Unit II

Root-Finding and Nonlinear Systems

Root-finding and nonlinear systems

- strategy and .... tactics
- fixed-point iteration
- bracketing and bisection
- interpolation methods
- Newton's method
- special tactics for polynomials
- nonlinear systems
  - fixed point iteration
  - Newton-Rhapson method

Numerical equation solving

- an equation that needs a solution can be written in the form f(x) = 0
  - so 'finding roots' covers ALL types of equations
- there are <u>single-variable</u> problems ...
  - a one-dimensional problem or one independent variable
  - there are many, [often] straightforward, methods available but....
  - some pitfalls still need careful avoidance tactics
- and there are <u>multi-variable</u> problems ...
  - these are a very different battlefield
  - they are MUCH more difficult and DEMAND insight

Single-variable (nonlinear) equations

- location(s) of root(s) can be determined/estimated
- roots can be trapped and hunted down
- all nonlinear methods are based on iteration
  - one good reason for studying Jacobi and Gauss-Seidel
- proceed from an approximate trial solution until some convergence criterion is met
- convergence speed and success can be quantified to some extent
- for smooth functions a good algorithm will always succeed given a good enough initial guess

Hamming* said....

'The purpose of computing is insight not numbers'

* famous 20th C numerical analyst

Insight

- with a nonlinear problem insight can be critical simply to avoid total failure
  - black box + nonlinear problems = a bad combination
- algorithms may fail because ...
  - they find a highly accurate, but totally incorrect, root
  - there is no root to find but they find one anyway
  - they fail to find a root because the initial guess was too far away

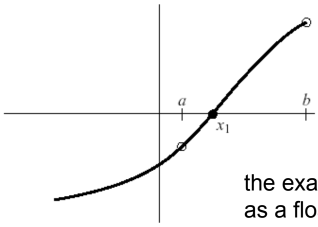
Three pieces of strategic advice

1. Examine the function graphically.
   - to locate roughly where the roots are and how many there may be
   - curve sketching is one way or...
   - best: use a Matlab plot to get the lay of the land
2. ALWAYS ALWAYS ALWAYS bracket a root.
   - find a range of x-values over which the function changes sign
3. Keep your iterations on a leash inside the bracketing interval.
   - unless you like to live dangerously

Bracketing a root

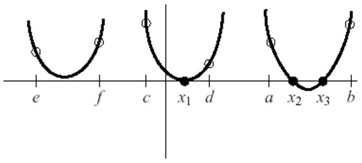
- a root is *bracketed* in the interval (a,b) if f(a) and f(b) have different signs
- a continuous function
  - is guaranteed to have at least one root in the interval (a,b)
  - remember the intermediate value theorem?
- a discontinuous function
  - may have a step inside the interval (a,b)
- numerically these statements are not so clear ....
  - a continuous function may have a floating point 'step' where the root is supposed to be
  - the zero can occur between two floating point numbers which are adjacent to machine precision (in the hole)

Bracketing a root



the exact root x_1 may not exist as a floating point number

Extremum complication

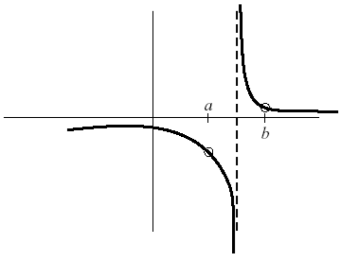


- 'walk downhill until you hit a sign change'
- doesn't work if there is ....
  - an extremum point (local max or min) or
  - a multiple root

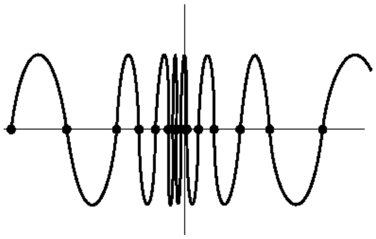
Singularities

- if f(x) → ∞ at x=r inside (a,b) most algorithms will converge to the singularity
- detecting this pathology is easy
  - check |f(r)| and observe that it is very large instead of close to zero as it should be near a root
- it's difficult to bracket roots for blackbox functions because you need a feeling for shape and characteristics

Bracketing a singularity



A pathological function with many roots



Refining a root

- a root can be refined iteratively once it is trapped inside the bracketing interval
- some methods are slow but sure (safe investments)
  - you know you will always find the root
  - convergence efficiency may be very weak
- other methods are fast but risky (stock market)
  - you can rapidly disappear to infinity without any warning
  - counter measures can make them safer

Fixed-point iteration

- to solve f(x) = 0 re-express the equation in the form: x = g(x)
- use this to derive the iteration expression

  $x_{new} = g(x_{old})$
- why learn this method?
  - it is simple and easily applied in hand calculations
  - provides an important theoretical framework for analysing numerical root-finding techniques of all kinds
  - can be generalized to nonlinear systems for which there is a dearth of methods

Fixed-point iteration: example

Find the roots of $f(x) = x - x^{1/3} - 2$ using three different fixed-point iteration functions:

- $g_1(x) = x^{1/3}+2$
- $g_2(x) = (6+2x^{1/3})/(3-x^{-2/3})$
- $g_3(x) = (x-2)^3$

Fixed-point iteration: convergence

- in the example
  - $g_1(x)$ converges slowly [7 digits/9 iterations]
  - $g_2(x)$ converges quickly [11 digits/3 iterations]
  - $g_3(x)$ always diverges regardless of the initial root guess
- fixed-point iteration will converge to a root in the interval [a,b] if
  - [a,b] contains a root
  - $|g'(x)| < 1$ on [a,b]
- the iterations
  - oscillate around the root if $-1 < g'(x) < 0$
  - converge monotonically if $0 < g'(x) < 1$

Bisection

- one of the safest methods
  - always finds a root once it is bracketed inside (a,b)
  - if there is more than one root the method converges to one of them
  - converges to a singularity if there is one in (a,b)
- check the function at the midpoint m of (a,b)
- determine which half-interval has the sign change
  - either (a,m) or (m,b)
- repeat with the half-interval

Two practical issues with bisection

1. Evaluation of the midpoint
   - $m = (a + b)/2$ can lead to roundoff problems
   - $m = a + (b - a)/2$ is better
2. Checking for a sign change
   - the test ‘f(a)*f(b) < 0?’ is susceptible to underflow
   - better to use the exact test ‘sign(f(a)) ~= sign(f(b))’? based on the floating point sign bit
   - Matlab has a built-in *sign* function for this
   - [see *brackPlot* illustrative function m-file]

Bisection: Matlab*

- *demoBisect* illustrate bisection with $f(x) = x - x^{1/3} - 2$ [slide 16]
- *brackPlot* can be used to locate the initial bracketing intervals
- *bisect* is a general implementation of the bisection method to find multiple roots

** These m-files are NOT provided in raw Matlab.*

Convergence rate

- after n iterations the root lies in an interval size δ_n
  $\delta_n = \delta_{n-1}/2 = ... = \delta_0/2^n$
  where $\delta_0 = b - a$ is the initial bracketing interval size
- to achieve a tolerance of δ therefore requires n iterations where
  $n = \log_2(\delta_n/ \delta_0)$
- for n=50 we have $\delta_n/ \delta_0 \sim$ *eps* so maximum number of iterations ever required with bisection is about 50

Convergence rates for iterative processes

- suppose a process coverges so the successive levels of uncertainty are given by

$$\delta_{n+1} = k\, \delta_n^{\,m}$$

  (with the k < 1)
- the method is said to *converge*
  - *linearly* if m = 1 (like the bisection method)
  - *super-linearly* if m > 1
- ‘linear’ convergence is a misnomer ....
  - it is really geometric convergence
  - the ‘linear’ means that successive significant digits are won linearly

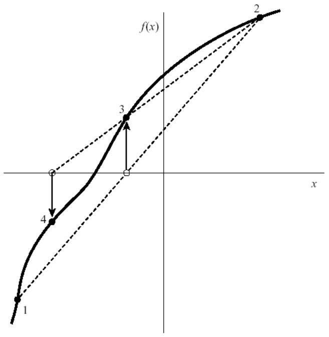
Convergence criteria

- in floating point arithmetic f(x) is unlikely to evaluate to zero even if there is an obvious root available (roundoff error)
- so we need a test to stop the iterative process
- can check tolerance on
  - x iterates and/or....
  - f(x) iterates
- some tolerance checks:
  - absolute test ... ok near 1 but stupid near 10^{40}
  - relative test ... not feasible near zero
  - hybrid test ... tol < $\varepsilon_m (|a| + |b|)/2$
  - backup test is also good (e.g. limit on max # iterations)

Interpolation methods

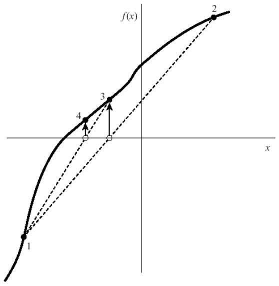
- based on local approximation of a smooth function near a root
  - linear (secant, *regula falsi*)
  - quadratic (Brent’s method)
- converge faster than bisection
- the next approximate root is found where the interpolating function intersects the x-axis
  - replaces one of the two endpoints of the iteration interval
  - which endpoint should we choose?

Secant method



- most recent prior estimate is retained
- the older estimate is discarded

Regula falsi method

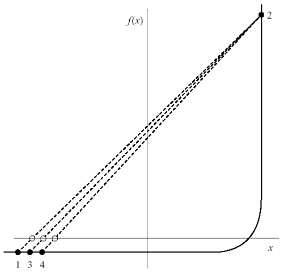


- next estimate is based on maintaining a sign-change bracketing in the interval

Secant method: issues

- converges faster than *regula falsi* but ....
- convergence is not guaranteed because sign-change bracketing is not maintained
- secant method is superlinear with convergence power the golden ratio (ϕ~1.618)
- problems with divergence can occur for insufficiently continuous functions due to local behaviour

Pathological example



Secant method: practicalities

- previous estimates for the root are x_{k-1}& x_k
- new estimate x_{k+1} is found using
  - simple linear equation formula and
  - find the x-intercept:

$$x_{k+1} = x_k - f(x_k)\left[\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right]$$

- this formula is good numerically because ...
  - it is $x_{k+1} = x_k + \Delta$
  - as $f(x_k)$-$f(x_{k-1})$ gets close to zero it accumulates roundoff
  - but Δ will still be small because $f(x_k)$ is close to zero
  - so the change in x_k close to the root is small as it should be

Secant method: practicalities

- a not-so-good algebraic re-arrangement is

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

  - same mathematically but not as good numerically
  - subject to catastrophic cancellation as successive $f(x_k)$ values get close
  - subject to underflow as $|f(x)| \to 0$

Root-finding by linear interpolation

- secant and *regula falsi* methods based on linear interpolation of f(x) within the current iteration interval (a,b) and sub-intervals
- these are *two-point methods* since the approximation is with respect to an interval
- typically faster convergence than bisection
- for pathological functions (e.g. not smooth, or smooth with rapidly changing second derivative) bisection may actually be faster
  - linear approximation methods may proceed slowly through many cycles to get close to the root

Root finding by quadratic interpolation

- a more rapid convergence than linear methods
- use quadratic interpolation in the iteration intervals
- interpolation requires three points (a,f(a)), (b,f(b)) and (c,f(c)) on the graph of the function
- the required quadratic should give x in terms of y, since an x-value (i.e. the root) is being estimated (i.e. when y = 0)
  - i.e. this is inverse quadratic interpolation
  - this topic comes in Unit III but is easy enough

Brent’s method

- the best all-round method [not in text]
- combines the speed of a superlinear quadratic interpolation with the safeness of bisection
- guaranteed to find a root, as long as the function can be evaluated in the initial bracketing interval
- book-keeping checks that the root estimate falls in the bracketing interval
  - if not the quadratic step is rejected
  - a bisection step is interspersed to bring the root back on side
  - a bisection step can also be introduced if the convergence is proceeding too slowly

Brent’s method

- for the three points given previously, the inverse quadratic interpolation is given by:

$$x = \frac{[y-f(a)][y-f(b)]c}{[f(c)-f(a)][f(c)-f(b)]} + \frac{[y-f(b)][y-f(c)]a}{[f(a)-f(b)][f(a)-f(c)]} + \frac{[y-f(c)][y-f(a)]b}{[f(b)-f(c)][f(b)-f(a)]}$$

- easy to see how this function is constructed to satisfy the three given points

Brent’s method: practicalities

- put y=0 and solve for x
- simple algebra (or substitute and check) gives the new estimate: **x = b + p/q**

$$\begin{aligned} r &= f(b)/f(c) \\ s &= f(b)/f(a) \\ t &= f(a)/f(c) \\ p &= s[t(r-t)(c-b)-(1-r)(b-a)] \\ q &= (t-1)(r-1)(s-1) \end{aligned}$$

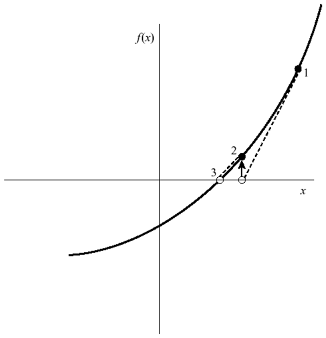
Brent’s method: practicalities

- in x = b + p/q the b term is the current best estimate for the root and p/q is supposed to be a small correction factor
- if f is not smooth then q may turn out to be very small, pulling x outside the bounds
- then you take a bisection step
- Matlab function for root-finding is based on Brent: *fzero(fun, x0, options, arg1, arg2, ...)*
  - fun = (string) name of the function to be evaluated
  - x0 = scalar starting point or vector root bracket
  - options = tolerances etc
  - arg1 etc = parameters to be passed to fun

Newton’s method

- previous methods require only the function’s values to be evaluated at points in the bracketing interval
- a faster convergence can be obtained if both the function f(x) and its derivative f′(x) can be evaluated for arbitrary x∈(a,b)
- for practical reasons this means the symbolic forms of f(x) and f′(x) should be available, not just values
- (geometrically) consists of extending the tangent line to f(x) at the current x_i to its x-intercept x_{i+1} which becomes the next root estimate .....

Newton’s method



Newton’s method: algebraic reasoning

- take the Taylor series expansion of f about x:

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \ldots .$$

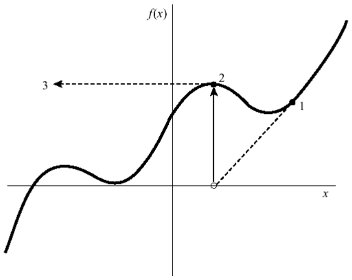
- drop the second-order and higher terms to get the linear approximation of f near x:

  $f(x+\delta) \approx f(x) + f'(x)\delta$
- for small enough δ and well-behaved f(x) approximate the required root putting f(x+δ) = 0
- get the famous *Newton formula*: **δ = - f(x)/f' (x)**
- this tells you that: **$x_{i+1} = x_i - f(x_i)/f'(x_i)$**

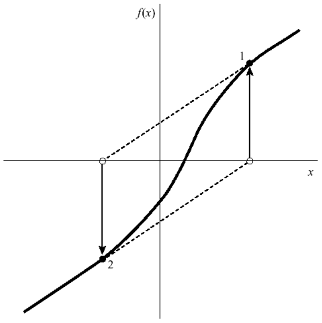
Newton risks

- a good initial guess is critical for success
- if too far from the true root the neglected higher-order terms in the Taylor expansion ARE important
- root estimate may lead far from the true root
  - very inaccurate and meaningless corrections δ are calculated
- problems are compounded for functions that are not smooth near the root
- a local method based on a single point with no intrinsic root bracketing (risky stuff) .....
  - bracketing bounds can be introduced to avoid shooting off to infinity

Newton pathology: local extremum



Newton pathology: non-convergent cycle



Newton convergence

- Taylor expansion of f(x) about the exact root α gives

$$0 = f(\alpha) = f(x_i) + (\alpha - x_i)f'(x_i) + \frac{(\alpha - x_i)^2}{2} f''(x_i) + \cdots$$

- the (i+1)th error term is

$$\begin{aligned} \varepsilon_{i+1} &= x_{i+1} - \alpha \\ &= (x_i - \alpha) - \frac{f(x_i)}{f'(x_i)} \qquad \text{using the Newton formula} \\ &= (x_i - \alpha) + \frac{(\alpha - x_i)f'(x_i) + \frac{(\alpha - x_i)^2}{2} f''(x_i) + \cdots}{f'(x_i)} \\ &\approx \frac{(\alpha - x_i)^2}{2} \frac{f''(x_i)}{f'(x_i)} \end{aligned}$$

Newton convergence

$$\varepsilon_{i+1} \approx \frac{f''(\alpha)}{2f'(\alpha)} \varepsilon_i^2$$

- so Newton is superlinear since $\varepsilon_{i+1} \approx \varepsilon_i^2$
- faster convergence rate than previous methods
- quadratic convergence means significant figures are approx. **DOUBLED** with each iteration .....

  ***provided* you are near a root [the payback]**
- terminate iterations when
  - the increment $|f(x_i) / f'(x_i)| < tol$ ..... or
  - $|f(x_{i+1})| > |f(x_i)|$

Newton: unpredictable global convergence

- consider the set of starting values from which Newton converges to a root
- for example $z^3 - 1 = 0$ has one real root $z = 1$ and two complex roots $z = \exp(\pm 2\pi i/3)$
- basins of convergence (starting points which converge to one of these roots) occupy 1/3 of the complex plane but....
- the boundary is a fractal
- see http://www.math.hawaii.edu/lab/newton.html

Toward polynomial roots: finding square roots

- use Newton to solve $f(x) = x^2 - a = 0$:

  $x_{i+1} = x_i - (x_i^2 - a) / 2x_i$

  $= (x_i + a/x_i) / 2$
  - Matlab exercise: how many iterations to get √17 to four decimals?
- solve $f(x) = x^3 - a = 0$ to get cube roots:

  $x_{i+1} = (2x_i - a/x_i^2)/3$
- these party tricks lead to general polynomial root finding but this is a minefield and needs very careful consideration .....

Roots of polynomials: special problems

- polynomials can be surprisingly very ill-conditioned
  - sensitive to perturbations in the coefficients (wild root behaviour)
- an nth degree polynomial should have n roots but some may be ...
  - repeated
  - complex (conjugate pairs if real coefficients)
  - so closely spaced as the cause numerical problems distinguishing them or converging separately

The problem of multiple roots

- consider $p(x) = (x - a)^2 = 0$
  - repeated root x = a
  - cannot bracket the root(s) with a sign change
  - slope-following methods (e.g. Newton) may fail (or at least be inefficient and inaccurate) due to roundoff error, because both $p(x)$ & $p'(x) = 0$ at the root
- can adopt a suitable method if the pathology is known in advance, but
  - we can't always know about it, or where it is, and ...
  - the 'repeatedness' may depend on numerical precision

Polynomial deflation

- as each root r is found (or estimated) p(x) is factored into p(x) = (x-r)q(x), where q(x) is degree one less then p(x)
  - root-finding effort is reduced as degree of q(x) is gradually reduced
  - avoid possibility of converging to the same (single) root as already found
- coefficients of successive polynomials q(x) become increasingly less accurate, since each root is approximate so ....
- the successive roots become increasingly less accurate as well

Stability and polynomial deflation

- stability
  - YES if inaccuracies are related simply to multiples of ε_m
  - NO if successive significant figures are eroded and answers become meaningless
- *forward deflation* divides out factors by finding the highest power of x each time
  - stable if the factor being divided corresponds to the root with smallest absolute value
- roots found should be considered tentative and *polished* using the original non-deflated polynomial
- two deflated roots may be inaccurate enough to polish to the same non-deflated root (spurious root-multiplicity)
  - need to back up and re-deflate using just the offending root

Two political camps

1. Go after the easy catch(es)
   - find the real, distinct roots using one of the root-finding methods previously discussed
   - proceed by deflation with linear and/or quadratic factors (if complex roots)
2. Use a safe method that always finds every root
   - find the real, complex, single and/or repeated roots
   - then polish all the roots
   - companion matrix is one way
   - Laguerre's method is another ... very clever way

Safe method #1: companion matrix

- the characteristic polynomial of A is
  p(x) = det(A - xI) = 0
- turn this around .... define the m×m *companion matrix* for p(x) = $a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0$ by:

$$\mathbf{A} = \begin{pmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & \cdots & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$$

- the roots of p(x) are the eigenvalues of A
- use a fancy (not-root-finding-based) eigenvalue method to get the roots of p(x) .... voila

Safe method #2: Laguerre's method

- uses complex arithmetic, even to find real roots
- convergence to a root is guaranteed from any starting point if coefficients are real
- some relations used in the algebra ....

$$P_n(x) = (x - x_1)(x - x_2)\cdots(x - x_n)$$

$$\begin{aligned} P_n'(x) &= (x - x_2)(x - x_3)\cdots(x - x_n) \\ &\quad + (x - x_1)(x - x_3)\cdots(x - x_n) \\ &\quad + \cdots + (x - x_1)(x - x_2)\cdots(x - x_{n-1}) \\ &= P_n(x)\left(\frac{1}{x - x_1} + \frac{1}{x - x_2} + \cdots + \frac{1}{x - x_n}\right) \end{aligned}$$

Laguerre's method

$$\frac{d}{dx}\ln|P_n(x)| = \frac{1}{x - x_1} + \frac{1}{x - x_2} + \cdots + \frac{1}{x - x_n} = \frac{P_n'(x)}{P_n(x)} \equiv G(x)$$

[eqn 1]

$$-\frac{d^2}{dx^2}\ln|P_n(x)| = \frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \cdots + \frac{1}{(x - x_n)^n}$$

$$= \left[\frac{P_n'(x)}{P_n(x)}\right]^2 - \frac{P_n''(x)}{P_n(x)} \equiv H(x)$$

[eqn 2]

Laguerre’s method

- now comes a weird and wonderful step ...
- to find the root x_1 make some oddball assumptions
  - x_1 is distance a from the current guess x, i.e. $x-x_1 = a$
  - ALL the other roots x_i are distance b from x_1, i.e. $x-x_i = b$
- now re-write equations [1,2] in terms of these assumed values:

$$\frac{1}{a} + \frac{n-1}{b} = G$$
$$\frac{1}{a^2} + \frac{n-1}{b^2} = H$$

Laguerre’s method

- next eliminate b and solve for a to give

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}}$$

(where the sign is +ve if G>0, -ve if G<0)

- finally proceed with iterations to find x_1
  - starting from x → G, H, a
  - then use x-a as next (improved) attempt at the root
  - repeat and get new a value
  - a gets small with **CUBIC** convergence rate
  - the tentative root choice is $x = x_1$
  - these can be polished afterwards

Matlab and polynomials

- a polynomial $p(x) = c_1x^n + c_2x^{n-1} + ... + c_nx + c_{n+1}$ is represented in Matlab by a vector of coefficients:
  c = [c_1 c_2 ... c_{n+1}]
- to evaluate p(x) use *px = polyval(c,x)*
  - x and px can also be vectors of points
- Matlab can ...
  - make a polynomial with given roots v: *c = poly(v)*
  - find the roots of a polynomial: *v = roots(c)*
  - differentiate a polynomial: *d = polyder(c)*
  - do synthetic division of polys: *[q,r] = deconv(c,d)*

Laguerre’s method: example

Find the roots of the polynomial $p(x) = x^5 + 3x^4 - 8x^3 - 12x^2 + 16x$.

The bad and the ugly

- to show how bad polynomials can be numerically .... check out Wilkinson’s perfidious polynomial:
  roots(poly(1:20)) ... roots(poly(1:21) ... etc.

Newton with numerical derivative?

- how about approximating f′(x) local derivative numerically by
  f′((x) = [f(x+dx) - f(x)]/dx?
- not recommended for single variable problems
  - requires one extra function evaluation per step, so convergence rate is reduced to √2
  - if dx is too small roundoff kills you
  - if dx is too large convergence goes linear and you might as well use the initial derivative $f'(x_0)$ for every iteration (or the secant method)
- for multi-dimensional problems it’s a different story ...

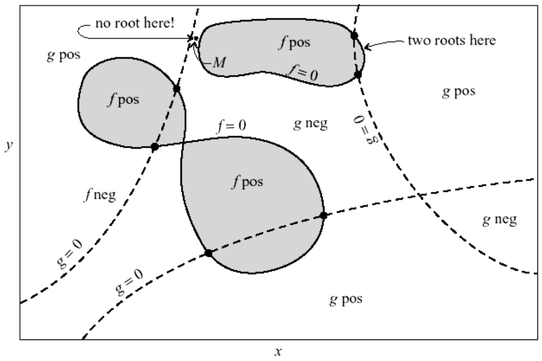
Systems of nonlinear equations

- there are no good general methods for solving nonlinear systems with more than one equation
- consider the simplest 2-dimensional problem:

$$f(x,y) = 0 \qquad g(x,y) = 0$$

- f & g are
  - arbitrary functions with...
  - no connection in general
- the equations establish zero contours that divide the xy plane into regions where
  - $f(x,y)>0$ or $f(x,y)<0$ and
  - $g(x,y)>0$ or $g(x,y)<0$

Systems of nonlinear equations



Systems of nonlinear equations

- solutions to be found are points common to the zero contours of both f and g
- these common points have no special significance to either f or g
- to find a solution requires mapping out the contours for each function, then finding their intersection(s)
- in general these contours consist of an unknown number of disjoint curves in the xy plane

- hmm....not an easy problem

Simple geometric example

$$x^2 + y^2 = 4$$
$$e^x + y = 1$$

Graph these first and observe two solutions: the intersection of the circle and the exponential curve at about (-1.8, 0.8) and (1, -1.7)

n-variable system

- in n-dimensions you need to find points common to n zero-contour hyper-surfaces of dimension n-1
- insight is critical for any hope of success
  - use the characteristics and special properties of the functions
- solution methods must be problem-specific
  - is any solution at all expected?
  - is a unique solution expected?
  - where are solution(s) expected?
- a simple method which sometimes works for not too nonlinear systems is *fixed-point iteration*

Fixed point iteration: example

Examine the capabilities of fixed point iteration for solving the system on slide 64.

Newton-Rhapson method

- the scalar Newton method can be generalized to multi-dimensions
- for one nonlinear equation f(x)=0 we had

  $x_{i+1} = x_i - f(x_i) / f'(x_i)$
- this can be written as a linear equation giving the correction δ (in terms of the function and its derivative at the current point)

  $f'(x_i)\ \delta = - f(x_i)$
- this δ correction moves the function closer to zero
- how can we generalize this to n-dimensions?

Newton-Rhapson method

- consider the problem of zero-ing n functions F_i each a function of n unknowns x_i

  $F_i(x_1,\ldots,x_n) = 0 \qquad i= 1,\ldots n$
- in vector notation we can write F(x) = 0, where F = $(F_1, F_2, \ldots F_n)$ is the vector of functions
- each function F_i can be expanded in the neighbourhood of the point x as a multi-dimensional Taylor series:

$$F_i(\mathbf{x}+\delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2)$$

- the matrix of partial derivatives is called the *Jacobian matrix* J:

$$J_{ij} \equiv \frac{\partial F_i}{\partial x_j}$$

Newton-Rhapson method

- in matrix notation the Taylor exansion is written

$$\mathbf{F}(\mathbf{x}+\delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J}\cdot\delta\mathbf{x} + O(\delta\mathbf{x}^2)$$

- neglect terms higher than first order and set $F(x+\delta x)=0$
- this gives an equation for the correction vector δx which moves all the F_i functions simultaneously closer to zero:

$$\boxed{\mathbf{J}\cdot\delta\mathbf{x} = -\mathbf{F}}$$

Practical considerations

- the linear system given in the N-R equation can be solved by LU decomposition (or other method)
- iteration proceeds by putting

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x}$$

and checking convergence condition(s)

- check both the functions and the roots (||F(x)|| < ftol and $||\delta||$ < xtol)
- once either reaches machine precision nothing further will change
- examine behaviour frequently to ensure the process is converging on a root, and onto the desired root
- J can be supplied symbolically or by finite differences if necessary

Newton-Rhapson: simple example

$f(x,y) = 4 - x^2 - y^2 = 0$

$g(x,y) = 1 - e^x - y = 0$

- partial derivatives are $f_x = -2x$, $f_y = -2y$, $g_x = -e^x$ and $g_y = -1$
- Jacobian matrix is

$$J = \begin{bmatrix} f_x & f_y \\ g_x & g_y \end{bmatrix} = \begin{bmatrix} -2x & -2y \\ -e^x & -1 \end{bmatrix}$$

- beginning with $x_0 = (1,-1.7)$ we have to solve the linear system:

$$J(1,-1.7)\begin{bmatrix} \delta x \\ \delta y \end{bmatrix} = \begin{bmatrix} f(1,-1.7) \\ g(1,-1.7) \end{bmatrix}$$

Newton-Rhapson: simple example (cont.)

$$\begin{bmatrix} -2 & 3.4 \\ 2.7183 & -1.0 \end{bmatrix}\begin{bmatrix} \delta x \\ \delta y \end{bmatrix} = \begin{bmatrix} 0.1100 \\ -0.0183 \end{bmatrix}$$

- solution is $(\delta x, \delta y) = (0.0043,-0.0298)$.
- this gives $x_1 = (1.0043,-1.7298)$
- now repeat to get $x_2 = (1.004169, -1.729637)$ which satisfies very nicely $f(x_2)$ = 1e-07, $g(x_2)$ = 1e-08

Newton-Rhapson: practical considerations

- N-R reduces a n-dimensional nonlinear problem to a linear system in n unknown corrections (the δ vector)
- converges quadratically (like Newton)
  - but only if the starting point is near a root
- expensive in function evaluations
  - e.g. for 2x2 example there are six evals. per step
  - nxn requires n^2+n evals. per step
- N-R not easy to implement if n is large
- can try eliminating variables to reduce the size
  - e.g. in the previous example solve for $y = 1 - e^x$ and sub. in eqn 1 to get $4 - x^2 - (1-e^x)^2 = 0$, or $3 - x^2 + 2e^x - e^{2x} = 0$, an equation which can be solved as a nonlinear equation in one variable (previous methods)

Jacobian estimation

- for larger systems can simplify the calcs. by estimating the Jacobian at successive steps in terms of an earlier Jacobian
  - e.g. for n equations re-compute J every n steps
- example $f(x,y) = e^x - y = 0$ and $g(x,y) = xy - e^x = 0$
  - start with $x_0 = (0.95, 2.7)$
  - in step 2 keep J fixed at the J of step 1
  - converges to six decimal precision of exact solution (1,e) after three iterations
- or can use an approximate J which satisfies

$$\mathbf{B}_{i+1} \cdot \delta\mathbf{x}_i = \delta\mathbf{F}_i$$

- this is a multi-dimensional generalization of the secant method, which estimates df/dx (Broyden)

Newton-Rhapson & minimization

- multi-dimensional minimizing is equivalent to finding a zero of a gradient function
- so why is multi-dimensional minimization relatively simple compared to root-finding?
- the components of the grad are related and satisfy strong conditions
- minimizing is equivalent to sliding down a one-dimensional surface
- root-finding is equivalent to simultaneously minimizing n independent functions, i.e. sliding down n surfaces simultaneously
  - tradeoffs are needed
  - how is progress in one dimension traded against progress in another?

Nonlinear systems: conclusions

- apart from the simplest of problems solving nonlinear systems is a very difficult task
- all methods are iterative
- there are very few basic methods available
- more advanced methods impinge on the study of nonlinear optimization
- Matlab symbolic toolbox can evaluate:
  - jacobian(w,v) ... the Jacobian of symbolic column vector w w.r.t. symbolic row vector v
  - diff(S,'x') .... the derivative of a symbolic expression S w.r.t x