} = \{\alpha_0, \alpha_1, ..., \alpha_m, \alpha_{m+1}, ..., \alpha_{n-1}, \alpha_n\}$ where all nodes $\alpha_i, i \in [m, n]$ are green or yellow and none of child nodes $W_n = w_{_{n,1}}, w_{_{n,2}}, ...w_{_{n,k}}$ of α_n are green or yellow.

- Green Path Length: the sum of all steps sizes along the Green Path

$$L_G = \sum_{i=m+1}^{n} h_{\alpha_i}, \text{ where } \alpha_i \in U_G$$

- Yellow Path Length: the sum of all steps sizes along the Green Path

$$L_Y = \sum_{i=m+1}^{n} h_{\alpha_i}, \text{ where } \alpha_i \in U_Y$$

- Green Path Iterations: the total number of concurrent corrector steps that have been performed on the green path U_G, denoted by ν_G

- Yellow Path Iterations: the total number of concurrent corrector steps that have been performed on the green path U_Y, denoted by ν_Y

Below we provide the brief data structure of a node α_m in the Pipeline Tree:

Brief Node Data Structure	
$Color$	Represents node color; which can be Green, Yellow, Red or Black
ν_{node}	Number of Newton corrector steps that have been performed on that node.
ν_{parent}	Number of parent's iteration to the moment the node was spawned
L_G	Length of Green path from current node α_m to the deepest node on the green path α_n
L_Y	Length of Yellow path from current node α_m to the deepest node on the yellow path α_n
ν_G	Number of iterations that have been performed on the Green Path.
ν_Y	Number of iterations that have been performed on the Yellow Path
h_m	Step size with which it was spawned from α_{m-1}
W_m	Set of child nodes

Now we perform a comparison similar to the one in the Fork technique where we considered the speed $S_i = t_i h/\nu^{(i)}$. However, now we consider the total length of each green and yellow path L_G and L_Y and number of corrector steps ν_G and ν_Y. Obviously each node can have multiple green and yellow paths, however we consider only the longest green and the longest yellow paths. By longest we refer to the green path whose length L_G is bigger than of any other green path and to yellow path whose length L_Y is bigger than of any other yellow path.

Below we provide a complete pseudo code.

Pipeline Tree($\alpha_m, t = \{t_1, t_2, ...t_k\}$)

T_{α_m} is a unit direction.

$\tilde{\alpha}_m$ is a point associated with node α_m from which the prediction is performed.

t are multiplication factors

h_{α_n} is a step size associated with α_m

1. For each leaf α_n associated with the subtree α_m generate k predictions $w_{n,i} = \tilde{\alpha}_n + t_i h_{\alpha_n} T_{\alpha_n}$.

2. Perform a Tree Execution (Tree Execution is defined in Chapter 3.6).

3. Traverse the tree recursively from the leaves up, painting the nodes according to the residual criteria.

   - ● Green: converged, i.e., $\tilde{\alpha}_m \in \Gamma$ within a prescribed residual tolerance.
   - ● Yellow: almost converged, i.e., $\tilde{\alpha}_m$ is close to Γ and the iteration should converge with one more correction step;
   - ● Red: unknown, i.e., the iteration may or may not converge. The current iterate $\tilde{\alpha}_m$ is not yet within the basin of attraction of a suitable zero of $G(z)$ (which may or may not exist)
   - ● Black: diverged, i.e., $\tilde{\alpha}_m$ is far from Γ (the iteration has not made a sufficient progress or has diverged).

4. Truncate redundant subtrees

   If $\dfrac{L_G}{\nu_G} > \dfrac{L_Y}{\nu_Y + 1}$ then

   Truncate W_m, leave only one $w_{m,i} = w_{m,G}$ where $w_{m,G} \in W_m$ is a child node of α_m with associated subtree

5. Remove all the dead black nodes with their subtrees.

6. Move the root to the deepest node of the green path attached to it.

7. Return to step (1) and make prediction steps on appropriate nodes.

Let us assign step sizes to nodes and see how the truncation happens in practice (see Figure 3.30).

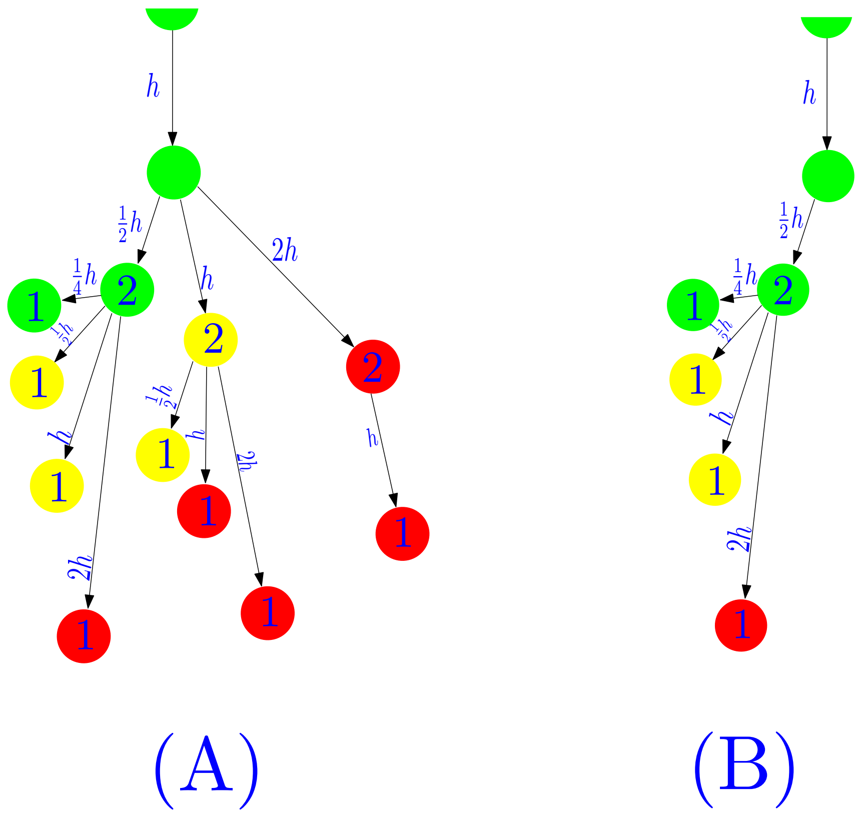

Figure 3.30: Step sizes $t_i h$ and number of iterations that every node has performed.

Next we update the root and spawn new nodes again from each leaf, see Figure 3.31.

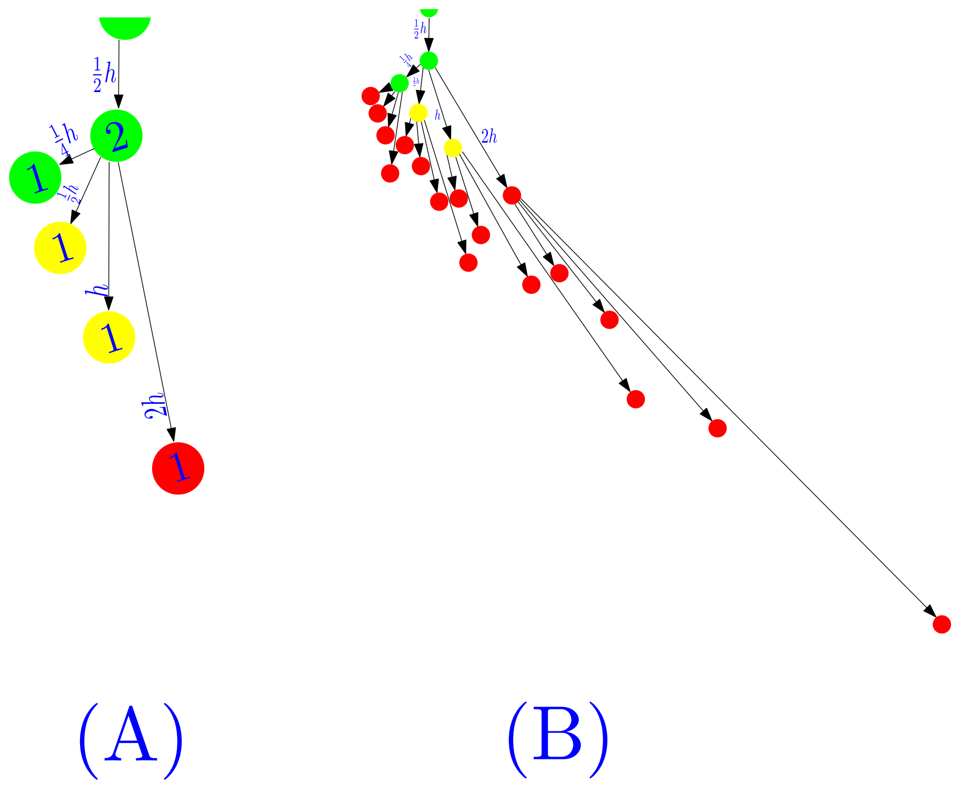

Figure 3.31: Here we can see step sizes $t_i h$ for every node

In principle the tree can grow exponentially. However, we limit the maximum depth of the tree and thus force the tree to truncate eventually. We can also calculate the maximum number of processors this software may require as we know the maximum depth of the tree and the number of child nodes that spawned from each Leaf, and we present the formula in the next section.

3.6 Upper Bounds

First, let us calculate the maximum number of processors that the software may require. Given maximum depth, d, and the number of child nodes k that we spawn for each leaf. The maximum number of processes required is less than or equal to the number of nodes in the tree excluding the root node. Where the total number of nodes in the tree excluding the root node is $\sum_{i=1}^{d} k^i = \frac{k - k^{(d+1)}}{1-k}$

Second, we analyze the case when assumption that all Newton corrector steps complete in exactly the same time, does not hold.

- Let τ_{min} be the minimal time through the entire computation for a corrector step to complete.
- Let τ_{max} be the maximal time through the entire computation for a corrector step to complete.
- Let Φ be the time in which the entire computation completes.
- Let η be the number of tree executions to complete the entire computation.
- We neglect communication and synchronization times.
- Then $\Phi_{min} = \eta\tau_{\text{min}}$ and $\Phi_{max} = \eta\tau_{\text{max}}$
- Let $\eta\tau_{\text{min}} = x\eta\tau_{\text{max}} \Rightarrow x = \frac{\tau_{\text{min}}}{\tau_{\text{max}}}$
- Since $\Phi_{min} \le \Phi \le \Phi_{max} \Rightarrow \eta\tau_{min} \le \Phi \le \eta\tau_{max}$
- Then $\eta\tau_{min} \le \Phi \le \frac{1}{x}\eta\tau_{min}$
- Then $0 \le \Phi - \eta\tau_{min} \le \eta\tau_{max}\left(\frac{1}{x} - 1\right)$
- Finally, $0 \le \Phi - \Phi_{min} \le \Phi_{min}\left(\frac{\tau_{max} - \tau_{min}}{\tau_{min}}\right)$

This means that if the user can estimate the value of τ_{min} and τ_{max} then she can find the maximum possible delay imposed by the difference in time to complete different Newton steps.

Chapter 4

Numerical Experiments

We have run our software on two different test problems. The first problem is an artificial problem for [8] debugging and initial testing. The second problem is an existing problem in turbulence for which the computation is extremely time consuming [3]. The reason we needed both a synthetic and a turbulence problem is two fold. First, we wanted to experiment on problems of a different nature to monitor how our software behaves. Second it would be almost impossible to debug the software and perform initial testings during the development stage on the turbulence problem as it would take a several hours to get results from a single test, and to develop this software we have performed about five thousands tests. Since each test on the turbulence problem runs for at least three hours, the entire debugging and testing would have taken 1.5×10^4 hours, which is equivalent to 625 days. Even though all the initial testing and debugging have been performed on the synthetic problem, the software behaved very similar when applied to the turbulence problem. It has also provided a significant speed up only with little adjustments.

Another technique that we came up with due to a time limit and shortage of resources is to run the software in an environment that simulates multiple processors. The technique might seem artificial at first, however our tests have shown that it is accurate up to a few percent for the turbulence problem, and even more accurate for the synthetic problem when we artificially slowed down the Newton method. Basically we find the ratio of the number of Tree Executions for both parallelized and non-parallelized pseudo-arclength continuation and subtract the ratio over their computation times. A clear definition of accuracy is: $\epsilon = \left|\frac{\tau_1}{\tau_2} - \frac{\nu_1}{\nu_2}\right|$, while the smaller the ϵ the higher the accuracy. We provide a detailed explanation of this technique and theoretical analysis of its accuracy in Chapter 3.

We have tested the synthetic problem on nine tree configurations and the turbulence problem on six tree configurations. Here, by tree configuration we refer to the maximum depth of the tree and to the number of child nodes spawned from each leaf. For both problems all configurations provide a speed up over one processor while there are configurations that provide a significant speed up on feasible numbers of processors. We provide explicit results later in this chapter.

Our tests cover almost the entire domain of such problems since results of our software depend mostly on high level factors. The main aspects are that variation in time to perform a Newton method is small (Chapter 3), the time to perform a Newton iteration is large compare to communication and synchronization times (Chapter 3) and that the curve geometry changes significantly from place to place (see Figure 4.1).

4.1 The synthetic problem

We are solving the nonlinear system of equations (see [8])

$$f_i(x, \lambda) = x_i - \exp\left[\lambda \cos\left(i \sum_{i=1}^{N-1} x_i\right)\right], i \in [1, N-1], x_i, \lambda \in \mathbb{R} \tag{4.1}$$

We can write it as $F(x, \lambda) = 0$

Then we are looking for the curve

$\Gamma = \{(x, \lambda) \in \mathbb{R}^{N-1} \times \mathbb{R} \mid F(x, \lambda) = 0\}$

By inspection we find a trivial solution $(\lambda_0, x_0) \in \Gamma$ where $\lambda_0 = 0, x_0 = 1$. Figure 4.1 shows a projection of the curve Γ.

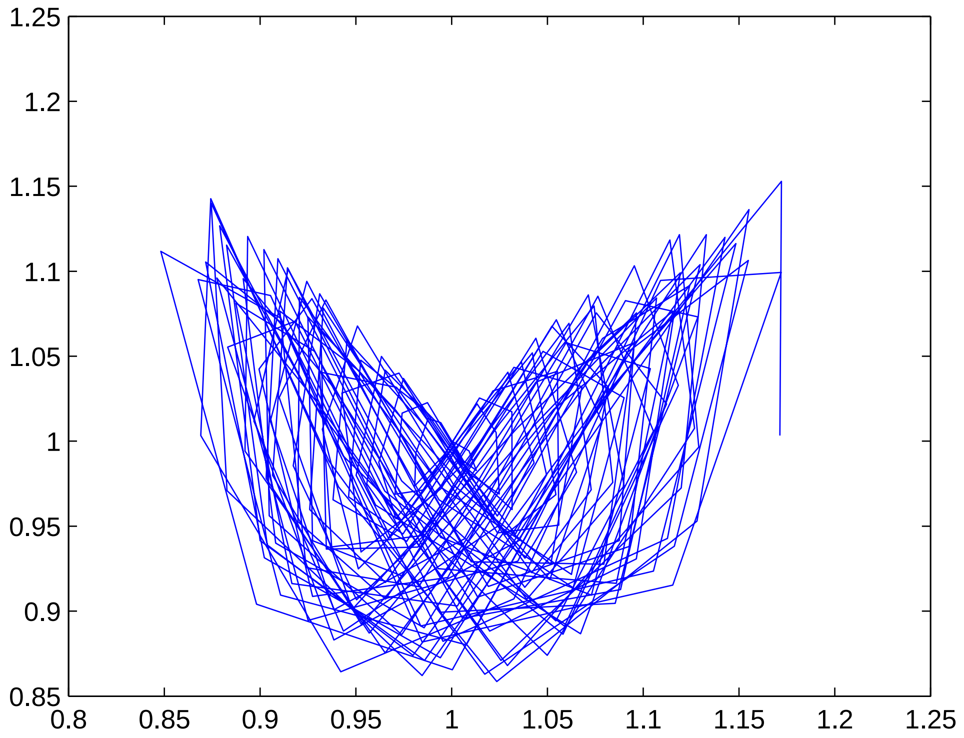

Figure 4.1: A projection of the solution curve of the synthetic problem onto randomly chosen components of x.

4.2 The turbulence problem

The turbulence problem is studied by continuation of time-periodic solutions to the Navier-Stokes equation for the motion of incompressible, viscous fluid in a box with periodic boundary conditions in every direction. The variables in the simulation code are the Fourier coefficients of the vorticity field, truncated to a finite number. For a turbulent flow this number is extremely large and even for weakly turbulent flow this number is of order $O(10^6)$ and it increases algebraically with the Reynolds number. To make the study feasible, symmetry reduction was applied, bringing this number down to approximately 10^4. Such a flow is called *high symmetric* [11].

The resulting system is a set of coupled, nonlinear ODEs with a single parameter, namely the kinematic viscosity which in this case corresponds to λ and determines the Reynolds number of the flow. In this problem the spatial resolution is fixed to 2^6 in every direction, which after de-aliasing and symmetry reduction, gives $N = 704$ variables.

A periodic solution was filtered from turbulent data at the highest viscosity, at which the flow is relatively quiescent. Subsequently, pseudo-arclength continuation was used to track this solution to a smaller viscosity.

Since the resolution is relatively low, we compute the solution curve for a smooth viscous flow, where Reynolds numbers are low. The initial solution was found simply by looking at a time series starting from random initial conditions. Some segments of the chaotic signal looked almost periodic. From such an approximately periodic segment, a periodic orbit was found using Newton iterations. Subsequently, pseudo-arclength continuation was used to track the solution for different Reynolds numbers. In the correction step of pseudo-arclength continuation, the linear problem associated with the Newton-Raphson iteration is solved by the iterative Generalized Minimal Residual (GMRES) method [7]. Newton-Krylov continuation is the combination of pseudo-arclength continuation with a Krylov subspace method and was first implemented by [9].

Each linear problem takes from twelve to fifteen GMRES iterations to solve in this continuation, and each GMRES iteration requires a simulation of the flow along the whole periodic orbit. In Fig. 4.2 we show the result of pseudo-arclength continuation.

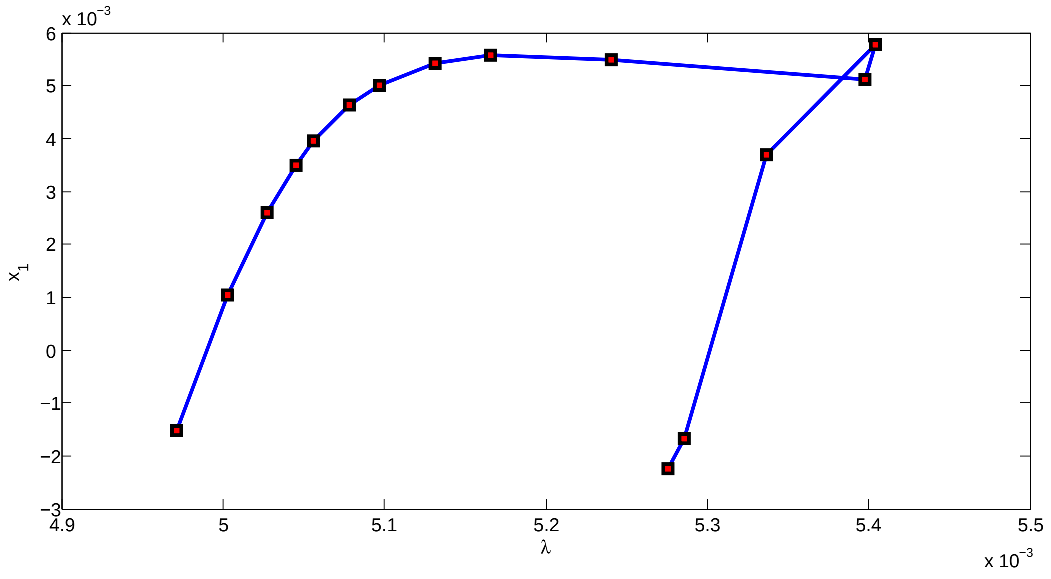

Figure 4.2: Continuation of a periodic solution in high-symmetric flow by pseudo-arclength continuation. Where the vertical axis represents x_1 and the horizontal axis represents λ.

In section 4.3.2, we show that the number of iteration and, equivalently, the wall time of this continuation can be brought down by a factor of three by application of our software.

4.3 Results

4.3.1 Results for the synthetic problem

We have tested the synthetic problem on nine different tree configurations with $N = 300$. Figure 4.3 shows a three dimensional bar graph which represents those configurations with corresponding results. The x-axis of the graph represents the maximum depth of the Pipeline Tree, the y-axis, the number of child nodes that each leaf node spawns and finally the z-axis (the height) of the graph represents the number of tree executions for the given tree configuration. Here we can see that the $(d = 3, k = 3)$ configuration runs approximately 60% faster than the $(d = 1, k = 1)$ configuration. The configuration the $(d = 1, k = 1)$ corresponds to a serial version of pseudo-arclength continuation running on a single processor. Care has been taken to ensure that the serial version of pseudo-arclength continuation is computed in a nearly optimal time. All the parameter values are listed in appendix A.1 for both the synthetic and the turbulence problems.

We can immediately notice that the green bar in the middle $(d = 2, k = 2)$ is a little higher than the green bar on its right $(d = 2, k = 1)$ even though it corresponds to a larger tree configuration and therefore requires more processors. This happens because the software uses heuristics to determine an optimal path in the tree so there is no guarantee that increasing number of spawned child nodes leads to a better performance. However, as we can see on the graph, the software has performed better in 8/9 configurations for increasing number of child nodes which suggests that in general increasing number of spawned child nodes does lead to a better performance.

Another idea that we get from this graph is that for the synthetic problem the increasing depth of the tree plays a larger role in the minimization of the number of tree executions than the increasing number of spawned child nodes. Hence, if only a limited number of processors is available a user will choose either the $(d = 3, k = 1)$ or the $(d = 3, k = 2)$ configuration.

Since, we distribute Newton methods among CPUs that do not share memory, each CPU stores the matrix 300×300 and a vector 300×1 of double precision floating point numbers. While the memory used by our algorithm to store the tree and other variables is negligible.

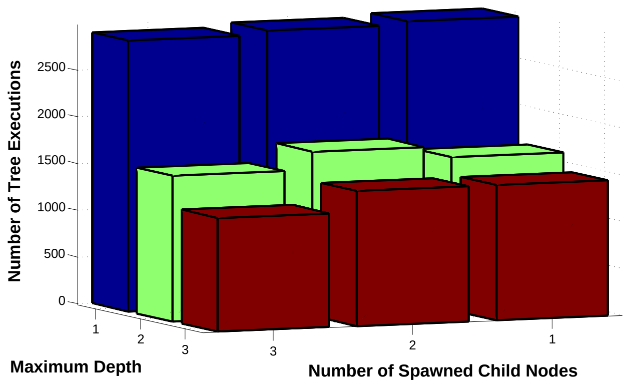

Figure 4.3: Number of tree executions for different configurations of the synthetic problem.

4.3.2 Results for the turbulence problem

We have tested the turbulence problem on six different tree configurations with $n = 704$. Similar to the synthetic problem, Figure 4.4 shows a three dimensional bar graph which represents those configurations with corresponding results. Here we can see that the $(d = 3, k = 2)$ configuration runs approximately 60% faster than a serial version of pseudo-arclength continuation.

In contrast to Figure 4.3, in Figure 4.4 the graph represents a significant gain from both increasing the number of child nodes and increasing the maximal depth of the tree. This is because the fluids problem consists of $N = 704$ variables and appears to change much more rapidly from one continuation point to the next. Basically the variation of sensitivity to the initial guess is higher then in the synthetic problem since N is larger. For that reason, there is a greater benefit of trying many step sizes in parallel.

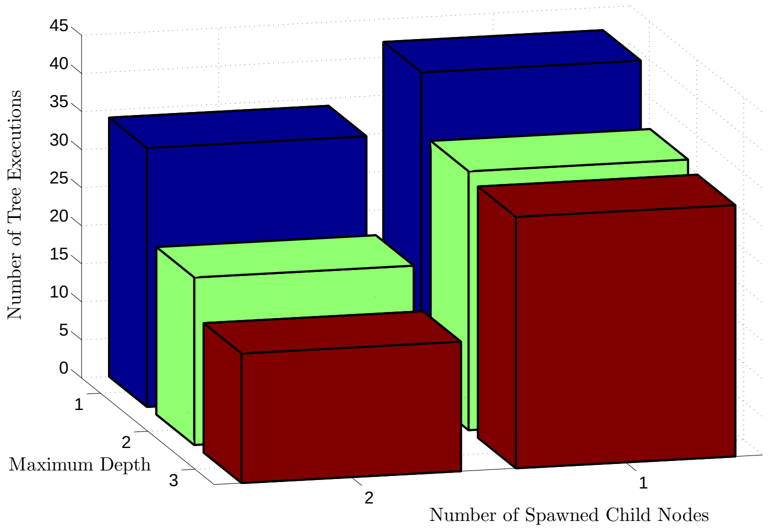

Figure 4.4: Number of tree executions for different configurations of the turbulence problem.

Since, in the turbulence problem the Newton iteration involves computing the Krylov subspace, each CPU typically stores 15 to 25 Krylov vectors 700×1 of double precision floating point numbers. While the memory used by our algorithm to store the tree and other variables is negligible.

Chapter 5

Conclusion

5.1 Summary

We have developed and implemented an algorithm that parallelizes pseudo-arclength continuation. Our approach is completely innovative and in contrast to already existing work our parallelization algorithm is completely independent of the internal solver. In principle pseudo-arclength continuation is inherently sequential, where each step largely depends on the previous one and therefore it is hard to parallelize in a generic case. The key of our algorithm is to extrapolate on intermediate results and to try many points with different step sizes at the same time. All this leads to a very complex structure with many uncertainties. For instance it is not clear which results to choose and which processes to eliminate or to replace by others. We cope with the difficultly to synchronize multiple trials and extrapolations by applying a tree structure and a recursive algorithm to traverse the tree and to make appropriate decisions. We have tried our software on two problems, the synthetic problem and the turbulence problem. We have performed numerical experiments that indicate a 60% performance increase of our parallel version vs. the serial version of pseudo-arclength continuation.

The software provides a better speed up when N increases as in a higher dimensions the step size selection is often less predictable and therefore trying multiple step sizes at a time lead to a more significant performance improvement. This is supported by our results for synthetic and turbulence problems, where for the synthetic problem $N = 300$ and for the turbulence problem $N = 704$. The performance in the turbulence problem improves more significantly when the number of spawned child nodes increases than that in the synthetic problem. In general it is very problem dependent whether increasing the number of child nodes spawned or having a larger tree depth increases the performance. Our results provide some intuition on which tree configuration to choose for each problem.

5.2 Rule of thumb

Here we would like to provide a few suggestions to potential users of our software. These suggestions are based on our experience and intuition, however they are consists with numerical experiments from Chapter 4 as on the Figures 4.3 and 4.4. At the beginning we recommend the user to try configurations $(d = 3, k = 1)$ and $(d = 1, k = 3)$ that correspond to pipeline and fork techniques. The reason for this is twofold; first these configurations does not require many CPUs and can be tested on a personal computer, second this would provide the user with information on effectiveness of spawning multiple child nodes vs. having a greater depth of the tree. Our intuition, is whenever the complexity of the curve increases, the gain from spawning multiple child nodes increases, while the gain from a larger tree depth remains constant or decreases. Provided enough child nodes for a complex problem, the gain from a larger depth provides a comparable gain to a less complex problem. Since, given enough child nodes there is a high probability that at least one of them will converge, meaning that we can extrapolate from it if the tree depth is large enough.

5.3 Future Work

We would like to extend our algorithm to be able to provide a performance increase for problems where the deviation in time between Newton steps is significant. Also we want to try our software on extremely large problems, for instance where $N = 10^5$ as we suspect that our software will perform significantly better when the dimension of the Jacobian increases. Another interesting test would be to see how our software behaves for an extremely large tree structure, for instance $(d = 5, k = 5)$. However, such a test would require access to large amount of resources as it requires over 3000 processors and it cannot be run in a simulated environment on a desktop machine since it would take eternity to complete. It would also be very interesting to implement this software on a GPU, since it has hundreds of processors available. This might especially improve performance for problems with relatively fasts correction steps, since communication is significantly faster on a shared memory rather than on distributed CPU clusters.

Appendix A

Appendices

A.1 Pseudo-arclength continuation parameters

We have run the synthetic problem with the following parameters:

- $\lambda_{init} = 0$
- $\lambda_{end} = 0.8$
- $\epsilon = 10^{-5}$ - Residual Tolerance
- $\alpha_{color} = Yellow$, **if** $\dfrac{\log(\parallel \mathbf{r}^{(\nu)} \parallel_2^2)}{2} < \log(\epsilon)$
- $Progress = \dfrac{\log(\parallel \mathbf{r}^{(\nu-1)} \parallel_2^2)}{\log(\parallel \mathbf{r}^{(\nu)} \parallel_2^2)}$
- $Progress_{min} = 1.3$
- Step size distributions $\mathbf{t} = \{1.1\}$, $\{1.1, 1.21\}$, $\{1.1, 1.21, 1\}$
- $N = 300$

We have run the turbulence problem with the following parameters:

- $\lambda_{init} = 5.27 \times 10^{-3}$

- $\lambda_{end} = 5 \times 10{-}3$
- $\epsilon = 10^{-5}$ - Residual Tolerance
- $\alpha_{color} = Yellow$, **if** $\frac{\log(\| \mathbf{r}^{(\nu)} \|_2^2)}{2} < \log(\epsilon)$
- $Progress = \frac{\log(\| \mathbf{r}^{(\nu-1)} \|_2^2)}{\log(\| \mathbf{r}^{(\nu)} \|_2^2)}$
- $Progress_{min} = 1.3$
- Step size distributions $\{1.1\}$, $\{0.75, 1.5\}$, $\{0.75, 1.5, 1\}$.
- $N = 704$

A.2 Experimental Details

Synthetic problem

Number of spawned Child nodes	Maximal tree depth	Number of tree exections
1	1	2983
1	2	1634
1	3	1447
2	1	2945
2	2	1757
2	3	1446
3	1	2901
3	2	1561
3	3	1219

Fluids problem

Number of spawned Child nodes	Maximal tree depth	Number of tree exections
1	1	42
1	2	34
1	3	33
2	1	34
2	2	22
2	3	17

A.3 Implementation and Design decisions

In order to create a flexible software that is easy to understand and modify, we divided its implementation into four modules. Three of those modules perform isolated tasks and the main fourth module combines it all together. The main module implements the essential part of our algorithm and contains only high level details. Essentially it is very close to a pseudo code. It contains a main loop which repaints, traverses and executes the tree. It includes very few mathematical statements and seems to be completely isolated from MPI communication. The second module provides a communication mechanism which encapsulates all MPI communication and provides high level functions to distributed the work to slave processes. The third module contains mathematical functions and imports other functions such as Newton method execution from external sources. Similar to the second module, the third module encapsulates all mathematical implementation and provide a high level functions to the key algorithm. The fourth module loads all necessary data from the files and fills out structures, such that other modules assume that all data has been loaded.

Bibliography

[1] Keller, H. B., "Numerical Solution of Bifurcation and Nonlinear Eigenvalue Problems," *Applications of Bifurcation Theory*, P. Rabinowitz ed., Academic Press, 1977.

[2] Doedel, E. J. and Champneys, A. R. and Dercole, F. and Fairgrieve, T. F. and Kuznetsov, Yu. A. and Oldeman, B. and Paffenroth, R. and Sandstede, B. and Wang, X. and Zhang, C., "AUTO-07P: Continuation and bifurcation software for ordinary differential equations" *Concordia University, Montreal, Canada. Available from http://sourceforge.net/projects/auto-07p/*, 2008.

[3] Veen, L. v., Kida, S., and Kawahara, G., "Periodic motion representing isotropic turbulence". *Fluid Dyn. Res.*, vol. 38, pp. 19-46, 2006.

[4] Barney, B., "Introduction to Parallel Computing," *Lawrence Livermore National Laboratory. Available from http://www.scribd.com/doc/35620491/Parallel-Computing*, 2010.

[5] Kelley, C.T., "Solving Nonlinear Equations with Newton's Method", SIAM, 2003.

[6] Varga, R.S., "Matrix Iterative Analysis," *Springer Series in Computational Mathematics*, Springer, vol. 27, pp. 1999.

[7] Saad, Y. and Schultz, M. H., "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems", *SIAM J. Sci. and Stat. Comput.*, vol. 7, pp. 856-869, 1986.

[8] Watson, L.T., “A Globally Convergent Algorithm for Computing Fixed Points of C^2 Maps”, *Appl. Math. and Comput.*, vol. 5, pp. 297-311, 1979

[9] Sánchez, J. and Net, M. and García-Archilla, B. and Simó, C., “Newton-Krylov continuation of periodic orbits for Navier-Stokes flows”, *J. Comp. Phys.*, vol. 201, pp. 13-33, 2004.

[10] Monin, A. S. and Yaglom, A. M., “Statistical fluid mechanics”, *Dover publications*, vol. 2, 2007.

[11] Kida, S., “Three-dimensional periodic flows with high-symmetry”, *J. Phys. Soc. Japan*, vol. 54, pp. 2132-2136, 1985.