An inverse field of values problem

2008 Inverse Problems 24 055019

(http://iopscience.iop.org/0266-5611/24/5/055019)

Inverse Problems **24** (2008) 055019 (19pp) doi:10.1088/0266-5611/24/5/055019

An inverse field of values problem

Frank Uhlig

Department of Mathematics and Statistics, Auburn University, Auburn, AL 36849-5310, USA

E-mail: uhligfd@auburn.edu

Received 7 June 2008, in final form 12 August 2008
Published 12 September 2008
Online at stacks.iop.org/IP/24/055019

Abstract

For a given complex matrix A we use methods of geometric computing to find complex unit vectors that map to a given point p inside the field of values of A. This is an inverse problem for the quadratic field of values map

$$x \in \mathbb{C}^n, \|x\|_2 = 1 \longrightarrow p = x^*Ax \in \mathbb{C}.$$

We describe and test an algorithm and include computed examples, as well as related questions on the geometric generation of points in the field of values, i.e., on the inner geometry of the field of values.

Dedicated to Richard S Varga for his love of ovals and complex curves.

(Some figures in this article are in colour only in the electronic version)

1. Introduction

This paper deals with methods from geometric computing to discern geometric properties that relate the geometry of the complex unit n-sphere to that of the field of values points x^*Ax in $\mathbb{C}$ in terms of A and vice versa. The field of values map that maps the complex n-sphere to points in $\mathbb{C}$ involves a large drop of dimension. This paper is but one of first discovery, theory, and computations, as well as open questions.

For any real or complex n by n matrix A, the field of values $F(A)$ is defined as the image of the complex unit sphere $S_n(\mathbb{C}) = \{x \in \mathbb{C}^n \mid \|x\|_2 = 1\}$ under the (generalized) quadratic form map

$$x \in S_n(\mathbb{C}) \to x^*Ax \in \mathbb{C}.$$

Clearly the field of values $F(A) = \{x^*Ax \in \mathbb{C} \mid x \in \mathbb{C}^n, \|x\|_2 = 1\}$ is a compact subset of $\mathbb{C}$ as the continuous image of the compact unit sphere. Furthermore the Hausdorff–Toeplitz theorem of 1918 asserts that $F(A)$ is convex for any A.

The boundary $\partial F(A)$ of the field of values of a square matrix A can be computed approximately by using a moderate number of Hermitian eigenvalue/eigenvector evaluations involving the Hermitian and skew-Hermitian parts of A, see [6].

The geometry and lay of $F(A)$ and its boundary $\partial F(A)$ have been studied for 70 years, see e.g. the exposition in [5, chapter 1], or [11, p 220] for a survey using a slightly different

framework. Moreover, the $F(A)$ boundary curve was proved an algebraic curve by Fiedler [3] and certain inner $F(A)$ sub-extremal eigenvalue curves were found to be differentiable except for swallowtails in [7].

Specifically applied to continuous or discrete systems $\dot{x} = Ax$ or $x_{k+1} = Ax_k$, whether $0 \in F(A)$ or not influences the stability of the underlying system as defined in terms of two Hermitian matrices H and K; see [4]. By considering the two matrices H and K as the Hermitian and skew-Hermitian parts $H = (A + A^*)/2$ and $K = (A - A^*)/(2\mathrm{i})$ of the square matrix $A = H + \mathrm{i} \cdot K$, the stability problem reduces to computing the minimal distance from $0 \in \mathbb{C}$ to $F(A)$, called the Crawford number, see [4, 12] for example. On the other hand, the maximal distance between $0 \in \mathbb{C}$ and points in $F(A)$, called the numerical radius of A, determines the convergence behavior of the associated dynamical system $\dot{x} = Ax$ or $x_{k+1} = Ax_k$, see [9, introduction] for theory and [9, 13] for algorithms and MATLAB codes.

Moreover, for two symmetric or Hermitian matrices S and T, the lay of $0 \in \mathbb{C}$ with respect to the field of values of $A = S + \mathrm{i} \cdot T$ determines whether the matrix pencil $P(S, T) = \{aS + bT \mid a, b \in \mathbb{R}\}$ contains a positive-definite matrix or not, see [10]. This in turn has consequences on the ability to simultaneously diagonalize S and T by congruence, see [11] for a survey of these results.

Our current interest lies in the inner geometry of the field of values of a given matrix $A \in \mathbb{C}^{n,n}$. Concretely, we consider the inverse computational problem how any one point p inside $F(A)$ can be generated, i.e., for a given matrix A, we want to find unit vectors in $\mathbb{C}^n$ that map to a specific given point $p \in F(A)$. How can this be done and how easily? Can this be done at all?

Further questions come to mind such as how many distinct or linearly independent generating unit vectors exist for any one point p in the field of values of A and how, if at all, does their number depend on the location of $p \in F(A)$ and on the matrix A. It may be somewhat audacious to try to find generating vectors and understand their geometry on the complex unit n-sphere solely from their complex images under x^*Ax. In fact, this inverse problem has been presented over the last dozen years to many colleagues knowledgeable in this area without yielding any tangible results. First results on this inverse problem in the normal matrix case are in [14].

In this paper we outline a numerical algorithm that finds successively closer approximations to a given point p in the field of values for a given matrix A, as well as a generating vector x on $S_n(\mathbb{C})$ for p. Simply speaking, for a given square matrix $A \in \mathbb{C}^{n,n}$ we want to solve the two equations

$$x^*Ax = p \in \mathbb{C} \qquad \text{and} \qquad x^*x = 1 \tag{1}$$

simultaneously for $x \in \mathbb{C}^n$ with $p \in \mathbb{C}$ chosen so as to make the system solvable. This is an algebraic problem described by two-coupled complex quadratic equations in the n complex components of x. The equivalent real problem defined by the respective real and imaginary parts equations involves an equivalent set of three-coupled real quadratic equations in $2n$ unknowns. Finding an algebraic solution can be handled by computer algebra systems such as MAPLE, MAGMA or MATEMATICA for moderate dimensions n only. Once n exceeds 3 or 5, the problem generally becomes unreasonably hard to be solved algebraically on most modern desk- and laptop computers. Other approaches that come to mind for solving (1) are steepest descent methods from a starting vector on $S_n(\mathbb{C})$. But these unfortunately will only converge linearly, i.e., very slowly. For low values of n, one could also try a Lagrange multiplier solution. On the other hand, our numerical method will generally converge rapidly and achieve a high-accuracy solution in 5–12 iterative steps regardless of the actual dimension n with each step at $O(n^2)$ cost. Since we are interested in finding solution vectors $x \in \mathbb{C}^n$ of

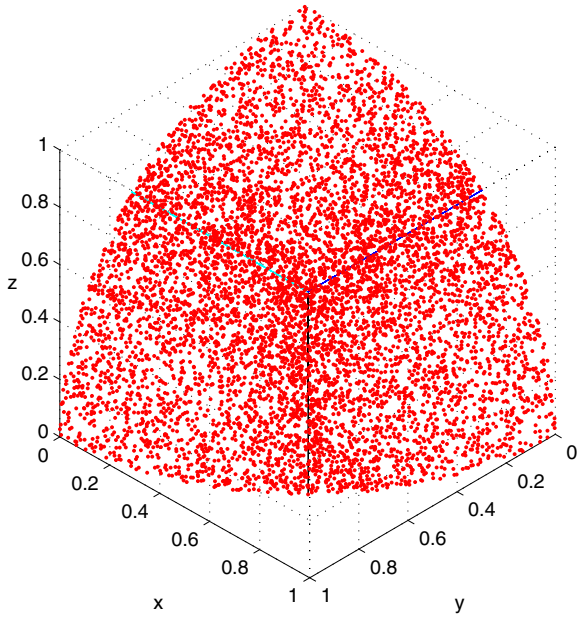

Figure 1. The images of 7600 random vectors in the first orthant of the unit cube in $\mathbb{R}^3$, after their projection onto the unit sphere $S_3(\mathbb{R})$.

(1) for much larger n than algebraic or analytic methods can provide, such as n in the hundreds and thousands, we rely on a numerical algorithm based on floating point arithmetic in order to solve the inverse problem (1).

Our algorithm involves two phases; in the first we try to encircle a given point $p \in \mathbb{C}$ by field of values points with known generating vectors until $p \in \mathbb{C}$ lies in their convex hull, if possible. If we succeed, then we shrink the enclosure in phase 2 iteratively until a certain error threshold is reached in approximating p by FOV points with known generating vectors. The algorithm is essentially geometric in nature. Besides, it uses random computations in phase 1 and—only if needed—up to six matrix eigenvalue and eigenvector evaluations in phase 2.

We use the acronym 'FOV' for 'field of values' for brevity from now on.

2. A geometric algorithm to approximate a given FOV point by FOV points with known generating vectors

FOV points can be generated in several ways. For one, a natural method is to evaluate x^*Ax for a number of unit vectors and see whether the given point $p \in \mathbb{C}$ lies inside their convex hull. For this it would be ideal to start out with a uniformly distributed set of unit vectors in $\mathbb{C}^n$. But a uniform distribution on $S_n(\mathbb{C})$ is apparently unknown or impractical as we shall elaborate. Inside the unit cube $[-1, 1]^{2n} \subset \mathbb{R}^{2n} \equiv \mathbb{C}^n$, however, we can obtain a uniform distribution by for example using MATLAB's `rand` function in each of the $2n$ dimensions. If we normalize uniformly distributed vectors in the complex unit cube to lie on $S_n(\mathbb{C})$ we can evaluate their induced FOV points x^*Ax in $O(n^2)$ time for each one. However, the normalized vectors on $S_n(\mathbb{C})$ are not random and become less and less uniformly distributed the larger n becomes. To illustrate, we consider the first orthant of the unit cube in $\mathbb{R}^3$ in figure 1.

Figure 1 shows the first orthant of the real unit sphere, viewed head on from the corner $(1, 1, 1)$. Note the increasing density of points near the cube's three edges that connect

(0, 1, 1), (1, 0, 1) and (1, 1, 0), respectively, with the corner (1, 1, 1). This shows up as a fuzzy, denser region in the shape of the letter Y in figure 1 and is due to the fact that the corner (1,1,1) of the cube is $\sqrt{3} \doteq 1.71$ units distant from the origin, while the points (0,1,0), (1,0,0) and (0,0,1) lie on the unit sphere itself, creating around 1.7 times as many points near the cube's (1,1,1) corner projection $(\sqrt{3}, \sqrt{3}, \sqrt{3})/3$ on $S_3(\mathbb{R})$. The uneven density ratio along the n inner orthant edges and near its projected corners will increase for $S_n(\mathbb{C})$ as n does. $S_n(\mathbb{C})$ is isomorphic to $S_{2n}(\mathbb{R})$ of the doubled dimension. Thus for the complex case and $n = 100$, the disparity factor will be $\sqrt{2n} = \sqrt{200} \doteq 14$ and for $n = 1000$ it will be $\sqrt{2000} \doteq 45$. In fact, normalizing a set of vectors that is uniformly distributed inside the complex unit cube will generate a 'fishnet'-type distribution on each of the 2^n orthant segments of the n-dimensional complex unit sphere giving us a highly non-normal distribution on $S_n(\mathbb{C})$.

This could be remedied by using a normal distribution on the unit sphere via—for example—generalizing 3D real spherical coordinates to dimensions $2n$ in $S_{2n}(\mathbb{R}) \equiv S_n(\mathbb{C})$. But even if this succeeds, more devastating for finding truly random points of $F(A)$ from random unit vectors would be the large number $2n$ of faces of the complex unit cube (or $4n$ of the equivalent real unit cube) as n increases. Counting the number 2^{2n} of real orthants for $n = 10, 100, 1000$ we have approximately $2^{20} \doteq 10^6$, $2^{200} \doteq 1.6 \times 10^{60}$ and $2^{2000} \doteq 2 \times 10^{602}$ real faces on a real unit cube of dimension $2n$, respectively. Thus any realistic number of random or pseudo-random unit vectors can only reach a very small portion of the unit cube's faces and subsequently of the unit sphere's orthant surfaces and it can do so only more and more sparsely as n increases. Thus there is but a slim chance to encircle a point p in $F(A)$ by evaluating x^*Ax repeatedly, and this chance becomes significantly lower as n increases. In our algorithm [15] we typically start with 40–100 pseudo-random complex unit vectors and evaluate their generated FOV points. From our experience, these pseudo-randomly produced FOV points are generally located near and around the center of the convex field of values. Sometimes they suffice to encircle p.

Besides evaluating x^*Ax (pseudo-)randomly on $S_n(\mathbb{C})$ we must therefore employ more effective methods such as eigenvalue/eigenvector evaluations in our algorithm in case the (pseudo-)random starting vectors generate FOV points that do not encircle p.

Boundary points of the field of values $F(A)$ of a matrix $A \in \mathbb{C}^{n,n}$ can be computed and plotted according to [6] by decomposing the matrix A into its Hermitian and skew-Hermitian parts $H = (A + A^*)/2$ and $K = (A - A^*)/(2\mathrm{i})$ and computing the extreme real eigenvalues $\lambda_{\min}(\theta)$ and $\lambda_{\max}(\theta) \in \mathbb{R}$ with their corresponding unit eigenvectors $ev_{\min}(\theta)$ and $ev_{\max}(\theta) \in \mathbb{C}^n$, respectively, for the Hermitian matrix

$$A(\theta) = \cos(\theta) \cdot H + \sin(\theta) \cdot K$$

and varying values of $0 \leqslant \theta < \pi$. This is equivalent to a rotation of the field of values of A by the angle θ and evaluating the extreme horizontal extensions of the rotated field of values. The extreme eigenvalues of the matrix $A(\theta)$ in fact determine two sides of Bendixson's eigenvalue inclusion rectangle for $A(\theta)$, see [1]. The points $ev^*_{\min}(\theta) \cdot A \cdot ev_{\min}(\theta)$ and $ev^*_{\max}(\theta) \cdot A \cdot ev_{\max}(\theta) \in \mathbb{C}$ lie on the boundary $\partial F(A)$ as explained in [6, theorem 2, p 597]. This is illustrated in figure 2. In it we plot three distinct FOV curves that reflect the location of the leading three eigenvalues of $A(\theta)$ as θ varies through 40 angles from 0 to π for a random 6 by 6 matrix A. For $\theta = 0$ we additionally plot the six eigenvalues $\lambda_i \in \mathbb{R}$ of $A(0) = (A + A^*)/2$ by six * symbols on the real axis as well as the corresponding six images $ev^*(\lambda_i) \cdot A \cdot ev(\lambda_i) \in F(A) \subset \mathbb{C}$ for $i = 1, \ldots, 6$ by small circles 'o' in figure 2. These field of values points are local extreme points on the three FOV curves. They lie on perpendiculars to the real axis ($\theta = 0$) in figure 2. The inner FOV points and their generating vectors are obtained at very little extra cost, once an eigenanalysis of $A(\theta)$ has been computed. The above

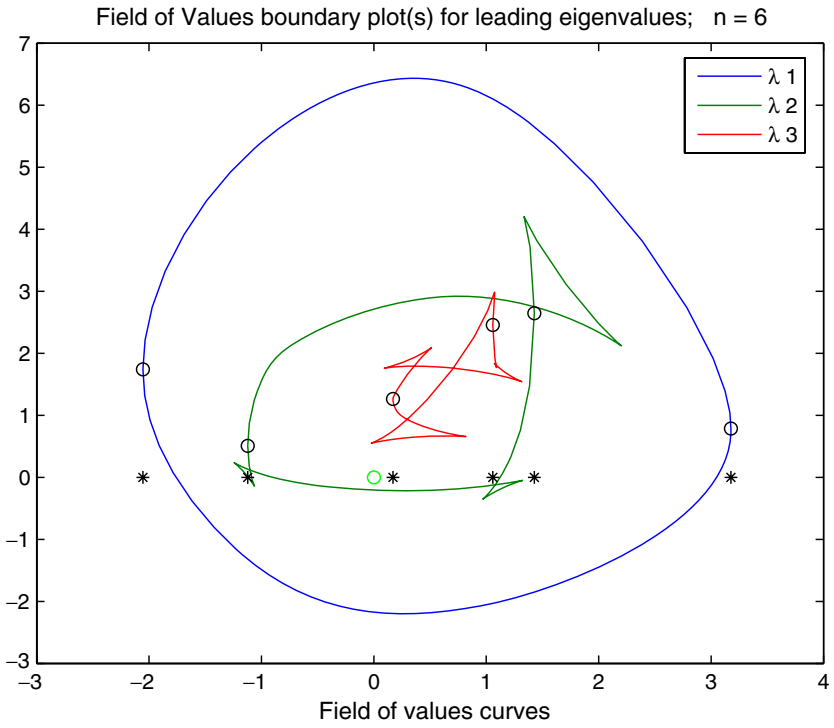

Figure 2. $*$ = real eigenvalues of $A(0)$; o = associated FOV points for $A(0)$.

motivates the first, the encircling phase of our algorithm. This is followed in the algorithm by a second phase of approximating the given point $p \in F(A)$ with FOV points with known generating vectors on $S_n(\mathbb{C})$. We now give details of the two phases.

2.1. *Phase 1: encircling $p \in F(A)$ with field of values points with known generators*

The complicated relationship, described and depicted earlier, between the real eigenvalues $*$ of $A(\theta)$ on the real axis of $\mathbb{C}$ rotated by θ and the points x^*Ax (o in figure 2) that are generated inside $F(A)$ by the corresponding unit eigenvectors $x \in \mathbb{C}^n$ under the map x^*Ax shows that there is no simple way to approximate a given FOV point $p \in \mathbb{C}$ from matrix eigenanalyses alone in the general non-normal matrix case. However, we can make use of matrix eigenanalyses to encircle $p \in \mathbb{C}$ by FOV points in the following fashion.

Note in figure 2 that the least squares interpolating line through the six computed FOV points o is almost horizontal, i.e. nearly parallel to the angle of rotation $\theta = 0$. If we set $a = \text{trace}(A)/n = \big(\sum(\text{eigenvalues of A})\big)/n$ and a is near to p, then a small number of pseudo-random unit vector x^*Ax evaluations will probably suffice to encircle the point p because $a = \text{trace}(A)/n$ represents a reasonable 'center' point for $F(A)$. If p and a are further apart, then by setting θ equal to the angle that the line through a and p forms with the real axis and evaluating the FOV points and their generating unit eigenvectors through an eigenanalysis of $A(\theta)$—at $O(n^3)$ cost—will create a string of points in the field of values that generally will lie on or adjacent to the line through a and p. And chances are good that these FOV points will encircle p in all four quadrants around it, unless $p \in F(A)$ is very close to $\partial F(A)$ or n is very small, see figure 3 in section 5 for an illustration. In fact, for large n and p not too near

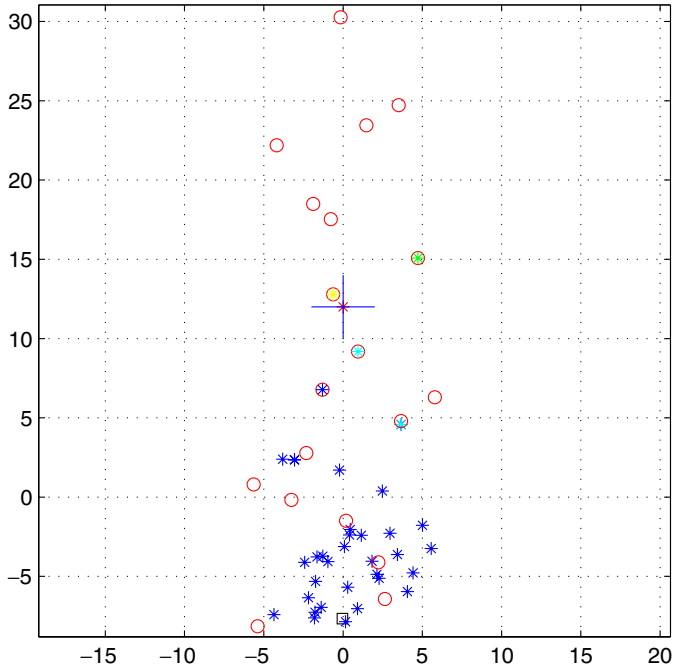

Figure 3. (∗) symbols of random FOV points, circles (○) for eigenanalysis generated FOV points (□) for the FOV 'center' a near -8i, large cross (+) at the location of $p = 12$i, the cross's axes separate the FOV data into the four quadrants around p; circles with centers filled in are the four quadrant closest-to-p FOV points.

the boundary $\partial F(A)$, the large number of FOV points found in one eigenanalysis of $A(\theta)$ for an appropriate value of θ will often be enough to encircle p. If a single eigenanalysis does not suffice, our algorithm modifies the first angle θ in $A(\theta)$ further and tries to encircle p inside a convex subset of $F(A) \subset \mathbb{C}$ with known generating unit vectors using further eigenanalyses and quadratic form evaluations. If this fails after three eigenanalyses that leave at least two quadrants around p empty, we conclude that $p \notin F(A)$ and stop. If on the other hand, the computed FOV points lie in three of the four quadrants around p, we find the two FOV points that sustain the smallest angle with the boundary rays of the empty quadrant and use their two vector generators. For these we evaluate several FOV points on the ellipse in $F(A)$ that is generated by the vectors on the real great circle through them on $S_n(\mathbb{C})$, see figures 4 and 5. If these interpolates still remain outside the empty quadrant, we employ further eigenanalyses of $A(\theta)$, each at $O(n^3)$ cost for three additional judiciously chosen values of θ and if one quadrant around p remains empty after all, we conclude that p likely lies outside $F(A)$; or inside, but extremely close to its boundary. False negatives have been observed occasionally for small relative distances between p and $\partial F(A)$ of magnitudes 10^{-7} to 10^{-13}, depending on A, the shape of $F(A)$, and the lay of $p \in F(A)$.

2.2. Phase 2: finding an approximate generating unit vector x with $x^*Ax \approx p \in F(A)$ from four encircling FOV points and their unit vector generators

Once we know four encircling FOV boundary or interior points f_i for p, one in each of the four quadrants around p, together with their unit vector generators x_i from section 2, we reduce the size of their convex hull inside $F(A)$ iteratively until the nearest-to-p computed FOV point

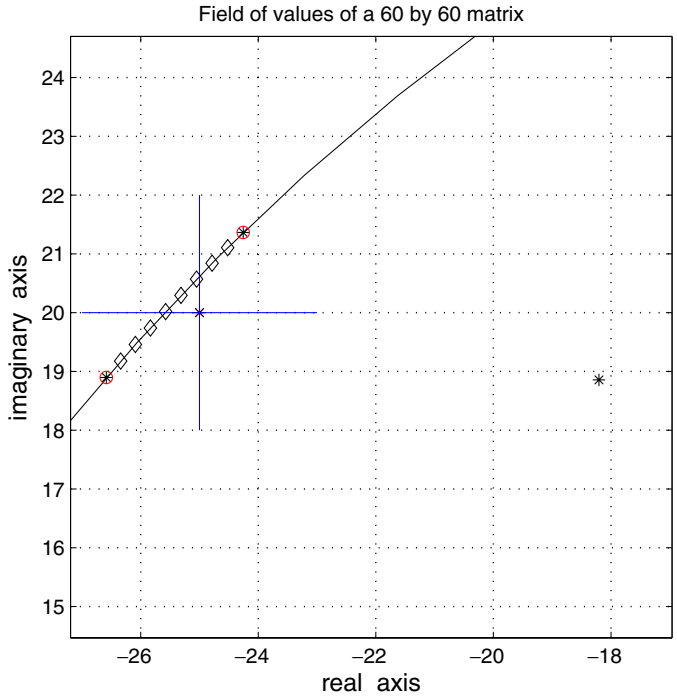

Figure 4. Enlarged view of the $F(A)$ area around p at the center of the cross, with an approximate boundary curve for $F(A)$ drawn here.

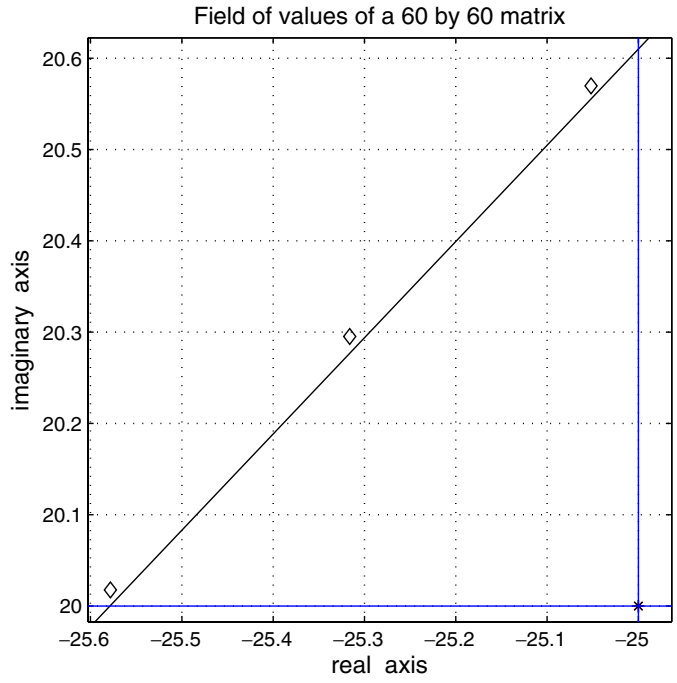

Figure 5. Further enlargement of figure 4.

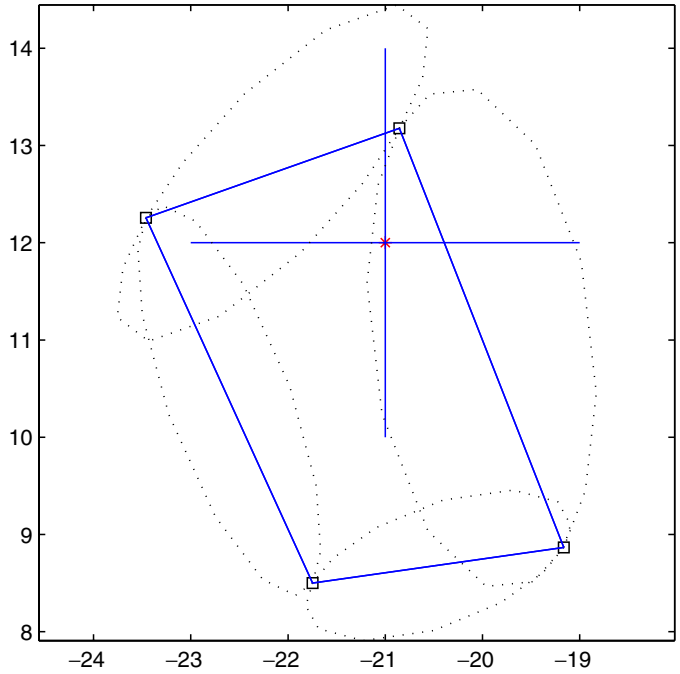

Figure 6. FOV elliptic interpolation points ··· for quadrilateral sides—, with p at large +, with ellipses generated on four real great circles through two generating vectors for adjacent $\partial F(A)$ corners (□) of the quadrilateral surrounding p.

lies within 10^{-9} of p in absolute terms. And we accept its generating unit vector as a solution to the inverse problem.

How to shrink the quadrilateral formed by the current best approximations of p in the four quadrants around p? It is well known that the images of great circles on $S_n(\mathbb{C})$ are ellipses in $F(A)$ under the map $x \to x^*Ax$, see [2] or [8]. Therefore we evaluate a limited number of FOV points around p from equally spaced vectors on the six great circles that contain one pair of generating vectors from the four quadrilateral corners f_i that surround $p \in F(A)$. The line connecting f_1 and $f_2 \in F(A)$ is a secant of the associated ellipse in $F(A)$. If the quadratic form image under x^*Ax of the great circle through the generating vectors x_1 and $x_2 \in S_n(\mathbb{C})$ of f_1 and $f_2 \in F(A)$ is not degenerate, i.e., if it is not line segment in $\mathbb{C}$, then it will contain points on the opposite side of the line through f_1 and $f_2 \in \mathbb{C}$ when viewed from p, as well as others that lie on the same side of this line as p; i.e., these elliptical images inside $F(A)$ may contain points that lie closer to p than f_1 and f_2 did. Therefore we compute points on these ellipses for the four edges and the two diagonals of the quadrilateral formed by the approximating $f_i \in F(A)$ of p. This may generate additional close-to-p FOV points with known generators, see figure 6. However, it may happen that the ellipses all encircle the given point p very widely and thus fail to improve on the original four corner approximations f_i of p. Even worse, sometimes the four generating vectors x_i of the f_i appear to lie nearly on the same great circle of $S_n(\mathbb{C})$ that encircles p. In either of these cases our algorithm is bound to stall. To avoid any premature stalling, we therefore also consider generators on planar cuts of the unit sphere other than great circles. These are chosen to contain two generating vectors x_1 and $x_2 \in S_n(\mathbb{C})$ but not $0 \in \mathbb{C}^n$. For points on these special planar cuts of $S_n(\mathbb{C})$ we compute a limited number of FOV image points under the quadratic map x^*Ax. More precisely, we

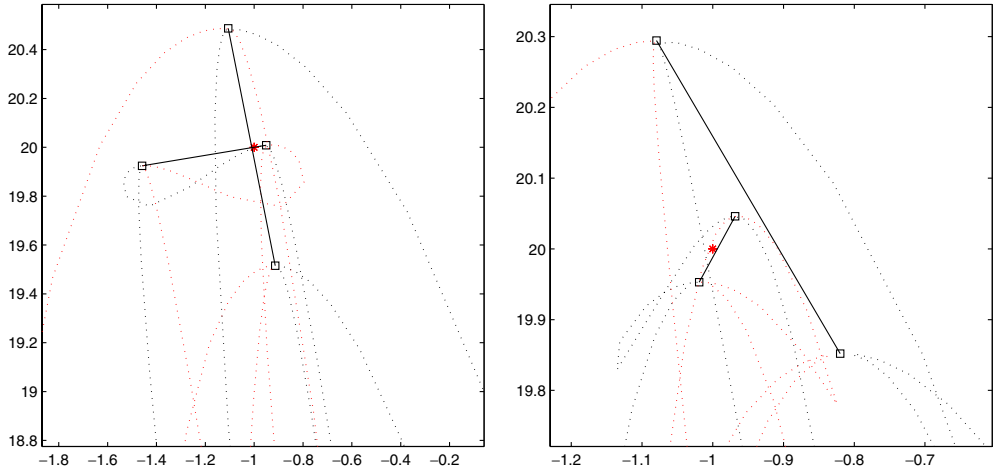

Figure 7. FOV non-elliptic interpolation points $\cdots$ for quadrilateral diagonals—; with p at $(*)$, generated by tilted unit sphere circles through two generating vectors with FOV images at opposite corners ($\square$) of the quadrilateral around p.

chose tilted planes that contain two points x_1 and $x_2 \in S_n(\mathbb{C})$ from among those that currently yield the best four quadrant approximates $f_i = x_i^* A x_i$ of $p \in \mathbb{C}$. These special planes are tilted by random angles around the line through x_1 and x_2. The image points of any such tilted plane intersection with the unit sphere describe a curve that passes through both FOV points $f_1 = x_1^* A x_1$ and $f_2 = x_2^* A x_2$ by construction. This curve in $F(A)$ is closed but generally not an ellipse, see figure 7 for example images.

After several such elliptic or non-elliptic FOV curve evaluations, we re-sort the old and new computed FOV points into the four quadrants around p and continue anew from the closest approximations f_i of p in each quadrant and their respective unit vector generators x_i. If the process does not stall, it shrinks the quadrilateral of FOV points that encircles p. Taken as a whole, this process needs from 4 to 12, and typically 6 to 9 iterations at $O(n^2)$ cost to find a generating unit vector for a point $\hat{p}$ in $F(A)$ within a distance of less than 10^{-9} from p. The number of needed iterations appears to be independent of the dimension n. If p lies close to the edge $\partial F(A)$, however, then the number of iterations until convergence may increase slightly by a few steps.

3. Some fine points of the preparatory and of the iterative phases of our algorithm

Our algorithm [15] is realized in MATLAB and was tested under `MATLAB_R2007b` both on a Sunblade 100 from 2002 and a MAC Pro from 2007. Its uses MATLAB's basic functions such as `sort` and `find`, as well as the QR eigenvalue algorithm `eig` and other built-in MATLAB functions, however, it does not use any optimization software available in MATLAB or elsewhere. The algorithm is adaptive and uses situation geometry specific search directions and step sizes from its computed geometric data.

In the *preparatory phase* the angles θ for the eigenanalyses of $A(\theta)$—if any are needed—are chosen as follows: the first trial angle θ is the angle between the real axis and the line through the eigenvalue or trace center $a = \text{trace}(A)/n \in F(A)$ for A and p. The lay of the extreme eigenvector generated FOV points in relation to p, whether to the left or right of the line through p and a, is then modified to try and ensure that the second extreme FOV point

evaluation lies on the other side of the line $\overline{ap}$ than the first in case the first eigenanalysis did not encircle p. If p is not encircled after two eigenanalyses and subsequent quadratic form x^*Ax evaluations, a third angle θ is chosen by interpolating between the distances of the computed FOV points from the line $\overline{ap}$. The distances of these FOV points and on which side of the line $\overline{ap}$ they lie are determined from the normal vector of the line $\overline{ap}$ and the sign of vector dot products. If after three such eigenanalyses and FOV point computations there are at least two empty quadrants around p, we conclude that p lies outside $F(A)$. If not and there are points in three quadrants around p but not in four, we select the two known FOV points (and their generating vectors) that form the smallest angle with the nearest edges of the empty quadrant and evaluate FOV points on the elliptical image of the real great circle through their vector generators on $S_n(\mathbb{C})$. We continue the process to evaluate FOV points that interpolate finer and finer between the FOV points that are closest to p. These refined elliptical interpolations are performed up to nine times. Into this process we insert one further set of up to three eigenanalyses of $A(\theta)$ with refined angles from the interpolation data normals. If all this fails to encircle p in all four quadrants, we take p to lie outside of $F(A)$. False negative conclusions are rare and happen only when p is very close (generally of order 10^{-8} or less) to the boundary $\partial F(A)$.

In the *iterative* $O(n^2)$ *phase* we use vectorized and matrix computations rather than `for` loops. A typical example is our—sequentially wasteful, but faster—vectorized MATLAB code for evaluating ellipse FOV points. It interpolates between two FOV points `fxin` and `fxout` and their generating n-vectors `xin` and `xout`. Here is a piece of the code.

```
Fline = [[xin,xout];[fxin,fxout]]; Step = 1/step:1/step:1-1/step; % xout -> xin
XX = xout*ones(1,step-1) + (xin-xout)*Step; % FOV pts of unit vectors from line
XX = XX*diag(1./(diag(XX'*XX).^.5)); Fline = [Fline,[XX;diag(XX'*A*XX).']];
```

Here `XX` contains `step-1` column vectors in $\mathbb{C}^n$ that interpolate from `xout` to `xin` $\in S_n(\mathbb{C})$ along their connecting line segments. Instead of normalizing each of these columns individually inside a more customary `for` loop, we postmultiply `XX` by the convoluted matrix expression `diag(1./(diaXX'*XX).^.5))`. This is a diagonal matrix with the reciprocals of the individual vector norms on its diagonal. Likewise we evaluate the FOV points via the matrix expression `diag(XX'*A*XX).'` in which all mixed bilinear forms $x_i^*Ax_j$ are evaluated in `XX'*A*XX` while only the diagonal entries $x_i^*Ax_i$ are needed. According to MATLAB's Profiler, for large n the vectorized process saves around 70% of the equivalent `for` loop time spent with individual vector normalization and FOV point evaluation on both, single-core and multicore computers. Thus we are sequentially wasteful, but vectorially frugal and save CPU time where it counts. Incidentally for small $n \leqslant 100$, the run times of our algorithm are generally so short that the vectorized parts of our code have no noticeable effect. In the above code lines, `Fline` gathers the complex generating vectors in its first n rows, column by column, followed by the respective FOV points in $\mathbb{C}$ in row $n+1$.

When we compute tilted planar intersections with $S_n(\mathbb{C})$ we use the stable form of the quadratic equations solver explicitly, rather than MATLAB's `roots` function which cannot compete timewise with the stabilized direct quadratic formula. To find a randomly tilted plane through `xin` and `xout` we construct a random unit vector b that is orthogonal to the vector connecting `xin` and `xout` $\in S_n(\mathbb{C})$ and use the point $c = (\text{xout} + \text{xin})/2 + b$. For any point x of a regular partition of the line segment from `xin` to `xout` in $\mathbb{C}^n$ we compute $\alpha \in \mathbb{R}$ so that the vector from c through x lies on the unit sphere of $\mathbb{C}^n$ by solving the quadratic equation $\|c + \alpha(x - c)\|_2^2 = 1$ for α. With $z = x - c$, this leads to the quadratic equation $z^*z \cdot \alpha^2 + 2\alpha\,\mathrm{Re}(c^*z) + c^*c - 1 = 0$. Finally we evaluate the associated FOV point for the

parameter $\alpha = \alpha_{\max}$, the maximal solution, and add its generating vector and associated FOV point data to our data set. This for all x in `XX`.

Note further that when we compute images x^*Ax on a great circle, we need to do so only for half of the great circle, say from `xout` to `xin` and from `xout` to `-xin`, but not beyond, since antipodes x and $-x \in S_n(\mathbb{C})$ have the same FOV image $x^*Ax = (-x)^*A(-x)$.

Finally, since the algorithm starts with pseudo-random FOV points and their pseudo-random generating vectors and then proceeds via random data points and vectors in its iterative phase when intersecting randomly tilted planes with the complex unit sphere, the iterations, their number, and found generating vectors for p will differ for the same input matrix A and point $p \in \mathbb{C}$ with each run of the algorithm.

4. Geometric and numerical output of the algorithm

Here we explore several test examples with our algorithm [15]. This will lead to open questions of theory and numerical efficiency. The computations in examples 1–6 below were done on a 2002 vintage Sunblade 100 with 500 Mhz CPU and 1 GB of memory. Towards the end of this section we also include run time comparisons for various dimensions n on the Sunblade and on a 2007 MAC PRO with two dual core Xeon 2.66 Ghz processors and 2 GB of memory.

Example 1. Our first example involves the test matrix

$$A = \begin{pmatrix} 0.3 & 0.4 & 0.3 \\ \mathrm{i} & 0.5 & 0.5 \\ 0.7 & 0.1 & 0.2 \end{pmatrix}.$$

Here is a typical on-screen output of our algorithm for this matrix and the chosen point $p = -0.2 + 0.02\mathrm{i} \in F(A)$ when the initial 40 pseudo-random FOV points encircle p and no eigenanalysis is needed to prepare for phase 2 iterations.

```
>> [v,fv,error] = wberpoint(A,-.2+.02*i);
    format long, fv, format short
Given point p lies inside F(A)
In iteration 1, the FOV point deviation from p is 0.0169045
In iteration 2, the FOV point deviation from p is 0.00200869
In iteration 3, the FOV point deviation from p is 7.61592e-05
In iteration 4, the FOV point deviation from p is 9.08591e-08
In iteration 5, the FOV point deviation from p is 3.63583e-09
In iteration 6, the FOV point deviation from p is 1.48841e-11
0.00535401 sec prep time for encircling p (using 0 eigenanalyses);
1.03164 sec for 6 iterations to zoom in onto p with given error;
1.037 total time spent.
fv = -0.19999999998930 + 0.02000000001034i
```

Here fv is the computed FOV point near p and v is a complex unit vector that actually generates p to within 1.5×10^{-11}.

Next we look at a case when the initial FOV points do not surround p and we need additional eigenanalyses of $A(\theta)$. Note that both cases may occur for the same A and p in

different runs, due to the random initial process.

```
>> [v,fv,error] = wberpoint(A,-.2+.02*i);
    format long, fv, format short
Given point p lies inside F(A)
In iteration 1, the FOV point deviation from p is 0.0357188
In iteration 2, the FOV point deviation from p is 0.000956193
In iteration 3, the FOV point deviation from p is 1.16118e-05
In iteration 4, the FOV point deviation from p is 4.14772e-07
In iteration 5, the FOV point deviation from p is 1.20478e-09
In iteration 6, the FOV point deviation from p is 2.46781e-11
0.0111204 sec prep time for encircling p (using 2 eigenanalyses);
1.10441 sec for 6 iterations to zoom in onto p with given error;
1.11553 total time spent.
fv = -0.19999999999446 + 0.020000000024405i
```

The input parameters for our MATLAB m file `wberpoint.m` are the matrix A and the desired point p in $\mathbb{C}$, as well as an optional parameter `zeich` that initiates plotting output by setting `zeich = 1`, if desired.

The output parameters are the computed complex unit vector v with $v^*Av = fv \approx p$ and the distance $error$ between the computed FOV point fv and p upon convergence. In both tests for A, the algorithm uses six iterations to compute a complex unit vector $v \in S_3(\mathbb{C})$ whose FOV point fv differs from the desired point p by less than 3×10^{-11} in absolute terms. Besides, we have displayed overall and partial timings of the two phases of our algorithm. Note that each case, with and without eigenanalyses, takes about 1 s of computer time on our Sunblade 100 from 2002 with 500 Mhz CPU speed.

Example 2. Our second example looks at a later stage of the algorithm for a 30 by 30 complex matrix A with random entries and the point $p = 12\mathrm{i}$. Figure 3 shows a subset of the computed FOV points after the initial pseudo-random FOV points (marked by * in figure 3) have failed to encircle p and as the first eigenanalyis of $A(\theta)$ with $\theta \approx \pi/2$ has succeeded in this task. Convergence takes six iterations and about 1 s of CPU time with one eigenanalysis performed for $\theta \approx \pi/2$.

```
>> n=30; wberpoint(A,12*i,1);
Given point p lies inside F(A)
In iteration 1, the FOV point deviation from p is 0.290083
In iteration 2, the FOV point deviation from p is 0.0329377
In iteration 3, the FOV point deviation from p is 0.000679862
In iteration 4, the FOV point deviation from p is 2.21903e-06
In iteration 5, the FOV point deviation from p is 1.20687e-07
In iteration 6, the FOV point deviation from p is 8.51216e-11
0.216275 sec prep time for encircling p (using 1 eigenanalyses);
1.08 sec for 6 iterations to zoom in onto p with given error;
1.29627 total time spent.
```

Example 3. Our next example deals with a matrix $A \in \mathbb{C}^{60,60}$ with random entries and a point p very close to the boundary of $F(A)$. Here is a typical on screen output.

```
>> wberpoint(A,-25+20*i,2);
Given point p lies inside F(A)
In iteration 1, the FOV point deviation from p is 0.0995839
In iteration 2, the FOV point deviation from p is 0.00811658
In iteration 3, the FOV point deviation from p is 4.0238e-05
In iteration 4, the FOV point deviation from p is 1.05494e-06
In iteration 5, the FOV point deviation from p is 3.10774e-09
In iteration 6, the FOV point deviation from p is 1.67944e-10
0.167818 sec prep time for encircling p (using 3 eigenanalyses);
1.77306 sec for 6 iterations to zoom in onto p with given error;
1.94087 total time spent.
3.05302 sec time for drawing boundary of F(A) (40 eigenanalyses);
```

Note the efficiency of the three eigenanalyses in the preparatory phase and the speed of the iterative phase with its six iterations.

In addition to solving the inverse problem, we have also computed the approximate boundary of $F(A)$ to help the illustrations in figures 4 and 5. To evaluate and draw $\partial F(A)$ via 40 eigenanalyses according to [6] takes almost 150% of the total inverse problem effort. Figure 4 shows part of $F(A)$ near p just after we have succeeded to fill the four quadrants around $p \in F(A)$ by interpolating across the NW quadrant between two FOV points marked by o in the SW and NE quadrants around p near the boundary of $F(A)$.

Figure 4 depicts the three current best approximations * to p in three quadrants. Two of these FOV points lie near $\partial F(A)$ in the NE and SW quadrants around p. According to the on-screen output, these points have been computed via three eigenanalyses for matrices $A(\theta)$ and appropriate angles θ. The closest computed FOV point * in the third, the SE quadrant, lies at approximately $-18.2 + 18.9\mathrm{i}$ while the NW quadrant is still empty after the three phase 1 eigenanalyses. When p lies close to the boundary $\partial F(A)$ and precisely one quadrant around p is empty, our algorithm tries to fill the empty quadrant by creating FOV image points of unit vectors on the real great circle that passes through two associated vector generators for FOV points that lie in the adjacent non-empty quadrants. The interpolating FOV points are depicted by small diamonds $\Diamond$ in figures 4 and 5. Three of them lie in the NW quadrant so that now p is encircled and phase 2 of the algorithm can start its iterations.

A more careful analysis of the interpolating points on $\partial F(A)$ near p in figure 5 reveals that our elliptic interpolants give us points in $F(A)$ that lie beyond the originally computed approximation of $\partial F(A)$ (–) which took 40 eigenanalyses of $A(\theta)$ for 40 angles θ from 0 to π.

Example 4. This example shows the iterative phase at its first step for a random matrix $A \in \mathbb{C}^{120,120}$, once the preparatory phase has succeeded, i.e., when there are known unit vector generators for FOV points in each quadrant surrounding p as is the case in figure 6.

Example 5. Next we view FOV point images from tilted planar cuts of $S_n(\mathbb{C})$. Here we double n once more to $n = 240$ and choose two different random matrices in $\mathbb{C}^{n,n}$ for the left- and right-hand plots of figure 7. We depict the corners $\square$ of the quadrilateral that surround the

point p ($*$). The two diagonals from NE to SW and NW to SE are rendered by solid lines—in each plot. The image points on the randomly tilted planar cuts with the complex unit sphere generate non-elliptical data in $F(A)$ as the following pictures clearly indicate.

Recall that the left- and right-hand plots in figure 7 were created for differing random matrices $A \in \mathbb{C}^{240,240}$ and both figures show the very first stage of the iterative non-elliptic steps. The corner points $\square$ in the left-hand plot of figure 7 form a convex quadrilateral while those on the right do not.

Here is a typical on-screen output for $n = 240$ and A consisting of a specified linear combination of random real- and complex-part matrices.

```
>> n=240; A = randn(n)-5*i*rand(n); wberpoint(A,-1+20*i);
Given point p lies inside F(A)
In iteration 1, the FOV point deviation from p is 0.0104015
In iteration 2, the FOV point deviation from p is 6.82556e-05
In iteration 3, the FOV point deviation from p is 6.44217e-07
In iteration 4, the FOV point deviation from p is 2.40343e-08
In iteration 5, the FOV point deviation from p is 2.92079e-10
4.36932 sec prep time for encircling p (using 1 eigenanalyses);
12.9298 sec for 5 iterations to zoom in onto p with given error;
17.2991 total time spent.
```

Example 6. This example uses the same 3 by 3 matrix A as example 1, but now we try to find a generating complex unit vector for the point $p = 0$ for the shifted matrix $B = A - 0.38905 \cdot \sqrt{-1} \cdot I_3$. The shift parameter is chosen so that $p = 0$ lies very close to the boundary $\partial F(B)$ and we can read off distances easily from the MATLAB plot scales. First we list the MATLAB on-screen output of our algorithm.

```
>> A = [.3,.4,.3;i,.5,.5;.7,.1,.2]; B = A - 0.38905*i*eye(3);
    [v,fv] = wberpoint(B,0,1);
Given point p lies inside F(B)
In iteration 1, the FOV point deviation from p is 2.48741e-05
In iteration 2, the FOV point deviation from p is 6.75622e-06
In iteration 3, the FOV point deviation from p is 2.876e-07
In iteration 4, the FOV point deviation from p is 1.13115e-08
In iteration 5, the FOV point deviation from p is 3.4708e-11
0.540669 sec prep time for encircling p (using 6 eigenanalyses);
1.87451 sec for 5 iterations to zoom in onto p with given error;
2.41518 total time spent.
```

The corresponding FOV plots look as follows.

The right-hand plot shows that the chosen point $p = 0$ lies well outside the approximate $\partial F(B)$ curve, drawn as a solid egg-shaped curve. However, p lies inside $F(B)$ at a distance of around 10^{-5} units from $\partial F(B)$. In figure 8 a number of computed interior and boundary FOV points near p are shown in both plots.

Example 7. Our last graphed example involves a single Jordan block A of dimension $n = 188$ for the eigenvalue $1 + 3\mathrm{i}$ with all first upper diagonals set equal to 1. The field of values of A is

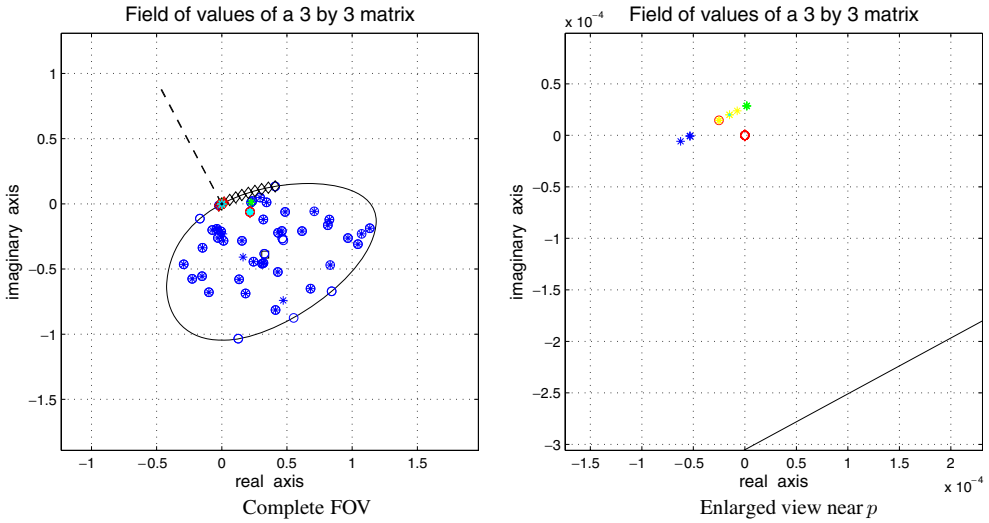

Figure 8. Field of values of a 3 by 3 matrix.

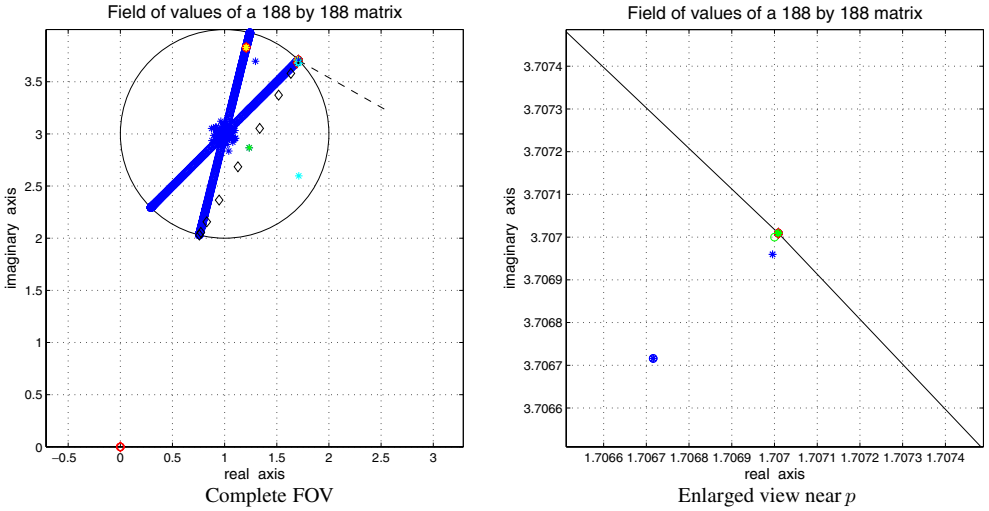

Figure 9. Field of values of a 188 by 188 matrix.

a disk of radius 0.999 861 8546 with center $1+3\mathrm{i}$ as computed according to [13]. We choose p close to the boundary of this disk at $1+3\mathrm{i}+0.707\cdot(1+\mathrm{i})$. Our algorithm produces a complex unit vector x that approximates $p \in \mathbb{C}$ in $fv \approx x^*Ax$ with 10 accurate leading digits.

```
>> n = 188; [x fv abserror] = ...
        wberpoint(diag(ones(n-1,1),1)+(1+3i)*eye(n),1.707+3.707i,1);
         fv, abserror
Given point p lies inside F(A)
fv = 1.706999999622030e+00 + 3.707000000012790e+00i
abserror = 3.781866650860910e-10
```

The graphs for this example look as follows.

Table 1. Runtimes for increasing dimensions.

n	Eigen-analyses	Iterations	Runtimes prepare/iterate	SunBlade100 total	Runtimes prepare/iterate	MAC PRO total
3	22/60	9.2	0.006/0.92	0.92	0.003/0.08	0.09
6	1/60	8.4	0.006/0.94	0.95	0.002/0.08	0.08
12	0/60	9.2	0.007/1.06	1.06	0.01/0.15	0.17
25	1/60	7.9	0.009/1.1	1.11	0.01/0.19	0.21
50	0/60	7.8	0.01/1.52	1.54	0.01/0.47	0.49
100	0/60	7.9	0.03/3.08	3.10	0.02/0.65	0.67
200	0/60	7	0.17/10.37	10.54	0.03/0.98	1.00
400	0/60	7.5	0.46/38.20	38.67	0.04/2.58	2.62
800	0/60	7.4	1.42/185.53	186.95	0.06/9.01	9.08
1500	0/20	6.1	4.32/443.5	447.82	0.15/28.33	28.48
3000	1/10	6.2	–	–	37.23/133.27	170.50
6000	0/10	6.9	–	–	1.77/524.66	526.44

Computations for this example require from 4 to 13 iterations. This number varies due to the random process. Each run involves four eigenanalyses to encircle p in phase 1 and the overall runtimes vary between 1.28 and 2.45 s on the MAC Pro.

We conclude this section with a table of runtimes of our algorithm for varying n, averaged over 30 runs for each of the platforms for $n < 1000$, and averaged over 10 runs for $n \geqslant 1000$. Here we double n until we run out of memory or the runtime exceeds 10 min. The point $p \in F(A) \subset \mathbb{C}$ was chosen near zero and kept fixed, while the matrices A were random complex matrices of increasing dimensions.

The first 'eigenanalyses' column of table 1 lists 'fractional expressions' in the form a/b for the overall number a of eigenanalyses performed in the preparatory phase and of $b = 60$, 20 or 10 runs (combined from both platforms for each n).

For low $n \leqslant 25$ the runtimes on the SunBlade hover around 1 s. This is due to overhead and its single processor CPU. Running MATLAB in multithreaded mode on the MAC PRO, we see only slight runtime increases up to 1 s as long as $n \leqslant 200$. From this point on, runtimes triple or quadruple each time n is doubled. The same holds for the Sun. This reflects the design of our algorithm by evaluating mainly matrix times vector products in $O(n^2)$ time in case eigenvalue and eigenvector evaluations are not needed in the preparation phase to encircle p. Note that the phase 1 preparation times are generally very short compared to the phase 2 iteration times and that we almost uniformly need just 6–9 iterations for all dimensions $n \leqslant 6000$.

However, there appear to be two 'glitches' in table 1:

(1) Why so relatively many eigenanalyses for very low values of n?
(2) Why is the prep time for $n = 3000$ so large compared to $n = 1500$ and $n = 6000$?

Re (1): Our test matrices $A \in \mathbb{C}^{n,n}$ with random complex entries were created via MATLAB's `rand` or `randn` functions. They have increasingly larger fields of values and increasing spectral radii as n increases. For small n the initial randomly created field of values points would often not encircle our fixed point p because it was not sufficiently close to the center point of $F(A)$. Therefore phase 1 eigenanalyses were needed to encircle it. As n increases, the FOV and the initial randomly created FOV points f_i both spread out while p stays fixed and therefore the

f_i encircle p more readily, thus almost never requiring eigenanalyses in tests for larger n in this example.

Re (2): The high average preparation time of 37 s for $n = 3000$ reflects one eigenvalue and eigenvector analysis of a Hermitian matrix at $O(n^3)$ cost in our ten test runs. A complete Hermitian matrix eigenanalysis via MATLAB's `eig` function takes around 344 s of CPU time on the MAC PRO when $n = 3000$. This accounts for the increased prep time average when $n = 3000$ in table 1.

If for a given matrix $A \in \mathbb{C}^{n,n}$, the maximally allowed number of six preparatory eigenanalyses were actually needed to encircle a given point p for $n = 200, 400, 800, 1500, 3000$ or 6000, respectively, then the preparation phase of the algorithm alone would take around 1, 6, 45, 270, 2060, 16 500 s, respectively, to perform these six complete $O(n^3)$ eigenanalyses via MATLAB's built-in `eig` QR algorithm on the MAC PRO. Thus the maximal prep time spent on maximally six eigenanalyses would dominate the CPU time of the iterative phase for $n \geqslant 200$ on the MAC PRO. This shows a disparity of the potentially very expensive $O(n^3)$ preparatory phase when using the QR algorithm and the inexpensive $O(n^2)$ iteration phase of the algorithm which leads us to conjectures and open problems below. One immediate remedy for this is to replace `eig` by the Krylov eigensolver `eigs` for $n > 100$ in our MATLAB code. `eigs` runs in $O(n^2)$ time when computing a few extreme eigenvalues and eigenvectors of a large matrix. This would be especially profitable for the second to last preparatory eigenanalyses that try to encircle p if p is not encircled from the random $f_i = x_i^* A x_i$ evaluations.

5. Open problems and conjectures: generating the field of values; generating vector sets and spaces for a given point or set of points inside the FOV

We first deal with the complexity of approximating the boundary of $F(A)$. Thanks to [6] one can generate a reasonable approximation of the field of values of a matrix $A_{n,n}$ via a moderate number of eigenanalyses for linear combinations $A(\theta)$ of the Hermitian and skew-Hermitian parts matrices of A. This, however, requires $O(n^3)$ effort if we use the QR algorithm in `eig` and slows down our otherwise $O(n^2)$ algorithm; see point (2) above. The images of figures 6 and 7 suggest a faster $O(n^2)$ approach: given the generating unit vectors x and $y \in S_n(\mathbb{C})$ of two FOV points $p_x = x^* A x$ and $p_y = y^* A y$, the points on the real great circle through x and y generate an ellipse in $F(A)$ that contains p_x and $p_y \in F(A)$. Thus one might investigate an expansive $O(n^2)$ process for generating $F(A)$ points all the way to its boundary by judiciously choosing FOV point generators for maximal elliptic (or other) expansion. This idea goes back to [8] where it was only used for small dimensions $n \leqslant 4$. First tests with larger n, however, always seem to run up against an 'inner FOV boundary' that is distinct from $\partial F(A)$ and whose outer curved edge stagnates long before reaching the actual $F(A)$ boundary. How to choose FOV points p_x and p_y more appropriately is open and poses a useful challenge so that our inverse algorithm may become an $O(n^2)$ algorithm even in its preparatory phase. Besides great circles with elliptic FOV images, other curves on $S_n(\mathbb{C})$ might help, such as the slanted planar cuts of $S_n(\mathbb{C})$, several of whose FOV images are shown in figure 7.

This experimentally observed 'inner limit' to finding all of $F(A)$ via elliptic or other $x^* A x$ evaluations of points in $F(A)$ and expansion may be structural or not. If it were structural and there were different—so far unknown—parts of $F(A)$ with differing 'densities' of generating vectors on $S_n(\mathbb{C})$ then here is a second open problem in this realm: what *covering numbers* $k \leqslant n$ are possible for FOV points p? For a general matrix $A \in \mathbb{C}^{n,n}$ we define the covering number $k(p)$ of any point $p \in F(A) \subset \mathbb{C}$ as follows.

Definition. *For a matrix $A \in \mathbb{C}^{n,n}$ and a point $p \in F(A) \subset \mathbb{C}$ we call the maximal size of a linearly independent set of complex unit vectors x with $x^*Ax = p$ the* ***covering number*** *$k(p)$ of p and A.*

The covering number studies the inner geometry of the field of values of a matrix. Finding a set of vectors x with $x^*Ax = p$ is the inverse problem relating the image $F(A)$ of the *quadratic map*

$$x \in \mathbb{C}^n, \|x\|_2 = 1 \quad \longrightarrow \quad x^*Ax \in \mathbb{C}$$

to its domain, the *complex unit sphere* $S_n(\mathbb{C}) = \{x \in \mathbb{C}^n \mid \|x\|_2 = 1\}$.

If A is normal then the covering number has been completely determined for any point $p \in F(A)$ in [14]. In particular $k(p)$ is n for any p inside the field of values' relative interior if A is normal. Regarding the covering number for points in the field of values of a general matrix, here is a list of open theoretical questions:

Which vectors of the unit sphere $S_n(\mathbb{C}) \subset \mathbb{C}^n$ map to the same point $p \in F(A) \subset \mathbb{C}$?
What kind of curve(s) or patch(es) do they describe on $S_n(\mathbb{C})$?
How can we count them?
What covering numbers $k(p)$ are possible for a given matrix $A \in \mathbb{C}^{n,n}$ and varying points $p \in F(A)$? $1 \leqslant ?? \leqslant k(p) \leqslant ?? \leqslant n$.
Is the covering number $k(p)$ equal to n for all points p inside the relative interior of $F(A)$ and all n by n matrices A as it is for normal matrices?
Or are there non-normal matrices A and points p with covering numbers less than n in the relative interior of $F(A)$?
What distinguishes regions or points inside the FOV with the same or with differing covering numbers k, if such differences exist?
Are these FOV 'regions' connected or convex, if they exist?
Is the concept of 'covering number' well defined and is it suitable for studying the generating vector geometry of the field of values?

Partial answers to these questions can be gleamed and conjectured from repeated random runs of our program for one $p \in F(A)$ and the same matrix A and then calling the SVD or rank function in MATLAB on the generating vector column matrix. We have run our algorithm repeatedly for one matrix $A \in \mathbb{C}^{n,n}$ and one fixed point $p \in F(A)$ for $2n$ times. The rank of the resulting n by $2n$ matrix with the computed generating vectors of p in its columns has always turned out to be n numerically. But apart from the normal matrix case in [14], we know of no proof for this conjecture for general matrices A.

Acknowledgments

Some of this work was inspired by and created for the 8th Workshop on Numerical Ranges and Radii, WONRA 2006, at the University of Bremen, Germany. I thank the three referees for their detailed comments, their very useful ideas and patience.

References

[1] Bendixson I 1902 Sur les racines d'une équation fondamentale *Acta Math.* **25** 358–65
[2] Davis C 1971 The Toeplitz–Hausdorff theorem explained *Can. Math. Bull.* **14** 245–46
[3] Fiedler M 1981 Geometry of the range of a matrix *Linear Algebr. Appl.* **37** 81–96
[4] Higham N J, Tisseur F and van Dooren P M 2002 Detecting a definite hermitian pair and a hyperbolic or elliptic quadratic eigenvalue problem, and associated nearness problems *Linear Algebr. Appl.* **351–352** 455–74

[5] Horn R A and Johnson C R 1991 *Topics in Matrix Analysis* (Cambridge: Cambridge University Press)
[6] Johnson C R 1978 Numerical determination of the field of values of a general complex matrix *SIAM J. Numer. Anal.* **15** 595–602
[7] Jonckheere E A, Ahmad F and Gutkin E 1998 Differential topology of numerical range *Linear Algebr. Appl.* **279** 227–54
[8] Marcus M and Pesce C 1987 Computer generated numerical ranges and some resulting theorems *Linear Multilinear Algebr.* **20** 121–57
[9] Mengi Emre and Overton Michael A 2005 Algorithms for the computation of the pseudospectral radius and the numerical radius of a matrix *IMA J. Num. Anal.* **25** 648–69
[10] Taussky O 1967 Positive-definite matrices *Inequalities* ed O Shisha (New York: Academic) pp 309–19
[11] Uhlig F 1979 A recurring theorem about pairs of quadratic forms and extensions: a survey *Linear Algebr. Appl.* **25** 219–37
[12] Uhlig F Geometric computation of the Crawford number (unpublished)
[13] Uhlig F Geometric computation of the numerical radius of a matrix (unpublished)
[14] Uhlig F On the covering number of points in the field of values of a matrix, the normal case (unpublished)
[15] Uhlig F 2008 *MATLAB m-file* `wberpoint.m` (available at http://www.auburn.edu/∼uhligfd/m_files/wberpoint.m)