

Application: LORAN

- LORAN – now obsolete, but was an important aid to ship navigation.
- <https://www.youtube.com/watch?v=PDtHuIWGMGg>



- Computing your position requires solving a system of nonlinear equations.

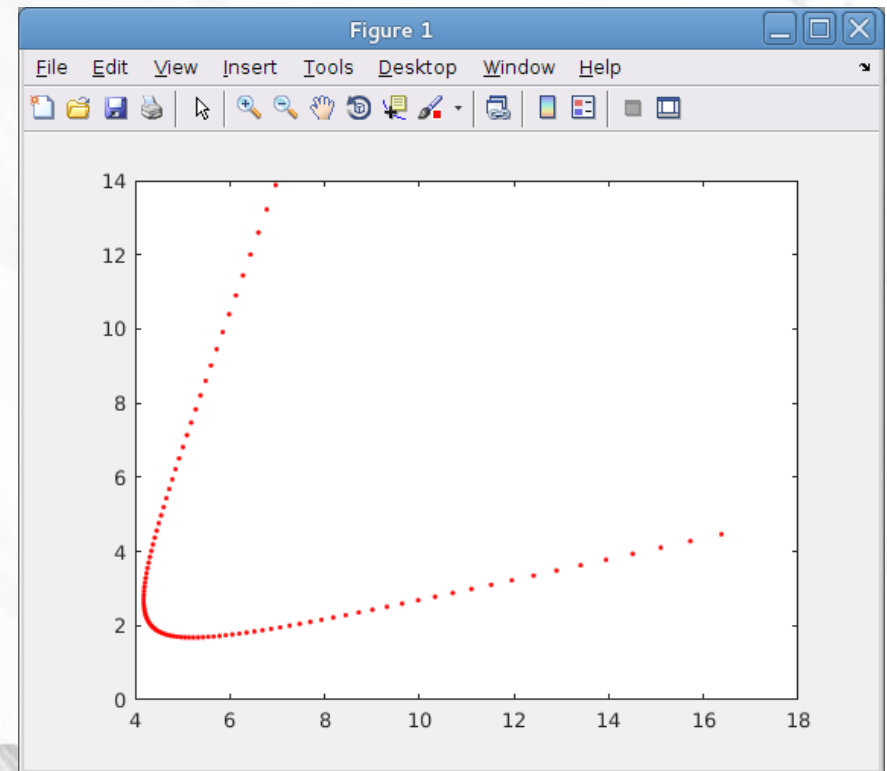
Hyperbola

- Arbitrary hyperbola as quadratic equation in Cartesian coordinates

$$f(x, y) = A_{xx}x^2 + 2A_{xy}xy + A_{yy}y^2 + 2B_x x + 2B_y y + C = 0$$

- Constraints on coefficients ensure this describes a hyperbola (as opposed to some other conic section)

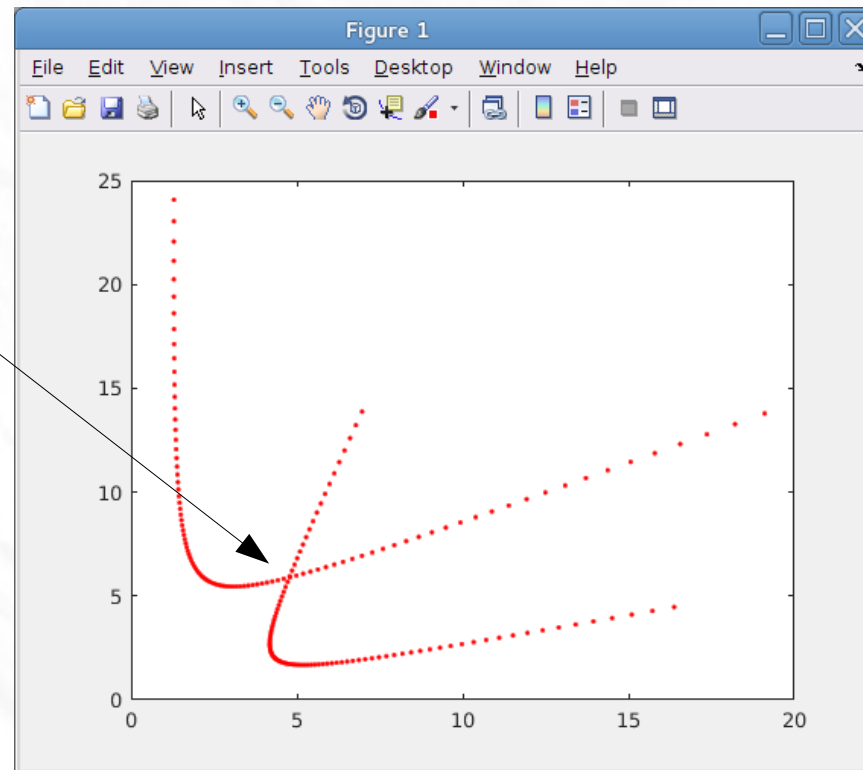
Axx = -0.288223;
Axy = 0.538223;
Ayy = -0.288223;
Bx = 0.595558;
By = -1.470558;
C = -2.051395;



LORAN problem

- To solve LORAN problem, we want to find point $[x, y]$ where two hyperbolas intersect

Find this point



- Knowing $[x, y]$ we know our ship's position.

To use Newton's Method...

- Write down the nonlinear system to solve.

$$f_1(x, y) = A_{xx}^1 x^2 + 2 A_{xy}^1 x y + A_{yy}^1 y^2 + 2 B_x^1 x + 2 B_y^1 y + C^1 = 0$$

Note that the upper numbers are indexes, not powers.

$$f_2(x, y) = A_{xx}^2 x^2 + 2 A_{xy}^2 x y + A_{yy}^2 y^2 + 2 B_x^2 x + 2 B_y^2 y + C^2 = 0$$

Axx1 = -0.288223;
Axy1 = 0.538223;
Ayy1 = -0.288223;
Bx1 = 0.595558;
By1 = -1.470558;
C1 = -2.051395;

Axx2 = -0.231738;
Axy2 = 0.197944;
Ayy2 = -0.003172;
Bx2 = -0.362094;
By2 = -0.188428;
C2 = -0.072621;

Using Newton's Method....

- Write down the Jacobian matrix

$$\frac{\partial f_1}{\partial x} = 2 A_{xx}^1 x + 2 A_{xy}^1 y + 2 B_x^1$$

$$\frac{\partial f_1}{\partial y} = 2 A_{yy}^1 y + 2 A_{xy}^1 x + 2 B_y^1$$

$$\frac{\partial f_2}{\partial x} = 2 A_{xx}^2 x + 2 A_{xy}^2 y + 2 B_x^2$$

$$\frac{\partial f_2}{\partial y} = 2 A_{yy}^2 y + 2 A_{xy}^2 x + 2 B_y^2$$

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix}$$

Newton's Method moving parts

- System of equations

$$f_1(x, y) = 0$$

$$f_2(x, y) = 0$$

- Jacobian matrix

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix}$$

- Starting point $u_0 = [x_0, y_0]$

- Iteration:

- Do linear solve to get step $\delta = -J^{-1}f$
- Check for convergence by examining step size. if $|\delta| < \text{tol}$
- Then either step to next point or return.

$$u_{n+1} = u_n + \delta$$

Using Newton's Method...

- Choose starting point, then iterate.

```
for cnt = 1:100
    % Compute fcn at this point.
    f1 = Axx1*p(1)*p(1) + 2*Axy1*p(1)*p(2) + Ayy1*p(2)*p(2) + 2*Bx1*p(1) + 2*By1*p(2) + C1;
    f2 = Axx2*p(1)*p(1) + 2*Axy2*p(1)*p(2) + Ayy2*p(2)*p(2) + 2*Bx2*p(1) + 2*By2*p(2) + C2;
    f = [f1;f2];

    % Now compute Jacobian at this point
    df11 = 2*Axx1*p(1) + 2*Axy1*p(2) + 2*Bx1;
    df12 = 2*Axy1*p(1) + 2*Ayy1*p(2) + 2*By1;
    df21 = 2*Axx2*p(1) + 2*Axy2*p(2) + 2*Bx2;
    df22 = 2*Axy2*p(1) + 2*Ayy2*p(2) + 2*By2;
    J = [df11, df12; df21, df22];

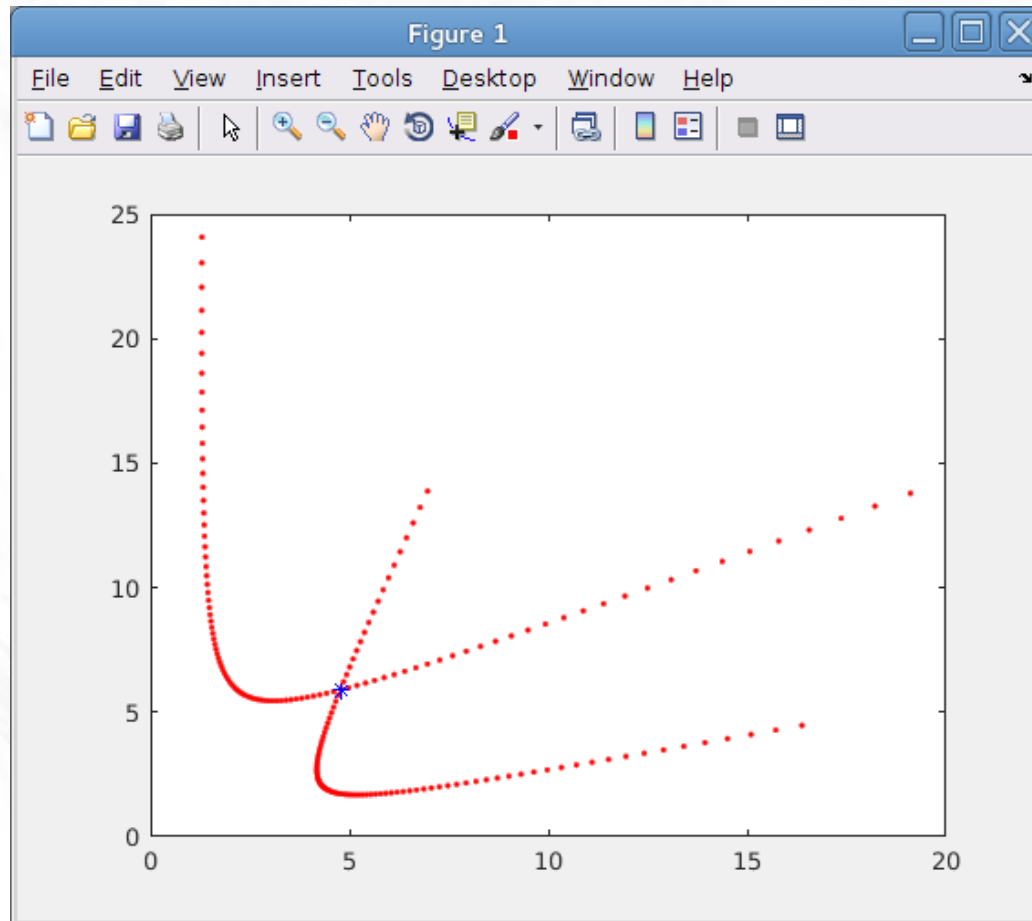
    % Now solve using Newton's method
    delta = -J\f;
    p = p + delta;

    % Test for convergence
    if (norm(delta) < tol)
        fprintf('Found solution at p = [%f, %f]\n', p(1), p(2))
        plot(p(1), p(2), 'b*')
        return
    end

    % Didn't converge -- iterate again
end
```

Linear solve

Result



Found solution at $p = [4.772893, 5.895822]$

Newton's method in ND

1. Write down system to solve. $f_1(x, y) = 0$

$$f_2(x, y) = 0$$

2. Write down derivatives
and Jacobian

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix}$$

3. Guess an initial position $u_0 = [x_0, y_0]$

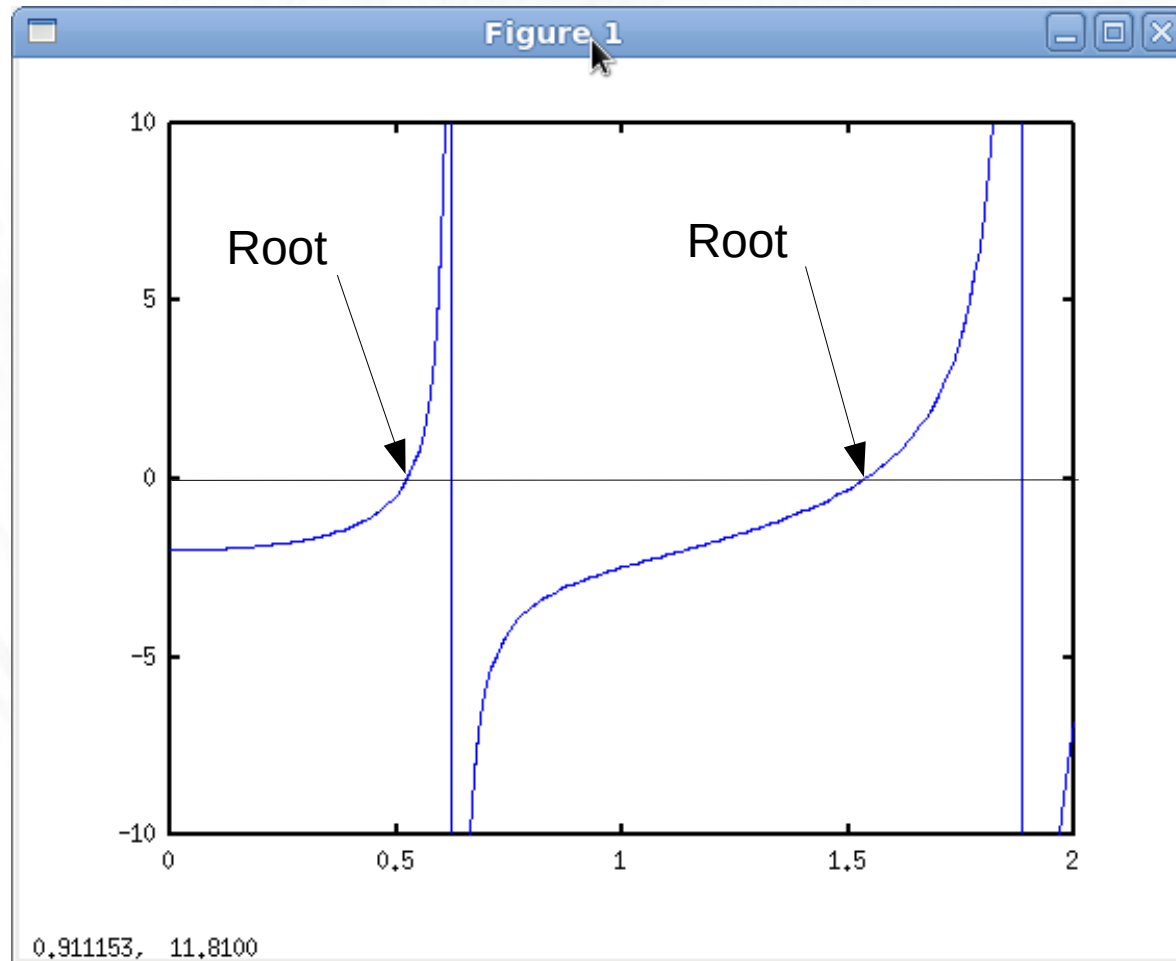
4. Iterate $\delta = -J^{-1}f$ $u_{n+1} = u_n + \delta$

5. Hopefully converge on a solution.

Next topic: Convergence Domain of Newton's Method

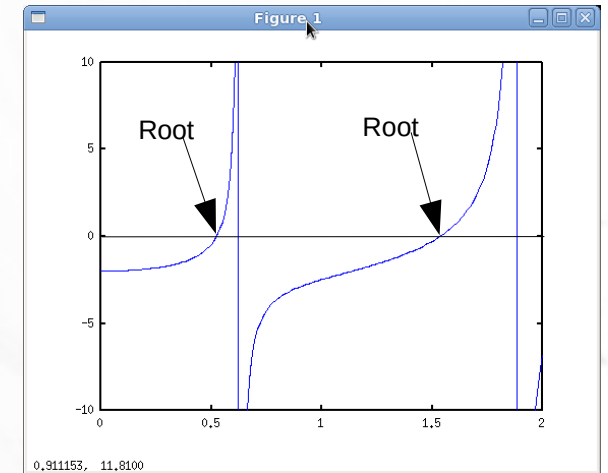
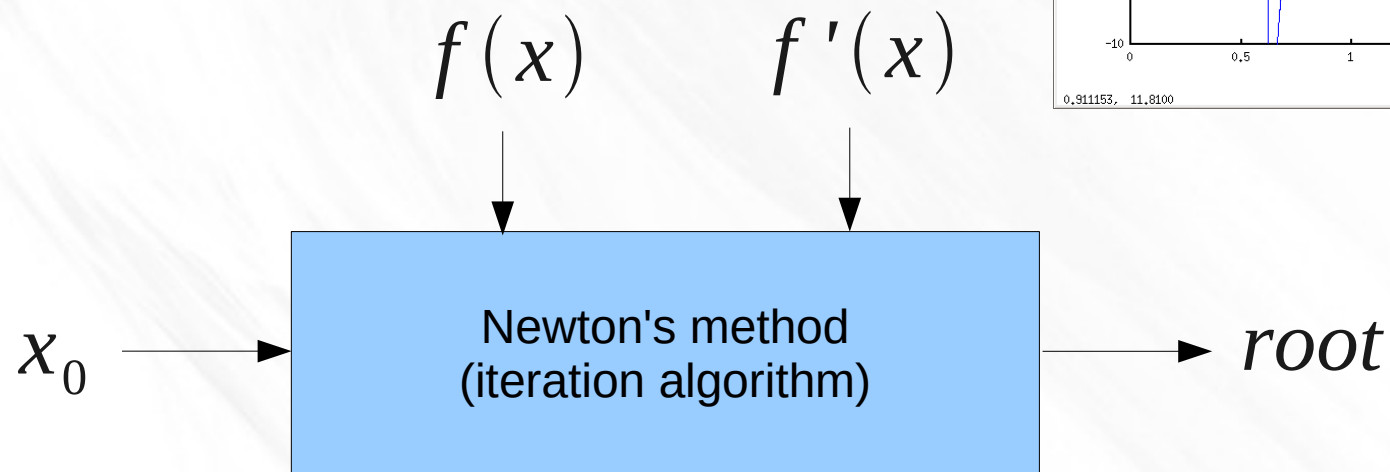
- Recall that Newton's method is very sensitive to the initial guess you give it.
- Different start points can converge to different roots.
- Concept: Convergence domain.
 - This is the domain over which a particular initial guess converges to a particular output.

Recall this 1D function



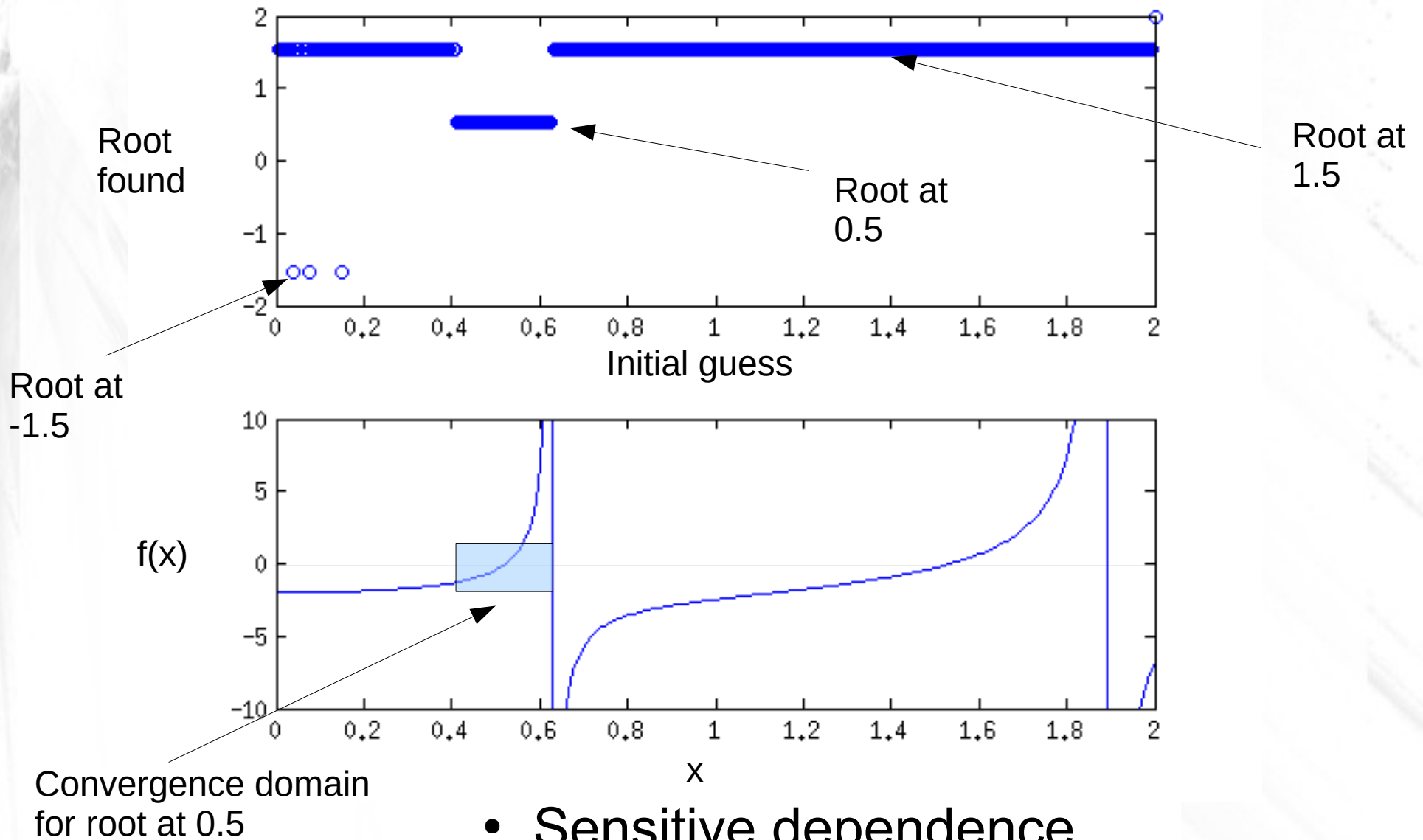
$$f(k) = k \tan\left(\frac{kL}{2}\right) - \sqrt{\beta^2 - k^2}$$

Consider Newton's Method as a mapping from initial guess to root



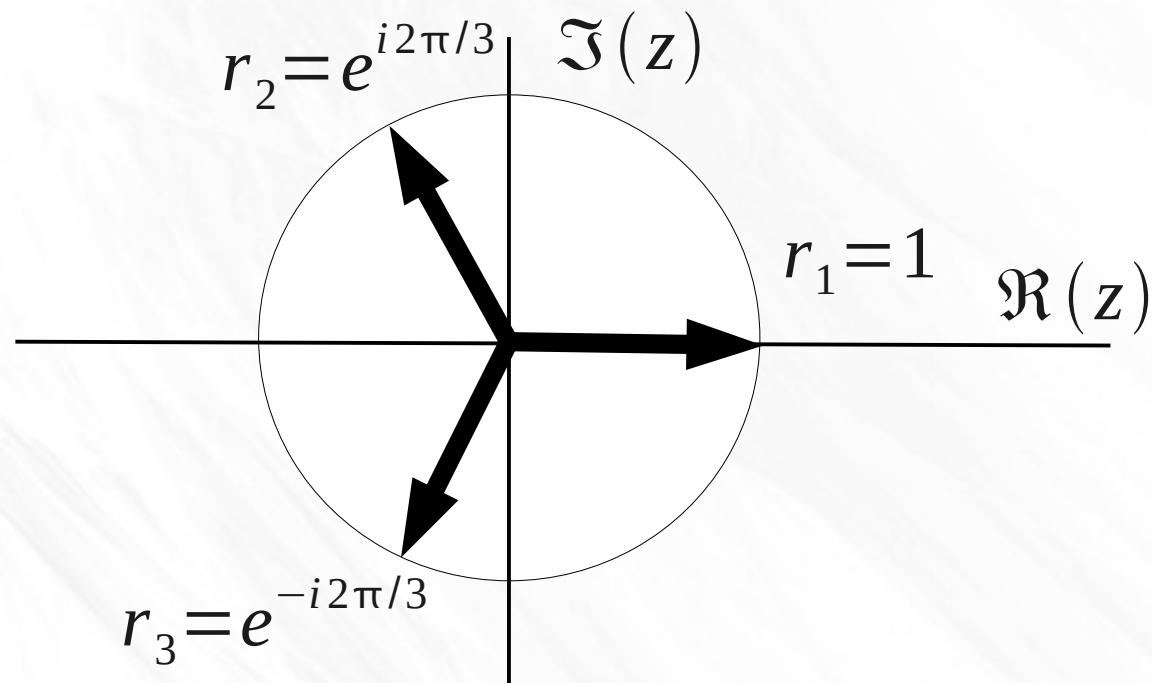
- What happens if there is more than one root?

Convergence of initial guess



A different system to solve: Consider simple cubic equation

- Roots of $x^3 - 1 = 0$



- These are called the “Roots of unity”

Newton iteration for $x^3 - 1 = 0$

- Function: $f(x) = x^3 - 1 = 0$

Note that Newton's method works equally well for complex numbers as for reals.

- Derivative: $f'(x) = 3x^2$

- Newton's method (1D):
$$x_{n+1} = x_n - \frac{f(x)}{f'(x)} \Big|_{x=x_n}$$

- Newton iteration for this $f(x)$:
$$x_{n+1} = x_n - \frac{x_n^3 - 1}{3x_n^2}$$

- Question: Where does an initial guess x_0 end up? $r_1 = 1$ $r_2 = e^{i2\pi/3}$ $r_3 = e^{-i2\pi/3}$

Code

- Colors complex plane depending upon where point converges.

```
tol = 1e-5;    % Convergence tol for Newton's method
for ridx = 1:NX
    for iidx = 1:NY
```

```
        % Create starting point in complex plane, then call Newton's method
        re = re_vec(ridx);
        im = im_vec(iidx);
        z0 = complex(re, im);
        z = newton1D(@f, @df, z0, tol);
```

Start
 x_0

```
        % Look at final value and assign color appropriately.
        if (abs(angle(z)) < .1)
            P(ridx, iidx) = 0;          % Blue
        elseif ((abs( angle(z) - 2*pi/3) ) < .1)
            P(ridx, iidx) = 255;        % Green
        elseif ((abs( angle(z) + 2*pi/3) ) < .1)
            P(ridx, iidx) = 127;        % Red
        end
```

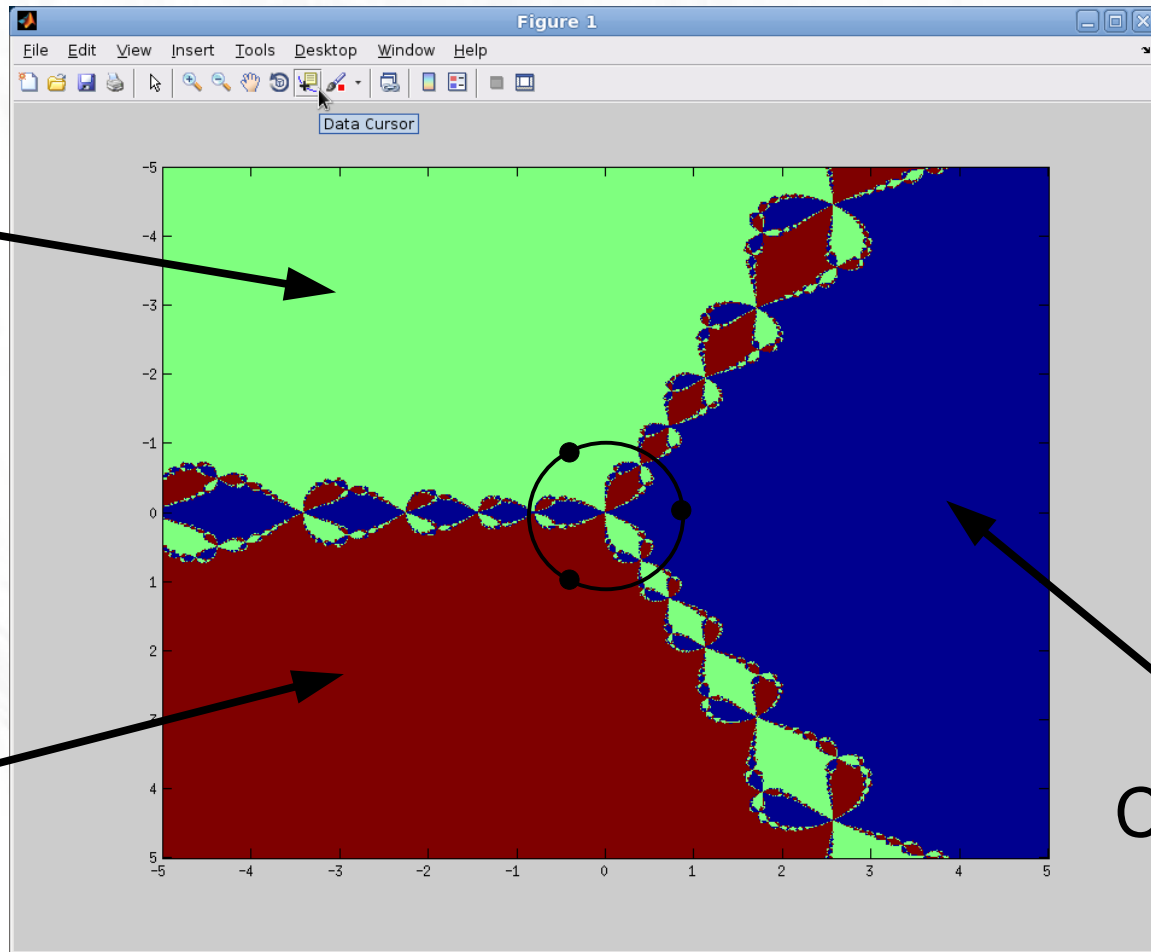
Iterate

$$x_{n+1} = x_n - \frac{x_n^3 - 1}{3x_n^2}$$

```
    end
end
```

Converge $r_1 = 1$ $r_2 = e^{i2\pi/3}$ $r_3 = e^{-i2\pi/3}$

Start Newton's method from different points in complex plane



Converges to

$$r_2 = e^{i2\pi/3}$$

Converges to

$$r_3 = e^{-i2\pi/3}$$

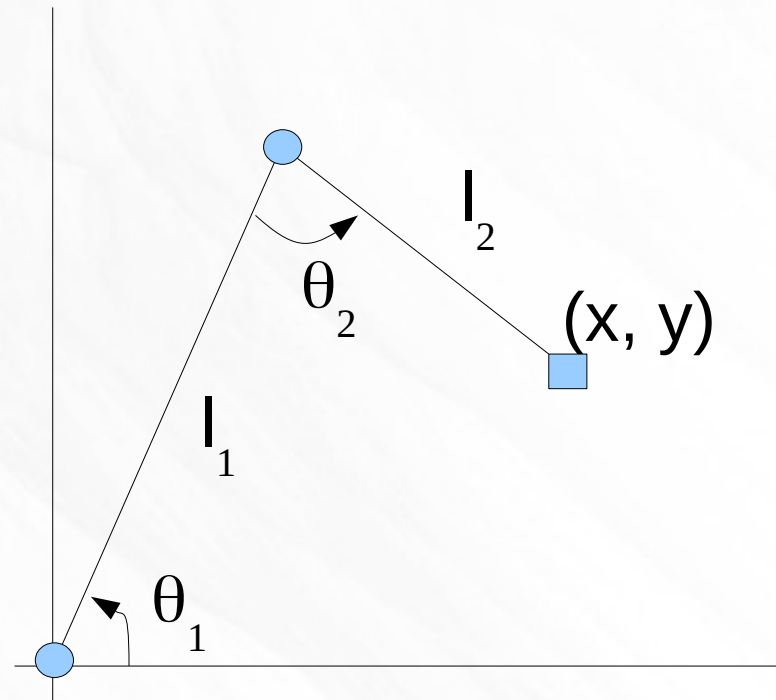
Converges to

$$r_1 = 1$$

- Boundaries of convergence domain are a fractal for $x^3 - 1 = 0$.

Recall the robot problem

- Arm with two rods.
- We control angles θ_1 and θ_2 .
- Want to move “hand” to position (x, y) .
- Must find θ_1 and θ_2 to correctly position “hand”.



Nonlinear system to solve

Function system to solve

$$\vec{f}(\theta_1, \theta_2) = \begin{pmatrix} -l_2 \cos(\theta_2 + \theta_1) + l_1 \cos(\theta_1) - x \\ -l_2 \sin(\theta_2 + \theta_1) + l_1 \sin(\theta_1) - y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

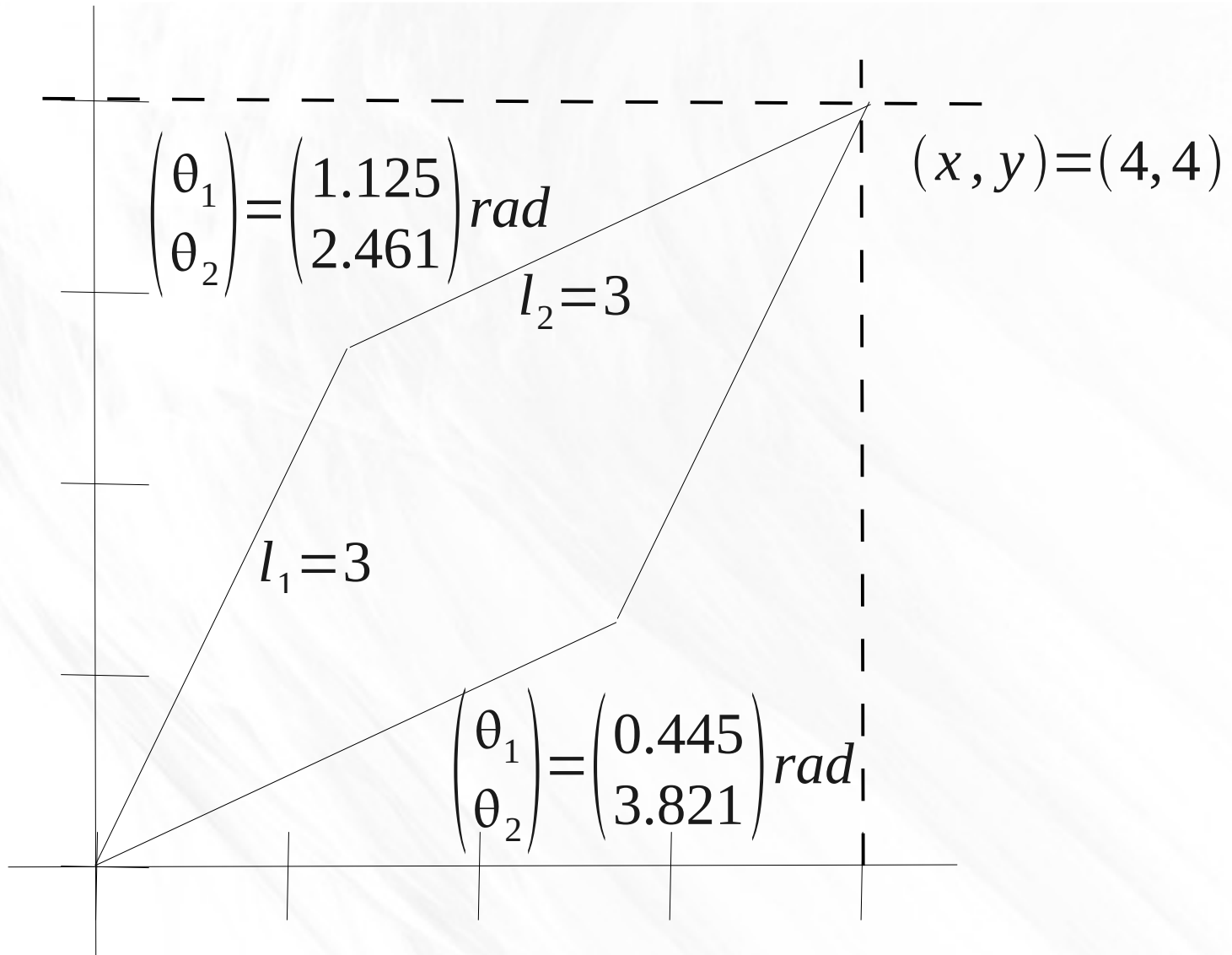
Jacobian matrix

$$J = \begin{pmatrix} l_2 \sin(\theta_2 + \theta_1) - l_1 \sin(\theta_1) & l_2 \sin(\theta_2 + \theta_1) \\ -l_2 \cos(\theta_2 + \theta_1) + l_1 \cos(\theta_1) & -l_2 \cos(\theta_2 + \theta_1) \end{pmatrix}$$

Iteration to implement

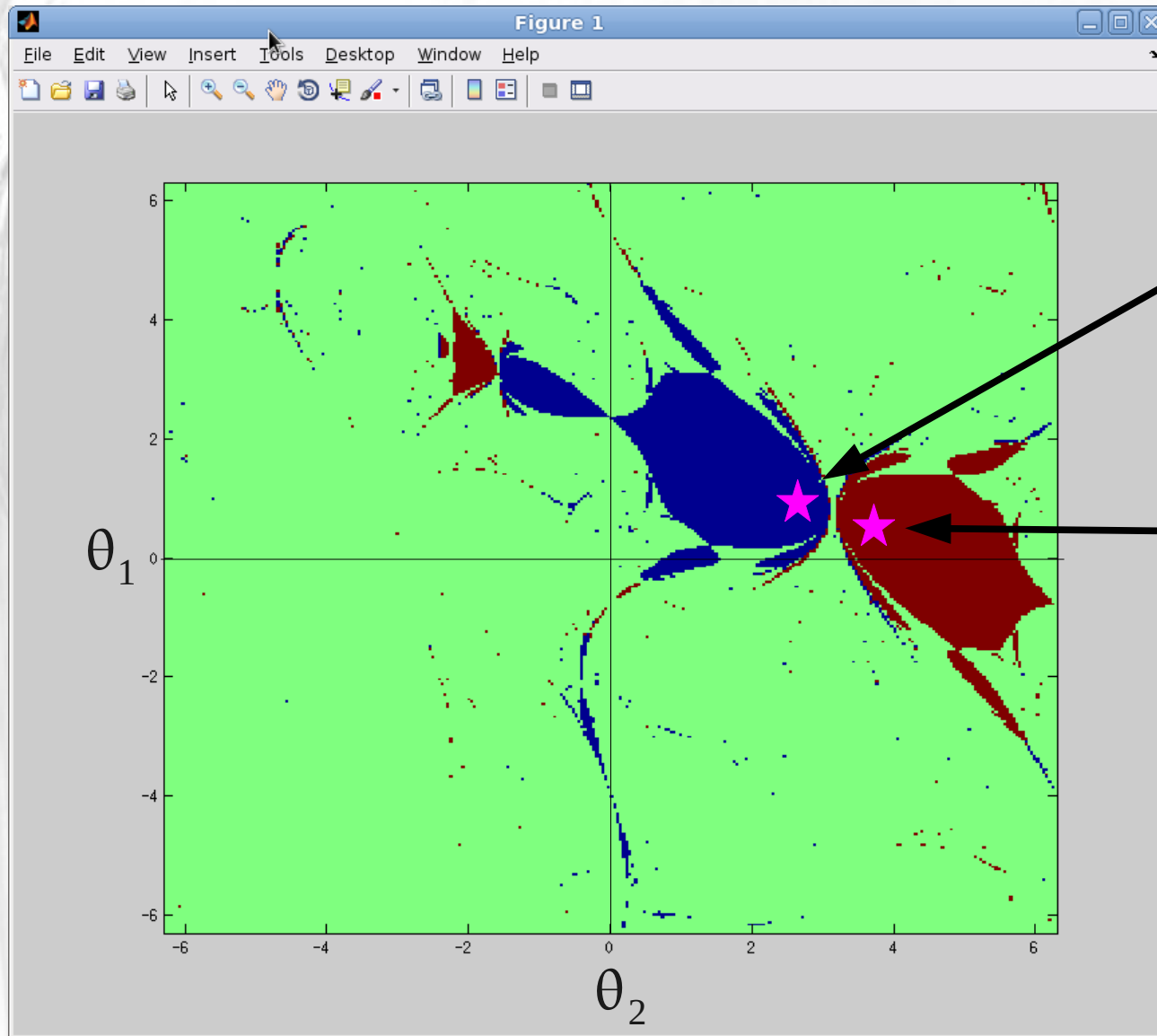
$$\begin{pmatrix} \theta_1^{n+1} \\ \theta_2^{n+1} \end{pmatrix} = \begin{pmatrix} \theta_1^n \\ \theta_2^n \end{pmatrix} - (J(\theta_1^n, \theta_2^n))^{-1} \cdot \vec{f}(\theta_1^n, \theta_2^n)$$

Two solutions for $(x, y) = (4, 4)$



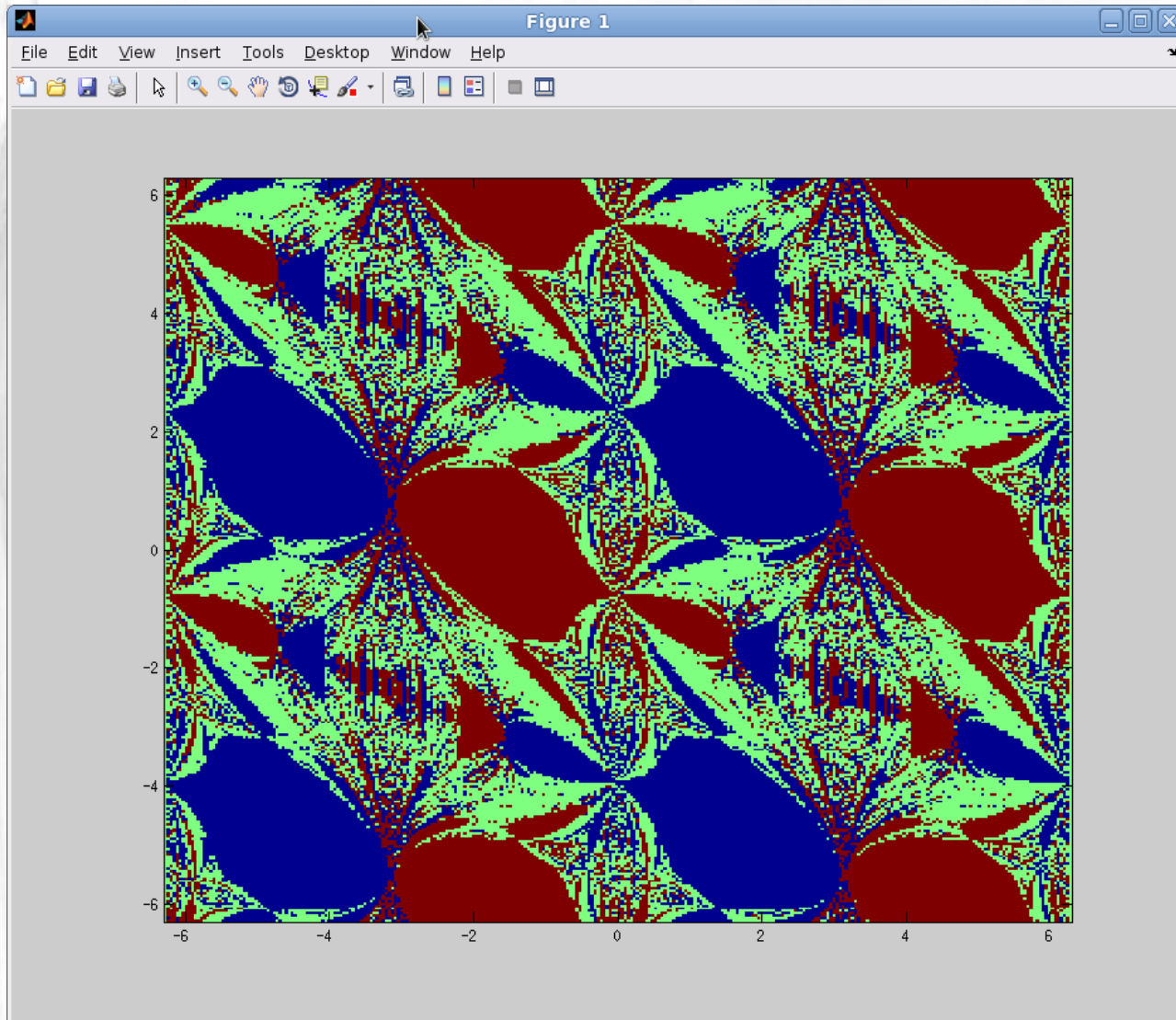
Solution we reach depends upon initial guess

Domain of convergence for each solution



- Blue: Converges to
$$\begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 1.125 \\ 2.461 \end{pmatrix} rad$$
- Red: Converges to
$$\begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 0.445 \\ 3.821 \end{pmatrix} rad$$
- Green: Converges to neither.

If you consider solutions mod 2π ...







- Blue: Converges to
$$\begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 1.125 \\ 2.461 \end{pmatrix} \bmod 2\pi$$
- Red: Converges to
$$\begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 0.445 \\ 3.821 \end{pmatrix} \bmod 2\pi$$
- Green: Converges to neither.

Why bother with Newton's method?

- Behavior of method is complex.
- Achieving convergence can be difficult if you don't know anything about your roots.
- You should start iteration as close to the root as possible for decent results.
- If your function is complicated, iteration can jump all over the place.
- But if you can start “close enough” to your root, convergence is extremely fast.

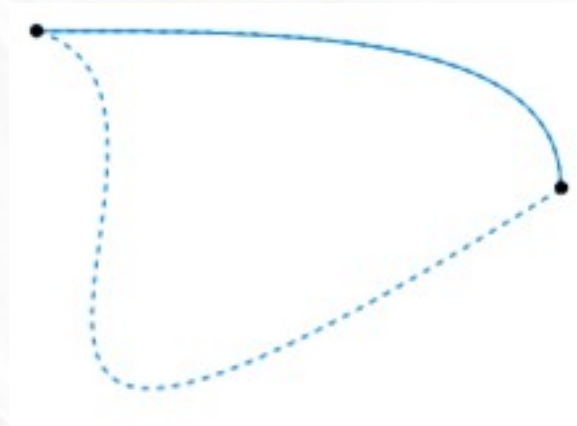
$$err_{n+1} = K (err_n)^2 \quad \text{Quadratic convergence}$$

Fix for the convergence problem?

- Homotopy methods
- We have difficult function $f(x)$  Don't know what initial value of x to use.
- Start with easy function $g(x)$  Know the locations of the roots.
- Then solve the equation:
 $(1-t)g(x) + t f(x) = 0$  Difficult problem
- Start with $t = 0$, then slowly walk to $t = 1$
- Sounds good in theory, sometimes tricks are required to make it work...

Concept of homotopy

- Term comes from topology
- Refers to continuous deformation of a path from point A to point B



- Two paths which are connected via a continuous mapping are “homotopic” to each other.

Example: 4th degree poly with random coefficients

- Find roots of

$$f(x) = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

- With randomly chosen real coefficients

$$[a_4, a_3, a_2, a_1, a_0]$$

- What we know:

- 1 dimensional problem
- 4 roots
- Complex roots come in conjugate pairs
- Roots can lie anywhere in complex plane

Where to start Newton's method?

- Start at 0?
 - You will always converge to the same root, missing the other 3.
- Choose 4 random starting points in complex plane?
 - No idea where roots like, so finding all roots is not guaranteed (or even likely).
- Cover the complex plane with a mesh of roots?
 - Lots of work, and no guarantee of finding all roots if two roots are spaced closer than your mesh spacing.

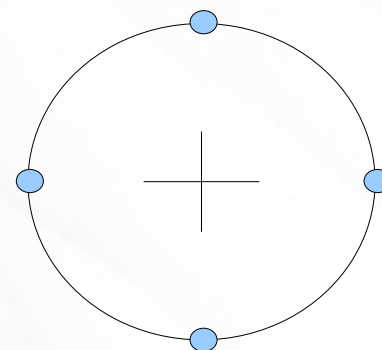
Idea: Start with known equation and solutions

- Start with

$$g(x) = x^4 - 1 = 0$$

- Roots

$$x = [1, -1, i, -i]$$



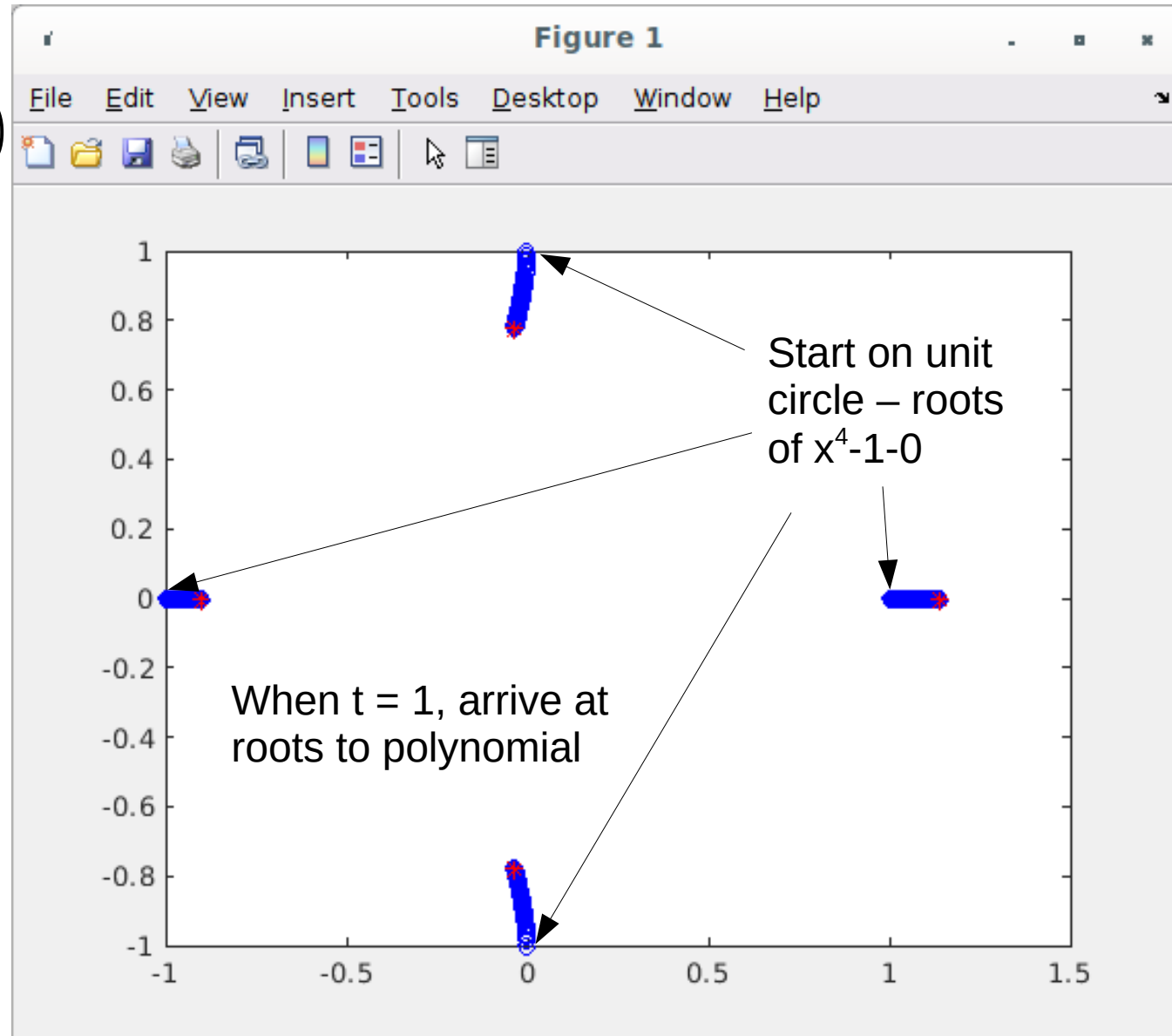
- Use these roots as start point using Newton's method to solve:

$$(1-t)g(x) + tf(x) = 0$$

- Start at $t = 0$ and walk to $t = 1$

Demo

- Find roots of $(1-t)(x^4-1)+tf(x)$
- When $t = 0$, start at roots of (x^4-1)
- When $t = 1$, end at roots of $f(x)$ 4th degree poly
- Doesn't always work....



The gamma trick

Homotopy Methods for Solving Polynomial Systems
tutorial at ISSAC'05, Beijing, China, 24 July 2005

Jan Verschelde*

draft of March 15, 2007

Abstract

Homotopy continuation methods provide symbolic-numeric algorithms to solve polynomial systems. We apply Newton's method to follow solution paths defined by a family of systems, a so-called homotopy. The homotopy connects the system we want to solve with an easier to solve system. While path following algorithms are numerical algorithms, the creation of the homotopy can be viewed as a symbolic operation. The performance of the homotopy continuation solver is primarily determined by its ability to exploit structure of the system. The focus of this tutorial is on linking recent algorithms in numerical algebraic geometry to the software package PHCpack.

- As proposed, the homotopy method tends to fail on polynomials.
- A trick: Multiply one term by random complex number γ :

$$\gamma(1-t)g(x) + tf(x) = 0$$

Code details

```
function y = find_roots_homotopy(f, df, g, dg, u0)
    u = u0;
    I = eye(size(dist(u)));
    for t = 0:.01:1
        d = dist(u) + I;
        h = @(x) t*f(x) + gamma*(1-t)*g(x);
        dh = @(x) t*df(x) + gamma*(1-t)*dg(x);
        u = find_roots(h, dh, u);
        figure(1)
        plot(real(u), imag(u), 'bo');
        hold on
        pause(.1)
    end
    y = u;
end
```

Use t to walk function from g(x) to f(x)

Use 1D Newton and starting point u to find roots. U is updated when find_roots returns

test_find_roots_homotopy

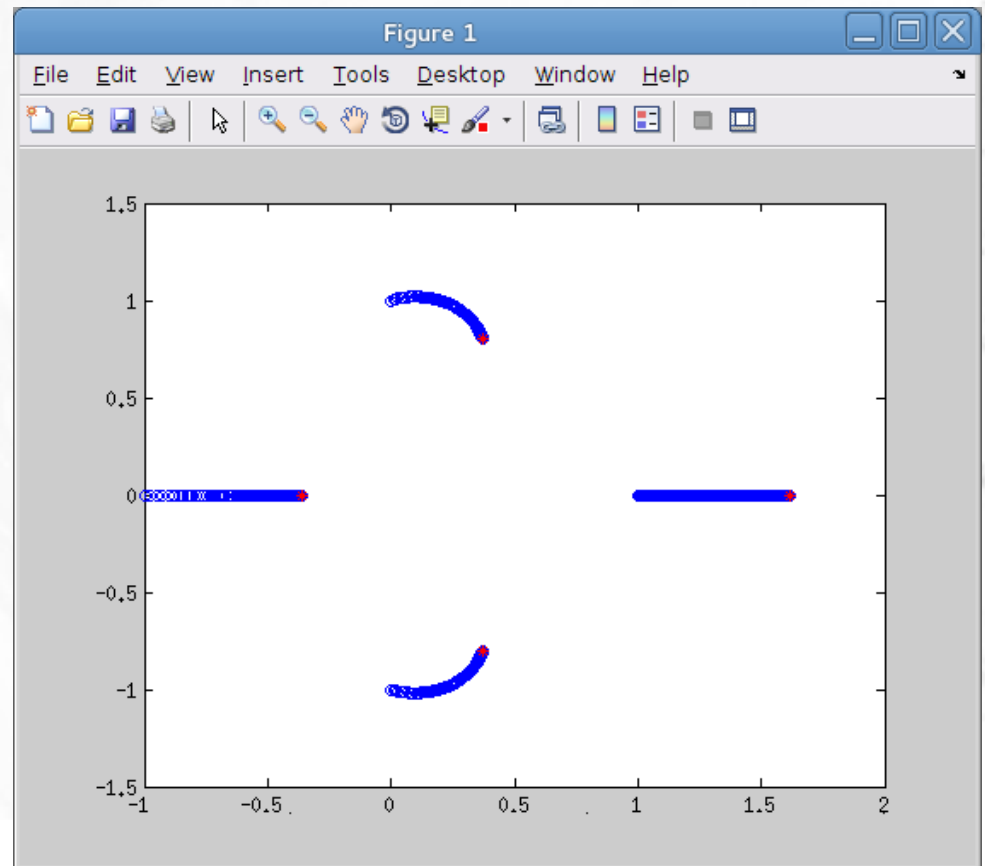
find_roots_homotopy

find_roots

newton1D

Successful run

- Homotopy run followed all roots – tracked true result.
- Direct Newton lost one root



```
>> test_find_roots_homotopy
```

```
Homotopy roots =
```

```
-0.3577 + 0.0000i    0.3697 - 0.8046i    0.3697 + 0.8046i    1.6181 + 0.0000i
```

```
Direct roots =
```

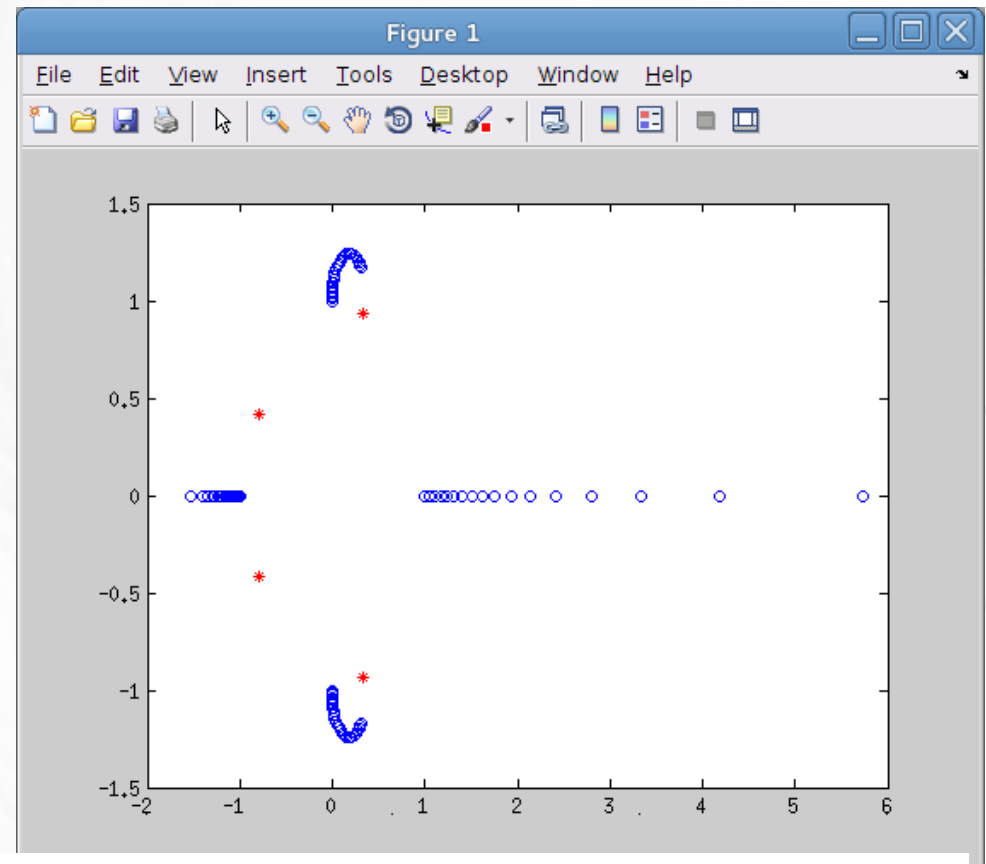
```
-0.3577 + 0.0000i    -0.3577 + 0.0000i    0.3697 - 0.8046i    0.3697 + 0.8046i
```

```
Matlab roots =
```

```
-0.3577 + 0.0000i    0.3697 + 0.8046i    0.3697 - 0.8046i    1.6181 + 0.0000i
```


Failed run

- Function went singular (root at infinity) – homotopy method couldn't follow.
- This run did not use the convergence trick.



```
>> test_find_roots_homotopy
Homotopy method did not converge!
Direct Newtons method did not converge!
Matlab roots =
    -0.8006 + 0.4204i    -0.8006 - 0.4204i     0.3234 + 0.9385i     0.3234 - 0.9385i
```

Homotopy remarks

- Homotopy – sometimes doesn't work.
 - Complex number trick helps for the polynomial root finder.
- If function $h(x) = (1-t) g(x) + t f(x)$ becomes singular for any t , then the method will fail.
- But sometimes it's the only hope you've got....

Next: Broyden's method

- Newton's method requires computation of Jacobian at every step.
- Requires knowing analytic Jacobian, and implementing large Jacobian matrix.
- What if you don't have (or don't want to compute) the whole Jacobian?
- Broyden's method – Compute Jacobian once (at beginning), then compute approximate updates to Jacobian at every step.

Newton vs. Broyden

Newton

0. Compute initial Jacobian

$$\mathbf{J}^{-1}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

1. Compute step using Jacobian:

$$\vec{x}_{n+1} = \vec{x}_n - \mathbf{J}^{-1}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n} \cdot \vec{f}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

2. Compute new Jacobian

$$\mathbf{J}^{-1}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

3. Check for convergence.

4. Loop back to 1.

Broyden

0. Compute initial Jacobian

$$\mathbf{B}^{-1}(\vec{x}) \Big|_{\vec{x}=\vec{x}_0}$$

1. Compute step using \mathbf{B} :

$$\vec{x}_{n+1} = \vec{x}_n - \mathbf{B}^{-1}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n} \cdot \vec{f}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

2. Update \mathbf{B} :

$$\mathbf{B}^{-1}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

← An approximate
“thing” which
behaves like
Jacobian

3. Check for convergence.

4. Loop back to 1.

What is B?

- Consider Taylor approximation approximation for $f(\mathbf{x})$:
$$\vec{f}(\vec{x}) = \vec{f}(\vec{x}_k) + \mathbf{J}(\vec{x}_k)(\vec{x} - \vec{x}_k)$$
- Iterating Newton's method creates converging sequence of Jacobians which become “close” to each other (in some sense) as they converge:

$$\mathbf{J}(\vec{x}_0), \mathbf{J}(\vec{x}_1), \mathbf{J}(\vec{x}_2), \mathbf{J}(\vec{x}_3), \dots$$

- Suppose we replace Jacobians with approximation \mathbf{B} which obeys

$$\vec{f}(\vec{x}_k) = \vec{f}(\vec{x}_{k-1}) + \mathbf{B}_k(\vec{x}_k - \vec{x}_{k-1})$$

Definition of B



How to find B?

- Rewrite definition of B:

$$\vec{f}(\vec{x}_k) - \vec{f}(\vec{x}_{k-1}) = \mathbf{B}_k (\vec{x}_k - \vec{x}_{k-1})$$

Δ_k δ_k

- So, we have constraint on B:

$$\Delta_k = \mathbf{B}_k \delta_k$$

Constraint 1.
Note, this is secant
condition in 1D

- Suppose we know x_k and x_{k-1} . Then we can get Δ_k and δ_k .
- However, B is our unknown. Also since Δ_k and δ_k are vectors, B is underdetermined.

Need another constraint on B

- B is underconstrained. Another plausible condition to ask of B: Choose B_k to be “close” to B_{k-1}

$$\text{minimize } \|B_k - B_{k-1}\|_F$$

Use Frobenius norm

- Frobenius norm for a matrix:

Elements of B

Trace

$$\|B\|_F = \sqrt{\sum_{i=1}^N \sum_{j=1}^N |b_{ij}|^2} = \sqrt{\text{Tr}(B B^T)} = \sqrt{\text{Tr}(B^T B)}$$

- Constraint: Update minimizes Frobenius norm of the update matrix.

Require rank 1 change to matrix

- Consider how to update B

$$B_k = B_{k-1} + uv^T$$

Rank 1 update.
u, v unknown at this
time.

- The object uv^T is a matrix. But each row/col is a scalar multiple of the first row/col.
 - Matrix uv^T has rank 1

- Therefore,

$$\|B_k - B_{k-1}\|_F = \|uv^T\|_F$$

Derivation of iteration

- Require rank 1 update:

$$B_k = B_{k-1} + u v^T \quad \leftarrow \text{But what are } u, v?$$

- Multiply on right with δ_k :

$$\Delta_k = B_k \delta_k \rightarrow B_k \delta_k = B_{k-1} \delta_k + u (v^T \delta_k)$$

- Solve for u under assumption $v^T \delta_k \neq 0$

$$u = \left(\frac{\Delta_k - B_{k-1} \delta_k}{v^T \delta_k} \right)$$

Now we have u in terms of stuff we know.

Derivation continued...

- Insert into rank 1 requirement:

$$B_k = B_{k-1} + \left(\frac{\Delta_k - B_{k-1} \delta_k}{v^T \delta_k} \right) v^T$$

But what is v ?

- This expression is valid for all $v^T \delta_k \neq 0$

Therefore, choose $v = \delta_k$ to get

$$B_k = B_{k-1} + (\Delta_k - B_{k-1} \delta_k) \left(\frac{\delta_k^T}{\delta_k^T \delta_k} \right)$$

Many choices for v , so we just choose one which makes our lives easy.

Update rule for B

$$B_k = B_{k-1} + \left(\frac{\Delta_k - B_{k-1} \delta_k}{\delta_k^T \delta_k} \right) \delta_k^T$$

- Update is rank 1 matrix
- Update satisfies

$$\Delta_{k-1} = B_{k-1} \delta_{k-1} \quad \Rightarrow \quad \Delta_k = B_k \delta_k$$

- Minimizes change on each step
- B converges as “residual” goes to zero.

$$r_k = \Delta_k - B_{k-1} \delta_k$$

Broyden's method algorithm

0. Compute initial values for iteration:

$$B_0, x_{k-1}, f(x_{k-1}), d_k, x_k$$

1. Compute $f(x_k)$, $\Delta_k = f(x_k) - f(x_{k-1})$ $\delta_k = x_k - x_{k-1}$

2. Update B :

Note subscripts....

$$B_k = B_{k-1} + \left(\frac{\Delta_k - B_{k-1} \delta_k}{\delta_k^T \delta_k} \right) \delta_k^T$$

3. Compute Newton step using B :

$$x_{k+1} = x_k - B_k \setminus f(x_k)$$

4. Check for convergence: $|x_{k+1} - x_k| < tol$

5. If not converged, loop back to 1.

```
function xret = broyden2D(F, J, x0, tol)
```

```
% Initialize computation
```

```
Bnm1 = eye(2); % J(x0); % This is the "Jacobian-like thing"
```

```
xnm1 = x0; % Starting point
```

```
Fnm1 = F(xnm1);
```

```
dn = -Bnm1\Fnm1;
```

```
xn = xnm1 + dn;
```

```
% Do root finding in a loop to prevent infinite loops from nonconvergence
```

```
for i = 1:25
```

```
Fn = F(xn);
```

```
Deln = Fn - Fnm1;
```

```
Bn = Bnm1 + (Deln - Bnm1*dn)*(dn'/(dn'*dn)); % Broyden update
```

```
% Now take step to new point
```

```
dnpl = -Bn\Fn;
```

```
xnpl = xn + dnpl;
```

```
% Check for convergence
```

```
if (norm(dnpl) < tol)
```

```
    xret = xnpl;
```

```
    return
```

```
end
```

```
% Move values back
```

```
xn = xnpl;
```

```
dn = dnpl;
```

```
Bnm1 = Bn;
```

```
Fnm1 = Fn;
```

```
end % end of for loop
```

```
fprintf('Terminated without convergence!\n')
```

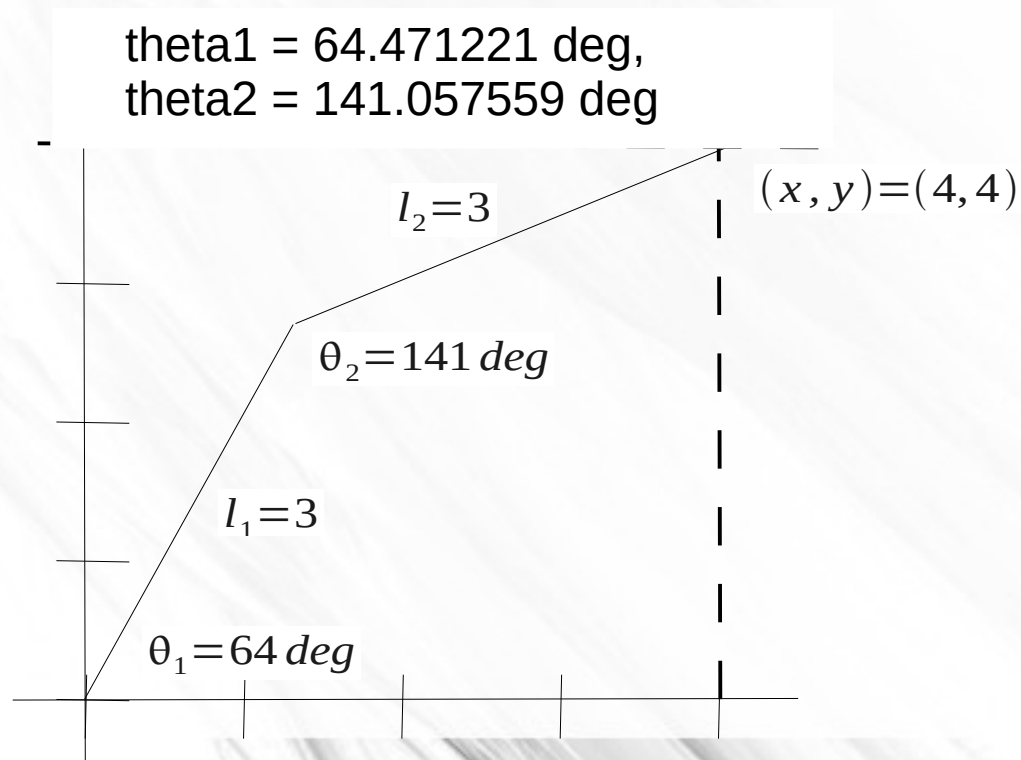
```
end
```

$$\leftarrow f(x_k), \quad \Delta_k = f(x_k) - f(x_{k-1})$$

$$B_k = B_{k-1} + \left(\frac{\Delta_k - B_{k-1} \delta_k}{\delta_k^T \delta_k} \right) \delta_k^T$$

$$x_{k+1} = x_k - B_k \setminus f(x_k)$$

```
>> test_broyden2D_robot
xn = [ 1.570796326795, 1.570796326795]
xn = [ 1.237462993462, 2.237462993462]
xn = [ 1.162527209471, 2.387334561443]
xn = [ 1.130040325334, 2.452308329716]
xn = [ 1.125473100396, 2.461442779592]
xn = [ 1.125236677998, 2.461915624389]
xn = [ 1.125235073392, 2.461918833601]
theta1 = 1.125235 rad, theta2 = 2.461919 rad
theta1 = 64.471221 deg, theta2 = 141.057559 deg
```



Remarks on Broyden

- You can think of Broyden as an ND analog of the secant method.
- You need a starting Jacobian.
 - You can often start with an approximate Jacobian, e.g. diagonal matrix, or finite-difference Jacobian.
 - In fact, you can sometimes start with an identity matrix. (Example under BroydensMethod).

Items discussed in this session

- Using Newton's method to find intersection point of two hyperbolas (LORAN).
- Convergence domain for Newton iteration.
- Homotopy method (Newton's method)
- Broyden's method (no Jacobian needed)