Technische Universität München
Zentrum Mathematik
Wissenschaftliches Rechnen

Automatic Contour Deformation of Riemann–Hilbert Problems

Georg Wechslberger

Vollständiger Abdruck der von der Fakultät für Mathematik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Simone Warzel
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Folkmar Bornemann
	2.	Senior Lecturer Sheehan Olver, Ph.D. The University of Sydney, Australien

Die Dissertation wurde am 14.07.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Mathematik am 21.9.2015 angenommen.

Acknowledgements

First of all I wish to thank my advisor Prof. Bornemann for giving me the opportunity to write this thesis, learn so many things and work on interesting projects. I am very grateful for all the support and help during the last couple of years. Furthermore, thank you for the freedom I enjoyed while working on this thesis.

Sheehan, thank you for taking the time to read this thesis and write an evaluation. In addition, thanks for all the research you have done, upon which this thesis is build. Without you this thesis would look totally different today.

Prof. Klüppelberg, thank you for providing funding for this thesis during the first three years.

Karin, thanks a million for all your help and support during the last few years. Without you, working on this thesis would have been a lot less fun. Especially thanks for helping organize events, create doctoral caps and proofreading parts of this thesis.

Christian, thank you for always being so incredibly helpful. Working with you on any kind of IT project is always a good time.

Benedict, thank you for being such a good "roomie" in the office.

Steffi, Katharina, Michi, Horst, Johannes, thank you for all the fun at the SFB meet-ups.

All members of M3, thank you for creating such a pleasant working atmosphere. It was always fun organizing events for all of you.

Marion and Andreas, thank you for fearlessly proofreading parts of this thesis.

Petér, when you read this, it should be obvious that you lost the bet. Thanks for the cake, I'm already looking forward to it. ;-)

All the friends and family members, which have always been so supportive. It is worth a lot to know that no matter what happens, there are always people around you which you can rely on. So, thank you. You are awesome!

This research was supported by the German Research Foundation (DFG), Collaborative Research Center SFB-TR 109.

Abstract

In this thesis we propose a proof of concept algorithm for preconditioning Riemann–Hilbert Problems. It is based on the idea of converting the problem of deforming a contour into the problem of finding shortest paths subject to certain topological constraints in a graph with suitable chosen weights. To evaluate the effectiveness of the contours computed by our algorithm, we compare them with contours derived analytically using the method of nonlinear steepest descent.

Kurzfassung

In dieser Dissertation wird ein "Proof of Concept" Algorithmus für die Vorkonditionierung von Riemann–Hilbert Problemen vorgestellt. Der Algorithmus basiert auf der Idee, die Konturderformation als ein Problem kürzester Wege aufzufassen, die gewissen toplogischen Beschränkungen in einem Graph mit passend gewählten Gewichten genügen. Um die Effektivität der von unserem Algortihmus berechneten Konturen zu bestimmten, vergleichen wir sie mit Konturen, die analytisch mit der Methode des "nonlinear steepest descent“ bestimmt werden.

Contents

1. Introduction 1

2. Motivation 11
- 2.1. Problem Description . . . 11
- 2.2. Asymptotic Expansion . . . 12
- 2.3. Interval Splitting . . . 16
- 2.4. Reference Solution . . . 20
- 2.5. Numerical Results . . . 22

3. Toy Problem: Cauchy's Integral Formula 27
- 3.1. Introduction . . . 27
- 3.2. Contour Graphs and Shortest Enclosing Walks . . . 29
- 3.3. Implementation Details . . . 30
  - 3.3.1. Edge Weight Calculation . . . 30
  - 3.3.2. Reducing the Size of $\tilde{\Pi}$ . . . 31
  - 3.3.3. Size of the Grid Domain . . . 32
  - 3.3.4. Multilevel Refinement of the SEW . . . 32
  - 3.3.5. Quadrature Rule for the Cauchy Integral . . . 34
- 3.4. Numerical Results . . . 34

4. The RHP Deformation Algorithm 39
- 4.1. Condition of a Riemann–Hilbert Problem . . . 40
- 4.2. Preconditioning as a Discrete Optimization Problem . . . 43
- 4.3. Admissible Deformations of Riemann–Hilbert Problems . . . 46
  - 4.3.1. Simple Deformations . . . 46
  - 4.3.2. Multiple Deformations and Factorization: Lensing . . . 49
- 4.4. The Greedy Algorithm . . . 49
  - 4.4.1. Notation . . . 49
  - 4.4.2. Optimized Simple Deformations . . . 50
  - 4.4.3. Admissible Contour Types . . . 56
  - 4.4.4. Calculate Shortest Paths . . . 56
  - 4.4.5. Shared Subpath Improvement . . . 62
  - 4.4.6. Calculate Left and Right Sides of Splits . . . 68
- 4.5. Optimized Lensing Deformations . . . 71
- 4.6. Extensions for infinite contours . . . 78

4.7. Deformation Verification . . . 82
4.8. Implementation Details . . . 90
4.8.1. The Weights . . . 90
4.8.2. The Graph . . . 90
4.8.3. Contour Simplification . . . 90
4.8.4. Restrictions on Paths . . . 92
4.8.5. Restriction to Machine Precision . . . 93
4.8.6. Removal of Wiggling Paths . . . 95
4.8.7. Runtime Optimizations . . . 96

5. Numerical Results 97
5.1. Painlevé II . . . 97
5.2. modified Korteweg–de Vries . . . 98
5.3. Nonlinear Schrödinger . . . 105
5.4. Discrete z^α . . . 109
5.4.1. Original Contour . . . 109
5.4.2. Singular Linear System . . . 112
5.4.3. Deformed Contour . . . 112

6. Future Work 117
6.1. g–Function Deformations . . . 117
6.2. Local Refinements . . . 118
6.3. Recover Original Solution . . . 119

7. Summary 121

A. Software Documentation 123

B. Deformation Verification: Definition of ν 135

C. Visualization Styles 141

1. Introduction

Quite many integrable systems can be formulated as a Riemann–Hilbert problem (RHP). For example there are RHPs for orthogonal polynomials, special functions, Painlevé transcendents, nonlinear PDEs, combinatorial problems and even discrete functions (Its 2003). They have proven a successful tool for application such as deriving asymptotic connection formulae for Painlevé transcendents. Just recently with the development of a solver based on a spectral collocation method by Olver (2011b), RHPs also became of interest for numerical analysis. Thereby it turned out that numerical stability, and with it accurate results, for this solver could in most cases only be achieved by deforming the RHP using the method of nonlinear steepest descent (NSD) before it is used as input for the solver. In fact Olver and Trogdon (2012) argue: "One can expect that whenever the method of nonlinear steepest descent produces an asymptotic formula, the numerical method can be made asymptotically stable".

On the upside this enables the numerical treatment of a wide variety of RHPs, but there is also the downside that it effectively limits the audience of the numerical method to people who have the required expert knowledge to perform these deformations[1]. To improve this situation we propose in this thesis an algorithmic approach for this preconditioning step of deforming the RHP, which would upgrade the solver to a black box method for calculating numerical solutions of RHPs. We provide a "proof of concept" of such an algorithm, which supports only some of the deformations available for NSD.

The basic idea of our algorithmic approach is to cast the problem of deforming a RHP into the problem of finding a system of shortest paths in a weighted graph subject to some topological constraints. For now we are not able prove that the algorithm, we present in this thesis, will yield a deformation that is within a certain range of the optimal one. But instead we apply our algorithm to a few RHPs and compare the results with deformations which can be found in the literature and have been derived by people using the method of nonlinear steepest descent. In these example both ways of deriving the deformations yield similar contours and also their effectiveness for preconditioning the RHP is quite similar. After this brief introduction we give a more detailed view of what we want to achieve.

[1] Copying the deformations form the literature is also not as easy at may sound, because the deformations are seldom written for an audience that has no knowledge of NSD.

Riemann–Hilbert problems

To fix the notation, we consider RHPs given by an oriented contour Γ and a matrix-valued jump function $G : \Gamma^0 \to \mathrm{GL}(m, \mathbb{C})$. Thereby an oriented contour is a union of simple smooth curves[2] Γ_j ($j = 1, \dots, k$) in $\mathbb{C}$, with Γ_j having a designated direction in which it is traversed. Traversing a curve in its designated direction gives us a left and right hand side of the curve which we also denote by $+$ side and $-$ side. Γ^0 is obtained by removing the finitely many points of self-intersection from Γ and we denote the jump matrix corresponding to the part Γ_j of the contour by

$$G_j = G|_{\Gamma^0 \cap \Gamma_j}.$$

and call the set of all tuples (Γ_j,G_j) the data of a RHP. This data determines a holomorphic function $\Phi : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(m, \mathbb{C})$ satisfying

$$\begin{aligned} &\Phi^+(z) = \Phi^-(z)G(z) \qquad \text{for } z \in \Gamma^0, \\ &\Phi(\infty) = \mathrm{I}. \end{aligned}$$

Here, $\Phi^{\pm}(z)$ denotes the non-tangential limit of $\Phi(z')$ as $z' \to z$ from the positive respectively negative side of the contour. The positive side is also the left hand side of the contour and analogous the negative side is also the right hand side. Existence and uniqueness of the solution Φ can be shown by using some analytic properties of the jump function G and symmetries in the jump function and the contour, see e.g. (Fokas, Its, Kapaev and Novokshenov 2006, p. 104ff) for an overview of the corresponding theory. Under certain regularity assumptions, solving the RHP is equivalent to solving the singular integral equation

$$\Phi^-(z) = I + \frac{1}{2\pi i} \int_\Gamma \frac{\Phi^-(y)(G(y) - I)}{y - z^-} \, dy. \tag{1.1}$$

Thereby the solution of the RHP is then given by

$$\Phi(z) = I + \frac{1}{2\pi i} \int_\Gamma \frac{\Phi^-(y)(G(y) - I)}{y - z} \, dy.$$

In the scalar case ($m = 1$) or in case G is Abelian

$$G(z_1)G(z_2) = G(z_2)G(z_1) \quad \forall z_{1,2} \in \Gamma$$

there exists the explicit integral representation

$$\Phi(z) = \exp\left(\frac{1}{2\pi i} \int_\Gamma \frac{\log G(y)}{y - z} \, dy \right) \tag{1.2}$$

for the solution of a RHP (Fokas et al. 2006). In many cases the full solution $\Phi(z)$ is not of particular interest, but instead a derived quantity as e.g. the residue at ∞ is of interest

$$\operatorname*{res}_{z=\infty} \Phi(z) = \lim_{z \to \infty} z(I - \Phi(z)).$$

[2]A simple smooth curve is a curve, which does not intersect itself and is given by a continuous map.

Example 1: modified Korteweg–de Vries

A relative simple example is the RHP for modified Korteweg–de Vries (mKDV) equation

$$\begin{aligned} y_t - 6y^2 y_x + y_{xxx} &= 0 \quad \text{for } x \in \mathbb{R}, t \geq 0 \\ y(x, t = 0) &= y_0(x). \end{aligned}$$

According to (Deift and Zhou 1993), the corresponding RHP consists of the contour $\Gamma = \mathbb{R}$ (from $-\infty$ to $+\infty$) and the jump matrix

$$G(z; x, t) = \begin{pmatrix} 1 - r(z)r(-z) & -r(-z)e^{-\theta(z;x,t)} \\ r(z)e^{\theta(z;x,t)} & 1 \end{pmatrix}$$

with the phase function

$$\theta(z; x, t) = i(2zx + 8z^3 t)$$

and $r(z)$ being a function of the Schwartz space satisfying

$$\sup_{z \in \mathbb{R}} |r(z)| < 1.$$

r is also called the reflection coefficient. Thereby the RHP also depends on regularity conditions on $y_0(x)$. The solution of the ODE can be obtained from the solution of the RHP using the connection

$$y(x, t) = 2i \lim_{z \to \infty} z \, \Phi_{1,2}(z).$$

Example 2: Painlevé II

Another example is the RHP representing the Painlevé II equation

$$u_{xx} = xu + 2u^3.$$

As it is a second-order ODE, its general solution $u(x) = u(x; s_1, s_2)$ in the complex domain will depend on two independent complex parameters s_1 and s_2. These parameters also appear in the RHP which is given as follows, see also (Fokas et al. 2006). The contour consists of the six rays (see Fig. 1.1)

$$\Gamma_j = \{re^{i\pi(2j-1)/6} : r \geq 0\} \qquad (j = 1, \ldots, 6),$$

and the jump matrices

$$G_j(z) = \begin{cases} \begin{pmatrix} 1 & s_j e^{-\theta(z)} \\ 0 & 1 \end{pmatrix} & j \text{ even}, \\ \begin{pmatrix} 1 & 0 \\ s_j e^{+\theta(z)} & 1 \end{pmatrix} & j \text{ odd}, \end{cases}$$

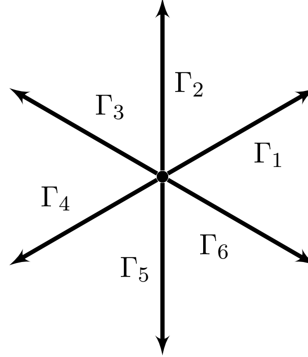


Figure 1.1.: The six rays Γ_j of the RHP associated with the Painlevé II ODE.

with the phase function

$$\theta(z) = \frac{8i}{3} z^3 + 2ixz.$$

The 6 parameters s_j $(j = 1, \ldots, 6)$ of the jump matrices have to satisfy

$$s_1 - s_2 + s_3 + s_1 s_2 s_3 = 0, \quad s_4 = -s_1, \quad s_5 = -s_2, \quad s_6 = -s_3,$$

so by fixing s_1 and s_2 all parameters are already well defined. The solution Φ of the RHP yields

$$u(x; s_1, s_2) = -2 \operatorname*{res}_{z=\infty} \Phi_{1,2}(z) = 2 \lim_{z \to \infty} z\, \Phi_{1,2}(z).$$

We should note here, that the independent variable x of the ODE is just a parameter of the phase function θ in the RHP, whoose independent variable is z. Therefore the RHP enables a pointwise evaluation of the Painlevé II function $u(x; s_1, s_2)$.

Condition of Problems

As we will discuss the condition of RHPs and other problems throughout this thesis, we would like to briefly recall its definition as e.g. given in (Deuflhard 2003). The condition describes the error amplification between input and output data. If f is the problem we want to compute, x is the input data and ϵ is the error of the input data, then the absolute condition number κ_{Abs} is defined as the smallest number satisfying

$$\|f(x + \epsilon) - f(x)\| \leq \kappa_{\text{Abs}} \|\epsilon\| \quad \text{for } \epsilon \to 0.$$

Likewise the relative condition number κ_{Rel} is defined as the smallest number satisfying

$$\|f(x + \epsilon) - f(x)\| / \|f(x)\| \leq \kappa_{\text{Rel}} \|\epsilon\| / \|x\| \quad \text{for } \epsilon \to 0.$$

Unless stated otherwise, we always consider the relative condition number which we will denote by κ. Given a stable numerical method it estimates the error caused by round-off in the last significant digit of the output data by

$$\text{\# loss of significant digits} \approx \log_{10} \kappa$$

In general, no numerical method is able to calculate an accurate solution for a badly conditioned system, i. e. a problem with a large condition number. It can be possible though that additional structure of the problem can be exploited to cast the problem into one that is not badly conditioned.

Due to the lack of an explicit formula for the condition number of a RHP, we are not able to directly compute it for a given RHP. Instead we approximate it throughout this thesis in the following way. We apply the solver from (Olver 2011b) to the RHP using 20 collocation points for each curve of the contour and compute the condition number of the linear system created by the solver. Though this is just a rough approximation of the condition number of the RHP, it is good enough for our purposes as we are only interested in its magnitude.

Nonlinear Steepest Descent

We presented both example RHPs in their original or vanilla form. Though this form is relatively simple, it is most of the time not the most useful one to either derive analytical results or to calculate a numerical solution. Usually one of two numerical problems occurs for the original form. Either the numerical solution converges very slowly or the solver encounters a badly conditioned linear system. The first problem occurs for example for the mKDV RHP whereas the second one occurs for the Painlevé II RHP. Both problems have been successfully resolved by deforming the contour using the method of nonlinear steepest descent (NSD), e.g. Olver and Trogdon (2012) use it for both the Painlevé II and mKDV RHPs. How big the impact of such a deformation on the convergence speed and the condition number is, can be observed in Fig. 1.2 for the mKDV RHP and in Fig. 1.3 for the Painlevé II RHP. The contours in these figures as well as all other colored illustrations of RHPs throughout this thesis are created using the following style.

Visualization Style 1 (RHP Contours)**.**

- *The color encodes* $||G_j(z) - I||_F$ *along* Γ_j *with a logarithmic scale.*
- *The values of* $||G_j(z) - I||_F$ *are truncated to* $[10^{-16}, 10^{16}]$.
- *We use the color coding green* $\approx 10^{-16}$, *yellow* $\approx 10^0$ *and red* $\approx 10^{16}$.
- *The blue points indicate the location of the stationary points of the phase function in* G.

Originally NSD was developed in (Deift and Zhou 1993) for the asymptotic analysis of RHPs, i.e. the behaviour of $\Phi(z)$ for $x \to \infty$, with x being a parameter of the jump

function G. We will not discuss in detail how NSD works, but its basic idea is as follows. Just as for the two examples we have presented, the jump matrices of RHPs are often of the form

$$G_j(z) = \begin{pmatrix} 1 & e^{\pm\theta(z)} \\ 0 & 1 \end{pmatrix} \quad , \quad G_j(z) = \begin{pmatrix} 1 & 0 \\ e^{\pm\theta(z)} & 1 \end{pmatrix}$$

or a product of these two forms. Thereby it is common for θ to be a holomorphic complex function. The goal of the nonlinear steepest descent method is to deform the contour such that for each part Γ_j holds

$$\operatorname{Re}\theta(z) \geq 0, \operatorname{Re}\theta(z) \to +\infty \text{ along } z \in \Gamma_j \text{ if } G_j \text{ contains } e^{-\theta(z)}$$
$$\operatorname{Re}\theta(z) \leq 0, \operatorname{Re}\theta(z) \to -\infty \text{ along } z \in \Gamma_j \text{ if } G_j \text{ contains } e^{+\theta(z)}$$

A contour satisfying these conditions yields jump matrices G_j decaying exponentially fast to the identity matrix[3] along Γ_j.

Further examples where NSD is used as a preconditioning step before a numerical solution is calculated can be found in (Trogdon and Olver 2012) [nonlinear Schrödringer RHP], (Trogdon, Olver and Deconinck 2012) [KDV and mKDV RHP] and (Olver and Trogdon 2014) [orthogonal polynomial RHP]. The results in these papers indicate that preconditioning a RHP by deforming it is a mandatory step before an accurate numerical solution can be calculated. The only exception so far are cases with small values of the involved parameters. For example for the Painlevé II RHP accurate results can be obtained for about $|x| \leq 5$. Deriving these deformations is not an easy task, as it requires quite some expert knowledge about NSD. Furthermore, there is usually not one deformation for a RHP but several. For example for mKDV there are three different deformations. Which of them has to be used depends on the parameters x and t.

Main Goal

We want to develop a preconditioning algorithm for RHPs which works as a "black box". It takes a RHP as input and without any further information form the user, it computes a deformed RHP which is well conditioned and yields a fast converging solution. Thus it completely eliminates any manual steps currently involved in the preconditioning step. What we present in this thesis is a "proof of concept" for such an algorithm. We consider a subset of all the tools available for NSD, develop an algorithm using this subset and verify that it is able to successfully precondition RHPs for which analytically derived deformations exists that also use just this subset of tools.

To give just a glimpse of how such and algorithm could work, we consider the following. NSD is about finding a curve Γ_j such that $G_j \to I$ is as fast as possible along Γ_j. This problem can be discretized by finding the shortest path P_j in a graph with edgeweights given by integrating $||G_j - I||$ along the edges. So if we can find for each part Γ_j a system

[3] $G_j(z) = I$ means effectively that there is no jump for Φ at z

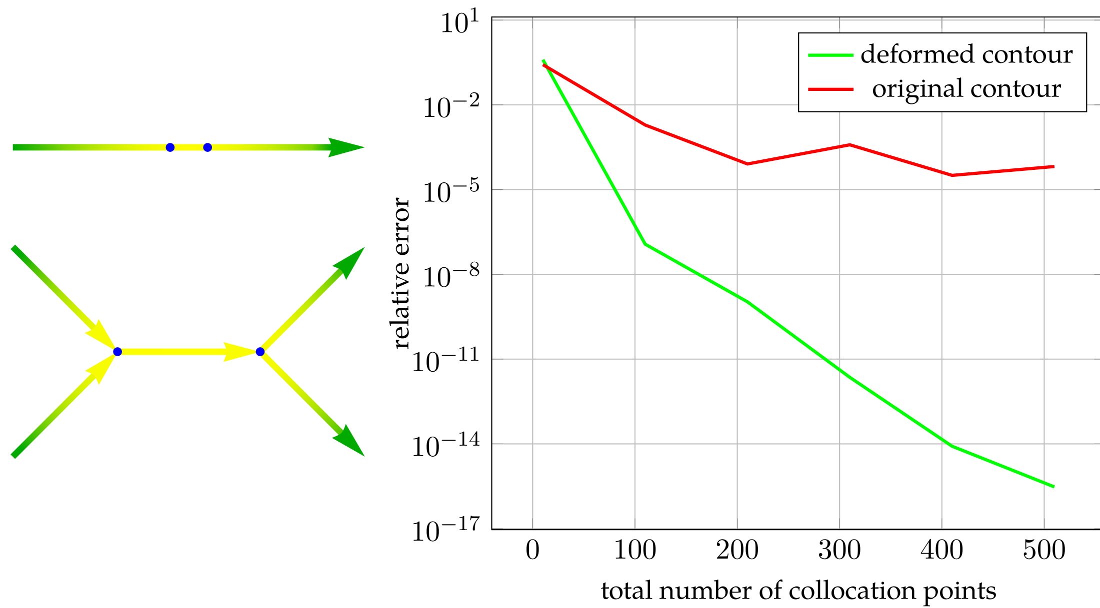


Figure 1.2.: Visualization according to style 1; *Left*: Original (top) and deformed contour (bottom) of the mKDV RHP for $t = 1$,$x = -5$ and $r(z) = \frac{1}{2}e^{-z^2}$; the blue points at the stationary points of the phase function θ (i. e. points x satisfying $\theta'(x) = 0$) also visualize the scaling applied to the top plot, which gives an impression how fast $G \to I$ in the deformed contour compared to the original. The condition number κ is about 20 for both contours. *Right*: The numerical solution of the deformed contour converges a lot faster than the one of the original contour. The reference solution, which we use to determine the error, is the numerical solution of the deformed contour calculated with 600 collocation points.

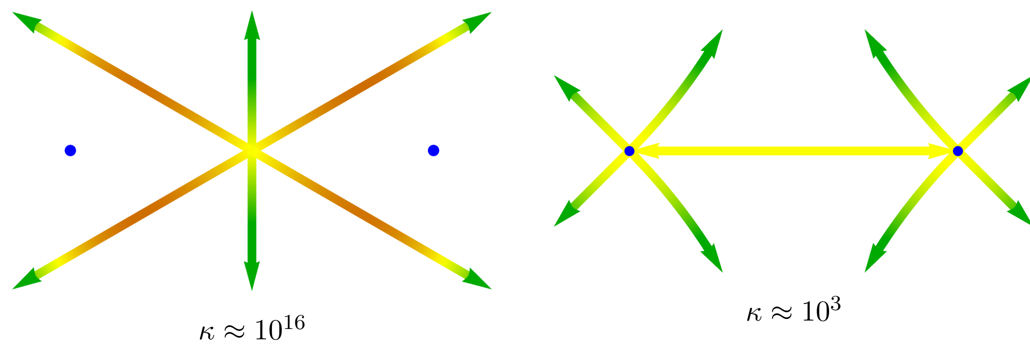


Figure 1.3.: Visualization according to style 1; original (left) and deformed (right) contour of the Painlevé II RHP for $s_1 = 1, s_2 = 2$ and $x = -20$; Deforming the contour has a quite big influence on the condition number. The original contour does not allow to calculate even a single accurate digit of the solution whereas we can expect 13 accurate digits for the deformed contour.

of shortest paths P_j in a way such that the union of these paths yield a valid deformation of the corresponding RHP, then this system of paths should resemble a deformation obtained by NSD.

Outline

Chapter 2 As the example RHPs we presented are "just" equivalents of ODEs / PDEs , the question might arise whether all the work involved for calculating numerical solutions of RHPs is actually worth the effort or if using standard solvers for ODEs / PDEs is not an easier approach. Therefore we provide a motivational example of a problem, which is very challenging to solve in its ODE formulation. Solving it in a RHP setting should be easier as it avoids the main problem which occurs in ODE formulation.

Chapter 3 We present the general process of optimizing a contour by casting it to the problem of finding the shortest path in a graph subject to topological constraints. In order to prevent the main ideas being hidden behind technical details, we do not yet optimize the contours of RHPs. Instead we consider the problem of optimizing the contour of a contour integral. To be specific, we use the Cauchy's integral formula to illustrate our ideas. As RHPs are equivalent to singular integral equations and in the scalar case can even be solved by contour integrals, this seems to be a suitable simplification of our main goal.

Chapter 4 By extending these ideas from the previous chapter to RHPs, we develop a "proof of concept" algorithm, which optimizes the contour of RHPs. It optimizes the contour with regard to reducing its condition number. We call it "proof of concept" as it does not support all the tools available for NSD and therefore its use is limited to deformations which do not require the omitted tools. The algorithm is presented with great detail.

Chapter 5 We evaluate the performance of our algorithm by comparing automatically computed deformations with analytically derived ones. For the examples we present here, both ways of obtaining deformations yield very similar contours and are equally or very similar effective for preconditioning RHPs.

Chapter 6 As already stated our "proof of concept" algorithm has a few limitations. We discuss these limitations and provide an outlook how the algorithm could be improved in future work.

Chapter 7 Finally, we provide a short summary of our results.

Appendix A As a little bonus for the interested reader, we present a short documentation (including examples) for our implementation of the contour deformation algorithm for RHPs discussed in § 4. This implementation was also used to to calculate the results shown in § 5. It is available at `https://github.com/tauu/AutoDeform` .

2. Motivation

In the two examples for RHPs, we have given in the introduction, the RHP corresponded to a 1 dimensional ODE (Painlevé II) respectively a PDE (mKDV). If we view these problems just from the points of numerical analysis one might ask oneself if there is an advantage in solving them using the RHP formulation instead of using the wide variety of numerical solvers, which are already available for ODEs and PDEs. To motivate the numerical study of RHPs in this chapter we give an example of an ODE, which can be solved using standard methods, but doing so is very challenging. The main numerical problem which we encounter thereby would not appear in the RHP formulation of the ODE. We would like to note here, that not all RHPs have a corresponding ODE / PDE and this is just one reason to be interested in the numerical study of RHPs.

2.1. Problem Description

We consider the BVP given by the Painlevé V ODE

$$(x\sigma_{xx})^2 = (\sigma - x\sigma_x - 2\sigma_x^2 + (2k+n)\sigma_x)^2 - 4\sigma_x^2(\sigma_x - k)(\sigma_x - k - n) \tag{2.1}$$

and the boundary conditions

$$\begin{aligned} \sigma(x) &\simeq \tfrac{k}{(n+1)!}\binom{k+n}{n} x^{n+1} && \text{for} \quad x \to 0 \\ \sigma(x) &\simeq k(x-n) + \tfrac{k^2 n}{x} && \text{for} \quad x \to \infty \end{aligned} \tag{2.2}$$

with $k, n \in \mathbb{N}$ and $\sigma : [0,\infty) \to \mathbb{R}$. We want to calculate a numerical solution on the whole interval $[0,\infty)$ which is as accurate as possible with IEEE double-precision arithmetic. This particular BVP appears in Random Matrix Theory, see (Tracy and Widom 1994).

To simplify the notation we define the following functions.

$$\begin{aligned} \sigma_0^1(x) &= \frac{k}{(n+1)!}\binom{k+n}{n} x^{n+1} \\ \sigma_\infty^1(x) &= k(x-n) + \frac{k^2 n}{x} \end{aligned} \tag{2.3}$$

σ_0^1 and σ_∞^1 are the first terms of the asymptotic expansions of σ for $x \to 0$ respectively $x \to \infty$. Correspondingly we will use σ_0^j and σ_∞^j to refer to the sum of the first i terms of the asymptotic expansion for $x \to 0$ respectively $x \to \infty$. We will call j the level of the asymptotic expansion.

The RHP corresponding to the Painlevé V ODE is available in (Fokas et al. 2006).

Outline

Our approach to solve this problem is to use a standard BVP solver. To make this possible we have to reduce the solution interval $[0, \infty)$ to a finite one. Therefore we calculate further levels of the asymptotic expansions σ_0 and σ_∞ in § 2.2. These are used to get accurate boundary conditions for a finite subset of $[0, \infty)$. Outside of this interval we calculate the solution by evaluating one of the asymptotic expansions. The next step is to construct initial values for the BVP solver. It turns out that constructing initial values for the whole interval is not feasible. Therefore the BVP is split into a left and right part in § 2.3. Initial values can be constructed for each of these parts. In § 2.4 we will discuss how we can calculate a reference solution. Finally we determine the accuracy of our numerical solution by comparing it against our reference solution in § 2.5.

The methods described in this chapter can in principal be used for any value of k and n. Nevertheless we will see later that we have to determine some parameters to be able to calculate a numerical solution. Determining the parameters for general k and n is difficult, therefore we perform numerical experiments just for one case: $k = 80$, $n = 40$.

2.2. Asymptotic Expansion

At first we define the operator R that maps a function to its residue in the Painlevé V equation (2.1).

$$\begin{aligned} \mathrm{R} : C^2(\mathbb{R}) &\to C(\mathbb{R}) \\ \sigma &\mapsto (x\sigma_{xx})^2 - (\sigma - x\sigma_x - 2\sigma_x^2 + (2k+n)\sigma_x)^2 \\ &\quad + 4\sigma_x^2(\sigma_x - k)(\sigma_x - k - n) \end{aligned} \tag{2.4}$$

Using this operator we can now prove the form of further levels of the asymptotic expansions.

Lemma 1. *If we define*

$$\sigma_0^j(x) = \sum_{i=1}^{j} w_i \frac{1}{(n+i)!} \binom{k+n}{n} x^{n+i}$$
$$\sigma_\infty^j(x) = \sum_{i=-j}^{1} w_i x^i$$

and fix the values $w_1 = k, w_0 = -kn, w_{-1} = k^2 n$ *then there is a unique way to choose the remaining constants* $w_i \in \mathbb{R}$ *such that*

$$\mathrm{R}\,\sigma_0^j(x) = \mathcal{O}(x^{2n+j}) \quad \text{for } x \to 0\,, \qquad \mathrm{R}\,\sigma_\infty^j(x) = \mathcal{O}(x^{-2-j}) \quad \text{for } x \to \infty \tag{2.5}$$

for all $j \in \mathbb{N}$.

Sketch of Proof. All calculations needed to prove this lemma are very long and technical but not difficult. Therefore instead of the actual calculations we will only show the Mathematica code for these calculations and their results. We start by defining two replacement patterns to extract the first/last term of a sum, the operator R and the asymptotic expansions.

```
(* utility patterns to extract the first/last term of a sum *)
extractfirst  = {
    HoldPattern[Sum[f_, {i, lo_, up_}]] :> (f /. i -> lo)
 };
extractlast  = {
    HoldPattern[Sum[f_, {i, lo_, up_}]] :> (f /. i -> up)
 };
(* operator R *)
R[s_] := (x D[D[s,x],x])^2 - (s - x D[s,x] -
         2 D[s,x]^2 + (2 k + n) D[s,x])^2 +
         4 D[s,x]^2 (D[s,x] - k) (D[s,x] - k - n) ;
(* σ_0^j *)
as[0,1] := k/(n + 1)! (Binomial[k + n, n]) x^(n + 1);
as[0,j_] := as[0,1] + Sum[w[i]/(n + i)!
              (Binomial[k + n, n]) x^(n + i) , {i, 2, j}];
(* σ_∞^j *)
as[Infinity,1]  := k*(x - n) + k^2 n/x;
as[Infinity,j_] := as[Infinity,1] + Sum[w[i] x^i, {i, -j, -2}]
```

Furthermore we add the global assumption that n and j are positive numbers.

```
$Assumptions = {n > 0, j > 0};
```

We prove the lemma by induction over j. But before we start with the induction, we note that for all $j \in \mathbb{N}$

$$\mathrm{R}\,\sigma_0^j(x) = \sum_{i=r}^{t} v_i x^i \qquad \mathrm{R}\,\sigma_\infty^j(x) = \sum_{i=p}^{q} u_i x^i \tag{2.6}$$

for some values of v_i, u_i, r, t, p, q, as σ_j^0 and σ_∞^j are sums of powers of x and R consists of finite differentiation, multiplications and additions of it. Now let us begin with the induction. At first we calculate the residue of both asymptotic expansions for $j = 1$ and determine the lowest, respectively the largest exponent of x.

```
Exponent[
  Expand @ R @ as[0,1],
  x,
  Min
] // Simplify
```

```
6    (* result: 1 + 2 n *)
7
8    Exponent[
9      Expand @ R @ as[Infinity,1],
10     x,
11     Max
12   ] // Simplify
13   (* result: -3 *)
```

Due to (2.6) the Mathematica results yield

$$\mathrm{R}\,\sigma_0^1(x) = \mathcal{O}(x^{2n+1}) \quad \text{for } x \to 0\,, \qquad \mathrm{R}\,\sigma_\infty^1(x) = \mathcal{O}(x^{-3}) \quad \text{for } x \to \infty$$

which is (2.5) for $j = 1$. For the induction step we assume that (2.5) holds for a specific $j \in \mathbb{N}$. Combined with (2.6) we get

$$\mathrm{R}\,\sigma_0^j(x) = \sum_{i=2n+j}^{t} v_i x^i \qquad \mathrm{R}\,\sigma_\infty^j(x) = \sum_{i=p}^{-2-j} u_i x^i$$

for some $v_i, u_i \in \mathbb{R}$ and $t, p \in \mathbb{N}$. To prove that (2.5) also holds for $j+1$ we use Mathematica to find a suitable choice for w_{j+1} and w_{-j-1} by solving

$$\mathrm{R}\,\sigma_0^{j+1} = \mathrm{R}\,\sigma_0^{j+1} - \mathrm{R}\,\sigma_0^j + \mathrm{R}\,\sigma_0^j = \mathrm{R}\,\sigma_0^{j+1} - \mathrm{R}\,\sigma_0^j + \sum_{i=2n+j}^{t} v_i x^i = \mathcal{O}(x^{2n+j+1})$$

$$\mathrm{R}\,\sigma_\infty^{j+1} = \mathrm{R}\,\sigma_\infty^{j+1} - \mathrm{R}\,\sigma_\infty^j + \mathrm{R}\,\sigma_\infty^j = \mathrm{R}\,\sigma_\infty^{j+1} - \mathrm{R}\,\sigma_\infty^j + \sum_{i=p}^{-2-j} u_i x^i = \mathcal{O}(x^{-2-j-1})$$

for v_{j+1} respectively u_{-j-1}. The corresponding code is

```
1    (* evaluate Rσ_0^{j+1} − Rσ_0^j + Σ_{i=2n+j}^t v_i x^i
2     * respectively Rσ_∞^{j+1} − Rσ_∞^j + Σ_{i=p}^{−2−j} u_i x^i ,
3     * split the result into summands and reduce every sum
4     * to its first / last term (= term with lowest / highest exponent)
5     * the last term of σ_0^{j+1}, σ_∞^{j+1} is split apart from the sum,
6     * to help Mathematica simplifying the result
7     *)
8    res0 = Level[
9      Expand[ R[as[0,j] + w[j+1]/(n+j+1)!  (Binomial[k+n, n]) x^(n+j+1) ]
10            - R @ as[0,j] + Sum[ v[i] x^i ,{i,2n+j,t}] ],
11     1] //. extractfirst;
12   resInf = Level[
```

```
13        Expand[ R[as[Infinity,j] + w[-j-1] x^(-j-1) ]
14              - R @ as[Infinity,j] + Sum[ u[i] x^i,{i,p,-2-j}] ],
15        1] //. extractlast;
16      (* lowest / highest exponent of x in res0 / resInf *)
17      le0   = Min @ Exponent[res0  ,x,Min] // Simplify
18      (* result: j + 2n *)
19      leInf = Max @ Exponent[resInf,x,Max] // Simplify
20      (* result: -2 - j *)
21      (* coefficient of x^(j+2n) / x^(-2-j)
22       * ( the Coefficient function does not work if
23       *  the exponent is a sum of symbols )
24       *)
25      c0   = res0   /. x^e_ :> If[e === le0  , 1, 0];
26      cInf = resInf /. x^e_ :> If[e === leInf, 1, 0];
27      (* determine all possible values of w_{j+1}, w_{-j-1}
28         which eliminates this coefficient *)
29      Solve[Total @ c0 == 0, w[j+1]]
30      (* result: w_{j+1} = -((n+1)!(j+n+1)! v_{j+2n}) / (2jk(n^2+n)(j+n+1) binom(k+n,n)^2) *)
31      Solve[Total @ cInf == 0, w[-j-1]]
32      (* result: w_{-j-1} = u_{-j-2} / (4k^2 n) *)
```

So if we know w_i with $|i| \leq j$ satisfying (2.5), then there is exactly one possible choice for w_{j+1}, w_{-j-1} such that

$$\mathrm{R}\,\sigma_0^{j+1}(x) = \mathcal{O}(x^{2n+j+1}) \quad \text{for } x \to 0\,, \qquad \mathrm{R}\,\sigma_\infty^{j+1}(x) = \mathcal{O}(x^{-2-j-1}) \quad \text{for } x \to \infty.$$

□

The lines 30 and 32 of the last code listing suggest an easy method to calculate the constants w_i. If we know the values of the constants w_i for $-j \leq i \leq j$ for which

$$\begin{aligned} \mathrm{R}\,\sigma_0^j(x) &= \sum_{i=2n+j}^{-2+3(j+n)} v_i x^i \\ \mathrm{R}\,\sigma_\infty^j(x) &= \sum_{i=-2-3j}^{-2-j} v_i x^i \end{aligned} \tag{2.7}$$

is satisfied with $v_i \in \mathbb{R}$ then w_{j+1} and w_{-j-1} are given by

$$\begin{aligned} w_{j+1} &= -\frac{v_{j+2n}(1+n)!(1+j+n)!}{2jk(1+j+n)(n+n^2)\binom{k+n}{n}^2} \\ w_{-j-1} &= \frac{v_{-2-j}}{4k^2 n} \end{aligned}$$

As both equations (2.7) are satisfied for $j = 1$ with the definitions given in (2.3) we can use the following recursive definition of w_j to calculate it.

```
1     w[j_?Negative] :=
2         w[j] = Coefficient[
3             Expand @ R @ as[Infinity, -j - 1],
4             x,
5             -1 + j
6         ] / (4 k^2 n)
7
8     w[j_?Positive] :=
9         w[j] = - ( Expand @ R @ as[0, j - 1]
10                /. x^(e_) :> If[e === 2 n + j - 1, 1, 0]
11            ) (1+n)! (j+n)!
12            / (2 (j-1) k (j+n) (n+n^2) Binomial[k+n, n]^2)
```

Remark. *Line 10 is essentially the same as* Coefficient [... , x, 2 n + j - 1] *but the* Coefficient *function does not work correctly if the requested exponent contains a symbol*[1]*. Therefore this workaround is used.*

As the time and memory consumption to calculate the next term of these asymptotic expansion increases quite fast with the level of the asymptotic expansion we were only able to calculate σ_0^j up to $j = 17$ and σ_∞^j up to $j = 42$.

2.3. Interval Splitting

The standard approach to solve the BVP given by (2.1) and (2.2) would be to apply a standard BVP solver. As no standard BVP solver is able to solve a BVP over an infinite interval like $[0, \infty]$, we have to reduce the interval to a finite one first. This is possible because we do not need to solve the ordinary differential equation in regions, where the asymptotic expansion approximates the solution σ up to machine precision. Consequently we can define new boundary conditions as follows:

$$\sigma(a) = \sigma_0^i(a) \qquad\qquad \sigma(c) = \sigma_\infty^j(c) \tag{2.8}$$

with $0 < a < c < \infty$ chosen such that

$$\left|\frac{\sigma(a) - \sigma_0^i(a)}{\sigma(a)}\right| < \text{eps} \qquad\qquad \left|\frac{\sigma(c) - \sigma_\infty^j(c)}{\sigma(c)}\right| < \text{eps}$$

holds with eps being the machine precision[2] and the interval $[a, c]$ is as small as possible. We will refer to the BVP defined by (2.1) and (2.8) with BVP_c. To solve it, we have tried

[1] Bug exists at least in Mathematica 7 - 10

[2] For double precision number we have eps $\approx 10^{-16}$.

to apply the following BVP solvers

- BVPSOL (version of 11.0.2006); multiple shooting method, available at `http://www-m3.ma.tum.de/Software/ODEHome`
- bvp4c (version of Matlab 2009a); collocation method integrated in Matlab
- bvp5c (version of Matlab 2009a); successor of bvp5c, integrated in Matlab
- bvp6c (version of 12.06.2006); improved version of bvp4c, available at `http://www.mathworks.com/matlabcentral/fileexchange/11315`
- sbvp (version 1.0); collocation method, available at `http://www.mathworks.de/matlabcentral/fileexchange/1464`
- a solver based on the chebops package (version v3_1111), available at `http://www.comlab.ox.ac.uk/projects/chebfun/chebops.html`.

(2.9)

It turns out that none of these solvers is successful in calculating a numerical solution. The main problem is that it is difficult to create good initial values for this BVP. For x close to a or c it is possible to use the asymptotic expansions to determine initial values, but for the major part of the interval $[a, c]$ we have no information about the solution and are therefore limited to use an interpolation method to create initial values. This is no easy task as can be seen in Fig. 2.1.

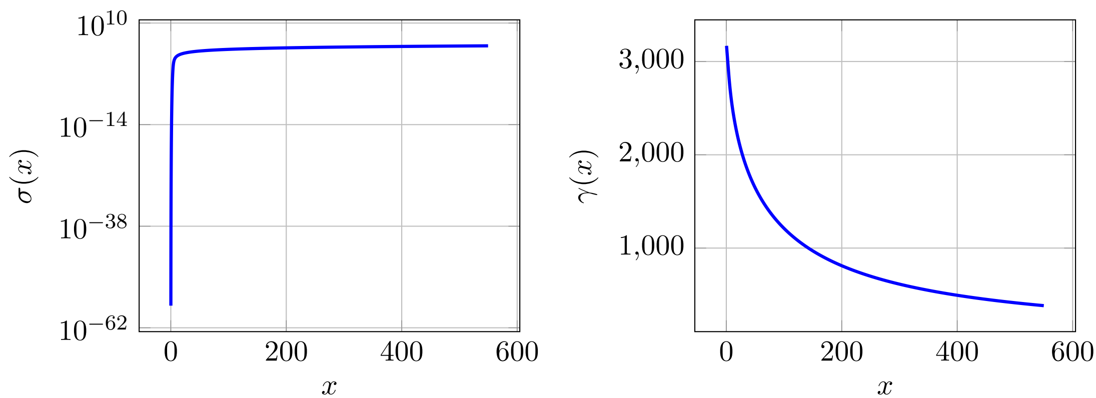


Figure 2.1.: Visualization of σ (left) and γ (right) for $k = 80, n = 40$; σ behaves completely different at both ends of the interval $[a, c]$ and the order of magnitude of σ is very different at both borders. γ should be far easier to approximate using an interpolation. (We will later on determine $a = 0.15$ and $c = 550$.)

Therefore we modify the BVP to make it easier to create initial values. By defining

$$\gamma(x) := \sigma(x) - k(x-n) \tag{2.10}$$

we obtain a function which can be approximated much easier by an interpolation than σ, see also Fig. 2.1 (right). Inserting definition (2.10) into the equations (2.1) and (2.8) yields the new BVP, BVP$_r$, given by

$$\begin{aligned}(x\gamma_{xx})^2 &= -4\gamma_x(k+\gamma_x)^2(-n+\gamma_x) + (s + (-2k+n-x)\gamma_x - 2\gamma_x^2)^2 \\ \gamma(d) &= \sigma_0^i(d) - k(d-n) \\ \gamma(e) &= \sigma_\infty^j(e) - k(e-n)\end{aligned} \tag{2.11}$$

with some constant $d < e$ and $d, e \in \mathbb{R}, i, j \in \mathbb{N}$, which have yet to be determined. For this problem almost all solvers in list (2.9) also do not yield a solution, the only exception is sbvp. Although sbvp can solve this BVP, there remains a problem. Using the numerical solution γ^N of (2.11), we can define

$$\sigma_r^N(x) = \gamma^N(x) + k(x-n)$$

and use it as an approximation for $\sigma(x)$. But as we can see in Fig. 2.2 there is an area around d where evaluating $\gamma^N(x) + k(x-n)$ will not yield a good approximation of σ due to substantial cancellation. As $\sigma_0^i(x)$ is also not a good approximation of σ in this area, we do not yet have an accurate solution of BVP$_c$.

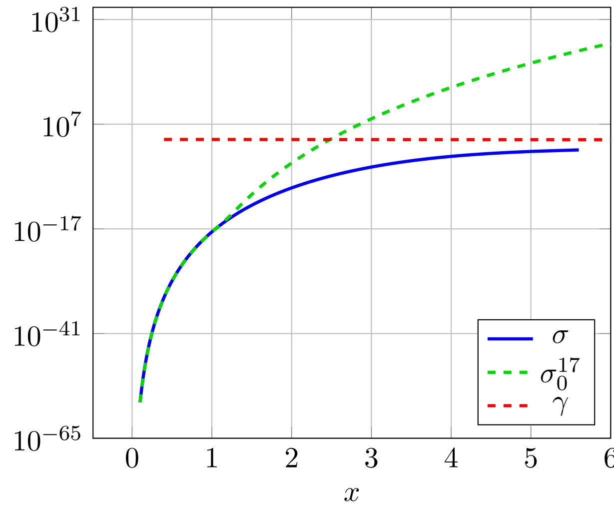


Figure 2.2.: Comparison of σ, γ, $\sigma_0^{17}(x)$ for x close to 0 for $k = 80$, $n = 40$

To fill this gap we introduce the additional BVP: BVP$_l$ given by (2.1) and

$$\sigma(a) = \sigma_0^i(a) \qquad \sigma(b) = \sigma_r^N(b)$$

where $b \in [a, c]$ is chosen such that it is the smallest point at which σ_r^N can be evaluated without subtractive cancellation. Once again we were only able to solve this BVP with sbvp.[3] The numerical solution of BVP_l will be denoted by σ_l^N. In summary we can now define the numerical solution σ^N of BVP_c as follows

$$\sigma^N(x) = \begin{cases} \sigma_0^i(x) & \text{for } 0 \le x \le a \\ \sigma_l^N(x) & \text{for } a < x \le b \\ \sigma_r^N(x) & \text{for } b < x < c \\ \sigma_\infty^j(x) & \text{for } c \le x < \infty \end{cases} \tag{2.12}$$

Obviously σ^N is only well defined if $c < e$, so we add this as an additional restriction for the choice of e. How the constants a, b, c, d, e, i and j are determined is described in the next section.

Parameter values for the case $k = 80$,$n = 40$

BVP_r: borders c,d,e and level j

At first we consider the left border d. As

$$\sigma_0^i(x) << \text{eps } k(x-n) \tag{2.13}$$

holds with eps being the machine precision in the whole region where $\sigma_0^i(x)$ is a good approximation of $\sigma(x)$ we can simplify the left boundary condition in (2.11) to

$$\gamma(d) = -k(d-n)$$

Therefore we have to choose d such that

$$\left| \frac{\sigma(d)}{k(d-n)} \right| \le \text{eps} .$$

The largest value we can choose is $d = 1.5$, see also Fig. 2.3.

Next we consider the right border e. Fig. 2.4 shows the relative difference between two consecutive levels of $\sigma_\infty^j(x) - k(x-n)$. According to the figure, $\sigma_\infty^{42}(x) - k(x-n)$ should approximate $\gamma(x)$ with machine precision for $x \ge 600$ and it does not seem likely that higher levels of σ_∞^j will yield a significantly lower value. So we set $j = 42$ and $e = 600$.

We have to verify that $c < e$ as otherwise (2.12) would not be well defined. Analogous to what we did for the border e, we plot the relative difference of consecutive levels of σ_∞^j and see that σ_∞^{42} approximates $\sigma(x)$ with machine precision for $c = 550 < e$. Consequently we set $c = 550$.

[3] A modified version of BVPSOL is also able to solve BVP_l, but it requires different initial values than the ones presented here.

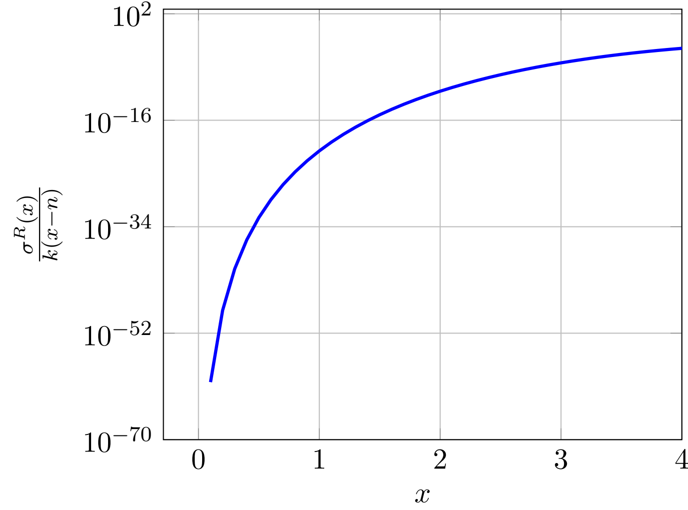


Figure 2.3.: Magnitude of $k(x-n)$ compared to $\sigma(x)$

BVP$_l$: borders a,b and level i

Analogous to the approach we used for the borders e and c, we compare the relative difference between two consecutive levels of σ_0^i, see Fig. 2.6, and determine $a = 0.15$ and $i = 17$. The remaining parameter b is determined later in § 2.5.

2.4. Reference Solution

To estimate the error of the numerical solution of the Painlevé V equation, we compare it against a reference solution. To ensure the accuracy of the reference solution we use Mathematica's ability to perform calculations with arbitrary precision. With a high enough precision, we will get a solution which is accurate up to machine precision and is used as a reference solution.

Unfortunately there are only some very basic BVP solvers available for Mathematica and none suits our problem very well. Therefore instead of solving the BVPs, we solve an equivalent initial value problem consisting of equation (2.1) and the initial values.

$$\begin{aligned}\sigma(x_1) &= \sigma_0^j(x_1)\\ \sigma_x(x_1) &= \frac{d}{dx}\sigma_0^j(x)|_{x=x_1}\end{aligned} \tag{2.14}$$

Thereby we use the built-in extrapolation solver of Mathematica, as it is the fastest built-in solver for multi machine precision calculations, according to the Mathematica documentation (Mathematica n.d.). As base integration method for the extrapolation solver we use the `LinearlyImplicitMidpoint` method. The level j is set to the highest level that we were able to calculate, which is 17. Furthermore a good choice for the initial

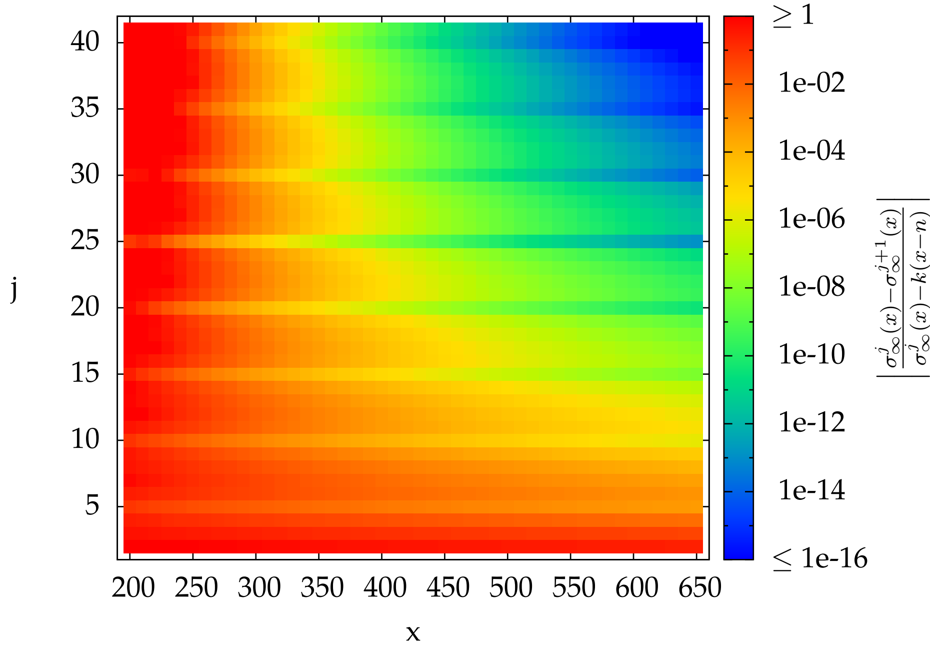


Figure 2.4.: Comparison of consecutive levels of $\sigma_\infty^j - k(x-n)$

point x_1 and the precision for the calculations has been determined by some experiments. The most interesting choices are shown in Tab. 2.1. x_2 is the point at which the solver essentially stopped due to a too small step size. As we can see, in general smaller values of x_1 yield larger values for x_2 if a sufficiently high precision is used. As the calculation time for one of the settings presented in Tab. 2.1 is already about one month even though we do not get close to c, it does not seem practical to calculate a reference solution for the whole interval $[a, c]$. Instead we use the best setting we found ($x_1 = 10^{-5}$ and 24-fold machine precision) and use the resulting solution as a reference solution for the interval $[10^{-5}, 65]$ and denote it by σ^R. For $x > 65$ the accuracy of this reference solution seems to be less than machine precision.

x_1	n-fold machine precision	x_2
10^{-5}	24	81.63
10^{-4}	20	63.19
10^{-5}	20	81.6
10^{-6}	20	58.22

Table 2.1.: We calculated reference solution with the setting presented in this table. Thereby x_1 is the initial point and x_2 the point at which the solution blows up if the solver uses the corresponding precision.

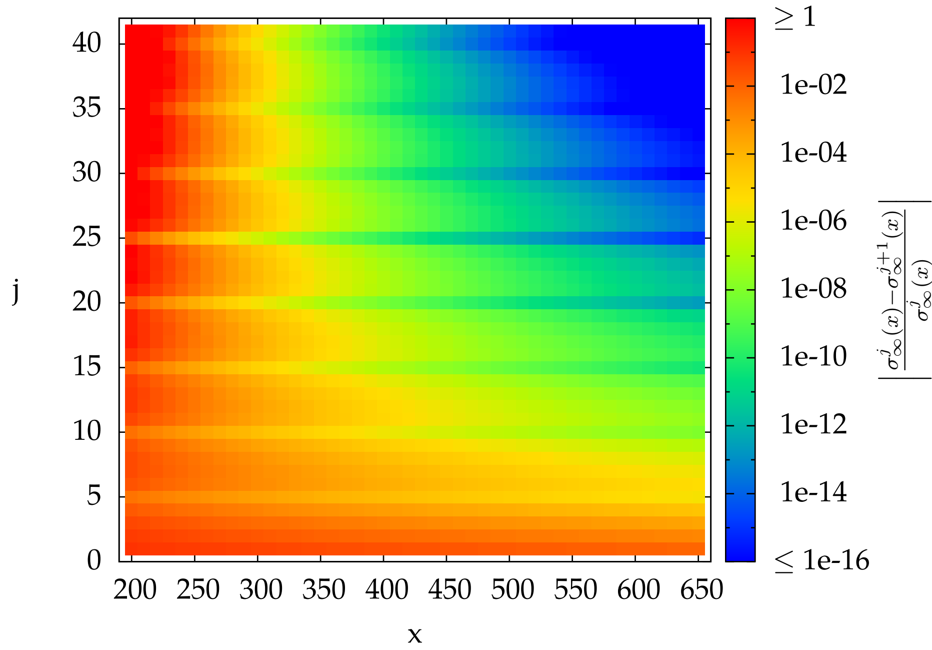


Figure 2.5.: Comparison of consecutive levels of σ_∞^j

2.5. Numerical Results

The results we present here are for the parameters $k = 80$, $n = 40$ and are calculated with the BVP solver `sbvp`. To determine the accuracy of the numerical solutions and initial values, we compare them to the reference solution in the interval $[0, 65]$.

BVP$_r$

To calculate a numerical solution we need initial values for the BVP solver. These are created using the interpolation polynomial p given by

$$\begin{aligned} & p \in \mathbb{P}_{r+l+1} \\ \frac{\mathrm{d}^i}{\mathrm{d}\mathrm{x}^i} p(x)\Big|_{x=d} &= \frac{\mathrm{d}^i}{\mathrm{d}\mathrm{x}^i} - k(x-n)\Big|_{x=d} \qquad \text{for } 0 \le i \le l \\ \frac{\mathrm{d}^i}{\mathrm{d}\mathrm{x}^i} p(x)\Big|_{x=e} &= \frac{\mathrm{d}^i}{\mathrm{d}\mathrm{x}^i} \sigma_\infty^{42}(x) - k(x-n)\Big|_{x=e} \qquad \text{for } 0 \le i \le r \end{aligned} \tag{2.15}$$

The best choice for the parameters d, e, l and r that we were able to find is

$$d = 0.4 \qquad l = 1$$
$$e = 550 \qquad r = 21.$$

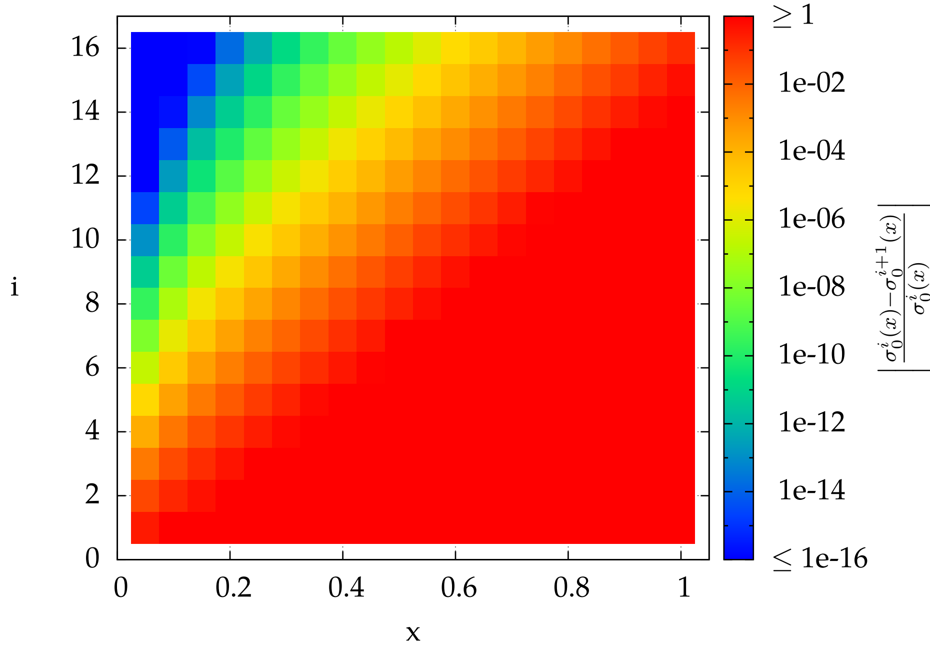


Figure 2.6.: Comparison of consecutive levels of σ_0^i

Though we have determined earlier that $e > 600$ is optimal, we set the border e to a lower value, because higher values yield a less accurate numerical solution. Initial values are created by sampling p with a step size of 0.1 for x lower than 12 and with a step size of 1 for x greater than 12. We would like to note here that even small alterations to this set of parameters can cause the BVP solver to not converge to a solution anymore. E.g. we did not get a solution for any value of r apart from the one we presented.

Fig. 2.7 shows the accuracy of the initial values, γ^N and $\sigma_r^N(x)$.

BVP$_l$

Just as for BVP$_r$ we need initial values for the BVP solver. Basically there are three different sources for initial values in this case. The first is $\sigma_0^{17}(x)$, the second is $\sigma_r^N(x)$ and the third is an interpolation between these two. Fig. 2.8 shows the accuracy of the first two. As we can see the first two options yield bad initial values between 1.0 and 1.7, but for the remaining interval always one source yields good initial values. Therefore we use the interpolation polynomial that satisfies the following conditions in the interval

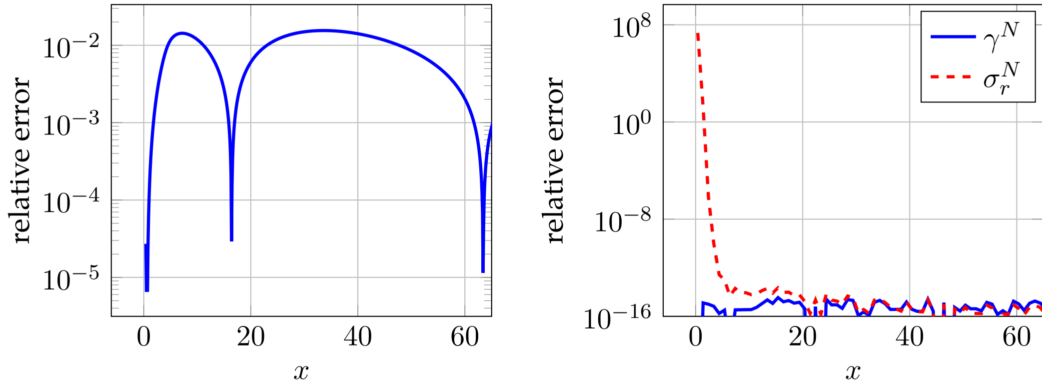


Figure 2.7.: *Left*: Accuracy of the initial values for BVP_r *Right*: Accuracy of γ^N and σ_r^N; The accuracy of σ_r^N decreases close to $x = 0$ due to numeric cancellation, but is very accurate for $x \geq 6$. Therefore, we choose $b = 6$.

$[1.0, 1.7]$ to get better initial values for this interval.

$$\begin{aligned} p &\in \mathbb{P}_3 \\ p(1.0) &= \log(\sigma_0^{17}(1.0)) \\ p(1.7) &= \log(\sigma_r^N(1.7)) \\ \frac{\mathrm{d}}{\mathrm{dx}} p(x)\Big|_{x=1.0} &= \frac{\mathrm{d}}{\mathrm{dx}} \log(\sigma_0^{17}(x))\Big|_{x=1.0} \\ \frac{\mathrm{d}}{\mathrm{dx}} p(x)\Big|_{x=1.7} &= \frac{\mathrm{d}}{\mathrm{dx}} \log(\sigma_r^N(x))\Big|_{x=1.7} \end{aligned}$$

In summary the initial values α are then given by

$$\alpha(x) = \begin{cases} \sigma_0^{17}(x) & \text{for} \quad x \leq 1.0 \\ \exp(p(x)) & \text{for} \quad 1.0 < x < 1.7 \\ \sigma_r^N(x) & \text{for} \quad 1.7 \leq x \leq 6 \end{cases}$$

This uncommon interpolation is chosen because $\sigma_0^{17}(1.0)$ and $\sigma_r^N(1.7)$ have very different orders of magnitude, which results in a poor approximation quality for common interpolation methods. In the interval $[0.15, 1.0]$ we use a step size of 0.001 and in the remaining interval a step size of 0.01. Fig. 2.8 shows the accuracy of the initial values and of the numerical solution σ_l^N calculated with them.

Combined Solution

Finally, Fig. 2.9 shows the accuracy of $\sigma^N(x)$ as well as a plot of σ. The solution has a very good accuracy of 14-15 significant digits in most regions for which we were able calculate a reference solution. Just for x close to 0 the accuracy is a bit lower.

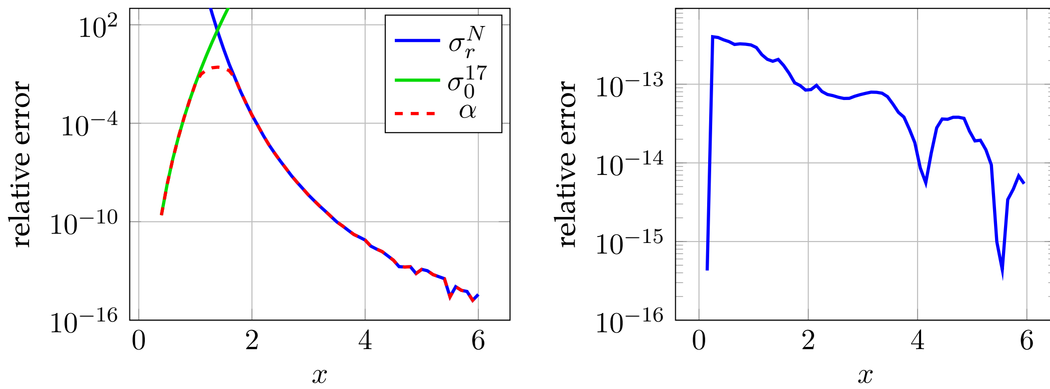


Figure 2.8.: *Left*: Accuracy of initial values for BVP$_l$. *Right*: Accuracy of σ_l^N.

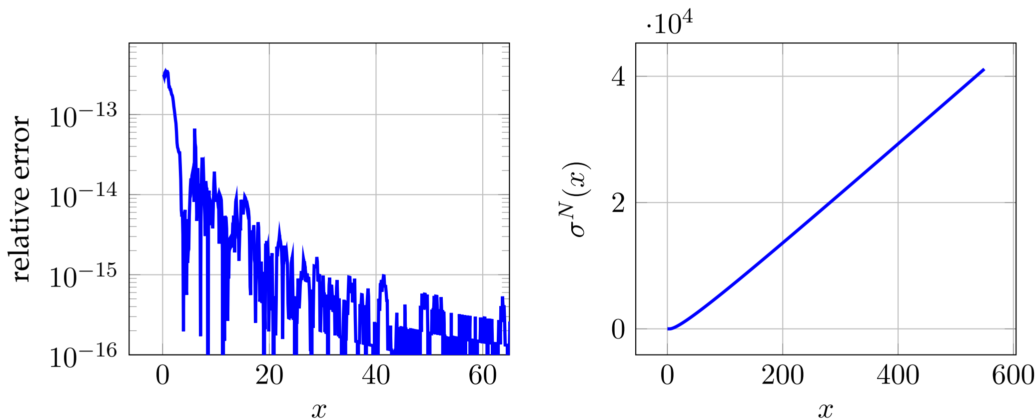


Figure 2.9.: *Left*: Accuracy of σ^N. *Right*: Plot of σ^N.

Conclusion

Though it is possible to compute a numerical solution to BVP$_c$ with an accuracy of 14-15 digits, this chapter demonstrates that it is a very challenging task and it involves quite a lot of work. If we recall from § 2.4, that even 24-fold machine precision was not enough to solve the initial value problem for Painlevé V (2.14) in the interval $[0, 80]$, we observe that this initial value problem has a condition number of 10^{384} for this interval with respect to the initial conditions. As we have already stated in the introduction, the RHP formulation allows a pointwise evaluation of the corresponding ODE and thus avoids the enormous error amplification occurring for initial value problems. Furthermore we do not need initial values or further terms of the asymptotic expansion to calculate a numerical solution of the RHP. What we would need though is a deformation to precondition the Painlevé V RHP, but as we will see in the next chapters, there is a good chance that this could be determined automatically by an algorithm.

3. Toy Problem: Cauchy's Integral Formula

3.1. Introduction

Before we start discussing how the contour of RHPs can be optimized, we will first discuss an easier problem which demonstrates the basic ideas in a more simple way. As we already mentioned in § 1 a RHP is equivalent to a singular contour integral equation. So an obvious simplification of finding an optimal contour for a RHP would be the problem finding an optimal contour for the evaluation of a contour integral. And that is exactly what we will do in this chapter. Thereby we pick a particular contour integral, namely the Cauchy integral.

This results presented in this chapter have been published in (Bornemann and Wechslberger 2013) and it is for the most part identical to the paper.

To escape from the ill-conditioning of difference schemes for the numerical calculation of high-order derivatives, numerical quadrature applied to Cauchy's integral formula has on various occasions been suggested as a remedy (for a survey of the literature, see Bornemann 2011). To be specific, we consider a function f that is holomorphic on a complex domain $D \ni 0$; Cauchy's formula gives[1]

$$f^{(n)}(0) = \frac{n!}{2\pi i} \int_\Gamma z^{-n-1} f(z)\, dz \tag{3.1}$$

for each cycle $\Gamma \subset D$ that has winding number $\operatorname{ind}(\Gamma; 0) = 1$. If Γ is not carefully chosen, however, the integrand tends to oscillate at a frequency of order $O(n^{-1})$ with very large amplitude (Bornemann 2011, Fig. 4). Hence, in general, there is much cancelation in the evaluation of the integral and ill-conditioning returns through the backdoor. The condition number of the integral is (Deuflhard 2003, Lemma 9.1)

$$\kappa(\Gamma, n) = \frac{\int_\Gamma |z|^{-n-1} |f(z)|\, d|z|}{\left|\int_\Gamma z^{-n-1} f(z)\, dz\right|}$$

and Γ should be chosen as to make this number as small as possible. Equivalently, since the denominator is, by Cauchy's theorem, independent of Γ, we have to minimize

$$d(\Gamma) = \int_\Gamma |z|^{-n-1} |f(z)|\, d|z|. \tag{3.2}$$

[1] Without loss of generality we evaluate derivatives at $z = 0$.

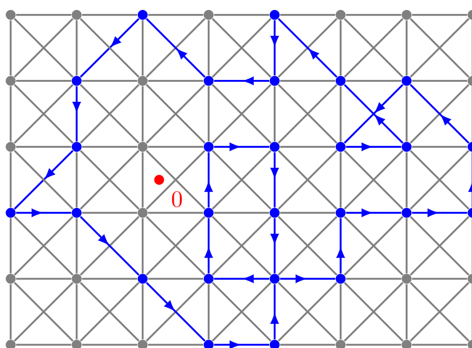


Figure 3.1.: Path Γ with $\operatorname{ind}(\Gamma;0) = 1$ in a grid-graph of step size h.

Bornemann (2011) considered circular contours of radius r; he found that there is a unique $r_* = r(n)$ solving the minimization problem and that there are different scenarios for the corresponding condition number $\kappa_*(n)$ as $n \to \infty$:

- $\kappa_*(n) \to \infty$, if f is in the Hardy space[2] H^1;
- $\limsup_{n\to\infty} \kappa_*(n) \leqslant M$, if f is an entire function of completely regular growth which satisfies a non-resonance condition of the zeros and whose Phragmén–Lindelöf indicator possesses M maxima (a small integer).

Hence, though those (and similar) results basically solve the problem of choosing proper contours for entire functions, much better contours have to be found for the class H^1.

In this chapter, we solve the contour optimization problem within the more general class of grid paths of step size h (see Fig. 3.1; we allow diagonals to be included) as they are known from Artin's proof of the general, homological version of Cauchy's integral theorem (Lang 1999, IV.3). Such paths are composed from horizontal, vertical and diagonal edges taken from a (bounded) grid $\Omega_h \subset D$ of step size h. Now, the weight function (3.2), being *additive* on the abelian group of path chains[3], turns the grid Ω_h into an edge-weighted graph such that each optimal grid path W_* becomes a *shortest enclosing walk* (SEW); "enclosing" because we have to match the winding number condition $\operatorname{ind}(W_*;0) = 1$. An efficient solution of the SEW problem for embedded graphs was found by Provan (1989) and serves as a starting point for our work.

[2]The Hardy space $H^1(B_R)$ is the set of all function f which are holomorphic in the disc B_R and satisfy

$$\sup_{0<r<R} \frac{1}{2\pi} \int_0^{2\pi} |f(re^{i\pi\theta})|\, d\theta < \infty.$$

[3]We use the standard choice of joining paths for the group action.

Outline

In § 3.2 we discuss general embedded graphs in which an optimal contour is to be searched for; we discuss the problem of finding a shortest enclosing walk and recall Provan's algorithm. In § 3.3 we discuss some implementation details and tweaks for the problem at hand. Finally, in § 3.4 we give some numerical examples; these can easily be constructed in a way that the new algorithm outperforms, by orders of magnitude, the optimal circles of Bornemann (2011) with respect to accuracy and the direct symbolic differentiation with respect to efficiency.

3.2. Contour Graphs and Shortest Enclosing Walks

By generalizing the grid Ω_h, we consider a finite graph $G = (V, E)$ *embedded to* D, that is, built from vertices $V \subset D$ and edges E that are smooth curves connecting the vertices within the domain D. We write uv for the edge connecting the vertices u and v; by (3.2), its weight is defined as

$$d(uv) = \int_{uv} |z|^{-n-1} |f(z)|\, d|z|. \tag{3.3}$$

A *walk* W in the graph G is a *closed* path built from a sequence of adjacent edges, written as (where $\dot{+}$ denotes joining of paths)

$$W = v_1 v_2 \dot{+} v_2 v_3 \dot{+} \cdots \dot{+} v_m v_1;$$

it is called *enclosing* the obstacle 0 if the winding number is $\operatorname{ind}(W; 0) = 1$. The set of all possible enclosing walks is denoted by Π. As discussed in § 3.1, the condition number is optimized by the shortest enclosing walk (not necessarily unique)

$$W_* = \operatorname*{argmin}_{W \in \Pi} d(W)$$

where, with $W = v_1 v_2 \dot{+} v_2 v_3 \dot{+} \cdots \dot{+} v_m v_1$ and $v_{m+1} = v_1$, the *total weight* is

$$d(W) = \sum_{j=1}^{m} d(v_j v_{j+1}).$$

The problem of finding such a SEW was solved by Provan (1989): the idea is that with $\mathcal{P}_{u,v}$ denoting a shortest path between u and v, any shortest enclosing walk $W_* = w_1 w_2 \dot{+} w_2 w_3 \dot{+} \cdots \dot{+} w_m w_1$ can be cast in the form (Provan 1989, Thm. 1)

$$W_* = \mathcal{P}_{w_1, w_j} \dot{+} w_j w_{j+1} \dot{+} \mathcal{P}_{w_{j+1}, w_1}$$

for at least one j. Hence, any shortest enclosing walk W_* is already specified by one of its vertices and one of its edges; therefore

$$W_* \in \tilde{\Pi} = \{\mathcal{P}_{u,v} \dot{+} vw \dot{+} \mathcal{P}_{w,u} : u \in V, vw \in E\}.$$

Provan's algorithm finds W_* by, first, building the finite set $\tilde{\Pi}$; second, by removing all walks from it that do not enclose $z = 0$; and third, by selecting a walk from the remaining candidates that has the lowest total weight. Using Fredman and Tarjan's (1987) implementation of Dijkstra's algorithm to compute the shortest paths $\mathcal{P}_{u,v}$, the run time of the algorithm is known to be (Provan 1989, Corollary 2)

$$O(|V|\,|E| + |V|^2 \log |V|). \tag{3.4}$$

3.3. Implementation Details

We restrict ourselves to graphs Ω_h given by finite square grids of step size h, centered at $z = 0$ — with all vertices and edges removed that do not belong to the domain D. Since Provan's algorithm just requires an embedded graph but not a planar graph, we may add the diagonals of the grid cells as further edges to the graph (see Fig. 3.1).[4] For such a graph Ω_h, with or without diagonals, we have $|V| = O(h^{-2})$ and $|E| = O(h^{-2})$ so that the complexity bound (3.4) simplifies to

$$O(h^{-4} \log h^{-1}).$$

3.3.1. Edge Weight Calculation

Using the edge weights $d(uv)$ on Ω_h requires to approximate the integral in (3.3). Since not much accuracy is needed here,[5] a simple trapezoidal rule with two nodes is generally sufficient:

$$\begin{aligned} d(uv) &= \int_{uv} |z|^{-(n+1)} |f(z)| d|z| \\ &= \frac{|u-v|}{2} \left(d(u) + d(v)\right) + O(h^3) = \tilde{d}(uv) + O(h^3) \end{aligned}$$

with the *vertex weight*

$$d(z) = |z|^{-(n+1)} |f(z)|. \tag{3.5}$$

Although $\tilde{d}(uv)$ will typically have an accuracy of not more than just a few bits for the rather coarse grids Ω_h we work with, we have not encountered a single case in which a more accurate computation of the weights would have resulted in a different SEW W_*.

[4] These diagonals increase the number of possible slopes which results, e.g., in improved approximations of the direction of steepest descent at a saddle point of $d(z)$ (Bornemann 2011, §9) or in a faster U-turn around the end of a branch-cut, see Fig. 3.5. The latter case leads to some significant reductions of the condition number, see Fig. 3.4.

[5] Recall that optimizing the condition number is just a question of order of magnitude but not of precise numbers. Once the contour Γ has been fixed, a much more accurate quadrature rule will be employed to calculate the integral (3.1) itself, see § 3.3.5.

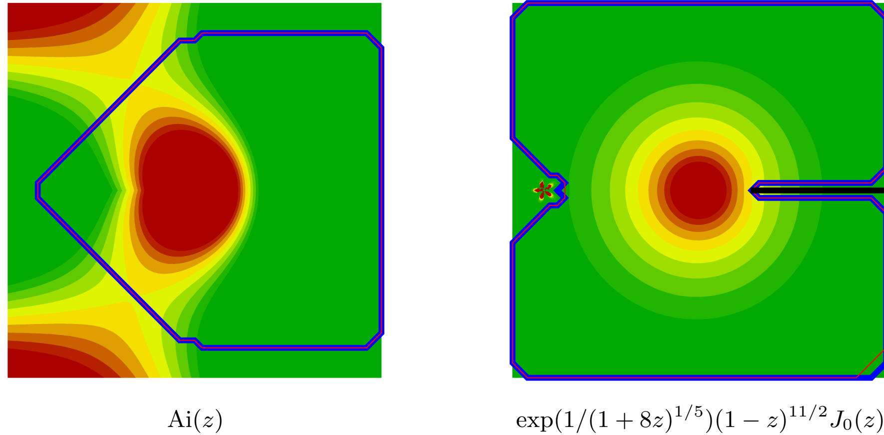


Figure 3.2.: W_* (red) vs. W_{v_*} (blue): the color coding shows the size of $\log d(z)$; with red for large values and green for small values. The smallest level shown is the threshold, below of which the edges of W_* do not contribute to the first couple of significant digits of the total weight. The plots illustrate that W_* and W_{v_*} differ typically just in a small region well below this threshold; consequently, both walks yield about the same condition number. On the right note the five-leaved clover that represents the combination of algebraic and essential singularity at $z = -1$.

3.3.2. Reducing the Size of $\tilde{\Pi}$

As described in § 3.2, Provan's algorithm starts by building a walk for every pair $(v, e) \in V \times E$ and then proceeds by selecting the best enclosing one. A simple heuristic, which worked well for all our test cases, helps to considerably reduce the number of walks to be processed: Let

$$v_* = \operatorname*{argmin}_{v \in V} d(v)$$

and define W_{v_*} as a SEW subject to the constraint

$$W_{v_*} \in \tilde{\Pi}_{v_*} = \{\mathcal{P}_{v_*,u} \dot{+} uw \dot{+} \mathcal{P}_{w,v_*} : uw \in E\}.$$

Obviously W_* and W_{v_*} do not need to agree in general, as v_* does not have to be traversed by W_*. However, since v_* is the vertex with lowest weight, both walks differ mainly in a region that has no, or very minor, influence on the total weight and, consequently, also no significant influence on the condition number. Actually, W_* and W_{v_*} yielded precisely the same total weight for all functions that we have studied (Fig. 3.2 compares W_* and W_{v_*} for two typical examples). Using that heuristic, the run time of Provan's algorithm improves to $O(|E| + |V| \log |V|)$ because its main part reduces to applying Dijkstra's shortest path algorithm just once. In the case of the grid Ω_h this bound simplifies to

$$O(h^{-2} \log h^{-1}).$$

3.3.3. Size of the Grid Domain

The side length l of the square domain supporting Ω_h has to be chosen large enough to contain a SEW that would approximate an optimal general integration contour. E.g., if f is entire, we choose l large enough for this square domain to cover the optimal circular contour: $l > 2r_*$, where r_* is the optimal radius given in Bornemann (2011); a particularly simple choice is $l = 3r_*$. In other cases we employ a simple search for a suitable value of l by calculating W_* for increasing values of l until $d(W_*)$ does not decrease substantially anymore. During this search the grid will be just rescaled, that is, each grid uses a *fixed* number of vertices; this way only the number of search steps enters as an additional factor in the complexity bound.

3.3.4. Multilevel Refinement of the SEW

Choosing a proper value of h is not straightforward since we would like to balance a good approximation of a generally optimal integration contour with a reasonable amount of computing time. In principle, we would construct a sequence of SEWs for smaller and smaller values of h until the total weight of W_* does not substantially decrease anymore. To avoid an undue amount of computational work, we do not refine the grid everywhere but use an adaptive refinement by confining it to a tubular neighborhood of the currently given SEW W_* (see Fig. 3.3):

1: calculate W_* within an initial grid;

2: subdivide each rectangle adjacent to W_* into 4 rectangles;

3: remove all other rectangles;

4: calculate W_* in the newly created graph.

As long as the total weight of W_* decreases substantially, steps 2 to 4 are repeated. It is even possible to tweak that process further by not subdividing rectangles that just contain vertices or edges of W_* having weights below a certain threshold. By geometric summation, the complexity of the resulting algorithm is

$$O(H^{-4} \log H^{-1}) + O(h^{-2} \log h^{-1})$$

where H denotes the step size of the coarsest grid and $h = H/2^k$ the step size after k loops of adaptive refinement. An analogous approach to the constrained W_{v_*}-variant of the SEW algorithm given in §3.2 reduces the complexity further to

$$O(H^{-2} \log H^{-1}) + O(h^{-1} \log h^{-1}),$$

which is close to the best possible bound $O(h^{-1})$ given by the work that would be needed to just list the SEW.

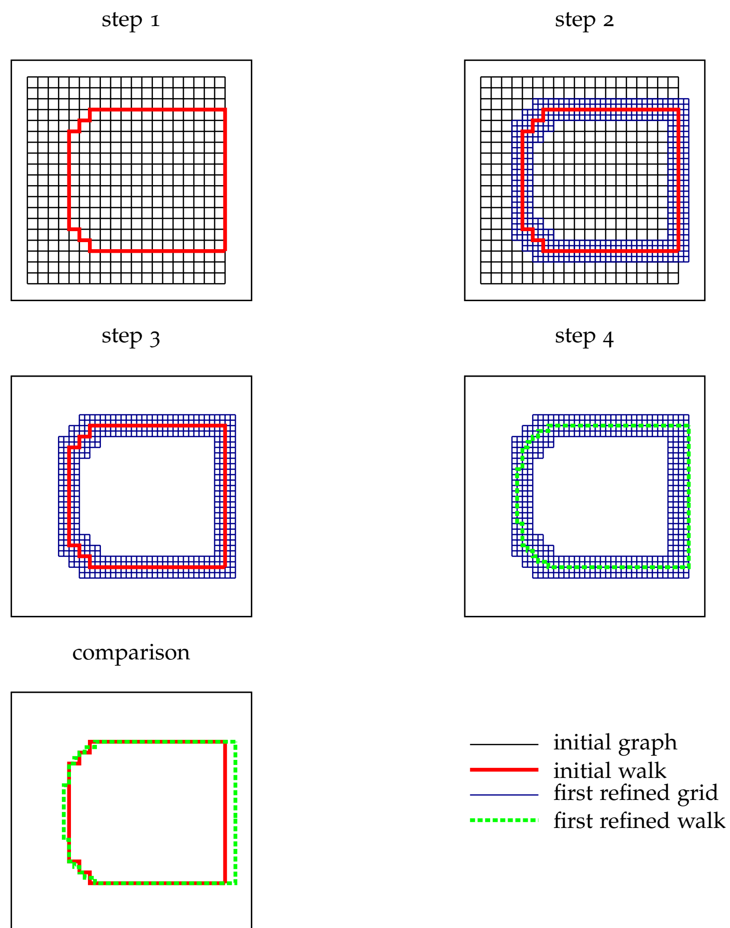


Figure 3.3.: Multilevel refinement of W_* $(f(z) = 1/\Gamma(z),\ n = 2006)$

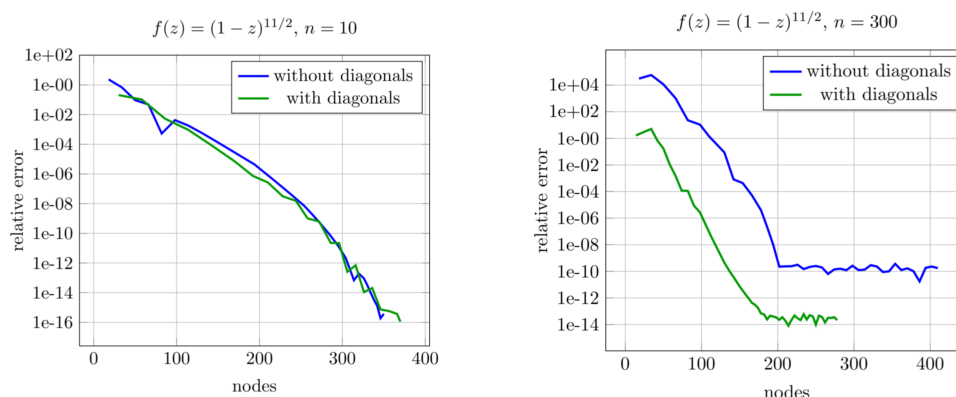


Figure 3.4.: Illustration of the spectral accuracy of piecewise Clenshaw–Curtis quadrature on SEW contours for a function with a branch-cut singularity. For larger n, we observe a significant improvement by adding diagonals to the grid. We get to machine precision for $n = 10$ and loose about two digits for $n = 300$. (Note that for optimized circular contours the loss would have been about 6 digits for $n = 10$ and about 15 digits for $n = 300$; cf. Bornemann 2011, Thm. 4.7).

3.3.5. Quadrature Rule for the Cauchy Integral

Finally, after calculation of the SEW $\Gamma = W_*$, the Cauchy integral (3.1) has to be evaluated by some *accurate* numerical quadrature. We decompose Γ into maximally straight line segments, each of which can be a collection of many edges. On each of those line segments we employ Clenshaw–Curtis quadrature (Clenshaw and Curtis 1960) in Chebyshev–Lobatto points[6]. Additionally we neglect segments with a weight smaller than 10^{-24} times the maximum weight of an edge of Γ, since such segments will not contribute to the result within machine precision. This way we not only get *spectral accuracy* but also, in many cases, less nodes as would be needed by the vanilla version of trapezoidal sums on a circular contour: Fig. 3.4 shows an example with the order $n = 300$ of differentiation but accurate solutions using just about 200 nodes which is well below what the sampling condition would require for circular contours (Bornemann 2011, §2.1). Of course, trapezoidal sums would also benefit from some recursive device that helps to neglect those nodes which do not contribute to the numerical result.

3.4. Numerical Results

Tab. 3.1 displays condition numbers of SEWs W_* as compared to the optimal circles C_{r_*} for five functions; Tab. 3.2 gives the corresponding CPU times and Fig. 3.5 shows some of the contours. (*All experiments were done using hardware arithmetic.*) The purpose of these examples is twofold, namely to demonstrate that:

[6]The n Chebyshev–Lobatto points for the interval $[-1, 1]$ are $\{x_j = \cos(\pi j/n); j = 0, \ldots, n\}$

Table 3.1.: Condition numbers for some $f(z)$: r_* are the optimal radii given in Bornemann (2011); W_* was calculated in all cases on a 51×51-grid with $l = 3r_*$ (in the last two cases l was found as in § 3.3.3). For $1/\Gamma(z)$, the peculiar order of differentiation $n = 2006$ is one of the very rare resonant cases (specific to this entire function) for which circles give exceptionally large condition numbers (cf. Bornemann 2011, Table 5). In the last example, differentiation is for $z = 1/\sqrt{2}$.

$f(z)$	n	$\kappa(W_*, n)$	$\kappa(C_{r_*}, n)$
e^z	300	1.1	1.0
$\mathrm{Ai}(z)$	300	1.3	1.2
$1/\Gamma(z)$	300	1.7	1.6
$1/\Gamma(z)$	2006	$7.8 \cdot 10^4$	$4.7 \cdot 10^4$
$(1-z)^{11/2}$	10	1.4	$5.0 \cdot 10^5$
$\exp(1/(1+8z)^{1/5})(1-z)^{11/2} J_0(z)$	100	$7.2 \cdot 10^2$	$4.3 \cdot 10^{12}$

Table 3.2.: CPU times for the examples of Tab. 3.1. Here t_{W_*} and $t_{W_{v_*}}$ denote the times to compute W_* and W_{v_*} and t_{quad} denotes the time to approximate the integral (3.1) on such a contour by quadrature. (There is no difference between W_* and W_{v_*} from the point of quadrature, see Fig. 3.2.) In the last example, differentiation is for $z = 1/\sqrt{2}$. The timings for the grids of size 25×25 and 51×51 match nicely the $O(h^{-4} \log h^{-1})$ complexity for W_* and the $O(h^{-2} \log h^{-1})$ complexity for W_{v_*}.

$f(z)$	n	grid	t_{W_*}	$t_{W_{v_*}}$	t_{quad}
e^z	300	51×51	$4.4 \cdot 10^2$ s	1.5 s	0.3 s
$\mathrm{Ai}(z)$	300	25×25	$2.1 \cdot 10^1$ s	0.5 s	1.7 s
$\mathrm{Ai}(z)$	300	51×51	$4.0 \cdot 10^2$ s	2.1 s	2.1 s
$1/\Gamma(z)$	300	25×25	$2.0 \cdot 10^1$ s	0.5 s	1.5 s
$1/\Gamma(z)$	300	51×51	$3.6 \cdot 10^2$ s	2.4 s	1.3 s
$1/\Gamma(z)$	2006	51×51	$3.6 \cdot 10^2$ s	2.3 s	3.1 s
$(1-z)^{11/2}$	10	51×51	$1.4 \cdot 10^3$ s	5.9 s	0.2 s
$\exp(1/(1+8z)^{1/5})(1-z)^{11/2} J_0(z)$	100	51×51	$7.0 \cdot 10^2$ s	3.5 s	0.3 s

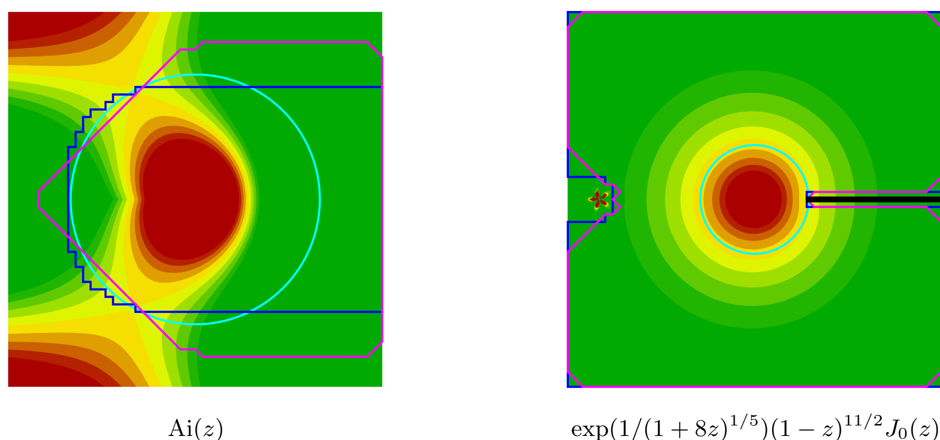


Figure 3.5.: W_{v_*} (blue: Ω_h without diagonals, magenta: Ω_h with diagonals) vs. C_{r_*} (cyan) for some examples of Tab. 3.1: the color coding shows the size of $\log d(z)$; with red for large values and green for small values. The smallest level shown is the threshold, below of which the edges of W_{v_*} do not contribute to the first significant digits of the total weight.

1. the SEW algorithm *matches* the quality of circular contours in cases where the latter are known to be optimal such as for entire functions;

2. the SEW algorithm is *significantly better* than the circular contours in cases where the latter are known to have severe difficulties.

Thus, the SEW algorithm is a flexible automatic tool that covers various classes of holomorphic functions in a completely algorithmic fashion; in particular there is no deep theory needed to just let the computation run.

In the examples of entire f we observe that W_* and W_{v_*}, like the optimal circle C_{r_*} would do, traverses the saddle points of $d(z)$. It was shown in Bornemann (2011, Thm. 10.1) that, for such f, the major contribution of the condition number comes from these saddle points and that circles are (asymptotically, as $n \to \infty$) paths of steepest decent. Since W_* can cross a saddle point only in a horizontal, vertical, or (if enabled) diagonal direction, somewhat larger condition numbers have to be expected. However, the order of magnitude of the condition number of C_{r_*} is precisely matched. This match holds in cases where circles give a condition number of approximately 1, as well as in cases with exceptionally large condition numbers, such as for $f(z) = 1/\Gamma(z)$ in the peculiar case of the order of differentiation $n = 2006$ (cf. Bornemann 2011, §10.4).

For non-entire f, however, optimized circles will be far from optimal in general: Bornemann (2011, Thm. 4.7) shows that the optimized circle C_{r_*} for functions f from the Hardy space H^1 with boundary values in $C^{k,\alpha}$ yields a lower condition number bound of the form

$$\kappa(C_{r_*}, n) \geqslant cn^{k+\alpha};$$

for instance, $f(z) = (1-z)^{11/2}$ gives $\kappa(C_{r_*}, n) \sim 0.16059 \cdot n^{13/2}$. On the other hand,

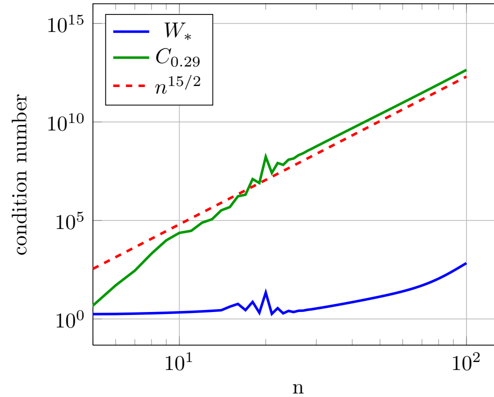


Figure 3.6.: An example with essential and algebraic singularities: the condition number of the Cauchy integral for $\exp(1/(1+8z)^{1/5})(1-z)^{11/2}J_0(z)$ for varying order n of differentiation at $z = 1/\sqrt{2}$; blue: optimal contour W_* in a 51×51 grid graph; green: circular contour with near optimal radius $r = 0.29 \approx 1 - 1/\sqrt{2}$; red: prediction of the growth rate from Bornemann (2011, Thm. 4.7).

W_* gives condition numbers that are orders of magnitude better than those of C_{r_*} by automatically following the branch cut at $(1, \infty)$.

The latter example can easily be cooked-up to outperform symbolic differentiation as well: using *Mathematica 8*, the calculation of the n-th derivative of $f(z) = \exp(1/(1+8z)^{1/5})(1-z)^{11/2}J_0(z)$ at $z = 1/\sqrt{2}$ takes already about a minute for $n = 23$ but had to be stopped after *more than a week* for $n = 100$. Despite the additional difficulty stemming from the combination of an algebraic and an essential singularity at $z = -1$, the W_{v_*} version of the SEW calculates this $n = 100$ derivative to an accuracy of 13 digits in less than 4 s; whereas optimized circular contours would give only about 3 correct digits here (see Fig. 4.1).

Conclusion

We can conclude that our algorithmic approach for finding optimal contours for Cauchy's integral formula was indeed successful. For many examples we were able to compute contours that yield similar condition numbers as optimal circles. The contours we obtained were also similar to those we would get by using the method of steepest descent. The next is now to generalize the ideas we used for Cauchy's integral formula to RHPs, hoping that this will result in an algorithm that yields contours similar to those obtained by using the method on nonlinear steepest descent.

4. The RHP Deformation Algorithm

As we have shown in the previous chapter, it is possible to reduce the problem of finding an optimal contour for the evaluation of a contour integral to the problem of finding the shortest path in a graph with suitable chosen weights. In this chapter we apply this approach to the problem of finding optimal contours for RHPs. We perform the same basic steps, we did for Cauchy's integral formula.

§ 4.1 We formulate a conjecture for the condition number of a RHP and explain the reasons which lead us to it. Furthermore we derive a weight which correlates a local part of the contour with its effect on the condition number of the RHP.

§ 4.2 Using this weight, we cast the problem of reducing the condition number of a RHP through deformations into a discrete optimization problem.

§ 4.3 For Cauchy's integral formula any deformation of the contour was valid if it preserved the winding number around a certain point. Similarly, there are also constraints on the contour and different types of deformations for RHPs. The two types of deformations we consider are simple and lensing deformations. Here, we give a basic description how they work.

§ 4.4 We present an algorithm, which serves the same purpose as Provan's shortest enclosing walk algorithm did for Cauchy's integral formula. It solves the discrete optimization problem on a graph. The algorithm is described in multiple stages. The first is a greedy algorithm that can perform simple deformations of a RHP.

§ 4.5 The next stage is extending the algorithm so that it can also perform lensing deformations.

§ 4.6 We discus another extension that improves handling of infinite contours, i.e. contours containing rays or lines.

§ 4.7 Although we build most constraints, which have to be satisfied to get a valid deformation, into the structure of the graphs or the algorithm, we did not manage to do so for all of them. The remaining constraint is verified by the algorithm as described in this section.

§ 4.8 Finally, we give some further details about about our implementation of the algorithm we described in the previous sections.

We illustrate our ideas with the Painlevé II RHP (see § 1) throughout this chapter.

This main results presented in this chapter have been published in (Wechslberger and Bornemann 2014) and the chapter is based on this paper. The first two section are very similar to this paper, the remaining part has been heavily extended.

We recall from the introduction that we consider RHPs with the following form:

RHP 1 (example)**.** *Find a holomorphic function* $\Phi : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(m, \mathbb{C})$ *satisfying*

$$\begin{aligned} &\Phi(\infty) = I \\ &\Phi^+(z) = \Phi^-(z)G(z) \quad \textit{for } z \in \Gamma^0. \end{aligned}$$

Where Γ^0 is obtained by removing points of self intersection from Γ. If the contour Γ consists of several parts $\Gamma_1, \ldots, \Gamma_k$ we denote the jump matrix corresponding to the part Γ_j by

$$G_j|_{\Gamma^0 \cap \Gamma_j} = G|_{\Gamma^0 \cap \Gamma_j}.$$

and to simplify the discussion about contour deformations later on, we assume that all of these jump matrices $G_j|_{\Gamma^0 \cap \Gamma_j}$ can be continued of Γ_j with an entire function

$$G_j : \mathbb{C} \to \mathrm{GL}(m, \mathbb{C}).$$

4.1. Condition of a Riemann–Hilbert Problem

The numerical method of Olver

Olver (2011b) constructed his spectral collocation method by recasting RHPs as a particular kind of singular integral equation. Upon writing

$$\Phi(z) = I + C_\Gamma U(z)$$

with the Cauchy transform of a matrix-valued function $U : \Gamma \to \mathrm{GL}(m, \mathbb{C})$, namely

$$C_\Gamma U(z) = \frac{1}{2\pi i} \int_\Gamma \frac{U(\zeta)}{\zeta - z} \, d\zeta,$$

the RHP becomes the linear operator equation

$$AU(z) = U(z) - C_\Gamma^- U(z) \cdot (G(z) - I) = G(z) - I. \tag{4.1}$$

Here, $C_\Gamma^\pm U(z)$ denotes the non-tangential limit of $C_\Gamma U(z')$ as $z' \to z$ from the positive (negative) side of the contour; there is the operator identity $C_\Gamma^+ - C_\Gamma^- = I$. The residue at ∞ becomes simply the integral

$$\operatorname*{res}_{z=\infty} \Phi(z) = \frac{1}{2\pi i} \int_\Gamma U(\zeta) \, d\zeta.$$

Without going into details, in this chapter it suffices to note that the n-point numerical approximation of (4.1) yields a finite-dimensional linear system

$$A_n U_n = b_n$$

where the jth component of the solution U_n is a matrix that approximates $U(z_j)$ at the collocation point $z_j \in \Gamma$ ($j = 1, \ldots, n$). The stability of the method is essentially described by the condition number

$$\kappa_n = \kappa(A_n) = \|A_n^{-1}\| \cdot \|A_n\|$$

of this linear system: altogether, one would typically suffer a loss of $\log_{10} \kappa$ significant digits. Under an additional assumption, which can be checked *a posteriori* within the numerical method itself, Olver and Trogdon (2012, Assumpt. 6.1 and Lemma 6.1) proved a bound of the form[1]

$$\kappa_n = O(\kappa(A))$$

in terms of the condition number $\kappa(A) = \|A^{-1}\| \cdot \|A\|$ of the continuous operator A, with constants that are midly growing in the number of collocation points n. Here, the operator norm of A is obtained by acting on $L^2(\Gamma)$. Extending U_n to all of Γ by interpolation, Olver and Trogdon (2012, Eq. (6.1)) also state an error estimate of the form

$$\|U - U_n\|_{L^2(\Gamma)} \leqslant c\kappa(A) n^{2+\beta-k} \|U\|_{H^k(\Gamma)}$$

with some $\beta > 0$. Since, for jump matrices G that are piecewise restrictions of entire functions, k can be chosen arbitrarily large, one gets spectral accuracy.

Preconditioning of RHPs

Thus, stability and accuracy of the numerical method depend on $\kappa(A)$, which blows up in many problems of interest. We recall the example in Fig. 1.3, where the undeformed version of the RHP for Painlevé II is $\kappa_n \approx 10^{16}$ for $s_1 = 1$ and $s_2 = 2$ and $x = -20$ (see also Fig. 4.1 vor varying x).

Now, it is important to understand that $\kappa(A)$ is the condition number of the RHP for the restricted data (Γ, G) but not for the jump data G_j ($j = 1, \ldots, k$) which are obtained from analytic continuation. If the continued data are *explicitly* given, and are not themselves part of the computational problem,[2] it should be possible to deform the RHP to an equivalent one with data $(\tilde{\Gamma}, \tilde{G})$ and

$$\kappa(\tilde{A}) \ll \kappa(A).$$

We call such a deformation a *preconditioning* one. In fact, Olver and Trogdon (2012) argued that preconditioning *is* possible whenever the method of nonlinear steepest

[1] They employ the estimate $\|A\| \leq \sqrt{2}(1 + \|G - I\|_{L^\infty(\Gamma)} \|C_\Gamma^-\|)$ in their statements.

[2] Analytic continuation corresponds to solving a Cauchy problem for the elliptic Cauchy–Riemann differential equations; it is, therefore, an ill-posed problem.

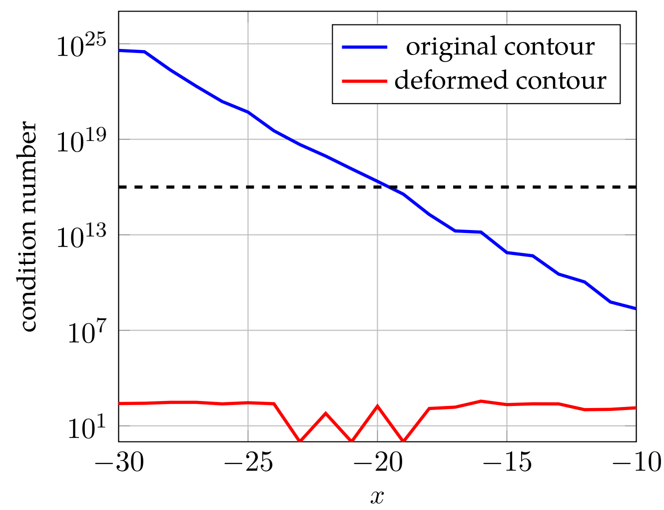


Figure 4.1.: Comparison of the condition number of the original contour and of a deformed contour optimized by the greedy algorithm of § 4.4 for the Painleve II RHP with $(s_1, s_2) = (1, 2)$. The condition number of the deformed contour is roughly constant for all values of x while the condition number of the original contour grows exponentially fast for decreasing values of x. Note that condition numbers larger than 10^{16} (dashed line) obstruct the computation of even a single accurate digit in machine arithmetic, indicating severe numerical instability.

descent produces an asymptotic formula; Fig. 4.4 shows a typical sequence of such manually constructed preconditioning deformations for the Painlevé II RHP.

Though it seems to be difficult to extract a single governing principle for all the ingenious deformations that are used in the asymptotic analysis of RHPs, we base our algorithmic approach on the following simple observations:

- If there are no jumps in the RHP, that is if $G \equiv I$, we have $A = I$ and therefore $\kappa(A) = 1$. By continuity, $G \to I$ in some sufficiently strong norm would certainly imply $\kappa(A) \to 1$, such that a reasonably small $\|G - I\|$ will probably yield a moderately sized condition number $\kappa(A)$.
- The method of nonlinear steepest descent has already successfully been used for preconditioning and, as described before in § 1, it deforms the RHP into a state where $G \to I$ exponentially fast. This in turn also yields reasonably small $\|G - I\|$.
- Numerical experiments comparing $\kappa_n(A)$ and $\|G - I\|$ indicate that there is a connection between the two. See Fig. 4.2 for more details.

All things considered, we conjecture that an estimate for $\kappa(A)$ can be cast in the form

$$\kappa(A) \leqslant \phi(\|G - I\|_{W^{s,p}(\Gamma)}) \tag{4.2}$$

for some Sobolev $W^{s,p}$-norm and some monotone function ϕ that is independent of (Γ, G). A good preconditioning strategy would then be to make $\|G - I\|_{W^{s,p}(\Gamma)}$ as small as possible, we call it the *relative strength* of the jump matrix G.

In the lack of any better understanding of the precise dependence of $\kappa(A)$ on the RHP data (Γ, G) we suggest to use $\|G-I\|_{L^1(\Gamma)}$ as a measure of relative strength: optimizing it led to significant reductions of the condition number in all of our experiments. However, the deformation algorithm itself will just use that the measure $d(\Gamma; G)$ can be written as an integral over Γ, namely in the form

$$d(\Gamma; G) = \int_\Gamma d(G(z))\, d|z|$$

for some function $d : \mathrm{GL}(m, \mathbb{C}) \to [0, \infty)$, which we call the *local weight*.

4.2. Preconditioning as a Discrete Optimization Problem

Since our objective is preconditioning, the relative strength of the jump matrices does not really have to be minimized over all equivalent deformations $(\tilde{\Gamma}, \tilde{G})$ of a given RHP (Γ, G). For all practical purposes it suffices to consider just a very coarse, finite set of possible contours, namely paths within a planar graph.

The basic idea is as follows: first, we restrict the problem to a bounded region of the complex plane and embed the part of the contour Γ belonging to that region as paths into

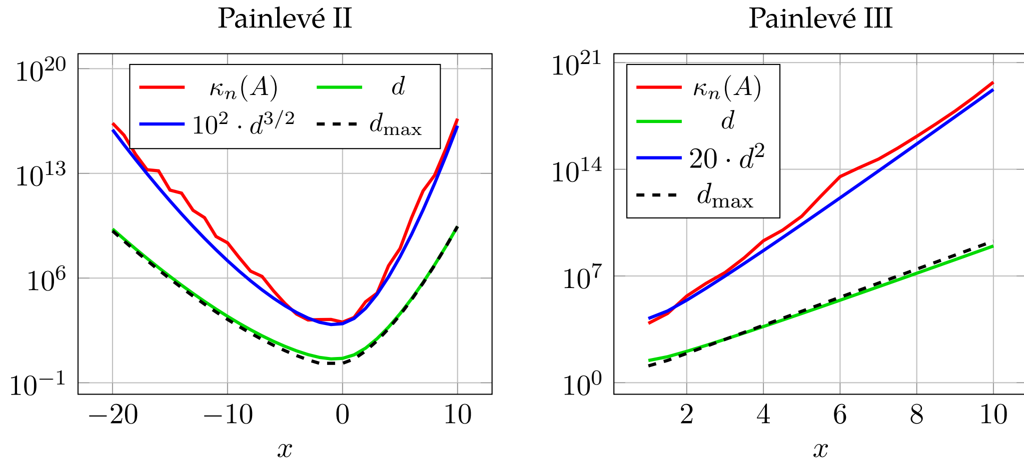
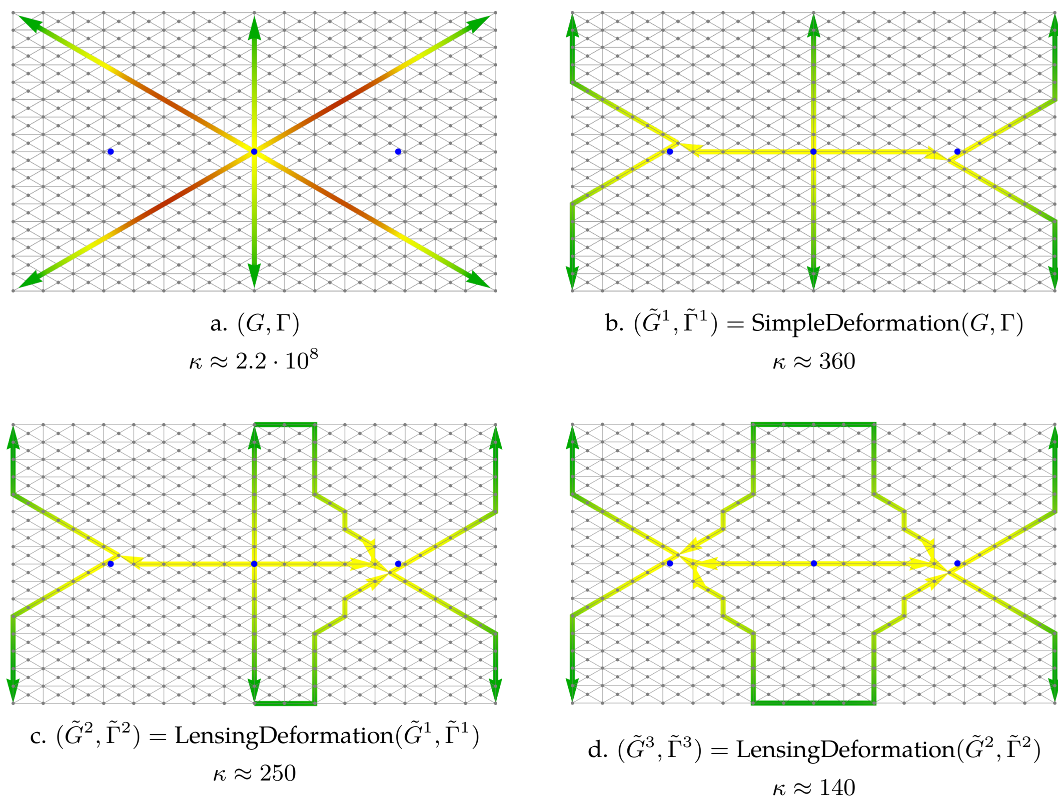


Figure 4.2.: Comparison of $\kappa_n(A)$ and $\|G - I\|$ for the Painlevé II and III RHP for varying x; The parameters of the RHPs are $s_1 = 1$, $s_2 = 2$ for Painlevé II and $s_1^{(0)} = 1, s_2^{(0)} = 2, s_2^{(\infty)} = 3, \theta_0 = 3 + 43/100, \theta_{\text{inf}} = 1 + 123/1000$ for Painlevé (standard example available in RHPackage), see (Fokas et al. 2006, p.201) for the definition of the Painlevé III RHP. Here, $d = \int_\Gamma \|G(z) - I\|_F d|z|$ and $d_{\max} = \max_{z \in \Gamma} \|G(z) - I\|_F$. The behaviour of $\kappa(A)$ and d is similar for varying x and a simple polynomial applied to d even allows to closely predict the magnitude of $\kappa_n(A)$. In these two examples, $d_{\max}$ is also a good estimate for d and consequently also for $\kappa_n(A)$.



a. (G, Γ)
$\kappa \approx 2.2 \cdot 10^8$

b. $(\tilde{G}^1, \tilde{\Gamma}^1) = \text{SimpleDeformation}(G, \Gamma)$
$\kappa \approx 360$

c. $(\tilde{G}^2, \tilde{\Gamma}^2) = \text{LensingDeformation}(\tilde{G}^1, \tilde{\Gamma}^1)$
$\kappa \approx 250$

d. $(\tilde{G}^3, \tilde{\Gamma}^3) = \text{LensingDeformation}(\tilde{G}^2, \tilde{\Gamma}^2)$
$\kappa \approx 140$

Figure 4.3.: Application of SimpleDeformation and LensingDeformation of § 4.4 and § 4.5 to the Painlevé II RHP with $s_1 = 1$, $s_2 = 2$ and $x = -10$. On the top left is the original contour of this RHP and the other contours are deformed versions of it. All contours have been calculated on a 17×17 grid. The color encodes the magnitude of $\|G(z) - \mathrm{I}\|_F$, with green = 10^{-16}, yellow = 1, and red = 10^4. The blue dots indicate the origin $z = 0$ and the stationary points of the phase function θ at $z = \pm\sqrt{-x}/2$. The reduction of the condition number from (a) to (d) corresponds to an accuracy gain of about six digits.

a coarse, grid-like planar graph $g = (V, E)$ (see Fig. 4.3.a for an example of the Painlevé II RHP: because of a super exponential decay as $z \to \infty$ along each of the rays, $G - I$ is already zero if evaluated on a computer outside the indicated rectangle).

Second, for each j, the analytic continuation G_j of the jump data on Γ_j turns the graph g into an edge-weighted graph g_j by using the (edge) weights

$$d_j(e) = \int_e d(G_j(z))\, d|z| \qquad (e \in E).$$

Last, we replace Γ_j (within the bounded domain) by the *shortest* path (with the same endpoints as Γ_j) with respect to g_j subject to the following constraint: the thus deformed RHP must be equivalent to the original one.

It is this latter constraint which adds to the algorithmic difficulty of the problem: the Γ_j cannot be optimized independent of each other. We will address this problem by a greedy strategy: the largest contribution to the weight constraints the admissible paths of the second largest one and so on; this will be accomplished by modifying the underlying graphs g_j in the corresponding order.

Fig. 4.3.b shows the result of such an algorithmic deformation for the Painlevé II RHP ($s_1 = 1$, $s_2 = 2$, $x = -10$): the condition number is reduced by about six orders of magnitude. Further improvement is possible by performing a "lensing" deformation, that is, by introducing multiple edges based on a factorization of G (see § 4.5). The results of two such steps are shown in Fig. 4.3.c and d (more steps would not pay off). Though the improvement of the condition number is more modest in these two steps, it is instructive to compare the algorithmic contour in Fig. 4.3.d with the manual construction of Olver and Trogdon (2012) shown in Fig. 4.4.

Fig. 4.1 compares, for varying values of x, the condition number of the original contour with that of the deformed contour optimized by the greedy algorithm of § 4.4: a uniform stabilization by preconditioning is clearly visible.

4.3. Admissible Deformations of Riemann–Hilbert Problems

We briefly recall two of the deformations that can be applied to RHPs. A more detailed description can be found in (Fokas et al. 2006).

4.3.1. Simple Deformations

Fig. 4.5 shows an example of such a deformation. In general, simple deformations allow to continuously move a contour part in the complex plane (thereby covering a region Ω)

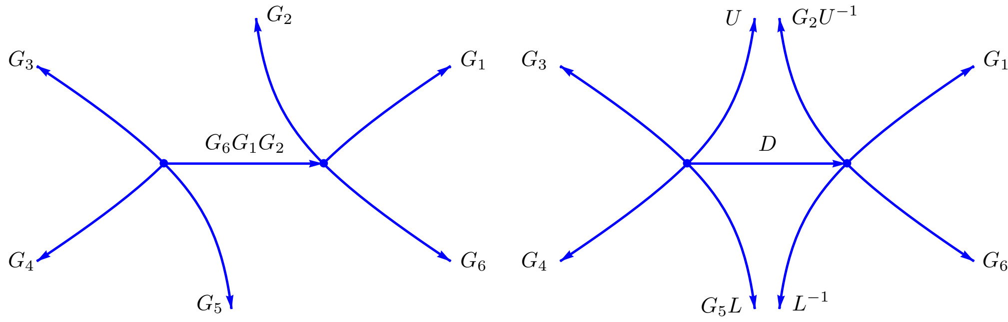


Figure 4.4.: Some manual constructions for the Painlevé II RHP taken from Olver and Trogdon (2012, p. 20). Left: Deformation along the paths of steepest descent; right: deformation after lensing. The contours bifurcate at the stationary points of the phase function θ.

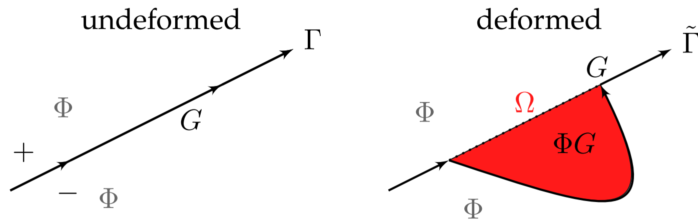


Figure 4.5.: A simple deformation.

as long as the following conditions are satisfied:

(i) $\tilde{\Gamma}$ does not cross other parts of Γ,

(ii) Ω does not contain any other contour parts,

(iii) G has a holomorphic continuation in Ω.

Then, the deformed RHP in Fig. 4.5 is solved by the function

$$\tilde{\Phi} = \begin{cases} \Phi G & : x \in \Omega, \\ \Phi & : x \notin \Omega. \end{cases}$$

Conditions (i)-(iii) can be mapped to graph-constrained deformations as follows:

Condition (i) can be handled by *splitting* a graph as shown in Fig. 4.6: if a path p corresponding to a part of a contour is given, like the path highlighted in blue, we duplicate the vertices of p and change all edges on the right side of p so that they are connected to the newly created vertices but not to the vertices of p itself. This way no

path in the graph can cross p anymore. We will use $g[p_1, p_2, \dots]$ to denote a graph g, which has been split in this fashion along the paths $p_1, p_2, \dots$.

Condition (ii) is difficult to be built into the structure of a graph *a priori*, but it is easy to check for it *a posteriori*: the circle composed by Γ_i and $\tilde{\Gamma}_i$ must not enclose an endpoint of another arc. If all candidates for a path violate this condition, the algorithm simply stops (this never happened in our experiments; dealing with such a situation would require to break the deformations into smaller pieces).

Condition (iii) can be handled by removing those regions from the graph where G does not have a holomorphic continuation.

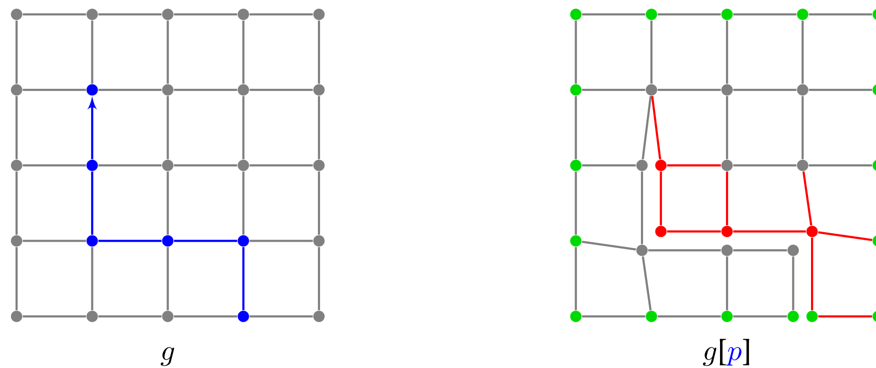


Figure 4.6.: Illustration of split graphs. *Left*: The original graph g is about to be split along the path p (blue). *Right*: After splitting g along p, we get the depicted split graph $g[p]$. All vertices and edges that have been changed or created are highlighted in red. To handle infinite contours (rays, lines), paths ending on the boundary (green) are implicitly extended to infinity. Consequently, g is also split at endpoints of p on the boundary, so that no path can cross this implicit extension. For a clear visualization the split in the graph has been enlarged by moving the vertex positions of the vertices on both sides of the split; in the actual graphs used by the algorithm the duplicated vertices stay at exactly the same position. All graphs are undirected, the arrow at the end of the blue path just indicates its orientation.

Just as the graph in Fig. 4.6, all graphs in this chapter are drawn using the following style.

Visualization Style 2 (Graphs)**.**

- *All edges are undirected.*
- *Arrows just indicate the orientation of the paths.*
- *All splits are enlarged for a clear visualization.*
- *Without enlargement the vertices on the left and right sides of splits coincide.*

4.3.2. Multiple Deformations and Factorization: Lensing

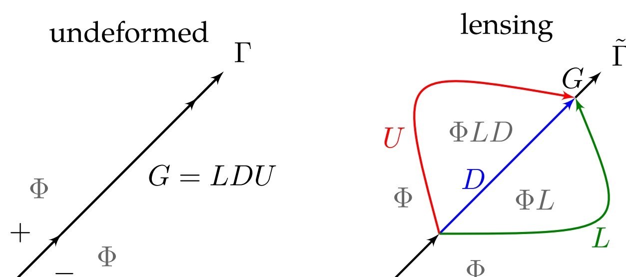


Figure 4.7.: A lensing deformation.

Fig. 4.7 shows an example of such a deformation. To initialize, several copies of a contour part are created at one and the same location, where each copy corresponds to a factor of a given multiplicative decomposition of the jump matrix G. We call these copies the *factors* of this part of the contour Γ. These factors are then moved around in the complex plane subject to conditions (i)-(iii) and, additionally, the following condition:

(iv) the mutual orientation of the factors must be preserved.

For example, in Fig. 4.7 the order of the decomposition $G = LDU$ requires that the factor U is to the left of the factor D and that the factor D is to the left of the factor L. To preserve this orientation in our deformation algorithm, we calculate the shortest path for just one of the factors. For the other factors we use a modification of the shortest enclosing circle algorithm of (Provan 1989), see § 4.5.

4.4. The Greedy Algorithm

In this section we describe an algorithm that can performed simple deformations of RHPs (as described in § 4.3.1), using the idea of § 4.2.

4.4.1. Notation

- $d(e)/d(p)$: weight of the edge e / the path p
- $p_1 \dot{+} p_2$: path p_1 joined with path p_2
- $\overleftarrow{p}$: reversed path p
- $p[u, v]$: subpath from vertex u to vertex v within the path p

- $\mathrm{sp}(g, u, v)$: shortest path from vertex u to v in the weighted graph g
- $\mathrm{int}(W)$: interior of a walk, that is, all vertices v with $\mathrm{ind}(W, v) = \pm 1$
- p_+/p_- : path that forms the left/right side of the split along p in $g[p]$
- $p_+[w]/p_-[w]$: vertex in p_+ / p_- that corresponds to $w \in p_+ \cup p_-$ (see Fig. 4.13)
- $\mathcal{B}(g)$: set of vertices which form the boundary of the graph g
- $\mathrm{nv}(g, a)$: set of vertices in g which are at the position $a \in \mathbb{C}$
- $\mathrm{nv}_{\mathcal{B}}(g, a)$: subset of vertices in $\mathcal{B}(g)$ with the same argument as a

4.4.2. Optimized Simple Deformations

The idea of algorithm 1 goes as follows:

1. (lines 11 to 13) For each of the contour parts Γ_j and the corresponding jump matrices G_j (which are assumed to have a holomorphic continuation to the rectangular region supporting the grid), a separate weighted graph g_j with edge weights
$$d(e) = \int_e d(G_j(z))\, d|z|$$
is created, see Fig. 4.8. A copy of these inital graphs is stored in $g^\star$ for later and the graphs g are modified by the algorithm.

2. (lines 16 to 18) For each Γ_j a shortest path is calculated that shares the same endpoints, see Fig. 4.9. The thus separately optimized paths, however, will in general not satisfy condition (i) of § 4.3, that is, they will cross each other. Therefore, some of the paths have to be modified to match this condition, which increases the corresponding weight.

3. (lines 19 to 21) By keeping the path P of dominant total weight fixed, we restrict such modifications to the other parts that contribute less to the condition number.

4. (lines 28 to 30) By splitting all graphs along P and repeating the calculation of the shortest paths in the split graphs (step 2, lines 16 to 18), we come up with paths that do not cross P, see Fig. 4.10.

5. (line 15) This procedure is then repeated, until all paths are fixed and, hence, non-crossing. (In each round of this loop another path gets fixed.)

6. (line 33) Finally, the algorithm constructs the deformed contour data from the just calculated set of paths. For paths and subpaths that do not share an edge with another path we simply use the path and the corresponding jump matrix as new contour data. Subpaths which occur in more than one path will be mapped to new contour data by performing an "inverse lensing": the new jump matrix is calculated as the (properly ordered) product of all the jump matrices sharing that subpath. For example, if the paths P_i and P_j have a common subpath s and P_i is to the left of P_j, the function MapToRHP creates a new contour part s with the jump matrix $G_j G_i$.

7. (lines 22 to 26) In a situation as shown in Fig. 4.10, where the new optimal paths share a subpath with the already fixed ones, further improvement is possible by optimizing the shared subpath with respect to the weight obtained from combining the corresponding jump matrices (that is, the just mentioned "inverse lensing"). This procedure (algorithm 6) is illustrated schematically in Fig. 4.17; the application of this procedure to the example of Fig. 4.10 is shown in Fig. 4.11.

8. (line 31) The paths in P may not be valid paths in $g_i^\star[P(F)]$ if two of them share a subpath. An example for this situation is shown in Fig. 4.12. To compensate for this effect, we calculate the left and right sides (P^+ and P^-) of splits created by the paths in P after a path has been fixed . The paths in P^+ and P^- are used instead of those in P for mapping paths to a RHP (line 33) and calculating shortest paths (line 17). Fig. 4.13 provides an illustration of the difference between P,P^+,P^-,$(P_i)_+$ and $(P_i)_-$.

After this general overview of the algorithm, in the next sections we will discuss the individual steps

§ 4.4.4 : calculating shortest paths (line 17)

§ 4.4.5 : improving a shared subpath (line 24)

§ 4.4.6 : calculating left and right side of splits (line 31)

in more detail. For the sake of simplicity, the presented steps are not optimized with respect to runtime. A list of possible optimizations is available in § 4.8.7. Though the main algorithm 1 does not put any restrictions on the contour, we will describe the just mentioned steps only for the type of contour discussed in the next section.

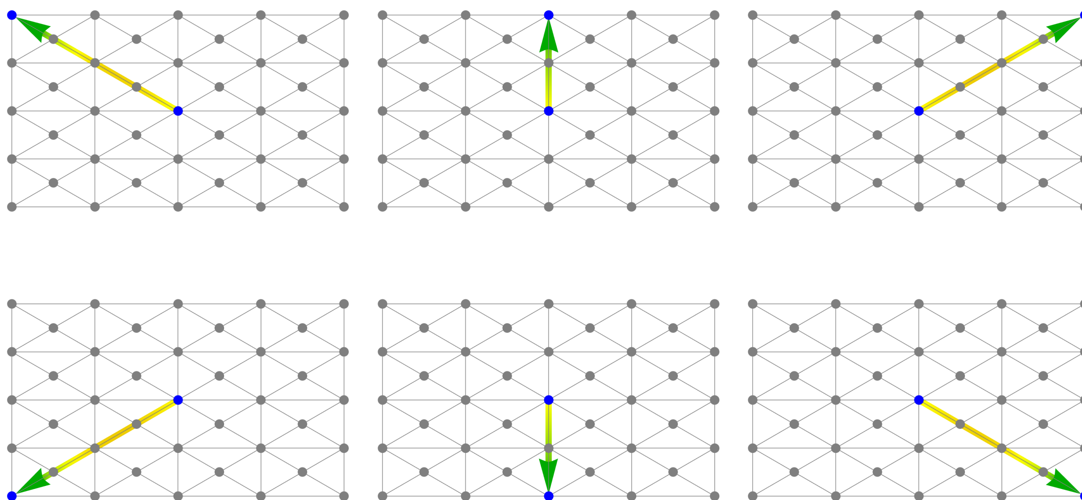


Figure 4.8.: Create separate weighted graphs for each contour part Γ_j with weights depending on the jump matrix G_j (line 12 in algorithm 1).

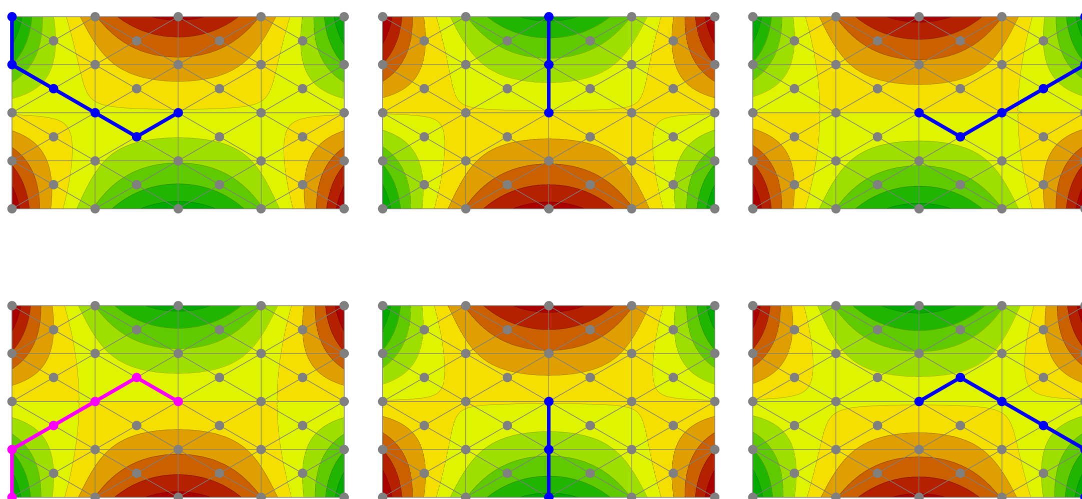


Figure 4.9.: Calculate the shortest paths for each G_j separately, highlighted in blue (line 17 in algorithm 1). The one with the largest total weight is shown in magenta (line 19). The color encodes the magnitude of $\|G_j(z) - \mathrm{I}\|_F$, with green = 10^{-16}, yellow = 1, and red=10^{16}.

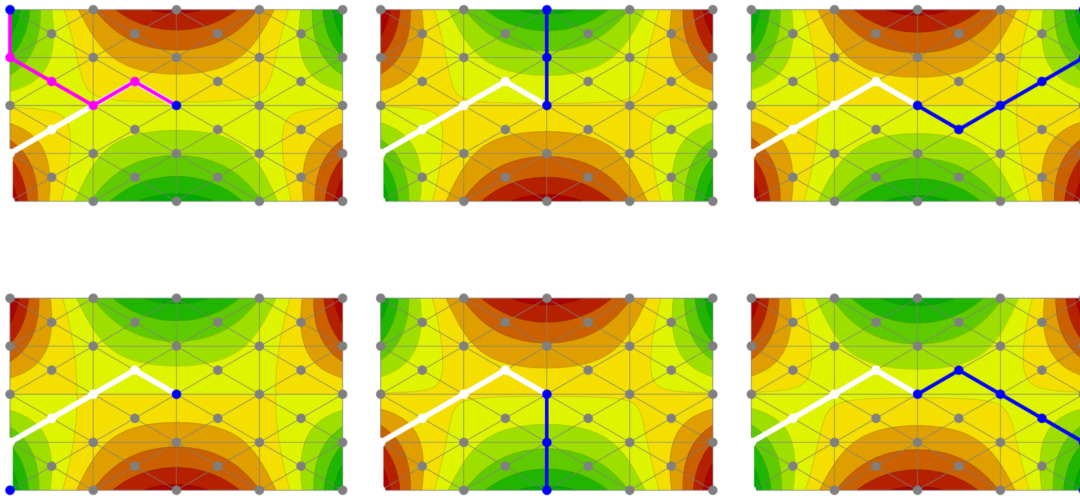


Figure 4.10.: Recalculate the shortest paths (line 17 in algorithm 1) in the graphs split along the optimal path of largest weight from Fig. 4.9, here shown in white. The new shortest paths are highlighted in blue, the one of maximal weight in magenta.

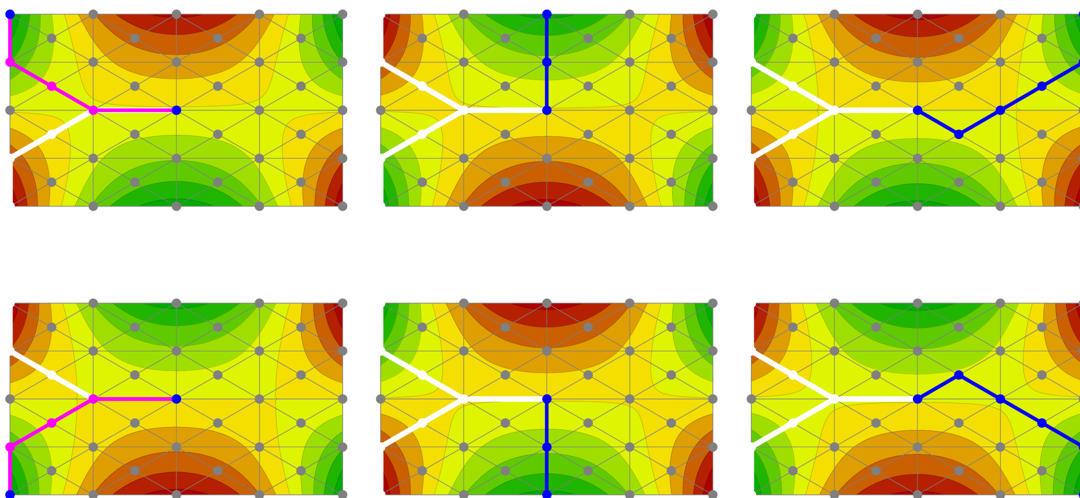


Figure 4.11.: Improvement of the subpath shared by the fixed white path and the magenta path of Fig. 4.10 (algorithm 6). After this improvement, both paths are fixed and the graphs are split along their union (the white Y-shaped contour).

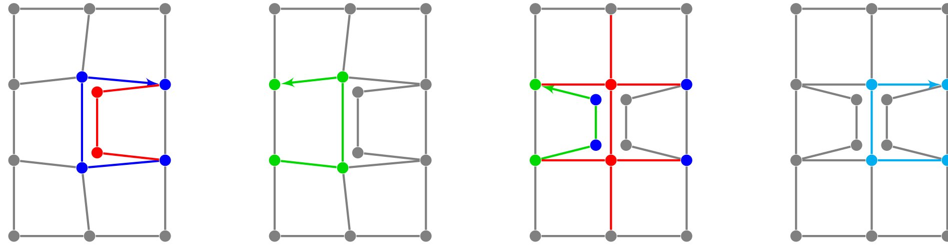


Figure 4.12.: Visualization according to style 2; Example for a path in P being an invalid path in $g[P]$; *left*: Splitting the graph g along P_1 (blue) creates additional vertices (red). *middle left*: Next, the graph is split along the path P_2 (green). *middle right*: This creates additional vertices (red) in $g[P_1, P_2]$ and cuts P_1. The vertices of P_1 (blue) no longer define a valid path as some edges of P_1 were removed by the split along P_2. *right*: As P_1 is not a valid path in $g[P_1, P_2]$, we use the path P_1^+ (cyan) whenever we need the left side of the split along P_1.

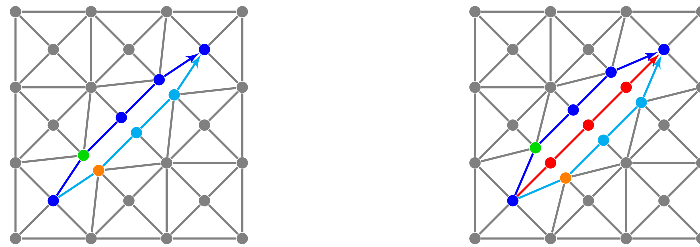


Figure 4.13.: Show case for the notations $p_\pm$,P,$P^\pm$; *left*: The graph g is split along the path p (blue), which creates the left side p_+ (blue) and the right side p_- (cyan). $p_\pm[v]$ is used to denote the 1 to 1 relation between vertices in p_+ and p_-. E.g. the vertices u (green) and w (orange) can also be denoted by $p_+[u] = p_+[w] = u$ and $p_-[u] = p_-[w] = w$. *right*: Here the graph g is split along $P = (p, p)$, so it is split along the same path p (blue) twice. $(P_i)_\pm$ refers to the left/right side of the split along P_i at the time they were created, which yields $(P_1)_+ = (P_2)_+$ (blue), $(P_1)_-$ (cyan) and $(P_2)_-$ (red). In contrast $P_i^\pm$ refers to the the left / right side corresponding to the split along P_i after all splits have been created. This yields P_1^- (cyan), $P_1^+ = P_2^-$ (red) and P_2^+ (blue). Just as before $(P_i)_\pm[v]$ refers to a vertex on a specific side of a split. Once again the vertices u (green) and w (orange) can also be denoted by $(P_1)_+[u] = (P_1)_+[w] = u$ and $(P_1)_-[u] = (P_1)_-[w] = w$.

Algorithm 1 Optimized Simple Deformation

1: **function** SimpleDeformation(G, Γ)
2: $n = |\Gamma|$ ▷ *number of contour parts*
3: $P = ()$ ▷ *fixed new paths*
4: $P^+ = ()$ ▷ *left side of paths in P*
5: $P^- = ()$ ▷ *right side of paths in P*
6: $p = ()$ ▷ *candidates for new paths*
7: $F = ()$ ▷ *already processed contour parts*
8: $Q = (1, \ldots, n)$ ▷ *unprocessed contour parts*
9: $g = ()$ ▷ *graphs corresponding to contour parts*
10: $g^\star = ()$ ▷ *initial graphs without splits*
11: **for all** $i \in Q$ **do**
12: g_i = graph with edge weights $d(e) = \|G_i - I\|_{L^1(e)}$
13: **end for**
14: $g^\star = g$
15: **while** $Q \neq ()$ **do**
16: **for all** $i \in Q$ **do**
17: $p_i = \text{ShortestPath}(g_i, \Gamma, i, P^+, P^-, F)$ ▷ *shortest path for Γ_i in g_i*
18: **end for**
19: $i_\star = \operatorname{argmax}_{i \in Q} d(p_i)$
20: $P_{i_\star} = p_{i_\star}$
21: $Q = Q \backslash (i_\star)$
22: **for all** $i \in F$ **do** ▷ *try to improve the path if there is an intersection*
23: **if** $P_{i_\star} \cap P_i^+ \neq \emptyset$ or $P_{i_\star} \cap P_i^- \neq \emptyset$ **then**
24: $(P, F, Q) = \text{ImproveSharedSubpath}(g^\star, G, \Gamma, P, P^+, P^-, F, i, i_\star)$
25: **end if**
26: **end for**
27: $F = F \cup (i_\star)$
28: **for all** $i \in Q$ **do**
29: $g_i = g_i^\star[P(F)]$ ▷ *split graphs along paths P ordered by F*
30: **end for**
31: $(P^+, P^-) = \text{CalculateSides}(P, F)$ ▷ *update left and right sides*
32: **end while**
33: $(\widetilde{G}, \widetilde{\Gamma}) = \text{MapToRHP}(G, P^+)$ ▷ *create new contour parts*
34: **return** $(\widetilde{G}, \widetilde{\Gamma})$
35: **end function**

4.4.3. Admissible Contour Types

Our implementation currently supports contours consisting of

$$\begin{array}{rll} \text{points} & P_a & = \{a\} \\ \text{intervals} & I_{a,b} & = [a,b] \\ \text{rays} & R_c(a) & = \{a + rc : r \in \mathbb{R}_0^+\} \\ \text{lines} & L_c & = \{rc : r \in \mathbb{R}\} \\ \text{oriented circles} & C_r^s(a) & = \{a + re^{i\beta} : \beta \in [0, 2\pi]\} \end{array}$$

with $a, b, c \in \mathbb{C}, r \in \mathbb{R}^+, s \in \{+, -\}$. This list of contour part types is sufficient to handle the contours of all the RHPs we used in our numerical experiments and it can even handle all RHPs we saw during our research. Some special types of contours (arcs, circles with an attached ray) have to be emulated with multiple intervals.

Furthermore, we use the convention that parts of a RHP, which originate from a lensing decomposition (see § 4.3.2) are given from left to right. By just factorizing the jump function and not altering the contour, we get a contour which contains identical curves. This convention is used to encode the order from left to right of the factors of the decomposition.

Thereby points are not associated with jumps. They are merely used as a simple way to preserve the information about the centers of circles. This information would otherwise be lost if we discretize the circle with a path, but e.g. if there is a singularity at the center of the circle we need its position ensure that we do not move a path across it. Each of the just mentioned continuous contour parts is associated with one specific type of path in a graph $g = (V, E)$. These types are

$$\begin{array}{rl} \text{point paths} & P_P = \{v : v \in V\} \\ \text{interval paths} & P_I = \{v_1 \dots v_k : v_1, v_k \notin \mathcal{B}(g)\} \\ \text{ray paths} & P_R = \{v_1 \dots v_k : v_1 \notin \mathcal{B}(g), v_k \in \mathcal{B}(g)\} \\ \text{line paths} & P_L = \{v_1 \dots v_k : v_1, v_k \in \mathcal{B}(g)\} \\ \text{circle paths} & P_C = \{v_1 v_2 \dots v_1 : v_1 \in V\} \end{array}$$

with $v_1, \dots, v_k \in V$. So we encode the fact that a ray / line path extends to infinity by using a start / end vertex on the boundary of the graph. Furthermore, the direction with which a line or ray path approaches infinity corresponds to the argument of the position of the vertex on the boundary. Rays in the opposite direction (from infinity to a finite point) are also supported by our implementation of the algorithm but we will not discuss them here. They are handled analogous to the rays which we will discuss.

4.4.4. Calculate Shortest Paths

Algorithm 2 calculates a shortest path for the contour part Γ_j in the graph g. To do so it has to handle a few different cases, which we divide into two groups.

Group 1: "no lensing" (lines 3 to 19)
This group contains all cases, in which we do not have to take the lensing restriction (iv)

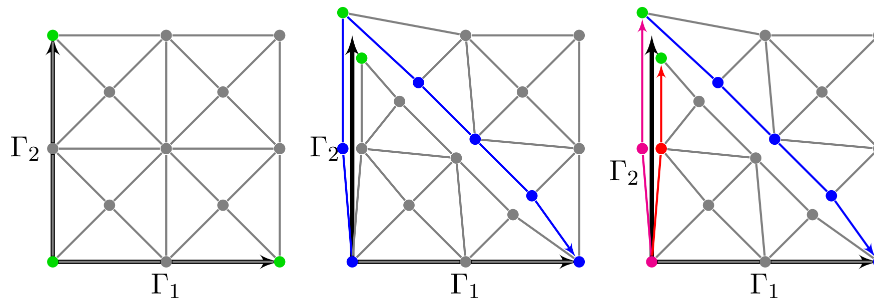


Figure 4.14.: Visualization according to style 2; Situation with multiple shortest paths for a single contour part; *left*: We have a contour with two parts Γ_1 and Γ_2, which is discretized by a graph g (grey). Each of the endpoints of the contour can be mapped to exactly one vertex (green). *middle*: The path p (blue) for the contour part Γ_1 passes through the right endpoint of Γ_2. Therefore after fixing p, we have two vertices (green) in $g[p]$, to which we can map the right endpoint of Γ_2 *right*: For each of these mappings algorithm 2 calculates a shortest path, this is q_1 (magenta) and q_2 (red) (line 9). But as the last vertex of q_2 is within the area enclosed by Γ_1 and p, q_2 is not a valid choice and is discarded (line 32).

from § 4.3.2 into account. That means that either Γ_j is not part of a lensing deformation or that there is not yet a fixed path for any factor of the lensing deformation, to which Γ_j belongs.

For the first case, in which Γ_j is a point, we just map the point to a vertex. The next three cases (interval, ray, line) are handled in a similar way. We determine all possible mappings of the endpoints to vertices (lines 7, 8, 11, 12, 15 and 16) and then calculate the shortest path for all of these mappings (lines 9, 13 and 17). If an endpoint is not finite, it is mapped to a vertex on the boundary of the graph. There is more than one mapping for an endpoint, if the graph is split along a path which passes through one of the endpoints. See Fig. 4.14 for an example. The last case in this group is Γ_j being a circle (line 19). We handle this case by calculating circles enclosing the center of the circle with the correct orientation using algorithm 5, which is simply the shortest enclosing walk algorithm of (Provan 1989) without the last step of choosing the shortest walk.

Group 2: "lensing" (lines 20 to 31)
This part handles lensing deformations and is discussed later on in § 4.5, where we also provide all the other information regarding how lensing deformations are performed by the algorithm.

Final Steps (lines 32 to 34)
After a set of paths W has been calculated in group 1 or 2, we return the shortest valid path in this set, (lines 32 and 33). A path is valid if it does not violate condition (ii) of

§ 4.3.1 and this is checked by algorithm 3. This algorithm creates a temporary contour $\widetilde{\Gamma}$ replacing all parts of Γ with their corresponding fixed path in P and Γ_j with a path in W (lines 3 to 6). Afterwards we verify that no endpoint is certainly[3] in an area between the original contour Γ and the deformed contour $\widetilde{\Gamma}$ (line 7). The definition of $V(\Gamma)$ and further details which have to be considered to determine whether condition (ii) is violated or not are discussed in § 4.7.

Optimality of the paths
We should note that algorithm 2 does not necessarily return the shortest possible path in all cases. For the case that Γ_j is a circle $C_r^s(a)$ (line 19) we get according to (Provan 1989), that the shortest walk in W is also the shortest walk enclosing a. But if further restrictions on the path have to be honoured (e.g. endpoints of other parts of the contour have to be enclosed), it can happen that the shortest walk in W is not a valid choice and gets discarded by SelectValidPaths in line 32. A similar situations could occur for the cases handled by lines 27 and 29. As we will discuss later, the shortest path in W is also the shortest path to the left / right of Γ_j, but this shortest path may also be discarded in line 32.

Nevertheless the shortest path in W is in all of these cases a lower bound for the shortest path in g which honours all restriction on a path for Γ_j. Therefore we do at least know how far we could be away from the optimal path. In addition we did not encounter a situation during our numerical experiments where a better shortest path algorithm would have yielded better results. Therefore, it did not seem to be worthwhile to invest time in finding or researching better shortest path algorithms. Instead we postpone this task until the need for it arises. Especially as exchanging one of the shortest paths algorithms, can be done without altering any other part of the algorithm.

What if $Q = \emptyset$?
In principle, it could happen that SelectValidPaths discards all paths in line 32, although it never happened in our experiments. Nevertheless, algorithm 1 is prepared for this situation. If no valid path can be found, it just ignores Γ_j in the current step of the algorithm. As the weight of the empty path is 0 in line 19, a path for another part of the contour will be fixed during this step. By fixing another path we add another split to the graph, which is also another restriction that will be honoured next time algorithm 2 tries to find a shortest path. This additional restriction can help the shortest path algorithms to find a valid shortest path.

To illustrate this process, we consider the following example. Let us assume algorithm 2 tries to calculate the shortest path for Γ_j and it turns out W contains only paths such that one but not both endpoints of another part $\Gamma_k = I_{a,b}$ is in the area between the path and Γ_j. None of the paths in W is valid and algorithm 1 fixes a paths for another part of

[3] For the remaining continuous contour parts in $\widetilde{\Gamma}$ there might me several possible mappings for their endpoints to vertices. If that is the case, we consider a path to be valid as long as there is for every endpoint a mapping which does not cause a violation of condition (ii), see Fig. 4.14 for an example.

the contour, until it eventually fixes a path p for Γ_k. Then the next time algorithm 2 tries to find a shortest path for Γ_j, the set W will be different. Due to the just created split along p it is no longer possible, that exactly one endpoint of Γ_k in in the area between a path in W and Γ_j.

Should no valid path be available in algorithm 1, line 19, the algorithm aborts. That did not happen during our numerical experiments though.

What if $W = \emptyset$?
Well then the implementation contains a bug. There always has to be at least one candidate for a path to replace Γ_j, otherwise Γ would not be a valid contour for a RHP in the first place.

Algorithm 2 finds a shortest path for a part of the contour

```
1: function ShortestPath(g, Γ, j, P⁺, P⁻, F)
2:   if ∄ i ∈ F with Γ_i = Γ_j then                          ▷ check for lensing case
3:     switch Γ_j do                                          ▷ standard case
4:       case P_a
5:         W = nv(g, a)
6:       case I_{a,b}
7:         L = nv(g, a)                                       ▷ map endpoints to vertices
8:         R = nv(g, b)
9:         W = {sp(g, l, r) : l ∈ L, r ∈ R}                   ▷ shortest paths for mappings
10:      case R_c(b)
11:        L = nv(g, a)
12:        R = nv_B(g, c)
13:        W = {sp(g, l, r) : l ∈ L, r ∈ R}
14:      case L_c
15:        L = nv_B(g, −c)
16:        R = nv_B(g, c)
17:        W = {sp(g, l, r) : l ∈ L, r ∈ R}
18:      case C_r^s(a)
19:        W = enclosingCircles(g, a, s)
20:  else                                                     ▷ lensing case
21:    i = { max{h : h ∈ F, h < j, Γ_h = Γ_j}   if {h : h ∈ F, h < j, Γ_h = Γ_j} ≠ ∅
           { ∞                                 else
22:    k = { min{h : h ∈ F, h > j, Γ_h = Γ_j}   if {h : h ∈ F, h > j, Γ_h = Γ_j} ≠ ∅
           { −∞                                else
23:    if i < j < k then          ▷ there is a fixed path for Γ_j / Γ_k to the left /right of Γ_j
24:      lqr = P_i^+
25:      W = {sp(g [P_k^+ ∔ ←P_i^−], l, r)}                   ▷ path between P_i^− and P_k^+
26:    else if i < j then                  ▷ there is a fixed path for Γ_i to the left of Γ_j
27:      W = EnclosingPaths(g, P_i^+, P_i^−, −1)
28:    else if j < k then                 ▷ there is a fixed path for Γ_k to the right of Γ_j
29:      W = EnclosingPaths(g, P_k^+, P_k^−, +1)
30:    end if
31:  end if
32:  Q = SelectValidPaths(Γ, P⁺, F, W, j)                     ▷ remove invalid choices for Γ_j
33:  p = argmin_{q∈Q} d(q)                                    ▷ select valid paths with lowest weight
34:  return p
35: end function
```

Algorithm 3 selects paths from a set, which are valid choices for a part of the contour

```
1: function SelectValidPaths(Γ, P, F, W, j)
2:     n = |Γ|                                          ▷ number of contour parts
3:     Γ̃ = (Γ̃_1, ..., Γ̃_n) with Γ̃_h = { P_h  if h ∈ F
                                        { Γ_h  else
4:     Q = {}          ▷ filtered subset of W containing only valid choices for Γ_j
5:     for all q ∈ W do
6:         Γ̃_j = q                 ▷ new contour which uses the paths in P and q
7:         if V(Γ) ∈ V(Γ̃) then          ▷ check if q violates condition (ii) in § 4.3.1
8:             Q = Q ∪ q                              ▷ q is valid choice for Γ_j
9:         end if
10:    end for
11:    return Q
12: end function
```

Algorithm 4 determines paths enclosing the left / right side of another path

```
1: function EnclosingPaths(g = (V, E), p⁺, p⁻, s)
2:     l v₊ ... r = p⁺          ▷ split left/right side of path into endpoints (l,r) and
3:     l v₋ ... r = p⁻          ▷ one vertex in between on the left/right side (v₊/ v₋)
4:     T = {sp(g, l, u) ∔ uw ∔ sp(g, w, r) : uw ∈ E}
5:     W = { {t : t ∈ T, ind(t⃖ ∔ p⁺, v₋) = −1    if s = −1
           { {t : t ∈ T, ind(t⃖ ∔ p⁻, v₊) = +1    if s = +1
6:     return W
7: end function
```

Algorithm 5 calculates circles enclosing a given point

```
1: function EnclosingCircles(g = (V, E), a, s)
2:     T = {sp(g, v, u) ∔ uw ∔ sp(g, w, v) : uw ∈ E v ∈ V}
3:     v = nv(g, a)
4:     W = {t : t ∈ T or t⃖ ∈ T, ind(T, v) = s}
5:     return W
6: end function
```

4.4.5. Shared Subpath Improvement

Algorithm 6 tries to improve a shared subpath s of the paths $P_j^{\pm}$ and $P_{j_\star}$. We recall that $P_{j_\star}$ is the path that is fixed in the current iteration of the algorithm and $P_j^{\pm}$ is the left/right side of a path that has been fixed in a previous iteration. Both paths were chosen by algorithm 1 to optimize the weight corresponding to their jumps G_j respectively $G_{j_\star}$, but this does not necessarily mean that the subpath s is optimal with regard to the combined jump across both contour parts occurring at s. So the algorithm creates a graph with weights corresponding to the combined jump across P_j and $P_{j_\star}$ and calculates a shortest path as a replacement for s within this graph. Just as in algorithm 2 there are several different cases, which require their own method to calculate a new valid subpath. The individual steps are the following.

Setup (lines 2 to 9)
The first step is to reset the state (P, F, Q) to the state, in which algorithm 1 calculated P_j. This is done by algorithm 8, which discards P_j as well as all paths fixed after P_j and adjusts the lists F and Q as well as the left (P^+) and right (P^-) sides of the paths in P accordingly (line 2). Afterwards a graph $\widetilde{g}$ is created with weights corresponding to the combined jump at the longest shared subpath of P_j and $P_{j_\star}$ (lines 3 to 6). See Fig. 4.15 for details about how the combined jump is determined by algorithm 7. After this setup phase a shortest path between the endpoints of s is calculated to replace s in P_j.

Shortest Path Group 1: "No Lensing" (lines 10 to 17)
The first three cases, Γ_j being an interval, ray or line, are all handled by calculating the shortest path connecting the endpoints of s in the just created graph $\widetilde{g}$ (line 13).

For the last case, Γ_j being a circle, the shortest path connecting the endpoints of s does not necessarily yield a new path that encloses the center of the circle. Therefore we construct a path q connecting the endpoints of s with the center of the circle and determine a path to the right or to the left of the just constructed path depending on the orientation of the circle (line 16). This might not yield the shortest possible valid path, but it also limits the effect of the new subpath on the full path. We want that the new subpath improves the region of the original subpath, but not that it completely alters the circle path P_j. The last step is to collapse the split along q, i.e. map the vertices of q_- to q_+ for all paths in $\widetilde{S}$, to ensure that we have valid paths in $\widetilde{g}$ (line 17). This method is visualized by Fig. 4.16.

Shortest Path Group 2: "Lensing" (lines 18 to 32)
This part handles lensing deformations and is discussed later on in § 4.5, where we also provide all the other information regarding how lensing deformations are performed by the algorithm.

Create New Path (lines 33 to 41)
To create the new and improved path for P_j we start with creating a set of all paths, which we get by replacing the subpath s of P_j with a path in $\widetilde{S}$ (line 33). As the subpaths

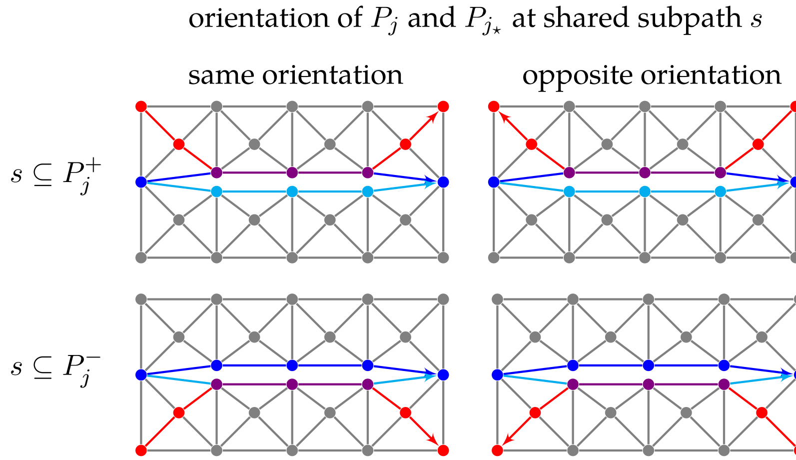


Figure 4.15.: Visualization according to style 2; Illustration of the different cases in CombinedJump (algorithm 7) The paths P_j^+ (blue) / P_j^- (cyan) and $P_{j_\star}$ (red) share a subpath s (violet) and we want to determine the combined jump across P_j and $P_{j_\star}$ at s with the orientation of P_j. Using "inverse lensing" we get the following. *top left*: $P_{j_\star}$ is to the left of $P_{j_\star}$, so inverse lensing gives the combined jump $\widetilde{G} = G_j G_{j_\star}$ (line 3). *bottom left*: $P_{j_\star}$ being to the right of P_j yields the combined jump $\widetilde{G} = G_{j_\star} G_j$ (line 6) *right*: Inverting both the jump matrix $G_{j_\star}$ and the orientation of $P_{j_\star}$ does not alter the jump and reduces the cases on the right side to the cases on the left side yielding *top right*: $\widetilde{G} = G_j G_{j_\star}^{-1}$ (line 9) and *bottom right*: $G_{j_\star}^{-1} G_j$ (line 12).

in $\widetilde{S}$ can intersect the left and right subpaths p_l and p_r of P_j, the just created paths can contain circular subpaths (see Fig. 4.17, top right). These subpaths do not serve any purpose and are dropped (line 34). Afterwards, all invalid paths are discarded and the shortest path $\widetilde{P}_j$ of the remaining paths is selected to replace P_j. If the new path $\widetilde{P}_j$ differs from the original path P_j, then the original path and state is returned, which causes the main algorithm 1 to continue as if nothing happened (line 38). Otherwise the new path and reseted state is returned and henceforth replaces the previous state in the main algorithm 1, effectively discarding all paths fixed after P_j (line 40).

Why reset the state?

At first glance it may not seem very efficient to reset the state, because it discards all paths that have already been fixed after P_j. But in our experiments we did rarely encounter a case in which any path except for $P_{j_\star}$ was discarded. The shared subpath improvement step is intended for the case that fixing $P_{j_\star}$ changed the shortest path for $\Gamma_{j_\star}$ in a way such that its weight becomes the dominating part for the condition of the RHP. Due to the policy that the path with the largest weight is fixed in each step, the shortest path for $\Gamma_{j_\star}$ will then be fixed immediately after P_j has been fixed. Consequently only the path

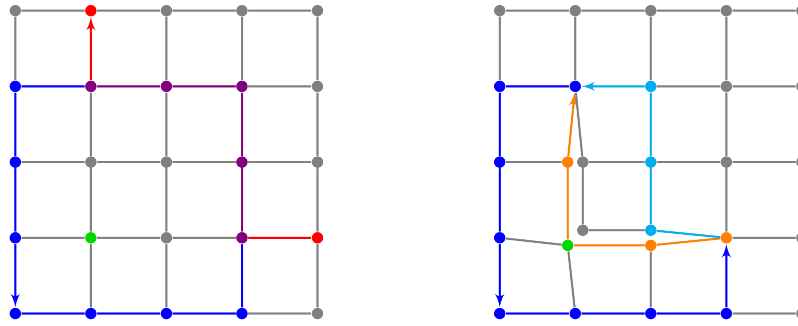


Figure 4.16.: Visualization according to style 2; Example of IMPROVESHAREDPATH (algorithm 6) for the case of Γ_j being a circle $C_r^+(c)$; *Left*: The circle P_j (blue) enclosing c (green) shares a subpath s (violet) with the path $P_{j_\star}$ (red). *Right*: The algorithm splits $\widetilde{g}$ along the path q (orange) formed by the shortest paths connecting the endpoints of s with c (line 15) and creates candidates for a valid shortest path to the right of q in $\widetilde{g}$, e.g. the cyan path (line 16). As the first edge of the cyan path is part of q_-, it is mapped to q_+ (line 17) to get a path that forms an improved circle enclosing c in $\widetilde{g}$ if combined with the remaining left and right subpaths of P_j (blue).

$P_{j_\star}$ is discarded.

Furthermore discarding $P_{j_\star}$ is also advantageous compared to just replacing the subpath s with the just calculated new path. Changing P_j increases in general the freedom we have to choose $P_{j_\star}$. Therefore calculating a completely new path for $P_{j_\star}$ potentially yields a better result than just replacing the subpath. Often the newly calculated path $P_{j_\star}$ also shares a longer subpath than s with P_j and so algorithm 1 will try to further improve the shared subpath of P_j and $P_{j_\star}$. See Fig. 4.17 for an example.

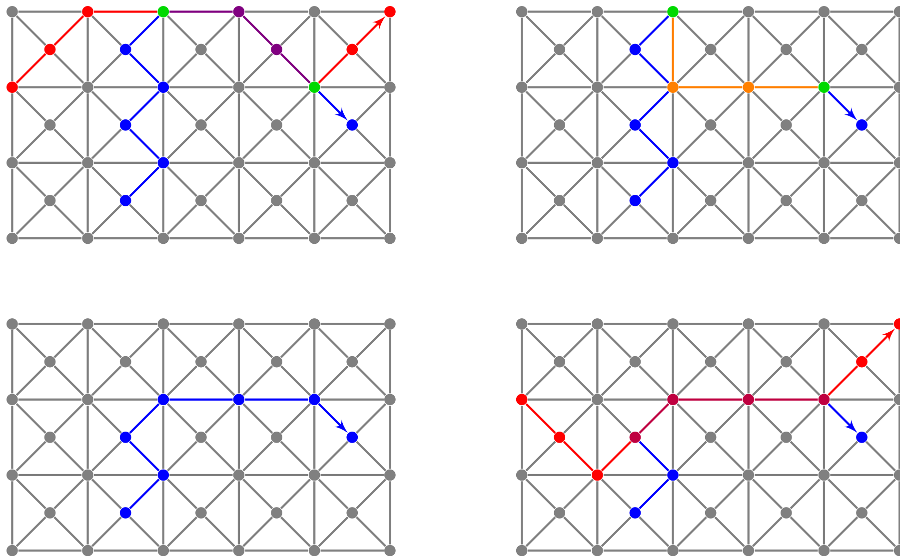


Figure 4.17.: Visualization according to style 2; An illustration of applying one round of ImproveSharedSubpath (algorithm 6). *Top left*: a graph in which the two paths P_i (blue) and P_j (red) share a subpath (violet). *Top right*: after replacing the shared subpath in P_i with a shortest path (orange) connecting the endpoints (green) of the shared subpath with regard to the combined weight of both jumps, we get a new path for P_i which contains a circle. *Bottom left*: this circle gets removed from the blue path. *Bottom right*: an updated shortest path for P_j (red) results in a new shared subpath (violet) that could be further improved by applying ImproveSharedSubpath recursively.

Algorithm 6 Shared Subpath Improvement

1: **function** ImproveSharedSubpath($g^\star, G, \Gamma, P, P^+, P^-, F, j, j_\star$)
2: $(\widetilde{P}, \widetilde{F}, \widetilde{Q}, \widetilde{P}^+, \widetilde{P}^-) = \textsc{ResetState}(P, F, Q)$
3: $s^\pm =$ longest shared subpath of P_j^+ and $P_{j_\star}$ or P_j^- and $P_{j_\star}$
4: $\widetilde{G} = \textsc{CombinedJump}(G, P^+, P^-, s^\pm, j, j_\star)$ ▷ *combine jumps of both parts*
5: $g =$ copy of $g_j^\star$ with edge weights $d_e = \|\widetilde{G} - I\|_{L^1(e)}$
6: $\widetilde{g} = g[\widetilde{P}(\widetilde{F})]$
7: $s =$ subpath of P_j corresponding to $s^\pm$
8: $l \dots r = s$ ▷ *left (l) and right (r) endpoint of s*
9: $p_l \dotplus s \dotplus p_r = P_j$ ▷ *split P_j into left, shared and right subpaths*
10: **if** $\nexists i \in \widetilde{F}$ with $\Gamma_i = \Gamma_j$ **then** ▷ *check for lensing case*
11: **switch** Γ_j **do** ▷ *standard case*
12: **case** $I_{a,b}$ or $R_c(b)$ or L_c
13: $\widetilde{S} = \{\mathrm{sp}(\widetilde{g}, l, r)\}$
14: **case** $C_r^o(a)$
15: $q = \mathrm{sp}(\widetilde{g}, l, \mathrm{nv}(g, a)) \dotplus \mathrm{sp}(\widetilde{g}, \mathrm{nv}(g, a), r)$
16: $\widetilde{S} = \textsc{EnclosingPath}(\widetilde{g}[q], q_+, q_-, -o)$
17: $\widetilde{S} = \textsc{UndoSplit}(q, \widetilde{S})$
18: **else** ▷ *lensing case*
19: $i = \begin{cases} \max\{h : h \in \widetilde{F}, h < j, \Gamma_h = \Gamma_j\} & \text{if } \{h : h \in \widetilde{F}, h < j, \Gamma_h = \Gamma_j\} \neq \emptyset \\ \infty & \text{else} \end{cases}$
20: $k = \begin{cases} \min\{h : h \in \widetilde{F}, h > j, \Gamma_h = \Gamma_j\} & \text{if } \{h : h \in \widetilde{F}, h > j, \Gamma_h = \Gamma_j\} \neq \emptyset \\ -\infty & \text{else} \end{cases}$
21: **if** $i < j < k$ **then** ▷ *there are paths for Γ_i to the left and Γ_k to the right of Γ_j*
22: $\widetilde{S} = \{\mathrm{sp}(\widetilde{g}\left[\widetilde{P}_k^+ \dotplus \overleftarrow{\widetilde{P}_i^-}\right], l, r)\}$
23: **else if** $i < j$ **then** ▷ *there is a path for Γ_i to the left of Γ_j*
24: $q = \overleftarrow{p_l} \dotplus \widetilde{P}_i^- \dotplus \overleftarrow{p_r}$
25: $\widetilde{S} = \textsc{EnclosingPath}(\widetilde{g}[q], q_+, q_-, -1)$
26: $\widetilde{S} = \textsc{UndoSplit}(q, \widetilde{S})$ ▷ *remove split along q from all paths in $\widetilde{S}$*
27: **else if** $j < k$ **then** ▷ *there is a path for Γ_k to the right of Γ_j*
28: $q = \overleftarrow{p_l} \dotplus \widetilde{P}_i^+ \dotplus \overleftarrow{p_r}$
29: $\widetilde{S} = \textsc{EnclosingPath}(\widetilde{g}[q], q_+, q_-, +1)$
30: $\widetilde{S} = \textsc{UndoSplit}(q, \widetilde{S})$ ▷ *remove split along q from all paths in $\widetilde{S}$*
31: **end if**
32: **end if**
33: $W = \{p_l \dotplus \widetilde{s} \dotplus p_r : \widetilde{s} \in \widetilde{S} \cup \{s\}\}$
34: $W = \textsc{DropCircleSubpaths}(W)$ ▷ *drop circular subpaths from paths in W*
35: $T = \textsc{SelectValidPaths}(\Gamma, \widetilde{P}^+, F, W, j)$ ▷ *remove invalid choices for Γ_j*
36: $\widetilde{P}_j = \mathrm{argmin}_{t \in T}\, d(t)$ ▷ *select valid paths with lowest weight*
37: **if** $\widetilde{P}_j = P_j$ **then**
38: **return** P, F, Q
39: **else**
40: **return** $\widetilde{P}, \widetilde{F}, \widetilde{Q}$
41: **end if**
42: **end function**

Algorithm 7 determines the combined jump of two paths with a shared subpath

```
1: function CombinedJump(G, P⁺, P⁻, s, j, j⋆)
2:     if s ⊆ P_j^+ then
3:         G̃ = G_j G_{j⋆}            ▷ P_{j⋆} is to the left of P_j, same orientation of P_{j⋆} and P_j at s
4:     end if
5:     if s ⊆ P_j^- then
6:         G̃ = G_{j⋆} G_j            ▷ P_{j⋆} is to the right of P_j, same orientation of P_{j⋆} and P_j at s
7:     end if
8:     if s⃖ ⊆ P_j^+ then
9:         G̃ = G_j G_{j⋆}^{-1}       ▷ P_{j⋆} is to the left of P_j, opposite orientation of P_{j⋆} and P_j at s
10:    end if
11:    if s⃖ ⊆ P_j^- then
12:        G̃ = G_{j⋆}^{-1} G_i       ▷ P_{j⋆} is to the right of P_j, opposite orientation of P_{j⋆} and P_j at s
13:    end if
14:    return G̃
15: end function
```

Algorithm 8 resets the state $(P, F, Q, \widetilde{P}^+, \widetilde{P}^-)$ to the state in which P_j was fixed

```
1: function ResetState(P, F, Q, j)
2:     h = index of j in F
3:     F̃ = F(1 : h − 1)               ▷ drop P_j and all paths which were fixed after it
4:     Q̃ = (1, . . . , |Γ|) \ F
5:     P̃ = ()
6:     P̃(F̃) = P(F̃)
7:     (P̃⁺, P̃⁻) = CalculateSides(P̃, F̃)
8:     return P̃, F̃, Q̃
9: end function
```

4.4.6. Calculate Left and Right Sides of Splits

In this section we discuss algorithm 9 which determines the left (P^+) and right (P^-) sides of splits in the graph $g[P(F)]$, which has been split along all paths in P in the order as they appear in F. This is also the order in which the algorithm processes the paths.

1. (lines 6 to 8) The left and right side of the current path p is initialized with its left / right side ($(P_i)_+$ / $(P_i)_-$) immediately after the split was created.
2. (lines 9 to 18) If any of the vertices v in p is contained in the left side of an already processed path and both paths have the same orientation at v, the left side of the processed path is adjusted by replacing v with $p_-[v]$. The same adjustment is also made if v is part of the right side of a processed path and both paths have opposite orientations at v. An illustration of these adjustments is shown in Fig. 4.18. See Fig. 4.29, for more details about SimilarOrientation and how it is defined in corner cases.
3. (line 19) The current path is added to the list of already processed paths. After all paths have been processed, P^+ and P^- contain the left and right sides of the splits created by the paths P in the graph $g[P(F)]$.

Algorithm 9 determine left and right side of paths after multiple splits

```
1:  function CalculateSides(P, F)
2:      P+ = ()                                          ▷ left side of paths in P
3:      P− = ()                                          ▷ right side of paths in P
4:      Q = ()                                           ▷ indexes of paths in P+,P−
5:      for all i ∈ F do
6:          p = P_i                                      ▷ initialize P_i+ and P_i−
7:          P_i+ = p_+
8:          P_i− = p_−
9:          for all v ∈ p do
10:             for all j ∈ Q do
11:                 if v ∈ P_j+ and SimilarOrientation(P_j+, p, v) then
12:                     replace v in P_j+ by p_−[v]      ▷ top left case in Fig. 4.18
13:                 end if
14:                 if v ∈ P_j− and not SimilarOrientation(P_j−, p, v) then
15:                     replace v in P_j− by p_−[v]      ▷ bottom right case in Fig. 4.18
16:                 end if
17:             end for
18:         end for
19:         Q = Q ∪ (i)
20:     end for
21:     return (P+, P−)
22: end function
```

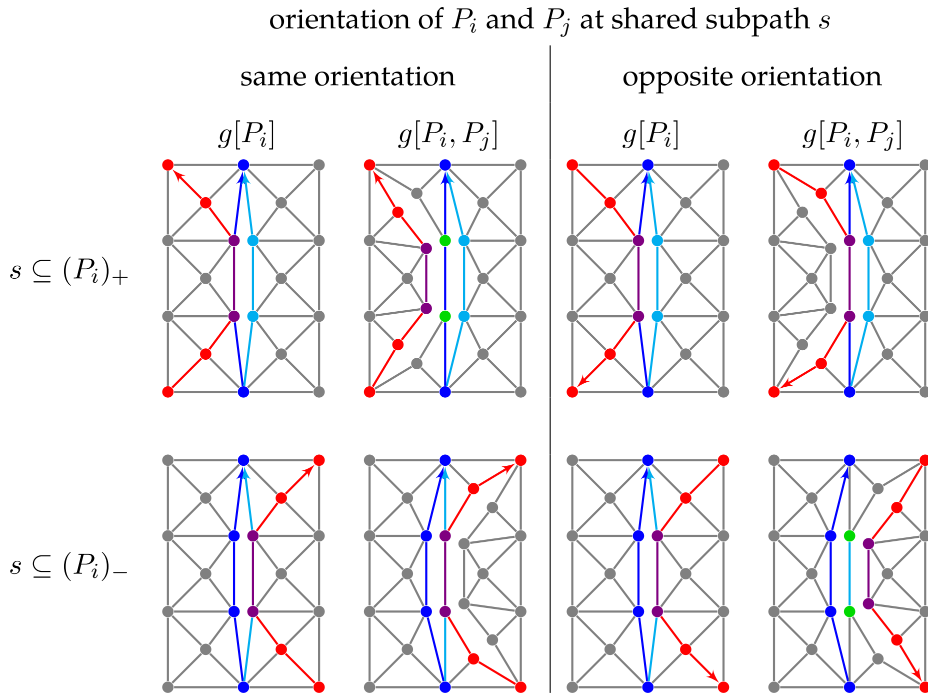


Figure 4.18.: Visualization according to style 2; An illustration of all cases, which have to be considered in algorithm 9. The graph g is split along the two paths P_j (blue) and P_i (red) which share a subpath s (violet). Thereby g is split along P_j first, yielding the left side $(P_j)_+$ (blue) and right side $(P_j)_-$. *Top Left*: P_j is to the right of P_i, so splitting along P_i creates new vertices (green) to the right of s forming a new left side of the split along P_j. Consequently the vertices of s in $(P_j)_+$ have to be mapped to the green vertices to get the new correct left side P_j^+ of the split along P_j (line 12). *Bottom Right*: Analog to the previous case P_j is again to the right of P_i, but this time the right side has to be adjusted as s is part of $(P_j)_-$ (line 15). *Top Right & Bottom Left*: P_j is to the left of P_i and no adjustment to the left/right side of P_j is necessary.

4.5. Optimized Lensing Deformations

Algorithm 10 Optimized Lensing Deformation

1: **function** LENSINGDEFORMATION(G, Γ)
2: select Γ_j with highest weight
3: **for** $\mathcal{D} \in \{LDU, UDL\}$ **do**
4: $G^{\mathcal{D}} = (G_1, \ldots, G_{j-1}, \text{decomposition } \mathcal{D} \text{ of } G_j, \ldots, G_{j+1}, \ldots, G_n)$
5: $\Gamma^{\mathcal{D}} = (\Gamma_1, \ldots, \Gamma_{j-1}, \underbrace{\Gamma_j, \Gamma_j, \Gamma_j, \ldots, \Gamma_j, \Gamma_j, \Gamma_j}_{\#\text{ copies} = \#\text{ factors in } \mathcal{D}}, \Gamma_{j+1}, \ldots, \Gamma_n)$
6: $(\tilde{G}^{\mathcal{D}}, \tilde{\Gamma}^{\mathcal{D}}) = \text{SIMPLEDEFORMATION}(G^{\mathcal{D}}, \Gamma^{\mathcal{D}})$
7: **end for**
8: return $(\tilde{G}^{\mathcal{D}}, \tilde{\Gamma}^{\mathcal{D}})$ with lowest weight
9: **end function**

A single step of the optimized lensing deformation (algorithm 10) aims at improving the dominant part of the contour by trying various decompositions[4] (factorizations) of its jump matrix to which, then, the optimized simple deformation (algorithm 1) is applied. However, the contour parts which originate from such a lensing deformation (e.g. L, D and U in Fig. 4.7) have to satisfy an additional constraint, namely condition (iv) of § 4.3: their spatial order has to be preserved. To calculate shortest paths (line 17 in algorithm 1) or improve a shared subpath (line 24 in algorithm 1) for the contour part Γ_j subject to this additional condition, we distinguish between the following three cases:

We recall that F is the list which contains the indices of all contour parts for which a path has been fixed and that we use the convention that contour parts originating from a lensing deformation have to be given from left to right. Furthermore the difference between the split paths P and the left / right sides P^+ / P^- of the associated splits is of particular importance for lensing deformations; see Fig. 4.21.

- Case 1: $\nexists\, i \in F$ with $\Gamma_i = \Gamma_j$

  As no path belonging to the lensing deformation at Γ_j has been fixed, there are no constraints yet to be observed and we can simply calculate the shortest path for Γ_j in the same way as we do for regular contour parts (lines 3 to 19 in algorithm 2 and lines 10 to 17 in algorithm 6).

- Case 2: $\exists\, i, k \in F$ with $\Gamma_i = \Gamma_j = \Gamma_k$ and $i < j < k$

  We select i and k to be the indices of the contour parts closest to the left respectively to the right of Γ_j (lines 21 to 22 in algorithm 2 / lines 19 to 20 in algorithm 6). The

[4]Our current implementation considers only the LDU and UDL decompositions, as these two turned out to be sufficient for our examples. Nevertheless it is easy to add further decompositions.

shortest path for Γ_j, subject to the constraint that it is to the right of Γ_i and to the left of Γ_k, can only contain vertices inside the circle formed by joining P_i^- and P_k^+. Therefore, we split the graph along this circle which effectively disconnects the inside of the circle from the rest of the graph. In this graph we can then calculate the shortest path connecting either the endpoints of Γ_j (line 25 in algorithm 2) or the endpoints of the shared subpath s (line 22 in algorithm 6). An illustration of this case can be seen in Fig. 4.19.

- Case 3: $\exists\, i \in F$ with $\Gamma_i = \Gamma_j$

  Without loss of generality, we assume that we have to construct a shortest path for Γ_j subject to the constraint that it is to the right of an already fixed path P_i $(i < j)$. Now, let c be a point that is located between the left and right side of the split in the graph g_j caused by P_i. Then, the order constraint is identical to finding the shortest path P_j for which the circle formed by joining p_j and P_i^+ encloses c. Lemma 2, stated at the end of this section, will show that p_j is actually given by

  $$\Pi = \{\mathrm{sp}(g_j, v_i^l, u) \dotplus uw \dotplus \mathrm{sp}(g_j, w, v_i^r) :\ uw \text{ edge in } g_j\},$$

  $$P_j = \mathrm{argmin}\{d(p) : p \in \Pi;\ \mathrm{ind}(p \dotplus \overleftarrow{P_i^+}, c) = 1\}.$$

  Thereby v_i^l / v_i^r is the left / right endpoint of P_i^+. Hence, P_j can be constructed by a minor modification of the polynomial algorithm for shortest enclosing walks in embedded graphs (Provan 1989). An implementation of this modification except for the last step of choosing the shortest path is provided by algorithm 4 and used to calculates a shortest path for the just mentioned case by algorithm 2 in lines 27 and 29. Fig. 4.20 (Top Left) provides an illustration of this case.
  If we just want to improve a subpath s of P_j, we are faced with the problem of finding a shortest path $\widetilde{s}$ connecting the endpoints of s such that with $q = p_r \dotplus \overleftarrow{P_i^+} \dotplus \overleftarrow{p_l}$ the circle $q \dotplus \widetilde{s}$ encloses a point c in between the split along q in $g_j[q]$. Consequently algorithm 6 creates the temporary path q and a split graph $g_j[q]$ and uses the modified shortest enclosing walks algorithm 4 to determine $\widetilde{s}$ (lines 24 to 26). See Fig. 4.20 for an illustration of this case.

Lemma 2. *Let $q = (q_1, \dots, q_n)$ be a path in a weighted planar graph $g = (V, E)$, let c be a point considered*[5] *to be in the split along q in $g[q]$ and define*

$$\Pi = \{\mathrm{sp}(g[q], q_1, u) \dotplus uv \dotplus \mathrm{sp}(g[q], v, q_n) :\ uv \in E\}.$$

Then the shortest walk $p_\star$ in g, subject to the constraint $\mathrm{ind}(p_\star \dotplus \overleftarrow{q_\pm}, c) = \pm 1$, *satisfies*

$$p_\star = \mathrm{argmin}\{d(p) :\ p \in \Pi;\ \mathrm{ind}(p \dotplus \overleftarrow{q_\pm}, c) = \pm 1\}. \tag{4.3}$$

[5] We can chose any point on q for c and treat it as if it were to the right of q_+ and to the left of q_-.

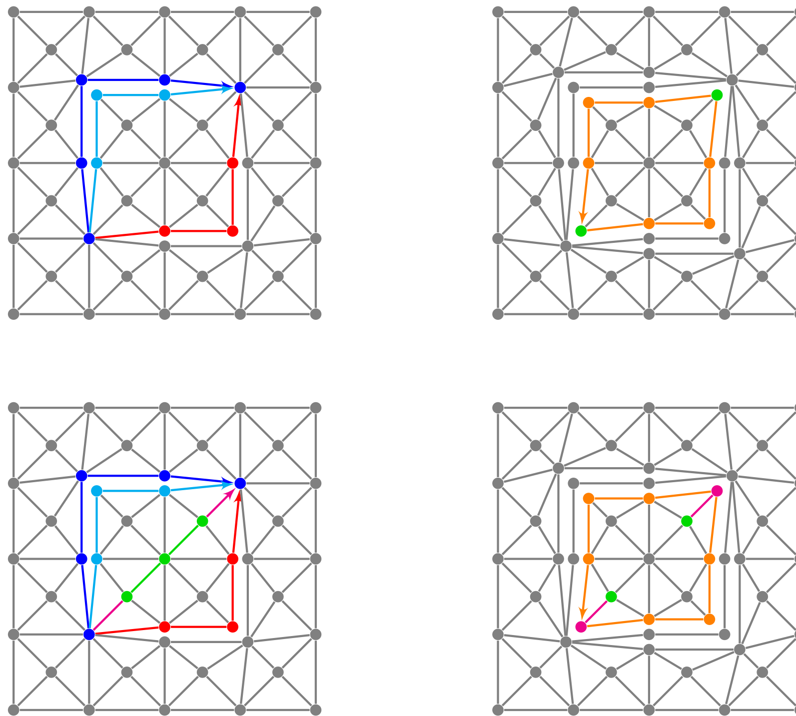


Figure 4.19.: Visualization according to style 2; Illustration of case 2; top row visualizes algorithm 2 and bottom row visualizes algorithm 6. *Top Left*: We search for a path P_j which is to the right of P_i (blue) and to the left of P_k (red). This path has to be in the area enclosed by P_i^- (cyan) and P_k^+ (red). *Top Right*: After splitting the graph along $q = P_k^+ \dotplus \overleftarrow{P_i^-}$ (orange), we can determine the path we are searching for by calculating the shortest path between the endpoints of P_i^- / P_k^+ (green) (line 25). *Bottom Left*: In this case we already have a path P_j (magenta), which is to the left of P_k (red) and to the right of P_i (blue), and we want to improve a subpath s of it (green). *Bottom Right*: Splitting the graph along $q = P_k^+ \dotplus \overleftarrow{P_i^-}$ (orange) enables us to determine an improved subpath by calculating the shortest paths between the endpoints of s (green) (line 22).

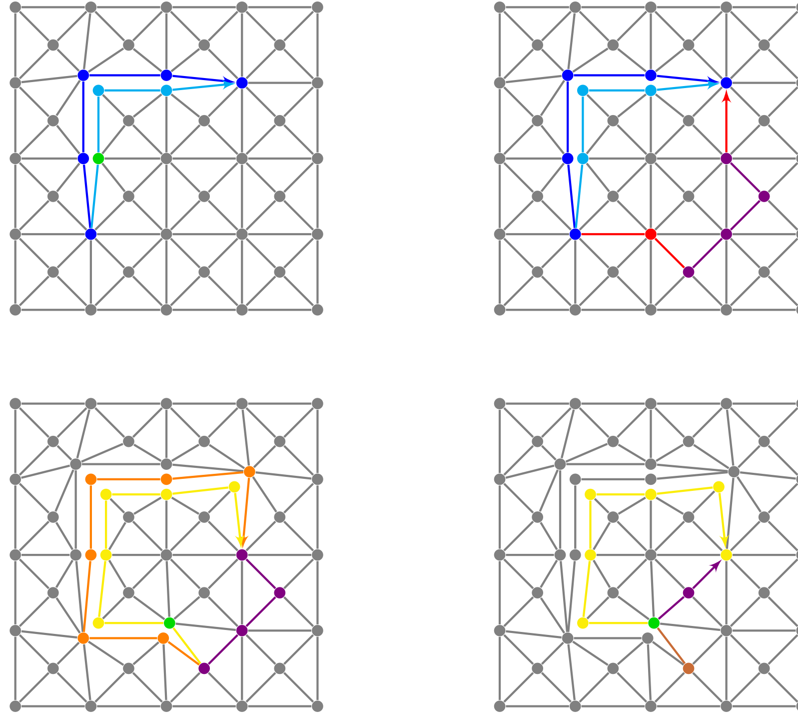


Figure 4.20.: Visualization according to style 2; Illustration of case 3; top left visualizes algorithm 2 and the rest visualizes algorithm 6. *Top Left*: We search for a path P_j which is to the right of the split caused by P_i (blue). This path is the shortest path p such that $\overleftarrow{p} \dotplus P_j^+$ forms a path enclosing c (green). For paths containing c we treat c as if it would be in between the split to determine the angle (line 27). *Top Right*: We have a path P_i (blue) and a path P_j (red) to the right of it. The subpath s (violet) of P_j should be optimized. *Bottom Left*: To optimize the subpath, the graph is split along $\overleftarrow{p_l} \dotplus P_1^- \dotplus \overleftarrow{p_r} = q$ (orange) and the shortest path $\widetilde{s}$ in $g_j[q]$, with which q forms a circle enclosing c (green) is determined (lines 24 to 25). *Bottom Right*: As the first edge (brown) of the new subpath $\widetilde{s}$ (violet) is part of q_- (yellow) this edge has to be mapped back to q_+ to make $\widetilde{s}$ a valid path in g_j (line 26).

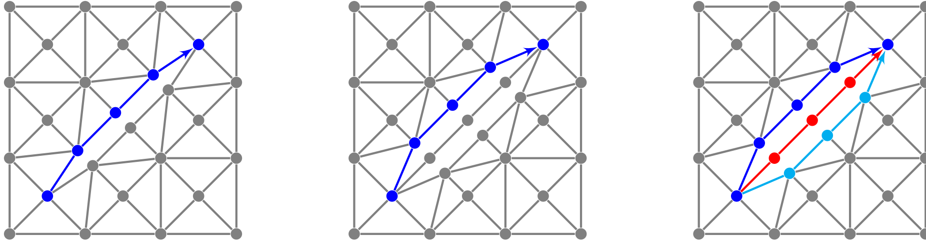


Figure 4.21.: Visualization according to style 2; Example depicting the importance to distinguish between P and P^+/P^- for lensing deformations. *Left*: a graph split once along q (blue). *Middle*: a graph split two times along the same path q (blue). *Right*: Setting $P = (q, q)$ yields the following left and right sides of splits in $g[P]$: P_1^- (cyan), $P_1^+ = P_2^-$ (red) and $P_2^+ = q$ (blue). So if we search for a path which is to the right of the split caused by P_1 we have to actually search for a path which is to the right of P_1^+. E.g. the red path is to the right of the path P_1 in $g[P]$ but not to the right of the associated split.

Proof. We restrict ourselves to the case $\operatorname{ind}(p \dotplus \overleftarrow{q}) = 1$, because the proof for the other case differs just in the sign of some winding numbers. We will assume to the contrary that $p_\star$ is not given by (4.3) and will get a contradiction.

If there is more than one shortest walk $p_\star$, we choose the one which encloses the least number of vertices. If $p_\star \notin \Pi$, then there has to be a vertex $v \in p_\star$ with $p_\star = l \dotplus v \dotplus r$ such that $s_l = \operatorname{sp}(g[q], q_1, v)$ and $s_r = \operatorname{sp}(g[q], v, q_n)$ satisfy the following conditions

$$\operatorname{ind}(s_l \dotplus r \dotplus \overleftarrow{q}, c) \neq 1, \qquad \operatorname{ind}(l \dotplus s_r \dotplus \overleftarrow{q}, c) \neq 1. \tag{4.4}$$

If no such vertex v exists, then either l or r is a shortest path for all partitions $l \dotplus v \dotplus r$ of $p_\star$. Consequently there has to be an edge $uw \in p_\star$, such that $p_\star = l \dotplus uw \dotplus r$ and l, r are both shortest paths.

We will now show that there is no such vertex v.

Step 1. To begin with, we prove for $W = p_\star \dotplus \overleftarrow{q}$ that

$$s_l \cap \operatorname{int}(W) = \emptyset, \qquad s_r \cap \operatorname{int}(W) = \emptyset. \tag{4.5}$$

If (4.5) would not hold then either s_l or s_r contains a path $p = (p_1, \dots, p_m)$ with $p_2, \dots, p_{m-1} \in \operatorname{int}(W)$ and $p_1, p_m \in p_\star$. As s_l and s_r are shortest paths, such a subpath p has to be the shortest path from p_1 to p_m. Consequently, the walk

$$W' = p_\star[q_1, p_1] \dotplus p \dotplus p_\star[p_m, q_n] \dotplus \overleftarrow{q}$$

would satisfy

$$d(W') \leq d(W), \qquad \operatorname{ind}(W', c) = 1, \qquad |\operatorname{int}(W')| < |\operatorname{int}(W)|,$$

which contradicts our choice of $p_\star$. Therefore, (4.5) holds.

Step 2. We now prove that

$$\begin{aligned} \operatorname{ind}(W_1^l, c) = 1 \quad &\text{with } W_1^l = l \dotplus \overleftarrow{s_l}, \\ \operatorname{ind}(W_1^r, c) = 1 \quad &\text{with } W_1^r = r \dotplus \overleftarrow{s_r}. \end{aligned} \tag{4.6}$$

To this end, we consider the walk

$$W^l = l \dotplus \overleftarrow{s_l} \dotplus s_l \dotplus r \dotplus \overleftarrow{q}$$

which consists of the two circles

$$W_1^l = l \dotplus \overleftarrow{s_l}, \qquad W_2^l = s_l \dotplus r \dotplus \overleftarrow{q},$$

and satisfies

$$\operatorname{ind}(W, c) = \operatorname{ind}(W^l, c) = \operatorname{ind}(W_1^l, c') + \operatorname{ind}(W_2^l, c') = 1. \tag{4.7}$$

If $s_l \cap W = \emptyset$, then neither W_1^l nor W_2^l contains any vertex more than once. As g is a planar graph, these walks correspond to simple closed curves in the complex plane and therefore their winding numbers around c can only be -1, 0 or 1. If we also take (4.7) and (4.4) into account, the only possible option is

$$\operatorname{ind}(W_1^l, c) = 1, \qquad \operatorname{ind}(W_2^l, c) = 0.$$

Unfortunately, $s_l \cap W = \emptyset$ does not necessarily have to be satisfied. But s_l cannot cross l and $r \dotplus \overleftarrow{q}$ because of (4.5), which means that

$$s_l = \operatorname{sp}(g[q], q_1, v) = \operatorname{sp}(g[q, \overleftarrow{l}, q \dotplus \overleftarrow{r}], q_1, v).$$

Hence, we can find the following walks in $g[q, \overleftarrow{l}, q \dotplus \overleftarrow{r}]$

$$U_1^l = l_+ \dotplus \overleftarrow{s_l}, \qquad U_2^l = s_l \dotplus r_+ \dotplus \overleftarrow{q_+},$$

which are equivalent to W_1^l and W_2^l but do not contain duplicate vertices. If we move the paths along the splits $\overleftarrow{l}$ and $q \dotplus \overleftarrow{r}$ a little bit apart, then U_1^l and U_2^l correspond to simple connected curves, too. As we can move the split paths apart without crossing c or any other part of U_1^l or U_2^l, these walks can only have winding numbers of -1, 0 or 1 with respect to c. This means that W_1^l and W_2^l can only have these winding numbers even if $s_l \cap W \neq \emptyset$. This proves the first relation in (4.6). The second one follows likewise.

Step 3. We claim that

$$\begin{aligned} q_n \in \operatorname{int}(W_1^l), \quad q_n \notin \operatorname{int}(W_1^r), \\ q_1 \in \operatorname{int}(W_1^r), \quad q_1 \notin \operatorname{int}(W_1^l). \end{aligned} \tag{4.8}$$

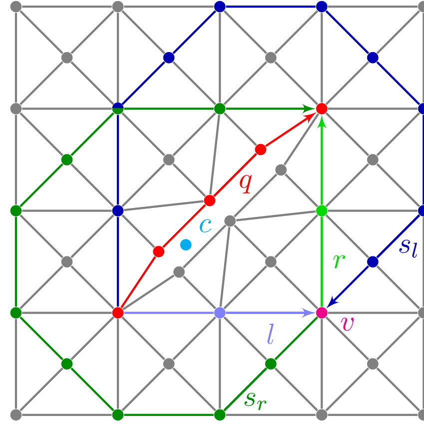


Figure 4.22.: Illustration of Step 4 in the proof of lemma 2.

Combining (4.5) and (4.6) yields $\mathrm{int}(W) \subseteq \mathrm{int}(W_1^l)$. Therefore, all vertices in W either have to be in $\mathrm{int}(W_1^l)$ or in W_1^l. It follows that we just have to show that $q_n \notin W_1^l$. We consider the following closed walk

$$W' = W_1^l \dotplus W_1^r = l \dotplus r \dotplus \overleftarrow{s_r} \dotplus \overleftarrow{s_l}$$

with

$$\mathrm{ind}(W') = \mathrm{ind}(W_1^l) + \mathrm{ind}(W_1^r) = 2.$$

If $q_n \in W_1^l$, then $s_l[q_n, v] = \overleftarrow{s_r}$ and, consequently, W' contains a circle that does not enclose any vertex. Removing this circle from W' results in the walk

$$W'' = l \dotplus r \dotplus \overleftarrow{s_l}[q_n, q_1]$$

without changing the winding number. So $\mathrm{ind}(W'') = 2$, and the argument that we have used before to show $\mathrm{ind}(W_1^l, c) \in \{-1, 0, 1\}$ works for W'', too. Consequently, we get $q_n \notin W_1^l$. The second claim in (4.8) follows likewise.

Step 4. (see Fig. 4.22) We combine the results of the previous steps to show that our initial assumption leads to a contradiction. As $c \in \mathrm{int}(W_1^l) \cap \mathrm{int}(W_1^r)$ by (4.6), the two circles W_1^l and W_1^r have a non empty intersection. Furthermore, because of (4.8), none of them is completely contained within the other. It follows that W_1^l and W_1^r have to cross each other at two or more points. One of these can be v, but there is actually no other vertex at which the circles could cross: s_l cannot cross r and, vice versa, s_r cannot cross l due to (4.5); also s_l and s_r cannot cross because they are both shortest paths. □

4.6. Extensions for infinite contours

There are a two improvements for contours which extend towards infinity, which can be applied to algorithm 2. Currently it finds paths which share the same endpoints for all the factors of a lensing deformation of a ray or line type contour part. But we do have more freedom to choose a vertex corresponding to an endpoint at infinity. The direction with which a factor approaches infinity can be changed as long as the spatial order of the factors is preserved and the corresponding jump matrix decays to the identity matrix along the new direction. Choosing a different direction for each factor may be beneficial, because in general the direction of steepest descent is different for each of them. To verify that the spatial ordering of factors is preserved we create a list of all infinite contour parts and sort them by the direction with which they approach infinity. The resulting list for the original and deformed contour have to be the same up to cyclic shifting (to compensate for the jump of the argument function). If not all parts of the contour have a fixed path yet, we consider only the parts which do. See Fig. 4.23 for an example.

Taking this information into account, the first improvement to algorithm 2 is as follows. If the algorithm encounters a lensing case with an infinite part of the contour, it does not use the methods for the lensing case (lines 20 to 31). Instead it uses the methods for the "no lensing" case (lines 3 to 19), whereby the mapping functions in lines 12, 15 and 16 are altered to return all vertices which satisfy the following constraints

(i) The weight of the vertex[6] is lower than 10^{-16}.

(ii) Choosing the vertex preserves the spatial order of infinite contour parts.

(iii) The direction with which the original part of the contour approaches infinity and the argument of the position of the vertex do not differ more than a given threshold.

The first constraint should select a direction along which the jump matrix decays to the identity matrix and the last constraint prevents pointless paths like e.g. a path starting and ending at the same vertex[7]. For the second constraint there are two cases in which it is violated. The first case is that choosing the vertex does not preserve the spatial order and the second case it that choosing it has the effect that there are no longer any vertices which satisfy all constraints for another endpoint. If there is no vertex satisfying all constraints for one factor at one of its infinite endpoints, then this endpoint is mapped to the same vertex for all factors belonging to the same lensing decomposition. The vertex is chosen so that it corresponds to the direction of the original part of the contour. An example for these vertex mappings is shown in Fig. 4.24.

In accordance with this change, we have to alter algorithm 6, too. If without loss of generality the left endpoint of the subpath s is part of the boundary of g_j and it is also the

[6] Analogous to the edge weight, the weight of a vertex v is $d(v) = \|G(v) - I\|$

[7] Let us assume Γ is just a single line. Without constraint (iii) both its left and right endpoint could be mapped to the same vertex, yielding a very short, but pointless path.

left endpoint of P_j, then we do not just search the shortest path connecting the endpoints of s. Instead we determine the shortest path connecting the right endpoint of s with any vertex which is a suitable endpoint for P_j as described before.

A second improvement, which is rather cosmetic, is to enable the same freedom for all ray and line type contour parts even if they are not factors of a lensing decomposition. As most of the time the contour parts are already given such that their direction roughly resembles the path of steepest descent for their corresponding jump matrices, this freedom does not make a big difference but it results in nicer looking contours.

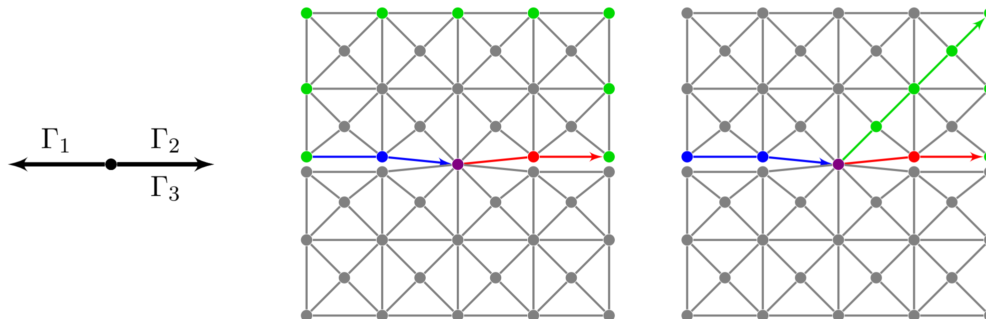


Figure 4.23.: Visualization according to style 2; *Left*: Contour consisting of three rays, whereby the two rays on the right side originate from a lensing decomposition. Sorting them by increasing angle with which they approach infinity yields $(\Gamma_3, \Gamma_2, \Gamma_1)$. We keep in mind that factors of a lensing decomposition are given ordered left to right. *Middle*: After the contour optimization algorithm fixed the paths P_1 (blue) and P_3 (red) for Γ_1 and Γ_3 respectively, it remains to find a path P_2 to the left of P_3. Since we consider each vertex on the boundary to be connected to infinity, any shortest path connecting the left endpoint of P_2 (violet) and one of the green vertices on the boundary is a valid candidate for P_2. These green vertices are also all the vertices for which sorting the paths by the argument of their endpoint on the boundary yields (P_3, P_2, P_1), which is the same as the one of the original contour. *Right*: A large difference between the argument of the right endpoint of P_2 and the angle with which Γ_2 approaches infinity, blocks a significant area of the graph for other paths. The part of the graph between P_2 and P_3 can not be used by other paths. Therefore we limit this deviation. In our implementation we choose a maximum deviation of $\pm\pi/4$. For our example this reduces the set of possible endpoints for P_2 to the green vertices. The green path is an example for a valid choice for P_2. The hardcoded value of $\pi/4$ could be improved by analysing the oscillator of the jump function. However we do not expect a significant improvement from doing so. Therefore this is left as future work. We would also like to note here, that this examples illustrates only how to choose vertices in order to preserve the spatial ordering of all parts of the contour. There are further points that have to be taken into account, as illustrated in Fig. 4.24.

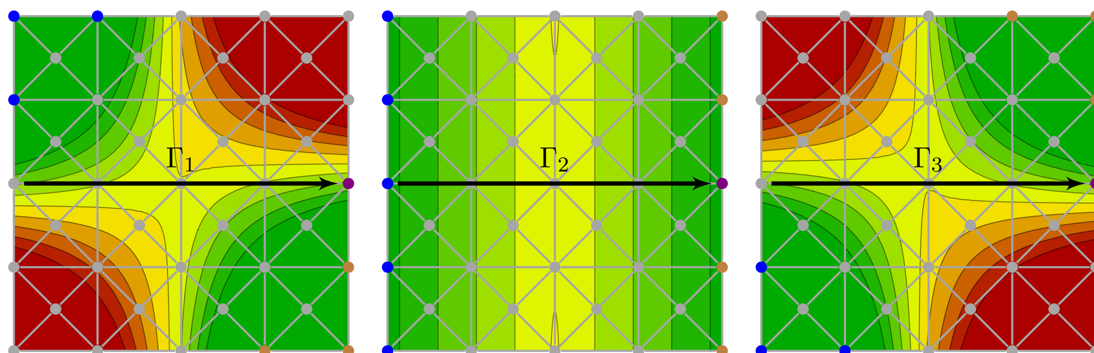


Figure 4.24.: Illustration of a lensing deformation for an infinite contour; A line from $-\infty$ to $+\infty$ is decomposed into the 3 factors Γ_1,Γ_2,Γ_3, which are given left to right. That means Γ_1 to the left of Γ_2 which in turn is to the left of Γ_3. The background contains a contour plot of the weight corresponding to the jump matrix associated with the lensing factor. The color of the weight is the same as used by Style 1. The blue vertices are valid mappings for the left endpoint of a factor. They have a weight lower than 10^{-16} and choosing one of these mapping does not leave another factor without a valid mapping. At the right endpoint we have a different situation. Though there are vertices (brown) with a sufficiently low weight for each of the factors, we cannot choose one of them for each factor and preserve the spatial order of the factors at the same time. Therefore the right endpoint of all factors is mapped to the same vertex (violet) which corresponds to direction of Γ_j.

4.7. Deformation Verification

In this section we discuss some details of the method we use to verify that constraint (ii) from § 4.3.1 is satisfied for the paths calculated by algorithms 2 and 6. We recall this means that there is no endpoint of another part of the contour in the region between the original and the deformed version of a part of the contour. At first glance this does seem to be fairly easy. For the most part this is actually true, but there are a few not so obvious but important details that have to be considered.

To verify the constraint for a contour consisting of n parts (continuous, discrete or a mix of both), we map the contour to the $n \times n$ matrix

$$\mathrm{V} : \Gamma = \Gamma_1 \cup \cdots \cup \Gamma_n \mapsto \begin{pmatrix} \nu(\Gamma_1, \Gamma_1) & \cdots & \nu(\Gamma_1, \Gamma_n) \\ \vdots & & \vdots \\ \nu(\Gamma_n, \Gamma_1) & \cdots & \nu(\Gamma_n, \Gamma_n) \end{pmatrix}$$

where the i, j element $\nu(\Gamma_i, \Gamma_j)$ contains all the angles we get by moving a line connecting Γ_i to an endpoint of Γ_j from the beginning of Γ_i to its end. These angles are the same for the original and the deformed contour if and only if constrained (ii) is satisfied. See Fig. 4.25 for a few examples.

The exact definition of $\nu(\Gamma_i, \Gamma_j)$ can be found in appendix B, it is rather long but pretty much straight forward. There are four different cases, which occur when we evaluate it:

1. Γ_i and Γ_j are continuous
2. Γ_i and Γ_j are discrete
3. Γ_i is continuous and Γ_j is discrete
4. Γ_i is discrete and Γ_j is continuous

Of these four cases the first one does not pose any problems. The third and fourth one can be reduced to the first two by mapping the endpoints of Γ_j to the corresponding coordinates of the vertex or the coordinates to the corresponding vertices respectively. The remaining case 2 is of interest. If both Γ_i and Γ_j are paths, which we denote now P_i and P_j, then there can be an endpoint v of P_j which is also part of P_i or which is at the same location as another vertex in P_i. To determine the angle of P_i around v we have to decide if we consider v to be left or to the right of P_i. Fig. 4.26 contains examples of the different cases which have to be considered for this decision. To handle all the cases we need an order from left to right of all the vertices at the same location as v with respect to P_i. Such an order is calculated by algorithm 11.

Order of Vertices at Splits

Algorithm 11 calculates a sorted list of all vertices which are at the same location as $v \in q$ in the graph $g[P]$. Thereby the vertices are sorted from left to right with left and

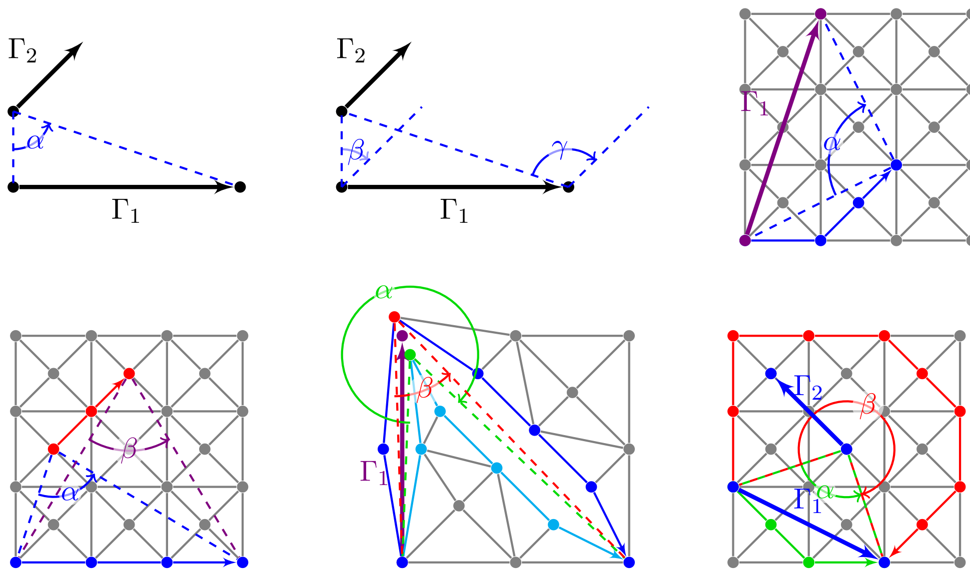


Figure 4.25.: Visualization according to style 2; *Top Left*: $\nu(\Gamma_1, \Gamma_2) = (\alpha)$ *Top Middle*: $\nu(\Gamma_2, \Gamma_1) = (\beta, \gamma)$ *Top Right*: $\nu(\Gamma_1, \Gamma_2) = (\alpha)$ with Γ_2 being the blue path *Bottom Left*: $\nu(\Gamma_1, \Gamma_2) = (\alpha, \beta)$ with Γ_1 being the blue path and Γ_2 the red path *Bottom Middle*: $\nu(\Gamma_2, \Gamma_1) = (\alpha, \beta)$ with Γ_2 being the blue path; The right endpoint of Γ_1 can be mapped to either the red or the green vertex, resulting in different angles. This information can be used to filter invalid vertex mappings. *Bottom Right*: The green path is a valid choice for Γ_1 as it does preserve the angle α in contrast to the red path which does not preserve it.

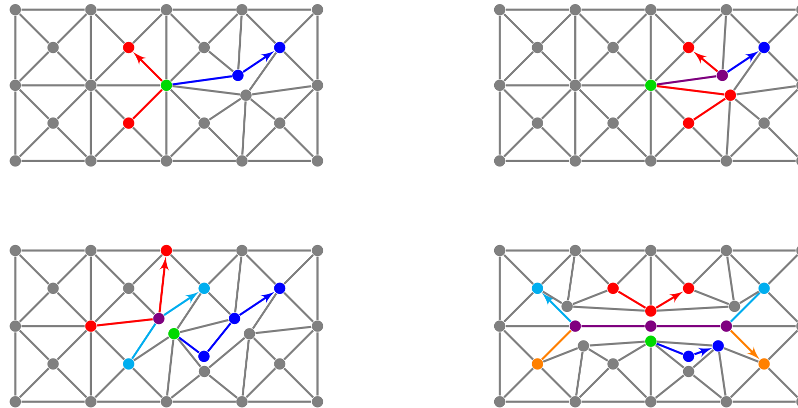


Figure 4.26.: Visualization according to style 2; All 4 graphs show different situations which can occur, when we determine the angle between a path p (red) and the left endpoint of the path q (blue), which is its first vertex v (green). All graphs are split along the blue, cyan and orange paths. Furthermore vertices and edges which are used by two paths are coloured violet. With this visualization it is not too difficult to decide if v is to the left or to the right of p. But we should keep in mind that this information was also used to create this illustration. *Top Left*: v is contained in p, but it is considered to be right of p, as the first edge of q is on the right side of p. *Top Right*: v is also considered to be right of p. If p contains an edge of q_+, this edge is considered to be on the right side and if p contains an edge of q_-, this edge is considered to be to the left of p. Analogously, if p contains the reversed edge of q_+ / q_- the edge is considered to be of the left / right side. *Bottom Left*: In this case v is not contained in p. Instead p contains a vertex (violet) which is at the same location as v. Since v is on the right side of the split along the cyan path and p is on the left side of it, we can determine that v is to the right of p. *Bottom Right*: The graph is split two times along the violet path in opposite directions. Therefore, there is no direct relation between a vertex in p and v as in the previous case. To decide if v is to the left or to the right of p we determine an order of all vertices at the location of v with respect to p using algorithm 11.

Algorithm 11 Order of vertices at splits

1: **function** VERTEXORDER(g, P, q, v)
2: $O = (v)$ ▷ *initial ordering of vertices at* v
3: $N = (v)$ ▷ *list of vertices added to* O *in the previous iteration*
4: $F = ()$ ▷ *list of vertices added to* O *in the current iteration*
5: $I = \begin{cases} (j) & \text{if } v \in p_{j_-} \in P = (p_1, p_2, \dots) \\ (0) & \text{else} \end{cases}$
6: $R(v) := q$ ▷ *reference path to determine the orientation at* v
7: **while** $N \neq \emptyset$ **do**
8: **for all** $p_i \in P = (p_1, p_2, \dots)$ **do** ▷ *process all paths in* P
9: $p = p_i$
10: **for all** $w \in (p_+ \cup p_-) \cap N$ **do** ▷ *which contain a vertex in* N
11: **if** $p_+[w] \neq p_-[w]$ **then**
12: **if** $w \in p_+$ **then**
13: $u = p_-[w]$
14: **if** SIMILARORIENTATION($p_+, R(w), w$) **then**
15: $r =$ True ▷ u *is to the right of* w *with respect to* $R(w)$
16: $R(u) := p_-$ ▷ *set* p_- *as reference path for* u
17: **else**
18: $r =$ False ▷ u *is to the left of* w *with respect to* $R(w)$
19: $R(u) := \overleftarrow{p_-}$ ▷ *set* $\overleftarrow{p_-}$ *as reference path for* u
20: **end if**
21: **else** ▷ $w \in p_-$
22: $u = p_+[w]$
23: **if** SIMILARORIENTATION($p_-, R(w), w$) **then**
24: $r =$ False ▷ u *is to the left of* w *with respect to* $R(w)$
25: $R(u) := p_+$ ▷ *set* p_+ *as reference path for* u
26: **else**
27: $r =$ True ▷ u *is to the right of* w *with respect to* $R(w)$
28: $R(u) := \overleftarrow{p_+}$ ▷ *set* $\overleftarrow{p_+}$ *as reference path for* u
29: **end if**
30: **end if**

Algorithm 11 Ordering of vertices at splits (continued)

31: l = index of w in O

32: $m = \begin{cases} (j) & \text{if } u \in p_{j_-} \in P = (p_1, p_2, \dots) \\ (0) & \text{else} \end{cases}$

33: **if** $u \notin F$ **then**

34: **if** r = True **then** ▷ *sort u into O and m into I*

35: $k = \begin{cases} |O| & \text{if } l = |O| \text{ or } I_j > i \text{ for } l < j \leq |O| \\ -1 + \operatorname*{argmin}_{l<j\leq|I|} I_j < i & \text{else} \end{cases}$

36: insert u and m immediately after position k into O and I

37: **else**

38: $k = \begin{cases} 1 & \text{if } l = 1 \text{ or } I_j > i \text{ for } 1 \leq j < l \\ 1 + \operatorname*{argmax}_{1\leq j<l} I_j < i & \text{else} \end{cases}$

39: insert u and m immediately before position k into O and I

40: **end if**

41: Append u to F

42: **end if**

43: **end if**

44: **end for**

45: **end for**

46: $N = F$; $F = ()$

47: **end while**

48: Return O

49: **end function**

right being defined by the path q. The basic idea of the algorithm is to start with a list O containing only the vertex v. Then all vertices u which are on the left / right side of a split which contains v are sorted into O. This process is repeated for all vertices just sorted into O until there are no more vertices to sort. In more detail the algorithm works as follows.

1. (lines 2 to 6) The data structures which are initialized in these lines have the following contents. O is the ordered list of all vertices, N a list containing all vertices sorted into O during the previous iteration and F contains all vertices which have been sorted into O in the current iteration. I contains for each vertex w in O the in index j of the path $p_j \in P$ for which $w \in (p_j)_-$ holds. Last, R maps each vertex w in O to a reference path which contains w and has an orientation similar to q at w.

2. (lines 8 to 13 and 22) For all paths in P we determine if the left or right side of their corresponding split in $g[P]$ contains a vertex w in N. If they do, u is set to be the vertex coressponding to w on the other side of that split.

3. (lines 14 to 30) We determine if u is to the left or to the right of w with respect to $R(w)$. We know that $p_-[w]$ is to the right of $p_+[w]$ in reference to p. If p and $R(w)$ have a similar orientation, then $p_-[w]$ is also to the right of $p_+[w]$ with respect to $R(w)$, otherwise the opposite is true. In addition, we also store a reference path for the vertex u. Since all reference paths are chosen to have an orientation similar to each other and the initial reference path is q, the sorting is done with respect to q. What we mean by two paths having a similar orientation is explained in Fig. 4.29.

4. (lines 31 to 40) The next step is to figure out how far to the left / right u is. In general u will be immediately to the left / right of w. Only if we move from an inner split to an outer split there can be vertices in between u and w. See Fig. 4.28 for an example and further details.

5. (line 7) This process is repeated for all newly inserted vertices (N) until there are no more vertices to sort into O.

6. (line 11) The purpose of this line is to filter out all paths which do not create a new vertex at the location of v. E.g. if a split starts / ends at v, the algorithm would process this path, but the path does not create a vertex which can be added to O.

Remark. *The list O calculated by algorithm 11 can always be used to determine which vertices are to the left and to the right of v with respect to q and it will determine a total ordering if such an ordering exists. But that is not always the case. See Fig. 4.27 for an example.*

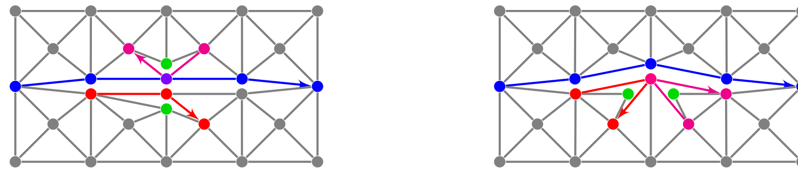


Figure 4.27.: Visualization according to style 2; The graphs are split along p_1 (blue), p_2 (red) and p_3 (purple) in a way such that there are 4 vertices in the split graph which are at the same location; two green, one purple and one red vertex (left) respectively two green, one blue and one pink (right). *Left*: In this case we can determine a total ordering of all 4 vertices in reference to the blue path. *Right*: Though we can say that both green vertices are right of the pink vertex in reference to the blue path, both green vertices are equally to the right of the blue path. Therefore we cannot determine a total ordering of the 4 vertices in reference to the blue path.

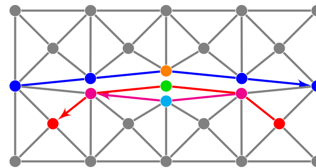


Figure 4.28.: Visualization according to style 2; An example which illustrating the effect of line 35 in algorithm 11. The input to the algorithm is in this case the graph g, the paths $P = (p_1(\text{blue}), p_2(\text{magenta}))$, a reference path q (red) and the vertex v (green). During the first iteration, the algorithm initializes $O = (v), I = (2)$ and inserts $u_1 = p_{2_+}[v]$ (cyan) to the left of v in O and 1 to the left of 2 into I, respectively. This yields $O = (u_1, v), I = (1, 2)$ after the first iteration. During the second step, the algorithm recognizes that $u_2 = p_{1_+}[u_1]$ (orange) is to the right of u_1. As g is split along p_1 first and along p_2 afterwards, u_2 is also to the right of v. Due to $I_2 > i = 1$ the algorithm takes this fact into account (line 35) and determines that u_2 has to be inserted immediately after position $k = 2$ into O.

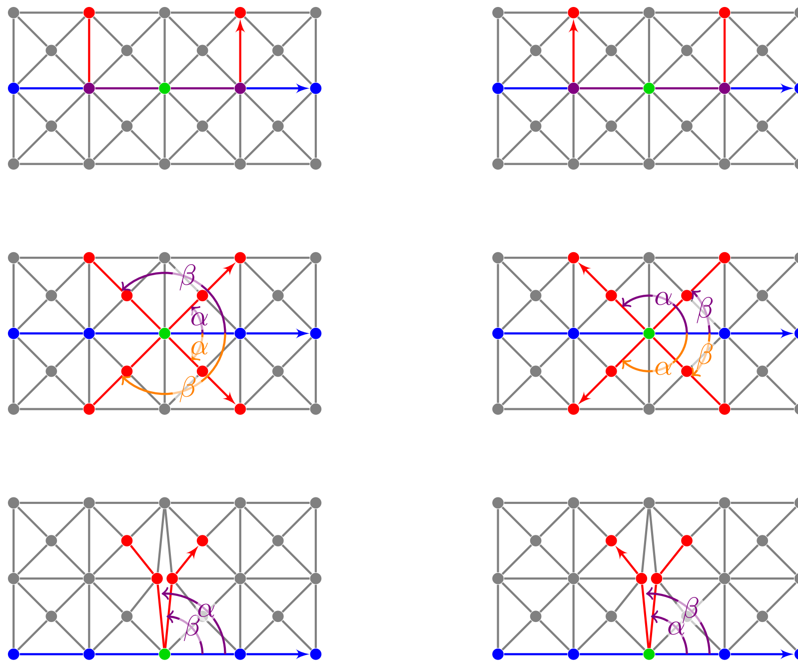


Figure 4.29.: Visualization according to style 2; The plots illustrates the situations when two paths $p = \ldots avb \ldots$ (blue) and $q = \ldots rvs \ldots$ (red) have a similar or not similar orientation at their common vertex v (green). In the cases on the left, p and q have a similar orientation at v and in the cases on the right they do not. Essentially, if p and q have a similar orientation, either the right side of p is part of the right side of q or vice versa the right side of q is part of the right side of p. *Top*: If p and q have a common subpath (violet) containing v, they have a similar orientation at v if the subpath is traversed by both in the same direction. Otherwise they do not have a similar orientation at v. *Middle*: This is a generalization of the previous case. If p and q do not have a common subpath, they have a similar orientation if q is to the left of p and $\measuredangle bvs = \alpha < \beta = \measuredangle bvr$ or if q is to the right of p and $\beta < \alpha$. The conditions for a not similar orientation are analogous. *Bottom*: The case $\alpha = \beta$ can occur, but only if q wraps around a split, since we do not allow paths containing circles. In this case we check if s is to the right or to the left of q at r to determine whether we are in a limiting case of $\alpha < \beta$ or $\beta < \alpha$. **Remark.** *This check is performed with algorithm 11, which also involves comparing the orientation of paths. Therefore we could end up in an infinite loop, if this situation also occurs when we execute algorithm 11 at the vertex r. This in turn means that s is also an endpoint of another path. Consequently, there are two endpoints of contour parts, which are mapped to adjacent vertices (s and v). Our implementation of algorithm 1 detects this situation and subdivides the edge between those vertices in the setup phase, thus avoiding infinite loops. We could also resolve the case leading to infinite loops properly, but that is far more work.*

4.8. Implementation Details

4.8.1. The Weights

The weight $d(e)$ of an edge e should be an approximation of

$$\int_e \|G(z) - \mathrm{I}\| \, d|z| \tag{4.9}$$

with a suitable matrix norm. Our experiments indicate that we generally need fewer collocation points if we aim at minimizing all components of $G - I$ instead of just focussing on its largest component, for which reason we choose the Frobenius (or Hilbert-Schmidt) norm. The integral is sufficiently well approximated by the two point trapezoidal quadrature rule (we recall that the aim of optimizing the weight is just preconditioning, that is, getting a particular good *order of magnitude* of the condition number). We thus take

$$d(e) = \frac{1}{2}|b - a|(\|G(b) - \mathrm{I}\|_F + \|G(a) - \mathrm{I}\|_F)$$

as the weight of an edge e with the endpoints a and b.

4.8.2. The Graph

The algorithm of § 4.4 is based on *planar* graphs. If the graphs were not planar, paths could cross each other even without having any vertices in common and, therefore, the graph splitting described in § 4.3 would not ensure that paths calculated by algorithm 1 do not cross each other. We choose planar graphs built from rectangular grids to which a vertex in the center of each box is added that is connected to the vertices of the border of that box. Such a graph is chosen to subdivide a rectangle that contains all finite endpoints of Γ. We take this rectangle large enough so that outside of it $\|G - \mathrm{I}\|_F$ is below machine precision on all arcs with an infinite endpoint, see Fig. 4.30. For numerical purposes, the jump matrix G is then indistinguishable from the identity matrix in the exterior of this rectangle: the RHP needs to be solved only in the interior.

If the contour contains an infinite endpoint, we add a leaf edges to each vertex on the boundary of the graph. Every path for an infinite part of the contour will contain one of these leaf edges but none of the other paths will. This facilitates mapping paths back to a contour of a RHP, as shown by the example in Fig. 4.31.

4.8.3. Contour Simplification

The algorithm described in § 4.4 returns a contour composed of a set of paths in the underlying graph. The collocation method of Olver (2011b), which is finally employed for the numerical solution of the RHP, would have to place individual Chebyshev points on each smooth (that is, linear) part of this piecewise linear contour. For efficiency reasons it would thus be preferable to have a contour with fewer breakpoints. Consequently,

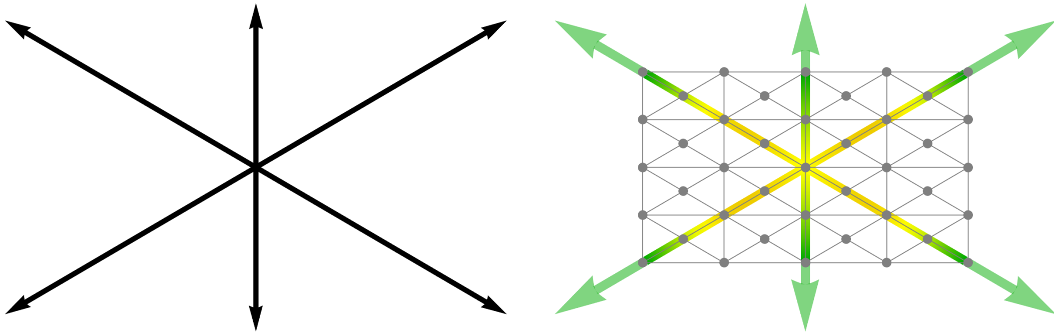


Figure 4.30.: The rectangle to be covered by the grid is determined by the condition $\|G - \mathrm{I}\|_F > 10^{-16}$ along Γ. The color coding is as in Fig. 4.3.

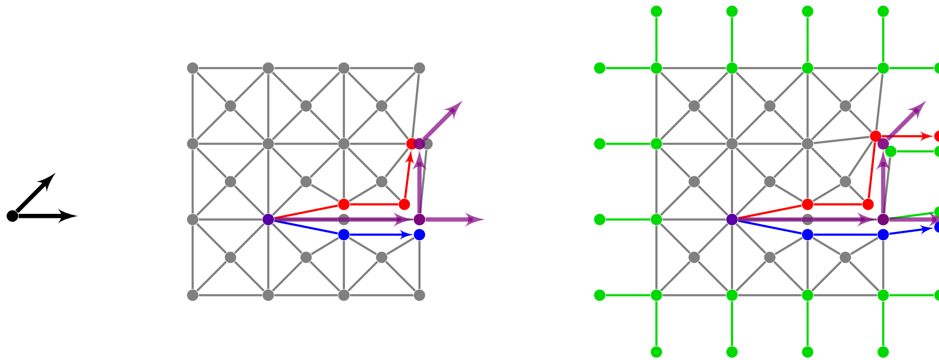


Figure 4.31.: Visualization according to style 2; Example illustrating one advantage of adding leaf edges to the base rectangular graph; *Left*: Example RHP consisting of two rays which is discretized by the next two graphs. *Middle*: After applying the contour optimization algorithm, we get the two paths P_1^- (blue) and P_2^+ (red). These paths are mapped to a RHP with the violet contour. As we can see there are parts of the violet contour which do not have a corresponding edge. Furthermore some vertices on the boundary of the graph are mapped to rays, but not all of them. *Right*: Applying the contour optimization on a graph with added leaf edges (green), results in similar paths P_1^- (blue) and P_2^+ (red). The paths are mapped to the same contour (violet) as in the example in the middle, but this time there is at least one edge for each part of the contour. In addition all vertices are just mapped to points, with points on the boundary being always mapped to infinity. Overall this situation can be handled more elegantly as it contains less implicit information.

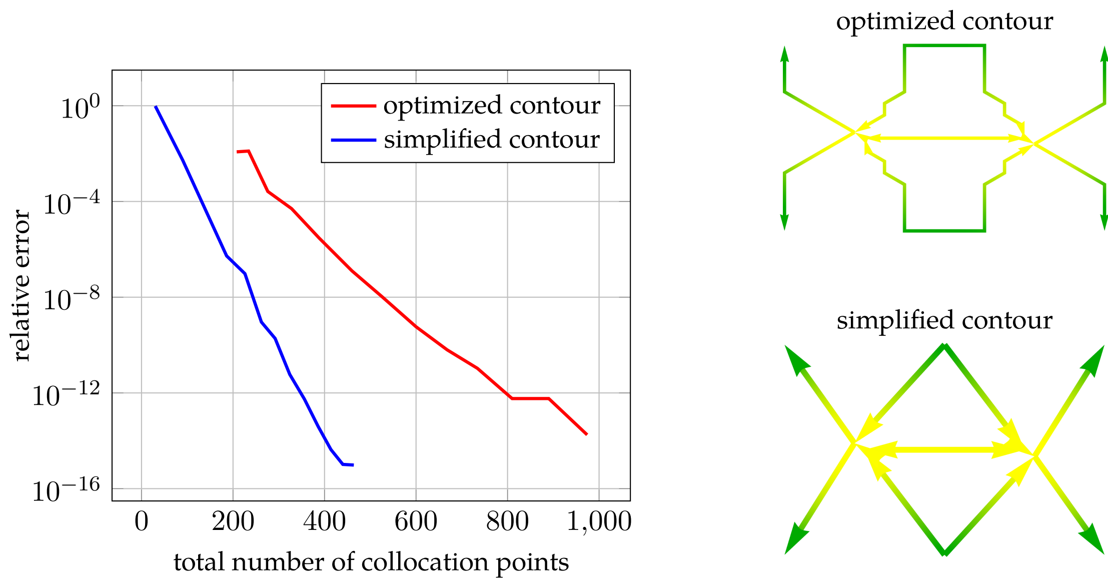


Figure 4.32.: Improvement of the convergence rate by contour simplification: the simplified contour needs only about half the number of collocation points to reach the same accuracy as the optimized contour of Fig. 4.3.d (the color coding is the same as there). This simplification does not, however, worsen the order of magnitude of the condition number which grows from about 140 to just about 200. The similarity with the manually constructed contour in Fig. 4.4 is even more striking after this simplification step.

for each optimized path, we calculate a coarse piecewise linear approximation that has about the same weight. Quite often just a straight line connecting the endpoints of a path is already a sufficient approximation. Fig. 4.32 shows an example of this simplification process when applied to the final contour of Fig. 4.3: it cuts the number of collocation points by more than a factor of two while keeping the order of magnitude of the condition number constant.

4.8.4. Restrictions on Paths

Splitting a graph along a path containing a circle needs to be treated differently from a regular path without circles. See Fig. 4.33 for details. We want to avoid the associated extra complexity. In general a simpler algorithm is easier to check for errors, which is of particular interest as we want the algorithm to be treated as a black box. Furthermore, the paths calculated by algorithms 2 and 6 do not contain circles anyway. Just temporary paths as e.g. the one in line 25 of algorithm 2 can contain any. The cases in which paths of this type arise can be handled without splitting a path along them. Fig. 4.34 illustrates

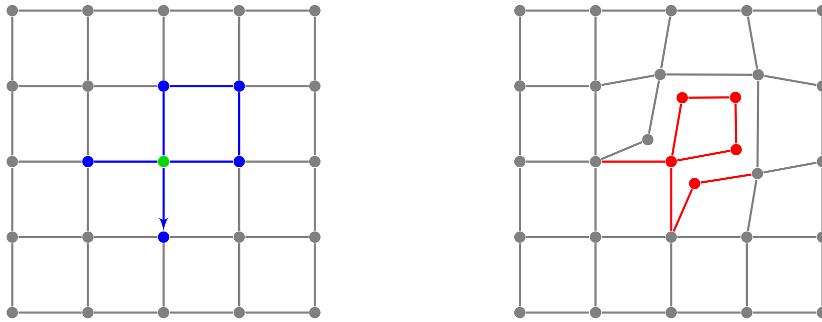


Figure 4.33.: Visualization according to style 2; Illustration of a split along a path containing a circle with negative orientation. *Left*: Graph g with and a path p (blue) containing a circle. *Right*: Splitting g along p yields this graph $g[p]$ and creates new vertices as well as new or modified edges (red). The vertex v (green) visited twice by p requires some attention. As it is visited twice, two new vertices are created for it and not just one. Therefore we lose some properties, if we allow this type of path for splitting graphs. $p_+ = p$ is no longer true in $g[p]$ and there is no longer a one to one mapping between vertices in p_+ and p_-. Obviously a vertex does not define a unique position in such a path. All of this would have to be taken into account by the algorithms we presented in this chapter. As we want to avoid this complexity, we simply do not allow splitting a graph along a path containing circles.

an example. Therefore we do not support splitting graphs along paths containing a circle and the algorithms presented in this chapter do not take paths of this type into account. Splitting a graph along circles is supported though.

4.8.5. Restriction to Machine Precision

The shortest path algorithms in Mathematica require the weights of a graph to be machine precision numbers. Though we are only interested in the magnitude of the total weight of a path, machine precision numbers cause problems. If the range of values of the weights in a path P is larger than $1/\text{eps}$ (with eps being machine epsilon[8]), then all subpaths $P[u, v]$ of P, for which $d(P[u, v]) < \text{eps}\ d(P)$ holds, do not necessarily have to be shortest paths. The shortest path algorithm can not distinguish between the shortest path connecting u and v and any other path Q connecting u and v having $d(Q) < \text{eps}\ d(P)$. Therefore, these subpaths are often similar to random walks, which is not desirable. Random walks contain more turns, which leads to contours with more parts and makes solving the resulting deformed RHP more expensive. Furthermore, random walks will likely not coincide with other paths and therefore they will not trigger the shared subpath

[8]For IEEE 754 double-precision arithmetic, we have eps $\approx 2 \cdot 10^{16}$.

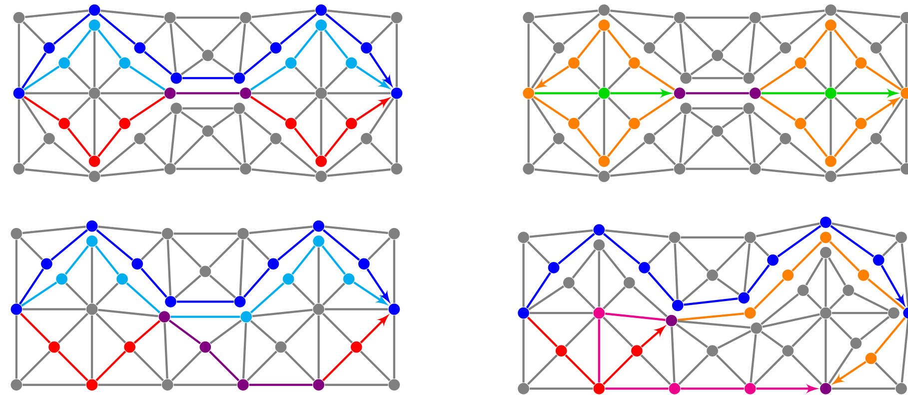


Figure 4.34.: Visualization according to style 2; *Top Left*: Situation handled by line 25 of algorithm 2 with a graph split along the paths P_i (blue) and P_k (red). We search for a path P_j to the right of P_i^- (cyan) and to the left of P_k^+ (red). Per default the algorithm constructs a path $q = P_k^+ \dotplus \overleftarrow{P_i^-}$ and splits the graph along it. But we want to avoid this split as q contains a subpath which is a circle due to P_i^- and P_k^+ having a shared subpath (violet). *Top Right*: Instead of splitting the graph along q we split it along circular subpaths (orange) of q. Calculating the shortest paths (green) inside these circles (orange) and combining the results with shared subpath (violet) of P_i^- and P_k^+ yields the path we are looking for. In order to show this path, the graph displayed here is not split along the orange paths. *Bottom Left*: Situation handled by lines 24 to 26 of algorithm 6. We search for a path $\widetilde{s}$ which improves the subpath s (violet) of P_j (red) within the graph split along the paths P_i (blue). Thereby replacing s in P_j by $\widetilde{s}$ has to preserve the property that P_j is to the right of P_i^-. In this case the algorithm would split the graph along $q = \overleftarrow{p_l} \dotplus \widetilde{P_i^-} \dotplus \overleftarrow{p_r}$ (with p_l / p_r being the red parts at the start / end of P_j and P_i^- being the blue path). As q contains a circle we want to avoid splitting along this path. *Bottom Right*: This is accomplished by removing the circle from q and splitting the graph along the remaining path (orange). In the resulting graph we search the shortest path (magenta) to the right of the just created split and get the path $\widetilde{s}$ we are searching for. Replacing s in P_j by $\widetilde{s}$ can create a circle as in this example. This circle is removed later on (line 34).

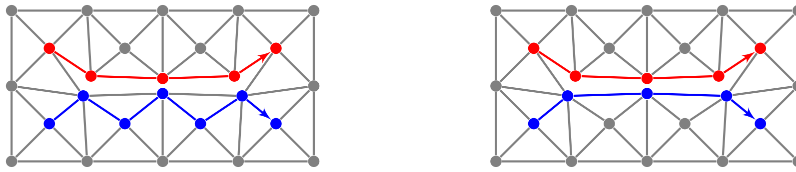


Figure 4.35.: Visualization according to style 2; *Left*: p (red) and q (blue) wiggle along each other in the graph $g[p]$ and don't have a common subpath. *Right*: The wiggling is removed by snapping the wiggling part to p_-.

improvement algorithm 6.

Nevertheless, as matching the speed of this highly optimized implementations of the shortest path algorithms in Mathematica is rather difficult and requires quite a lot of work, we want to use this implementation. Therefore we perform two additional steps to prevent the problems we have just discussed. The first step is to clip the weights of edges to machine precision numbers. The second step is to post process all shortest paths P we get from Mathematica. If it contains a subpath $P[u, v]$ with $d(P[u, v]) < \text{eps}\ d(P)$, we replace this subpath with the shortest path connecting u and v. Obviously, the resulting path does not have to be the shortest path connecting the endpoints of P but its total weight will be the same up to machine precision. This process eliminates random walks.

4.8.6. Removal of Wiggling Paths

A situation which occurs quite frequently is that two paths wiggle along each other in a region, but do not have a common subpath. See Fig. 4.35 for an example. As there is no common subpath, the algorithm will not attempt to optimize any subpath, although the combined wiggling subpaths are promising candidates for optimization. To use this potential we check if a new path shows this behaviour and in case it does, we snap the new path to the one it is wiggling along. Once again see Fig. 4.35 for an example. To check if such a wiggling behaviour occurs between two paths P_i and P_j, we take all the edges of their left and right sides $P_i^{\pm}, P_j^{\pm}$ and all select three tuples of these edges that form triangles. The wiggling behaviour occurs at each of these triangles.

Another option would be to alter the subpath detection routine, so that it takes this behaviour into account. But this is less useful, as in case the subpath improvement step does not yield a better path, the two paths wiggling along each other will remain and cause quite many small contour parts in the resulting RHP. We want to avoid this, because it increases the computation time in comparison to one contour part for the combined subpath.

4.8.7. Runtime Optimizations

There are a few optimizations which speed up the runtime of the algorithms presented in this section.

- Algorithm 1 calculates a new shortest path for each contour part after a path has been fixed. But we can instead also just keep the path we calculated previously if it still valid.
- Algorithm 2 calculates the shortest paths for all possible mappings of the endpoints of Γ_j in lines 9, 13 and 17 and later on checks if the paths are actually valid. But invalid mappings can also be filtered out beforehand by comparing the j-th column of $V(\Gamma)$ and $V(\widetilde{\Gamma})$. Thereby $\widetilde{\Gamma}$ is the intermediate deformed contour as constructed by algorithm 3 in line 6.
- Algorithm 6 only considers improving a shared subpath of two paths. It can be extended to improve a subpath of arbitrary many paths.

5. Numerical Results

In this chapter we apply our contour deformation algorithm to a few RHPs and evaluate the resulting contours regarding its associated condition number and the convergence speed of its numerical solution. As our algorithm is not (yet) able to perform all types of deformations which are available for RHPs, we will compare our automatically derived contours to analytically derived contours which only use the deformations described in § 4.3. Assuming that the analytically derived contours cannot be improved (or just a little), this should give a good impression on how big the difference is between what the algorithm could achieve and what it actually achieves.

5.1. Painlevé II

The results for the RHP corresponding to the Painlevé II equation

$$u_{xx} = xu + 2u^3$$

have already been show throughout this thesis, so we will only provide a short summary of the results. We recall from the introduction that the RHP from (Fokas et al. 2006) is given by

RHP 2 (Painlevé II). *Find a holomorphic function $\Phi : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(2, \mathbb{C})$ satisfying $\Phi(\infty) = I$ and $\Phi^+(z) = \Phi^-(z)G(z)$ for $z \in \Gamma$ where $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_6$, $\Gamma_j = R_{e^{(2j-1)\pi/6}}(0)$ and*

$$G(z) = \begin{cases} \begin{pmatrix} 1 & s_j e^{-\theta(z)} \\ 0 & 1 \end{pmatrix} & \text{for } z \in \Gamma_j \text{ and } j \text{ even} \\ \\ \begin{pmatrix} 1 & 0 \\ s_j e^{+\theta(z)} & 1 \end{pmatrix} & \text{for } z \in \Gamma_j \text{ and } j \text{ odd.} \end{cases}$$

Thereby G contains the phase function

$$\theta(z) = \frac{8i}{3} z^3 + 2ixz$$

and the parameters s_j $(j = 1, \ldots, 6)$, which are interrelated by

$$s_1 - s_2 + s_3 + s_1 s_2 s_3 = 0, \quad s_4 = -s_1, \quad s_5 = -s_2, \quad s_6 = -s_3.$$

As defined in § 4.4.3 we use the notation $R_\alpha(0)$ for a ray starting at 0 and approaching infinity with an argument of α. Fig. 5.1 provides an example for a deformation for this RHP including the intermediate steps, which are performed by our algorithm. Some further contours calculated by the algorithm for different parameters x are shown in Fig. 5.2. We are able to reduce the condition number of the contour in all of these cases from more than 10^{16} to 10^2 – 10^3.

5.2. modified Korteweg–de Vries

We recall from the introduction that the RHP corresponding to the modified Korteweg–de Vries (mKDV) equation

$$\begin{aligned} y_t - 6y^2 y_x + y_{xxx} &= 0 \quad \text{for } x \in \mathbb{R}, t \geq 0 \\ y(x, t=0) &= y_0(x) \end{aligned}$$

is according to (Deift and Zhou 1993) given by

RHP 3 (mKDV)**.** *Find a holomorphic function* $\Phi : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(m, \mathbb{C})$ *satisfying* $\Phi(\infty) = I$ *and* $\Phi^+(z) = \Phi^-(z)G(z; x, t)$ *where* $\Gamma = L_1 = \mathbb{R}$ *and*

$$G(z; x, t) = \begin{pmatrix} 1 - r(z)r(-z) & -r(-z)e^{-\theta(z;x,t)} \\ r(z)e^{\theta(z;x,t)} & 1 \end{pmatrix}$$

with the phase function

$$\theta(z; x, t) = i(2zx + 8z^3 t)$$

and $r(z)$ *being a function of the Schwartz space satisfying*

$$\sup_{z \in \mathbb{R}} |r(z)| < 1.$$

According to our conjecture (4.2) regarding the condition of a RHP, this original form of the RHP is not badly conditioned for any $(x, t) \in \mathbb{R} \times \mathbb{R}_0^+$. For $(z, x, t) \in \mathbb{R}^2 \times \mathbb{R}_0^+$ we get

$$\theta(z; x, t) = ki \text{ with } k \in \mathbb{R} \Rightarrow |e^{\pm\theta(z;x,t)}| \leq 1 \Rightarrow |r(\pm z)e^{\pm\theta(z;x,t)}| \leq 1 \Rightarrow |G_{i,j}(z; x, t)| \leq 1.$$

Consequently $\|G - I\|$ will not be large anywhere on the contour, indicating that the contour is well conditioned for all $(x, t) \in \mathbb{R} \times \mathbb{R}_0^+$. In fact we were not able to find any parameter combination that yields a bad conditioned system. Nevertheless we can apply our algorithm to optimize the contour. As we have stated in the introduction, the numerical solution of the original form of this RHP converges quite slowly. G is highly oscillatory along the real axis, causing the slow convergence and by deforming the contour those highly oscillatory areas can be avoided.

Deift and Zhou (1993) distinguish the following three regions requiring different deformations:

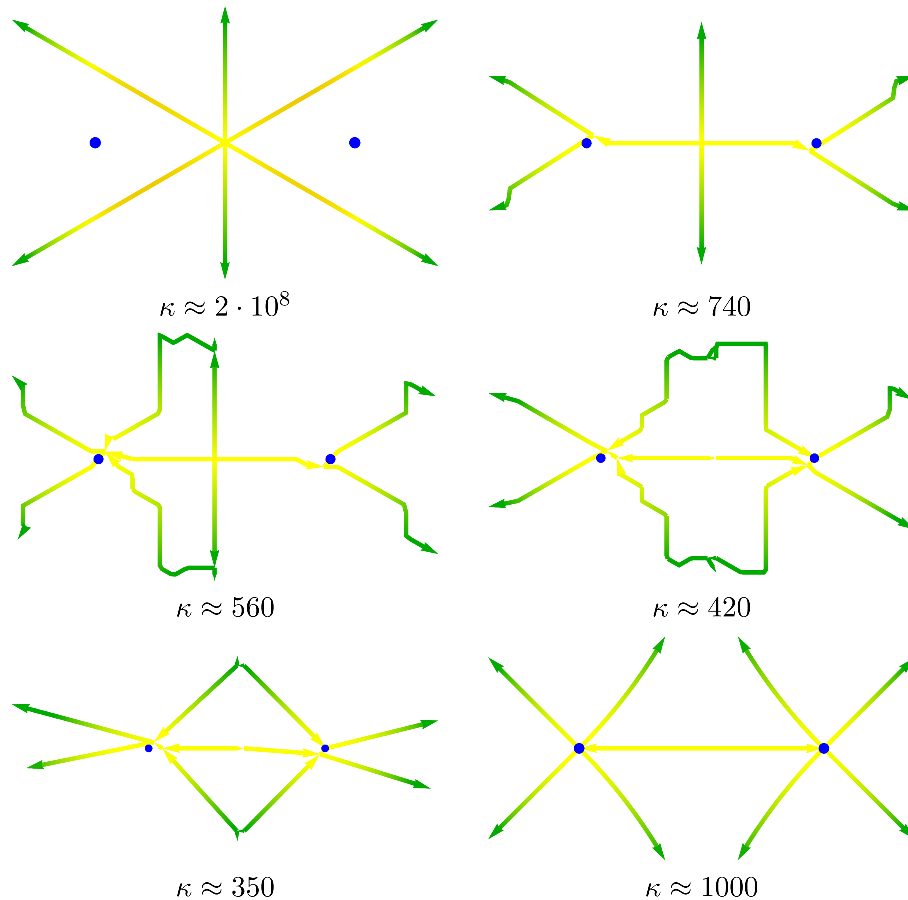


Figure 5.1.: Visualization according to style 1; Illustration of the deformation process for the Painlevé II RHP for $x = -10$, $s_1 = 1$ and $s_2 = 2$; *Top Left*: Original contour; *Top Right*: Simple deformation algorithm 1 applied to the original contour; *Middle Left*: Applying the lensing algorithm 10 to the previous contour, results in a UDL factorization of the left half of the central part. *Middle Right*: Applying the lensing algorithm 10 once more factorizes the remaining right half of the central part with a LDU decomposition of the jump matrix. *Bottom Left*: After converting all paths to straight lines connecting their endpoints, we get this simplified contour. *Bottom Right*: This is a manually deformed contour, taken from (Olver and Trogdon 2012, Fig. 4). The deformation uses the same factorizations of the jump matrices as our algorithm. We tried to achieve $\kappa < 10^3$ for this contour by adjusting the angles with which the rays emanate from the bifurcation points, but we did not succeed. Since we just measure the condition number of the linear system created by the solver, this could be caused by some numerical effect and does not necessarily mean the manually deformed RHP is worse conditioned than the automatically deformed one. *All*: Applying our algorithm significantly reduces the condition number of the contour by six orders of magnitude. Furthermore after the simplification step, the automatically deformed contour is very similar to the manually derived one. We should note that the algorithm repeatedly creates bifurcations at the stationary points. This is somewhat remarkable as no knowledge of these points is incorporated into the algorithm.

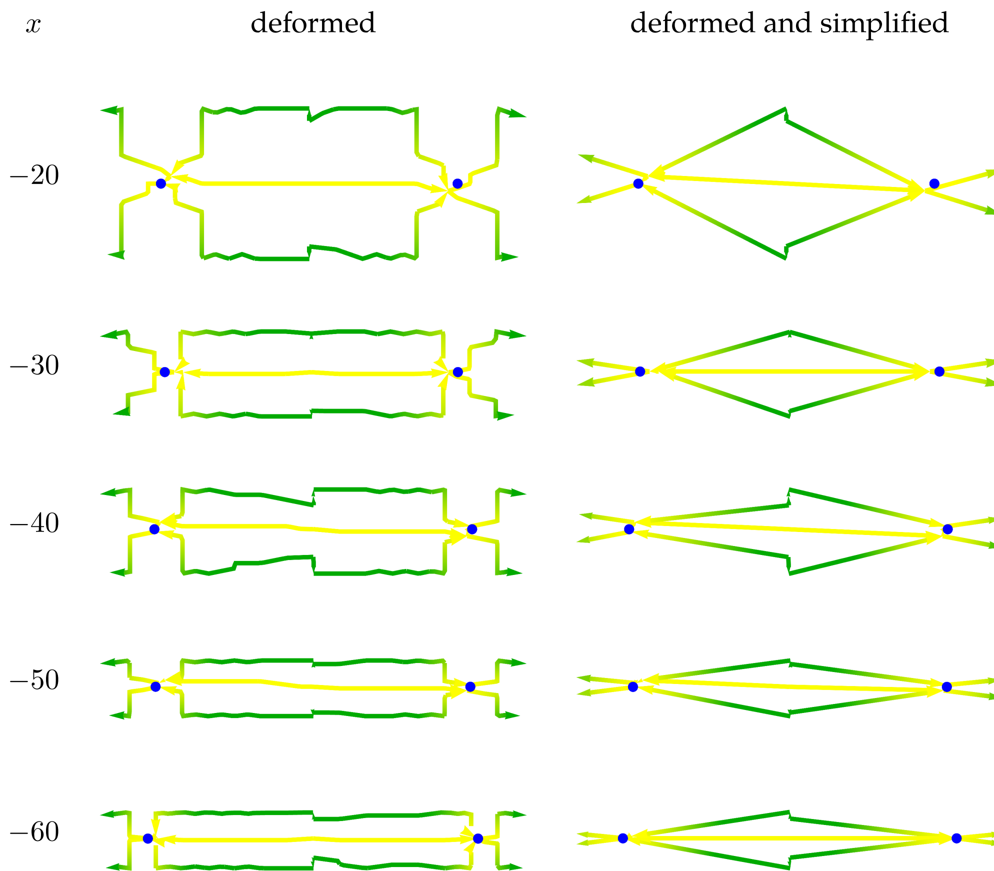


Figure 5.2.: Visualization according to style 1; Contour deformations of the Painlevé II RHP for some values of x and fixed parameters $s_1 = 1$, $s_2 = 2$; The blue points indicate the location of the stationary points of the phase function $z_0 = \pm i\sqrt{x}/2$. On the left side are the contours we got after applying the simple deformation algorithm 1 once and the lensing algorithm 10 twice. All contours were computed on a graph with a base grid of 17×17 vertices. We get the contours on the right side by converting each path to a straight line. In all cases the resulting contour consists of paths meeting at or close to the stationary points. The condition number κ is for all contours between 350 and 1000 whereas the original contour results in $\kappa > 10^{16}$ for $x \geq -20$. As the stationary points move apart, the aspect ratio changes and the contours appear to flatten in the plots. To verify the correctness of the deformation we used them to calculate the solution of the Painlevé II ODE and compared the results with the PainleveII function from RHPackage, which does the same using manually deformed contours.

- "Soliton Region": $x > ct^{1/3}$
- "Painlevé Region": $|x| < ct^{1/3}$
- "Dispersive Region": $x < -ct^{1/3}$

with some $c > 0$. For all of these region Trogdon et al. (2012) apply the method of nonlinear steepest descent to the original RHP and derive contours suitable to calculate a numerical solution. Thereby they suggest that c should be chosen on case-by-case basis, such that the best numerical results are obtained. In contrast to that we perform the deformation for all possible choices of t, x in the same way independent of c and the just mentioned regions. We simply apply our algorithm and see what contour it determines in the given situation.

This way we get in each of the regions a deformation which is very similar to the analytically derived ones. An example for each of these regions is shown in Fig. 5.3 (Solition Region), Fig. 5.4 (Painlevé Region) and Fig. 5.5 (Dispersive Region). The rather large negative value of $x = -100$ for the Dispersive region is due to the fact, that we have observed neither a significant benefit nor a significant penalty for switching from the deformation used in the Painlevé region to the deformation for the Dispersive region upto $x = -40$. We did not evaluate the situation for $x \in [41, 99]$. Somewhere in between is the point at which the deformation should be switched to achieve better results. The condition numbers of all shown contours are between 5 and 40. We only consider the case that the $r(z)$ is an entire function. The basic idea of our algorithm would also work if $r(z)$ is not an entire function, but we would need to extended our implementation to support jump matrices which are not holomorphic everywhere. That would require to support removing edges from a graph in areas where the corresponding jump matrix is not holomorphic.

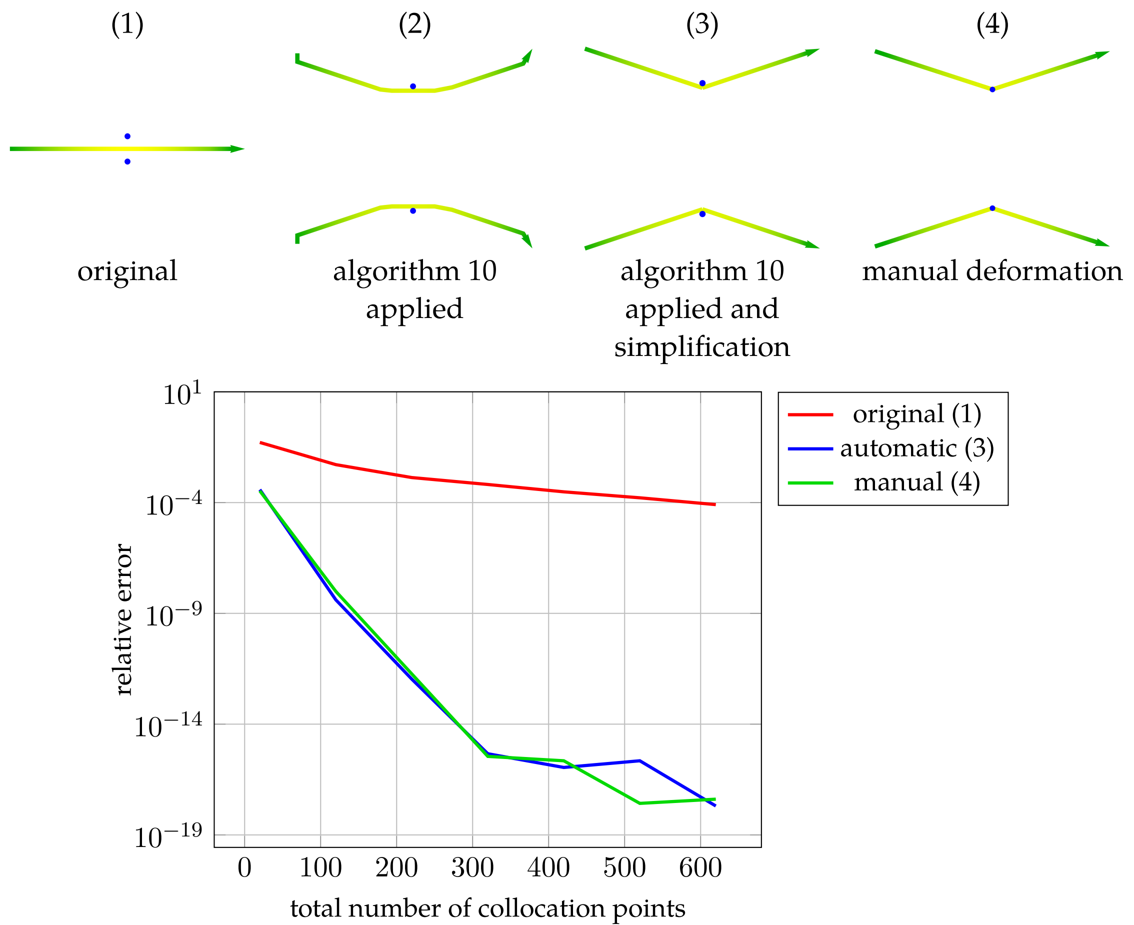


Figure 5.3.: Visualization according to style 1; *Top*: Deformations of the mKDV RHP for $t = 1, x = 5$ and $r(z) = \frac{1}{2}e^{-z^2}$. The blue points indicate the location of the stationary points $z_0 = \pm\sqrt{t/(12x)}$ of the phase function. The contours are obtained as follows. (1) Original contour of the RHP; (2) Lensing algorithm 10 applied to first contour; The algorithm decides to use a UDL type factorization (the contour for D is dropped, as D turns out to be the identity matrix). (3) Second contour simplified by converting rays to straight lines; (4) Manual deformation taken from (Trogdon et al. 2012, Fig. 8b). It is derived using the same UDL factorization. Even though our algorithm does not know the concept of stationary points, the automatically deformed contour is getting quite close to the stationary points. *Bottom*: Comparison of the convergence rates of the solution at $z = (|z_0| + 0.5)i$ for the deformed contours; Both contours derived by manual and automatic deformation yield numerical solutions which converge equally fast. As both contours are nearly the same that is also too be expected. The defomed contours provide a big improvement compared to the original contour.

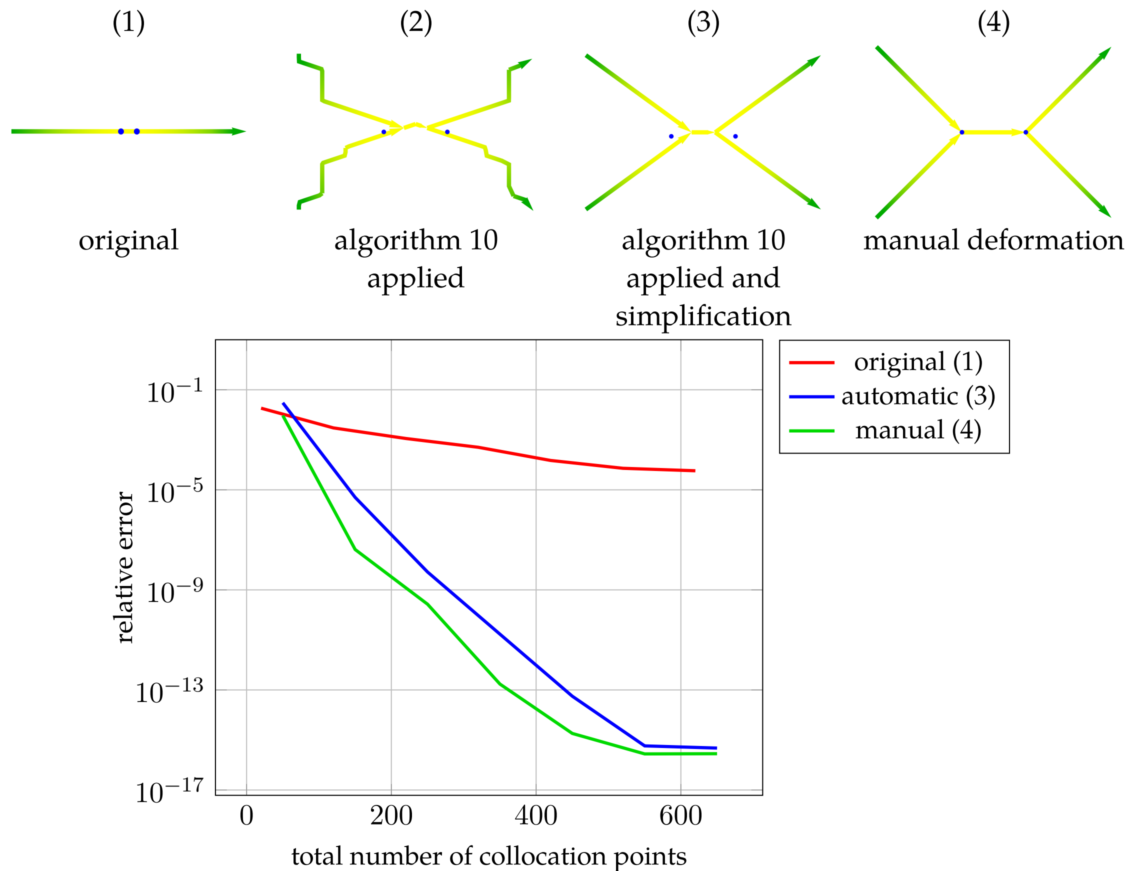


Figure 5.4.: Visualization according to style 1; *Top*: Deformations of the mKDV RHP for $t = 1, x = -2$ and $r(z) = \frac{1}{2}e^{-z^2}$. $x = -2$ is about the point at which the algorithm no longer yields a contour similar to the one in Fig. 5.3. The blue points indicate the location of the stationary points $z_0 = \pm\sqrt{t/(12x)}$ of the phase function. The contours are obtained as follows. (1) Original contour of the RHP; (2) Lensing algorithm 10 applied to first contour; Same factorization as in Fig. 5.3 gets chosen, but this time the two parts of the contour meet in the middle. (3) Second contour simplified by converting all parts to straight lines; (4) Manual deformation taken from (Trogdon et al. 2012, Fig. 8a). It employs the same factorization as our algorithm. The automatically deformed contour is once again routing the contour close to the stationary points. *Bottom*: Comparison of the convergence rates of the solution at $z = 0.5i$ for contours; The numerical solution of the manually deformed contour converges a bit faster, just as we might expect from the fact that it gets closer to the stationary points. The difference is quite small though, especially if we compare them to original contour.

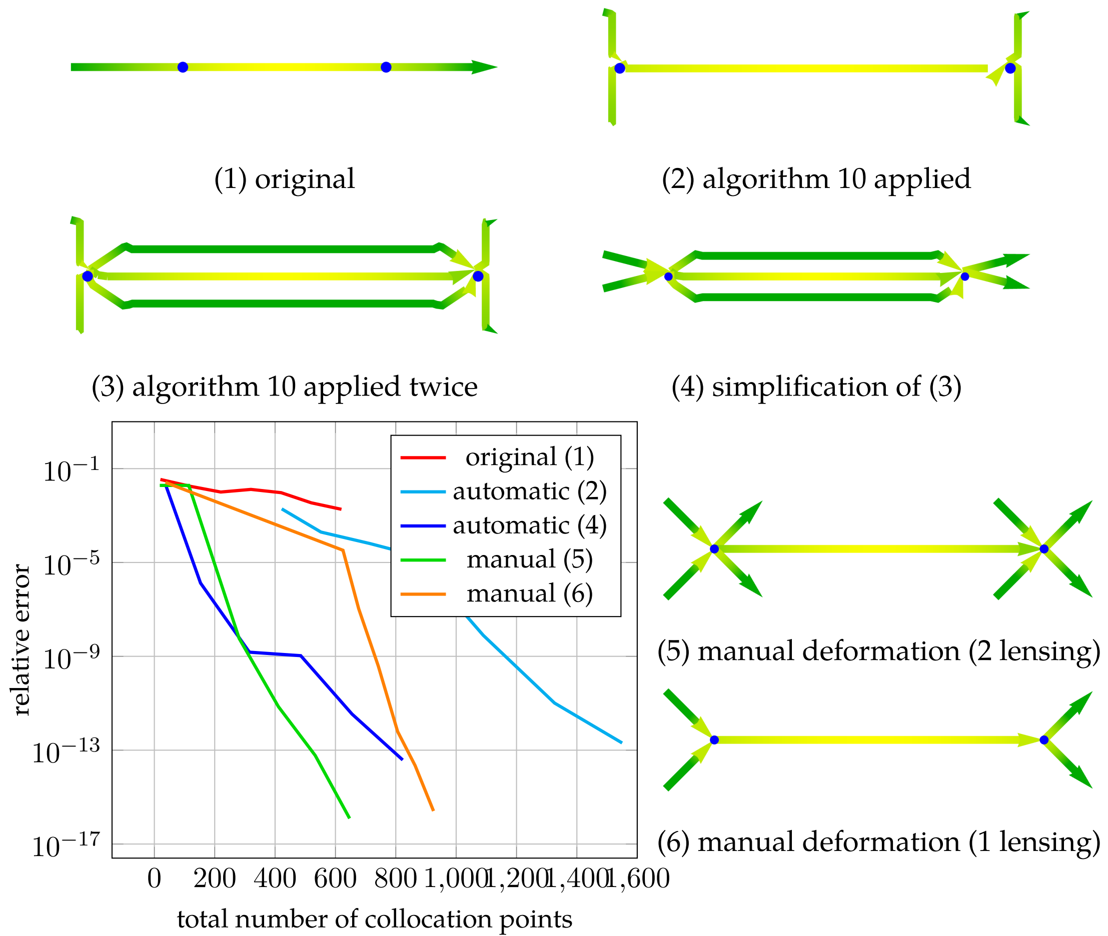


Figure 5.5.: Visualization according to style 1; *Top / Right*: Deformations of the mKDV RHP for $t = 1, x = -100$ and $r(z) = 1/2e^{-z^2}$. The blue points indicate the location of the stationary points $z_0 = \pm\sqrt{t/(12x)}$ of the phase function. (1): Original contour of the RHP; (2) Lensing algorithm 10 applied to (1); The algorithm chooses a UDL type factorization. (3) Lensing algorithm 10 applied to (2); A LDU factorization is chosen for the middle part of the contour. (4) Third contour simplified by converting rays to straight lines; (5) Manual deformation from (Trogdon et al. 2012, Fig. 5b) for this case. The deformation uses the same factorizations as our algorithm. (6) Manually deformed contour for the Painlevé region, see Fig. 5.4; The automatically deformed contour is once again creating bifurcations at the stationary points. *Bottom Left*: Comparison of the convergence rates of the solution at $z = -|z_0|+0.5i$ for the deformed contours; The numerical solution of the manually deformed contour converges faster but the difference it not particularly huge. Applying the second lensing deformation leads to a faster converging solution for both the manually and the automatic deformed contours.

5.3. Nonlinear Schrödinger

We consider the nonlinear Schrödinger (NLS) equation from (Deift and Zhou 1993) which is given by

$$iu_t + u_{xx} + 2\lambda|u|^2 u = 0 \tag{5.1}$$
$$u(x,0) = u_0(x) \tag{5.2}$$

and called "focusing" for $\lambda = 1$ respectively "defocusing" for $\lambda = -1$. The RHP corresponding to this equation is given by

RHP 4 (NLS)**.** *Find a holomorphic function* $\Phi : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(m, \mathbb{C})$ *satisfying* $\Phi(\infty) = I$ *and* $\Phi^+(z) = \Phi^-(z)G(z; x, t)$ *where* $\Gamma = L_1\mathbb{R}$ *and*

$$G(z; x, t) = \begin{pmatrix} 1 + \lambda r(z)\overline{r(\overline{z})} & \lambda\overline{r(\overline{z})}e^{-\theta(z)} \\ r(z)e^{-\theta(z)} & 1 \end{pmatrix} \quad \text{for } z \in \Gamma \in L_1 = \mathbb{R}.$$

with the phase function

$$\theta(z; x, t) = i(2xz + 4tz^2)$$

and $r(z)$ *being a function of the Schwartz space satisfying*

$$\sup_{z\in\mathbb{R}} |r(z)| < 1.$$

If we know the solution of this RHP, we can recover the solution of the NLS using

$$u(x,t) = 2i \lim_{|z|\to\infty} z\Phi_{1,2}(z).$$

Just as for the mKDV RHP, $|G_{i,j}(z; x, t)| \leq 1$ holds for $(z, x, t) \in \mathbb{R}^2 \times \mathbb{R}_0^+$ and as expected, we were also not able to find a set of parameters which yields a badly conditioned system. In addition the convergence of the numerical solution is again quite slow due to G being highly oscillatory along the real axis and a suitable deformation of the contour can avoid these highly oscillatory regions. The deformed contour we get by applying our deformation algorithm and a manually derived contour from (Trogdon and Olver 2012) are shown in Fig. 5.6. The effect of these deformations on the convergence rate is illustrated in Fig. 5.7.

This example is sort of a best case situation in which the algorithm produces a contour, which very closely resembles the one derived by hand. Fig. 5.8 illustrates some randomly chosen cases in which these contours look a bit more different. As expected the manually deformed contour always yields a numerical solution that converges faster than the one corresponding to the automatically deformed contour. Nevertheless the difference is not particularly large. So the automatically deformed contours are actually quite usable. Especially considering the fact that it takes only about 20 seconds to calculate them.

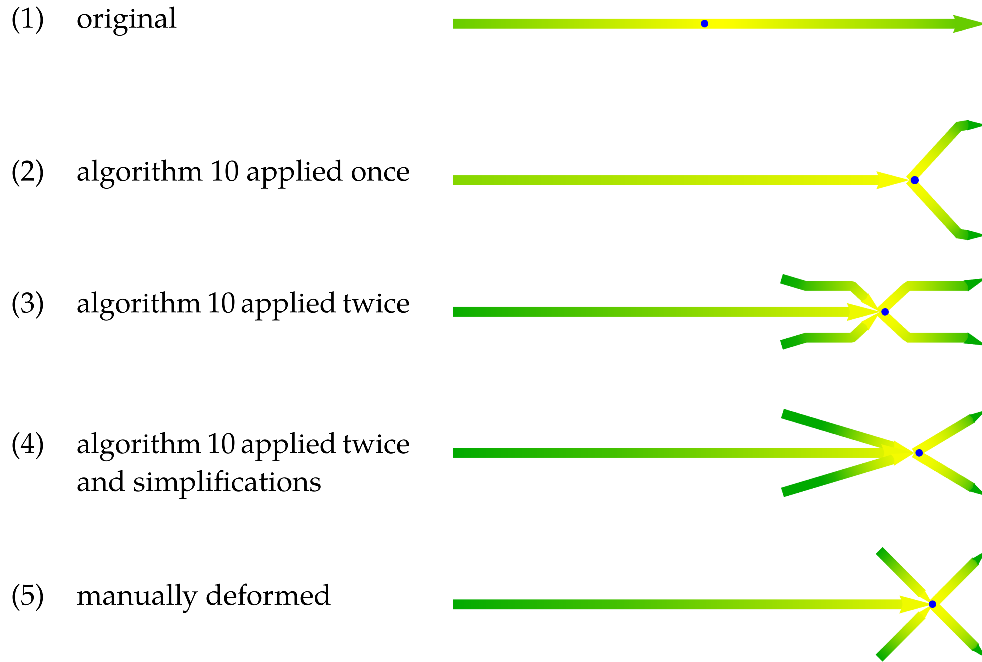


Figure 5.6.: Visualization according to style 1; Deformations of the NLS RHP for $\lambda = 1$,$t = 1$,$x = 5$ and $r(z) = 1/2\,\mathrm{sech}(z)$. The blue points indicate the location of the stationary point $z_0 = -x/(4t)$ of the phase function. The contours from top to bottom are obtained as follows. (1) Original contour of the RHP; We should note here that the plot has been rescaled, as it is about 4 times as wide as the others. (2) Lensing algorithm 10 applied to the original contour, resulting in a UDL type factorization with D being the identity matrix; (3) Lensing algorithm 10 applied to the second contour, resulting in a LDU type factorization; (4) Simplification of the third contour; (5) Manually deformed contour from (Trogdon and Olver 2012) using the same lensing factorizations. Our algorithm yields a contour which is very similar to the one obtained by manual deformations. The corresponding condition number satisfies $\kappa \approx 20$ for all contours.

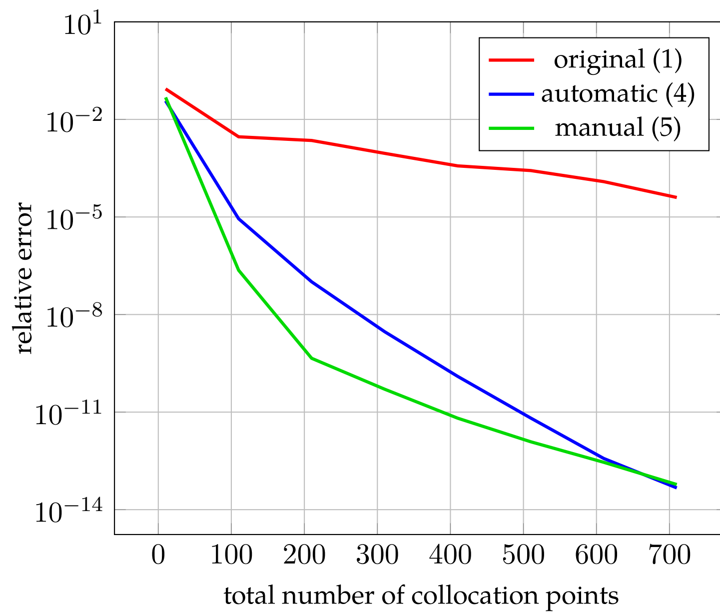


Figure 5.7.: Comparison of the convergence rates of the solution at $z = -1.5 + 0.5i$ for the deformed contours from Fig. 5.6; The contour calculated by our algorithm is almost as good as the one obtained by manual deformations and both are quite an improvement compared to the original contour.

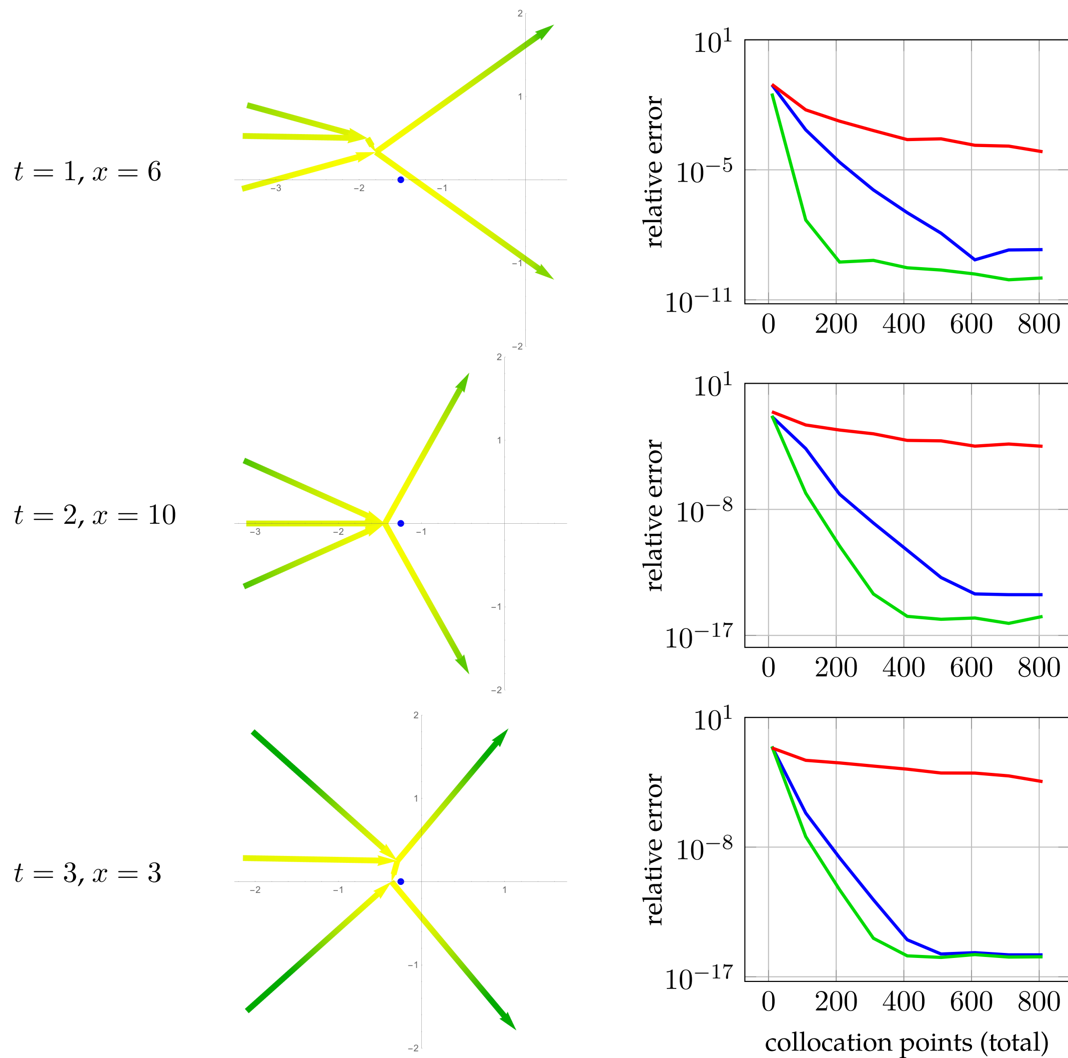


Figure 5.8.: Visualization according to style 1; Comparison of contours obtained by the lensing algorithm 10 twice to the original contour of the NLS RHP; The parameters which have not been stated above are $\lambda = 1$ and $f(z) = \frac{1}{2}\operatorname{sech}(z)$ for all three examples. In contrast to the plots of contours in Fig. 5.6, we provide a zoomed in view of the area around the stationary point z_0 of the phase function (blue). The plots on the right show the convergence of the numerical solution at $z_0 + i$ for the contours: original (red), automatically deformed (blue), manually deformed (green). Thereby z_0 is again the stationary point of the phase function. The solution converges faster if the contour resembles the manually deformed one more closely. In all cases the contour we obtained with our algorithm is a big improvement compared to the original contour.

5.4. Discrete z^α

5.4.1. Original Contour

(Bobenko and Pinkall 1996) defines a discrete conformal map as a function $f : \mathbb{Z}^2 \to \mathbb{C}$ satisfying

$$\frac{(f(n,m) - f(n+1,m))(f(n+1,m+1) - f(n,m+1))}{(f(n+1,m) - f(n+1,m+1))(f(n,m+1) - f(n,m))} = -1 \tag{5.3}$$

which means that the cross ratio of f is -1 for each quadrilateral in the grid. Building upon this definition (Bobenko and Pinkall 1998) defines a discrete version of the continuous z^α function as the function satisfying (5.3),

$$\begin{aligned}\alpha f(n,m) = {} & 2n\frac{(f(n+1,m) - f(n,m))(f(n,m) - f(n-1,m))}{f(n+1,m) - f(n-1,m)} \\ & + 2m(\frac{f(n,m+1) - f(n,m))f(n,m) - f(n,m-1)}{f(n,m+1) - f(n,m-1)}\end{aligned}$$

and the initial conditions[1]

$$f(0,0) = 0 \quad f(1,0) = 1 \quad f(0,1) = e^{\alpha\pi i/2} \tag{5.4}$$

To study the behaviour of f as $n, m \to \infty$, (Bobenko and Its 2014) derives a RHP corresponding to the difference equations for f if $m + n$ is even. This RHP is given by

RHP 5 (z^α). *Find a holomorphic function $X : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(2, \mathbb{C})$ satisfying*

$$X(z) = A\left(\mathrm{I} + \left(\frac{1}{z}\right)\right) z^{\frac{m+n}{2}\sigma_3} \quad \text{for } z \to \infty \quad \text{, with } A = \begin{pmatrix} \bullet & 0 \\ \bullet & \bullet \end{pmatrix}.$$

and $X^+(z) = X^-(z)G(z)$ for $z \in \Gamma$ where $\Gamma = C^+_{1/2}(1) \cup C^+_{1/2}(-1)$ and

$$G(z) = \begin{pmatrix} 1 & e^{\alpha\pi i/2} z^{-\alpha/2}(z-1)^{-m}(z+1)^{-n} \\ 0 & 1 \end{pmatrix}.$$

We recall that $C^+_r(c)$ denotes a circle around c with radius r and positive orientation. The contour Γ is shown in figure (5.9). Furthermore the dots in the matrix A indicate that we don't have any knowledge about these entries. The branch cut for the root in the jump function G is chosen to be $[0, -i\infty)$. If X solves the RHP we can calculate $f(n,m)$ using the relation

$$X(0) = \begin{pmatrix} 1 & f(n,m)(-1)^{m+1} \\ 0 & (-1)^m \end{pmatrix}.$$

[1] (5.3) and the initial conditions $f(n,0) = n^\alpha$ and $f(0,m) = (im)^\alpha$ also yield a unique solution, but it is far away from the continuous z^α function. See (Bobenko and Its 2014) for details.

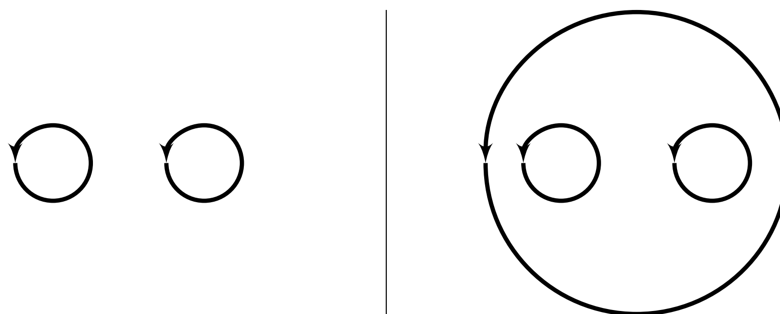


Figure 5.9.: The contours Γ (left) of the original RHP (5) and $\tilde{\Gamma}$ (right) of the normalized RHP (6)

We cannot numerically solve RHP 5 in its current form, as the normalization at infinity is not the same as the one assumed by the RHP solver we are using (see also § 4.1). To normalize the RHP we pull the behaviour at infinity down to a finite region by setting

$$\tilde{X}(z) = \begin{cases} A^{-1}X(z)z^{-\frac{m+n}{2}\sigma_3} & \text{for } |z| > 2 \\ A^{-1}X(z) & \text{for } |z| < 2 \end{cases}. \tag{5.5}$$

and get a deformed RHP for $\tilde{X}$.

RHP 6 (z^α normalized)**.** *Find a holomorphic function* $\tilde{X} : \mathbb{C} \setminus \tilde{\Gamma} \to \mathrm{GL}(2, \mathbb{C})$ *satisfying*

$$\tilde{X}(z) = \mathrm{I} + \left(\frac{1}{z}\right) \quad \textit{for } z \to \infty$$

and $\tilde{X}^+(z) = \tilde{X}^-(z)G(z)$ *for* $z \in \tilde{\Gamma}$ *where* $\tilde{\Gamma} = \Gamma \cup C_2^+(0)$ *and*

$$G(z) = \begin{cases} \begin{pmatrix} 1 & e^{\alpha\pi i/2}z^{-\alpha/2}(z-1)^{-m}(z+1)^{-n} \\ 0 & 1 \end{pmatrix} & \textit{for } z \in \Gamma \\ z^{\frac{m+n}{2}\sigma_3} & \textit{for } z \in C_2^+(0) \end{cases}$$

This deformed RHP possesses a normalization at infinity, which is compatible with the solver and we can still recover $f(n, m)$ from $\tilde{X}$. As

$$\tilde{X}(0) = A^{-1}X(0) = \begin{pmatrix} \bullet & 0 \\ \bullet & \bullet \end{pmatrix} \begin{pmatrix} 1 & f(n,m)(-1)^{m+1} \\ 0 & (-1)^m \end{pmatrix}.$$

it holds that

$$f(n, m) = (-1)^{m+1} \frac{\tilde{X}_{1,2}(0)}{\tilde{X}_{1,1}(0)}$$

Though our algorithm is successful in deforming the contour of the normalized RHP 6, see Fig. 5.10, the condition number of the deformed RHP is still far too large to calculate a solution with at least 1 bit of accuracy. The reason for this problem is likely the following.

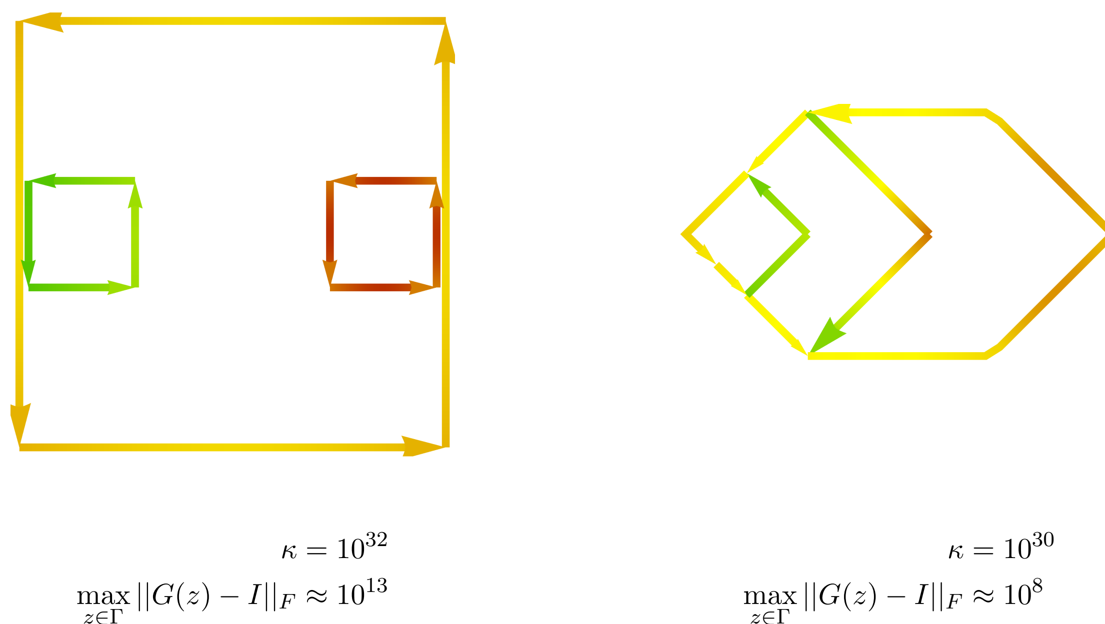


Figure 5.10.: Visualization according to style 1; The original contour of the normalized RHP 6 for $m = 30, n = 2, \alpha = 2/3$ before (left) and after (right) our algorithm has been applied. The circles of the contour have been replaced by squares, as linear parts in general work better than curved parts. Our algorithm calculates a contour, which reduces the maximum weight along the contour by 10^5, but this improvement has just a small effect on the condition number. Furthermore the condition number is far larger than what we would expect due to the maximum weight. The reason for this effect is discussed in § 5.4.2.

5.4.2. Singular Linear System

If we consider just the $(1,1)$ component of $\tilde{X}$ for $m = n = 1$ we get the following RHP.

RHP 7 ((1,1) component of RHP 6)**.** *Find a holomorphic function* $\tilde{X}_{1,1} : \mathbb{C} \setminus C_2^+(0) \to \mathbb{C}$ *with* $\tilde{X}_{1,1}(\infty) = 1$ *and*

$$\tilde{X}_{1,1}^+(z) = \tilde{X}_{1,1}^-(z)z \text{ for } z \in C_2^+(0)$$

As can be easily verified, RHP 7 does not have a unique solution. If $\tilde{X}_{1,1}$ solves RHP 7, then

$$Y_{1,1}(z) = \begin{cases} \tilde{X}_{1,1}(z) + a/z & \text{for } |z| > 2 \\ \tilde{X}_{1,1}(z) + a & \text{for } |z| < 2 \end{cases} \tag{5.6}$$

also solves RHP 7 for all $z \in \mathbb{C}$. For the first row[2] of RHP 6 to be uniquely solvable, this degree of freedom for $\tilde{X}_{1,1,}$ has to be removed by the equation for $\tilde{X}_{1,2}$. But this fact is not captured by the discretization which the solver uses. As RHP 7 does not depend on $\tilde{X}_{1,2}$ the linear system created by solver, which we discussed in § 4.1, has the following block structure

$$\begin{pmatrix} B & 0 \\ C & D \end{pmatrix} \begin{pmatrix} v_{1,1} \\ v_{1,2} \end{pmatrix} = \begin{pmatrix} b_{1,1} \\ b_{1,2} \end{pmatrix}. \tag{5.7}$$

Thereby B and D are square matrices and $v_{1,1}, v_{1,2}$ contain the coefficients corresponding to $\tilde{X}_{1,1}$ respectively $\tilde{X}_{1,2}$. B is essentially a discretization of RHP 7. Furthermore B is a singular matrix, due to the fact that RHP 7 does not have a unique solution. Consequently the linear system (5.7) is also singular and causes the large condition number.

The deformations which our algorithm can currently perform, preserve the fact that the RHP for $\tilde{X}_{1,1}$ does not depend on $\tilde{X}_{1,2}$. Therefore we will always end up with a singular linear system of equations and a large condition number. But if the discretization of the RHP can be changed in a way such that it reflects the structure of this RHP more closely, it might be possible to calculate a solution using the contour shown in Fig. 5.10. Nevertheless this is outside of the scope of this thesis.

5.4.3. Deformed Contour

Even though calculating a solution of the original version of the z^α RHP 5 was not successful, we want to discuss a deformed version of it. This deformed problem does yield a correct solution and can be considered as a proof of concept that the problem we encountered in the last section can be circumvented. Though we currently do not know how to avoid it algorithmically. We start with the RHP in (Bobenko and Its 2014) which is called the S - RHP and can be derived from RHP 5 through several deformation steps. Details about the deformations are available in (Bobenko and Its 2014). Performing this deformation automatically would at least require to extend our approach to include g-function deformations (see § 6) and develop a better approximation of condition number

[2] Rows of a matrix valued RHP can be solved individually.

for this case (Fig. 5.10 already showed that our current heuristic does not work very well in this case).

Before we can state this RHP we need to define the following functions

$$H(z) = \left(\frac{1+\sqrt{z}}{1-\sqrt{z}}\right)^m \left(\frac{i+\sqrt{z}}{i-\sqrt{z}}\right)^n$$

$$h_0(z) = m\log\frac{1+\sqrt{z}}{1-\sqrt{z}} + n\log\frac{1-i\sqrt{z}}{1+i\sqrt{z}} \quad \text{for} \quad |z| < \delta < 1$$
$$h_\infty(z) = m\log\frac{1+\frac{1}{\sqrt{z}}}{1-\frac{1}{\sqrt{z}}} + n\log\frac{1-\frac{i}{\sqrt{z}}}{1+\frac{i}{\sqrt{z}}} \quad \text{for} \quad |z| > \frac{1}{\delta} > 1$$

$$\begin{aligned} w_0 &= 2i\sin\left(\alpha\frac{\pi}{2}\right) \\ w_1(z) &= w_0 z^{-\alpha/2} \\ w_2(z) &= w_0 e^{\pi i\alpha} z^{-\alpha/2} \\ w(z) &= w_0 z_+^{-\alpha/2}. \end{aligned}$$

$$\begin{aligned} C_0 &= \begin{pmatrix} 1 & e^{i\pi\gamma/2} \\ 0 & 1 \end{pmatrix} \\ C_r &= C_0 \begin{pmatrix} 1 & 0 \\ -w_0^{-1} & 1 \end{pmatrix} \\ C_l &= C_0 \begin{pmatrix} 1 & 0 \\ w_0^{-1}e^{-i\pi\gamma} & 1 \end{pmatrix} \\ C(z) &= \begin{cases} C_r & \text{for } -\frac{\pi}{2} < \arg z < \frac{\pi}{4} \\ C_0 & \text{for } \frac{\pi}{4} < \arg z < \frac{3}{4}\pi \\ C_l & \text{else} \end{cases} \end{aligned}$$

Thereby λ_+ indicates that if λ is on the branch cut of the square root, we use the limit from the side with positive real part to evaluate it. The branch cuts of the involved function are chosen as follows

- $\log z: \quad -\pi < \arg z < \pi$
- $\sqrt{z}: \quad -\frac{\pi}{2} < \arg z < \frac{3\pi}{2}$
- $z^{-\alpha/2}: \quad -\frac{\pi}{2} < \arg z < \frac{3\pi}{2}.$

The S - RHP is then given by

RHP 8 (z^α deformed)**.** *Find a holomorphic function* $S : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(2, \mathbb{C})$ *satisfying*

$$\Phi(z) = A\left(I + \left(\frac{1}{z}\right)\right) z^{-\frac{\alpha}{4}\sigma_3} C(z) z^{\frac{\alpha}{4}\sigma_3} e^{-\frac{1}{2}h_\infty}(z)\sigma_3 \textit{ for } z \to \infty \textit{ with } A = \begin{pmatrix} \bullet & 0 \\ \bullet & \bullet \end{pmatrix}$$

and $S^+(z) = S^-(z)G(z)$ *for* $z \in \Gamma^0$*, where* $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$ *and*

$$G(z) = \begin{cases} \begin{pmatrix} 0 & w(z) \\ -w^{-1}(z) & 0 \end{pmatrix} & \textit{for } z \in \Gamma_1 = R_{-i}(0) \\ \begin{pmatrix} 1 & 0 \\ H^{-1}(z)w_1^{-1}(z) & 1 \end{pmatrix} & \textit{for } z \in \Gamma_2 = R_{1+i}(0) \\ \begin{pmatrix} 1 & 0 \\ H^{-1}(z)w_2^{-1}(z) & 1 \end{pmatrix} & \textit{for } z \in \Gamma_3 = R_{-1+i}(0) \end{cases}$$

See Fig. 5.11 for a plot of the contour. Here $m + n$ has to be even, just as for the original RHP 5. If S solves this RHP, the solution has the following behaviour

$$S(z) = \begin{pmatrix} 1 & f(n,m)(-1)^{m+1} \\ 0 & 1 \end{pmatrix} e^{-\frac{i\pi}{2}n\sigma_3} \left(I + (z)\right) z^{-\frac{\alpha}{4}\sigma_3} C(z) z^{\frac{\alpha}{4}\sigma_3} e^{-\frac{1}{2}h_0}(z)\sigma_3 \text{ for } z \to 0 \tag{5.8}$$

Analogous to the situation in the original RHP 5, we have to normalize the condition at infinity before we can compute a numerical solution. Furthermore we also normalize the behaviour at $z = 0$, to facilitate recovering $f(n, m)$ from the solution of the RHP. In both cases we use the same approach as for the normalization of the original RHP 5. We pull the behaviour at $z = 0$ and $z = \infty$ away from these points by defining a function $\tilde{S}$ with additional jumps.

$$\tilde{S} = \begin{cases} A^{-1} S e^{\frac{1}{2}h_\infty(z)\sigma_3} z^{-\frac{\gamma}{4}\sigma_3} C(z)^{-1} z^{\frac{\gamma}{4}\sigma_3} & \text{for } |z| > r_0 \\ A^{-1} S e^{\frac{1}{2}h_0(z)\sigma_3} z^{-\frac{\gamma}{4}\sigma_3} C(z)^{-1} z^{\frac{\gamma}{4}\sigma_3} & \text{for } |z| < r_\infty \\ A^{-1} S & \text{else} \end{cases} \tag{5.9}$$

r_0 and r_∞ may be chosen arbitrarily as long as $r_0 < 1$ and $r_\infty > 1$ holds. With these extra jumps we get the following normalized RHP.

RHP 9 (z^α deformed & normalized)**.** *Find a holomorphic function* $\tilde{S} : \mathbb{C} \setminus \tilde{\Gamma} \to \mathrm{GL}(2, \mathbb{C})$

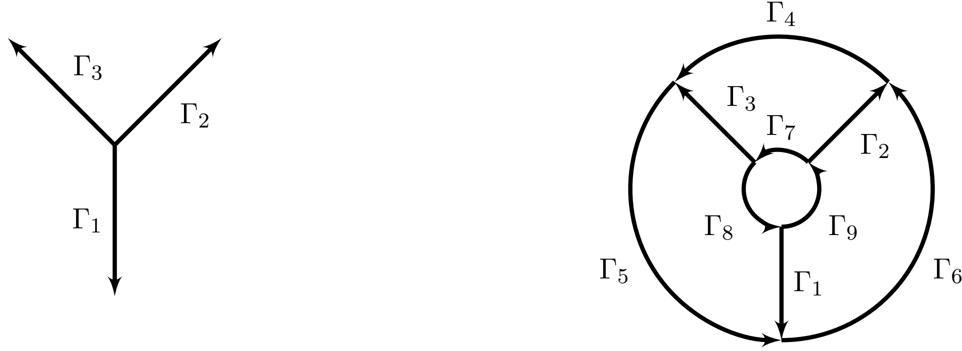


Figure 5.11.: contours Γ of RHP 8 (left) and $\tilde{\Gamma}$ (right) of RHP 9; The origin $z = 0$ is located where the rays Γ_1,Γ_2 and Γ_3 meet.

satisfying $\tilde{S}(\infty) = I$ *and* $\tilde{S}^+(z) = \tilde{S}^-(z)G(z)$ *for* $z \in \tilde{\Gamma}^0$, *where* $\tilde{\Gamma} = \Gamma_1 \cup \Gamma_2 \cup \ldots \cup \Gamma_9$

$$G(z) = \begin{cases} \begin{pmatrix} 0 & w(z) \\ -w^{-1}(z) & 0 \end{pmatrix} & \text{for } z \in \Gamma_1 \\ \begin{pmatrix} 1 & 0 \\ H^{-1}(z)w_1^{-1}(z) & 1 \end{pmatrix} & \text{for } z \in \Gamma_2 \\ \begin{pmatrix} 1 & 0 \\ H^{-1}(z)w_2^{-1}(z) & 1 \end{pmatrix} & \text{for } z \in \Gamma_3 \\ z^{-\frac{\alpha}{4}\sigma_3} C_0 z^{\frac{\alpha}{4}\sigma_3} e^{-\frac{1}{2}h_\infty(z)\sigma_3} & \text{for } z \in \Gamma_4 \\ z^{-\frac{\alpha}{4}\sigma_3} C_l z^{\frac{\alpha}{4}\sigma_3} e^{-\frac{1}{2}h_\infty(z)\sigma_3} & \text{for } z \in \Gamma_5 \\ z^{-\frac{\alpha}{4}\sigma_3} C_r z^{\frac{\alpha}{4}\sigma_3} e^{-\frac{1}{2}h_\infty(z)\sigma_3} & \text{for } z \in \Gamma_6 \\ e^{\frac{1}{2}h_0(z)\sigma_3} z^{-\frac{\alpha}{4}\sigma_3} C_0^{-1} z^{\frac{\alpha}{4}\sigma_3} & \text{for } z \in \Gamma_7 \\ e^{\frac{1}{2}h_0(z)\sigma_3} z^{-\frac{\alpha}{4}\sigma_3} C_l^{-1} z^{\frac{\alpha}{4}\sigma_3} & \text{for } z \in \Gamma_8 \\ e^{\frac{1}{2}h_0(z)\sigma_3} z^{-\frac{\alpha}{4}\sigma_3} C_r^{-1} z^{\frac{\alpha}{4}\sigma_3} & \text{for } z \in \Gamma_9 \end{cases}$$

The contour $\tilde{\Gamma}$ is shown in figure Fig. 5.11. Due to the normalization (5.9) the behaviour (5.8) for $z \to 0$ simplifies to

$$\tilde{S}(0) = A^{-1} \begin{pmatrix} 1 & f(n,m)(-1)^{m+1} \\ 0 & (-1)^m \end{pmatrix} e^{-\frac{i\pi}{2}n\sigma_3} = \begin{pmatrix} e^{-i\frac{\pi}{2}n} & f(n,m)(-1)^{m+1}e^{i\frac{\pi}{2}n} \\ 0 & (-1)^m e^{i\frac{\pi}{2}n} \end{pmatrix}$$

As A^{-1} is a lower triangular matrix and $m + n$ is even, we get the relation

$$\begin{aligned} -\frac{\tilde{S}_{1,2}(0)}{\tilde{S}_{1,1}(0)} &= -\frac{f(n,m)(-1)^{m+1}e^{i\frac{\pi}{2}n}}{e^{-i\frac{\pi}{2}n}} = -(f(n,m)(-1)^{m+1}e^{i\pi n}) = f(n,m)(-1)^{m+n+2} \\ &= f(n,m). \end{aligned}$$

After solving the RHP, we can use this relation to calculate $f(n, m)$. Before we solve the RHP we could apply our contour deformation algorithm, but choosing $r_0 = 1/8, r_\infty = 8$ is already enough to leave very little room for further improvements, see also Fig. 5.12 (left). In addition our goal is to automatically handle the original version of the RHP and not the heavily deformed version. So we will just use the standard contour from RHP 9, with all curved lines replaced by straight lines as usual. The results we get for $f(m, n)$ can be seen in Fig. 5.12 (right).

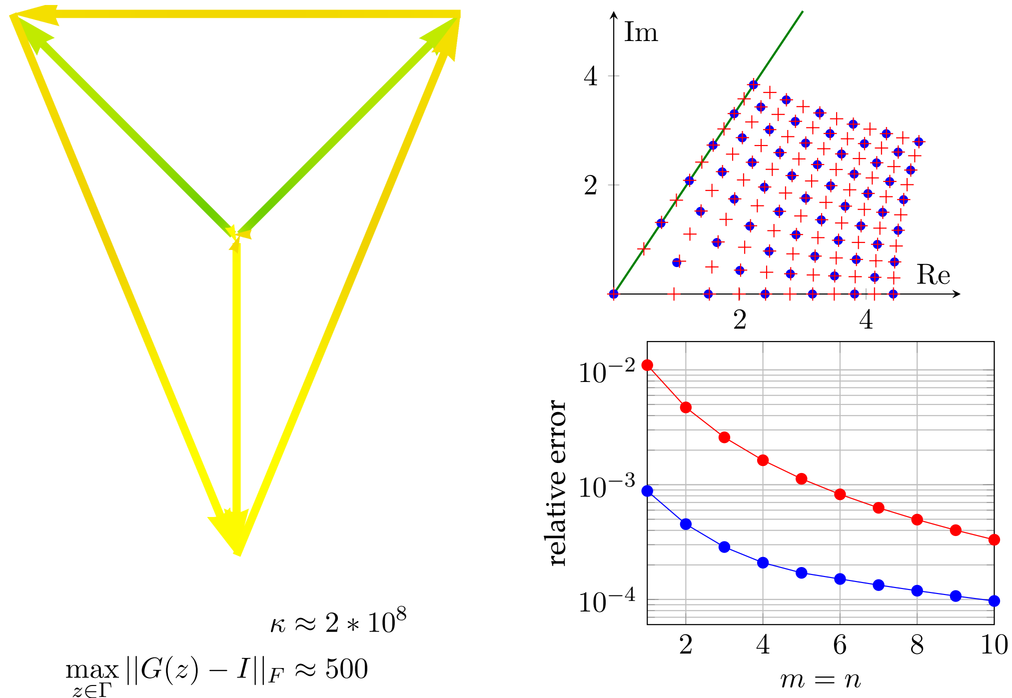


Figure 5.12.: *Left*: Contour of RHP 9 for $m = n = 10, \alpha = 2/3, r_0 = 1/8$ and $r_\infty = 8$ with arcs converted to straight lines; It is worth noting that similar to the example in Fig. 5.10 the condition number is larger than what we would expect for the maximum weight. This could be a remnant of the problem discussed in § 5.4.2. *Right*: Numerical solution of $f(m, n)$ (blue) for $m, n \in \{0, 1, \ldots, 10\}$ with $m + n$ being even and $\alpha = 2/3$ compared to the asymptotic formula for $f(m, n)$ (red) derived in (Bobenko and Its 2014); The green line indicates the ray $ri^{2/3}$, $r > 0$. The numerical solution has been calculated using 400 collocation nodes on each part of the contour. To estimate the error of the numerical solution along the line $m = n$, we compare it against another numerical solution calculated using 600 collocation nodes on each part of the contour. Furthermore for $m = n = 1$ we can use (5.3) and (5.4) to determine the exact value of $f(1, 1)$ and compare it against both solutions.

6. Future Work

As we have presented in the previous chapter, our algorithm is already capable of deriving various deformations of RHPs, which have proven to be useful for preconditioning or stabilizing the numerical method for solving them. Nevertheless there is still room for improvement and in this chapter we will describe a few improvements that could be made.

6.1. g–Function Deformations

The probably most useful improvement would be to include g–function deformations in the set of supported deformation types. This particular type of deformation is used in nonlinear steepest descent deformations, when simple and lensing deformations are not enough to get the contour into a state where $G \to I$ fast along each part of the contour. We will only give a brief explanation of this deformation here. Fokas et al. (2006) provide more details and apply this deformation among others to the Painlevé II RHP. In a nutshell g-function deformations work as follows. Suppose we have a standard RHP, e.g.

RHP 10 (g–function example, original)**.** *Find a holomorphic function* $\Phi : \mathbb{C} \setminus \Gamma \to \mathrm{GL}(m, \mathbb{C})$ *with* $\Phi(\infty) = I$ *and* $\Phi^+(z) = \Phi^-(z)G(z)$ *for* $z \in \Gamma$ *.*

By taking a holomorphic function $A : \mathbb{C} \setminus \Gamma_0 \to \mathrm{GL}(m, \mathbb{C})$ with $A(\infty) = I$, we can then define a new function $\widetilde{\Phi} = \Phi A$ which solves the following RHP.

RHP 11 (g–function example, deformed)**.** *Find a holomorphic function* $\widetilde{\Phi} : \mathbb{C}\setminus\widetilde{\Gamma} \to \mathrm{GL}(m, \mathbb{C})$ *with* $\widetilde{\Phi}(\infty) = I$ *and* $\widetilde{\Phi}^+(z) = \widetilde{\Phi}^-(z)\widetilde{G}(z)$ *for* $z \in \widetilde{\Gamma}$*, where* $\widetilde{\Gamma} = \Gamma \cup \Gamma_0$

$$\widetilde{G}(z) = \begin{cases} (A^-)^{-1}(z)G(z)A^+(z) & \text{for } z \in \Gamma \\ (A^-)^{-1}(z)A^+(z) & \text{for } z \in \Gamma_0 \setminus \Gamma. \end{cases}$$

Thereby common forms for A are

$$A(z) = \begin{pmatrix} 1 & e^{ig(z)} \\ 0 & 1 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} e^{ig(z)} & 0 \\ 0 & e^{-ig(z)}. \end{pmatrix}$$

for some function $g : \Gamma_0 \to \mathbb{C}$, thus the name g-function deformation. If A is chosen appropriately, the new jump matrix $\widetilde{G}$ decays faster to the identity matrix along Γ than G. We can even remove parts of the contour if $\Gamma_0 \subset \Gamma$ and

$$\begin{aligned} I &= (A^-)^{-1}(z)G(z)A^+(z) \\ (A^+)^{-1}(z) &= (A^-)^{-1}(z)G(z) \end{aligned}$$

holds for $z \in \Gamma_0$. Which means if we can solve the original RHP on a restricted contour $\Gamma_0 \subset \Gamma$, we can perform a g–function deformation on the original RHP to remove the jump at Γ_0.

Incorporating g-function deformations into our algorithm will not be an easy task, but we believe it is possible to some extent at least. Deciding to which part of the contour a g-function deformation should be applied is no big problem. It is most useful to apply it to the part which is limiting the condition number or the convergence speed, which is the part with the largest weight. The main difficulty though is to find a suitable function g respectively A. A case for which this should be doable, is $G(z)$ being a diagonal matrix for $z \in \Gamma_j \subset \Gamma$. In this case an explicit solution of the RHP on the restricted contour Γ_j is given by the integral representation from (1.2). As discussed this solution allows for a g-function deformation that removes the jump at Γ_j from the original contour Γ.

For more general situations, we might be able to approximate a g-function. If we consider only the aspect of reducing the condition number of the RHP, any function g respectively matrix A which yields $\|\widetilde{G} - I\| \leqslant \|G - I\|$ along the contour is an improvement. Furthermore (Olver 2011a) provides a formula to calculate candidates for a g-function and (Trogdon et al. 2012) even already computed a g-function numerically and successfully used it to deform the mKDV RHP. So doing this in an automatic way does seem to a reachable goal.

6.2. Local Refinements

So far our algorithm successfully moved the contours through or close to stationary points of the phase function in all examples. But how well the stationary points are resolved varies a bit. Sometimes the deformed contours go almost exactly through them and sometimes they pass by them at a close distance. Identifying possible locations of stationary points is not very difficult even if we use only information available through the paths and graphs already used by our algorithm. They are always located at or close to points where two or more paths meet and the vertex at which they meet has a weight of approximately 1 in the respective graphs.

We can exploit this situation and apply a local refinement of the contour. Let $p_1, p_2, \ldots$ bet a set of paths, that meet at the vertex v. We can move this meeting point around by choosing a vertex u in the neighbourhood of v and replacing $p_1, p_2, \ldots$ with shortest paths $q_1, q_2, \ldots$ connecting the endpoints of $p_1, p_2, ...$ with u. Selecting the vertex $u_\star$ which yields the lowest total weight of the corresponding set of paths improves the contour further and should also resolve the stationary point close to it better. Even if there is no stationary point close $u_\star$, this should result in a better contour nonetheless. The points we get with our algorithm could in some cases even be better than the true stationary points. For example, in the case of the mKDV RHP (see), the true stationary points are calculated using only the oscillator, but our algorithm also takes the reflection coefficient into account.

6.3. Recover Original Solution

For now we can automatically deform a contour of a given RHP and calculate its solution. What we are not yet able to do automatically, is recovering the solution of the original RHP. Being able to do so automatically would be a nice feature to have, even though it is not often necessary to actually recover the original solution. Quite often only the residue of the solution at infinity is of interest and the residue is not changed by the deformations we are considering so far. Recovering the original solution is not a particular difficult task. As we recall from Fig. 4.5 and Fig. 4.7, the solution of the deformed and the original RHP are related by a simple matrix multiplication. Therefore we only have to keep track of which matrix has to be used in which area during the deformation process.

7. Summary

We were able to show that our basic idea of casting the problem of finding optimal contours into the problem of finding shortest paths in graph enables an algorithmic approach for optimizing contours. In the case of Cauchy's integral formula we were able to calculate the optimal solution of the resulting discrete optimization problem. This optimal solution induces contours, which yield similar condition numbers for the integral as optimal circles derived analytically. Consequently, we also developed a "black box" algorithm for calculating arbitrary derivatives of holomorphic functions with great accuracy.

Applying the same approach to a RHP yields a more complicated discrete optimization problem, so that we are currently not able to compute its optimal solution. But the greedy algorithm we presented is for all the discussed example RHPs able to compute deformations which are very similar to analytically derived deformations. Thereby just as the method of nonlinear steepest descent, the automatic deformation process also creates bifurcations of the contour at or close to stationary points of the phase function in the RHP. This is remarkable as the algorithm does not have any knowledge about bifurcation or stationary points. Furthermore the deformations produced by our algorithm in the investigated examples are equally or at least very similarly effective for preconditioning the RHPs, compared to the analytically derived ones. The deformations reduce the condition number by several orders of magnitude or increase speed of the convergence of the solution a lot. Due to the good results so far, we assume that if there is a deformation useful for preconditioning which can be derived with the deformation types considered by our algorithm, the algorithm will find an approximation of this deformation.

In addition, considering the close resemblance of automatically and analytically derived deformations in the discussed cases, it could even be possible to use the automatically derived ones as an initial draft for analytic deformations. As a final visualization of our results we provide the numerical results we got for the two cases used in the introduction to illustrate the frequently occurring numerical instability, see Fig. 7.1 (Painlevé II) and Fig. 7.2 (mKDV).

We should also note that no information besides the RHP itself is necessary for the automatic deformation with our algorithm. The main limitation at the moment, is the fact, that our algorithmic approach is not yet able to perform g–function deformations. Whenever this type of deformation is necessary, we are not able to achieve useful results with our automatic deformation algorithm. Lifting this limitation and achieving a truly universal preconditioning algorithm is postponed to future work.

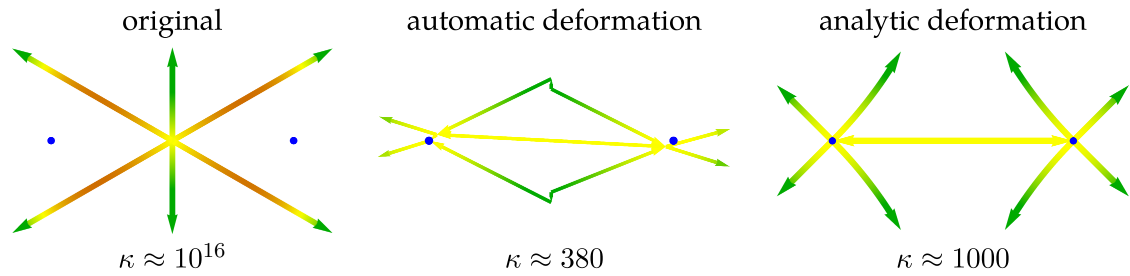


Figure 7.1.: Visualization according to style 1; deformations of the Painlevé II RHP for $x = -20$, $s_1 = 1$ and $s_2 = 2$; The automatically deformed contour reduces the condition number κ by 14 orders of magnitude and produces a contour similar to the one created by analytic deformation using the method of nonlinear steepest descent, including the bifurcations at the stationary points of the phase function.

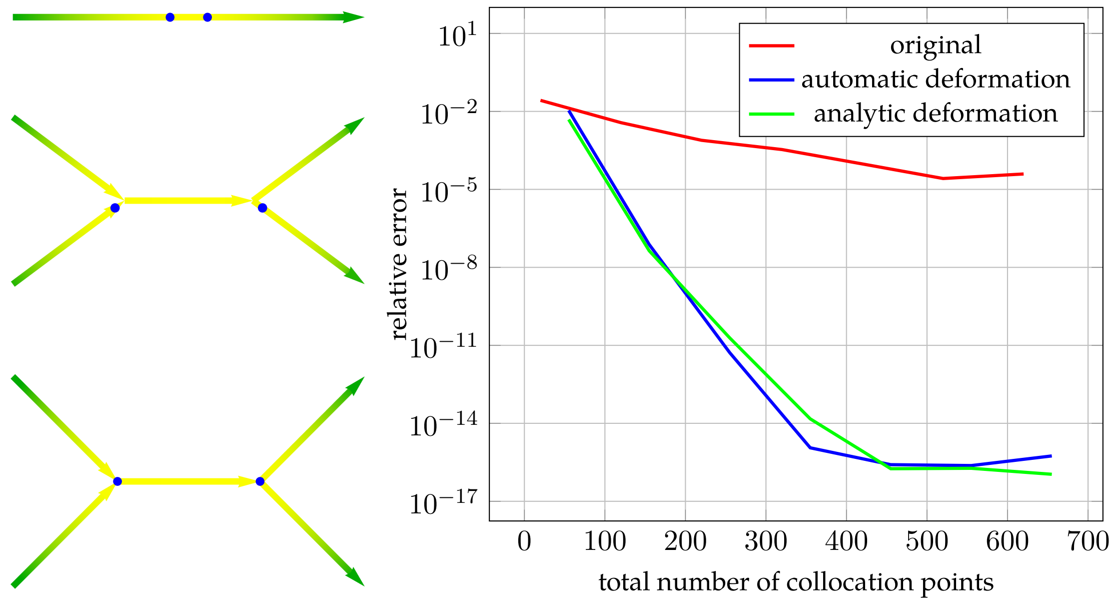


Figure 7.2.: Visualization according to style 1; *Left*: Original (top) automatically deformed (middle) and analytically deformed contour (bottom) of the mKDV RHP for $t = 1$,$x = -5$ and $r(z) = \frac{1}{2}e^{-z^2}$; The condition number is about 20 for all three contours. Once again both deformation methods yield similar contours and create bifurcations at the stationary points. *Right*: Comparison of the convergence rates for the numerical solution for the contours on the left; Both deformations yield equally fast converging solutions, which is no big surprise considering the huge similarity between both contours.

A. Software Documentation

The software documented here is a Mathematica implementation of the RHP deformation algorithm presented in § 4. It is an addon for RHPackage developed by Sheehan Olver.

A.1. Defining a RHP

A RHP is defined with the `rhproblem` tag as follows

```
rhp = rhproblem[{G1,G2 ,...} ,{Γ1,Γ2 ,...}]
```

thereby the G_j are jump matrices and the Γ_j are the corresponding contour parts. Each G_j has to be a matrix valued function depending on exactly one complex variable. E.g.

```
G1 = Function[{z}, {{1 ,z} ,{0 ,1}}]
```

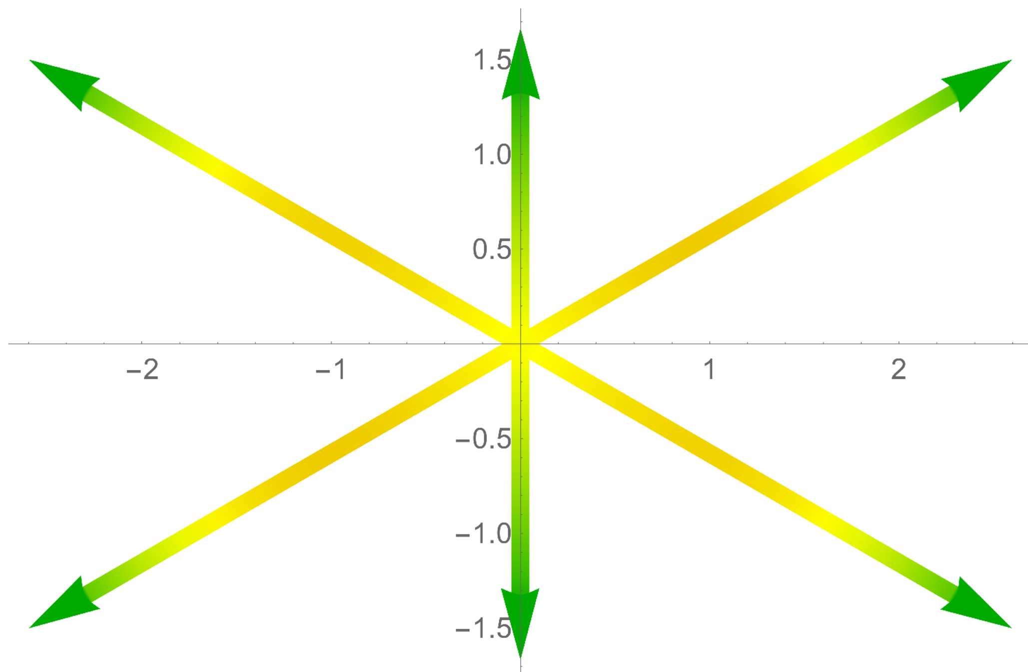
Each part of the contour can have one of the following forms

```
(* line between the points a ∈ ℂ and b ∈ ℂ *)
Γ1 = Line[{a,b}]

(* ray given by a+r*b with r going from 0 to ∞ *)
Γ2 = Line[{a, DirectedInfinity[b]}]

(* ray given by a+r*b with r going from ∞ to 0 *)
Γ3 = Line[{DirectedInfinity[b],a}]

(* union of Line[{DirectedInfinity[a],0}]
   and Line[{0, DirectedInfinity[b]}] *)
Γ4 = Line[{DirectedInfinity[a], DirectedInfinity[b]}]

(* union of line segments between consecutive
   points in the list {a,b,c,...} *)
Γ5 = Line[{a,b,c, ... }]

(* circle around c with radius r and
   mathematical orientation o ∈ {−1,1} *)
Γ6 = Circle[c, r, o]
```

Example

The function `p2rhp[x,s1,s2,s3]` defined in the following code, returns the Painlevé II RHP for the given parameters x,s1,s2 and s3.

```
g[x_, z_] := (8 I)/3 z^3 + 2 I x z

p2rhp[x_,s1_,s2_,s3_] := rhproblem[
    {
        Function[{z}, {{1, 0}, {s1 Exp[g[x, z]], 1}},
        Function[{z}, {{1, s2 Exp[-g[x, z]]}, {0, 1}},
        Function[{z}, {{1, 0}, {s3 Exp[g[x, z]], 1}},
        Function[{z}, {{1, -s1 Exp[-g[x, z]]}, {0, 1}},
        Function[{z}, {{1, 0}, {-s2 Exp[g[x, z]], 1}},
        Function[{z}, {{1, -s3 Exp[-g[x, z]]}, {0, 1}}
    } , {
        Line[{0, DirectedInfinity[Exp[I Pi  1/6]]}],
        Line[{0, DirectedInfinity[Exp[I Pi  3/6]]}],
        Line[{0, DirectedInfinity[Exp[I Pi  5/6]]}],
        Line[{0, DirectedInfinity[Exp[I Pi  7/6]]}],
        Line[{0, DirectedInfinity[Exp[I Pi  9/6]]}],
        Line[{0, DirectedInfinity[Exp[I Pi 11/6]]}]
    }
]
rhp = p2rhp[-10,1,2,1/3]
```



A.2. Deformation operations

A.2.1. SimpleDeformation

This function is an implementation of algorithm 1 and performs automatic contour deformation. The only required input is a RHP in the just described form of **rhproblem** data structure. There are a few options available though, which are listed below in combination with their default values.

"Nodes"−> 17
admissible values: ℕ or {y,x} $y, x \in \mathbb{R}$
The number of vertices in y and x direction which is used to create the base grid of the graphs described in § 4.8.2. If only a single value is given, both directions use the same number of vertices.

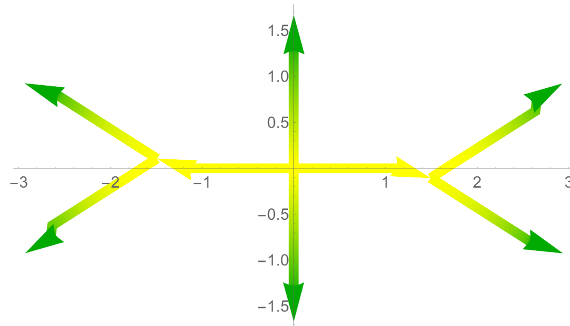
"VertexCoordinatesRange"−> **Automatic**
admissible values: {{a,b},{c,d}} $a, b, c, d \in \mathbb{R}$
Specifies the region of the complex plane to discretize. It is either chosen automatically (default) or the rectangle defined by $a + bi, c + di$ is used. Automatic region choosing is likely to fail for RHPs whoose contour does only consists of the real axis, as in this case the height has to be guessed. The default guess for these cases is {−1,1}

DataRange −> {10^−16,10^16}
admissible values: {a,b}; **$MinMachineNumber** < a < b < **$MaxMachineNumber**
Weights of graphs are clipped to the specified range.

"ShowProgress"−> **False**
admissible values: **True**, **False**
If set to true, intermediate steps of the algorithm are printed.

Example

```
rhpDeform1 = SimpleDeformation[rhp]
```



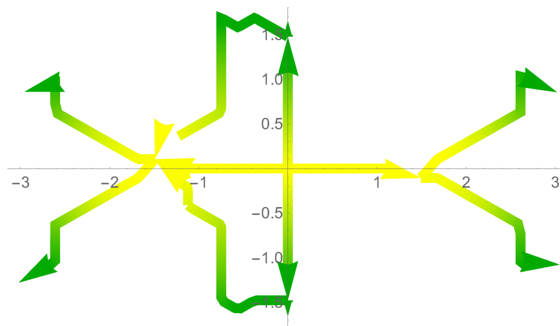
A.2.2. LensingDeformation

This function is an implementation of algorithm 10. Just as SimpleDeformation it has only one required argument, a **rhproblem** datastructure. All the options of SimpleDeformation are also available for this function. There is only one additional option.

"decompositions"−> {"LDR", "RDL"}
admissible values: d ⊆ {"LDR", "RDL", "LR", "RL"}
This is the list of matrix decompositions, which are tried while perfomring a lensing deformation.

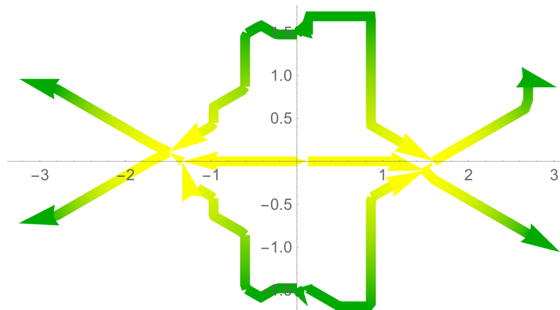
Example

```
rhpDeform2 = LensingDeformation[rhpDeform1]
```



Example

```
rhpDeform3 = LensingDeformation[rhpDeform2]
```



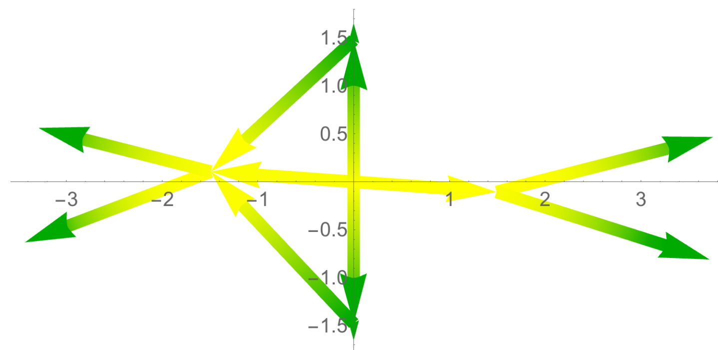
A.2.3. SimplifyContour

Both of the deformation functions create a contour by converting the calculated paths into segments with the same combined jump. Afterwards these segments are converted to contours of the form **Line**[{a,b,c ,..., d}] with a,b,c ,..., d being the positions of the vertices in each segment. To actually solve the RHP with this contour, the contour is further divied into the parts {**Line**[{a,b }], **Line**[{b,c }],...} . This can create quite a lot of parts, but as we have stated in § 4.8.3, solving a contour with fewer parts is in general faster than solving a contour with more parts. Therefore it is in general advisable to simplify the contour with SimplifyContour before solving the RHP. The method used for the simplification is selected with the following option.

Method –> "StraightRays"
admissible values: {"Safe","StraightRays","Endpoints"}
"Safe" drops all redudant points from **Line**[l]. If l contains three consecutive points a,b,c on a line, the one in the middle (b) is removed from l. Choosing "Endpoints" simply converts each part **Line**[{a ,..., b}] to a straight line **Line**[{a,b}]. "StraigRays" works the same way as "Endpoints" but restricts its operation to parts for which a or b is infinity.

Example

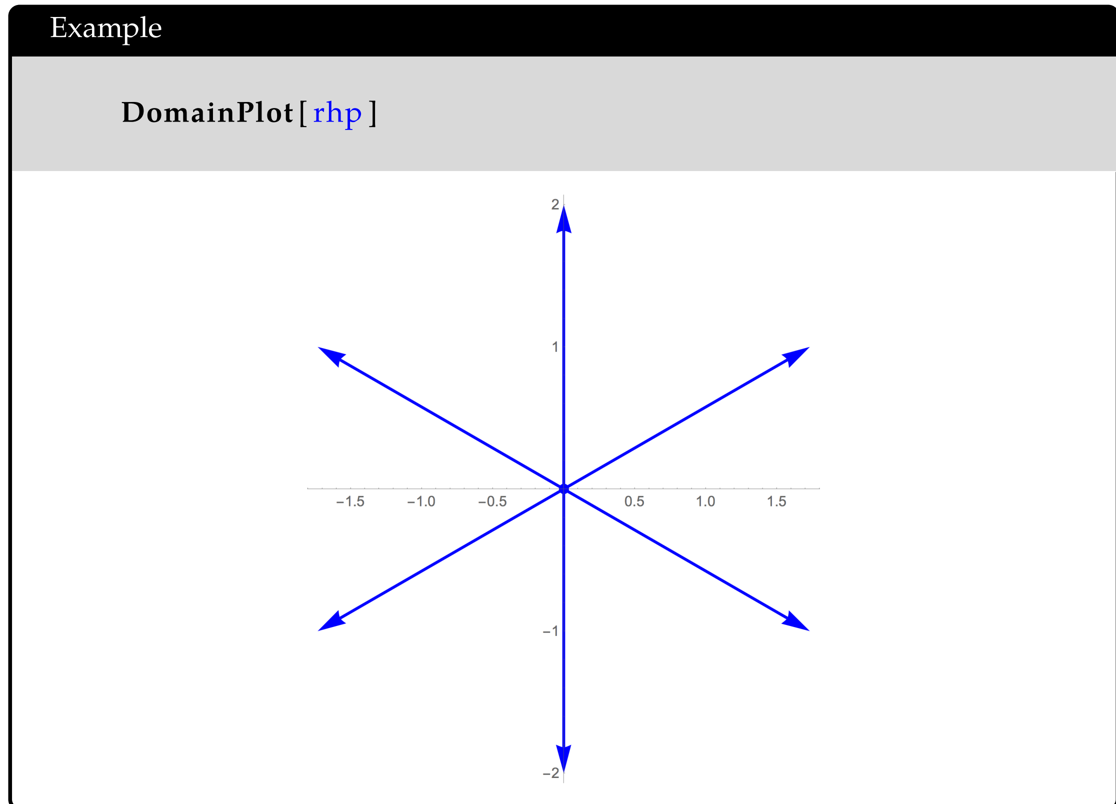
```
rhpDeform2Simple =
    SimplifyContour[rhpDeform2,Method->"Endpoints"]
```



A.3. Visualizations

A.3.1. DomainPlot

DomainPlot is part of RHPackage and plots the contour of a RHP. The arrows indicate the orientation of each part.



A.3.2. JumpScaleDomainPlot

This is the default visualization of RHPs and visualizes them according to the style 1. Apart from the common Mathematica options for **Graphics** objects there are the following options.

PlotRange −> **Automatic**
admissible values: {{a,b},{c,d}} $a, b, c, d \in \mathbb{R}$
This option specifies the region of the complex plane in which the contour is plotted. The default is to determine this region automatically, but it is also possible to manually define one. {{a,b},{c,d}} specifies the rectangle defined by the the points $a + bi, c + di$ in the complex plane.

"ColorRange"−>{−16,16}
admissible values: {a,c} or {a,b,c} ; $a, b, c \in \mathbb{R}$

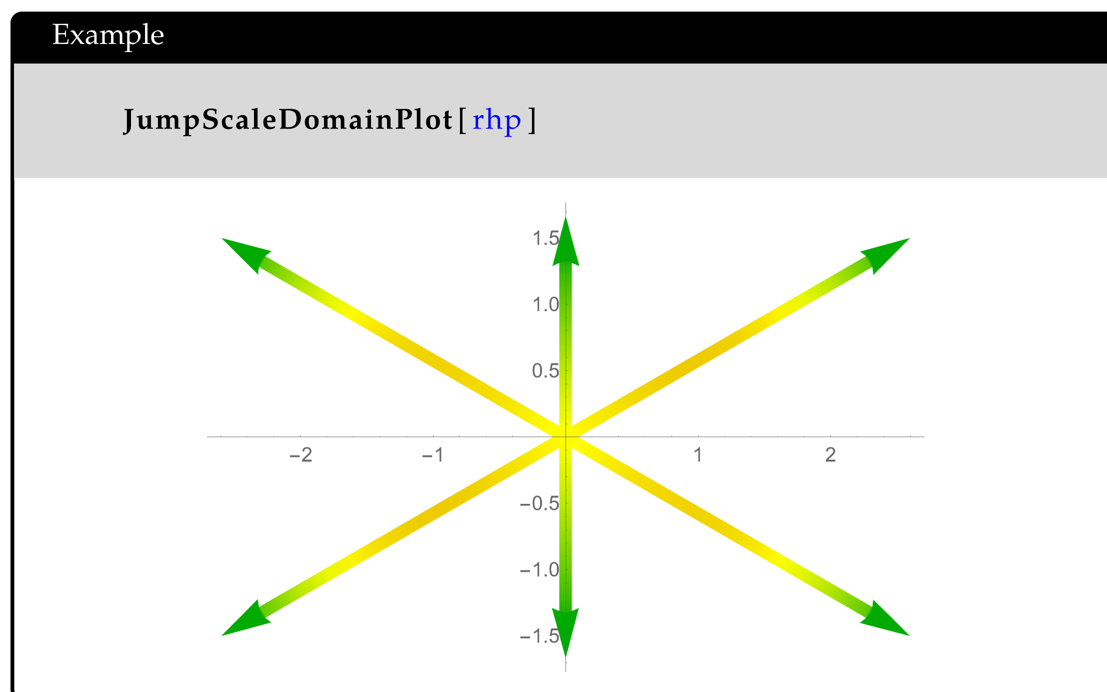
This option defines which weights corresponds to colors green (10^a), yellow(10^b) and red (10^c). In between these values the contour is colored by blending these three colors. If b is omitted it defaults to $(a + c)/2$.

PlotPoints −> 40
The number of points used to plot each part of the contour.

"Thickness"−>0.0175
The base width used to plot all the parts of the contour.

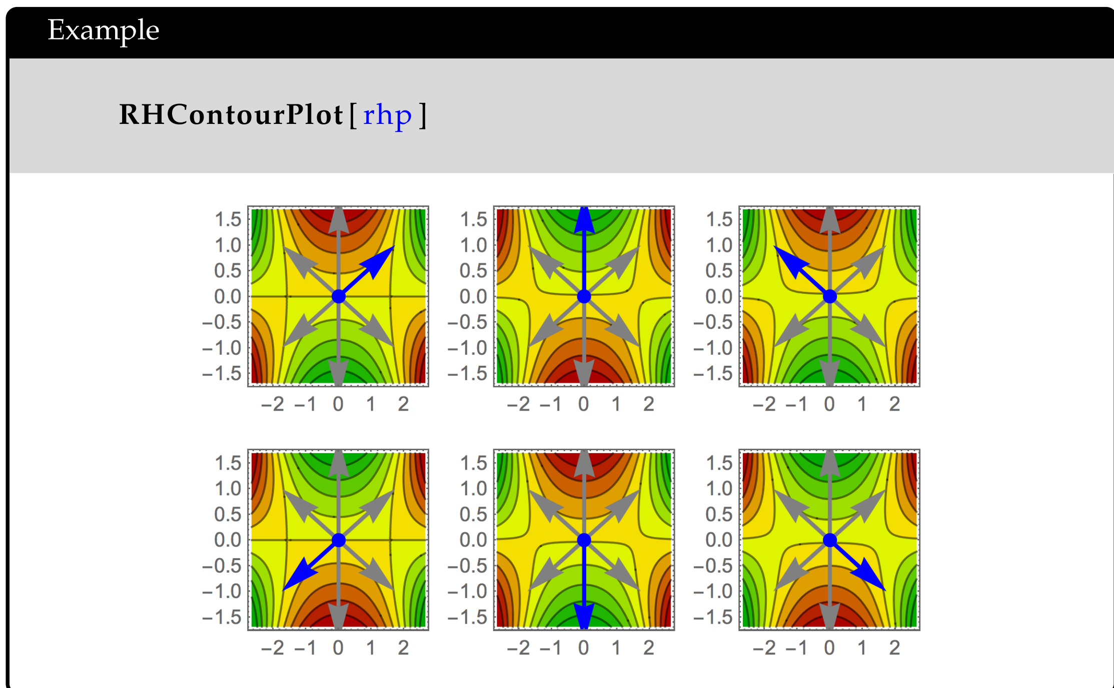
Example

```
JumpScaleDomainPlot[rhp]
```



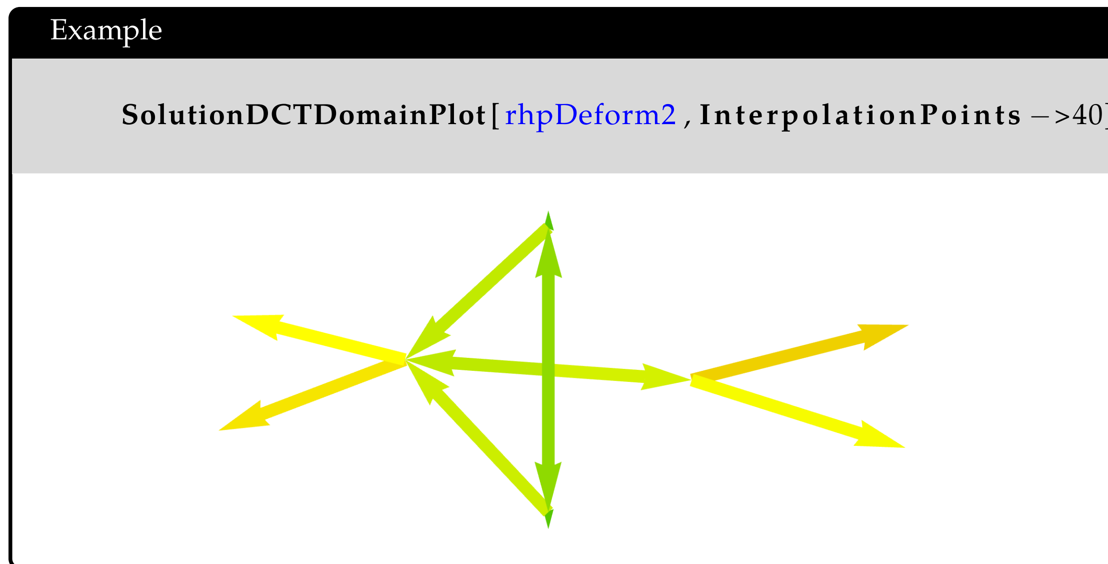
A.3.3. RHContourPlot

This plot combines for each part of the contour Γ_j a **DomainPlot** with a contour plot of corresponding weight $d(z) = \|G_j(z) - I\|$. Thereby Γ_j is highlighted in blue in the plot visualizing the weight corresponding to G_j. The colorcoding is the same one which is also used by **JumpScaleDomainPlot**. **RHContourPlot** has the two options **PlotRange** and "ColorRange" which have the same effect as for **JumpScaleDomainPlot**.



A.3.4. SolutionDCTDomainPlot

This visualization is useful to figure out where the solution of a RHP converges and where it does not. The plot is similiar to **JumpScaleDomainPlot** but in this case the color of each part of contour encodes the fraction of the last coefficient divided by the first coefficient of the discrete cosinus transform of the solution on this part. Once again green stands for 10^{-16} (converged), yellow for 1 (not converged) and red for 10^{8} (diverging). The solution of the RHP is calculated by calling **RHSolve** with the options passed to this function. As expected the options for this function are the combination of all options for **JumpScaleDomainPlot** and **RHSolve** (see next section).



A.4. Solving RHPs

A.4.1. RHSolve / RHSolveTop

RHPs can be solved with the **RHSolve** and **RHSolveTop** functions. The first one calculates a solution of the whole RHP while the second one will only solve the first row of the RHP. Both functions are wrappers that convert the **rhproblem** data structure to the one expected by the RHSolver respectively RHSolverTop functions of RHPackage and calls them. Before the RHP is solved, both functions convert all parts of the contour to straight lines. The resulting contour can be seen by applying **ToStraightLines** to the RHP. **Remark:** Currently these function don't check that this deformation to straight lines is actually valid. Therefore this has to be verified manually. E.g. if the RHP contains a circle, converting the circle to straight lines may cause it to intersect other parts of the contour.

InterpolationPoints −> **Automatic**
admissible values: n with $n \in \mathbb{N}$ or {n1,n2 ,...} with $n1, n2, ... \in \mathbb{N}$
This settings specifies how many collocation points should be used to discretize each contour part. It is possible to either speicify one value n which is used for all parts or a list that contains one number for each contour part.

InterpolationPrecision −> 10^−8
admissible values: $(10^{-16}, 1)$
InterpolationPoints has to be set to **Automatic** for this option to have any effect. The number of collocation points for each contour part is chosen such that the jump functions are approximated upto the given precision. This does not guarantee that the solution

will have the same precision, but it is a good initial guess for the number of collocation points.

"BuildSolution" −> **True**
admissible values: **True**,**False**
If set to **False**, it does not return the solution of the RHP but the data structure return by RHSolver. This datastructure represents U as in § 4.1. The solution of the RHP is then given by a Cauchy transform of this data structure.

Example

```
phi = RHSolve[rhpDeform2Simple];
phi[1+1 i]
```

```
{{0.893822   - 0.0512298 I, 0.0104871 - 0.00886034 I},
 {0.00110543 - 0.0730774 I, 1.11447   + 0.063008   I}}
```

A.4.2. ConditionNumber

ConditionNumber returns the condition number of the linear system solved by **RHSolve** / **RHSolveTop**. Its options are identical to RHsolve, though the default value is **InterpolationPoints** −> 2
. In generel 20 points turned out to be enough to give a reasonable accourate approximation of the condition number of a RHP.

Example

```
ConditionNumber[rhp]
ConditionNumber[rhpDeform2Simple]
```

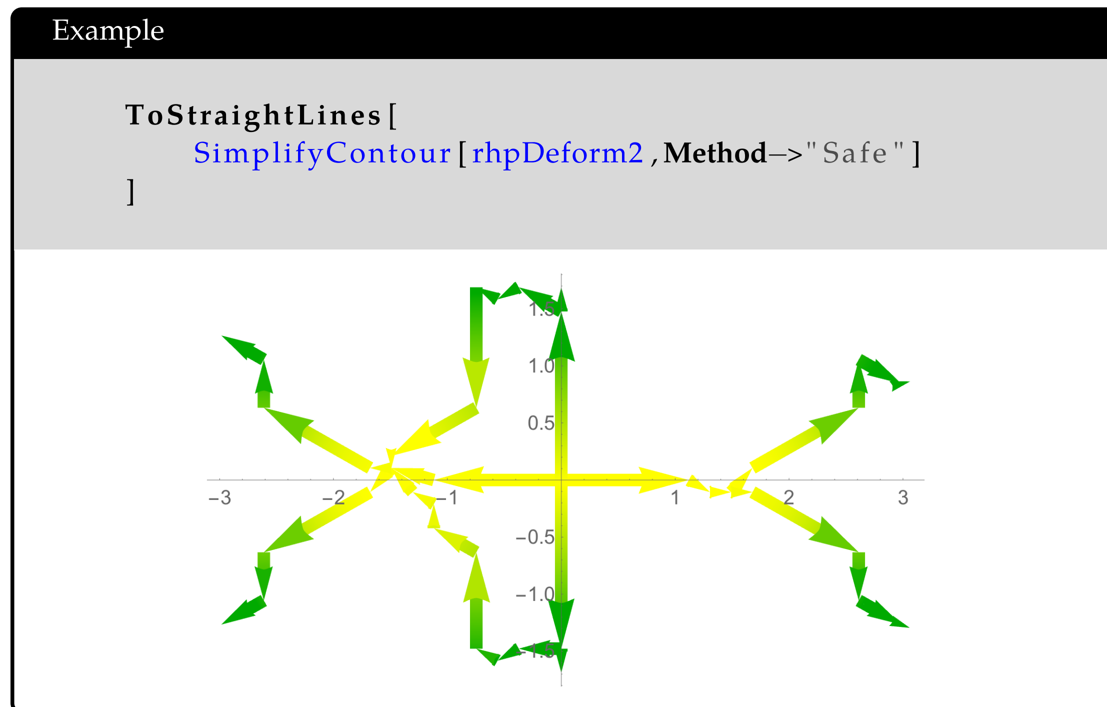
```
2.28081*10^8
423.875
```

A.5. Utility Functions

A.5.1. ToStraightLines

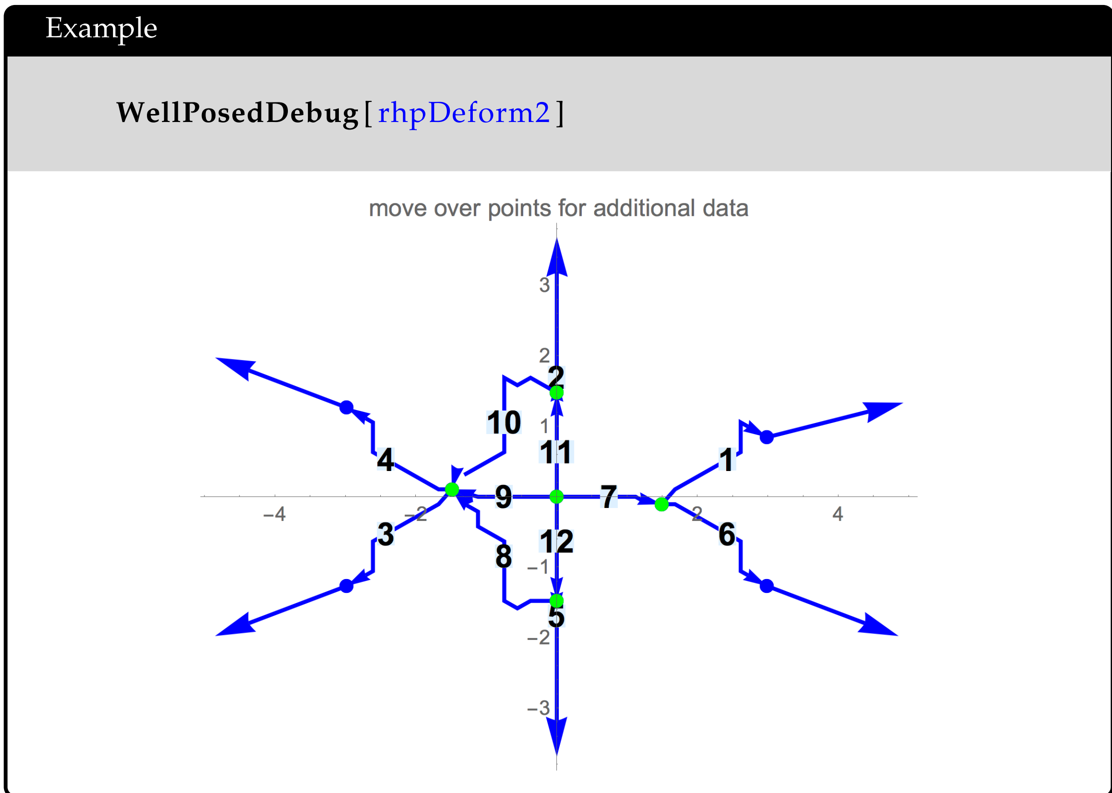
This functions converts all parts of the contour of a RHP into a set of straight lines. E.g. a line with multiple segments (such as Γ_5 in appendix A.1), indivual lines are created for

each segment.



A.5.2. WellPosedDebug

This is a basic check for errors in the RHP defintion or implementation. If we multiply the jump matrices we encounter by going in positive direction around a point where parts of the contour meet, the resulting product has to be identity matrix. This fact is verified by this function. In the plot created by this function all points satisfying this condition are marked green and the remaining points are marked red.


Example
WellPosedDebug[rhpDeform2]
move over points for additional data
1
2
3
4
5
6
7
8
9
10
11
12
−4
−2
2
4
3
2
1
−1
−2
−3

B. Deformation Verification: Definition of ν

This chapter contains the exact definition of ν mentioned in section 4.7 for all possible cases.

B.1. Notation

- $\measuredangle\, acb$: the angle from the line ca to the line cb
- $\arg$: common choice of the argument function with the range $(-\pi, \pi)$
- $\arg_{2\pi}$: alternative choice of the argument function with the range $(0, 2\pi)$
- $\dot{v}$: position (in $\mathbb{C}$) of the vertex v
- $\measuredangle(g, p, a)$: the angle at a which we obtain, if we take a straight line from a to the path p and move the endpoint on p from the beginning of p to its end
- $h : \omega \mapsto \begin{cases} 2\pi - \omega & \text{if } \omega > \pi \\ \omega & \text{else} \end{cases}$

B.2. Only Continuous Contour Parts

Points

$$\nu(P_a, \bullet) = 0$$

Our algorithm doesn't move points, therefore we don't have to check any restrictions for them.

Intervals

$$\begin{aligned} \nu(I_{a,b}, P_c) &= (\measuredangle\, acb) \\ \nu(I_{a,b}, I_{c,d}) &= (\nu(I_{a,b}, P_c), \nu(I_{a,b}, P_d)) \\ \nu(I_{a,b}, R_c(d)) &= (\nu(I_{a,b}, P_d)) \\ \nu(I_{a,b}, L_c) &= 0 \\ \nu(I_{a,b}, C_r^s(c)) &= 0 \end{aligned}$$

As changing the path between two points has no effect on whether this path is left / right of a line or inside / outside of a circle, we do not have to check anything in the last two cases.

Rays

$$\begin{aligned}
\nu(R_c(a), P_b) &= (h(\arg c - \arg(b-a))) \\
\nu(R_c(a), I_{b,d}) &= (\nu(R_c(a), P_b), \nu(R_c(a), P_d)) \\
\nu(R_c(a), R_d(b)) &= \begin{cases} 0 & \text{if } a = b \\ \nu(R_c(a), P_b & \text{else} \end{cases} \\
\nu(R_c(a), L_d) &= 0 \\
\nu(R_c(a), C_r^s(b)) &= 0
\end{aligned}$$

The last 2 cases can also be ignored in this case. As neither lines nor circles have endpoints, there are not restrictions to check.

Lines

$$\begin{aligned}
\nu(L_c, P_a) &= \begin{cases} (\pi) & \text{if } \arg(-c) > \arg a > \arg c \quad \text{and } \arg(-c) > 0 \\ (\pi) & \text{if } \arg_{2\pi}(-c) > \arg_{2\pi} a < \arg_{2\pi} c \quad \text{and } \arg(-c) < 0 \\ (-\pi) & \text{else} \end{cases} \\
\nu(L_c, I_{a,b}) &= (\nu(L_c, P_a), \nu(L_c, P_b)) \\
\nu(L_c, R_d(a)) &= (\nu(L_c, P_a)) \\
\nu(L_c, L_d) &= 0 \\
\nu(L_c, C_r^s(a) &= 0
\end{aligned}$$

The second to last case cannot happen, as two lines will always intersect and intersecting contour parts are not allowed. The last case is also not of interest, as the information whether a circle is above or below a line is already handled by the point added for the center of the circle.

Circles

$$\begin{aligned}
\nu(C_r^s(a), P_b) &= \begin{cases} (s2\pi) & \text{if } b \in B_r(a) \\ (0) & \text{else} \end{cases} \\
\nu(C_r^s(a), I_{b,c}) &= (\nu(C_r^s(a), P_b), \nu(C_r^s(a), P_c)) \\
\nu(C_r^s(a), R_c(d)) &= (0) \\
\nu(C_r^s(a), L_c) &= 0 \\
\nu(C_{r_1}^{s_1}(a), C_{r_2}^{s_2}(b) &= \begin{cases} (s_1 2\pi) & \text{if } b + r_2 \in B_{r_1}(a) \\ (0) & \text{else} \end{cases}
\end{aligned}$$

A circle can never contain a ray or a line. In the case of lines this does add any restrictions, as we can always find a line around the circle, but in the case of rays we have to add the restriction that the circle does not contain the endpoint of the ray. Otherwise it could happen that we discretize the circle with a path that encloses this endpoint and end up in a situation where no valid discretization for the ray exists anymore. The last definition handles the case that a circle is inside another circle by distinguishing which of the two is the inner and which is the outer circle. This information is not provided by the centers of the circle, if both circles enclose both centers.

B.3. Only Paths

Obviously if one of the arguments of ν is a path, we need to know the graph which corresponds to the path to evaluate ν. In order to avoid cluttering the notation we not explicitly include this dependence in the following formulas.

Points : $p \in P_P$

$$\nu(p, \bullet) = 0$$

Intervals : $p \in P_I$

$$\begin{aligned}
\nu(p, v \in P_P) &= (\measuredangle(g, p, v)) \\
\nu(p, v \ldots w \in P_I) &= (\measuredangle(g, p, v), \measuredangle(g, p, w)) \\
\nu(p, v \ldots w \in P_R) &= (\measuredangle(g, p, v)) \\
\nu(p, q \in P_L) &= 0 \\
\nu(p, q \in P_C) &= 0
\end{aligned}$$

Rays : $p \in P_R$

$$\begin{aligned}
\nu(p, v \in P_P) &= h(\arg(w) - \arg(w - v)) + \measuredangle(g, p, v) \\
\nu(p, v \ldots w \in P_I) &= (\nu(p, v), \nu(p, w)) \\
\nu(p = u \ldots, v \cdots \in P_R) &= \begin{cases} 0 & \text{if } u = v \\ (\nu(p, v)) & \text{else} \end{cases} \\
\nu(p, q \in P_L) &= 0 \\
\nu(p, q \in P_C) &= 0
\end{aligned}$$

Lines : $p \in P_L$

$$\nu(p, v \in P_P) = \begin{cases} (\pi) & \text{if } \measuredangle(g,p,v) > 0 \\ (-\pi) & \text{if } \measuredangle(g,p,v) < 0 \end{cases}$$
$$\nu(p, v \ldots w \in P_I) = (\nu(p,v), \nu(p,w))$$
$$\nu(p, v \ldots w \in P_R) = (\nu(p,v))$$
$$\nu(p, q \in P_L) = 0$$
$$\nu(p, q \in P_C) = 0$$

Circles : $p \in P_C$

$$\nu(p, v \in P_P) = \measuredangle(g,p,v)$$
$$\nu(p, v \ldots w \in P_P) = (\nu(p,v), \nu(p,w))$$
$$\nu(p, v \cdots \in P_R) = (0)$$
$$\nu(p, q \in P_L) = 0$$
$$\nu(p, v \ldots v \in P_C) = \nu(p,v)$$

B.4. First Argument: Path, Second Argument: Continuous Contour

In this case we cannot always determine a single value for ν. The path may contain a vertex which is at the same location as one of the endpoints of the continuous contour. The corresponding endpoint can be mapped to any vertex at its location, so it could be either left or right of the path. Therefore we determine the angles for all possible mappings. The resulting information can be used to determine valid mappings and detect software bugs, as there has to be at least one valid mapping at all times.[1] Basically we reduce these cases to the definitions in section B.3 by mapping the endpoints of the paths to vertices.

Points : $p \in P_P$

$$\nu(p, \bullet) = 0$$

[1]A vertex can only be left or right of the path, and for both options a vertex has to exist as there is always at least one vertex left and one right of a path along which a graph is split.

Intervals : $p \in P_I$

$$\begin{aligned}
\nu(p = u \dots w, P_a) &= \begin{cases} 0 & \text{if } \dot{u} = a \text{ or } \dot{w} = a \\ \{\measuredangle(g, p, v) : v \in \mathrm{nv}(g, a)\} & \text{else} \end{cases} \\
\nu(p, I_{a,b}) &= (\nu(p, P_a, \nu(p, P_b)) \\
\nu(p, R_c(a)) &= (\nu(p, P_a)) \\
\nu(p, L_c) &= 0 \\
\nu(p, C_r^s(a)) &= 0
\end{aligned}$$

Rays : $p \in P_R$

$$\begin{aligned}
\nu(p = u \dots w, P_a) &= \begin{cases} 0 & \text{if } \dot{u} = a \\ \{h(\arg(w) - \arg(w - u)) + \measuredangle(g, p, v) : v \in \mathrm{nv}(g, a)\} & \text{else} \end{cases} \\
\nu(p, I_{a,b}) &= (\nu(p, P_a), \nu(p, P_b)) \\
\nu(p, R_c(a)) &= \nu(p, P_a) \\
\nu(p, L_c) &= 0 \\
\nu(p, C_r^s(a)) &= 0
\end{aligned}$$

Lines : $p \in P_L$

$$\begin{aligned}
\nu(p, P_a) &= \{\nu(p, v) : v \in \mathrm{nv}(g, a)\} \\
\nu(p, I_{a,b}) &= (\nu(p, P_a), \nu(p, P_b)) \\
\nu(p, R_c(a)) &= (\nu(p, P_a)) \\
\nu(p, L_c) &= 0 \\
\nu(p, C_r^s(a)) &= 0
\end{aligned}$$

Circles : $p \in P_C$

$$\begin{aligned}
\nu(p, P_a) &= \{\nu(p, v) : v \in \mathrm{nv}(g, a)\} \\
\nu(p, I_{a,b}) &= (\nu(p, P_a), \nu(p, P_a)) \\
\nu(p, R_c(a)) &= (\nu(p, P_a) \\
\nu(p, L_c) &= 0 \\
\nu(p, C_r^s(a)) &= \nu(p, v)
\end{aligned}$$

B.5. First Argument: Continuous Contour, Second Argument: Path

The definitions for the cases in which the first argument of ν is a continuous contour are just reduced to the cases in section B.2 by converting the paths to continuous contours. This way we can at least verify that all endpoints are mapped to the correct vertices.

Points

$$\nu(P_a, \bullet) = 0$$

Intervals

$$\begin{aligned}
\nu(I_{a,b}, v \in P_P) &= \nu(I_{a,b}, P_{\dot{v}}) \\
\nu(I_{a,b}, v \dots w \in P_I) &= \nu(I_{a,b}, I_{\dot{v},\dot{w}}) \\
\nu(I_{a,b}, v \dots w \in P_R) &= \nu(I_{a,b}, R_{\dot{w}}(\dot{v})) \\
\nu(I_{a,b}, q \in P_L) &= 0 \\
\nu(I_{a,b}, q \in P_C) &= 0
\end{aligned}$$

Rays

$$\begin{aligned}
\nu(R_c(a), v \in P_P) &= \nu(R_c(a), P_{\dot{v}}) \\
\nu(R_c(a), v \dots w \in P_I) &= \nu(R_c(a), I_{\dot{v},\dot{w}}) \\
\nu(R_c(a), v \dots w \in P_R) &= \nu(R_c(a), R_{\dot{w}}(\dot{v}) \\
\nu(R_c(a), q \in P_L) &= 0 \\
\nu(R_c(a), q \in P_C) &= 0
\end{aligned}$$

Lines

$$\begin{aligned}
\nu(L_c, v \in P_P) &= \nu(L_c, P_{\dot{v}}) \\
\nu(L_c, v \dots w \in P_I) &= \nu(L_c, I_{\dot{v},\dot{w}}) \\
\nu(L_c, v \dots w \in P_R) &= \nu(L_c, R_{\dot{w}}(\dot{v})) \\
\nu(L_c, q \in P_L) &= 0 \\
\nu(L_c, q \in P_C) &= 0
\end{aligned}$$

Circles

$$\begin{aligned}
\nu(C_r^s(a), v \in P_P) &= \nu(C_r^s(a), P_{\dot{v}} \\
\nu(C_r^s(a), v \dots w \in P_I) &= (\nu(C_r^s(a), P_{\dot{v}}), \nu(C_r^s(a), P_{\dot{w}})) \\
\nu(C_r^s(a), v \dots w \in P_R) &= 0 \\
\nu(C_r^s(a), q \in P_L) &= 0 \\
\nu(C_r^s(a), q \in P_C) &= ?
\end{aligned}$$

In the last case we cannot determine if one of the circles is inside or outside of the other circle, as q may intersect the contionuous circle. Consequently this restricion should be ignored when checking for valid deformations. Therefore ν is set to be an undefined value which always yield true when compared to other values.

C. Visualization Styles

Visualization Style 1 (RHP Contours)**.**

- *The color encodes* $||G_j(z) - I||_F$ *along* Γ_j *with a logarithmic scale.*
- *The values of* $||G_j(z) - I||_F$ *are truncated to* $[10^{-16}, 10^{16}]$.
- *We use the color coding green* $\approx 10^{-16}$*, yellow* $\approx 10^0$ *and red* $\approx 10^{16}$.
- *The blue points indicate the location of the stationary points of the phase function in* G.

Visualization Style 2 (Graphs)**.**

- *All edges are undirected.*
- *Arrows just indicate the orientation of the paths.*
- *All splits are enlarged for a clear visualization.*
- *Without enlargement the vertices on the left and right sides of splits coincide.*

Bibliography

Bobenko, A. I. and Its, A.: 2014, The asymptotic behaviour of the discrete holomorphic map Z^a via the Riemann-Hilbert method, *ArXiv e-prints* .

Bobenko, A. I. and Pinkall, U.: 1998, Discretization of surfaces and integrable systems, *In: Discrete Integrable Geometry and Physics; Eds.: A.I. Bobenko and*, University Press, pp. 3–58.

Bobenko, A. and Pinkall, U.: 1996, Discrete isothermic surfaces., *J. Reine Angew. Math.* **475**, 187–208.

Bornemann, F.: 2011, Accuracy and stability of computing high-order derivatives of analytic functions by cauchy integrals, *Foundations of Computational Mathematics* **11**(1), 1–63.

Bornemann, F. and Wechslberger, G.: 2013, Optimal contours for high-order derivatives, *IMA Journal of Numerical Analysis* **33**(2), 403–412.
URL: *http://imajna.oxfordjournals.org/content/33/2/403.abstract*

Clenshaw, C. and Curtis, A.: 1960, A method for numerical integration on an automatic computer, *Numerische Mathematik* **2**(1), 197–205.
URL: *http://dx.doi.org/10.1007/BF01386223*

Deift, P. and Zhou, X.: 1993, A steepest descent method for oscillatory Riemann–Hilbert problems: Asymptotics for the MKdV equation, *Ann. of Math.* **137**, 295–368.

Deuflhard, P.: 2003, *Numerical Analysis in Modern Scientific Computing*, Springer.
URL: *https://books.google.de/books?id=1wjQE4wlR40C*

Fokas, A. S., Its, A. R., Kapaev, A. A. and Novokshenov, V. Y.: 2006, *Painlevé Transcendents: The Riemann-Hilbert Approach*, American Mathematical Society, Providence, RI.

Fredman, M. L. and Tarjan, R. E.: 1987, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM (JACM)* **34**(3), 596–615.

Its, A. R.: 2003, The Riemann–Hilbert problem and integrable systems, *Notices Amer. Math. Soc.* **50**, 1389–1400.

Lang, S.: 1999, *Complex analysis*, Vol. 103, Springer.

Mathematica: n.d., *Extrapolation*, Wolfram Research, Inc.
URL: *http://reference.wolfram.com/mathematica/tutorial/NDSolveExtrapolation.html*

Olver, S.: 2011a, Computation of equilibrium measures, *Journal of Approximation Theory* **163**(9), 1185–1207.

Olver, S.: 2011b, Numerical solution of Riemann-Hilbert problems: Painlevé II, *Found. Comput. Math.* **11**, 153–179.

Olver, S. and Trogdon, T.: 2012, Nonlinear steepest descent and the numerical solution of Riemann–Hilbert problems. e-print: `arXiv:1205.5604`.

Olver, S. and Trogdon, T.: 2014, Numerical solution of riemann–hilbert problems: Random matrix theory and orthogonal polynomials, *Constructive Approximation* **39**(1), 101–149.
URL: *http://dx.doi.org/10.1007/s00365-013-9221-3*

Provan, J. S.: 1989, Shortest enclosing walks and cycles in embedded graphs, *Information Processing Letters* **30**(3), 119–125.

Tracy, C. A. and Widom, H.: 1994, Fredholm determinants, differential equations and matrix models, *Comm. Math. Phys.* **163**(1), 33–72.
URL: *http://projecteuclid.org/euclid.cmp/1104270379*

Trogdon, T. and Olver, S.: 2012, Numerical inverse scattering for the focusing and defocusing nonlinear schrödinger equations, *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* **469**(2149).

Trogdon, T., Olver, S. and Deconinck, B.: 2012, Numerical inverse scattering for the korteweg–de vries and modified korteweg–de vries equations, *Physica D: Nonlinear Phenomena* **241**(11), 1003 – 1025.
URL: *http://www.sciencedirect.com/science/article/pii/S016727891200053X*

Wechslberger, G. and Bornemann, F.: 2014, Automatic deformation of riemann–hilbert problems with applications to the painlevé ii transcendents, *Constructive Approximation* **39**(1), 151–171.
URL: *http://dx.doi.org/10.1007/s00365-013-9199-x*