A NEW SCHUR METHOD FOR THE MATRIX-VALUED LAMBERT W FUNCTION

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2008

Daniel Kirk
School of Mathematics

Contents

Word count 20,062

List of Tables

List of Figures

The University of Manchester

Daniel Kirk

Master of Science

A new Schur Method for the Matrix-Valued Lambert W Function

September 5, 2008

A new method for computing the matrix Lambert W function is presented based on the Schur–Parlett method. The crucial modification is the new blocking algorithm which takes into account the singularities of W. The atomic blocks are evaluated using power series such as Taylor expansions. To facilitate the Taylor expansions two algorithms for computing the derivatives of W are presented and error bounds are derived for these. Expressions for the Fréchet derivative and its adjoint are derived which are then used to estimate the condition number of W. Upper and lower bounds are derived for the forward error of the algorithm which is then rigorously tested using a variety of test matrices. The method appears to offer a stable and efficient way of computing the Lambert W function of most matrices, and where it fails to do so the tools developed allow the error to be estimated.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

Acknowledgements

It is a pleasure to thank my supervisor Nick Higham for his advice and insight, as well as for his many relevant publications on the subject. I also thank Robert Corless who introduced me to automatic differentiation. I thank my family and friends for their patience and support. Finally I thank to the EPSRC for their financial assistance which enabled me to pursue this MSc.

Chapter 1

Introduction and Preliminaries

1.1 Introduction

The Lambert W function finds use in many diverse areas of physics and mathematics, from calculating the density and velocity of solar wind to the enumeration of trees in combinatorics. The function, often written as W is named for the Swiss Mathematician Johann Heinrich Lambert (1728–1777) who was the first to set a problem requiring W for the solution [6].

The Lambert W function (also known as the product log function) is essentially the inverse of xe^x much in the same way the logarithm is the inverse of the exponential function. Both the logarithm and exponential have long since been defined as matrix functions and plenty of robust and sophisticated algorithms exist for calculating them (See [16]). In [7] the Lambert W function was defined as a matrix function and given a firm theoretical grounding. To date little work has been done on developing or investigating algorithms for computing the Lambert W Function numerically. This thesis presents a new algorithm for W based on the Schur–Parlett method. Along the way two algorithms are given for calculating the scalar derivatives of W and a priori error bounds are given for each.

The conditioning of the Lambert W function is also investigated and various theorems about the Fréchet derivative are proved, and its adjoint is derived. In the final chapter the stability of the Schur–Parlett algorithm for the majority of matrices is empirically demonstrated through numerical experiments performed in MATLAB.

1.2 Complex Functions

For a complex variable $z \in \mathbb{C}$ the *real part* of z is denoted $\Re(z)$ and the *imaginary part* $\Im(z)$.

Definition 1.2.1. For a function $f\colon X \to Y$ we have the following terms:

- The domain is the set X
- The codomain is the set Y
- The range is the set $f(X) = \{y \in Y \colon \exists x \in X \text{ where } f(x) = y\}$
- The image of a subset of the domain: $S \subseteq X$ is $f(S) = \{y \in S \colon \exists x \in X \text{ where } f(x) = y\}$
- The pre-image of a subset of the codomain: $T \subseteq Y$ is $f^{-1}(T) = \{x \in X \colon f(x) \in T\}$.

Definition 1.2.2. A function $f\colon \Omega \to \mathbb{C}$ where $\Omega \subseteq \mathbb{C}$ is *analytic at* $\alpha \in \Omega$ (or *holomorphic*) if all derivatives $f^{(n)}$ are defined at α and in a neighbourhood of α one can write

$$f(x) = \sum_{k=0}^{\infty} a_k (x - \alpha)^k.$$

The function f is said to an *analytic function* (alternatively a *holomorphic function*) if it is analytic at every point in its domain.

1.3 Linear Algebra

Two matrices $A, B \in \mathbb{C}^{n\times n}$ are similar if there exists a nonsingular $P \in \mathbb{C}^{n\times n}$ such that $A = P^{-1}BP$. A matrix is *diagonalisable* if it is similar to some diagonal matrix. A^T denotes the *transpose* of A and A^* the *conjugate transpose* of A, $\rho(A)$ denotes the *spectral radius* of A.

Jordan Canonical Form

Definition 1.3.1. A square matrix $J(\lambda) \in \mathbb{C}^{n\times n}$ is a *Jordan block* if it has the value λ along the diagonal, ones all along the superdiagonal and zeroes everywhere else.

Definition 1.3.2. A matrix $A \in \mathbb{C}^{n\times n}$ is *in Jordan form* if it is block diagonal and has Jordan blocks along the diagonal.

Most square matrices are diagonalisable, that is the set of diagonalisable square matrices is dense in the set of all square matrices. Nondiagonalisable matrices are nevertheless similar to something almost as simple, namely a unique matrix in Jordan form (unique up to the ordering of the blocks). Note that a diagonal matrix is also a matrix in Jordan form, being made up of one by one Jordan blocks.

Theorem 1.3.3. Any matrix $A \in \mathbb{C}^{n\times n}$ is similar to a matrix in Jordan form.

Proof. See Halmos [13, pp. 112 ff.]. $\square$

This is a very useful result theoretically, in many cases it allows us to prove a theorem true for a matrix A by proving it true for its Jordan form.

Definition 1.3.4. A matrix $A \in \mathbb{C}^{n\times n}$ is said to be *defective* if it does not have a full set of n linearly independent eigenvectors, otherwise it is *nondefective.* A is said to be *derogatory* if it has more than one Jordan block with the same eigenvalue, otherwise it is *nonderogatory.*

Definition 1.3.5. A matrix $A \in \mathbb{C}^{n\times n}$ is *normal* if $AA^* = A^*A$; this is equivalent to being able to find a unitary matrix Q such that $A = Q^*DQ$ where D is diagonal. Thus normal matrices are precisely the class of matrices which can be diagonalised using unitary similarity transforms.

A is *Hermitian* (or *self-adjoint*) if it is equal to its own conjugate transpose, and *skew-Hermitian* if $A^* = -A$. If A is real then these terms become *symmetric* and *skew-symmetric* respectively. All eigenvalues of Hermitian matrices are purely real and all eigenvalues of skew-Hermitian matrices are purely imaginary. All Hermitian and skew-Hermitian matrices are normal. By their definition all normal matrices are nondefective, however they can still be derogatory.

Definition 1.3.6. Let $A \in \mathbb{C}^{n\times n}$ be a matrix with distinct eigenvalues $\lambda_1, \ldots, \lambda_m$, then the *algebraic multiplicity* of λ_i denoted $\text{algr}_A(\lambda_i)$ is the multiplicity of the root λ_i in the characteristic polynomial $c(\lambda) = \det(A - \lambda I_n)$. The *geometric multiplicity* of λ_i, denoted $\text{geom}_A(\lambda_i)$ is $\dim(\text{null}(A - \lambda_i I))$, that is the dimension of the eigenspace corresponding to λ_i which is defined as the vector space $\{v \in \mathbb{C}^n : Av = \lambda_i v\}$. If $\text{algr}_A(\lambda_i) = \text{geom}_A(\lambda_i)$ then λ_i is called *semi-simple*.

Definition 1.3.7. Given the Jordan form of A, $J = \text{diag}(J_1, \ldots, J_s)$, the *index* of λ_i denoted $\text{idx}(\lambda_i)$ is the size of the largest Jordan block associated with λ_i.

Theorem 1.3.8. Suppose A is similar to B by the relation $A = P^{-1}BP$, then A and B have the same Jordan form (up to the ordering of the Jordan blocks).

Proof. This follows from Theorem 1.3.3 due to the fact that if A is similar to B and J then B is similar to J (the transitivity property). □

1.4 Numerical Analysis

Floating Point Arithmetic

This thesis uses the standard model of floating point arithmetic with a guard digit. That is if $\text{fl}(x)$ represents x as it is stored in memory then

$$\text{fl}(x \,\text{op}\, y) = (x \,\text{op}\, y)(1+\delta), \text{ where } |\delta| \le \epsilon, \text{op} = +, -, \times, \div \text{ and } x, y \in \mathbb{R} \qquad (1.1)$$

and ϵ is the machine epsilon, or unit roundoff which is 10^{-16} for double precision floating point numbers in IEEE arithmetic.

The bounds (1.1) hold for real arithmetic, but for complex arithmetic the bounds are slightly larger; we use the notation $\gamma_n = \frac{n\epsilon}{1-n\epsilon}$ throughout. For $x, y \in \mathbb{C}$

$$\begin{aligned} \text{fl}(x \pm y) &= (x \pm y)(1+\delta), \text{ where } |\delta| \le \epsilon, & (1.2)\\ \text{fl}(xy) &= (xy)(1+\delta), \text{ where } |\delta| \le \sqrt{2}\gamma_2, & (1.3)\\ \text{fl}(x/y) &= (x/y)(1+\delta), \text{ where } y \ne 0 \text{ and } |\delta| \le \sqrt{2}\gamma_4. & (1.4) \end{aligned}$$

See [15, Chapter 2] for more on floating point arithmetic.

Stability

Definition 1.4.1. Let $f(x)$ be a differentiable function which gives exact answers and let $\hat{f}(x)$ be the same function but computed by an inexact algorithm [15, Sec. 1.5, 1.6], then we use the following terms.

1. Given an input x, the *forward error* is $|\hat{f}(x)-f(x)| = |\Delta y|$, the *relative forward error* is $|\Delta y|/|f(x)|$.

2. Given an input x if the computed result is $y+\Delta y = f(x)+\Delta y$ then the *backward error* is defined to be the smallest $|\Delta x|$ such that $f(x+\Delta x) = y + \Delta y$, the *relative backward error* is $|\Delta x|/|x|$.

3. The *absolute condition number* of f at x is $c_{\text{abs}}(f,x) = |f'(x)|$ while the *relative condition number* of f is $c_{\text{rel}}(f,x) = c_f(x) = \left|\frac{xf'(x)}{f(x)}\right|$.

4. The algorithm for computing f, (i.e. $\hat{f}$) is *backward stable* if for all inputs x the backward error is small[1], that is $\hat{f}(x) = f(x+\Delta x)$ where $|\Delta x|$ is small.

5. The algorithm for computing f is *forward stable* if for all inputs x $|\hat{f}(x) - f(x)|/c_f(x)$ is small.

6. The algorithm for computing f is considered *stable* if for all input x there exists some Δx where $|\Delta x|$ and $|f(x+\Delta x) - \hat{f}(x)|$ are both small.

Note that the condition number depends only on the function and not on the way it is calculated but stability refers explicitly to the algorithm by which f is calculated. The three notions of stability are obviously context dependent as the meaning one is prepared to ascribe to “small” depends on the problem one is trying to solve. We are most interested in algorithms which are stable. Backward stability implies stability and if f is well conditioned forward stability implies stability.

As a general rule of thumb we can think of the norm of the relative forward error being approximately less than or equal to the norm of the relative backward error magnified by the relative condition number [15, Sec. 1.6].

[1]The exact meaning of the term “small” is obviously dependent on the particular context.

Error Results

Theorem 1.4.2. Let $A, B \in \mathbb{R}^{n\times n}$ then $\mathrm{fl}(AB) = AB + \Delta$, where $\|\Delta\| \le \epsilon\|A\|\|B\|$ for any consistent norm.

Proof. See [15, Sec. 3.5] □

Corollary 1.4.3. Let $A_1, \dots, A_k \in \mathbb{R}^{n\times n}$ then $\mathrm{fl}(A_1 \dots A_k) = A_1 \dots A_k + \Delta$, where $\|\Delta\| \le \epsilon k\|A_1\| \dots \|A_k\|$ for any consistent norm.

Proof. By induction following on from Theorem 1.4.2, assume true for k prove for $k+1$. Let Δ_j represent an error term such that $\|\Delta_j\| \le \epsilon\|A_1 \dots A_{j-1}\|\|A_j\|$. Then

$$\begin{aligned}
\mathrm{fl}(A_1 \dots A_k A_{k+1}) &= \mathrm{fl}((A_1 \dots A_k)A_{k+1}) \\
&= \mathrm{fl}(A_1 \dots A_k)A_{k+1} + \Delta_{k+1}(\text{by Theorem 1.4.2}) \\
&= \mathrm{fl}(A_1 \dots A_{k-1})A_k A_{k-1} + \Delta_k A_{k+1} + \Delta_{k+1} \\
&= \mathrm{fl}(A_1 \dots A_{k-2})A_{k-1}A_k A_{k+1} + \Delta_{k-1}A_k A_{k+1} + \Delta_k A_{k+1} + \Delta_{k+1} \\
&= A_1 \dots A_{k+1} + \sum_{j=1}^{k+1} \Delta_j \prod_{i=j+1}^{k+1} A_i.
\end{aligned}$$

So the norm of the error term is bounded by

$$\begin{aligned}
\left\| \sum_{j=1}^{k+1} \Delta_j \prod_{i=j+1}^{k+1} A_i \right\| &\le \sum_{j=1}^{k+1} \|\Delta_j\| \prod_{i=j+1}^{k+1} \|A_i\| \\
&\le \epsilon \sum_{j=1}^{k+1} \|A_1 \dots A_{j-1}\|\|A_j\| \prod_{i=j+1}^{k+1} \|A_i\| \\
&\le \epsilon \sum_{j=1}^{k+1} \|A_1\| \dots \|A_{j-1}\|\|A_j\| \dots \|A_{k+1}\| \\
&= \epsilon(k+1)\|A_1\| \dots \|A_{k+1}\|.
\end{aligned}$$

□

1.5 Vectorisation and the Kronecker Product

Definition 1.5.1. Let $A \in \mathbb{C}^{n\times n}$, the vector $\mathrm{vec}(A) \in \mathbb{C}^{n^2}$ is the vector formed by stacking each of the columns of A on top of each other, with the left-most column at the top.

Definition 1.5.2. Let $A, B \in \mathbb{C}^{n\times n}$, the matrix $A \otimes B \in \mathbb{C}^{n^2\times n^2}$ is defined by

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nn}B \end{pmatrix}. \tag{1.5}$$

Definition 1.5.3. Let $A, B \in \mathbb{C}^{n\times n}$, the matrix $A \oplus B \in \mathbb{C}^{n^2\times n^2}$ is defined by $A \otimes I_n + I_n \otimes B$.

The following identities are used in the thesis, some of which are special cases of previous ones.

Lemma 1.5.4.

$$\begin{aligned} \operatorname{vec}(AXB) &= (B^T \otimes A)\operatorname{vec}(X), && (1.6)\\ \operatorname{vec}(AB) &= (I_n \otimes A)\operatorname{vec}(B) = (B^T \otimes I_n)\operatorname{vec}(A), && (1.7)\\ \exp(A \oplus B) &= \exp(A) \otimes \exp(B), && (1.8)\\ (AB \otimes CD) &= (A \otimes B)(C \otimes D). && (1.9) \end{aligned}$$

Proof. For the proofs of these see [16, Appendix B.13]. □

Chapter 2

The Scalar Valued Lambert W Function

2.1 Introduction

Starting with the equation

$$x = we^w, \tag{2.1}$$

the Lambert W function is defined, for a given $x \in \mathbb{C}$, to be any $w \in \mathbb{C}$ such that $we^w = x$; in other words, we define $W(x)$ to be the inverse function of (2.1). This is not entirely straight forward because (2.1) is not injective, meaning given x we cannot be sure of a single well-defined w. This makes $W(x)$ a multivalued function, which means that given x we can expect a number of values w such that $x = we^w$. We therefore assign a label to each w satisfying $x = we^w$ for a given x.

Logarithm Analogy

Because (2.1) is similar to the definition of the exponential we expect W to be similar to the logarithm, which is also a multivalued function. If $y = e^x$ then $x = \log(y)$ is a solution, where $\log(y)$ is the principal natural logarithm. But this is not the only solution; for any integer b, $\log(y) + 2\pi bi$ is also satisfies $y = e^x$. Indeed all solutions are of this form, so we can describe all solutions to $y = e^x$ by using a single integer parameter. So given $y = e^x$ all solutions are given by $\log_b(y) = \log(y) + 2\pi bi$. For each b, $\log_b(y)$ is a well-defined single-valued function.

We do the same thing for the Lambert W function: we define an infinite set of well-defined functions which are distinguished by a single integer parameter. Given $x = we^w$ all solutions are expressed by $w = W_b(x)$ for $b \in \mathbb{Z}$.

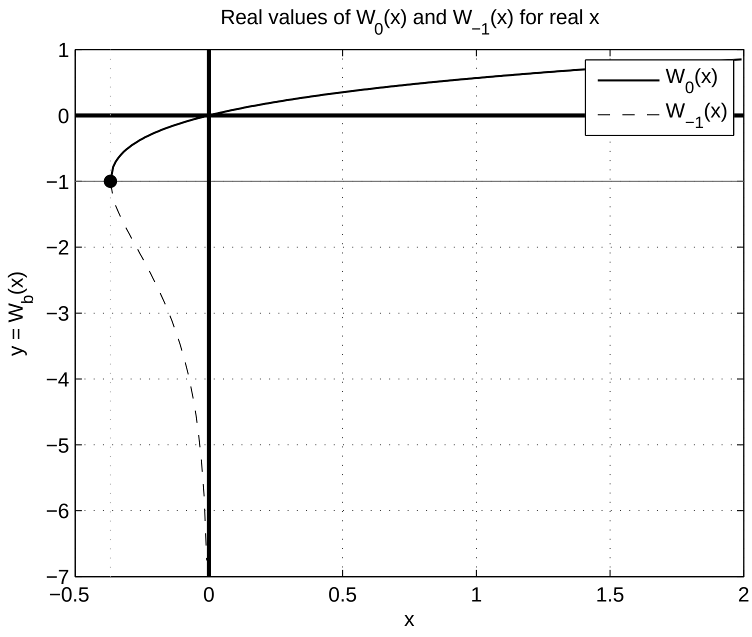

Figure 2.1: This figure shows the real values of $y = W_0(x)$ and $y = W_{-1}(x)$ for real values of x.

2.2 The Branches of $W(x)$

Following the convention used in [7] when discussing the Lambert W function we may refer to the domain as the z-plane and the codomain as the w-plane. We use these terms interchangeably with domain and codomain. In order to make sense of the infinite number of solutions we get from $W(z)$ for a given $z \in \mathbb{C}$ we divide the w-plane of W into an infinite number of connected regions called branches, where $B_k \subset \mathbb{C}$ denotes the kth branch, the idea being that each branch provides the range necessary to define $W_k \colon \mathbb{C} \to B_k$ as a single-valued inverse of $x = we^w$.

Definition 2.2.1. Given a multivalued function $f \colon \Omega \to \mathbb{C}$, where $\Omega \subseteq \mathbb{C}$ a *branch* B is a subset of the range of f such that $f \colon \Omega \to B$ is a single-valued function. If we partition the range of f into branches B_k (where k is a label such as an integer) then we refer to $f_k\colon \Omega \to B_k$ as the kth *branch* of f.

Definition 2.2.2. If we have an analytic multivalued function f defined as above then a *branch point* $\beta \in \mathbb{C} \cup \{\infty\}$ is a point in the the complex plane plus infinity such that f is not analytic at β but is analytic at all other points in some neighbourhood $\mathcal{N}$ of β; and if $p\colon [0,1] \to \mathcal{N}$ is a closed curve going around β then there exists some $s \in [0,1]$ such that $\lim_{t\uparrow s} f(p(t)) \neq \lim_{t\downarrow s} f(p(t))$ (where $t \uparrow s$ means t approaches s from below and $t \downarrow s$ means t approaches s from above).

Definition 2.2.3. For f defined as above, a *branch cut* is a curve in the domain Ω across which f is not continuous. Branch cuts must begin and end at a branch point (including ∞). If we have identified all possible branch cuts in the domain then the image of the cuts partition the range of f into branches.

The choice of branch cuts is not usually unique, although a careful choice can yield convenient algebraic properties. See [18] for details.

2.2.1 Branch Structure of the Lambert W function.

We define a countable infinity of branches for the range of W, one for each integer. The branch structure is symmetric about the real axis (apart from closure along the boundaries) and with one exception each branch extends horizontally across the entire complex plane from $-\infty$ to ∞. This one exception is the branch containing the origin. We call this the principal branch and label it $b = 0$. We identify every other branch by its position above or below this branch in the complex plane. For instance $b = -1$ is the branch directly below it, and $b = 3$ is the branch above $b = 0$ with two branches in-between.

In order to define the branch structure we first divide the w-plane using appropriate curves, then assign the integer labels (in the way we have already done above) and finally decide to which branches the points on the boundary curves belong.

The appropriate curves are derived in [7] by looking at the image of the negative real axis; as it is described in [7] we do not describe the derivation.

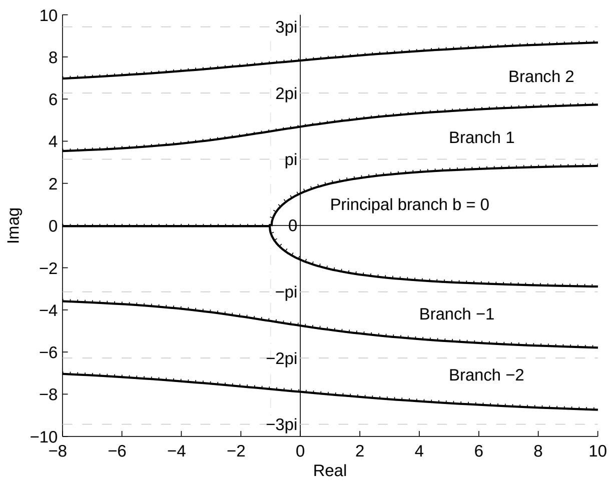

Figure 2.2: This figure shows the branch structure of the w-plane. The solid edges indicate which branch the boundaries belong to, for instance the interval $(-\infty, -1)$ belongs to the $b = -1$ branch.

The first curve we draw is defined as:

$$\{-\mu\cot(\mu) + \mu i\colon -\pi < \mu < \pi\}. \tag{2.2}$$

The branch $b = 0$ is the region to the right of this curve as well as some points on the boundary (which are addressed below). The next curve we must draw is defined by the interval:

$$(-\infty, -e^{-1}), \tag{2.3}$$

then we draw the countable infinity of curves defined by

$$\{-\mu\cot(\mu) + \mu i\colon 2\kappa\pi < \pm\mu < (2\kappa+1)\pi\} \quad \text{for all } \kappa = 1, 2, 3, \ldots. \tag{2.4}$$

This partition is shown in Figure 2.2. We now see how the curves divide the complex plane into a countable infinity of connected regions; it is these divisions which define our branches.

The $b = 0$ branch is the region which contains the positive real axis. The $b = 1$ branch is the region directly above branch $b = 0$, and branch $b = -1$ the region directly below. The $b = 2$ branch is the branch above the $b = 1$ branch and the $b = -2$ branch is directly below the $b = -1$ branch and so on in this fashion. The branches are labelled in Figure 2.2.

Branch Closure

Finally we need to define which branches the points on the boundaries of the curves belong to. This is almost arbitrary but we follow the convention in [7] and define them as follows.

Points z on (2.2) belongs to branch $b = 0$ if $\Im(z) > 0$ and to branch $b = -1$ if $\Im(z) < 0$. In the case where $\Im(z) = 0$ which occurs when $z = -e^{-1}$ we allow z to belong to both $b = 0$ and $b = -1$; this still leads to well defined functions however as $W_0(-e^{-1}) = W_{-1}(-e^{-1}) = -1$.

Points z on (2.3) all belong to branch $b = -1$. While points z on (2.4) belong to branch $b = \kappa$ if $\kappa > 0$ and branch $b = \kappa - 1$ if $\kappa < 0$.

The branches with closure information are shown in Figure 2.2. We now have a well-defined notion of all the solutions to (2.1), for (almost) every point $x \in \mathbb{C}$ there exists an infinite number of solutions given by

$$\ldots, W_{-2}(x), W_{-1}(x), W_0(x), W_1(x), W_2(x), \ldots.$$

The only times exceptions to this can occur is at the branch points (recall Definition 2.2.2). For W_0 there is only one branch point $-e^{-1}$ which is obviously mapped to -1, however W_0 is not analytic at this point. For the branches W_{-1} and W_1 there are two branch points $-e^{-1}$ and 0. For all other branches the only branch point is 0.

2.2.2 The Principal Branch

The $b = 0$ branch, also called the *principal branch* features a lot in this investigation. It is important as it is one of only two branches to contain real values and the only branch containing any part of the positive real axis. It is also the only branch to be defined for 0. Some important subsets in $\mathbb{C}$ are described below. The image of the interval $(-e^{-1}, \infty)$ is $(-e^{-1}, \infty)$; the image of $(-\infty, -e^{-1})$ is $\{-\mu \cot(\mu) + \mu i : 0 < \mu < \pi\}$; the image of the imaginary axis is given by: $\{\mu \tan(\mu) + \mu i : \mu \in \mathbb{R}\}$.

The Maclaurin series is known for $W_0(z)$ and is given in [7, p.339]. It can be derived using the Lagrange Inversion theorem. The series is

$$W_0(z) = \sum_{n=1}^{\infty} \frac{(-n)^{n-1}}{n!} z^n, \tag{2.5}$$

which has radius of convergence e^{-1}. More on calculating the values of $W_k(x)$ can be found in Section 2.3.

2.3 Methods for Calculating $W_b(z)$

Various methods have been developed for calculating the Lambert W function, some of these are applicable only for certain branches, or for certain values of z. We do not discuss these methods in great detail but a few are briefly described below. In the remainder of this thesis we assume a stable and accurate method for computing all scalar values (where defined) on all branches of W is available.

2.3.1 Taylor Series

The Taylor series for a general branch about $\alpha \in \mathbb{C}$ where $\alpha \neq -e^{-1}$ if $b = 0$ or -1 and $\alpha \neq 0$ if $b \neq 0$ is given by

$$W_b(z) = \sum_{n=0}^{\infty} \frac{W_b^{(n)}(\alpha)}{n!} (z - \alpha)^n.$$

For the principal branch we have a neat expression for the Taylor series about the origin (i.e. the Maclaurin Series) given by (2.5). The Taylor series for W does not have an infinite radius of convergence due to the infinite derivative at $-e^{-1}$ for branches 0 and -1 and the singularity at the origin for nonprincipal branches, however the Taylor series is one of the most important series in this thesis so we discuss the derivatives of W_b in some detail in Section 2.4.

2.3.2 Iterative Methods

Newton's Method

Newton's method is a first order root-finding method. Given an initial estimate[1] $w_0 \in \mathbb{C}$ for a root of the residual equation $R(w) = we^w - z$ the method converges to the root under suitable conditions. For W each iterate for $n = 1, 2, \ldots$ is given the recurrence

$$w_{n+1} = w_n - \frac{w_n e^{w_n} - z}{1 + w_n} e^{-w_n}. \tag{2.6}$$

This method is not normally used to compute W as iteration methods of a higher order yield greater accuracy and are almost as easy to implement. In particular Halley's method is used; the initial estimate discussed below can also be used in Newton's method.

Halley's Method

Halley's method is a second order root-finding method. Given an initial estimate w_0 for $W_b(x)$ Halley's method is given by the recurrence [3]

$$w_{j+1} = w_j - \frac{w_j e^{w_j} - z}{e^{w_j}(w_j + 1) - \frac{(w_j+2)(w_j e_{w_j} - z)}{2w_j + 2}}. \tag{2.7}$$

For an initial guess we might take the first few terms of a Taylor expansion, or if x is real and non-negative the expression,

$$w_0 = \begin{cases} 0.665(1 + 0.0195 \log(x+1)) \log(x+1) + 0.04 & \text{if } 0 \le x \le 500 \\ \log(x-4) - (1 - \frac{1}{\log(x)}) \log(\log(x)) & \text{if } x > 500 \end{cases},$$

given in [19].

We can find the values for different branches by ensuring that the initial guess is in the intended branch.

[1]Here the subscript indicates the position in a sequence rather than a branch.

2.3.3 Series about the Branch Point

A series, given in [7] which converges to $W_0(z)$ for $|z+e^{-1}|<e^{-1}$ is given by

$$W_0(z)=\sum_{k=0}^{\infty}\mu_k p^k, \tag{2.8}$$

where $p=\sqrt{2(ez+1)}$ and the μ_k are constants that satisfy the following recurrence relation

$$\mu_k=\frac{k-1}{k+1}\left(\frac{\mu_{k-2}}{2}+\frac{\alpha_{k-2}}{4}\right)-\frac{\alpha_k}{2}-\frac{\mu_{k-1}}{k+1},\text{ where }\alpha_k=\sum_{j=2}^{k-1}\mu_j\mu_{k+1-j},$$

where $\alpha_0=2$, $\alpha_1=-1$ and where $\mu_0=-1$ and $\mu_1=1$.

If we instead take $p=-\sqrt{2(ez+1)}$ then by using (2.8) we obtain a formula for $W_{-1}(z)$ when $\Im(z)\leq 0$ and for $W_1(z)$ when $\Im(z)>0$.

2.4 Differentiation

From hereon the nth derivative of $W_b(x)$ is denoted either as $\frac{d^nW_b}{dx^n}$ or $W_b^{(n)}(x)$. To differentiate W_b we investigate two methods: the first involves forming an expression for $W_b^{(n)}(x)$ in terms of $W_b(x)$ where x is a variable, and for the second we develop a recursion formula for $W_b^{(n)}(c)$ where c is a constant. To distinguish between the two we call the first implicit differentiation (Algorithm 2.4.2) and the second automatic differentiation (Algorithm 2.4.4).

2.4.1 Implicit Differentiation

By using the relation $x=W(x)e^{W(x)}$ we can obtain an implicit expression for $\frac{dW_b}{dx}$.

$$\begin{aligned}
\frac{d}{dx}(W_b(x)e^{W_b(x)}) &= \frac{d}{dx}x=1,\\
&= e^{W_b(x)}\frac{d}{dx}W_b(x)+W_b(x)\frac{d}{dx}e^{W_b(x)}\\
&= e^{W_b(x)}\frac{dW_b}{dx}+W_b(x)\frac{dW_b}{dx}e^{W_b(x)}\\
&= (e^{W_b(x)}+W_b(x)e^{W_b(x)})\frac{dW_b}{dx},\\
\frac{dW_b}{dx} &= \frac{1}{e^{W_b(x)}(1+W_b(x))}. \qquad (2.9)
\end{aligned}$$

This is valid for all branches and values except where $W_b(x) = -1$. This occurs only when $b = 0$ or -1 and $x = -e^{-1}$ however in practice as we are never able to store $-e^{-1}$ exactly, it is more important to note when $W_0(x)$ tends to -1. For the principal branch this happens as x tends to $-e^{-1}$ from any direction, for $b = -1$, $W_{-1}(x)$ tends to -1 only when x tends to $-e^{-1}$ from the upper half of the complex plane (including the real axis), and for $b = 1$, $W_1(x)$ tends to -1 when x tends to $-e^{-1}$ from the strictly lower half. As $x \approx -e^{-1}$ could be on either side of the real axis[2] in practice we should prepare for the derivative to be undefined for $b = -1$, 0 or 1. These are the only circumstances in which $W_b(x) \to -1$ however on all nonprincipal branches, $W_b(0)$ is not defined which implies $\frac{dW_b}{dx}$ is not defined at 0 either for these branches.

Differentiating (2.9) this gives us

$$\begin{aligned}
\frac{d^2W_b}{dx^2} &= \frac{d}{dx}\frac{-e^{W_b(x)}}{1+W_b(x)} \\
&= \frac{d}{dx}(-e^{W_b(x)})\frac{1}{(1+W_b(x))} + e^{-W_b(x)}\frac{d}{dx}\frac{1}{(1+W_b(x))} \\
&= -\frac{dW_b}{dx}\frac{1}{e^{W_b(x)}(1+W_b(x))} - \frac{dW_b}{dx}\frac{1}{e^{W_b(x)}(1+W_b(x))^2} \\
&= -\left(\frac{dW}{dx}\right)^2\frac{2+W(x)}{1+W(x)} \\
&= -\frac{e^{-2W(x)}(2+W(x))}{(1+W(x))^3}. \qquad (2.10)
\end{aligned}$$

We need an algorithm for calculating all derivatives, particularly when we consider Taylor expansions of W; fortunately an expression exists for all derivatives given in [7, (3.3)], they are given by

$$\frac{d^nW_b}{dx^n} = \frac{e^{-nW_b(x)}p_n(W_b(x))}{(1+W_b(x))^{2n-1}}, \qquad (2.11)$$

where $n \geq 1$ and p_n is a polynomial satisfying the recurrence relation

$$p_{n+1}(w) = -(nw+3n-1)p_n(w) + (1+w)\frac{dp_n}{dw}, \text{ with } p_1(w) = 1.$$

This is given by [7, (3.3)], and can be proved inductively using the product rule.

[2]If we restricted our computations to real arithmetic so that $\Im(x) = 0$ then we could be sure that $W_{-1}(z) \to -1$ and $W_1(z) \not\to -1$ for $x \to -e^{-1}$, however in this thesis we do not do this.

Implicit Differentiation Algorithm

For a practical algorithm we would like to get rid of $\frac{dp_n}{dw}$ so we substitute in the expressions for p_n and p'_n to get the coefficients for p_{n+1}. Then we can build an algorithm which calculates p_n recursively then uses (2.11) to calculate $W_b^{(n)}(x)$.

As p is a polynomial we have

$$\begin{aligned} p_n(w) &= \sum_{k=0}^{n-1} a_k w^k = a_0 + \cdots + a_{n-1}w^{n-1}, & (2.12)\\ p'_n(w) &= \sum_{k=0}^{n-2} (k+1)a_{k+1}w^k = a_1 + 2a_2 w + \cdots + (n-1)a_{n-1}w^{n-2}. & (2.13) \end{aligned}$$

Substituting the above expressions into the formula for p_{n+1} gives

$$\begin{aligned} p_{n+1}(w) &= -(nw+3n-1)\sum_{k=0}^{n-1} a_k w^k + (1+w)\sum_{k=0}^{n-2}(k+1)a_{k+1}w^k \\ &= \sum_{k=0}^{n-1}\left(-(nw+3n-1)a_k w^k\right) + \sum_{k=0}^{n-2}(1+w)(k+1)a_{k+1}w^k \\ &= \sum_{k=0}^{n-1}\big(-(nw+3n-1)a_k w^k + (1+w)(k+1)a_{k+1}w^k\big) \\ &= \sum_{k=0}^{n-1}\big(-nwa_k w^k - 3na_k w^k + a_k w^k + (k+1)a_{k+1}w^k + (k+1)wa_{k+1}w^k\big) \\ &= \sum_{k=0}^{n-1}\big(((1-3n)a_k + (k+1)a_{k+1})w^k + ((k+1)a_{k+1} - na_k)w^{k+1}\big) \\ &= \sum_{k=0}^{n-1}((1-3n)a_k + (k+1)a_{k+1})w^k + \sum_{k=1}^{n}(ka_k - na_{k-1})w^k \\ &= \sum_{k=1}^{n-1}((1-3n)a_k + (k+1)a_{k+1})w^k + \sum_{k=1}^{n-1}(ka_k - na_{k-1})w^k \\ &\quad + \ (na_n - na_{n-1})w^n + (1-3n)a_0 + a_1 \\ &= ((1-3n)a_0 + a_1) - a_{n-1}nw^n \\ &\quad + \sum_{k=1}^{n-1}((1-3n+k)a_k + (k+1)a_{k+1} - na_{k-1})w^k. \qquad (2.14) \end{aligned}$$

Note that all coefficients of terms of p_n higher than $n-1$ such as a_n can be thought of as equal to zero.

Example 2.4.1. To demonstrate this we derive $\frac{d^2W}{dx^2}$ from (2.14) and (2.11) and compare it to (2.10). First we need $p_2(w)$ so we put $n = 1$ to get

$$\begin{aligned} p_2(w) &= ((1-3)a_0 + a_1) + (a_1 - a_0)w \\ &= a_1 - 2a_0 + (a_1 - a_0)w. \end{aligned}$$

Now we substitute $a_0 = 1$, $a_1 = 0$ and $a_2 = 0$ to get

$$p_2(w) = -2 - w.$$

Putting this in (2.11) gives us

$$\frac{d^2W_b}{dx^2} = -\frac{e^{-2W_b(x)}(2 + W_b(x))}{(1 + W_b(x))^3},$$

which agrees with the result we derived manually in (2.10).

Using this we can now build an algorithm for differentiating the Lambert W function. The algorithm in pseudocode is given in Algorithm 2.4.2.

Algorithm 2.4.2 Calculates the n^{th} Derivative of W_b at x.

```
Require: Branch $b \in \mathbb{Z}$, derivative $n \in \mathbb{N}$, and point $z \in \mathbb{C}$.
 1: $p_1(w) = a_{1,0} = 1$
 2: for $j = 1 : n$ do
 3:     $a_{j+1,0} = (1 - 3j)a_{j,0} + a_{j,1}$
 4:     for $k = 1 : j - 1$ do
 5:         $a_{j+1,k} = (1 - 3j + k)a_{j,k} + (k + 1)a_{j,k+1} - ja_{j,k-1}$
 6:     end for
 7:     $a_{j+1,j} = -ja_{j,j-1}$
 8:     $a_{j+1,j+1} = 0$
 9:     $p_{j+1}(w) = a_{0,j} + a_{1,j}w + a_{2,j}w^2 + \cdots + a_{j+1,j}w^{j-1} + a_{j,j}w^j$
10: end for
11: return $\exp(-nW_b(x))p_n(W_b(x))/(1 + W_b(x))^{2n-1}$
```

The algorithm takes approximately $\frac{9}{2}n^2$ flops plus the cost of one evaluation of $W_b(x)$, where n is the desired derivative. In practice we want to avoid repeatedly computing the polynomials p_n every time we want a derivative so we split this algorithm into two functions, one which computes the polynomials and stores them at a cost of n^2 units of memory which need only be called once, and another which calculates the actual derivative by accessing the pre-computed polynomials. The cost of computing the nth polynomial (and all intermediary polynomials) is approximately

$\frac{9}{2}n^2$, and the cost of evaluating the nth derivative given p_n is approximately $2n$. See Appendices A.1 and A.2 for the MATLAB code.

Computing the coefficients is equivalent to performing various tridiagonal matrix multiplications. Writing the process in this way will prove useful for finding a priori error bounds. Although of course we should not form matrices during the actual computations. If we let $a_{n,k}$ be the coefficient of the kth term in p_n then as a vector the coefficients of p_{n+1} are given by

$$(a_{n+1,0}, a_{n+1,1}, \dots, a_{n+1,n-1}, a_{n+1,n})^T = A_n A_{n-1} \dots A_2 A_1 e, \tag{2.15}$$

where $A_j \in \mathbb{Z}^{n+1 \times n+1}$, e is the vector of size $n+1$ consisting of ones and for $j = 1, \dots, n$ we have

$$\begin{aligned} A_j &= \begin{pmatrix} H_j & 0 \\ 0 & 0 \end{pmatrix} \text{ with } H_k \in \mathbb{Z}^{j+1 \times j} \text{ and} \qquad (2.16) \\ H_j &= \begin{pmatrix} 1-3j & 1 & 0 & \cdots & 0 & 0 & 0 \\ -j & 2-3j & 2 & \cdots & 0 & 0 & 0 \\ 0 & -j & 3-3j & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -j & (j-1)-3j & j-1 \\ 0 & 0 & 0 & \cdots & 0 & -j & (j)-3j \\ 0 & 0 & 0 & \cdots & 0 & 0 & -j \end{pmatrix}. \end{aligned}$$

To be absolutely clear, the diagonal, superdiagonal and subdiagonal of A_j are given respectively by the vectors

$$\begin{aligned} \text{diagonal} &= \Big(\overbrace{1-3j,\ 1-3j+1,\ \dots,\ 1-3j+(j-1),\ 1-3j+j,}^{j}\ \overbrace{0,\ \dots,\ 0}^{n+1-j}\Big), \\ \text{superdiagonal} &= \Big(\overbrace{1,\ 2,\ \dots,\ j-2,\ j-1,}^{j-1}\ \overbrace{0,\ \dots,\ 0}^{n+1-j}\Big), \\ \text{subdiagonal} &= \Big(\overbrace{-j,\ \dots,\ -j,}^{j}\ \overbrace{0,\ \dots,\ 0}^{n-j}\Big). \end{aligned}$$

This form of Algorithm 2.4.2 allows us to apply results from the error analysis of matrix multiplication. We bear it in mind and return to it later.

2.4.2 Automatic Differentiation

An alternative method exists for computing the derivative from the Taylor series; this is called automatic differentiation or Taylor arithmetic. It is known as automatic differentiation due to the fact that algorithms exist which implement the method to find the Taylor series of general functions. We can however specialise the method to the Lambert W function, as the next theorem demonstrates.

Theorem 2.4.3. [4]. Let $c_0 = W_b(\alpha)$, $d_0 = 1/(1+c_0)$, $c_1 = c_0 d_0/\alpha$, and for $k = 1\colon n$

$$d_k = -\frac{1}{1+c_0}\sum_{j=1}^{k} d_{k-j}c_j, \tag{2.17}$$

$$c_{k+1} = -\frac{1}{\alpha(k+1)}(kc_k + d_k), \tag{2.18}$$

then $W_b(\alpha+t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 + \cdots$.

Proof. Begin with $W'(x) = \frac{W(x)}{x(1+W(x))}$ and define $t = x - \alpha$; then

$$W(x) = W(\alpha) + W'(\alpha)t + W''(\alpha)t^2/2! + \cdots.$$

Put $c_n = W^{(n)}(\alpha)/n!$ so that

$$W(x) = \sum_{k=0}^{\infty} c_k t^k. \tag{2.19}$$

We know that $c_0 = W(\alpha)$ and $c_1 = \frac{W(x)}{x(1+W(x))}$, we write c_1 as

$$\frac{1+W(x)-1}{x(1+W(x))} = \frac{1}{x}\left(1 - \frac{1}{1+W(x)}\right). \tag{2.20}$$

Differentiating (2.19) gives us $W'(x) = \sum_{k=0}^{\infty}(k+1)c_{k+1}t^k$. If we multiply (2.20) by x we get

$$\begin{aligned}
xW'(x) &= (t+\alpha)W'(x) \\
&= (t+\alpha)\sum_{k=0}^{\infty}(k+1)c_{k+1}t^k \\
&= \sum_{k=1}^{\infty} kc_k t^k + \alpha\sum_{k=0}^{\infty}(k+1)c_{k+1}t^k \\
&= \sum_{k=1}^{\infty} kc_k t^k + \alpha c_1 + \alpha\sum_{k=1}^{\infty}(k+1)c_{k+1}t^k
\end{aligned}$$

$$\begin{aligned} &= \alpha c_1 + \alpha \sum_{k=1}^{\infty} ((k+1)c_{k+1} + kc_k)t^k \\ &= 1 - \frac{1}{1+W(x)}. \end{aligned}$$

Now let $\frac{1}{1+W(x)} = \sum_{k=0}^{n} d_k t^k$ with $d_0 = \frac{1}{1+c_0}$ and note that

$$\begin{aligned} 1 &= \sum_{k=0}^{\infty} d_k t^k \left(1 + c_0 + \sum_{k=1}^{\infty} c_k t^k\right) \\ &= \sum_{k=0}^{\infty} d_k t^k + \sum_{k=0}^{\infty} d_k t^k \sum_{k=0}^{\infty} c_k t^k \\ &= \sum_{k=0}^{\infty} d_k t^k + \sum_{k=0}^{\infty} \sum_{N=0}^{k} c_N t^N d_{k-N} t^{k-N} \\ &= \sum_{k=0}^{\infty} (d_k + \sum_{N=0}^{k} c_N d_{k-N}) t^k; \end{aligned}$$

here we have used the Cauchy Product Formula [4]. As t is not constantly zero we deduce that

$$\begin{aligned} 0 &= d_k(1+c_0) + \sum_{N=1}^{k} d_{k-N} c_N, \text{ for } k \geq 1, \text{ so} \\ d_k &= \frac{-1}{1+c_0} \sum_{N=1}^{k} d_{k-N} c_N. \end{aligned}$$

Now $\alpha(k+1)c_{k+1} + kc_k = -d_k$, for $k \geq 1$; therefore

$$c_{k+1} = \frac{-1}{\alpha(k+1)}(d_k + kc_k).$$

□

This method computes the terms of the Taylor series of W_b about α. As the terms of the Taylor series are the derivatives of $W_b(\alpha)$ divided by a factorial we can use this to compute the derivatives. The drawback of using this method is that computing $W_b^{(n)}(\alpha)$ and $W_b^{(n)}(\beta)$ where $\alpha \neq \beta$ requires computing two different series whereas using Algorithm 2.4.2 only requires $W_b^{(n)}(x)$ to be computed once. We also have a singularity at 0 even on the principal branch, however we already have an expansion for W_0 about the origin given by (2.5).

Automatic Differentiation Algorithm

The pseudocode for the automatic differentiation method is given in Algorithm 2.4.4.

Algorithm 2.4.4 Calculates $W_b^{(n)}(z)$ using automatic differentiation.

```
Require: Branch $b \in \mathbb{Z}$, derivative $d \in \mathbb{N}$, and point $z \in \mathbb{C}$.
1: Set $c_0 = W_b(z)$, $c_1 = W_b(z)/(z(1+c_0))$ and $d_0 = 1/(1+c_0)$
2: for $j = 1 : n-1$ do
3:     $Q_j = d_{j-1}c_1 + d_{j-2}c_2 + \cdots + d_1 c_{j-1} + d_0 c_j$
4:     $d_j = -Q_j/(1+c_0)$
5:     $c_{j+1} = -(jc_j + d_j)/(z(j+1))$
6: end for
7: return $c_n n!$
```

Algorithm 2.4.4 costs approximately n^2 flops plus one evaluation of $W_b(z)$ to compute and evaluate $W_b^{(n)}$ so is more efficient to evaluate than Algorithm 2.4.2 and can easily be modified to produce the terms of the Taylor expansion of W_b about a given value (i.e. by removing the factorial at the end). The down side of this algorithm is that there exists a removable singularity at $z = 0$ even on the principal branch, so it may have trouble evaluating at or around $z = 0$.

2.4.3 Bounding $|W_b^{(n)}(z)|$

We now look at deriving a priori error bounds for the derivative algorithms, but before we can do this we must derive a bound for the magnitude of the derivative. For simplicity we write $w = W_b(z)$. Then

$$\begin{aligned}\left|\frac{d^n W_b}{dz^n}\right| &= \left|\frac{e^{-nw}p_n(w)}{(1+w)^{2n-1}}\right| \\ &= \left|\frac{(1+w)e^{-nw}}{(1+w)^{2n}}\right| |p_n(w)| \\ &= \left|\frac{(1+w)(e^{-w})^n}{((1+w)^2)^n}\right| |p_n(w)| \\ &= |1+w| \left|\frac{e^{-w}}{(1+w)^2}\right|^n |p_n(w)|.\end{aligned}$$

As n grows there are two factors to take into account, we first look at $\left|\frac{e^{-w}}{(1+w)^2}\right|^n$. It is easy to see that this quantity grows with n if $\left|\frac{e^{-w}}{(1+w)^2}\right| > 1$ and tends to zero as n

grows if $\left|\frac{e^{-w}}{(1+w)^2}\right| < 1$. If $\left|\frac{e^{-w}}{(1+w)^2}\right| = 1$ then clearly the quantity remains unchanged.

Unfortunately we cannot find such an expression for $|p_n(w)|$ however by using the formulation of the coefficients in (2.16) we can find a bound for it. Let $q_n = (a_{n,0}, a_{n,1}, \dots, a_{n,n-1})^T$ be the vector of coefficients and $v_n = (1, w, \dots, w^{n-1})^T$, then $p_n(w) = q_n^T v_n$. So $|p_n(w)| = |q_n^T v_n| \le |q_n|^T |v_n| = \|q_n\|_1 \|v_n\|_1$. In fact due to all elements of q_n having the same sign we can say $|q_n^T v_n| \le \|q_n\|_1 |\sum_{i=0}^n w^i|$. Then using the fact that $\sum_{i=0}^n w^i$ is a geometric series we immediately get $|\sum_{i=0}^n w^i| = \left|\frac{1-w^{n+1}}{1-w}\right|$. Now we use the definition of q_n in (2.15) to find a bound for $\|q_n\|_1$.

$$\begin{aligned} \|q_n\|_1 &\le \|A_n \dots A_1\|_1 \|e_1\|_1 && (2.21) \\ &\le \|A_n\|_1 \dots \|A_1\|_1. && (2.22) \end{aligned}$$

Now $\|A_r\|_1 = \max_{1\le m \le n+1} \sum_{k=1}^{n+1} |a_{km}|$ so is equal to $\|H_r\|_1 = \max_{1 \le m \le r} \sum_{k=1}^{r+1} |a_{km}|$. For any $m = 1{:}\,r$ we get $\sum_{k=1}^{r+1} |a_{km}| = |m-1| + |1-3r+m| + |-r| = 4r-2$. So clearly $\|A_r\|_1 \le 4r-2$ and hence $\|A_n \dots A_1\|_1 \le 4^n n!$ and

$$|p_n(w)| \le 4^n n! \left|\frac{1-w^{n+1}}{1-w}\right|. \tag{2.23}$$

For a lower bound we make use of the observation that all coefficients of p_n are of the same sign, so we can safely say $|p_n(w)| \ge \min_{i=1:n-1} |a_{n,i}| |\sum_{i=0}^n w^i|$. It is quite easy to see directly from Algorithm 2.4.2 that the absolute value of each coefficient of p_{j+1} must be greater than or equal to $j!$. We can prove this by induction, assuming each $a_{j,k} \le (j-1)!$ for all $k = 1{:}\,j$. So we conclude

$$|p_n(w)| \ge (n-1)! \left|\frac{1-w^{n+1}}{1-w}\right|. \tag{2.24}$$

Putting the bounds into the expression for $|W_b^{(n)}(z)|$ gives an upper and lower bound.

$$\begin{aligned} |W_b^{(n)}(z)| &\le n! \frac{|1+w|}{|1-w|} \left|\frac{4e^{-w}}{(1+w)^2}\right|^n |1-w^n| && (2.25) \\ |W_b^{(n)}(z)| &\ge (n-1)! \frac{|1+w|}{|1-w|} \left|\frac{e^{-w}}{(1+w)^2}\right|^n |1-w^n|. && (2.26) \end{aligned}$$

For simplicity we refer to $\frac{1+w}{1-w}\left(\frac{4e^{-w}}{(1+w)^2}\right)^n (1-w^n)$ as $\Upsilon_n(w)$ (note that the lower bound would be written $4^{-n}(n-1)!\Upsilon_n(w)$). Clearly both these quantity tend to infinity as n

does due to the presence of the factorial $(n-1)!$; for large enough n this overwhelms any other factor, so even if $\left|\frac{4e^{-w}}{(1+w)^2}\right| \ll 1$ the factorial eventually dominates. However if $|4e^{-w}/(1+w)^2|$ is small (which occurs when $|w|$ is large and $\Re(w) > 0$) then for small n subsequent derivatives might initially decay.

Comparison with the Derivatives of the Logarithm

The derivatives of the principal branch of the logarithm function are given by

$$\log^{(n)}(z) = \frac{(n-1)!}{(-z)^n},$$

for $n \geq 1$ which when written as $\log^{(n)}(z) = e^{-nL}[(-1)^n(n-1)!]$ where $L = \log(z)$ looks more like the form of (2.11). In this case the equivalent polynomial factor is just the constant $(-1)^n(n-1)!$ for each n. Given this we have an explicit expression for $|\log^{(n)}(z)| = (e^{-L})^n(n-1)!$. Unfortunately the author does not know of any explicit expression for $|p_n(w)|$ so we can only bound the magnitude of the derivatives of $W_b(z)$.

2.4.4 Error Analysis

Conditioning of $W_b^{(n)}$

The (relative) condition number of $W_b^{(n)}(z)$ is $c_{W^{(n)}}(z) = \left|\frac{zW_b^{(n+1)}(z)}{W_b^{(n)}(z)}\right|$ which expands and simplifies to become

$$\begin{aligned}
c_{W^{(n)}}(z) &= \left| z\frac{p_{n+1}}{p_n}\frac{e^{-(n+1)w}/(1+w)^{2n+1}}{e^{-nw}/(1+w)^{2n-1}} \right| \\
&= \left| ze^{-w}(1+w)^{-2}\frac{p_{n+1}}{p_n} \right| \\
&= \left| w(1+w)^{-2}\frac{p_{n+1}}{p_n} \right| \\
&= \left| -w(1+w)^{-2}(nw+3n-1) + (1+w)^{-1}w\frac{p_n'}{p_n} \right| \\
&\leq \frac{1}{|1+w|}\left(\frac{|nw^2+3wn-w|}{|1+w|} + c_{p_n}(w) \right)
\end{aligned}$$

The condition number is invariably large when $w \approx -1$. Experimental evidence shows that c_{p_n} tends to a moderate constant as n goes to infinity for w not too close to the origin, so if $w \to \infty$ then $c_{W^{(n)}}(z) \to n$.

An a priori error bound for the Implicit Differentiation algorithm

Theorem 2.4.5. Suppose $\mathrm{fl}(W_b^{(n)}(x)) = W_b^{(n)}(x) + \Delta W_b^{(n)}(x)$, where $\mathrm{fl}(W_b^{(n)}(x))$ is computed value of x, then

$$|\Delta W_b^{(n)}(x)| \leq \epsilon n |\Upsilon_n(w)| n!. \tag{2.27}$$

Proof. Note that this theorem only takes into account the error in computing the polynomials. Recall that $p_n(w) = q_n^T v_n$ where q_n and v_n are as in Section 2.4.3. Recall also $q_n = A_n \dots A_1 e$. By Corollary 1.4.3 for any consistent norm the error term in computing $\|q_n\|$ is bounded above by $\epsilon n \|A_1\|\|A_2\| \dots \|A_n\|$. Then we have $|\,\mathrm{fl}(p_n(w)) - p_n(w)| \leq \epsilon n \|A_1\|\|A_2\| \dots \|A_n\|$ as each $\|A_k\| \geq 1$ which gives us

$$|\Delta W_b^{(n)}(x)| \leq \epsilon n \frac{|1+w|}{|1-w|} \left| \frac{4e^{-w}}{(1+w)^2} \right|^n |1 - w^n| n! = \epsilon n |\Upsilon_n(w)| n!.$$

$\square$

Corollary 2.4.6. The relative error is bounded by $\left| \frac{\Delta W_b^{(n)}(x)}{W_b^{(n)}(x)} \right| \leq \epsilon 4^n n^2$.

Proof. Divide (2.27) by (2.26). $\square$

The author acknowledges several problems with this bound, for one thing it does not take into account rounding errors accrued from evaluating the polynomials $p_n(w)$, although it is well known that the standard methods are backwards stable (See [15, Sec. 5.1]). A more pressing problem is that the relative error bound becomes exponentially large even for moderate n. However we must remember that this is an error *bound* and not an error *estimate* and in all practical observations the author has made, the actual relative error[3] has been much smaller than the bound would suggest. For this reason the author believes it reasonable to accept that Algorithm 2.4.2 is a stable algorithm and hopes much tighter a priori error bounds can be found in the future to verify this.

[3]Estimated by comparing to symbolic differentiation.

An a priori error bounds for the Automatic Differentiation algorithm

In Algorithm 2.4.4 it is reasonable to assume the main source of rounding error comes from computing $Q_n = \sum_{j=1}^{n} d_{n-j}c_j$, which is an inner product. Let ΔQ_n be the error for the computed value of Q_n, by [15, (3.5)]

$$|\Delta Q_n| \leq \gamma_n \sum_{k=1}^{n} |d_{n-k}||c_k|. \tag{2.28}$$

From (2.25) we know that $c_n \leq |\Upsilon_n(c_0)|$ and we can easily derive a bound for d_n using $c_{n+1} = \frac{-1}{\alpha(n+1)}(d_n + nc_n)$. Rearranging this gives $d_n = -c_{n+1}\alpha(n+1) - nc_n$ so

$$\begin{aligned} |d_n| &\leq |\Upsilon_{n+1}(c_0)||\alpha|(n+1) + n|\Upsilon_n(c_0)| \\ &= |\Upsilon_n(c_0)|\left(4\frac{|c_0|}{|1+c_0|^2}\frac{|1-c_0^{n+1}|}{|1-c_0^n|}(n+1) + n\right). \end{aligned}$$

Substituting these into (2.28) gives

$$|\Delta Q_n| \leq \gamma_n \sum_{k=1}^{n} |\Upsilon_{n-k}(c_0)\Upsilon_k(c_0)| \left(4\frac{|c_0|}{|1+c_0|^2}\frac{|1-c_0^{n-k+1}|}{|1-c_0^{n-k}|}(n-k+1) + n - k\right).$$

To reduce clutter we abbreviate $\Xi_k^n(c_0) = 4\frac{|c_0|}{|1+c_0|^2}\frac{|1-c_0^{n-k+1}|}{|1-c_0^{n-k}|}(n-k+1) + n - k$.

$$\begin{aligned} |\Delta Q_n| &\leq \gamma_n|\Upsilon_n(c_0)|\sum_{k=1}^{n}\frac{|1+c_0|}{|1-c_0|}\frac{|1-c_0^k|}{|1-c_0^n|}\Xi_k^j(c_0) \\ &= \gamma_n|\Upsilon_n(c_0)|\frac{|1+c_0|}{|1-c_0|}\sum_{k=1}^{n}\frac{|1-c_0^k|}{|1-c_0^n|}\Xi_k^j(c_0). \end{aligned}$$

As $d_n = -Q_n/(1+c_0)$ dividing $|\Delta Q_n|$ by a factor of $|1+c_0|$ gives a bound for the error Δd_n of d_n. Now we look at $|c_{n+1}|$, but first we expand (2.18) recursively to get an expression for c_{n+1} in terms of d_i for $i = 0\colon n$ only.

$$\begin{aligned} c_{n+1} &= \frac{1}{\alpha(n+1)}(nc_n + d_n) \\ &= \frac{n}{\alpha(n+1)}c_n + \frac{1}{\alpha(n+1)}d_n \\ &= \frac{n-1}{\alpha^2(n+1)}c_{n-1} + \frac{1}{\alpha^2(n+1)}d_{n-1} + \frac{1}{\alpha(n+1)}d_n \\ &= \frac{n-2}{\alpha^3(n+1)}c_{n-2} + \frac{1}{\alpha^3(n+1)}d_{n-2} + \frac{1}{\alpha^2(n+1)}d_{n-1} + \frac{1}{\alpha(n+1)}d_n \\ &= \frac{1}{n+1}\sum_{k=0}^{n}\frac{1}{\alpha^{n-k+1}}d_k, \end{aligned}$$

as this is linear in d_i for $i = 0\colon n$ we can replace d_i with the error term Δd_i to get the error term Δc_{n+1}.

$$\begin{aligned}|\Delta c_{n+1}| &\le \frac{1}{n+1}\sum_{k=0}^{n}\frac{1}{|\alpha|^{n-k+1}}\left(\gamma_k|\Upsilon_k(c_0)|\frac{1}{|1-c_0|}\sum_{j=1}^{n}\frac{|1-c_0^j|}{|1-c_0^k|}\Xi_j^k(c_0)\right)\\ &\le \frac{1}{n+1}\frac{1}{|1-c_0||\alpha|^{n+1}}\sum_{k=0}^{n}\gamma_k|\alpha|^k|\Upsilon_k(c_0)|\sum_{j=1}^{n}\frac{|1-c_0^j|}{|1-c_0^k|}\Xi_j^k(c_0)\end{aligned}$$

By (2.26) the lower bound on c_{n+1} is

$$\begin{aligned}c_{n+1} &= |W_b^{(n+1)}(\alpha)|/(n+1)!\\ &\ge 4^{-(n+1)}|\Upsilon_{n+1}(c_0)|/(n+1),\end{aligned}$$

hence the relative error for c_n is bounded by

$$\frac{|\Delta c_n|}{|c_n|} \le \frac{4^n}{|1-c_0||\alpha|^n}\sum_{j=0}^{n-1}\gamma_j|\alpha|^j\frac{|\Upsilon_j(c_0)|}{|\Upsilon_n(c_0)|}\sum_{k=1}^{j}\frac{|1-c_0^k|}{|1-c_0^j|}\Xi_k^j(c_0). \tag{2.29}$$

This relative error bound also applies to the derivative when calculated by automatic differentiation as the factor of $n!$ is cancelled out. As in Corollary 2.4.6 this bound suffers the problem of exponential growth which is not observed in the error in practice. As with the implicit differentiation algorithm the author hopes a tighter bound may be found in the future.

2.5 Geometry

2.5.1 Singularities

The derivatives $W_b^{(n)}$ for $n \ge 1$ inherit the singularities in W_b but are not limited to them. All branches except $b = 0$ have a singularity at 0 which appears in $W_b^{(n)}$ also, as $W_0(0) = 0$ it follows that $W_0^{(n)}$ has no singularity at 0.

For $b = -1$, 0 or 1 there is also a singularity at $-e^{-1}$ in $W_b^{(n)}$ for $n \ge 1$. Strictly speaking this is true only for $b = 0$ and -1 but in practical computations this can occur for $b = 1$ also. The singularity arises from the form of the derivative (2.11). The singularity is a pole of order $2n-1$, this is the exact order because $p_n(-e^{-1}) \ne 0$

due to the transcendence of e. It is also notable that $W_b(x)$ is not analytic at $-e^{-1}$ for $b = -1, 0, 1$ although it is defined there. The singularity does not occur for any other branches.

2.5.2 Discontinuities

Recall that the derivative for any branch of W (2.11), is

$$W_b^{(n)}(z) = \frac{e^{-nW_b(z)} p_n(W_b(z))}{(1 + W_b(z))^{2n-1}},$$

for $n > 1$ and where $p_n(W_b(z))$ is a polynomial.

The branch cut made in the domain of W_0 means that there is a discontinuity across the interval $(-\infty, -e^{-1})$. As this is the only source of discontinuity in the expression for the derivatives it therefore follows that there is a discontinuity across $(-\infty, -e^{-1})$ in $W_0^{(n)}$. In all other branches the discontinuity lies in $(-\infty, 0)$, so similarly there is a discontinuity across $(-\infty, 0)$ in $W_b^{(n)}$.

2.5.3 Visualising $W_b^{(n)}$

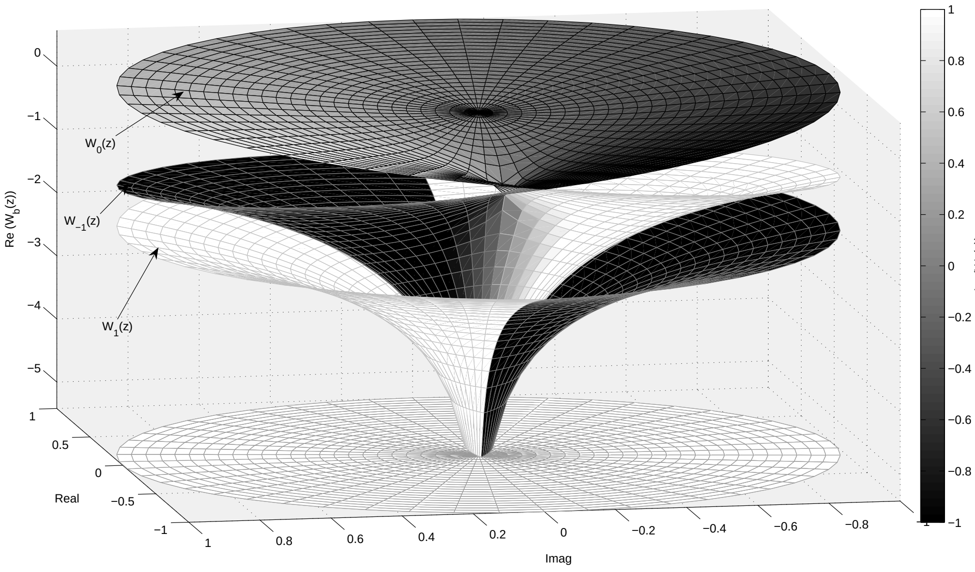

Figure 2.3: Surface plots for the three branches: $W_{-1}(z)$, $W_0(z)$, and $W_1(z)$. The vertical axis represents $\Re(W_b(z))$ and the colour axis $\Im(W_b(z))$. Note the singularity at the origin in $W_{-1}(0)$ and $W_1(0)$ but not in $W_0(0)$.

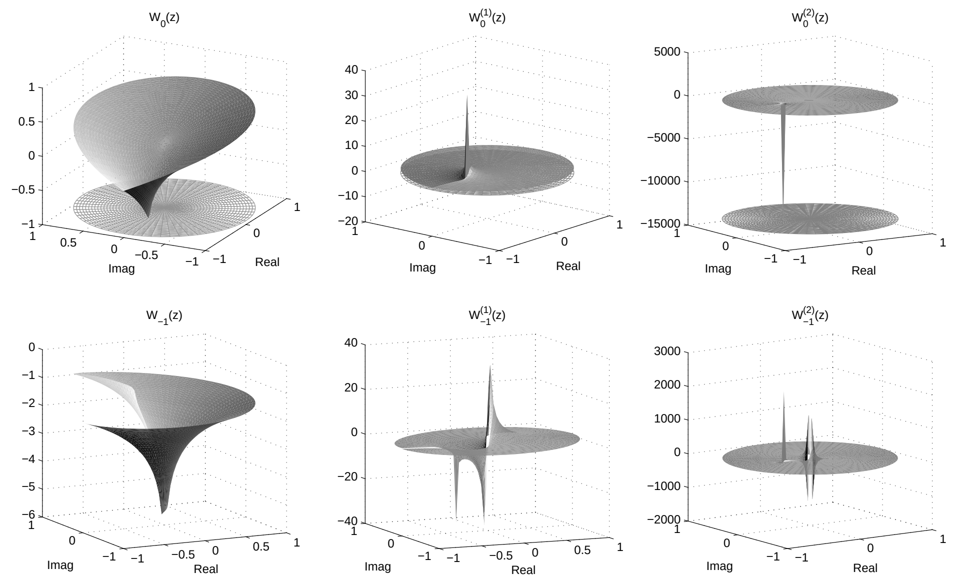

Figure 2.4: Surface plots for the the principal and $b = -1$ branch and their first and second derivatives. The vertical axis corresponds to $\Re(W_b^{(n)})$ and the colour of each face corresponds to $\Im(W_b^{(n)})$ (the colour tends to white as $\Im(W_b^{(n)}) \to \infty$ and tends to black as $\Im(W_b^{(n)}) \to -\infty$).

In Figure 2.4 surface plots are given for two of the branches and the first two of their derivatives. We can see quite vividly how at the branch points the derivatives go to infinity.

2.5.4 Symmetries

Lemma 2.5.1. $W_k(z)$ has near-conjugate symmetry, that is for $z \in \mathbb{C}^{n\times n}$ such that z is not in the branch cut of W_b, $W_k(\overline{z}) = \overline{W_{-k}(z)}$.

Proof. See [7, Sec. 4]. □

Theorem 2.5.2. All derivatives of W exhibit near-conjugate symmetry when not on the branch cuts.

Proof. Let $z \in \mathbb{C}^{n\times n}$ such that z is not in the branch cut of W_b.

$$\begin{aligned} W_b^{(n)}(\overline{z}) &= \frac{e^{-nW_b(\overline{z})}p_n(W_b(\overline{z}))}{(1+W_b(\overline{z}))^{2n-1}} \\ &= \frac{e^{-n\overline{W_{-b}(z)}}p_n(\overline{W_{-b}(z)})}{(\overline{1+W_{-b}(z)})^{2n-1}} \end{aligned}$$

$$
\begin{aligned}
&= \frac{\overline{e^{-nW_{-b}(z)}}\;\overline{p_n(W_{-b}(z))}}{\overline{(1+W_{-b}(z))^{2n-1}}} \\
&= \overline{W_{-b}^{(n)}(z)}.
\end{aligned}
$$

This makes use of the fact that for a function f holomorphic (continuously differentiable) at z whose restriction to the real line is real valued (such as real polynomials, exponentials and reciprocals) then $f(\overline{z}) = \overline{f(z)}$. $\square$

Chapter 3

Applications

3.1 Delay Differential Equations

Some physical systems can be modelled by functions which satisfy the linear time-delayed differential relation

$$y'(t) = ay(t - s), \tag{3.1}$$

where s and a are constants. We can solve this system explicitly in terms of the Lambert W function by a simple argument. If we assume the solution y is of the form $y(t) = e^{ct}$, then $y'(t) = ce^{ct}$. Substituting these into (3.1) gives us $ce^{ct} = ae^{c(t-s)} = ae^{ct}e^{-cs}$, rearranging this gives $c = ae^{-cs}$ so $ce^{cs} = a$ and $(cs)e^{(cs)} = as$. Here we introduce W and get $cs = W_b(as)$, so $c = W_b(as)/s$. Finally this gives us an explicit formulation for y.

$$y(t) = \exp\left(W_b(as)\frac{t}{s}\right). \tag{3.2}$$

Clearly we see that a solution to (3.1) always exists and moreover that it is not unique, we get an infinite number of solutions by varying b. In fact due to the linearity of (3.2) it follows that

$$y(t) = \sum_{k=-\infty}^{\infty} c_k e^{W_k(as)t/s}, \tag{3.3}$$

is also a solution wherever the sum makes sense [7].

Matrix Case

Suppose we wish to find the matrix function $Y : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ which satisfies

$$Y'(t) = A_d Y(t-s), \tag{3.4}$$

where A_d is nonsingular. As in the scalar case we proceed by assuming the function Y is of the form: $Y(t) = e^{tC}$. Again we derive $Y'(t) = Ce^{tC}$ and substitute them into (3.4) to get

$$\begin{aligned} Ce^{tC} &= A_d e^{(t-s)C} \\ &= A_d e^{-sC} e^{tC}, \\ C &= A_d e^{-sC}, \\ sCe^{sC} &= sA_d, \\ sC &= W_b(sA_d). \end{aligned}$$

So

$$Y(t) = e^{W_b(sA_d)t/s}. \tag{3.5}$$

Again we see that (3.4) has infinitely many solutions, and that

$$Y(t) = \sum_{k=-\infty}^{\infty} C_k e^{W_k(A_d s)t/s}, \tag{3.6}$$

is a solution where $C_k \in \mathbb{C}^{n\times n}$ and as long as the sum makes sense.

More General Systems

In [22] the solution to a more general system of linear time-delayed differential equations is given. Let

$$\begin{aligned} y'(t) &= Ay(t) + A_d y(t-s) + Bu(t) \quad &&\text{for} \quad t > 0, \\ y(t) &= g(t) &&\text{for} \quad -s \le t < 0, \\ y(0) &= y_0, \end{aligned}$$

where $A, A_d \in \mathbb{C}^{n\times n}$, $B \in \mathbb{C}^{n\times r}$, $u\colon \mathbb{R} \to \mathbb{C}^r$ is a vector function and $g\colon \mathbb{R} \to \mathbb{C}^{n\times n}$ is the pre-shape function. Then

$$y(t) = \sum_{k=-\infty}^{\infty} e^{S_k t} C_k^I + \int_0^t \sum_{k=-\infty}^{\infty} e^{S_k(t-\xi)} C_k^N Bu(\xi)d\xi,$$

where

$$S_k = W_k(A_d s Q_k)/s + A. \tag{3.7}$$

The constants $C_k^I, C_k^N, Q_k \in \mathbb{C}^{n\times n}$ and C_k^I depends on A, A_d, s and $g(t)$, C_k^N depends on A, A_d, s but not $g(t)$ and Q_k can be computed from the relation

$$W_i(A_d s Q_i)e^{W_i(A_d s Q_i)+As} = A_d s. \tag{3.8}$$

In the homogeneous case where $u(t) = 0$ the expression simplifies somewhat to

$$y(t) = \sum_{k=-\infty}^{\infty} e^{S_k t} C_k^I, \tag{3.9}$$

where S_k is the same as in (3.7). If A is the zero matrix, we get (3.4) and by (3.8) we can see that the $Q_k = I_n$ and by (3.7) we get $S_k = W_k(A_d s)$ so the solution reduces to (3.6).

3.2 Machine Chatter

One recent physical application of the Lambert W function which involve the matrix case is in the modelling of chatter in a cutting machine. In [21] the equation which governs machine chatter is

$$\frac{1}{\omega_n^2}\ddot{x}(t) + \frac{2\zeta}{\omega_n}\dot{x}(t) + x(t) = -\frac{k_c}{k_m}(x(t) - x(t-T)), \tag{3.10}$$

where $x(t)$ is the general coordinate of the tool edge position, k_m is the structural stiffness, k_c is the cutting coefficient, ω_n is the natural frequency of the undampened free oscillating system, ζ is the relative dampening factor and T is the time the machine take to make one revolution.

Using matrix/vector notation we can write this as

$$\begin{aligned}
0 &= \dot{\mathbf{x}}(t) + A\mathbf{x}(t) + A_d(t-T), \text{ where} \\
\mathbf{x} &= (x, \dot{x})^T \\
A &= -\begin{pmatrix} 0 & 1 \\ -\left(1+\frac{k_c}{k_m}\right)\omega_n^2 & -2\zeta\omega_n \end{pmatrix} \\
A_d &= -\begin{pmatrix} 0 & 0 \\ \frac{k_c}{k_m}\omega_n^2 & 0 \end{pmatrix}
\end{aligned}$$

Now we compute $S_k = W_k(A_d T Q_k)/T + A$ via W as a matrix function, where Q_k are matrices computed using (3.8). As this is a homogeneous linear time-delay equation we use (3.9) to get the solution computing the coefficients from a given pre-shape function.

3.3 Jet Plane Fuel Consumption

The following equations (See [2, pp. 312–323]) describe the endurance E_t (how long the plane can fly for given a certain amount of fuel) and the range R (how far the plane can travel given a certain amount of fuel) of a jet plane. We wish to find the thrust specific fuel consumption c_t (i.e. the rate of fuel consumption[1] per pound of thrust) and the weight of the fuel $w_0 - w_1$ where w_0 is the initial weight of the plane and w_1 is the final weight of the plane. The equations to describe this are

$$\begin{aligned} E_t &= \frac{C_L}{c_t C_D} \log\left(\frac{w_0}{w_1}\right), && (3.11) \\ R &= \frac{2}{c_t C_D}\left(\frac{2C_L}{\rho S}\right)^{1/2} (w_0^{1/2} - w_1^{1/2}), && (3.12) \end{aligned}$$

where C_L and C_D are the lift and drag coefficients respectively, ρ is the ambient air density and S is the area of horizontal projection of the wings. The equations can be simplified by grouping and nondimensionalising the parameters. We put

$$\begin{aligned} w &= \frac{w_0}{w_1}, && (3.13) \\ c &= E_t c_t \frac{C_D}{C_L}, && (3.14) \\ A &= -\frac{\sqrt{2}E_t}{R}\left(\frac{w_0}{\rho S C_L}\right)^{1/2}. && (3.15) \end{aligned}$$

The equations then become

$$\begin{aligned} c &= -\log(w), && (3.16) \\ 1 &= 2A\frac{1-\sqrt{w}}{\log(w)}. && (3.17) \end{aligned}$$

[1]The rate of fuel consumption is measured in pounds-per-hour rather than gallons-per-hour, as the energy in fuel depends on the mass rather than capacity.

As the right hand side of (3.17) is a strictly increasing function of w, it has one real solution if $A < 0$. In this case we get the following

$$w = \begin{cases} A^{-2}W_0^2(Ae^A), & \text{if } A \in (-\infty, -1], \\ A^{-2}W_{-1}^2(Ae^A), & \text{if } A \in [-1, 0). \end{cases} \tag{3.18}$$

Now that we know w we can solve for the specific fuel consumption by

$$c_t = -\frac{\log(w)C_L}{E_t C_D}, \tag{3.19}$$

and the weight by

$$w_0 - w_1 = w_0(1 - w). \tag{3.20}$$

3.4 Forces within Hydrogen Molecular Ions

A hydrogen molecular ion (H_2^+) is a molecule composed of two protons (p^+) and one electron (e^-). The Lambert W function was recently used to explain an anomaly in which the observed exchange forces between the protons did not match the predictions of the theoretical models.

By considering the one dimensional limit of the double-well Dirac system we can express the wave function as

$$-\frac{1}{2}\frac{d^2\psi}{dx^2} = q[\delta(x) + \lambda\delta(x - R)]\psi = E(\lambda)\psi. \tag{3.21}$$

The solution to this equation is a linear combination of atomic orbital solutions and is given by

$$\psi = Ae^{-c|x|} + Be^{-c|x-R|}. \tag{3.22}$$

We then enforce the condition that the function is continuous at each of the wells, $x = 0$ and $x = R$ which gives us the following transcendental equations for c

$$c_\pm = q[1 \pm e^{-c_\pm R}]$$

This can be rearranged so that

$$c_\pm = q + W(\pm qRe^{-qR})/R. \tag{3.23}$$

Scott et al. in [23] uses this formulation to show how exponentially subdominant terms in the true asymptotic expansion, which were ignored in previous numerical and asymptotic solutions, account for the discrepancy between observed results and the predictions.

3.5 Solution to Other Equations

There are a number of equations that can be rearranged so that their solution can be given in terms of W. For instance if $xb^x = a$ then $x = W(a\log(b))/\log(b)$, if $x^{x^a} = b$ then $x = \exp(W(a\log(b)/a))$ and if $a^x = x+b$ then $x = -b-W(-a^{-b}\log(a))/\log(a)$. Many other solutions were obtained by Lémeray in [10].

3.6 Tree Function

In graph theory, a tree is a graph in which any two vertices are connected by *exactly* one path; a tree is called *rooted* if there is a single vertex which has been designated the root. Given a graph with n vertices we are interested in a function which tells us how many distinct possible trees exist for this graph. If we let t_n be the number of rooted trees on n points then the exponential generating function of t_n is

$$T(x) = \sum_{k=0}^{\infty} t_n x^n/n!, \tag{3.24}$$

and satisfies the equation

$$T(x) = x + xT(x) + xT(x)^2/2! + \cdots = xe^{T(x)}, \tag{3.25}$$

So $T(x) = -W(-x)$, using this Pólya in [11] deduced that $t_n = n^{n-1}$ (cf. (2.5)).

3.7 Two Solar Wind Problems

In 2004 the Lambert W function was used in [8] to find the exact solution to two problems in astrophysics.

3.7.1 The Parker Solar Wind Problem

The first known as the Parker solar wind problem concerns the radial velocity of the solar wind, that is mass ejected from the sun into interplanetary space. In [8, (16)], under various assumptions the solution u to this is shown to satisfy

$$u^2 = \begin{cases} -a^2 W_0(-D(r)) & \text{if} \quad r \leq r_c, \\ -a^2 W_{-1}(-D(r)) & \text{if} \quad r \geq r_c, \end{cases} \tag{3.26}$$

where $D(r) = (r/r_c)^{-4} \exp(4(1 - r_c/r) - 1)$, a is an effective sound speed, r is the radial distance from the sun and r_c is known as the critical radius $r_c = GM_{☼}/(2a^2)$ where G is the gravitational constant and $M_{☼}$ is the mass of the sun. The choice of branch reflects the need to ensure the quantity $W(D(r))$ is real, so that the model gives physically sensible results.

3.7.2 The Mass Flux Problem

The second problem concerned the density of the solar wind at the base of the corona (the layer of the sun's atmosphere just above the chromosphere, the temperature of this is about 10^6K). The solution is actually a measure of the number of plasma particles per unit-volume. This is roughly equivalent to the density however as the plasma is predominantly composed of hydrogen. The particle density at the base of the corona is given by n_0 and in [8, (24)] is given as

$$n_0 = \frac{5u_0 k F_C}{\kappa A T^2 (1 + W_b(\omega))}, \tag{3.27}$$

where

$$\omega = -\exp\left(\frac{-25u_0^2 k^2}{\kappa A T}\right) - 1,$$

and u_0 is the velocity at the coronal base, F_C is the heat conductive flux term, k is Boltzmann's constant, κ is the Spitzer–Härm heat conductivity in ionised plasma, T is the temperature and $A = 1.9 \times 10^{-32} WmK^{1/2}$. The free terms in this expression are u_0, F_C and T.

The argument ω is always between $-e^{-1}$ and 0 hence we could obtain real values using W_0 or W_{-1} however as n_0 is a density we choose W_{-1} as W_0 in this problem would lead to a negative density.

3.8 Iterated Exponentiation

The iterated exponential or hyperpower function, also known as the power tower, is the function

$$h(x) = x^{x^{x^{x^{\cdot^{\cdot^{\cdot}}}}}}, \tag{3.28}$$

which can also be thought of as the limit of the recurrence $h_{n+1}(x) = x^{h_n(x)}$ for $n > 1$ and $h_1(x) = x$. If this limit exists then $h(x) = x^{h(x)}$ and in [7] the solution to this is given as

$$h(x) = -\frac{W_b(-\log(x))}{\log(x)}. \tag{3.29}$$

We can see this by substituting (3.28) into $h(x) = x^{h(x)}$.

Chapter 4

Matrix Functions

4.1 Introduction to Matrix Functions

The idea of extending the concept of a function from scalar values to matrices is a natural one and is useful in both theoretical and applied fields. In this thesis we restrict ourselves to looking at functions which take a square complex matrix of size $n \times n$ to a square complex matrix of size $n \times n$ where n is fixed. Applications of W as a matrix function have been discussed in Chapter 3. In this chapter we look briefly at matrix functions in general before focusing on W as a function of a matrix.

4.1.1 Conventions

In this thesis, unless stated otherwise we assume the following: $A \in \mathbb{C}^{n\times n}$, the set of eigenvalues is denoted $\Lambda(A) = \{\lambda_1, \dots, \lambda_n\}$, and A has m distinct eigenvalues. For each eigenvalue $\lambda_i \in \Lambda(A)$ we denote the index of λ_i as $n_i = \mathrm{idx}_A(\lambda_i)$.

4.2 Definition by Jordan Decomposition

Definition 4.2.1. Let A be a square matrix with distinct eigenvalues $\lambda_1, \dots, \lambda_m$, where λ_i has index $n_i = \mathrm{idx}_A(\lambda_i)$ $(i = 1 : m)$. A scalar function $f(x)$ is *defined on the spectrum of* A if $f^{(j)}(\lambda_i)$ is defined for $1 \le i \le m$ and $0 \le j \le n_i - 1$.

Definition 4.2.2. Given a single-valued scalar function $f : \Theta \to \mathbb{C}$, where $\Theta \subseteq \mathbb{C}$ defined on the spectrum of a matrix $A \in \mathbb{C}^{n\times n}$ we define the *matrix function* $f(A)$ by

$$f(A) = Z^{-1} f(J_A) Z, \tag{4.1}$$

where $J = ZAZ^{-1} = \operatorname{diag}\{J_1, \dots, J_s\}$ is the Jordan form of A (with $J_k \in \mathbb{C}^{m_k \times m_k}$ the Jordan blocks of A) and where $f(J) = \operatorname{diag}\{f(J_1), \dots, f(J_s)\}$ and where

$$\begin{aligned} f(J_k) &= f\begin{pmatrix} \lambda_k & 1 & \cdots & 0 & 0 \\ 0 & \lambda_k & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_k & 1 \\ 0 & 0 & \cdots & 0 & \lambda_k \end{pmatrix} \\ &= \begin{pmatrix} f(\lambda_k) & \frac{f'(\lambda_k)}{1!} & \cdots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} & \frac{f^{(m_k)}(\lambda_k)}{m_k!} \\ 0 & f(\lambda_k) & \cdots & \frac{f^{(m_k-2)}(\lambda_k)}{(m_k-2)!} & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & f(\lambda_k) & \frac{f'(\lambda_k)}{1!} \\ 0 & 0 & \cdots & 0 & f(\lambda_k) \end{pmatrix}. \end{aligned} \tag{4.2}$$

This definition does not require f to be continuous or even defined outside the spectrum of A. All that is required is for values of f and its derivatives (where appropriate) to be defined for the spectrum of A.

To understand why we define $f(A)$ this way let's look at the case where f is a function analytic at $\alpha \in \mathbb{C}^{n\times n}$ (so has a Taylor expansion about α) and the spectrum of A. Now substitute $A = ZJZ^{-1}$ into the Taylor expansion of f:

$$\begin{aligned} f(ZJZ^{-1}) &= \sum_{j=0}^{\infty} \frac{f^{(j)}(\alpha)}{j!}(ZJZ^{-1})^j - \alpha I \\ &= \sum_{j=0}^{\infty} \frac{f^{(j)}(\alpha)}{j!}(ZJZ^{-1} - \alpha Z Z^{-1})^j \\ &= Z\sum_{j=0}^{\infty} \frac{f^{(j)}(\alpha)}{j!}(J - \alpha I)^j Z^{-1} \\ &= Zf(J)Z^{-1}. \end{aligned}$$

As J is block diagonal we can apply the Taylor series to each block individually; hence $f(J) = \mathrm{diag}(f(J_1), \dots, f(J_s))$. Each Jordan block J_k can be written $J_k = \lambda_k I_{m_k} + N$ where N is zero save for ones along its superdiagonal and m_k is the size of J_k. As F is defined on the spectrum of A without loss of generality we can take $\alpha = \lambda_k$. Then

$$\begin{aligned} f(J_k) &= \sum_{j=0}^{\infty} \frac{f^{(j)}(\lambda_k)}{j!}(J - \lambda_k I_n)^j \\ &= \sum_{j=0}^{\infty} \frac{f^{(j)}(\lambda_k)}{j!} N^j. \end{aligned}$$

As N is nilpotent, repeated powering of N shifts the ones on the superdiagonal up to the next diagonal, in other words N^j has ones on its jth superdiagonal and zeroes everywhere else if $j < m_k$, and $N^j = 0$ if $j \geq m_k$. So the Taylor series is a finite sum with m_k terms $\frac{f^{(j)}(\lambda_k)}{j!} N^j, (j = 0 : m_k - 1)$. As N^j consists only of the jth diagonal the formula in Definition 4.2.2 follows.

The Jordan form of a matrix is not unique, therefore we need to determine whether or not Definition 4.2.2 depends on the Jordan form (i.e. whether Definition 4.2.2 is well-defined). For single-valued functions it is well defined but we defer proving it until we have introduced the multivalued case.

4.2.1 Multivalued Functions

Definition 4.2.2 assumes f to be single-valued but before we continue we must address the consequences of f being a multivalued function. If f is multivalued then $f(\lambda)$ could have more than one value and if λ is repeated we therefore have the chance of mixing different values of $f(\lambda)$. This leads to the definition of primary and nonprimary matrix functions.

Definition 4.2.3. If f is a multivalued function defined on the spectrum of A we define $f_{j_1 \dots j_m}(A)$ to be a *primary matrix function* of A if the same branch is used to evaluate Jordan blocks which are associated with the same eigenvalue and j_k denotes which branch of f is used to evaluate the kth eigenvalue[1]. Any other matrix function

[1] We are assuming that the function assigns each subscript to the correct eigenvalue (i.e. the one we intend). We could if necessary state explicitly which branch is used for each eigenvalue but for our purposes this notation is sufficient.

is called a *nonprimary matrix function.*

For single-valued functions, all matrix functions are vacuously primary functions. So from hereon we consider all functions to be potentially multivalued and distinguish only between primary and nonprimary matrix functions.

If we are using the same branch of f to evaluate all eigenvalues (so $j_1 = \cdots = j_m$) then $f_{j_1 \cdots j_m}(A)$ is abbreviated by $f_j(A)$ where j denotes the branch.

It is important that we know whether the choice of transform, i.e. Z, Z^{-1} affects the solution. For primary matrix functions the solution never depends on the choice of transform, however for nonprimary functions, the solution does depend on the choice of transform.

Theorem 4.2.4. Let $f_{j_1,\ldots,j_m}(A)$ be a primary matrix function of A, then $f_{j_1,\ldots,j_m}(A)$ does not depend on the Jordan form of A used. In other words if $A = Z_1 J_1 Z_1^{-1} = Z_2 J_2 Z_2^{-1}$ where J_1, J_2 are Jordan matrices and possibly $J_1 \neq J_2$ (i.e. the blocks are ordered differently) then $Z_1 f_{j_1,\ldots,j_m}(J_1) Z_1^{-1} = Z_2 f_{j_1,\ldots,j_m}(J_2) Z_2^{-1}$.

Proof. [16, Prob. 1.1]. □

4.3 Properties of Matrix Functions

There are several other ways to define a matrix function, and for practical computation the Jordan decomposition is hardly ever used. The virtue of the Jordan canonical form definition is that it is fairly straightforward to see and from it we can easily deduce many of the properties of matrix functions which may be useful in practical applications.

Theorem 4.3.1. The following additional properties hold for any primary matrix function $f(A)$.

1. $f(A^T) = f(A)^T$.

2. $f(ZAZ^{-1}) = Zf(A)Z^{-1}$.

3. If λ_k is an eigenvalue of A then $f_{j_k}(\lambda_k)$ is an eigenvalue of $f_{j_1 \cdots j_m}(A)$ where $1 \leq k \leq m$.

4. If v is an eigenvector of A then it is also an eigenvector of $f(A)$.

5. If A commutes with B then B commutes with $f(A)$.

6. If A is block triangular or block diagonal, then $f(A)$ has the same block structure.

7. $f(I_p \otimes A) = I_p \otimes f(A)$, where $\otimes$ is the Kronecker product.

8. $f(A \otimes I_p) = f(A) \otimes I_p$.

Proof. See [16, Thm. 1.15]. □

Theorem 4.3.2. If $f(A)$ is a primary matrix function, then $f(A)$ is a polynomial in A with scalar coefficients.

Proof. See [16, Rem. 1.8]. □

Note that in general the same cannot be said for nonprimary matrix functions. In such cases it is possible for $f(T)$ to be full where T is triangular, which cannot occur if $f(T)$ is a polynomial of T.

Theorem 4.3.3. Suppose $F = f(A)$ is a primary matrix function, then $FA = AF$, i.e. F and A commute.

Proof. As F is primary, it is a polynomial in A by Theorem 4.3.2 and as $AA^k = A^kA$ clearly it commutes with A. □

Theorem 4.3.4. Let $f(X)$ and $g(X)$ be two primary matrix functions defined on the spectrum of a matrix $A \in \mathbb{C}^{n\times n}$. Then $f(A) = g(A)$ if and only if $f^{(j)}(\lambda_i) = g^{(j)}(\lambda_i)$ for $1 \leq i \leq m$ and $0 \leq j \leq n_i - 1$.

Proof. See [16, Thm. 1.14]. □

Theorem 4.3.5. If f is $n-1$ times differentiable on an open subset $\Omega \subseteq \mathbb{C}$ then $f(A) = 0$ for all $A \in \mathbb{C}^{n\times n}$ with spectrum in Ω if and only if $f(A) = 0$ for all diagonalisable $A \in \mathbb{C}^{n\times n}$ with spectrum in Ω.

Proof. See [17, Thm. 6.2.27 (2)]. □

The above theorem implies that if we have a matrix identity of the form $f(A) = g(A)$ where f and g are $n-1$ times differentiable on the spectrum of A (for instance $\sin^2(A) = 1 - \cos^2(A)$) then we can write $f(A) - g(A) = h(A) = 0$. Then to prove the identity for all applicable A it suffices to consider only the case where A is diagonalisable.

4.4 The Lambert W Function

Recall from Chapter 2 that the Lambert W function is a multivalued function whose branches are parametrised by a single integer. The matrix function W is defined fundamentally as an inverse of the forward function $A = We^W$ and in [5] it is shown that Definition 4.2.2 is not sufficient to characterise all possible solutions. Corless et al. [5, Thm. 6] give an expression for all possible solutions: if $A = ZJZ^{-1} = Z\,\mathrm{diag}(J_1, \ldots, J_s)Z^{-1}$ and $-e^{-1} \notin \Lambda(A)$ then

$$W = ZU\,\mathrm{diag}(W_{b_1}(J_1), \ldots, W_{b_s}(J_s))U^{-1}Z^{-1}, \tag{4.3}$$

where U is an arbitrary nonsingular matrix which commutes with J. The different solutions are got by varying the integer values of $b_1, \ldots, b_s$ and the matrix U. In other words, given any Jordan decomposition of A (recall that the choice of Z affects nonprimary $W(A)$), all solutions can be obtained from (4.3). Note that Z depends on J, and U depends on J and Z.

This characterisation differs from Definition 4.2.2 in the similarity transform involving U and U^{-1} and that we can choose different branches for Jordan blocks even if they have the same eigenvalue. Note that if we force $b_j = b_k$ whenever J_j and J_k share an eigenvalue (i.e. make W a primary matrix function) then by Property 5 of Theorem 4.3.1 U commutes with $\mathrm{diag}(W_{b_1}(J_1), \ldots, W_{b_s}(J_s))$ so U and U^{-1} cancel out and we are left with something obtainable from Definition 4.2.2. This confirms that all primary solutions are obtainable from Definition 4.2.2 but not all nonprimary ones.

4.4.1 Standard Notation and Existence

In light of (4.3) we define three ways of expressing solutions to $A = We^W$. The first is for primary matrix functions, the second encompasses the most general way of expressing (4.3) and the third is used when the precise meaning is clear from the context.

- If we want $W(A)$ to be a primary function then we write $W_{b_1 \ldots b_m}(A)$ or $W_{\mathbf{b}}(A)$ where $\mathbf{b}$ is a vector (here there must be as many entries or subscript indices as there are distinct eigenvalues of A) or more simply $W_b(A)$ if we want to use only one branch.
- If we want $W(A)$ to refer to any possible solution then we write $W^{Z,U}_{b_1 \ldots b_s}(A)$ where Z is the Jordan transformation matrix and U is any nonsingular matrix which commutes with $J = Z^{-1}AZ$ and we have as many subscripts as Jordan blocks in A. We could also write $W^{Z}_{b_1 \ldots b_s}(A)$ where we combine Z and U. By varying Z we are able to obtain all solutions; however we may want to emphasise the different roles Z and U play, so for that reason we use $W^{Z,U}_{b_1 \ldots b_s}(A)$.
- If we want to refer to $W(A)$ irrespective of what kind of function it is, or when that is clear from the context we simply write $W(A)$.

Let $W_{b_1 \ldots b_m}(A)$ be primary where b_i is the branch used to evaluate Jordan blocks associated with the eigenvalue λ_i. There are two conditions in which $W_{b_1 \ldots b_m}(A)$ might not exist. If for some $i \leq m$ $\lambda_i = 0$ and $b_i \neq 0$ (i.e. the function is trying to evaluate a singular Jordan block on a nonprincipal branch) then $W_{b_1 \ldots b_m}(A)$ does not exist, if $b = 0$ however, there is no problem. If $\lambda_i = -e^{-1}$ appears in any Jordan block greater than size one, and $b_i = 0$ or -1 then $W_{b_1 \ldots b_m}(A)$ does not exist, one can see this from (4.2) in which $W_{b_i}^{(1)}(-e^{-1}) = \infty$ appears. Note that there does not appear to be any problems if $-e^{-1}$ only appears in Jordan blocks of size one however some theoretical results require that no eigenvalue equals $-e^{-1}$. Unless otherwise stated $W(A)$ refers to a primary matrix function such that for no $i = 1\colon m$ do we have $\lambda_i = 0$ and $b_i \neq 0$, or $\lambda_i = -e^{-1}$, $\mathrm{idx}(\lambda_i) > 1$ and $b_i = 0, -1$.

4.4.2 Properties of $W(A)$

The next theorem shows the notion of conjugate symmetry (cf. Theorem 2.5.2) extends to matrix functions.

Theorem 4.4.1.

$$W_{b_1\dots b_m}(A)^* = W_{-b_1\dots -b_m}(A^*),$$

where $W_{b_1\dots b_m}(A) \in \mathbb{C}^{n\times n}$ is a primary matrix function and no eigenvalue λ_i of A is in a branch cut of W_{b_i}.

Proof. As $W_{b_1\dots b_m}(A)^* = W_{b_1\dots b_m}(P^{-1}JP)^* = P^* W_{b_1\dots b_m}(J)^* P^{-*}$ it suffices to prove the theorem for the case where $A = J(\lambda)$ is a single Jordan block (and evaluated on a single branch).

$$W_b(J(\lambda))^* = \begin{pmatrix} \overline{W_b(\lambda)} & \overline{W_b^{(1)}(\lambda)} & \cdots & \overline{W_b^{(n)}(\lambda)} \\ 0 & \overline{W_b(\lambda)} & \cdots & \overline{W_b^{(n-1)}(\lambda)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \overline{W_b(\lambda)} \end{pmatrix}^T,$$

by Theorem 2.5.2 as λ is not in the branch cut of W_b we have

$$\begin{aligned} W_b(J(\lambda))^* &= \begin{pmatrix} W_{-b}(\overline{\lambda}) & W_{-b}^{(1)}(\overline{\lambda}) & \cdots & W_{-b}^{(n)}(\overline{\lambda}) \\ 0 & W_{-b}(\overline{\lambda}) & \cdots & W_{-b}^{(n-1)}(\overline{\lambda}) \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & W_{-b}(\overline{\lambda}) \end{pmatrix}^T \\ &= W_{-b}(J(\overline{\lambda}))^T = W_{-b}(J(\overline{\lambda})^T), \end{aligned}$$

by Property 1 of Theorem 4.3.1. Therefore

$$W_b(J(\lambda))^* = W_{-b}(J(\lambda)^*).$$

□

For $z_1, \dots, z_k \in \mathbb{C}$ the scalar Lambert W function satisfies the following equations

$$W_b(z_1) \dots W_b(z_k) = z_1 \dots z_k e^{-(W_b(z_1)+\dots+W_b(z_k))}. \tag{4.4}$$

This can be seen by noting that $W_b(z) = ze^{-W_b(z)}$ and that $e^{-W_b(z_1)}e^{-W_b(z_2)} = e^{-(W_b(z_1)+W_b(z_2))}$. Unfortunately (4.4) does not generalise to the matrix case as matrix multiplication is not commutative, although if we take all $Z_1, \dots, Z_k$ to be equal then we get the following result.

Theorem 4.4.2. Let $A \in \mathbb{C}^{n\times n}$

$$W_{\mathbf{b}}(A)^k = A^k e^{-kW_{\mathbf{b}}(A)}. \tag{4.5}$$

where $W_{\mathbf{b}}(A)$ is a primary matrix function.

Proof. By Theorem 4.3.5 we can take A to be diagonalisable. Let $D = ZAZ^{-1}$ be diagonal where Z is nonsingular, then $W_{\mathbf{b}}(A)^k - A^k e^{-kW_{\mathbf{b}}(A)} = Z^{-1}W_{\mathbf{b}}(D)^k Z - Z^{-1}D^k e^{-kW_{\mathbf{b}}(D)}Z = Z^{-1}(W_{\mathbf{b}}(D)^k - D^k e^{-kW_{\mathbf{b}}(D)})Z$ so the problem reduces to proving $W_{\mathbf{b}}(D)^k = D^k e^{-kW_{\mathbf{b}}(D)}$ which by applying Property 6 of Theorem 4.3.1 follows from the scalar case. □

This result is motivated by an attempt to adapt the inverse scaling and squaring method of the logarithm to the Lambert W function however this result does not appear to be of any help to us. The idea is to transform A into something for which W_b is easier to evaluate, whether such a transform exists for W is an open question.

4.4.3 Conditioning

Now we look at deriving and calculating information about the conditioning of the Lambert W function at the matrix we wish to evaluate. To do this we introduce the notion of the Fréchet derivative.

Definition 4.4.3. A matrix function f is *Fréchet differentiable* at X if there exists a linear mapping $L_f(X, E)$ such that $f(X + E) - f(X) - L_f(X, E) = O(\|E\|^2)$. If L_f exists then it is unique. (See [16, Prob. 3.3] for proof of uniqueness).

Theorem 4.4.4. If $A \in \mathbb{C}^{n\times n}$ has no eigenvalues on the closed negative real axis then W_b is Fréchet differentiable at A and the Fréchet derivative is continuous at A.

Proof. See [16, Thm. 3.8] using the facts that W_b is infinitely differentiable on $\mathcal{D} = \mathbb{C}^{n\times n} \setminus (-\infty, 0]$ and $\Lambda(A) \subset \mathcal{D}$. □

Corollary 4.4.5. If $A \in \mathbb{C}^{n\times n}$ has no eigenvalues on $(-\infty, -e^{-1}]$ then W_0 is Fréchet differentiable at A.

Proof. Same as above but uses the fact that W_0 is also infinitely differentiable on $(-e^{-1}, 0]$. □

The above results are sufficient conditions for the existence of the Fréchet derivative, but not necessary conditions, if A has eigenvalues on the negative axis then $L_{W_b}(A, E)$ may still exist.

Definition 4.4.6. The *absolute condition number* of a matrix function $f(A)$ is defined as

$$c_{\text{abs}}(f, X) = \lim_{t\to 0} \sup_{\|E\|\le t} \frac{\|f(X+E) - f(X)\|}{t},$$

and the *relative condition number* is defined as

$$c_f(X) = c_{\text{rel}}(f, X) = c_{\text{abs}}(f, X)\frac{\|X\|}{\|f(X)\|},$$

for some subordinate norm.

If f is Fréchet differentiable at A then (See Higham [16, Thm. 3.1]) we can write the absolute condition number as

$$c_{\text{abs}}(f, X) = \|L_f(X)\|,$$

where

$$\|L_f(X)\| = \max_{E\neq 0} \frac{\|L_f(X, E)\|}{\|E\|}.$$

Obtaining the condition number (or an indication of its magnitude, as is the only option in most cases) allows us to quantify how good we can expect our numerical results to be. In order to derive an expression for the Fréchet derivative of the Lambert W function we need the following lemma generalising some of the properties of the standard derivative.

Lemma 4.4.7. Let f and g be Fréchet differentiable at X then

1. (**Sum Rule**) $h_1(X) = \alpha f(X) + \beta g(X)$ is Fréchet differentiable at X and $L_{h_1}(X, E) = \alpha L_f(X, E) + \beta L_g(X, E)$ where $\alpha, \beta \in \mathbb{C}$,
2. (**Product Rule**) $h_2(X) = f(X)g(X)$ is Fréchet differentiable at the matrix X and $L_{h_2}(X, E) = L_f(X, E)g(X) + f(X)L_g(X, E)$,
3. (**Chain Rule**) $h_3(X) = f(g(X))$ is Fréchet differentiable at X and $L_{h_3}(X, E) = L_f(g(X), L_g(X, E))$,
4. (**Inverse Rule**) $f^{-1}(X)$ is Fréchet differentiable at X (assuming f^{-1} exists and is nonsingular at X) and $L_{f^{-1}}(X, E) = L_f^{-1}(f^{-1}(X), E)$. We can also rearrange this as $L_f(X, L_{f^{-1}}(f(X), E)) = E$.

Proof. [16, Thm. 3.2–3.5] □

Theorem 4.4.8. If X commutes with E then $L_f(X, E) = f'(X)E = Ef'(X)$.

Proof. [16, Prob. 3.8] □

The Fréchet derivative $L(X, E)$ is linear in E; because of this we can write $\operatorname{vec}(L(X, E)) = K(X)\operatorname{vec}(E)$ where $K(X) \in \mathbb{C}^{n^2 \times n^2}$ does not depend on E (see Higham [16, (3.17)]). The advantage of this form, called the *Kronecker form* is shown in the next lemma.

Lemma 4.4.9. If $\|\cdot\|_F$ is the Frobenius norm then

$$\|L(X)\|_F = \|K(X)\|_2.$$

Proof.

$$\begin{aligned}
\|L(X)\|_F &= \max_{E\neq 0} \frac{\|L(X,E)\|_F}{\|E\|_F} = \max_{E\neq 0} \frac{\|\operatorname{vec}(L(X,E))\|_2}{\|\operatorname{vec}(E)\|_2} \\
&= \max_{E\neq 0} \frac{\|K(X)\operatorname{vec}(E)\|_2}{\|\operatorname{vec}(E)\|_2} = \|K(X)\|_2.
\end{aligned}$$

□

So being able to compute the 2-norm of $K(X)$ gives us the absolute condition number in the Frobenius norm.

The Fréchet derivative of e^X is $L_{\exp}(X,E) = \int_0^1 e^{X(1-s)}Ee^{Xs}ds$ (see Higham [16, (10.15)]) and $L_X(X,E) = E$ is obvious from the definition. By using the product rule we get

$$L_{Xe^X}(X,E) = Ee^X + X\int_0^1 e^{X(1-s)}Ee^{Xs}ds;$$

and via the inverse rule (here $W_{\mathbf{b}}$ is a primary matrix function)

$$L_{W_{\mathbf{b}}}(X,E) = \left(Ee^{W_{\mathbf{b}}(X)} + W_{\mathbf{b}}(X)\int_0^1 e^{W_{\mathbf{b}}(X)(1-s)}Ee^{W_{\mathbf{b}}(X)s}ds\right)^{-1}. \qquad (4.6)$$

Let $Y = Xe^X$, we make use of the relation $L_{Xe^X}(X, L_{W_{\mathbf{b}}}(Xe^X,E)) = E$ to get

$$\begin{aligned} E &= L_{W_{\mathbf{b}}}(Xe^X,E)e^X + X\int_0^1 e^{X(1-s)}L_{W_{\mathbf{b}}}(Xe^X,E)e^{Xs}ds \\ &= L_{W_{\mathbf{b}}}(Y,E)e^{W_{\mathbf{b}}(Y)} + W_{\mathbf{b}}(Y)\int_0^1 e^{W_{\mathbf{b}}(Y)(1-s)}L_{W_{\mathbf{b}}}(Y,E)e^{W_{\mathbf{b}}(Y)s}ds. \end{aligned}$$

By (1.6) in Lemma 1.5.4, applying the vec operator gives

$$\begin{aligned} \operatorname{vec}(E) &= (e^{W_{\mathbf{b}}(Y^T)} \otimes I_n)\operatorname{vec}(L_{W_{\mathbf{b}}}(Y,E)) \\ &+ \int_0^1 \underbrace{\operatorname{vec}(W_{\mathbf{b}}(Y)e^{W_{\mathbf{b}}(Y)}e^{-sW_{\mathbf{b}}(Y)}L_{W_{\mathbf{b}}}(Y,E)e^{W_{\mathbf{b}}(Y)s})}_{(*)}\,ds. \end{aligned}$$

Now by (1.6) and (1.9) in Lemma 1.5.4

$$\begin{aligned} (*) &= \operatorname{vec}(W_{\mathbf{b}}(Y)e^{W_{\mathbf{b}}(Y)}e^{-sW_{\mathbf{b}}(Y)}L_{W_{\mathbf{b}}}(Y,E)e^{W_{\mathbf{b}}(Y)s}) \\ &= \operatorname{vec}(Ye^{-sW_{\mathbf{b}}(Y)}L_{W_{\mathbf{b}}}(Y,E)e^{W_{\mathbf{b}}(Y)s}) \\ &= ((e^{W_{\mathbf{b}}(Y)s})^T \otimes Ye^{-sW_{\mathbf{b}}(Y)})\operatorname{vec}(L_{W_{\mathbf{b}}}(Y,E)) \\ &= (I_n \otimes Y)((e^{W_{\mathbf{b}}(Y)s})^T \otimes e^{-sW_{\mathbf{b}}(Y)})\operatorname{vec}(L_{W_{\mathbf{b}}}(Y,E)). \end{aligned}$$

Substituting this into the expression above for vec(E), we get

$$\operatorname{vec}(E) = \left((e^{W_{\mathbf{b}}(Y^T)} \otimes I_n) + (I_n \otimes Y)\int_0^1 (e^{W_{\mathbf{b}}(Y^T)s} \otimes e^{-sW_{\mathbf{b}}(Y)})ds\right)\operatorname{vec}(L_{W_{\mathbf{b}}}(Y,E)).$$

As $\operatorname{vec}(L_{W_{\mathbf{b}}}(Y,E)) = K(Y)\operatorname{vec}(E)$, clearly

$$K(Y)^{-1} = (e^{W_{\mathbf{b}}(Y^T)} \otimes I_n) + (I_n \otimes Y)\int_0^1 (e^{W_{\mathbf{b}}(Y^T)s} \otimes e^{-sW_{\mathbf{b}}(Y)})ds.$$

By (1.8) in Lemma 1.5.4 this can be rewritten as

$$K(Y)^{-1} = (e^{W_{\mathbf{b}}(Y^T)} \otimes I_n) + (I_n \otimes Y) \int_0^1 (e^{s(W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y))}) ds. \tag{4.7}$$

Theorem 4.4.10. We get three representations for $K^{-1} = K(Y)^{-1}$;

$$K^{-1} = \begin{cases} e^{W_{\mathbf{b}}(Y^T)} \otimes I_n + (I_n \otimes Y)\psi_1(W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)), \\ e^{W_{\mathbf{b}}(Y^T)} \otimes I_n + (e^{W_{\mathbf{b}}(Y^T)/2} \otimes Ye^{-W_{\mathbf{b}}(Y)/2}) \operatorname{sinch}\left(\frac{W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)}{2}\right), \\ e^{W_{\mathbf{b}}(Y^T)} \otimes I_n + \frac{e^{W_{\mathbf{b}}(Y^T)} \otimes W_{\mathbf{b}}(Y) + I_n \otimes Y}{2} \tau\left(\frac{W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)}{2}\right), \end{cases} \tag{4.8}$$

where

$$\begin{aligned} \psi_1(x) &= \sum_{j=0}^{\infty} \frac{x^j}{(j+1)!}, \\ \operatorname{sinch}(x) &= \begin{cases} \sinh(x)/x & \text{if } x \neq 0, \\ 1 & \text{if otherwise}, \end{cases} \\ \tau(x) &= \begin{cases} \tanh(x)/x & \text{if } x \neq 0, \\ 1 & \text{if otherwise}, \end{cases} \end{aligned}$$

and the third expression in (4.8) is valid only when $\|W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)\| \leq \pi$.

Proof. First $\psi_1(x) = \frac{e^x - 1}{x}$ which equals $\int_0^1 e^{sx} ds$ (see [16, (10.19)] for this) so by letting $x = W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)$ and substituting this into (4.7) we get the first expression

$$K^{-1} = e^{W_{\mathbf{b}}(Y^T)} \otimes I_n + (I_n \otimes Y)\psi_1(W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)).$$

Again from [16, (10.19)] we see that $\int_0^1 e^{sx} ds = e^{x/2} \operatorname{sinch}(x/2) = \frac{e^x+1}{2}\tau(x/2)$, so putting $x = W_{\mathbf{b}}(Y^T) \oplus -W_{\mathbf{b}}(Y)$ gives us (we write $W_{\mathbf{b}}(Y)$ as W and $W_{\mathbf{b}}(Y^T)$ as W^T to reduce clutter)

$$\begin{aligned} K^{-1} &= e^{W^T} \otimes I_n + (I_n \otimes Y)e^{(W^T \oplus -W)/2} \operatorname{sinch}((W^T \oplus -W)/2) \\ &= e^{W^T} \otimes I_n + (I_n \otimes Y)(e^{W^T/2} \otimes e^{-W/2}) \operatorname{sinch}((W^T \oplus -Y)/2) \\ &= e^{W^T} \otimes I_n + (e^{W^T/2} \otimes Ye^{-W/2}) \operatorname{sinch}((W^T \oplus -W)/2), \end{aligned}$$

and

$$\begin{aligned}
K^{-1} &= e^{W^T} \otimes I_n + \frac{1}{2}(I_n \otimes Y)(e^{W^T \oplus -W} + I_{n^2})\tau((W^T \oplus -W)/2) \\
&= e^{W^T} \otimes I_n + \frac{1}{2}(I_n \otimes Y)(e^{W^T} \otimes e^{-W} + I_{n^2})\tau((W^T \oplus -W)/2) \\
&= e^{W^T} \otimes I_n + \frac{1}{2}((e^{W^T} \otimes Ye^{-W}) + (I_n \otimes Y))\tau((W^T \oplus -W)/2) \\
&= e^{W^T} \otimes I_n + \frac{(e^{W^T} \otimes W) + (I_n \otimes Y)}{2}\tau((W^T \oplus -W)/2).
\end{aligned}$$

□

Unfortunately none of these expressions seem to lend themselves to efficient or reliable computation, however they are useful theoretically. The next theorem shows when $K(A)$ is nonsingular.

Theorem 4.4.11. Let $W_{b_1 \dots b_m}(A)$ be a primary matrix function such that for no $i \le m$ do we have $\lambda_i = 0$ and $b_i \neq 0$, or $\lambda_i = -e^{-1}$ and $b_i = 0, -1$, and let $K(A)$ be the Kronecker form of the Fréchet derivative, then $K(A)$ is always nonsingular.

Proof. The eigenvalues υ_{ij} of $K(A)$ are

$$\upsilon_{ij} = W[\lambda_i, \lambda_j] = \begin{cases} \frac{W_{b_j}(\lambda_i) - W_{b_j}(\lambda_j)}{\lambda_i - \lambda_j} & \text{if} \quad \lambda_i \neq \lambda_j, \\ W'_{b_i}(\lambda_i) & \text{if} \quad \lambda_i = \lambda_j, \end{cases}$$

for $i, j = 1{:}n$ (see Higham [16, Thm. 3.9]). First we show that if $\lambda_i \neq \lambda_j$ then $W_{b_i}(\lambda_i) \neq W_{b_j}(\lambda_j)$, which gives us $\upsilon_{ij} \neq 0$. If $b_i \neq b_j$ then this is obvious from the definition of the branch structure of W (see Section 2.2), if $b_i = b_j$ then because W_{b_i} is an injective function (it must be as we^w is single-valued) distinct inputs go to distinct outputs. Therefore if $\lambda_i \neq \lambda_j$ then $\upsilon_{ij} = W[\lambda_i, \lambda_j] \neq 0$.

If $\lambda_i = \lambda_j$ then $\upsilon_{ij} = W'_{b_i}(\lambda_i) = e^{-W_{b_i}(\lambda_i)}/(1 + W_{b_i}(\lambda_i))$ which is never equal to zero because $W_b(x) \not\to +\infty$ as $x \to c$ for any branch or point $c \in \mathbb{C}$. □

For practical computation we look at estimating the condition number. If this can be integrated in with the actual computation of $W_{\mathbf{b}}(A)$ then some of the intermediate steps involved in computing $W_{\mathbf{b}}(A)$ could be reused to compute $\|L_{W_{\mathbf{b}}}(A)\|$. We return to the Fréchet derivative in Chapter 6.

4.5 Numerical Considerations

4.5.1 Computing the Residual

If we have an approximation $\widetilde{W}$ to $W = W(Z)$ (here the branches used are unimportant), then as W is defined to be any solution to $Z = We^W$ we can form the residual

$$R(\widetilde{W}, Z) = Z - \widetilde{W}e^{\widetilde{W}}. \tag{4.9}$$

The norm of the residual gives an indication of how much the errors in computation stop our computed solution from being a solution to $Z = We^W$. In practice some errors accumulate just from forming the residual via matrix multiplication. Therefore we compute the *normalised residual*

$$\overline{R}(\widetilde{W}, Z) = \frac{R(\widetilde{W}, Z)}{\|Z\| + \|W\|\|e^W\|}. \tag{4.10}$$

We divided by this quantity as by forming R we can expect rounding errors approximately equal to $(\|z\| + \|W\|\|e^W\|)\epsilon$ to build up. If our method to compute W is exact then $\|\overline{R}(\widetilde{W}, Z)\| \approx \epsilon$ hence the amount to which $\|\overline{R}\| > \epsilon$ indicates the amount to which W is not computed exactly. We can obtain a lower bound for the forward error from the residual, as described below.

4.5.2 Estimating Accuracy

Having computed the residual we can obtain an a posteriori lower bound on the forward error of $W(Z)$. To see the relationship between the residual and the error term let $\widetilde{W} = W + \Delta$ where W is the true solution and Δ is an error term. So by substituting this into the residual and using $e^{W+\Delta} = e^W + L_{\exp}(W, \Delta) + O(\|\Delta\|^2)$, where $L_{\exp}(W, \Delta)$ is the Fréchet derivative of e^W (see [1]), we get

$$\begin{aligned}
R(\widetilde{W}, Z) &= Z - (W + \Delta)e^{W+\Delta} \\
&= Z - (W + \Delta)(e^W + L_{\exp}(W, \Delta) + O(\|\Delta\|^2)) \\
&= Z - We^W + \Delta e^W + WL_{\exp}(W, \Delta) + O(\|\Delta\|^2) \\
&= \Delta e^W + WL_{\exp}(W, \Delta) + O(\|\Delta\|^2),
\end{aligned}$$

$$\|R(\widetilde{W}, Z)\| \leq \|\Delta\|\|e^W\| + \|W\|\|L_{\exp}(W, \Delta)\| + O(\|\Delta\|^2),$$

using any consistent norm. From here we neglect the higher order terms and use the inequality $\|L(X, E)\| \leq \|L(X)\|\|E\|$, which follows from the definition of $\|L(X)\|$.

$$\begin{aligned} \|R(\widetilde{W}, Z)\| &\leq \|\Delta\|\|e^W\| + \|W\|\|L_{\exp}(W)\|\|\Delta\| \\ &= \|\Delta\|(\|e^W\| + \|W\|\|L_{\exp}(W)\|), \\ \|\Delta\| &\geq \frac{\|R(\widetilde{W}, Z)\|}{\|e^W\| + \|W\|\|L_{\exp}(W)\|}. \end{aligned} \tag{4.11}$$

The calculation of $\|L_{\exp}(W)\|$ is discussed in [1] and the Matrix Function Toolbox [14] function `funm_condest_fro` can calculate $\|L_{\exp}(W)\|$ in the Frobenius norm.

4.6 Evaluating Nonprimary Functions

Most methods only allow primary functions to be calculated or even just single branch functions. This is because they do not have access to the Jordan form in Definition 4.2.2. Only when A is normal or very close to being normal is it numerically stable to compute the Jordan form, if these matrices have repeated eigenvalues (to working precision) then we could use different branches for the same eigenvalue. Hence obtaining nonprimary solutions. This is not the central theme of this thesis so is only mentioned briefly.

Chapter 5

Taylor and Branch Point Series

5.1 Introduction

In this chapter we look at adapting the Taylor series approximation in Section 2.3.1 to matrix functions; we also adapt the branch point series in Section 2.3.3 for matrices with eigenvalues clustered around $-e^{-1}$.

5.2 Taylor Series of General Functions

5.2.1 Applications

Taylor series are not generally used on their own to compute matrix functions, but they are an integral part of the general Schur–Parlett method (see Chapter 6). In this method Taylor series are often used to calculate the diagonal blocks of a matrix; as a result the input matrices are usually upper triangular and have near-constant diagonal. A thorough understanding of Taylor series computations allows us to construct methods that exploit this structure.

5.2.2 Computing Functions with Taylor Series

Definition 5.2.1. Suppose we have a scalar function $f\colon \mathbb{C} \to \mathbb{C}$ which is expressed by the infinite (or finite) sum $f(x) = \sum_{i=0}^{\infty} a_i(x-\alpha)^i$ for some coefficients a_i and some constant $\alpha \in \mathbb{C}$; then we express a matrix function $f\colon \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ analogously by letting

$$f(X) = \sum_{i=0}^{\infty} a_i (X - \alpha I_n)^i. \tag{5.1}$$

Like the scalar case we need to think about whether the series converges to the correct value or diverges (or oscillates). Theorem 5.2.2 gives a necessary and sufficient condition for convergence. In the case where the series might diverge then the Taylor series only computes a matrix function (i.e. be equivalent to the definition from the Jordan canonical form in Chapter 4) for certain matrices.

Theorem 5.2.2. Let f have a Taylor series expansion about $\alpha \in \mathbb{C}$ given by $f(x) = \sum_{i=0}^{\infty} a_i (x-\alpha)^i$ with radius of convergence r. If $A \in \mathbb{C}^{n\times n}$ then $f(A)$ is given by

$$f(A) = \sum_{i=0}^{\infty} a_i (A - \alpha I_n)^i$$

if and only if each eigenvalue $\lambda_i \in \Lambda(A) = \{\lambda_1, \ldots, \lambda_m\}$ of A satisfies one of the two conditions

1. $|\lambda_i - \alpha| < r$,

2. $|\lambda_i - \alpha| = r$ and the series for $f^{(n_i-1)}(x)$ converges at $\lambda = \lambda_i$ where $f^{(n_i-1)}(x)$ is the $(n_i - 1)$th derivative of f and $n_i = \mathrm{idx}_A(\lambda_i)$ is the index of λ_i.

Proof. See [16, Thm. 4.7] □

If the radius of convergence of the Taylor series of $f(x)$ is r, then if x is less than r in absolute value, the Taylor series of $f(x)$ converges x. Thus we can say that a sufficient condition for convergence in the matrix case is if all eigenvalues have absolute value less than r. As a consequence functions with an infinite radius of convergence always converge but for functions where this is not the case we may only be able to use this method for a matrices whose spectral radius is less than or possibly when equal to r. If the spectral radius is equal to r finite precision computations can sometimes produce unstable results, hence in practice we require the spectral radius to be less than r.

5.2.3 Error Analysis

As usual there are two types of error: roundoff and truncation error. As we increase the number of terms the truncation error tends to zero, however we cannot measure this by looking at the residual as when the truncation error becomes small rounding errors eventually dominate the residual.

One way to estimate the truncation error is from successive Taylor approximations $f_0, f_1, \dots f_k$ where $f(x) \approx f_j = \sum_{i=0}^{j} a_i x^i$, which should get closer and closer to the solution as k increases. The difference between successive approximations is called the *local error* and is defined by

$$e_j = \|W_{j+1} - W_j\|. \tag{5.2}$$

We should think of e_j as an estimate of the truncation error of W_j defined by $\|W_j - W^*\|$ where W^* is the true solution. Calculating the difference between successive estimates is easy enough, and the relative local error is just

$$\frac{e_j}{\|W_j\|}; \tag{5.3}$$

this makes it an attractive option in deciding when to terminate the series. Unfortunately Taylor series can have very nonmonotonic convergence (i.e. e_{j+1} can be bigger than e_j even though the series is convergent) so this is not always a good criterion. Fortunately Theorem 5.2.3 gives an upper bound on the norm of the remainder term from a truncated Taylor series. The remainder term is denoted

$$R_s(A) = f(A) - \sum_{i=0}^{s} a_i (A - \alpha I_n)^i. \tag{5.4}$$

$\|R_s(A)\|$ can be bounded in terms of the norm of the derivatives, as shown in this next theorem.

Theorem 5.2.3. *Let $R_s(A)$ be the remainder term in the truncated Taylor series of $f(A)$ with s. Then*

$$\|R_s(A)\| \le \frac{1}{s!} \max_{t \in [0,1]} \|(A - \alpha I_n)^s f^{(s)}(\alpha I_n - t(A - \alpha I_n))\|, \tag{5.5}$$

for any matrix norm.

Proof. See [20, Cor. 2]. □

If $Z = A - \alpha I_n$ and f is analytic on a closed convex set Ω and $\Lambda(Z) \subset \Omega$ then [9, Alg. 2.6] gives the inequality

$$\max_{t\in[0,1]} \|f^{(s)}(\alpha I_n - tZ)\|_\infty \le \max_{0\le r\le n-1} \frac{\omega_{s+r}}{r!} \|(I - |N|)^{-1}\|_\infty, \tag{5.6}$$

where $\omega_{s+r} = \sup_{z\in\Omega} f^{(s+r)}(z)$ and N is the strictly upper triangular part of the Schur decomposition of A.

If we can calculate the Schur form of A then this bound is easily evaluated except for ω_{s+r} which in most cases must be approximated. However for certain functions f we might be able to find reasonable bounds depending on its topological and geometric structure.

5.3 Taylor series approximation to $W_b(A)$

5.3.1 Calculating $W(A)$ using Taylor series

A truncated Taylor series is a polynomial, although a polynomial for which we cannot know the degree in advance. For this reason we evaluate the polynomial term by term until the global error bound reaches below a certain tolerance. To save computational time we only calculate the global error bound when the local error is below the given tolerance also. We can only use Taylor series to evaluate on a single branch of W.

5.3.2 Truncation Error Bound

By the inequalities (5.5) and (5.6) if $R_s(A) = W_b(A) - \sum_{i=0}^{s} \frac{W_b^{(i)}(\alpha)}{i!}(A - \alpha I_n)^i$ and $Z = A - \alpha I_n$ then we have

$$\|R_s(A)\|_\infty \le \|Z^s\|_\infty \max_{0\le r\le n-1} \frac{\omega_{s+r}}{r!} \|(I - |N|)^{-1}\|_\infty,$$

where $\omega_{s+r} = \sup_{z\in\Omega} |W_b^{(s+r)}(z)|$.

Now we need to bound ω_{s+r} which means finding the supremum of $|W_b^{(k)}(z)|$ on the smallest convex set containing $\Lambda(A)$. It is quite possible that the supremum is at one of the eigenvalues so we are able to approximate ω_{s+r} by checking for the maximum of the $|W_b^{(k)}(z)|$ at these points. In [9] this approximation is recommended unless a better one can be found using particular properties of the function. From our discussions in Section 2.5 on the geometry of W_b and its derivatives this approximation should be okay as long as no eigenvalue is "too close" to a branch point as if z moves around close to a branch point the values of $|W_b^{(k)}(z)|$ can change erratically for large k.

5.3.3 Computational Cost

Recall that in Chapter 2 two methods were developed for computing the derivatives of W_b, the first calculates the derivative function in terms of $W_b(x)$ while the second calculates the terms of the Taylor series of $W_b(\alpha)$. In calculating the truncation error bound we are required to compute the Taylor series not only at α but at all the eigenvalues as well, it has to use up to the $(k+n)$th derivative where k is the number of required terms (which is not known in advance) and n is the size of the matrix.

Implicit Differentiation

Algorithm 2.4.2 costs $\frac{9}{2}(k+n)^2$ flops initially and an extra $2j$ flops every time it evaluates the jth derivative. It evaluates the derivatives of n points up to the $(k+n)$th derivative and one point up to the kth derivative. So the cost is

$$\begin{aligned}
&\frac{9}{2}(k+n)^2 + \sum_{j=0}^{k+n} 2nj + \sum_{j=0}^{n} 2j \\
&= \frac{9}{2}(k+n)^2 + n(k+n+1)(k+n) + (n+1)n \\
&= \frac{9}{2}k^2 + nk^2 + 2n^2k + 10nk + n^3 + \frac{13}{2}n^2 + n \\
&\approx nk^2 + 2n^2k + n^3.
\end{aligned}$$

Here we have neglected lower order terms of n and k. We also perform k matrix multiplications which each costing $2n^3$, we can reduce this to $n^3/3$ flops by taking advantage of the triangular structure in the matrices. The total cost is approximately

$$kn^3/3 + nk^2 + 2n^2k. \tag{5.7}$$

Note that none of the terms from the initial cost of $\frac{9}{2}(k+n)^2$ appear in the higher order approximation. In practice we place an upper limit on the number of terms calculated; in double precision IEEE arithmetic we cannot calculate above the 143rd term due to overflow, therefore the upper limit on k should be about 100.

Automatic Differentiation

Algorithm 2.4.4 requires j^2 flops to calculate and evaluate up to the jth derivative at a single point. We evaluate n points up to the $(k+n)$th derivative and one point up to the kth derivative. So the cost is

$$k^2 + n(n+k)^2 \approx n^3 + 2n^2k + nk^2.$$

We also perform k matrix multiplications here so the total cost in flops is approximately

$$kn^3/3 + nk^2. \tag{5.8}$$

Unlike Algorithm 2.4.2 it may be possible to compute arbitrarily many terms depending where we expand the Taylor series about. Overflow only appears to be a problem when the series is expanded too close to a branch point.

5.3.4 Taylor Series Algorithms

We give two algorithms in pseudocode for the Taylor series. We use the implicit differentiation method in Algorithm 5.3.1 and the automatic differentiation method in Algorithm 5.3.2. The MATLAB code for Algorithm 5.3.2 can be found in Appendix A.5, the code for Algorithm 5.3.1 is very similar.

5.4 Branch Point Series approximation to $W_b(A)$

5.4.1 Calculating $W_0(A)$ using the Branch Point Series

As all derivatives of W_b at $-e^{-1}$ are infinite we cannot use a Taylor expansion, however in Section 2.3.3 we saw a series which calculates $W_0(z)$ where $|z+e^{-1}| < e^{-1}$.

Algorithm 5.3.1 Calculates $W_b(T)$ for upper-triangular T by the Taylor expansion of W_b using implicit differentiation.

```
Require: Let $T \in \mathbb{C}^{n\times n}$ be upper-triangular, $\alpha \in \mathbb{C}$, tol $\in \mathbb{R}$, $b \in \mathbb{Z}$ and maxterms $\in$
    $\mathbb{N}$ be user supplied constants.
 1: Let $\mu = \|(I_n - |N|)\,\mathrm{ones}(n,1)\|_\infty$, for $N$ the strict upper-triangular part of $T$.
 2: Let dn_max = 0
 3: Let $Z = T - \alpha I_n$ and set $W = W_b(\alpha) I_n$
 4: for $j = 1\colon$maxterms do
 5:   $W_{\text{old}} = W$
 6:   Update $W = W + W_b^{(j)}(\alpha) Z$ and update $Z = Z(T - \alpha I_n)$
 7:   if $\|W_{\text{old}} - W\|_\infty / \|W_{\text{old}}\|_\infty$ overflows then
 8:     return error code 2
 9:   end if
10:   if $\|W_{\text{old}} - W\|_\infty / \|W_{\text{old}}\|_\infty \le$ tol then
11:     for $r =$ max_dn$\colon k + n - 1$ do
12:       Let $\eta^\star$ be the vector of $r$th derivatives of the eigenvalues of $T$ (i.e. $(\eta^\star)_i =$
          $W_b^{(r)}(\lambda_i)$ for $i = 1\colon n$) and let $\psi_r = \|\eta^\star\|_\infty$
13:     end for
14:     Update max_dn $= j + n$
15:     $\omega = \max\{\psi_{j+r} r! \mid r = 0\colon n-1\}$ and $\tau = \|Z\|_\infty \mu\omega/j!$
16:     if $\tau$ overflows then
17:       return error 3
18:     else if $\tau \le$ tol $\|T\|_\infty$ then
19:       return $W$ and error code 0 (no error)
20:     end if
21:   end if
22: end for
23: return error code 1
```

Algorithm 5.3.2 Calculates $W_b(T)$ for upper-triangular T by the Taylor expansion of W_b using automatic differentiation.

```
Require: Let $T \in \mathbb{C}^{n\times n}$ be upper-triangular, $\alpha \in \mathbb{C}$, tol $\in \mathbb{R}$, $b \in \mathbb{Z}$ and maxterms $\in$
    $\mathbb{N}$ be user supplied constants.
 1: Let $w = W_b(\alpha)$ and let $\Gamma_i = W_b(\lambda_i)$ for all eigenvalues of $T$
 2: Initialise sequences $c$, $d$ and $\eta^i$, $\xi^i$ $(i = 1\colon n)$ as in Theorem 2.4.3, where $c_j$ is to
    be the $j$th term of the Taylor series about $\alpha$ and $\eta^i_j$ the $j$th term of the series
    about the $i$th eigenvalue
 3: Let $\mu = \|(I_n - |N|)\,\mathrm{ones}(n,1)\|_\infty$, for $N$ the strict upper-triangular part of $T$.
 4: Let dn_max = 0
 5: Let $Z = T - \alpha I_n$ and set $W = c_0 I_n$
 6: for $j = 1\colon$maxterms do
 7:   Calculate $d_{j-1}$ and $c_j$ using Theorem 2.4.3
 8:   $W_{\mathrm{old}} = W$
 9:   Update $W = W + c_j Z$ and update $Z = Z(T - \alpha I_n)$
10:   if $\|W_{\mathrm{old}} - W\|_\infty / \|W_{\mathrm{old}}\|_\infty$ overflows then
11:     return error code 2
12:   end if
13:   if $\|W_{\mathrm{old}} - W\|_\infty / \|W_{\mathrm{old}}\|_\infty \le$ tol then
14:     for $r =$ max_dn$\colon k + n - 1$ do
15:       Calculate $\xi^i_r$ and $\eta^i_{r+1}$ $(i = 1\colon n)$ using Theorem 2.4.3
16:       Let $\eta^\star$ be the vector of $r$th derivatives of the eigenvalues of $T$ (i.e. $(\eta^\star)_i = \eta^i_r$
          for $i = 1\colon n$) and let $\psi_r = \|\eta^\star\|_\infty$
17:     end for
18:     Update max_dn $= j + n$
19:     $\omega = \max\{\psi_{j+r} r! \mid r = 0\colon n-1\}$ and $\tau = \|Z\|_\infty \mu\omega/j!$
20:     if $\tau$ overflows then
21:       return error 3
22:     else if $\tau \le$ tol $\|T\|_\infty$ then
23:       return $W$ and error code 0 (no error)
24:     end if
25:   end if
26: end for
27: return error code 1
```

We can also substitute a matrix into the series to calculate $W_0(A)$ where all eigenvalues of A are less than a distance of e^{-1} from $-e^{-1}$. In order to do this we first transform A, for this we need the square root of a matrix.

Definition 5.4.1. Let $B \in \mathbb{C}^{n\times n}$, a matrix $X \in \mathbb{C}^{n\times n}$ is a *square root* of B if $X^2 = B$. A matrix has a square root as long as its *ascent sequence* defined by

$$d_i = \dim(\text{null}(B^i)) - \dim(\text{null}(B^{i-1})), \text{ for } i = 1, 2, \dots, \tag{5.9}$$

has no two terms of the same odd integer[16, Thm. 1.2]. If B is nonsingular then $\dim(\text{null}(B^i)) = 0$ for all $i = 1, 2, \dots$, so the ascent sequence consists only of zeros. The principal square root of B exists and is denoted $B^{1/2}$ as long as B has no eigenvalues on the closed negative real axis. If B does have negative real eigenvalues then a nonprincipal square root exists as long as the condition in (5.9) is satisfied.

The square root can be calculated efficiently and stably using a variety of algorithms (See Higham [16, Chapter 6]) and can be computed in MATLAB by the function `sqrtm`, for this reason we say no more about computing the square root.

Let $A \in \mathbb{C}^{n\times n}$ have all eigenvalues within the disc of radius e^{-1} centred on $-e^{-1}$. First we transform[1] A using

$$P = (2(eA + I_n))^{1/2}. \tag{5.10}$$

Now we sum the terms of the series calculating the coefficients as we go. Unfortunately we cannot use the truncation error in Theorem 5.2.3 as that is only applicable to series of the form $f(x) = \sum_{k=0}^{\infty} a_k x^k$ where as our series involves an initial transform of x and is of the form $f(x) = \sum_{k=0}^{\infty} a_k((2(ex+1))^{1/2})^k$. It is possible that an analogous truncation bound could be found but one was not found for this thesis. Instead we terminate when the local error (5.2) is sufficiently small. The nonmonotonic behaviour of the convergence may potentially degrade this as an estimate for the truncation error but at present this is the only method available.

[1]If $eA+I_n$ is nonsingular then P is guaranteed to exist. The only case in which $eA+I_n$ is singular is when $-e^{-1}$ is an eigenvalue of A. This may possibly violate the condition for the existence of the square root, but as $-e^{-1}$ cannot be stored precisely on a finite precision machine, we do not expect it to cause too many problems and in practice no problems have been observed.

Theorem 5.4.2. If $A \in \mathbb{C}^{n\times n}$ has all eigenvalues in the open disc of radius e^{-1} about the point $-e^{-1}$ then the series $\sum_{k=0}^{\infty} \mu_k P^k$ converges to $W_0(A)$ where $P = (2(eA + I_n))^{1/2}$ and μ_k are defined as in Section 2.3.3.

Proof. Let $B_m(A) = \sum_{k=0}^{m} \mu_k((2(eA + I_n))^{1/2})^k$, the eigenvalues λ_i of B_m are $B_m(\lambda_i) = \sum_{k=0}^{m} \mu_k((2(e\lambda_i + 1))^{1/2})^k$ which converge to $W_0(\lambda_i)$ as $m \to \infty$ as λ_i is within the region of convergence. Hence $B_m(A)$ does not diverge, let $B_m(A) \to B(A)$ as $m \to \infty$. So $B(A)$ has the same eigenvalues as $W_0(A)$ which by Theorem 4.3.4 implies $\lim_{m\to\infty} B_m(A) = W_0(A)$. □

This method is implemented in Algorithm 5.4.3 and can be easily implemented in MATLAB by modifying the code for the Taylor series with automatic differentiation in Appendix A.5. The algorithm requires A to be upper-triangular, whether to do this before or after the transformation (5.10) is not addressed here as if the branch point series is used in conjunction with the Schur–Parlett method A should be upper-triangular anyway.

Algorithm 5.4.3 Calculates $W_0(T)$ for upper-triangular T by the branch point expansion of W_0.

```
Require: Let $T \in \mathbb{C}^{n\times n}$ be upper-triangular, $\alpha \in \mathbb{C}$, tol $\in \mathbb{R}$, and maxterms $\in \mathbb{N}$ be
    user supplied constants.
 1: Initialise $\alpha_0 = 2$, $\alpha_1 = -1$ and $\mu_0 = -1$, $\mu_1 = 1$
 2: Transform $T = (2(eT - I_n))^{1/2}$ and set $W = -I_n + T$
 3: Let $Z = T$
 4: for $j = 2$: maxterms do
 5:   $W_{\text{old}} = W$
 6:   Calculate $\mu_j$ from previous $\mu_i$ and $\alpha_i$.
 7:   Update $W = W + \mu_j Z$ and update $Z = ZT$
 8:   if $\|W_{\text{old}} - W\|_\infty / \|W_{\text{old}}\|_\infty$ overflows then
 9:     return error code 2
10:   end if
11:   if $\|W_{\text{old}} - W\|_\infty / \|W_{\text{old}}\|_\infty \le$ tol then
12:     return $W$ and error code 0 (no error)
13:   end if
14: end for
15: return error code 1
```

5.4.2 Cost of Algorithm 5.4.3

In (2.8) the coefficient m_j costs approximately $2j$ flops to compute so the cost of Algorithm 5.4.3 is $\sum_{j=0}^{k}(n^3/3 + 2j) \approx kn^3/3$ (taking advantage of the triangular structure) where k is the requisite number of terms.

5.5 Summary

In Table 5.1 the pros and cons of each of the methods are compared. The most expensive method is the Taylor Series with Implicit Differentiation, although the cost only differs in the lower order terms. This method is also the most versatile, being able to evaluate matrices with eigenvalues close to zero. The automatic differentiation method is similar to the implicit differentiation except that it has trouble when eigenvalues are close to the origin. The branch point series is only for use on very specific matrices: ones with all eigenvalues close to the branch point which is when the other methods fail.

Method	Computational Cost	Additional Requirements
Taylor Series (I.D.)	$n^3k/3 + nk^2 + 2n^2k$	maxterms $\lesssim 143$
Taylor Series (A.D.)	$n^3k/3 + nk^2$	$\rho(A - \alpha I_n) < \lvert\alpha\rvert$
Branch Point Series	$n^3k/3 + k^2$	$\rho(A + e^{-1}I_n) < e^{-1}$ and $b = 0$

Table 5.1: Here we compare the three methods for evaluating $A \in \mathbb{C}^{n\times n}$ discussed in this chapter (I.D. stands for Implicit Differentiation and A.D. for Automatic Differentiation). In the "Computational Cost" column k denotes the number of required terms, and n the size of the matrix. Note how the higher order terms in the cost are the same for all three methods. In the "Additional Requirements" column b is the branch and $\alpha \in \mathbb{C}$ is the point about which we expand the Taylor series. For the Taylor series these requirements are in addition to requirements that if $b = -1$, 0 or 1 then $\rho(A - \alpha I_n) < |e^{-1} + \alpha|$, and if $b \neq 0$ then $\rho(A - \alpha I_n) < |\alpha|$.

Chapter 6

Schur–Parlett Algorithm

This algorithm is a method for computing general matrix functions. It uses the Schur decomposition rather than transforming the matrix into Jordan form, thereby avoiding a highly unstable decomposition. It is intended for functions which have Taylor series with an infinite radius of convergence although for other functions such as the logarithm or the Lambert W function, the method can be adapted. We first describe the method as it appears in the MATLAB function `funm`, then explain how we must adapt this for W. The description is taken from [16, Chapter 9] and [9].

6.1 Outline of the Method

There are several distinct stages in the method, first of all we perform a Schur decomposition of $A \in \mathbb{C}^{n\times n}$ so that $T = Q^*AQ$ is upper triangular and Q is unitary. Now we assign to each diagonal element $T_{ii} = \lambda_i$ of T an integer $q_i \in \{1, 2, \dots, p\}$ where $p \leq n$. The purpose of this is the map each diagonal element into a set S_{q_i} such that every set $S_1, \dots, S_p$ is nonempty and the following two conditions are met:

1. Separation between the sets:

$$\min\{|\lambda - \mu| \colon \lambda \in S_i, \mu \in S_j, i \neq j\} > \delta. \tag{6.1}$$

2. Separation within the sets: for each S_k where $|S_k| > 1$, for each $\lambda \in S_k$ there exists $\mu \in S_k$ such that $|\lambda - \mu| \leq \delta$. This has the consequence that for S_k with

$|S_k| = m$

$$\max\{|\lambda - \mu|\colon \lambda, \mu \in S_k, \lambda \neq \mu\} \leq (m-1)\delta. \tag{6.2}$$

This process is called blocking; more is said about finding a good blocking strategy below. Each element of the vector q tells us which block the corresponding eigenvalue must go so the next step is to find a confluent permutation of q. That is one which systematically reorders q so that equal elements are adjacent; we label the permuted vector q'. Finding such a permutation is nontrivial but is not described in detail here as the method is already implemented in `funm` in MATLAB 7.1, see [9, Sec. 4, Alg. 4.2] for details of this.

We use this permutation to reorder the Schur decomposition, so if P is the permutation matrix, $T' = PTP^{-1}$ and $Q' = PQP^{-1}$. Now we can partition the reordered matrix T' such that T' is block upper-triangular with square matrices T'_{ii} along its diagonal where the eigenvalues of T'_{ii} all belong to the same block. We call T'_{ii} an atomic block.

$$T' = PTP^{-1} = \begin{pmatrix} t'_{11} & t'_{12} & \cdots & t'_{1n} \\ 0 & t'_{22} & \cdots & t'_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & t'_{nn} \end{pmatrix} = \begin{pmatrix} T'_{11} & T'_{12} & \cdots & T'_{1p} \\ 0 & T'_{22} & \cdots & T'_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & T'_{pp} \end{pmatrix}.$$

6.1.1 Blocking Strategies

Depending on f and how we wish to evaluate the atomic blocks, the way we block the eigenvalues can affect both the accuracy and efficiency of the method. The strategy implemented in `funm` is given in [9, Alg. 4.1] and also given here in Algorithm 6.1.1.

That this strategy might not be appropriate for W becomes apparent from an experiment similar to one done on the matrix exponential in [16, Sec. 9.4] which involves the Forsythe matrix $A \in \mathbb{C}^{30\times 30}$ generated from the MATLAB function `gallery('forsythe',30)`. The eigenvalues of A cluster roughly around the circle of radius 0.5 about the origin. With a tolerance of 0.1 the blocking algorithm puts each eigenvalue in its own block. Calculating $W_0(A)$ using this blocking gives a computed

Algorithm 6.1.1 The blocking algorithm employed by `funm`.

```
Require: Triangular $T \in \mathbb{C}^{n\times n}$ with eigenvalues $\lambda_i = t_{ii}$, positive parameter $\delta$,
 1: $p = 1$, initialise $S_p = \emptyset$
 2: for $i = 1\colon n$ do
 3:    if $\lambda_i \neq S_q$ for all $1 \le q \le p$ then
 4:       Assign $\lambda_i$ to $S_p$
 5:       $p = p + 1$
 6:    end if
 7:    for $j = i + 1\colon n$ do
 8:       Denote by $S_{q_i}$, the set that contains $\lambda_i$.
 9:       if $\lambda_j \notin S_{q_i}$ then
10:          if $|\lambda_i - \lambda_j| \le \delta$ then
11:             if $\lambda_j \notin S_k$ for all $1 \le k < p$ then
12:                Assign $\lambda_j$ to $S_{q_i}$
13:             else
14:                Move the elements of $S_{\max\{q_i,q_j\}}$ to $S_{\min\{q_i,q_j\}}$
15:                Reduce by 1 the indices of the sets $S_q$ for $q > \max(q_i, q_j)$.
16:                $p = p - 1$
17:             end if
18:          end if
19:       end if
20:    end for
21: end for
```

error of roughly 10^{-4} which is far in excess of $c_{W_0}(A)\epsilon = 10^{-10}$. If we change the tolerance to 0.2 the blocking algorithm puts all of the eigenvalues into a single block, which produces an atomic block with spectral radius of about 0.5. We use the Taylor series expansion of W_0 to evaluate this, expanding about the arithmetic mean of the eigenvalues, which is equal to or very close to zero as the eigenvalues are arranged more or less regularly in a circle. The radius of convergence of this series[1] could not possibly be more than e^{-1} due to the branch point at $-e^{-1}$ so by Theorem 5.2.2 we cannot expect the Taylor expansion of this block to converge. Hence for W we require further conditions to ensure we get a viable blocking. We return to this problem in Section 6.2.2.

[1]Bear in mind that we are evaluating the principal branch of W, if we evaluated any other branch we would also have a branch point at the origin.

6.1.2 Calculating $f(T)$

We no longer need the original T and Q so from hereon we use T and Q to refer to the reordered Schur decomposition. Once we have reordered and partitioned T we solve $f(T) = F$ one block at a time. First we evaluate the diagonal blocks, which is discussed for the Lambert W function in Section 6.2; once we have the diagonal blocks F_{ii} we can use the block form of Parlett's recurrence to solve the off-diagonal blocks.

The block form of Parlett's recurrence is:

$$T_{i,i}F_{i,j} - F_{i,j}T_{j,j} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1}(F_{i,k}T_{k,j} - T_{i,k}F_{k,j}). \tag{6.3}$$

This is a Sylvester equation, that is one of the form $AX + XB = C$ where A, B, C are known matrices and X is the unknown. The equation is nonsingular because T_{ii} and T_{jj} have no eigenvalues in common (See Higham [16, Appendix B.14]). If $A, B \in \mathbb{C}^{n\times n}$ are upper-triangular then we solve $AX + XB = C$ a column at a time in the order $x_1, \dots, x_n$ (here n is the size of X)

$$x_i = (A + B_{ii}I_n)^{-1}(c_i - \sum_{j=1}^{n} X(j, 1\colon i-1)B(1\colon i-1, i)), \tag{6.4}$$

where c_i are the columns of C. Each column x_i is the solution of a triangular systems. Using (6.4) for (6.3) we can calculate F_{ij} given all the blocks of F directly to the left and directly below F_{ij} as illustrated in Figure 6.1. We can safely take the blocks below the diagonal to be zero as if $j > i$ then the right-hand side of (6.3) reduces to $F_{ii}T_{ij} - T_{ij}F_{jj} = 0$ as $T_{ij} = 0$. As the left-hand side is nonsingular, this implies $F_{ij} = 0$. This applies to both primary and nonprimary functions, though for primary functions we can immediately see from Theorem 4.3.2 that F is upper-triangular. If $f(A)$ is nonprimary then F is block upper-triangular. So we can build up F either a block column at a time going up then right, or a block superdiagonal at a time. Once we have $F = f(T)$ we form $f(A) = Q^*FQ$ and return it. We say no more about calculating $f(T)$ as it is implemented in `funm`.

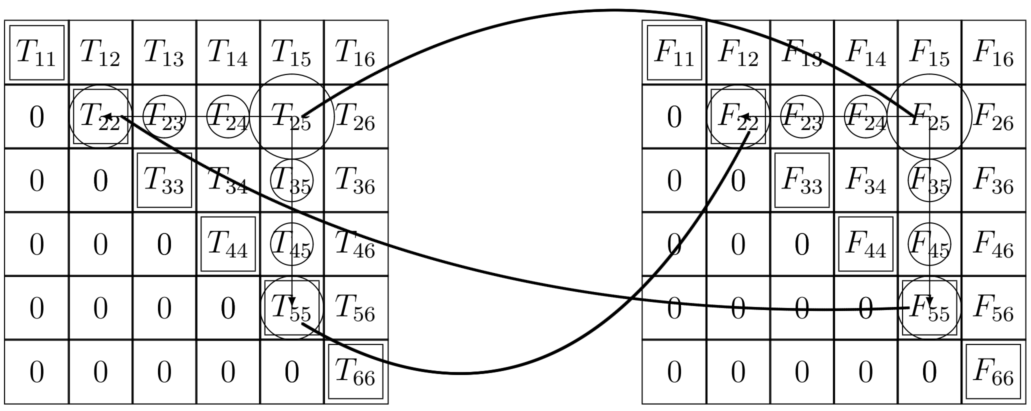
T_{11} T_{12} T_{13} T_{14} T_{15} T_{16}
0 T_{22} T_{23} T_{24} T_{25} T_{26}
0 0 T_{33} T_{34} T_{35} T_{36}
0 0 0 T_{44} T_{45} T_{46}
0 0 0 0 T_{55} T_{56}
0 0 0 0 0 T_{66}
F_{11} F_{12} F_{13} F_{14} F_{15} F_{16}
0 F_{22} F_{23} F_{24} F_{25} F_{26}
0 0 F_{33} F_{34} F_{35} F_{36}
0 0 0 F_{44} F_{45} F_{46}
0 0 0 0 F_{55} F_{56}
0 0 0 0 0 F_{66}

Figure 6.1: This figure shows the dependence of F_{25} on the blocks of F below to the left of it which must have been computed previously, and also its dependence on the blocks of T. The curved lines give a feel for how blocks of F and T are multiplied together.

6.2 Lambert W Function

In evaluating the atomic blocks we can most readily use the Taylor series computation in Chapter 5. We also look at the branch point series in the same chapter and develop a method which blocks the matrix in such a way that at least one of the two methods for evaluating the atomic blocks is always viable.

6.2.1 Primary and Nonprimary Matrices

By partitioning $T = (T_{ij})$ into blocks we are then free to use different methods to evaluate each diagonal block which opens the possibility of using different branches for different blocks. For instance if we use the Taylor expansion of W_b to calculate $W(T_{ii})$ and the Taylor expansion of another branch, say W_d to calculate another atomic block $W(T_{jj})$. Evaluating nonprimary solutions is problematic; we cannot simply assign two identical eigenvalues into different blocks and evaluate each block using a different branch as the eigenvalues of distinct blocks of T must be distinct. The ability to evaluate nonprimary solutions depends on how we evaluate the atomic blocks. As no such method is available this algorithm is clearly limited to computing primary matrix functions.

6.2.2 Blocking

As we have seen in Section 6.1.1 the blocking algorithm employed by `funm` can be unsuitable for W when computing the atomic blocks using Taylor series. We face problems of this sort regardless of which method we use, so choosing a good blocking is essential. We do not concentrate too much on finding the most efficient blocking strategy but merely finding a strategy that ensures the blocks can be calculated.

Suppose we have a blocking, each eigenvalue λ has an integer s_λ which specifies into which block it is to be placed. Let $S_1, \ldots, S_p$ be the list of sets containing the eigenvalues (i.e. $s_\lambda = j \Leftrightarrow \lambda \in S_j$). Let μ_j be the mean eigenvalue in S_j and let R_j be the value $\max\{|\lambda_k - \mu_j| \colon \lambda_k \in S_j\}$. The disc D_j is defined to be the closed disc in the complex plane centred on μ_j and of radius R_j. Suppose a disc D_j does not contain either the origin or the branch point $-e^{-1}$. If we compute the Taylor series of W_b for any branch about μ_j, then by Theorem 5.2.2 the block containing the eigenvalues of S_j converges, this is due to the fact that the radius of convergence of the Taylor series is greater than the radius of D_j. If D_j contains the origin then we must evaluate the block using the principal branch W_0 as the origin is a singularity for all nonprincipal branches. If D_j does not contain the origin but does contain the branch point $-e^{-1}$ then we can only use the Taylor series expansion to evaluate branches $b \neq -1, 0, 1$. We can however use the branch point series examined in Section 5.4 to evaluate W_0. If the disc D_j contains both the origin and branch point $-e^{-1}$ then the block cannot be evaluated.

We see that it is important to ensure that no disc contains both the origin and $-e^{-1}$. If a disc contains one of these then it restricts our choice of method and branch but as long as both points are not in the same disc we are guaranteed a valid method to evaluate the corresponding block. Ensuring this is not difficult and requires a simple modification of Algorithm 6.1.1, but in practice we may find that if a disc comes close to containing both points then this also causes problems therefore we define a circular buffer region around the branch points such that no disc may intersect the circle (i.e. each disc must either be totally inside or totally outside).

Figure 6.2 shows two blocking strategies for a random 55 by 55 complex matrix one which is suitable for W_0 and one which is not.

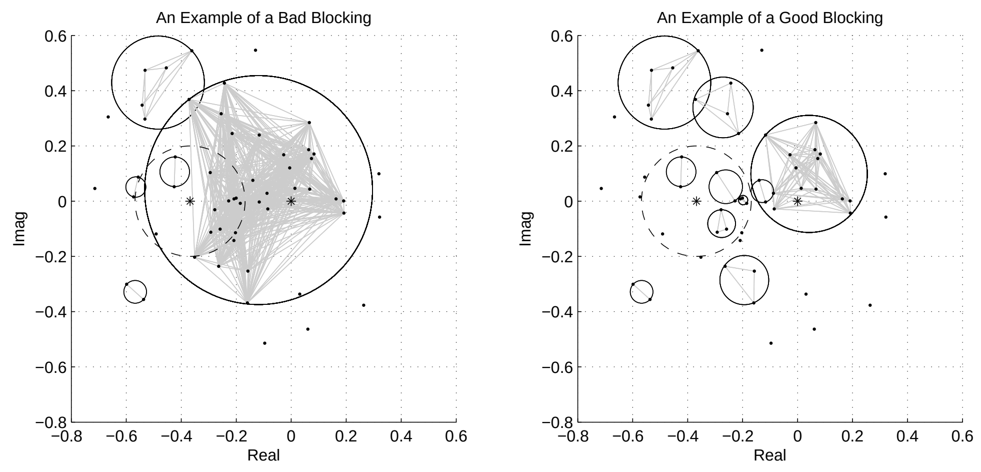

Figure 6.2: The left figure shows a blocking that is unsuitable for W_0, and the right figure shows one that is suitable. The black dots represent the eigenvalues in the complex plane, the dots are joined by a grey line if and only if the eigenvalues they represent are blocked together and the black solid circles show the boundaries of the D_k. The branch point and origin are represented by stars and the buffer is shown as a dashed circle around the branch point. If we are considering any branch other than the principal branch then there would be an analogous buffer around the origin.

To take account of these requirements the author proposes the blocking strategy must adhere to the following rules which include the ones give in Section 6.1. Under some circumstances some of these rules may be contradictory, in which case the preceding rule should take precedence. Note that b is the branch the user would *prefer* the block to be evaluated on and is subject to change if necessary by these rules.

Let β be a positive constant which denotes the size of the buffer around the branch point(s).

1. Separation within the blocks: for each S_k where $|S_k| > 1$ then for every $\lambda \in S_k$ there exists $\mu \in S_k$ such that $|\lambda - \mu| \le \delta$. This has the consequence that for S_k with $|S_k| = m$

$$\max\{|\lambda - \mu| \colon \lambda, \mu \in S_k, \lambda \ne \mu\} \le (m-1)\delta.$$

2. Let D_k be the disc centred on μ_k with radius R_k. If $b \neq 0$ then D_k must not intersect the circle of radius β around the origin. If D_k is inside the circle then the block must be evaluated using the Taylor series for the principal branch.

3. If $b = -1$, 0 or 1 then D_k must not intersect the circle of radius β about $-e^{-1}$. If D_k is inside the circle then the block should be evaluated using the branch point series for the principal branch.

4. Separation between the blocks:

$$\min\{|\lambda - \mu| \colon \lambda \in S_i, \mu \in S_j, i \neq j\} > \delta.$$

Note that the "Separation between blocks" condition is last and should be ignored when it interferes with any previous rules, such as if two close eigenvalues lie on opposite sides of the buffer circle (Say $|\lambda_i - \lambda_j| \ll 1$ but $|\lambda_i + e^{-1}| < \beta$ and $|\lambda_j + e^{-1}| > \beta$). In very rare situations this could be disastrous, but if two nearly equal eigenvalues lie on opposite sides of the buffer then reducing or expanding the buffer should correct the problem. Future algorithms may take this into account and choose the buffer intelligently. Note also that these rules are not the only ones we could adopt, for instance we could use only the Taylor expansion and insist that in the case of Rule 3 that a branch not equal to -1, 0 or 1 be used, however due to the importance of the principal branch the author believes it is important that one should always be able to evaluate it. More discussion on other potential blocking strategies is given in Chapter 8. We adapt Algorithm 6.1.1 to take account of the rules, the new method in pseudocode is given in Algorithms 6.2.1 and 6.2.2, and in MATLAB code in Appendix A.3.

The user does not usually know beforehand how the matrix will be blocked, which makes it difficult for them to specify which branch each distinct eigenvalue should be evaluated at. For the majority of this thesis we assume that all blocks are evaluated on the same branch, except of course when the block contains the origin which forces the principal branch to be chosen or $-e^{-1}$ which might force the principal branch if one attempts to use $b = -1$, 0, or 1.

Algorithm 6.2.1 A blocking strategy for the Lambert W function.

```
Require: upper-triangular $T \in \mathbb{C}^{n\times n}$ with eigenvalues $\lambda_i = t_{ii}$, positive parameters
    $\delta$ and $\beta$,
 1: Let $p = 1$
 2: for $i = 1 : n$ do
 3:   if $\lambda_i$ has not been assigned to a set then
 4:     Let $q_i = p$ and let $p = p + 1$
 5:   end if
 6:   for $j = i + 1 : n$ do
 7:     if $q_i \neq q_j$ then
 8:       if $\lambda_j$ is not assigned then
 9:         if blocking $\lambda_j$ with $\lambda_i$ and its set does not violate any rules then
10:           Set $q_j = q_i$.
11:         end if
12:       else
13:         if blocking the sets of $\lambda_j$ and $\lambda_i$ does not violate any rules then
14:           Amalgamate the sets of $\lambda_i$ and $\lambda_j$, setting $q_k = \min(q_i, q_j)$ wherever
              $q_k = \max(q_i, q_k)$ and reducing $q_t$ by one wherever $q_t > \max(q_i, q_j)$.
15:         end if
16:       end if
17:     end if
18:   end for
19: end for
```

6.2.3 The Schur–Parlett method for $W_b(A)$

Algorithm 6.2.3 computes $W_b(A)$. The methods used to evaluate the atomic blocks and compute the blocking are not specified explicitly and we assume that the blocking strategy is appropriate for whatever methods are used. The MATLAB code is listed in Appendix A.6.

If $\mathbf{i}$ and $\mathbf{j}$ are vectors of indices of length r and c respectively then we denote the submatrix $M(i_1\colon i_r, j_1\colon j_c)$ by $M_{\mathbf{ij}}$. Bold fonts denote vector subscript to distinguish them from scalar subscripts; for instance $M_{\mathbf{ij}}$ is a "matrix submatrix" of M while $M_{\mathbf{i}k}$ is a "vector submatrix" of M.

6.2.4 Computational Cost

The cost of computation depends on both the method used to evaluate the atomic blocks and the number of blocks the matrix is partitioned into. Suppose T has been partitioned into m blocks and the size of the jth block is n_j and we are using Taylor

Algorithm 6.2.2 Returns true if the given blocking is viable and false otherwise. Called only by Algorithm 6.2.1.

Require: buffer size β, λ_i and λ_j and the associated sets S_i and S_j.

```
1: if $|\lambda_i - \lambda_j| > \delta$ then
2:    return false
3: end if
4: Let $S_i$ and $S_j$ be the sets of $\lambda_i$ and $\lambda_j$, let $S = S_i \cup S_j$.
5: Let $\mu$ be the arithmetic mean of $S$, and let $R = \max\{|\mu - \lambda_i| \colon \lambda_i \in S\}$.
6: if $b = -1$, 0 or 1 then
7:    if $|\mu + e^{-1}| < \beta$ then
8:       if $|\mu + e^{-1}| + R \geq \beta$ then
9:          return false
10:      end if
11:   else
12:      if $|\mu + e^{-1}| - R \leq \beta$ then
13:         return false
14:      end if
15:   end if
16: end if
17: if $b \neq 0$ then
18:    if $|\mu| < \beta$ then
19:       if $|\mu| + R \geq \beta$ then
20:          return false
21:       end if
22:    else
23:       if $|\mu| - R \leq \beta$ then
24:          return false
25:       end if
26:    end if
27: end if
28: return true
```

Algorithm 6.2.3 Uses the Schur–Parlett Method to Calculate $W_b(A)$. Calls the functions `blocking`, `swapping` and `ordschur` found in the code of `funm` in MATLAB 7.1.

```
Require: Let $A \in \mathbb{C}^{n\times n}$ be the input matrix and let $b$ be the user chosen branch
 1: Let $T = UAU^*$ be the Schur decomposition.
 2: if $T$ is diagonal then
 3:    Let $W = U\,\mathrm{diag}(W_b(T_{11}),\ldots,W_b(T_{nn}))U^T$
 4:    return
 5: end if
 6: Calculate the blocking order by calling ord = blocking($T,\delta$), where ord is a
    vector of length $n$ which specifies which block $T_{ii}$ should be placed
 7: Call [ord, ind] = swapping(ord), and reverse the vector ord
 8: Call $[U,T]$ = ordschur($U,T,$ord) to reorder the Schur form,
 9: Let $m = \mathrm{length(ind)}$ be the number of blocks,
10: Initialise $F = 0$,
    % Loop through each diagonal block
11: for col $= 1\colon m$ do
12:    Let $c$ be the size of the $col$th diagonal block,
13:    Let $\mathbf{j} = j_1\colon j_c = \mathrm{ind(col)}$ and $T_{\mathbf{jj}} = T(j_1\colon j_c, j_1\colon j_c)$
14:    Calculate $F_{\mathbf{jj}} = W_b(T_{\mathbf{jj}})$
       % Calculate all blocks directly above, starting from the lowest
15:    for row $= \mathrm{col}-1\colon -1\colon 1$ do
16:       Let $r$ be the size of the $row$th diagonal block, and let $\mathbf{i} = i_1\colon i_r = \mathrm{ind(row)}$
          % Calculate the super-diagonal blocks
17:       if $c = 1$ and $r = 1$ then
18:          % Here $F_{ik}$, $F_{kj}$ and $F_{ii}$ are already known
19:          Let temp $= \sum_{k=i+1}^{j-1} F_{ik}T_{kj} - T_{ik}F_{kj}$,
20:          $F_{ij} = T_{ij}(F_{ii} - F_{jj} + \mathrm{temp})/(T_{ii} - T_{jj})$
21:       else
22:          % Here $F_{\mathbf{ii}}$, $F_{\mathbf{i}k}$ and $F_{k\mathbf{j}}$ are already known, and $F_{\mathbf{i}k} \in \mathbb{C}^{r\times 1}$ and $F_{k\mathbf{j}} \in \mathbb{C}^{1\times c}$
23:          Let rhs $= F_{\mathbf{ii}}T_{\mathbf{ij}} - T_{\mathbf{ij}}F_{\mathbf{jj}} + F_{\mathbf{i}k}T_{k\mathbf{j}} - T_{\mathbf{i}k}F_{k\mathbf{j}}$
24:          $F_{\mathbf{ij}} = \mathrm{sylv\_tri}(T_{\mathbf{ii}}, -T_{\mathbf{ii}}, \mathrm{rhs})$
25:       end if
26:    end for
27: end for
28: Let $W = UFU^T$,
29: if $A$ is real and $\|\Im(F)\|_1 < 10n\epsilon\|F\|_1$ then
30:    set $F = \Re(F)$
31: end if
32: print information to console if applicable and warnings if any blocks could not
    be calculated
33: return output.
```

series to evaluate the blocks, and let k_j be the number of terms necessary to ensure convergence of the jth block. If we evaluate all blocks using Algorithm 5.3.1 (the implicit differentiation method) then the cost is $\sum_{j=0}^{m} k_j n_j^3/3 + n_j k_j^2 + 2n_j^2 k_j$. As the diagonal of each block is nearly constant due to Rule 1, the matrix $T_{ii} - \mu_j I_{n_j}$ where μ_j is the mean of the eigenvalues, is nearly nilpotent. By [16, Sec. 9.1] the Taylor series terms after the nth should decay exponentially, therefore we can expect k to be not much greater than n, bearing this in mind the cost becomes approximately $\sum_{j=0}^{m} n_j^4/3 + 3n_j^3 \approx \sum_{j=0}^{m} n_j^4/3$.

By noticing that the dominant term in the computational costs of the automatic differentiation and branch point series algorithms are the same as for the implicit differentiation algorithm we can conclude that the asymptotic cost of calculating the diagonal elements is $\sum_{j=0}^{m} n_j^4/3$ for any of these methods. In fact this is the same for computing the Taylor series of any matrix function where the dominant cost is from matrix multiplication.

The cost of calculating the remaining blocks in the strictly upper-triangular part of $F(T)$ also depends on the blocking; in [16, Sec. 9.4] the total cost of the Schur–Parlett method is estimated to lie between $28n^3$ and $n^4/3$ depending on the number of blocks with $n^4/3$ flops being required when all eigenvalues are blocked together.

6.2.5 Errors in Computation

In [9, Sec. 3] the norm of the error ΔF_{ij} in each block F_{ij} of $F = F(T)$ is shown to be bounded by $\|\Delta F_{ij}\|_F \leq \operatorname{sep}(T_{ii}, T_{jj})^{-1} \|B_{ij}\|_F$ where

$$B_{ij} = T_{ii}\Delta F_{ij} - \Delta F_{ij} T_{jj} = R_{ij} + \Delta F_{ii} T_{ij} - T_{ij}\Delta F_{jj} + \sum_{k=i+1}^{j-1} \Delta F_{ik} T_{kj} - T_{ik}\Delta F_{kj}$$

is the error incurred by evaluating the previous blocks, R_{ij} is the corresponding block in the residual matrix (not the same residual as in (4.9) however) given by

$$R = (F + \Delta F)T - T(F + \Delta F) = \Delta F T - T\Delta F,$$

and sep is defined by [12, Thm. 7.2.4]

$$\operatorname{sep}(T_{ii}, T_{jj}) = \min_{X \neq 0} \frac{\|T_{ii}X - XT_{jj}\|_F}{\|X\|_F}. \tag{6.5}$$

The function sep represents the separation between two matrices. Computing sep exactly where both matrices are $n \times n$ costs $O(n^4)$, however [16, Sec. 9.2] gives the following approximation to the inverse of sep

$$\operatorname{sep}(T_{ii}, T_{jj})^{-1} \approx \frac{1}{\min\{|\lambda - \mu| \colon \lambda \in \Lambda(T_{ii}), \mu \in \Lambda(T_{jj})\}}. \tag{6.6}$$

6.3 The Fréchet derivative $L_{W_b}(A, E)$

In Section 4.4.3 we found three expressions for the Kronecker form of $L_{W_b}(A, E)$ but found none of them suitable for direct computation. We therefore look to other methods which calculate or approximate $\|L_{W_b}(A)\|$ by trying to maximise $\frac{\|L_{W_b}(A,E)\|}{\|E\|}$ over a sufficient range of E.

6.3.1 Computing $L_{W_b}(A, E)$

Finite Difference Approximation to $L_{W_b}(A, E)$

The simplest way of approximating $L_{W_b}(A, E)$ is by finite differences, that is

$$L_{W_b}(A, E) \approx \frac{W_b(A + \delta E) - W_b(A)}{\delta}, \tag{6.7}$$

where $\delta > 0$ is small, (See [16, (3.22)]). This requires two evaluations of W_b and more accurately approximates $L_{W_b}(A, E)$ for small δ. However smaller δ increases the potential for subtractive cancellation so the choice of δ is necessarily a compromise. [16, Sec. 3.4] advises choosing $\delta = (\epsilon \|W_b(A)\| / \|E\|^2)^{1/2}$; that is the value we take here.

This requires two matrix evaluation of W_b, although $W_b(A)$ is evaluated repeatedly so can be reused after it is computed.

Power Series of $L_{W_b}(A, E)$

Another method is to use the power series of $L_{W_b}(A, E)$. For any function with a power series expansion we get an expression for the Fréchet derivative (See [1, Thm.

3.1]) which is valid for $\|A\| < r$ where r is the radius of convergence of the power series. For W_b and some $\alpha \in \mathbb{C}$ this is

$$L_{W_b}(A, E) = \sum_{k=1}^{\infty} \frac{W_b^{(k)}(\alpha)}{k!} \sum_{j=1}^{k} (A - \alpha I_n)^{j-1} E (A - \alpha I_n)^{k-j}. \tag{6.8}$$

We can evaluate this using the expression from [1, Thm. 3.2] given by

$$L_{W_b}(A, E) = \sum_{k=1}^{\infty} \frac{W_b^{(k)}(\alpha)}{k!} M_k, \tag{6.9}$$

where $M_k = L_{x^k}(A, E)$ is calculated from the recurrence

$$M_k = M_{m-1}(A - \alpha I_n) + (A - \alpha I_n)^{k-1} M_1, \quad M_1 = E.$$

Unfortunately as the Taylor series of W_b does not have an infinite radius of convergence this method is not viable.

Computation via W_b

The following theorem gives a way to directly compute the Fréchet derivative.

Theorem 6.3.1. Let $A, E \in \mathbb{C}^{n \times n}$ where $\|E\| = O(\|A\|\epsilon)$ and A has no eigenvalues on the closed negative axis, then

$$W_b \begin{pmatrix} A & E \\ 0 & A \end{pmatrix} = \begin{pmatrix} W_b(A) & L_{W_b}(A, E) \\ 0 & W_b(A) \end{pmatrix}.$$

Proof. See [16, (3.16)]. □

Computing the Fréchet derivative this way involves computing W_b for a matrix double the size of either A or E which multiplies the cost by a factor between 8 and 16, depending on the blocking. This cost could potentially be reduced if the method takes into account the special structure of the matrix, however due to time restrictions the author did not have time to investigate this.

Summary

We have seen three ways of computing the Fréchet derivative of which only two are viable for $L_{W_b}(A, E)$. Due to time restrictions we implement only the finite difference method (6.7) in order to estimate $L_{W_b}(A, E)$.

6.3.2 Computing the Condition Number

Recall that $\operatorname{vec}(L_{W_b}(A,E)) = K(A)\operatorname{vec}(E)$ and that $\|K(A)\|_2 = \|L(A)\|_F$. So we can evaluate $\|L(A)\|_F$ by evaluating $\|K(A)\|_2$ and to cut down on cost we can use low accuracy norm estimation as we are only really interested in the order of magnitude of $\|L(A)\|_F$. We look at one algorithm for finding this, another algorithm also exists for finding $\|K(A)\|_1$ which may also be of interest, but because we have already used the Frobenius norm in Section 4.5.2 we only consider $\|L(A)\|_F$.

Adjoint Operator $L^{\star}_{W_b}(A,E)$

The algorithm given in [1, Alg. 7.1] makes use of the the adjoint operator of $L_{W_b}(A,E)$ which is denoted $L^{\star}_{W_b}(A,E)$ (not to be confused with the conjugate transpose symbol).

The adjoint of a bounded linear operator B on $\mathbb{C}^{n\times n}$ is the unique bounded linear operator $B^{\star}$ such that $\langle B(X),Y\rangle = \langle X, B^{\star}(Y)\rangle$ for all $X,Y\in\mathbb{C}^{n\times n}$ and where $\langle\cdot\,,\cdot\rangle$ is an inner product on $\mathbb{C}^{n\times n}$. In this instance we take $\langle\cdot\,,\cdot\rangle$ to be $\langle X,Y\rangle = \operatorname{trace}(Y^*X)$ and $B(X) = L_{W_b}(A,X)$.

Theorem 6.3.2. The adjoint operator for $L_{W_b}(A,E)$ is $L^{\star}_{W_b}(A,E) = L_{W_{-b}}(A^*,E)$ where A has no eigenvalue on the branch cut of W_b.

Proof. We put the Fréchet derivative and its adjoint into their Kronecker forms, using the fact that the adjoint of a linear operator is linear (we put subscripts on the Kronecker form to denote the branch). To do this we use a property of the trace which relates it to vectorisation: $\operatorname{trace}(A^*B) = \operatorname{vec}(A)^*\operatorname{vec}(B)$.

$$\begin{aligned}
\langle L_{W_b}(A,E_1),E_2\rangle &= \operatorname{trace}(E_2^* L_{W_b}(A,E_1))\\
&= \operatorname{vec}(E_2)^*\operatorname{vec}(L_{W_b}(A,E_1))\\
&= \operatorname{vec}(E_2)^* K_b(A)\operatorname{vec}(E_1),\\
\langle E_1, L^{\star}_{W_b}(A,E_2)\rangle &= \operatorname{trace}((L^{\star}_{W_b}(A,E_2))^* E_1)\\
&= \operatorname{vec}(L^{\star}_{W_b}(A,E_2))^*\operatorname{vec}(E_1)\\
&= (G(A)\operatorname{vec}(E_2))^*\operatorname{vec}(E_1)\\
&= \operatorname{vec}(E_2)^* G_b(A)^*\operatorname{vec}(E_1).
\end{aligned}$$

By the definition of the adjoint $\langle L_{W_b}(A, E_1), E_2\rangle = \langle E_1, L^\star_{W_b}(A, E_2)\rangle$ for all small enough E_1 and E_2, so as $K_b(A)$ is nonsingular we conclude $K_b(A) = G_b(A)^*$.

By (4.8) and using the properties $(X^{-1})^* = (X^*)^{-1}$, $(X \otimes Y)^* = (X^* \otimes Y^*)$ and the property that for functions f whose restriction to the real axis is real (such as $\exp(x)$ and $\psi_1(x)$) $f(X)^* = f(X^*)$ (see Higham [16, Thm. 1.18]), we get

$$\begin{aligned}
(K_b(A)^*)^{-1} &= (K_b(A)^{-1})^* = (e^{W_b(A^T)} \otimes I_n + (I_n \otimes A)\psi_1(W_b(A^T) \oplus -W_b(A)))^* \\
&= (e^{W_b(A^T)} \otimes I_n)^* + \psi_1(W_b(A^T) \oplus -W_b(A))^*(I_n \otimes A)^* \\
&= e^{W_b(A^T)^*} \otimes I_n + \psi_1(W_b(A^T)^* \oplus -W_b(A)^*)(I_n \otimes A^*) \\
&= e^{W_{-b}((A^*)^T)} \otimes I_n + \psi_1(W_{-b}((A^*)^T) \oplus -W_{-b}(A^*))(I_n \otimes A^*) \\
&= (K_{-b}(A^*))^{-1}.
\end{aligned}$$

So $G_b(A) = K_{-b}(A^*)$, so $G_b(A)$ is the Kronecker form of $L_{W_{-b}}(A, E)$. As the Kronecker form of $L^\star_{W_b}(A, E_1)$ is unique we must have $L^\star_{W_b}(A, E) = L_{W_{-b}}(A^*, E)$. □

Algorithm 6.3.3 This function computes L_{W_b} or $L^\star_{W_b}$.

```
Require: $A, E \in \mathbb{C}^{n\times n}$, and $W = W_b(A)$ (optional)
  if user has not specified $W_b(A)$ then
    Calculate $W = W_b(A)$
  end if
  if we want the adjoint then
    $A = A^*$, $b = -b$ and $W = W^*$ (see Theorem 4.4.1)
  end if
  Let $\delta = (\epsilon\|W\|/\|E\|)^{1/2}$
  Compute $L(A, E) = (W_b(A + \delta E) - W)/\delta$
  return $L(A, E)$
```

Computing the Condition Number

The algorithm for computing $\|L(X)\|_F$ via $\|K(X)\|_2$ is implemented in The Matrix Function Toolbox [14] in the function `funm_condest_fro`. It requires a user-defined function to evaluate the Fréchet derivative[2] $L_{W_b}(A, E)$, this is given by Algorithm 6.3.3. We do not describe the method used in `funm_condest_fro` but refer the reader to [14] for the function and [16, Appendix D] for information on the algorithm.

[2]In fact it can evaluate $L_{W_b}(A, E)$ automatically using finite differences but we form the Fréchet derivative ourselves in order to take advantage of the fact that we need only compute $W_b(A)$ once.

Chapter 7

Numerical Experiments

7.1 Introduction

Here we present a series of experiments designed to test and demonstrate particular aspects of the algorithms. The main focus of these tests is the quality of the computed solutions, that is aspects such as accuracy, stability and conditioning. A secondary focus is efficiency and runtime but these are not as important to us as the quality.

7.1.1 Generating the Test Matrices

We obtain our test matrices either by calling the function `A = matrix(k,n)` implemented in the Matrix Function Toolbox [14] where $k = 1 : 52$ denotes the type of matrix and n denotes the size, or we generate matrices randomly. To do the latter we use one of two methods. By the first method (referred to as Method One) we create a random diagonal matrix D with (either real or complex) entries chosen so that $\rho(D) \leq r$ for some $r > 0$ and then transform D using a random (real or complex) similarity transform P to get $A = P^{-1}DP$. For the second method (referred to as Method Two) we create a full matrix A with random (real or complex) entries chosen such that $|A| \leq r$ for some $r > 0$. All random numbers are chosen from the uniform distribution.

7.1.2 The Terminology

We use the following terminology: if $A \in \mathbb{C}^{n\times n}$ then $W = W_b(A)$ denotes the exact solution while $\widetilde{W} = W_b^{\mathcal{I}}(A)$ and $\widetilde{W} = W_b^{\mathcal{A}}(A)$ denote the approximate solution computed via the Schur–Parlett–Taylor–implicit algorithm and Schur–Parlett–Taylor–auto algorithm respectively. Given exact and computed solutions W and $\widetilde{W}$ we denote the absolute forward error $\Delta W = \|W - \widetilde{W}\|_F$ and the relative forward error $\Delta_{\mathrm{rel}} W = \Delta W/\|W\|_F$. The norm of the residual is expressed by $R(\widetilde{W}, A)$ and the norm of the normalised residual by $\overline{R}(\widetilde{W}, A)$ (See Section 4.5.1). We denote the absolute and relative condition numbers by $c_{\mathrm{abs}}(W_b, A)$ and $c_{W_b}(A)$ respectively (See Sections 4.4.3 and 6.3). In all cases the Frobenius norm is used.

7.1.3 Interpreting the results

Having established these terms we can use the tools we have developed. We cannot calculate $\Delta_{\mathrm{rel}} W$ directly but (4.11) gives an a posteriori lower bound for the absolute forward error and $c_{W_b}(A)\overline{R}(\widetilde{W}, A)$ is an upper bound, so by making these into bounds for the relative error we obtain an interval in which $\Delta_{\mathrm{rel}} W$ must lie; we denote the lower bound $\Delta_{\mathrm{rel}}^{\mathbb{L}} W$ and the upper bound $\Delta_{\mathrm{rel}}^{\mathbb{U}} W$. The tighter this interval is the better estimate we have for the relative forward error.

If the algorithm is stable in computing $W_b(A)$ then we expect $\overline{R}(\widetilde{W}, A)$ to be of order ϵ (machine precision) and consequently $\Delta_{\mathrm{rel}} W \lesssim c_{W_b}(A)\epsilon$. We measure the stability of the algorithms empirically by how much computed solutions tend to satisfy or fail this condition.

7.2 The Tests

All tests were performed in MATLAB 7.1 on a 1.73GHz dual processor system with 1024Mb of memory running Windows Vista.

Experiment 1. The first experiment computes $W_0^{\mathcal{I}}(A)$ and $W_0^{\mathcal{A}}(A)$ using 60 matrices $A_i \in \mathbb{C}^{n\times n}$ randomly generated via Method One, where $n \leq 30$ and $\rho(A_i) \leq 2$. The spectral radius is kept bounded to ensure a wide variety of blockings are used. The blocking parameter δ is kept constant at 0.1 and the buffer is $\beta = 0.1$. This test is designed to show the method is stable for 'typical random matrices' of different sizes under a diverse array of blockings.

For each A_i if $W_i = W_0(A_i)$ then we calculate $W_0^{\mathcal{I}}(A_i)$ and $W_0^{\mathcal{A}}(A_i)$, in Figure 7.1 the results are shown only for $\widetilde{W}_i = W_0^{\mathcal{I}}(A_i)$ as the results for both algorithms are qualitatively the same. For each A_i the values of: $\Delta_{\mathrm{rel}}^{\mathbb{L}} W_i$, $c_{W_0}(A_i)\epsilon$, $\overline{R}(\widetilde{W}_i, A_i)$ and $\Delta_{\mathrm{rel}}^{\mathbb{U}} W_i$ are shown in the top figure. Underneath is the runtime for each matrix plotted with the matrix size and the number of blocks used.

From the top figure we see that $\Delta_{\mathrm{rel}}^{\mathbb{L}} W_i$ is never more than a couple of orders of magnitude greater than $c_{W_0}(A_i)\epsilon$. As the relative forward error is less than or equal to this it implies the methods are fairly stable for this random sample of matrices. In the plot below we can see how the runtime of the algorithm for each matrix is strongly related to the order of the matrix. We would expect there to be a relationship between the number of blocks used and the runtime (recall from Section 6.2.4 the computation cost varies between $28n^3$ and $n^4/3$ depending on the blocking used) but no such relationship is apparent from the plots.

The results for the Schur–Parlett–Taylor–auto algorithm are qualitatively the same so we do not show them as plots. Instead Figure 7.2 shows the differences in the results. In the upper figure the runtimes are compared. In general it appears that Schur–Parlett–Taylor–auto has a slight lead over Schur–Parlett–Taylor–implicit and note how the difference is more prominent for the lowest runtimes. This is due to the lower order terms in the computational cost (see Table 5.1). The lower plot shows the norms of the differences of the solutions, that is $\|W_0^{\mathcal{I}}(A_i) - W_0^{\mathcal{A}}(A_i)\|$.

Experiment 2. This experiment is a repetition of the previous experiment but using random matrices generated via Method Two with entries less than one in absolute value. As a consequence the eigenvalues are less tightly bounded and most are blocked into blocks of size one. The purpose of this test is to see whether matrices generated in this way exhibit behaviour different from in Experiment 1 but no significant qualitative difference was observed.

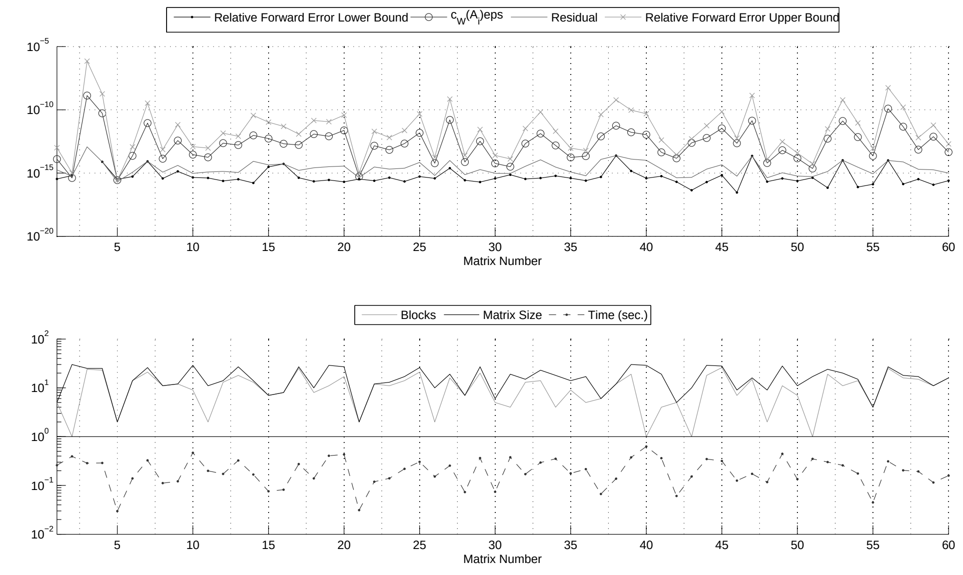

Figure 7.1: In the upper plot data pertaining to the quality of the solution is shown for each matrix $\widetilde{W_i} = W_0^{\mathcal{I}}(A_i)$. In the lower plot the order of each matrix is plotted with the runtime and the number of blocks used in the blocking. Although these plot are for of the Schur–Parlett–Taylor–implicit algorithm the data for the Schur–Parlett–Taylor–auto algorithm is qualitatively the same.

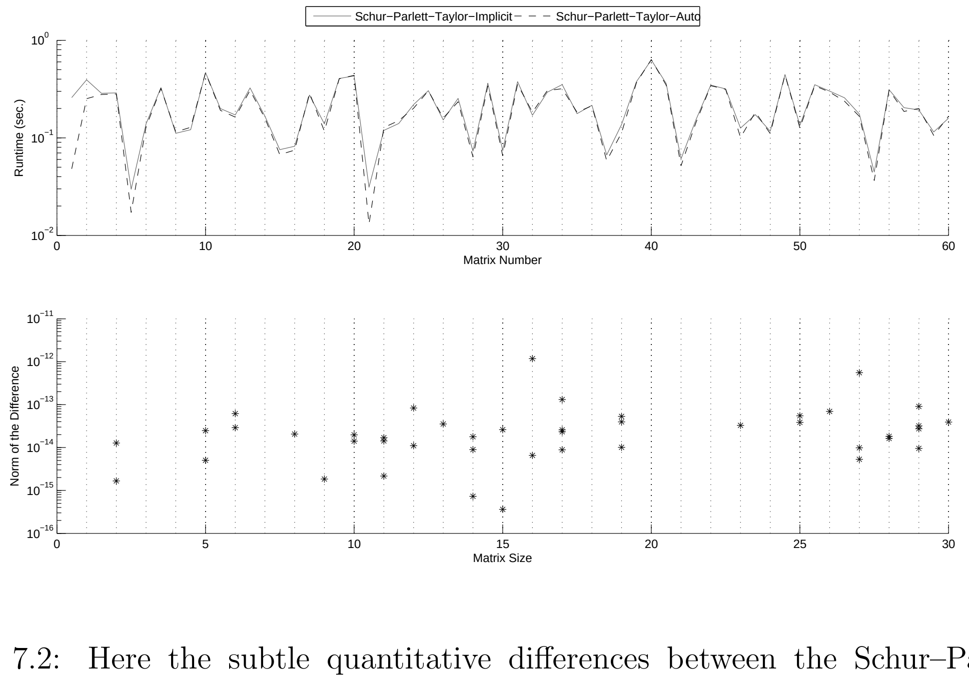

Figure 7.2: Here the subtle quantitative differences between the Schur–Parlett–Taylor–implicit and Schur–Parlett–Taylor–auto algorithms are shown for Experiment 1. The upper figure shows the plots of the runtimes for each matrix. In the lower plot the norm of the difference between the corresponding solutions for each matrix is plotted against the matrix size.

Experiment 3. This experiment is intended to show the effect of different buffer sizes, we use $\delta = 0.1$ and restrict ourselves to the principal branch. We generate random matrices $A_i \in \mathbb{C}^{n\times n}$ via Method One where $n \leq 20$ and $\rho(A_i) \leq 1$. In Figure 7.3 $\overline{R}(\widetilde{W}, A_i)$ is plotted against β for twenty matrices (each matrix has its own line plot in grey) and the maximum residual of all the matrices is plotted in bold against β. The experiment was performed using 64 equally spaced values of β ranging between zero and e^{-1} for each matrix.

Having no buffer can clearly be catastrophic for some matrices (with others it appears to have no effect) but Figure 7.3 seems to show that for some matrices, accuracy suffers greatly for all $\beta < 0.1$ but with $\beta > 0.1$ the buffer size makes little difference. By inspecting the plots we can see that the optimal buffer appears to be around $\beta = 0.2$.

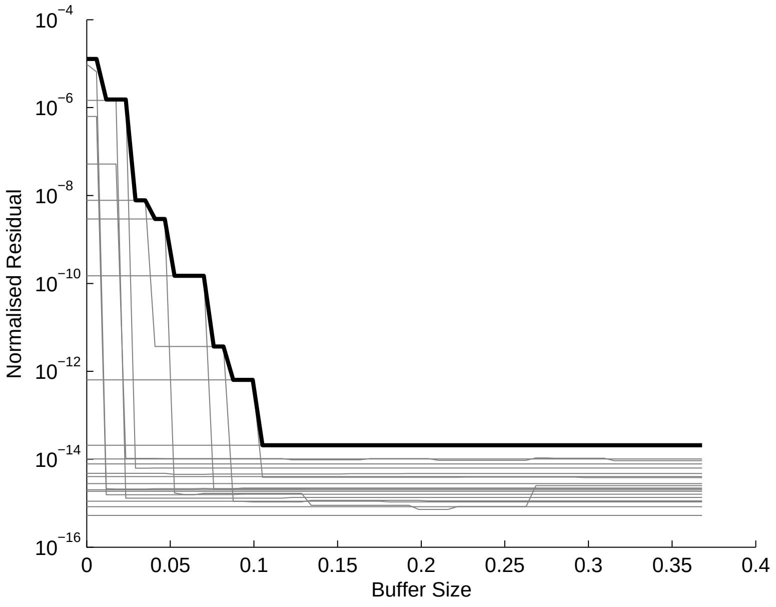

Figure 7.3: Each grey line shows the normalised residual for one of the twenty matrices in Experiment 3 plotted against different buffer sizes β. The bold black line highlights the largest residual at each buffer size.

Experiment 4. Here we look at the effects of different blocking parameters, where the test matrices are generated via Method One. We use $0.1 \leq \delta \leq 1.0$ and two buffer sizes $\beta = 0.2$ and $\beta = 0.1$. Figure 7.4 shows $\overline{R}(\widetilde{W}, A_i)$ plotted against δ for the each matrix calculated with the Schur–Parlett–Taylor–implicit algorithm with

$\beta = 0.2$. This figure is typical in that it shows no overall relationship between δ and $\overline{R}(\widetilde{W}, A_i)$ though there is great variation in accuracy, there is no pattern common to all matrices tested.

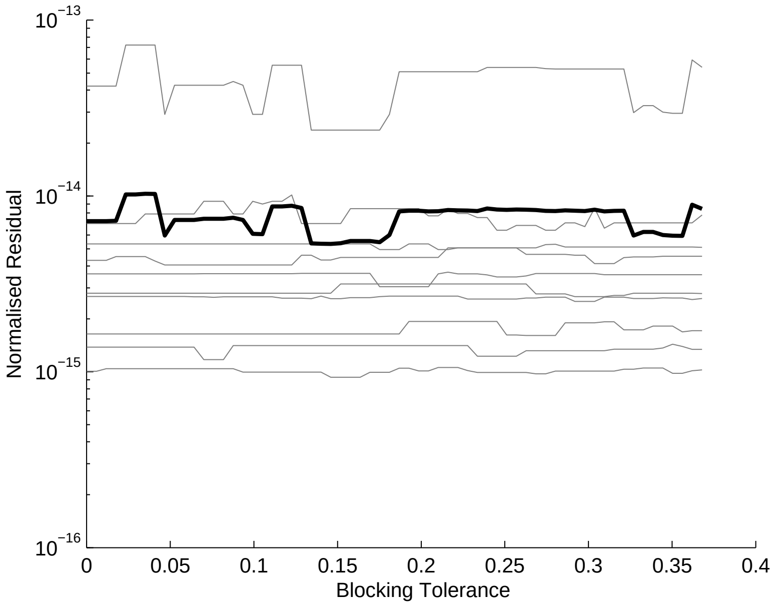

Figure 7.4: Each grey line shows the normalised residual for a different matrix in Experiment 4 plotted against the blocking tolerance δ. The bold black line shows the arithmetic mean of the plots.

Experiment 5. Now we examine $W_0^{\mathcal{I}}(A_i)$ and $W_0^{\mathcal{A}}(A_i)$ for a number of specifically selected test matrices. We use $A_i \in \mathbb{C}^{10\times 10}$ where A_i is generated from `matrix(k,10)` for $k = 1 : 52$ which is implemented in the Matrix Function Toolbox (see [14]). We use $\beta = 0.2$ and repeat the experiments for $\delta = 0.1$, 0.5 and 1.0.

For most of the test matrices the solution was computed accurately by both algorithms, Figure 7.5 shows plots of the results for the Schur–Parlett–Taylor–implicit algorithm with $\delta = 0.1$. However some matrices did cause problems with one or both algorithms. In each case either W_b appeared ill-conditioned at A_i or the algorithms proved unstable. In some cases the lower error bound for the relative forward error is higher than the upper bound, in these cases it is more likely that the upper bound is inaccurate as it relies on the condition number of W where as the lower bound relies on the condition number of exp which is more accurate to calculate due to exp being holomorphic. Unfortunately we cannot be more precise about this as we have not had time to derive stability results for the divided difference Fréchet derivative estimate. In Figure 7.5 the matrices which have eigenvalues on the branch cut $(-\infty, -e^{-1})$ are marked with a vertical bold dashed line at the bottom of the figure. It is interesting that most of these, but not all, correspond with very large condition number estimates. This suggests that these condition number estimates may not be reliable. We list some of the most "difficult" matrices in Table 7.1 and briefly discuss two of them.

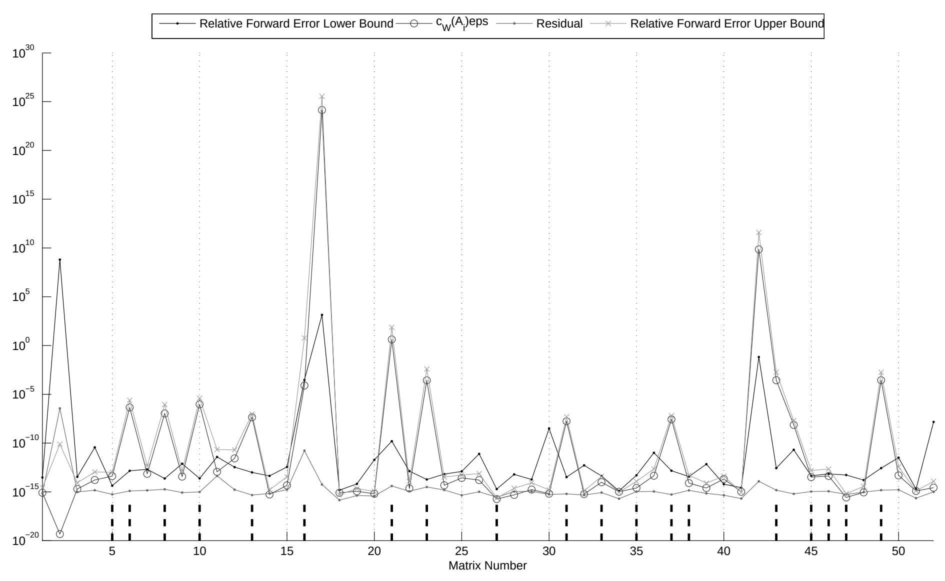

Figure 7.5: Shows solution quality data for the test matrices in Experiment 5 computed by Schur–Parlett–Taylor–implicit with $\delta = 0.1$. The dashed lines at the bottom indicate that the matrix had eigenvalues on $(-\infty, -e^{-1})$.

Number	Name	Nature of Problem
2	Chebyshev Spectral	Algorithm unstable
4	Chow	Algorithm unstable
16	Involutory	Ill-Conditioned
17	Hankel with Factorial Elements	Ill-Conditioned
21	Krylov	Ill-Conditioned
30	Prolate	Algorithm unstable
42	Inverse Hilbert	Ill-Conditioned
43/49	Magic	Ill-Conditioned
52	Vandermonde	Algorithm unstable

Table 7.1: Here the matrices which cause accuracy problems in the Schur–Parlett–Taylor–implicit and/or Schur–Parlett–Taylor–auto algorithms are listed along with the reason.

The Chebyshev spectral differentiation matrix. This matrix caused massive instability in both algorithms. It appears to have a very small relative condition number of the order 10^{-4} yet suffered a relative forward error bound of at least 10^7 (this rises to 10^8 for $\delta = 0.1$) and this is one instance of the upper bound being smaller than the lower bound.

The matrix $A \in \mathbb{C}^{n\times n}$ has a number of interesting properties: its Jordan form consists of a single Jordan block with zero eigenvalue, hence it is nilpotent with $A^n = 0$. Despite all eigenvalues begin zero the computed Schur form is nonsingular with eigenvalues clustered in a circle of radius about 0.2 around the origin. It is possible that the inaccurate computation of the Schur form is the source of the instability.

The Chow matrix. This matrix, though well conditioned appears to be very sensitive to changes in the blocking parameter δ. When the solution is computed using a variety of blocking tolerances between 0 and 1.4 it is revealed that the algorithms are highly unstable for $0.27 \leq \delta < 1.0$ and apparently stable elsewhere.

Experiment 6. Now we examine $W_b^{\mathcal{I}}(A_i)$ and $W_b^{\mathcal{A}}(A_i)$ for nonprincipal branches. The purpose of this test is two-fold: firstly to show that the algorithms perform adequately for nonprincipal branches, and secondly to determine if and how the quality of the solution is affected by taking nonprincipal branches. We use $\beta = 0.2$ and $\delta = 0.1$ and run the test on 20 random complex 10×10 matrices generated by Method Two. We compute the Lambert W function for branches $b = -7\colon 7$.

In Figure 7.6 plots are given for $\overline{R}(\widetilde{W}, A_i)$, $c_{W_b}(A_i)$, $\Delta^{\mathbb{L}}_{\text{rel}}W_i$ and $\Delta^{\mathbb{U}}_{\text{rel}}W_i$ for each matrix A_i using both Schur–Parlett–Taylor–implicit and Schur–Parlett–Taylor–auto. The upper and lower error bounds are shown on the same graph with the solid lines representing the lower bounds and the dashed ones representing the upper bounds. In each figure a bold line representing the mean value for all A_i is also plotted. From the plots we can see that the normalised residual is lowest for the principal branch and increases quite prominently when one moves to the $b = -1$ or $b = 1$ branch, however for branches less than -2 or higher than 2 it appears to increase linearly with $|b|$. The error bounds remain more or less constant across all branches, particularly the

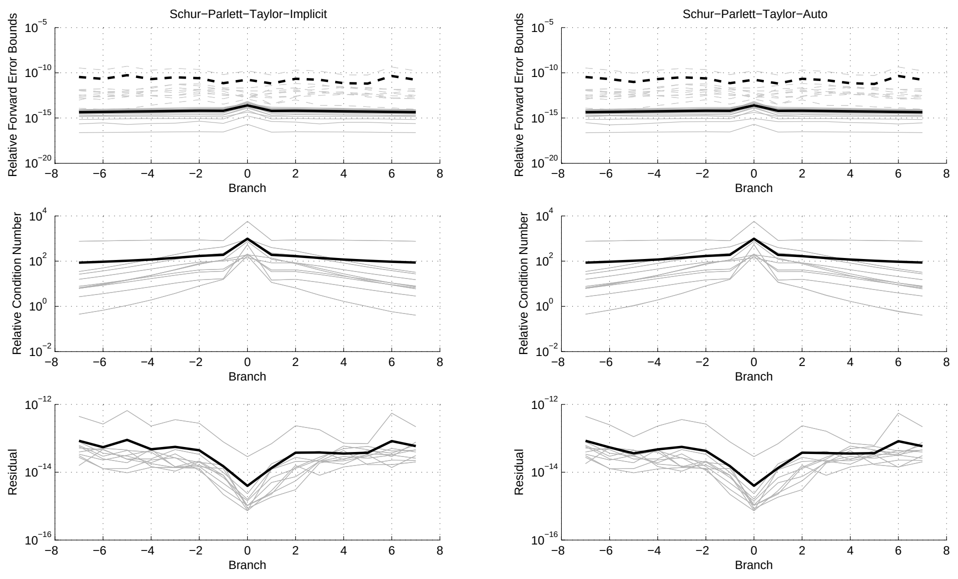

Figure 7.6: In each figure the grey lines represent the test matrices A_i, and the black bold line is the arithmetic mean of the grey lines. These plots show how the quality of the solution via both algorithms changes as we evaluate $W_b(A_i)$ using different branches. The figures on the left refer to Schur–Parlett–Taylor–implicit and the ones on the right to Schur–Parlett–Taylor–auto. In the two top figures the solid lines represent $\Delta^{\mathbb{L}}_{\text{rel}} W_i$ and the dashed lines $\Delta^{\mathbb{U}}_{\text{rel}} W_i$

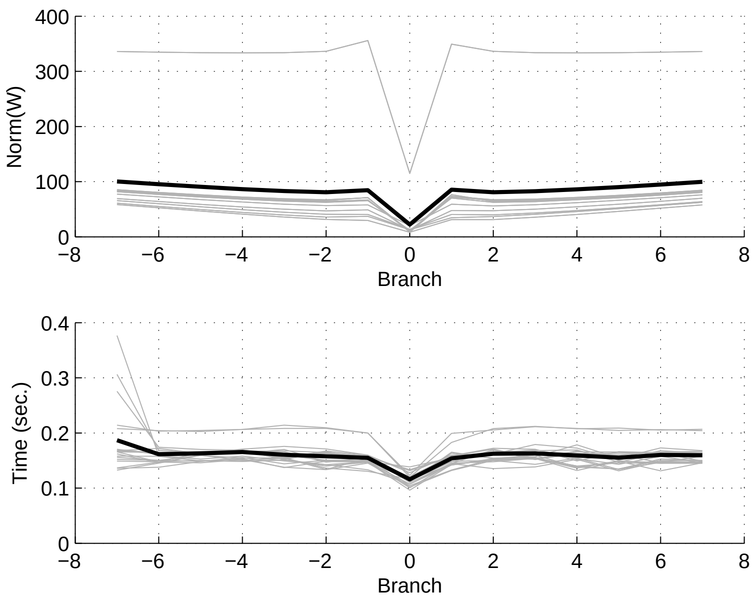

Figure 7.7: The upper plots show the norms of the solutions obtained for the matrices in Experiment 6 and the lower plot shows the runtimes of the algorithms. Both figures were qualitatively the same for both Schur–Parlett–Taylor–implicit and Schur–Parlett–Taylor–auto. Once again the bold black line is the arithmetic mean.

the lower bound. The relative condition number appears to attain its peak on the principal branch and decay as $|b|$ increases.

We can get some insight into this behaviour if we consider how the norm of $W_b(A_i)$ changes with b. If we plot the Frobenius norms as we have done in Figure 7.7 we see the same sort of pattern as we observe in the normalised residual and the inverse pattern of the relative condition number. The norm appears to vary linearly with $|b|$ until $b = 0$ at which there is a sudden drop. Interestingly we see for all 20 test matrices the norms for $|b| = 2$ are slightly smaller than the ones for $|b| = 1$. The general pattern is also reflected in the runtimes of the algorithms[1].

Experiment 7. Now we modify the algorithm to allow different branches to be used in different blocks. We generate 50 random complex matrices $A_i \in \mathbb{C}^{n\times n}$, using Method One where $n \leq 30$ and $\rho(A_i) \leq 2$. For each matrix A_i, once the blocking has been performed, a branch b_j is chosen for each block such that $|b_j| \leq r_i$ where $5 \leq r_i \leq 25$ is a positive integer associated with each matrix. The purpose of this test is to show if and how the quality of the solution is affected by taking different branches for different blocks. As we are using multiple branches we denote the Lambert W function $W_{\mathbf{b}}(A_i)$ where $\mathbf{b}$ is an integer vector.

In Figure 7.8 we display the results for the Schur–Parlett–Taylor–implicit algorithm; as we can see: the normalised residual, the relative condition number and the relative forward error bounds vary much more than in Experiment 1 and the gulf between $\Delta^{\mathbb{U}}_{\mathrm{rel}}W_i$ and $\Delta^{\mathbb{L}}_{\mathrm{rel}}W_i$ is wider. The upper error bound is usually not much higher than $c_{W_{\mathbf{b}}}(A_i)\epsilon$, however due to the breadth between the upper and lower error bounds it is difficult to tell how much instability there is. For the most part however the relative forward error cannot be much more than a couple of orders of magnitude higher than $c_{W_{\mathbf{b}}}(A_i)\epsilon$. The lower figure shows plots comparing the matrix size, the number of blocks used and the sum of the absolute values of the branches, that is $\sum_{j=1}^{m} |b_j|$ where there are m blocks and b_j is the branch used to evaluate the jth block. When comparing both figures we see that the lowest upper error bounds occur

[1]Note that branch $b = -7$ does not fit this pattern, however as it was computed first it is possible that this is due to the fact that the matrix had yet to be loaded into cache memory.

when only a few blocks are used. Note how matrices 5, 7, 22, 38 and 47 all have small upper error bounds but are also blocked using only one block.

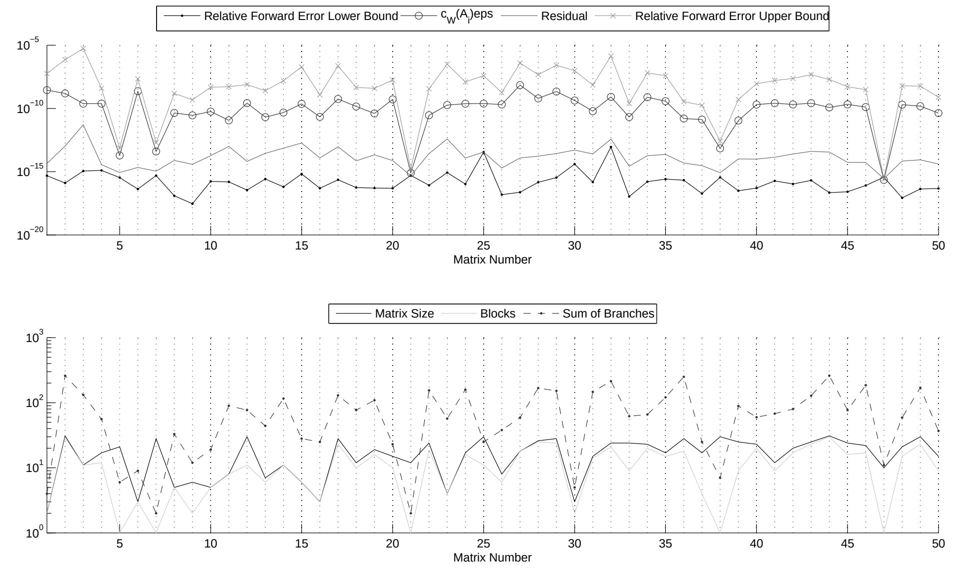

Figure 7.8: The upper plot shows the same information as in Figure 7.1 for the matrices in Experiment 7 except that the algorithms now choose branches randomly for each block. The lower plot shows for each matrix: the matrix size, the number of blocks and the sum of the absolute values of the branches used (i.e. 'Branches' equals $\sum_{j=1}^{m} |b_j|$ where m is the number of blocks used).

Experiment 8. Here we generate four classes of matrices designed to produce interesting results. The first class of matrices $A_1 \in \mathbb{C}^{25\times 25}$ is generated by Method One except that only twenty eigenvalues of A_1 are chosen randomly and the remaining five are equal to $-e^{-1}$ and semi-simple. So A_1 is derogatory but nondefective. The second class $A_2 \in \mathbb{C}^{25\times 25}$ is the same except the $-e^{-1}$ eigenvalue is not semi-simple and appears in a Jordan block of size five. Hence A_2 is defective but nonderogatory. The third class $A_3 \in \mathbb{C}^{25\times 25}$ has a Jordan form of five equally sized Jordan blocks with distinct eigenvalues, none of which are equal to $-e^{-1}$. A_3 is defective and nonderogatory. The forth class $A_4 \in \mathbb{C}^{25\times 25}$ is the same as A_3 but with only four distinct eigenvalues. So A_4 is defective and derogatory. We test ten matrices from each class and give the mean results in Table 7.2.

There is very little difference between the results obtained from Schur–Parlett–Taylor–implicit and those obtained from Schur–Parlett–Taylor–auto. Note that A_1

does not satisfy the conditions of Theorem 6.3.2 or Theorem 4.4.11 because $-e^{-1} \in \Lambda(A_1)$ so we expect the computed condition number to be nonsense (which it appears to be). Despite this the normalised residual does not appear much greater than ϵ, so the algorithms seem stable. The evaluation of solutions for A_2 have completely failed, verifying the observation in Section 4.4.1 that $W_b(A)$ does not exist if $\mathrm{idx}_A(-e^{-1}) > 1$. In both A_3 and A_4 the algorithms exhibit moderate instability and defective matrices appear to be more ill-conditioned than nondefective ones as can be seen by comparing the results with those from Experiment 1.

Schur–Parlett–Taylor–implicit				
Matrix	Error Bounds		$\overline{R}(\widetilde{W_i}, A_i)$	$c_{W_0}(A_i)$
A_1	9.4×10^{-16}	3.1×10^{-9}	3.75×10^{-15}	1.6×10^{-10}
A_2	N/A	N/A	N/A	N/A
A_3	7.7×10^{-11}	6.3×10^{-6}	9.9×10^{-11}	1.1×10^{5}
A_4	2.3×10^{-12}	3.6×10^{-7}	5.7×10^{-12}	2.5×10^{6}

Schur–Parlett–Taylor–auto				
Matrix	Error Bounds		$\overline{R}(\widetilde{W_i}, A_i)$	$c_{W_0}(A_i)$
A_1	7.7×10^{-16}	3.2×10^{-9}	3.74×10^{-15}	1.6×10^{-10}
A_2	N/A	N/A	N/A	N/A
A_3	1.7×10^{-10}	1.7×10^{-5}	2.1×10^{-10}	1.1×10^{5}
A_4	5.8×10^{-12}	3.8×10^{-7}	1.1×10^{-11}	2.5×10^{6}

Table 7.2: Shows the solution quality results in Experiment 8 for the Schur–Parlett–Taylor–implicit and Schur–Parlett–Taylor–auto algorithms.

7.3 Summary

Matrices with entries or eigenvalues generated randomly are almost certainly nondefective and nonderogatory so the results from Experiments 1, 2 and 7 show that the algorithms are stable for such matrices. Experiments 3 and 4 show that the correct choice of parameters can have a significant bearing on the quality of the solution, the optimal β seems to be around 0.2 but the optimal δ depends on the the matrix begin evaluated. Experiments 5 and 8 show the existence of matrices for which the method is not stable and matrices (particularly those with eigenvalues on the branch cuts) for which the condition number cannot be accurately estimated.

Chapter 8

Concluding Remarks

Algorithm 6.2.3 in conjunction with Algorithms 5.3.1, 5.3.2 and 5.4.3 has been shown in Chapter 7 to give reliable results for the Lambert W function of most matrices. We have also derived the tools necessary to compute the condition number of W_b for matrices whose eigenvalues do not appear in the branch cuts and derived upper and lower a posteriori bounds for the forward error. These tools allow one to compute the Lambert W function of a square matrix as well as information pertaining to the quality of the solution.

Because the Lambert W function does not have an infinite radius of convergence adapting Algorithm 6.1.1 to produce a valid blocking is a nontrivial task. The method presented in Section 6.2.2 ensures that it will always be possible to calculate the atomic blocks, though this is potentially at the expense of accuracy. The blocking strategy suggested in Section 6.2.2 is not the only possible strategy; while we have used one parameter β to control the radius of the buffers around both $-e^{-1}$ and the origin[1], we could use two different radii and determine the best values by experimentation. The optimal buffer size was found by inspection to be 0.2 although future algorithms could select a buffer size intelligently to ensure two close eigenvalues are not on opposite sides, thereby avoiding any potential loss of accuracy.

For computing the atomic blocks we have derived two types of power series, each of which is designed to be used in different circumstances. If the eigenvalues are

[1]This does not apply on the principal branch when Schur–Parlett–Taylor–implicit is used.

close to or scattered around $-e^{-1}$ then the branch point series should be used; in all other cases a Taylor expansion should be used. We have looked at two methods for computing the coefficients of a Taylor series: implicit differentiation and automatic differentiation. When the atomic block has eigenvalues close to or scattered around the origin, implicit differentiation is the only viable method; when this is not the case either method can be used. In terms of accuracy we have shown there is not much difference between the two and although a priori error bounds have been derived in Section 2.4.4 they are not sufficiently tight to tell us which method is more stable. In Table 5.1 we see the computational cost of automatic differentiation is slightly less than for implicit differentiation, even though asymptotically the costs are the same. This suggests that when the eigenvalues of the atomic block are not too close to the origin automatic differentiation is the most efficient choice for evaluation, particularly when the atomic block is small.

Throughout the investigation we have found several open questions which could form the basis for future work. For instance, sufficient conditions for the existence of the Fréchet derivative have been found, but necessary conditions would complete the picture. The condition number has been computed using divided differences but no error bounds have been found for this method; we also saw two other methods for the Fréchet derivative but they were not investigated here. The implicit differentiation and automatic differentiation algorithms could be further investigated to see if tighter error bounds can be derived.

The author believes the methods in this thesis give a robust way to evaluate the Lambert W function of a matrix, particularly nondefective matrices and ones with no eigenvalues on $(-\infty, -e^{-1})$ or $(-\infty, 0)$ (depending on the branch). If the method is found to be unstable for a particular matrix or class of matrices, then a new blocking strategy should be tried, either by repeating the evaluation with a different set of parameters or developing a new blocking algorithm specific to the problem.

This method currently has no rivals to compete with, the author hopes this method will prove useful but also that it will stimulate the discovery of other methods.

Appendix A

MATLAB Code

A.1 lambertwdiff_polys

```
function lambertwdiff_polys(to, doscaling)
%LAMBERTWDIFF_POLYS Returns the dth derivative of W_b
global lambertw_diff_globalpolyspace;
global lambertw_diff_globalpolyinfo;

if(nargin < 2)    doscaling = false;    end

if(to > lambertw_diff_globalpolyinfo.MaxDerivatives)
    error('Maximum number of allowed derivatives exceeded.');   end

from = lambertw_diff_globalpolyinfo.CurDerivatives;
lambertw_diff_globalpolyinfo.CurDerivatives = to;

% Returns a series of polynomials p_i whose degrees are deg(p_i) = i
for(j = from:to)
    lambertw_diff_globalpolyspace(j + 1, 1) =...
        (1 - 3*j)*lambertw_diff_globalpolyspace(j, 1) +...
        lambertw_diff_globalpolyspace(j, 2);
    for(k = 1:j - 1)
        lambertw_diff_globalpolyspace(j + 1, k + 1) =...
            (1 - 3*j + k)*lambertw_diff_globalpolyspace(j, k + 1)...
            + (k + 1)*lambertw_diff_globalpolyspace(j,k + 2)...
            - j*lambertw_diff_globalpolyspace(j,k);
    end
    lambertw_diff_globalpolyspace(j + 1, j + 1) =...
        -j*lambertw_diff_globalpolyspace(j,j);

    if(doscaling)
        lambertw_diff_globalpolyspace(j + 1,:)...
            = lambertw_diff_globalpolyspace(j + 1,:)/(j + 0);
    end
end
```

A.2 lambertwdiff

```
function w = lambertwdiff(z, d, b, tfDoLW )
%LAMBERTWDIFF Returns the dth derivative of W_b
global lambertw_diff_globalpolyspace;
global lambertw_diff_globalpolyinfo;

if(lambertw_diff_globalpolyinfo.MaxDerivatives < d)
    error('MATLAB:lambertwdiff:MaxExceeded', 'Not Enough Derivatives');
elseif(lambertw_diff_globalpolyinfo.CurDerivatives < d)
    lambertwdiff_polys(d);
end

if(nargin < 2)    d = 1;            end
if(nargin < 3)    b = 0;            end
if(nargin < 4)    tfDoLW = true;    end
lambw = z;
if(tfDoLW)  lambw = lambertw(b, z); end

if(d == 0)    w = lambw;    return; end

if(min(isfinite(double(lambertw_diff_globalpolyspace(d,d:-1:1)))) == 0)
    % Cannot calculate derivative
    warning('MATLAB:lambertwdiff:Overflow', 'Derivative Overflow.');
end

p = polyval(double(lambertw_diff_globalpolyspace(d,d:-1:1)), lambw);
w = exp(-d.*lambw) .* p ./((1 + lambw).^(2*d - 1));
```

A.3 wblocking

```
function m = wblocking(A, delta, buffer, branch, show)
%WBLOCKING  Produce blocking pattern for Lambert W Matrix Function.
%   M = WBLOCKING(A, DELTA, BUFFER, BRANCH, SHOW) accepts an upper
%   triangular matrix A and produces a blocking pattern, specified by the
%   vector M, for the block Parlett recurrence.
%   M(i) is the index of the block into which A(i,i) should be placed,
%   for i=1:LENGTH(A).
%   DELTA is a gap parameter (default 0.1) used to determine the blocking.
%   BUFFER is the size of the buffer about the branch points.

a = diag(A); n = length(a);
m = zeros(1,n); maxM = 0;

if nargin < 2 || isempty(delta), delta = 0.1; end
if nargin < 3 || isempty(buffer), buffer = 0.1; end
if nargin < 4 || isempty(branch), branch = 0; end

for i = 1:n
    if m(i) == 0
        m(i) = maxM + 1; % If a(i) hasn't been assigned to a set
        maxM = maxM + 1; % then make a new set and assign a(i) to it.
    end

    for j = i+1:n
```

```
        if m(i) ~= m(j)    % If a(i) and a(j) are not in same set.
            if abs(a(i) - a(j)) <= delta
                if m(j) == 0
                    if(test_lambda(GetSet(i),[j]))
                        m(j) = m(i); % If a(j) hasn‘t been assigned to a
                                     % set, assign it to the same set as
                                     % a(i).
                    end
                elseif(test_lambda(GetSet(i),GetSet(j)))
                    p = max(m(i),m(j)); q = min(m(i),m(j));
                    m(m==p) = q; % If a(j) has been assigned to a set
                                 % place all the elements in the set
                                 % containing a(j) into the set
                                 % containing a(i) (or vice versa).
                    m(m>p) = m(m>p) -1;
                    maxM = maxM - 1;
                                 % Tidying up. As we have deleted set
                                 % p we reduce the index of the sets
                                 % > p by 1.
                end
            end
        end
    end
end

if(show)
    gcf;    hold on;    grid on;
    xlabel('Real'); ylabel('Imag');
    for(i = 1:n)
        t = GetSet(i); numeigs = length(t);
        if(numeigs > 1)
            for(j = 1:numeigs)
                line(...
                    real([a(i), a(t(j))]), imag([a(i), a(t(j))]),...
                    'Color',0.8*[1, 1, 1], 'LineStyle', '-' );
            end
            [meaneig, eigrad] = GetSetStats(t);
            if(eigrad > 0)
                rectangle('Position',...
                    [real(meaneig) - eigrad, imag(meaneig) - eigrad,...
                    eigrad*2, eigrad*2],...
                    'Curvature',[1,1],'LineStyle','-' );
            end
        end
    end
    rectangle(...
        'Position',[-exp(-1) - buffer, - buffer, buffer*2, buffer*2],...
        'Curvature',[1,1],'LineStyle','--','EdgeColor','black' );
    if(branch ~= 0)
       rectangle(...
           'Position',[ - buffer, - buffer, buffer*2, buffer*2],...
           'Curvature',[1,1],'LineStyle','--','EdgeColor','blue' );
    end
    scatter(real(a),imag(a),'Marker','.','MarkerEdgeColor','black');
    scatter([0 -exp(-1)],[0 0],'Marker','*','MarkerEdgeColor','black');
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function tf = test_lambda(c,v)
%TEST_LAMBDA Returns true if c and v can be amalgamated.
    %% This function fuses the sets c and v and assesses whether it is
    %% a viable blocking

    %% If the block violates any of the rules then this variable will
    %% be set to false
    tf = true;

    % we must test to see whether this set violates any of the rules
    newblock = [c,v];

    %% Compute the mean and radius
    [meaneig, eigrad] = GetSetStats(newblock);

    %%  This code is applicable only to b = 0, -1, 1    %%
    if(branch == 0 || branch == -1 || branch == 1)
        %% If the mean is inside the disc about -e^{-1}
        if(abs(meaneig + exp(-1)) < buffer)
            %% If any eigenvalue causes the disc to go over the edge of
            %% the buffer then reject the blocking
            if(abs(meaneig + exp(-1)) + eigrad >= buffer)
                tf = false;
            end
        else  %% Otherwise it is on the outside (or on it exactly)
              %% If any eigenvalue causes the disc to go inside the
              %% buffer then reject the blocking
            if(abs(meaneig + exp(-1)) - eigrad <= buffer)
                tf = false;
            end
        end
    end

    %%  This code is applicable only to nonzero branches  %%
    if(branch ~= 0)
        if(abs(meaneig) < buffer)
            if(abs(meaneig) + eigrad >= buffer)
                tf = false;
            end
        else
            if(abs(meaneig) - eigrad <= buffer)
                tf = false;
            end
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function idxset = GetSet(g)
%GETSET Returns eigenvales blocked with g.
    setsize = 0;
    for(h = 1:n)
        if(m(h) == m(g))
            idxset(setsize + 1) = h;
            setsize = setsize + 1;
        end
        end
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [meanE, Erad] = GetSetStats(S)
    %GETSETSTATS Returns Mean and Radius of the Eigenvalues in S.
        meanE = 0;
        numE = length(S);
        for(k = 1:numE)
            meanE = meanE + a(S(k));
        end
        meanE = meanE/numE;

        Erad = 0;
        for(k = 1:numE)
            Erad = max(Erad, abs(meanE - a(S(k))));
        end
    end
end
```

A.4 lambertwm_atom

```
function [F, terms] = lambertwm_atom(T, b, method,...
    buffer, tol, maxterms, prnt)
%LAMBERTWM_ATOM Lambert W of triangular matrix with near constant diagonal.
%   [F, N_TERMS] = LAMBERTWM_ATOM(T, B, METHOD, TOL, MAXTERMS, PRNT)
%   evaluates lambertw at the upper triangular matrix T,%   where T has
%   nearly constant diagonal using either a Taylor or branch point series
%   taking at most MAXTERMS terms.
%   If PRNT ~= 0 information is printed on the convergence of the
%   Taylor series evaluation.
%   TERMS is the number of terms taken in the series.
%   TERMS = -1 signals lack of convergence.

%   Daniel Kirk

if (nargin < 4 || isempty(tol))             tol = eps;  end

% Deal with the case that T is scalar
n = length(T);
if (n == 1)
    F = lambertw(b, T);
    terms = 1;
    return;
end

eigen = diag(T);
meaneig = mean(eigen);
radeig = max(abs(eigen - meaneig));
if(b == 0 || b == -1 || b == 1)
    %% Here we trust that wblocking has done its job
    if(abs(meaneig + exp(-1)) <  buffer) method = 7; end
end
%% If we are using Sparta then any eigenvalues too close to the origin must
%% be evaluated using the traditional taylor series algorithm
```

```
if(method == 6  && abs(meaneig) < radeig) method = 0; end
if(b ~= 0       && abs(meaneig) < radeig)   b = 0;      end

switch(method)
    case 0,     % Taylor Approximation
        % Assuming T is nonscalar, first work out the average eigenvalue
        lambda = sum(diag(T))/n;

        %F = temp(T, lambda, maxterms);
        [F, out] = lambertwm_taylor(T, lambda, b,...
            tol, maxterms, prnt, false );
        terms = out.Terms;
    case 6,     % Taylor Approximation (Sparta: Schur--Parlett--Taylor--Auto)
        % Assuming T is nonscalar, first work out the average eigenvalue
        lambda = sum(diag(T))/n;
        [F, out] = lambertwm_taylor_ad(T, lambda, b, tol, maxterms, prnt);
        terms = out.Terms;
    case 7,     % Branch Point Approximation
        [F, out] = lambertwm_branchseries(T, tol, maxterms, prnt);
        terms = out.Terms;
end

if(prnt) fprintf('\n'); end
```

A.5 lambertwm_taylor_ad

```
function [W output] = lambertwm_taylor_ad(A, alpha, b, tol,...
    maxterms, prnt)
%LAMBERTWM_TAYLOR_AD Matrix Lambert W by Automatic Differentiation.

if(nargin < 3 || isempty(b))        b = 0;                          end
if(nargin < 4 || isempty(tol))      tol = 1.e-6;                    end
if(nargin < 5 || isempty(maxterms)) maxterms = ceil(1/(100*tol));   end
if(nargin < 6 || isempty(prnt))     prnt = 0;                       end

if(prnt)    fprintf('       -Lambert W Taylor Approximation-\n');   end

n = length(A);
w = lambertw(b, alpha);
W = w*eye(n);

%% To aid computation, get the triangular form of A, is A is not  %%
%% already triangular.                                            %%
if(triu(A,0) == A)
    T = A;
else
    [Q T] = schur(A, 'complex');
end

eigen = diag(T);
eigenw = lambertw(b,eigen);

cterms = zeros(1,maxterms + 2);
dterms = zeros(1,maxterms + 1);
etaterms = zeros(n,maxterms + n + 2);
```

```
xiterms = zeros(n,maxterms + n + 1);
init_autodiff();

mu = norm( (eye(n) - abs(triu(T,1)))\ones(n,1), inf );

itrs = 1;

%%  Variables used in global error testing %%
max_dn = 1;
max_dnW_spect = zeros(1, maxterms + n);

%%  Get T - alphaI %%
T_shift = T - eye(n)*alpha;
X = T_shift;

%% Initialise the error testing variables %%
ErrNorm = 0;
TrunBnd = 0;
exit = 0;

test = tol*norm(T,inf);

if(prnt)    fprintf('About %5.0e + %5.0ei, Spectral Radius %5.0e\n',...
        real(alpha), imag(alpha), max(abs(eig(T_shift))));        end

if (prnt)    fprintf('%3.0f: |f^(k)|/k! = %5.0e\n', 1, abs(w)); end


for(j = 1:maxterms)
    W_old = W;

    W = W + X*cterms(j + 1);
    X = X*T_shift;

    next_autodiff(j);

    %% Get the norm of the difference between each term %%
    ErrNorm = GetLErrNorm(W_old, W, j);

    %% If the relative error is small then that is a
    %% Indication that we should check the global error
    if(ErrNorm < tol)

        TrunBnd = GetGErrBound(j);

        if(TrunBnd == -1)
            warning('MATLAB:lambertwm_taylor:InfTru',...
                'Series diverges, incorrect result returned.' );
            exit = 3;
            itrs = j;
            break;
        end

        if(TrunBnd < test)
            itrs = j;
            break;
        end
    end

    if(ErrNorm == -1)
        warning('MATLAB:lambertwm_taylor:InfErr',...
            'Series diverges, incorrect result returned.' );
        exit = 2;
        itrs = j;
        break;
    end

    if(j == maxterms)
        exit = 1;
        warning('MATLAB:lambertwm_taylor:MaxExceded',...
            'Max terms exceded, results may be inaccurate.' );
        itrs = j;
        break;
    end

    if(prnt) fprintf('\n'); end

end
if(prnt) fprintf('\n\n'); end

if(nargout > 1)
    output = struct(...
        'Terms', itrs,...
        'ExitFlag', exit,...
        'LocalError', ErrNorm,...
        'TruncationBound', TrunBnd );
end

if(triu(A,0) ~= A)
    W = Q*W*Q'
end

if(prnt)
    [FError Res] = lambertwm_residual(A,W);
    fprintf(...
        'Residual = %5.0e, Forward Error = %5.0e\n', Res, FError);
    fprintf('      -End of Lambert W Taylor Approximation-\n');
end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function init_autodiff()
    %INIT_AUTODIFF Initiates Taylor sequences for W_b.
        cterms(1) = w;
        dterms(1) = 1/(1 + w);
        cterms(2) = w*dterms(1)/alpha;

        etaterms(:,1) = eigenw;
        xiterms(:,1) = 1./(1 + eigenw);
        etaterms(:,2) = eigenw.*xiterms(:,1)./eigen;
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function next_autodiff(idx)
```

```
%NEXT_AUTODIFF Continues Taylor sequence for W_b(\alpha).
    Q = 0;
    for(p = 1:idx)
        Q = Q + dterms(idx - p + 1)*cterms(p + 1);
    end
    dterms(idx + 1) = -Q/(1 + w);
    cterms(idx + 2) = -(idx*cterms(idx + 1)...
        + dterms(idx + 1))/(alpha*(idx + 1));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function next_eigen_autodiff(idx)
%NEXT_EIGEN_AUTODIFF Continues Taylor sequence for W_b(\lambda_i)
    Q = zeros(n,1);
    for(p = 1:idx)
        Q = Q + xiterms(:, idx - p + 1).*etaterms(:, p + 1);
    end
    xiterms(:, idx + 1) = -Q./(1 + eigenw);
    etaterms(:, idx + 2) = -(idx*etaterms(:, idx + 1)...
        + xiterms(:, idx + 1))./(eigen*(idx + 1));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function er = GetGErrBound(k)
%GETGERRBOUND Returns Truncation Error Estimate.
    for (r = max_dn : k + n - 1)
        max_dnW_spect(r) = norm(etaterms(:,r + 1),inf)*factorial(r);
        next_eigen_autodiff(r);
    end

    max_dn = k + n;

    omega_st = 0;
    for r = 0:n - 1
        omega_st = max(omega_st, max_dnW_spect(k + r)/factorial(r));
    end

    % norm(F) moved to RHS to avoid / 0.
    er = norm(X,inf)*mu*omega_st/factorial(k);

    if (prnt)fprintf('[trunc,test] = [%5.0e %5.0e]', er, test); end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function nm = GetLErrNorm(W_old, W, k)
%GETLERRBOUND Returns Local Error.
    error = W_old - W;
    if(min(min(isfinite(error)) == 0))
        nm = -1;
    else
        nm = norm(error, inf)/(tol + norm(W_old, inf));
    end
    if prnt
        fprintf('%3.0f: |f^(k)|/k! = %5.0e', k + 1, abs(cterms(k + 1)));
        fprintf('  ||N^k!|| = %7.1e', norm(X, inf));
        fprintf('  rel_diff = %5.0e', nm);
        fprintf('  abs_diff = %5.0e', norm(error, inf));
        end
    end
end
```

A.6 funm_w

```
function [F,exitflag,output] = funm_w(A,fun,options,varargin)
%FUNM_W  Evaluate general matrix function including Lambert W.
%   F = FUNM(A,FUN) evaluates the function_handle FUN at the square
%   matrix A. The MATLAB functions EXP, LOG, COS, SIN, COSH, SINH, LAMBERTW
%   can be passed as FUN, i.e., FUNM(A,@EXP), FUNM(A,@LOG), FUNM(A,@COS),
%   FUNM(A,@SIN), FUNM(A,@COSH), FUNM(A,@SINH), FUNM(A,@LAMBERTW) are all
%   allowed. If FUN ~= @LAMBERTW then FUNM should be used instead.

if isequal(fun,@lambertw)  || isequal(fun,'lambertw')
    fun = @lambertw;
else
    [F,exitflag,output] = feval(@funm,A,fun,options,varargin{:});
    return;
end

% Default parameters.
prnt = 0;
delta = 0.1;
tol = eps;
maxterms = 250;
ord = [];
reord = 1;

WMethod = 0;
WBranch = 0;
WBuffer = 0.1;
WShowBlk = false;

if (nargin > 2 && ~isempty(options))
   if isfield(options,'Display') && ~isempty(options.Display)
      switch lower(options.Display)
         case 'on',      prnt = 1;
         case 'verbose', prnt = 2;
      end
   end
   if isfield(options,'TolBlk') && ~isempty(options.TolBlk)
      delta = options.TolBlk;
      if delta <= 0, error('MATLAB:funm:NegTolBlk',...
              'TolBlk must be positive.' ); end
   end

   if isfield(options,'TolTay') && ~isempty(options.TolTay)
      tol = options.TolTay;
      if tol <= 0, error('MATLAB:funm:NegTolTay',...
              'TolTay must be positive.' ); end
   end
   if isfield(options,'MaxTerms') && ~isempty(options.MaxTerms)
      maxterms = options.MaxTerms;
      if maxterms <= 0
```

```
        error('MATLAB:funm:NegMaxTerms', 'MaxTerms must be positive.');
      end
   end
   if isfield(options,'Ord') && ~isempty(options.Ord)
      ord = options.Ord;
      if length(ord) ~= length(A)
        error('MATLAB:funm:WrongDimOrd', 'Incorrect dimension for Ord.');
      end
   end
   if(isfield(options,'Method') && ~isempty(options.Method))
       switch(lower(options.Method))
           case 'taylor',       WMethod = 0;
           case 'sparta',       WMethod = 6;
           case 'branchpoint',  WMethod = 7;
           otherwise
               error('MATLAB:funm:W:MethodInvalid', 'Invalid Method.');
       end
   end
   if isfield(options,'Branch') && ~isempty(options.Branch)
       WBranch = options.Branch;
       if(ceil(WBranch) ~= WBranch) error(...
               'MATLAB:funm:W:InvalidBranch',...
               'Branch must be an integer.' );
       end
   end
   if isfield(options,'Buffer') && ~isempty(options.Buffer)
       WBuffer = options.Buffer;
       if(options.Buffer < 0) error('MATLAB:funm:W:NegBufRad',...
               'Buffer Radius Must be Nonnegative.');  end
   end
   if isfield(options,'ShowBlocking') && ~isempty(options.ShowBlocking)
       WShowBlk = options.ShowBlocking;
       if(~islogical(options.ShowBlocking)) error(...
               'MATLAB:funm:W:ShowBlk',...
               'Field ShowBlocking must be true/false.' );  end
   end
end

[m,n] = size(A);
if ndims(A) > 2 || m ~= n
    error('MATLAB:funm:InputDim', 'First input must be a square matrix.');
end

% First form complex Schur form (if A not already upper triangular).
if isequal(A,triu(A))
    T = A; U = eye(n);
else
    [U,T] = schur(A,'complex');
end

if isequal(T,tril(T)) % Handle special case of diagonal T.
    F = U*diag(lambertw(diag(T)))*U';
    exitflag = 0;
    output = struct( 'terms', ones(n,1), 'ind', {1:n}, 'ord',1:n );
    return;
end
```

```
% Determine reordering of Schur form into block form.
if isempty(ord)
    if(isequal(WMethod,6) && isequal(WBranch,0))
        ord = wblocking(T, delta, WBuffer, -1, WShowBlk);
    else
        ord = wblocking(T, delta, WBuffer, WBranch, WShowBlk);
    end
end

[ord, ind] = swapping(ord);  % Gives the blocking.
ord = max(ord)-ord+1;        % Since ORDSCHUR puts highest index top left.
[U,T] = ordschur(U,T,ord);

%% Setup the global variables for the derivatives
lambertwdiff_setup(maxterms + n);

m = length(ind);
% Calculate F(T)
F = zeros(n);

totalbranches = 0;

for col=1:m
   j = ind{col};
   if prnt == 2 && max(j) > min(j)
      fprintf('Evaluating function of block (%g:%g)\n', min(j), max(j))
   end
   branch = round((rand(1)*2 - 1)*WBranch);
   totalbranches = totalbranches + abs(branch);
   [F(j,j), terms(col)] =...
       lambertwm_atom(T(j,j), branch, WMethod, WBuffer,...
       tol, maxterms, prnt>1 );

   for row=col-1:-1:1
      i = ind{row};
      if length(i) == 1 && length(j) == 1
         % Scalar case.
         k = i+1:j-1;
         temp = T(i,j)*(F(i,i) - F(j,j)) + F(i,k)*T(k,j) - T(i,k)*F(k,j);
         F(i,j) = temp/(T(i,i)-T(j,j));
      else
         k = cat(2,ind{row+1:col-1});
         rhs = F(i,i)*T(i,j) - T(i,j)*F(j,j) + F(i,k)*T(k,j) - T(i,k)*F(k,j);
         F(i,j) = sylv_tri(T(i,i),-T(j,j),rhs);
      end
   end
end

%% Clear the global variables from memory
lambertwdiff_flush;

F = U*F*U';

if isreal(A) && norm(imag(F),1) <= 10*n*eps*norm(F,1)
   F = real(F);
end
```

```
if prnt
  fprintf('  Block   Number of Taylor series terms\n')
  fprintf('          (or square roots in case of log):\n')
  fprintf('  ---------------------------------------\n')
  for i = 1:length(ind)
      fprintf(' (%g:%g)      %g\n', min(ind{i}), max(ind{i}), terms(i) )
  end
end

exitflag = 0;
if any(terms == -1)
   exitflag = 1;
   warning('MATLAB:funm:TaylorSeriesNotConverged',...
           ['Taylor series failed to converge.',...
            '        Try decreasing options.TolBlk or increasing',...
            'options.TolTay or options.MaxTerms.'])
end

if nargout >= 3
   output = struct('terms', terms, 'ind', {ind}, 'ord', ord, 'T', T,...
       'NumBranches', totalbranches);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function X = sylv_tri(T,U,B)
%SYLV_TRI    Solve triangular Sylvester equation.
%   X = SYLV_TRI(T,U,B) solves the Sylvester equation
%   T*X + X*U = B, where T and U are square upper triangular matrices.

m = length(T);
n = length(U);
X = zeros(m,n);

% Forward substitution.
for i = 1:n
    X(:,i) = (T + U(i,i)*eye(m)) \ (B(:,i) - X(:,1:i-1)*U(1:i-1,i));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [mm,ind] = swapping(m)
%SWAPPING  Choose confluent permutation ordered by average index.
%   [MM,IND] = SWAPPING(M) takes a vector M containing the integers
%   1:K (some repeated if K < LENGTH(M)), where M(J) is the index of
%   the block into which the element T(J,J) of a Schur form T
%   should be placed.
%   It constructs a vector MM (a permutation of M) such that T(J,J)
%   will be located in the MM(J)'th block counting from the (1,1) position.
%   The algorithm used is to order the blocks by ascending
%   average index in M, which is a heuristic for minimizing the number
%   of swaps required to achieve this confluent permutation.
%   The cell array vector IND defines the resulting block form:
%   IND{i} contains the indices of the i'th block in the permuted form.

mmax = max(m); mm = zeros(size(m));
g = zeros(1,mmax); h = zeros(1,mmax);

for i = 1:mmax
```

```
    p = find(m==i);
    h(i) = length(p);
    g(i) = sum(p)/h(i);
end

[x,y] = sort(g);
h = [0 cumsum(h(y))];

ind = cell(mmax,1);
for i = 1:mmax
    p = find(m==y(i));
    mm(p) = i;
    ind{i} = h(i)+1:h(i+1);
end
```

Bibliography

[1] Awad H. Al-Mohy and Nicholas J. Higham. Computing the Fréchet derivative of the matrix exponential, with an application to condition number estimation. MIMS EPrint 2008.26, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, February 2008.

[2] J.D. Anderson. *Introduction to Flight.* McGraw-Hill, New York, third edition, 1989.

[3] François Chapeau-Blondeau. Numerical evaluation of the Lambert W function and application to generation of generalized Gaussian noise with exponent 1/2. *IEEE Transactions on Signal Processing*, 50(9):2160–2165, September 2002.

[4] Robert M. Corless. Private Correspondence, 2008.

[5] Robert M. Corless, Hui Ding, Nicholas J. Higham, and David J. Jeffrey. The solution of $S\exp(S) = A$ is not always the Lambert W function of A. In *ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, pages 116–121, New York, 2007. ACM Press.

[6] Robert M. Corless, Gaston H. Gonnet, D. E. G. Hare, and David J. Jeffrey. Lambert's W function in Maple. *The Maple Technical Newsletter*, 9:12–23, 1993.

[7] Robert M. Corless, Gaston H. Gonnet, D. E. G. Hare, David J. Jeffrey, and Donald E. Knuth. On the Lambert W function. *Advances in Computational Mathematics*, 5(4):329–359, 1996.

[8] Steven R. Cranmer. New views of the solar wind with the Lambert W function. *AM.J.PHYS.*, 72:1397, 2004.

[9] Philip I. Davies and Nicholas J. Higham. A Schur–Parlett algorithm for computing matrix functions. *SIAM J. Matrix Anal. Appl.*, 25(2):464–485, 2003.

[10] E. M. Lemeray. Sur les racines de l'equation $x = a^x$. Racines imaginaires. *Nouvelles Annales de Mathématiques*, 16(3):54–61, 1897.

[11] G. Pólya. Kombinantorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, 68:145–254, 1937. English translation by Dorothee Aeppli in G. Pólya and R.C. Reed, Combinatorial Enumeration of Groups, Graphs and Chemical Compounds, Springer-Verlag, 1987.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[13] P. Halmos. *Finite Dimensional Vector Spaces*. Van Nostrand, New York, NY, USA, 1958.

[14] Nicholas J. Higham. The Matrix Function Toolbox. `http://www.ma.man.ac.uk/~higham/mftoolbox`.

[15] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[16] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[17] Roger A. Horn and Charles R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1991.

[18] W. Kahan. Branch cuts for complex elementary functions or much ado about nothing's sign bit. In A. Iserles and M. J. D. Powell, editors, *The State of the Art in Numerical Analysis*, pages 165–211. Oxford University Press, 1987.

[19] A. Ringwald M. Gibbs and F. Schrempp. Qcdins 2.0 - a Monte Carlo generator for instanton-induced processes in deep-inelastic scattering. In *in Proc. DIS (Paris)*, pages 341–344. J.-F. Laporte and Y.Sirois, Eds., 1995.

[20] Roy Mathias. Approximation of matrix-valued functions. *SIAM J. Matrix Anal. Appl.*, 14(4):1061–1063, 1993.

[21] Patrick W. Nelson, A. Galip Ulsoy, and Sun Yi. Delay differential equations via the matrix Lambert W Function and bifurcation analysis: Applications to machine tool chatter. *Math. Biosci. Eng*, 4:355–368, 2007.

[22] Patrick W. Nelson, A. Galip Ulsoy, and Sun Yi. Analysis and control of time delayed systems via the Lambert W Function. Technical report, IFAC, 2008.

[23] T. C. Scott, J. F. Babb, A. Dalgarno, and J. D. Morgan, III. Resolution of a paradox in the calculation of exchange forces for H_2^+. *Chemical Physics Letters*, 203:175–183, February 1993.