INTRODUCTION TO NUMERICAL RELATIVITY

LECTURE 5
TOOLS OF THE TRADE

2006 Spring School on Numerical Methods in Gravitation and Astrophysics

March 17, 2006, KIAS, Seoul Korea

Matthew Wm Choptuik

CIAR Cosmology & Gravity Program
Dept of Physics & Astronomy, UBC
Vancouver, BC, Canada

(Happy St Patrick's Day!)

Lecture Plan

(see http://bh0.phas.ubc.ca/410 and http://bh0.phas.ubc.ca/381C for more details)

- Overview of essential knowledge for dealing with nonlinearity in numerical treatment of many PDEs of mathematical physics
  - Newton's method for one unknown
  - Newton's method for systems of unknowns
  - Continuation (homotopy)

- Overview of essential SOFTWARE environments, tools, programs, utilities, "codes", …
  - Complete solution of the KdV equation

- Glimpses of "custom" development/visualization/analysis tools used by "Choptuik et al" group (RNPL, xvs, DV, PAMR, …)

NEWTON'S METHOD & CONTINUATION (HOMOTOPY)

- We will consider two cases

1. $f(x) = 0$ "1-dimensional"
2. $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ "d-dimensional"

$$\mathbf{x} \equiv [x_1, x_2, \ldots, x_d]$$

$$\mathbf{f} \equiv [f_1(x_1, x_2, \ldots, x_d), \ldots, f_d(x_1, x_2, \ldots, x_d)]$$

1. Solving Nonlinear Equations in One Variable

- We have briefly discussed bisection (binary search), will consider one other technique: *Newton's method* (Newton-Raphson method).

Preliminaries

- We want to find one or more roots of

$$f(x) = 0 \tag{1}$$

We first note that *any* nonlinear equation in one unknown can be cast in this canonical form.

- *Definition:* Given a canonical equation, $f(x) = 0$, the *residual* of the equation for a given x-value is simply the function f evaluated at that value.

- *Iterative technique:* Assume $f(x) = 0$ has a root at $x = x^\star$; then consider sequence of estimates (iterates) of $x^\star$, $x^{(n)}$

$$x^{(0)} \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \cdots \rightarrow x^{(n)} \rightarrow x^{(n+1)} \rightarrow \cdots \rightarrow x^\star$$

- Associated with the $x^{(n)}$ are the corresponding residuals

$$\begin{array}{ccccccccccc} r^{(0)} & \rightarrow & r^{(1)} & \rightarrow \cdots \rightarrow & r^{(n)} & \rightarrow & r^{(n+1)} & \rightarrow \cdots \rightarrow & 0 \\ \| & & \| & & \| & & \| & & \| \\ f(x^{(0)}) & & f(x^{(1)}) & & f(x^{(n)}) & & f(x^{(n+1)}) & & f(x^\star) \end{array}$$

$$\boxed{\text{Locating a root} \equiv \text{Driving the residual to 0}}$$

- *Convergence:* When we use an iterative technique, we have to decide *when* to stop the iteration. For root finding case, it is natural to stop when

$$|\delta x^{(n)}| \equiv |x^{(n+1)} - x^{(n)}| \leq \epsilon \qquad (2)$$

where ϵ is a prescribed convergence criterion.

- A better idea is to use a "relativized" δx

$$\frac{|\delta x^{(n)}|}{|x^{(n+1)}|} \leq \epsilon \qquad (3)$$

but we should "switch over" to "absolute" form (2) if $|x^{(n+1)}|$ becomes "too small" (examples in on-line code).

- *Motivation:* Small numbers often arise from "unstable processes" (numerically sensitive), e.g. $f(x+h) - f(x)$ as $h \to 0$, or from "zero crossings" in periodic solutions etc.—in such cases may not be possible and/or sensible to achieve stringent relative convergence criterion

- Requires "good" initial guess, $x^{(0)}$; "good" depends on specific nonlinear equation being solved

- Refer to Numerical Recipes for more discussion; we will assume that we have a good $x^{(0)}$, and will discuss one general technique for determining good initial estimate later.

- First consider a rather circuitous way of solving the "trivial" equation

$$ax = b \quad \longrightarrow \quad f(x) = ax - b = 0 \tag{4}$$

Clearly, $f(x) = 0$ has the root

$$x^\star = \frac{b}{a} \tag{5}$$

- Consider, instead, starting with some initial guess, $x^{(0)}$, with residual

$$r^{(0)} \equiv f(x^{(0)}) \equiv a x^{(0)} - b \tag{6}$$

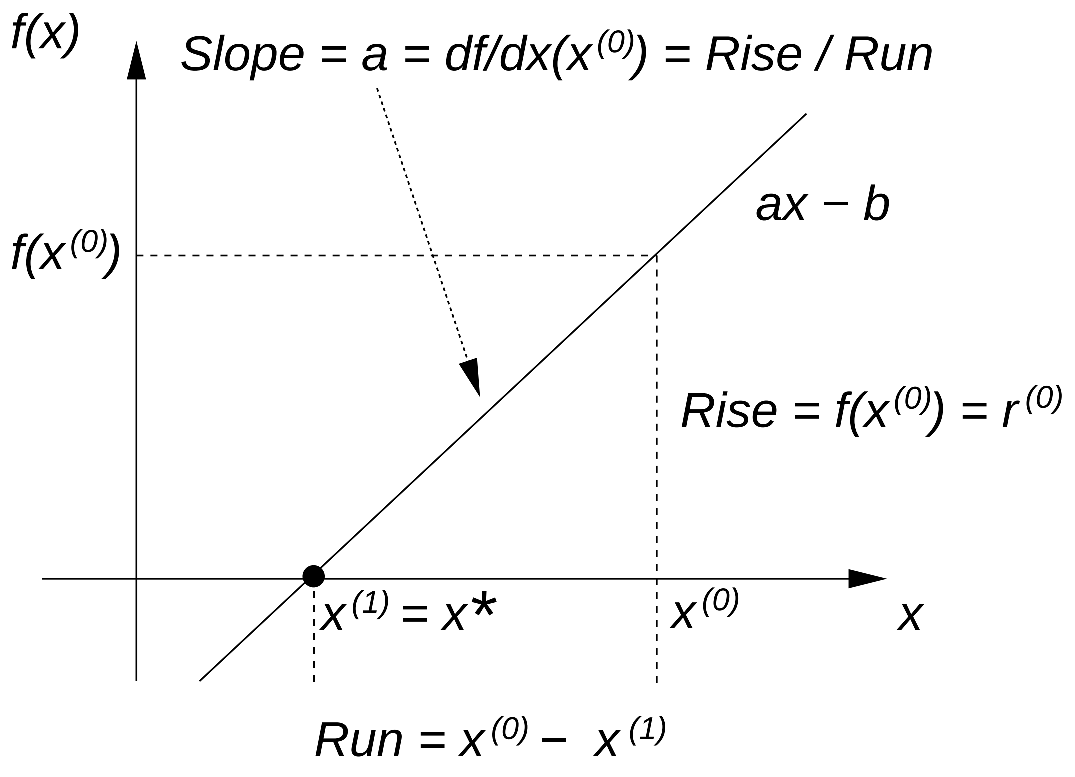
Then we can compute an improved estimate, $x^{(1)}$, which is actually the solution, $x^{\star}$, via

$$x^{(1)} = x^{(0)} - \delta x^{(0)} = x^{(0)} - \frac{r^{(0)}}{f'(x^{(0)})} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} \tag{7}$$

"Proof":

$$x^{(1)} = x^{(0)} - \frac{r^{(0)}}{a} = x^{(0)} - \frac{a x^{(0)} - b}{a} = \frac{b}{a} \tag{8}$$

- Graphically, we have



- *Summary*

$$x^{(1)} = x^{(0)} - \delta x^{(0)} \tag{9}$$

where $\delta x^{(0)}$ satisfies

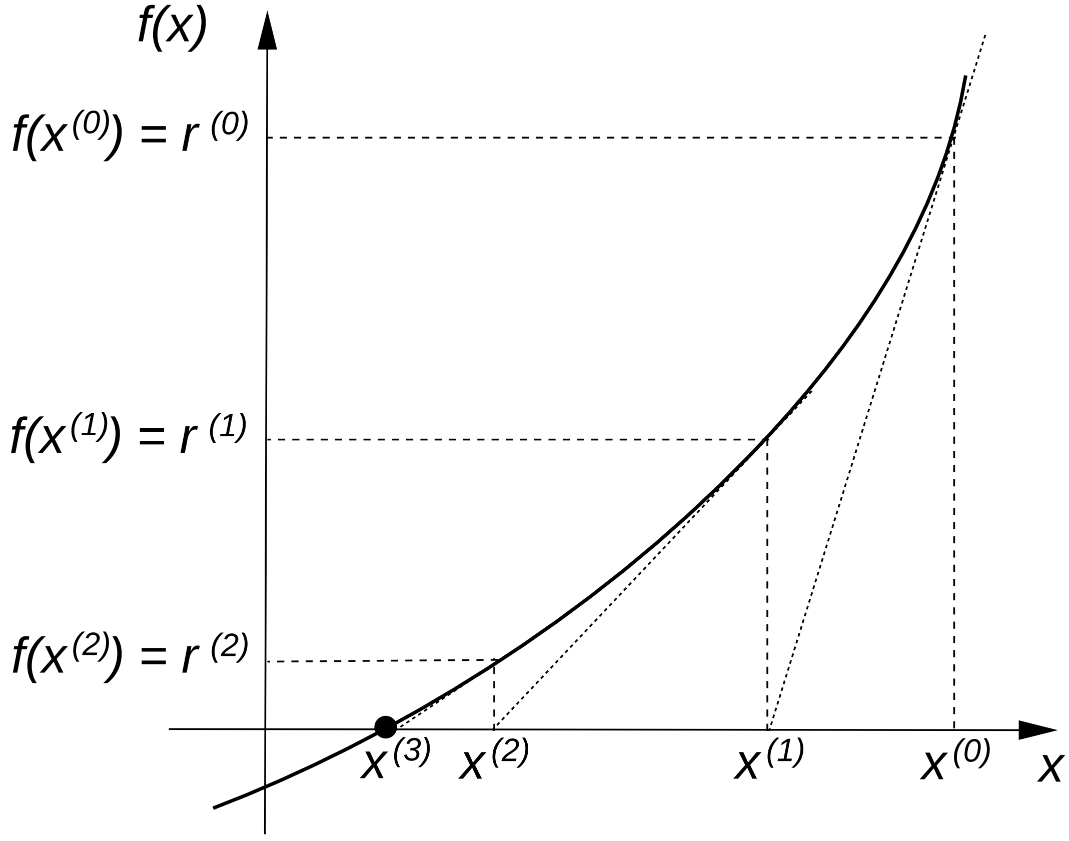
$$f'(x^{(0)})\delta x^{(0)} = f(x^{(0)}) \tag{10}$$

or

$$f'(x^{(0)})\delta x^{(0)} = r^{(0)} \tag{11}$$

- Equations (9-10) immediately generalize to non-linear $f(x)$ and, in fact, are precisely Newton's method.

- For a general nonlinear function, $f(x)$, we have, graphically



- *Newton's method for $f(x) = 0$:* Starting from some initial guess $x^{(0)}$, generate iterates $x^{(n+1)}$ via

$$x^{(n+1)} = x^{(n)} - \delta x^{(n)} \tag{12}$$

$$f'(x^{(n)})\delta x^{(n)} = r^{(n)} \equiv f(x^{(n)}) \tag{13}$$

  or more compactly

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})} \tag{14}$$

- *Convergence: When* Newton's method converges, it does so *rapidly*; expect number of significant digits (accurate digits) in $x^{(n)}$ to roughly *double* at each iteration (quadratic convergence)

- *Example: "Square Roots"*

$$f(x) = x^2 - a = 0 \quad \longrightarrow \quad x^\star = \sqrt{a} \tag{15}$$

Application of (14) yields

$$\begin{aligned} x^{(n+1)} &= x^{(n)} - \frac{{x^{(n)}}^2 - a}{2x^{(n)}} \\ &= \frac{2{x^{(n)}}^2 - \left({x^{(n)}}^2 - a\right)}{2x^{(n)}} \\ &= \frac{{x^{(n)}}^2 + a}{2x^{(n)}} \end{aligned}$$

which we can write as

$$x^{(n+1)} = \frac{1}{2}\left(x^{(n)} + \frac{a}{x^{(n)}}\right) \tag{16}$$

- Try it manually, compute $\sqrt{2} = 1.414\ \ 2135\ \ 6237$ using 12-digit arithmetic (hand calculator)

Iterate	Sig. Figs
$x^{(0)} = 1.\mathbf{5}$	1
$x^{(1)} = \frac{1}{2}(1.5 + 2.0/1.5) = 1.41\mathbf{6}\ \ 6666\ \ 6667$	3
$x^{(2)} = \frac{1}{2}(1.416\cdots + 2.0/1.416\cdots) = 1.414\ \ 21\mathbf{5}6\ \ 8628$	6
$x^{(3)} = \frac{1}{2}(1.4142\cdots + 2.0/1.4142\cdots) = 1.414\ \ 2135\ \ 623\mathbf{8}$	11

Alternate Derivation of Newton's Method (Taylor series)

- Again, let $x^\star$ be a root of $f(x) = 0$, then

$$0 = f(x^\star) = f(x^{(n)}) + (x^\star - x^{(n)})f'(x^{(n)}) + O((x^\star - x^{(n)})^2) \quad (17)$$

Neglecting the higher order terms, we have

$$0 \approx f(x^{(n)}) + (x^\star - x^{(n)})f'(x^{(n)}) \quad (18)$$

Now, treating the last expression as an equation, and replacing $x^{(n)}$ with the new iterate, $x^{(n+1)}$, we obtain

$$0 = f(x^{(n)}) + (x^{(n+1)} - x^{(n)})f'(x^{(n)}) \quad (19)$$

or

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})} \quad (20)$$

as previously.

2. Newton's Method for Systems of Equations

- We now want to solve

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{21}$$

where

$$\mathbf{x} = (x_1, x_2, \ldots, x_d) \tag{22}$$

$$\mathbf{f} = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_d(\mathbf{x})) \tag{23}$$

- *Example ($d = 2$):*

$$\begin{aligned} \sin(xy) &= \frac{1}{2} && (24) \\ y^2 &= 6x + 2 && (25) \end{aligned}$$

In terms of our canonical notation, we have

$$\begin{aligned} \mathbf{x} &\equiv (x, y) && (26) \\ \mathbf{f} &\equiv (f_1(\mathbf{x}), f_2(\mathbf{x})) && (27) \\ f_1(\mathbf{x}) &= f_1(x, y) = \sin(xy) - \frac{1}{2} && (28) \\ f_2(\mathbf{x}) &= f_2(x, y) = y^2 - 6x - 2 && (29) \end{aligned}$$

- The method is again iterative, we start with some initial guess, $\mathbf{x}^{(0)}$, then generate iterates

$$\mathbf{x}^{(0)} \to \mathbf{x}^{(1)} \to \mathbf{x}^{(2)} \to \cdots \to \mathbf{x}^{(n)} \to \mathbf{x}^{(n+1)} \to \cdots \to \mathbf{x}^\star$$

  where $\mathbf{x}^\star$ is *a* solution of (21)

- *Note:* The task of determining a good initial estimate $\mathbf{x}^{(0)}$ in the d-dimensional case is even more complicated than it is for the case of a single equation—again we will *assume* that $\mathbf{x}^{(0)}$ *is* a good initial guess, and that $\mathbf{f}(\mathbf{x})$ is sufficiently well-behaved that Newton's method will provide a solution (i.e. *will* converge).

- As we did with the scalar (1-d) case, with any estimate, $\mathbf{x}^{(n)}$, we associate the *residual vector*, $\mathbf{r}^{(n)}$, defined by

$$\mathbf{r}^{(n)} \equiv \mathbf{f}(\mathbf{x}^{(n)}) \tag{30}$$

- The analogue of $f'(x)$ in this case is the *Jacobian matrix*, $\mathbf{J}$, of first derivatives. Specifically, $\mathbf{J}$ has elements J_{ij} given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{31}$$

- For our current example we have

$$\begin{aligned} f_1(x,y) &= \sin(xy) - \frac{1}{2} \\ f_2(x,y) &= y^2 - 6x - 2 \end{aligned}$$

$$\mathbf{J} = \begin{bmatrix} \partial f_1/\partial x & \partial f_1/\partial y \\ \partial f_2/\partial x & \partial f_2/\partial y \end{bmatrix} = \begin{bmatrix} y\cos(xy) & x\cos(xy) \\ -6 & 2y \end{bmatrix}$$

- We can now derive the multi-dimensional Newton iteration, by considering a multivariate Taylor series expansion, paralleling what we did in the 1-d case:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}^\star) = \mathbf{f}(\mathbf{x}^{(n)}) + \mathbf{J}[\mathbf{x}^{(n)}] \cdot (\mathbf{x}^\star - \mathbf{x}^{(n)}) + O((\mathbf{x}^\star - \mathbf{x}^{(n)})^2) \quad (32)$$

  where the notation $\mathbf{J}[\mathbf{x}^{(n)}]$ means we evaluate the Jacobian matrix at $\mathbf{x} = \mathbf{x}^{(n)}$.

  Dropping higher order terms, and replacing $\mathbf{x}^\star$ with $\mathbf{x}^{(n+1)}$, we have

$$\mathbf{0} = \mathbf{f}(\mathbf{x}^{(n)}) + \mathbf{J}[\mathbf{x}^{(n)}](\mathbf{x}^{(n+1)} - \mathbf{x}^{(n)}) \quad (33)$$

Defining $\delta\mathbf{x}^{(n)}$ via

$$\delta\mathbf{x}^{(n)} \equiv -(\mathbf{x}^{(n+1)} - \mathbf{x}^{(n)}) \tag{34}$$

the d-dimensional Newton iteration is given by

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \delta\mathbf{x}^{(n)} \tag{35}$$

where the update vector, $\delta\mathbf{x}^{(n)}$, satisfies the $d \times d$ *linear system*

$$\mathbf{J}[\mathbf{x}^{(n)}]\,\delta\mathbf{x}^{(n)} = \mathbf{f}(\mathbf{x}^{(n)}) \tag{36}$$

- Again note that the Jacobian matrix, $\mathbf{J}[\mathbf{x}^{(n)}]$, has elements

$$J_{ij}[\mathbf{x}^{(n)}] = \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(n)}} \tag{37}$$

- At each step of the Newton iteration, the linear system (36) can, of course, be solved using an appropriate linear solver (e.g. general, tridiagonal, or banded).

General Structure of a Multidimensional Newton Solver

$\mathbf{x}$:	Solution vector
$\mathbf{res}$:	Residual vector
$\mathbf{J}$:	Jacobian matrix
$\mathbf{dx}$:	Update vector

```
$\mathbf{x} = \mathbf{x}^{(0)}$
do while $\|\mathbf{dx}\|_2 > \epsilon$
    do i = 1 , neq
        res(i) $= f_i(\mathbf{x})$
        do j = 1 , neq
            J(i,j) $= [\partial f_i / \partial x_j](\mathbf{x})$
        end do
    end do
    $\mathbf{dx} =$ solve$(\mathbf{J}\ \mathbf{dx} = \mathbf{res})$
    $\mathbf{x} = \mathbf{x}$ - $\mathbf{dx}$
end do
```

Finite Difference Example: Non-Linear BVP

- Consider the nonlinear two-point boundary value problem

$$u(x)_{xx} + (uu_x)^2 + \sin(u) = F(x) \tag{38}$$

which is to be solved on the interval

$$0 \le x \le 1 \tag{39}$$

with the boundary conditions

$$u(0) = u(1) = 0 \tag{40}$$

- As we did for the case of the linear BVP, we will approximately solve this equation using $O(h^2)$ finite difference techniques. As usual we introduce a uniform finite difference mesh:

$$x_j \equiv (j-1)h \qquad j = 1, 2, \cdots N \qquad h \equiv (N-1)^{-1} \tag{41}$$

- Then, using the standard $O(h^2)$ approximations to the first and second derivatives

$$u_x(x_j) = \frac{u_{j+1} - u_{j-1}}{2h} + O(h^2) \tag{42}$$
$$u_{xx}(x_j) = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + O(h^2) \tag{43}$$

the discretized version of (38-40) is

$$\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + (u_j)^2 \left[\frac{u_{j+1} - u_{j-1}}{2h}\right]^2 + \sin(u_j) - F_j = 0 ; \quad j = 2 \ldots N-1 \tag{44}$$
$$u_1 = 0 \tag{45}$$
$$u_N = 0 \tag{46}$$

Note that we have cast the discrete equations in the canonical form $\mathbf{f}(\mathbf{u}) = \mathbf{0}$

- In order to apply Newton's method to the algebraic equations (45-46), we must compute the Jacobian matrix elements of the system.

- We first observe that due to the "nearest-neighbor" couplings of the unknowns u_j via the approximations (42-43), the Jacobian matrix is *tridiagonal* in this case.

- For the *interior* grid points, $j = 2 \ldots N$, corresponding to rows $2 \ldots N$ of the matrix, we have the following non-zero Jacobian elements:

$$\begin{aligned} J_{j\,,\,j} &= -\frac{2}{h^2} + 2u_j \left[\frac{u_{j+1} - u_{j-1}}{2h}\right]^2 + \cos(u_j) && (47) \\ J_{j\,,\,j-1} &= \frac{1}{h^2} - (u_j)^2 \, \frac{u_{j+1} - u_{j-1}}{2h^2} && (48) \\ J_{j\,,\,j+1} &= \frac{1}{h^2} + (u_j)^2 \, \frac{u_{j+1} - u_{j-1}}{2h^2} && (49) \end{aligned}$$

- For the *boundary* points, $j = 1$ and $j = N$, corresponding to the first and last row, respectively, of $\mathbf{J}$, we have

$$\begin{aligned} J_{1,1} &= 1 && (50) \\ J_{1,2} &= 0 && (51) \end{aligned}$$

and

$$\begin{aligned} J_{N,N} &= 1 && (52) \\ J_{N,N-1} &= 0 && (53) \end{aligned}$$

- Note that these last expressions correspond to the "trivial" equations

$$f_1 = u_1 = 0 \tag{54}$$

$$f_N = u_N = 0 \tag{55}$$

which have associated residuals

$$r_1^{(n)} = u_1^{(n)} \tag{56}$$

$$r_N^{(n)} = u_N^{(n)} \tag{57}$$

- Observe that if we initialize $u_1^{(0)} = 0$ and $u_N^{(0)} = 0$, then we will *automatically* have $\delta u_1^{(n)} = \delta u_N^{(n)} = 0$, which will yield $u_1^{(n)} = 0$ and $u_N^{(n)} = 0$ as desired.

- This is an example of the general procedure we have seen previously for imposing Dirichlet conditions; namely the conditions are implemented as "trivial" (linear) equations (but it is, of course, absolutely crucial to implement them *properly* in this fashion!)

- *Testing procedure:* We adopt the same technique used for the linear BVP case—we *specify* $u(x)$, then compute the function $F(x)$ that is required to satisfy (38); $F(x)$ is then supplied as input to the code, and we ensure that as $h \to 0$ we observe second order convergence of the computed finite difference solution $\hat{u}(x)$ to the continuum solution $u(x)$.

- *Example:* Taking

$$u(x) = \sin(4\pi x) \equiv \sin(\omega x) \tag{58}$$

then

$$\begin{aligned} F(x) &= u_{xx} + (u u_x)^2 + \sin(u) \qquad (59) \\ &= -\omega^2 \sin(\omega x) + \omega^2 \sin^2(\omega x)\cos^2(\omega x) + \sin(\sin(\omega x)) \end{aligned}$$

- We note that due to the nonlinearity of the system, we will actually find *multiple* solutions, depending on how we initialize the Newton iteration; this is illustrated with the on-line code `nlbvp1d`.

3. Determining Good Initial Guesses: Continuation

- It is often the case that we will want to solve nonlinear equations of the form

$$\mathbf{N}(\mathbf{x};\bar{\mathbf{p}}) = 0 \tag{60}$$

where we have adopted the notation $\mathbf{N}(\cdots)$ to emphasize that we *are* dealing with a nonlinear system. Here $\mathbf{x} = (x_1, x_2 \ldots x_d)$ is, as previously, a vector of unknowns, with $\mathbf{x} = \mathbf{x}^\star$ *a* solution of (60).

- The quantity $\bar{\mathbf{p}}$ in (60) is another vector, of length m, which enumerates any additional parameters (generally adjustable) that enter into the problem; these could include: coupling constants, rate constants, "perturbation" amplitudes etc.

- The nonlinearity of any particular system of the form (60) may make it *very* difficult to compute $\mathbf{x}^\star$ without a good initial estimate $\mathbf{x}^{(0)}$; in such cases, the technique of *continuation* often provides the means to generate such an estimate.

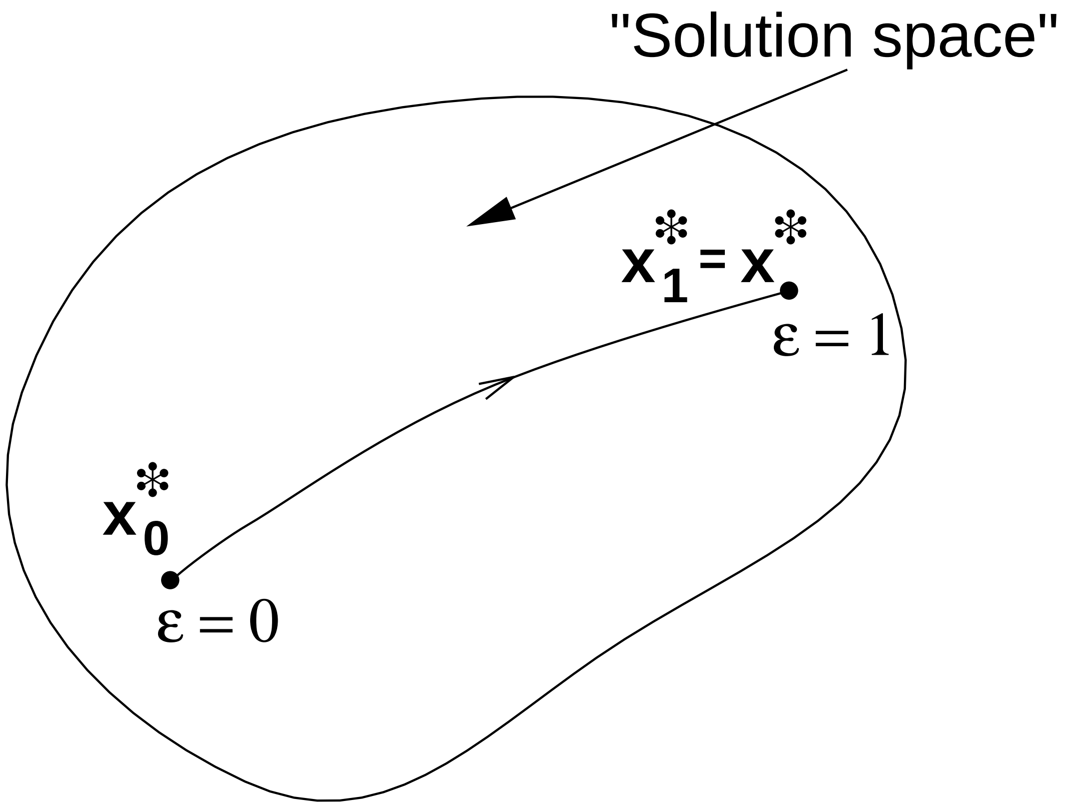
- *Continuation:* The basic idea underlying continuation is to "sneak up" on the solution by introducing an additional parameter, ϵ (the continuation parameter), so that by *continuously* varying ϵ from 0 to 1 (by convention), we vary *from*:

  1. A problem that we know how to solve, or for which we already have a solution.

  to

  2. The problem of interest.

- Schematically we can sketch the following picture:



- Note that we thus consider a *family* of problems

$$\mathbf{N}_\epsilon(\mathbf{x}; \bar{\mathbf{p}}) = 0 \tag{61}$$

with corresponding solutions

$$\mathbf{x}_\epsilon = \mathbf{x}_\epsilon^\star \tag{62}$$

- The efficacy of continuation generally depends on two crucial points:

  1. $\mathbf{N}_0(\mathbf{x};\bar{\mathbf{p}})$ has a known or easily calculable root at $\mathbf{x}_0^\star$.

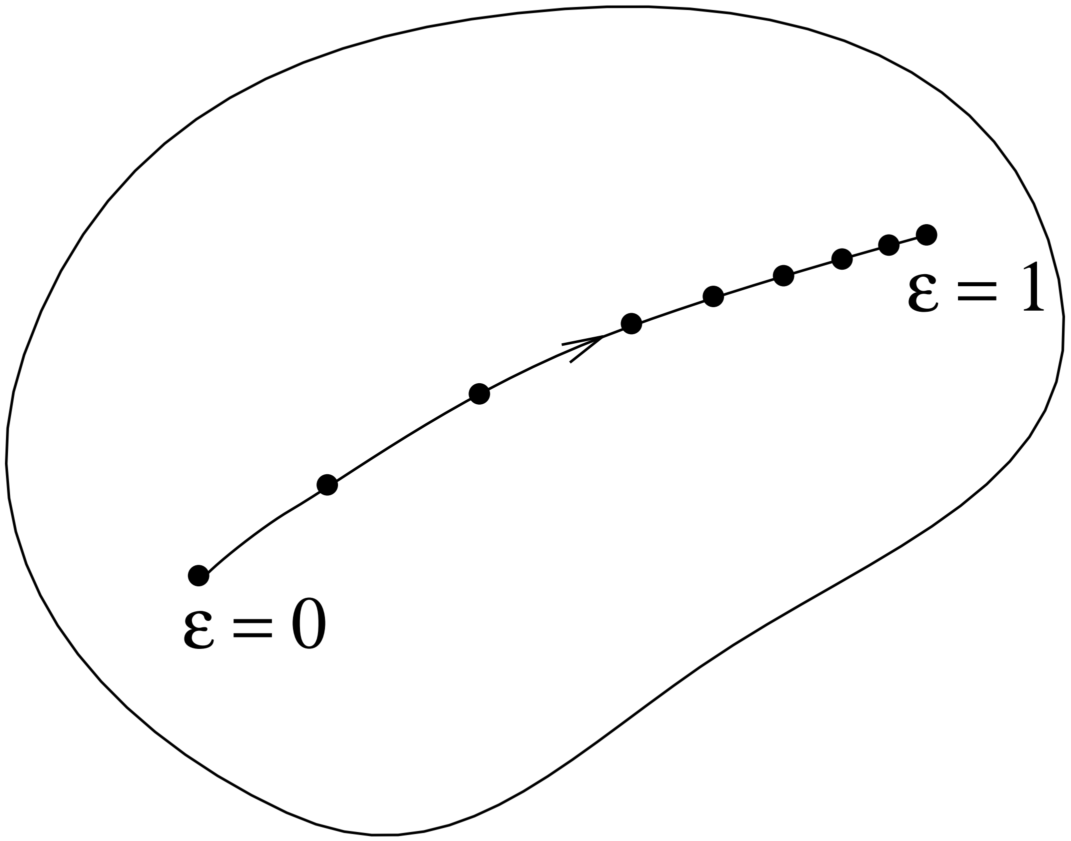
  2. Can often choose $\Delta\epsilon$ judiciously (i.e. sufficiently small) so that

  $$\mathbf{x}^\star_{\epsilon-\Delta\epsilon}$$

     is a "good enough" initial estimate for

  $$\mathbf{N}_\epsilon(\mathbf{x};\bar{\mathbf{p}}) = 0$$

- Again, schematically, we have



where we note that we may have to adjust (adapt) $\Delta\epsilon$ as the continuation proceeds.

Continuation: Summary and Comments

- Solve sequence of problems with $\epsilon = 0, \epsilon_2, \epsilon_3 \ldots 1$ using previous solution as initial estimate for each $\epsilon \neq 0$.

- Will generally have to tailor idea on a case-by-case basis.

- Can often identify ϵ with one of the p_i (intrinsic problem parameters) *per se*.

- The first problem, $\mathbf{N}_0(\mathbf{x}, \bar{\mathbf{p}}) = 0$, can frequently be chosen to be *linear*, and therefore "easy" to solve, modulo sensitivity/poor conditioning.

- For time-dependent problems, *time evolution* often provides "natural" continuation; $\epsilon \to t$, and we can use $\mathbf{x}^\star(t - \Delta t)$ as the initial estimate $\mathbf{x}^{(0)}(t)$.

SOLUTION OF THE THE KORTEWEG & DE VRIES (KDV) EQUATION

Introduction to Numerical Relativity
Lecture 6
Solving the KdV Equation

Matthew W. Choptuik
CIAR Cosmology & Gravity Program
Department of Physics & Astronomy
University of British Columbia
Vancouver BC

Spring School in
Computational Methods for Relativity & Astrophysics
KIAS, Seoul, Korea

March 17?, 2006

The Korteweg and de Vrie (KdV) Equation

- Consider the Korteweg and de Vries (KdV) equation for $u \equiv u\,(x,t)$:

$$u_t + u_x + 12\,u\,u_x + u_{xxx} = 0 \tag{1}$$

on the domain $-x_{\max} \leq x \leq x_{\max}$, $0 \leq t \leq t_{\max}$, and with initial and boundary conditions

$$u\,(x,0) = u_0\,(x) \tag{2}$$

and

$$u\,(-x_{\max},t) = u\,(x_{\max},t) = 0 \tag{3}$$

respectively

The KdV Equation

- The KdV equation admits propagating "particle-like" solutions, called *solitons*, that propagate in *one* direction ($-x_{\max} \rightarrow x_{\max}$; i.e. "to the right"). The "vacuum" (or quiescent) state is $u = \kappa$, for an arbitrary real constant κ, which, without loss of generality, we can choose to be $\kappa = 0$.

- The *boundary* conditions (BCs) are thus compatible with quiescence, and we will always ensure that the initial conditions, $u_0(x)$, are likewise compatible (i.e. $u_0(x)$ should always satisfy—at least to well within the level of truncation error—$u_0(-x_{\max}) = u_0(x_{\max}) = 0$)

- The right BC is *not* compatible with disturbances impinging on $x = x_{\max}$; thus, once any signal has reached $x = x_{\max}$, the "well-posedness" of the evolution is questionable, and one can expect "strange things" to happen.

- This problem could be remedied by working on a periodic domain (i.e. by identifying $-x_{\max}$ and $x_{\max}$), but this would also complicate the finite-difference solution of the equation.

Finite Differencing

- We use an $O(h^2)$ Crank-Nicholson finite-difference scheme combined with a multi- dimensional Newton iteration to approximately solve (1)

- Specifically, we use a difference scheme centred at $t = t^{n+1/2} \equiv t^n + \triangle t/2$ and of the form

$$\frac{u_j^{n+1} - u_j^n}{\triangle t} + \mu_t \left(D_x u_j^n\right) + 12 \left(\mu_t \, u_j^n\right) \, \mu_t \left(D_x u_j^n\right) + \mu_t \left(D_{xxx} u_j^n\right) = 0 \quad (4)$$

- Here, μ_t, is the time averaging operator

$$\mu_t \, v_j^n \equiv \frac{1}{2}\left(v_j^n + v_j^{n+1}\right) \quad (5)$$

and D_x and D_{xxx} are centred, $O(h^2)$ FD approximations of ∂_x and ∂_{xxx} respectively

$$D_x v_j \equiv \frac{v_{j+1} - v_{j-1}}{2h} \quad (6)$$

$$D_{xxx} v_j \equiv \frac{v_{j+2} - 2v_{j+1} + 2v_{j-1} - v_{j-2}}{2h^3} \quad (7)$$

Solving the Algebraic Equations

- The discretization defined above yields a set of nonlinear equations for the *interior* unknowns

$$F_j\left[u_{j'}^{n+1}\right] = 0 \qquad j = 3, 4, \ldots, \mathtt{nx} - 2\,; \quad j' = 1, 2, \ldots, \mathtt{nx} \tag{8}$$

  to which we adjoin the following 4 equations:

$$u_1^{n+1} = u_2^{n+1} = u_{\mathtt{nx}-1}^{n+1} = u_{\mathtt{nx}}^{n+1} = 0 \tag{9}$$

- We then have a set of J equations in the J unknowns u_j^{n+1}, $j = 1, 2, \ldots, J$.
- Jacobian matrix of system is *5-diagonal* (*pentadiagonal*)
- At each time step then, we solve for the u_j^{n+1} using an J-dimensional Newton method, and the `LAPACK` banded-solver, `dgbsv`, to solve the linear systems which arise in the

Convergence Criterion for Newton Iteration

- The specific convergence criterion used for the Newton iteration is

$$\frac{\|\delta \mathbf{u}\|_2}{\|\mathbf{u}\|_2} \le 1.0^{-10} \tag{10}$$

which ensures that, at the resolutions we are apt to be using, that the residuals with respect to the Newton iteration are far below the level of truncation error when convergence is achieved.

Initial Data

- For illustrative purposes, we initialize the solution at the initial time using the known, closed form soliton solutions

$$u_0(x) = \sum_{i=1}^{n_{\mathrm{p}}} \Pi\left(x;\, a_i, x_{0i}\right) \tag{11}$$

  wher where

$$\Pi\left(x;\, a, x_0\right) = \frac{1}{4} a^2 \cosh^{-2}\left[\frac{1}{2} a\left(x - x_0\right)\right] \tag{12}$$

- Note that $\Pi\left(x;\, a, x_0\right)$ is a "pulse" profile whose maximum amplitude scales with a and which is centred at $x = x_0$.

- Thus 11 generically represents the superposition of n_{p} separate pulses.

- One of interesting features of these solitons is that their propagation speed depends on their amplitude ("taller" solitons move more rapidly)

The Code

```
c===========================================================
c     Solves 1-dimensional KdV equation
c
c        u_t + u_x + 12 u u_x + u_xxx = 0
c
c        0 <= x <= 1   u(0,t) = u(1,t) = 0
c
c     using second-order finite difference techniques,
c     Newton's method and LAPACK banded solver DGBSV
c===========================================================
      program         kdv

      implicit        none
                    .
                    .
                    .
```

The Code

```
c===========================================================
c     Updates KdV difference equations.
c===========================================================
      subroutine update(unp1,un,nx,dx,dt,
     &                  jacb,res,ipiv,ldjacb,
     &                  tol,mxiter,rc)

         implicit       none

         real*8         dvnrm2

         real*8         drelabs

         integer        nx,         ldjacb
         real*8         jacb(ldjacb,nx),
     &                  unp1(nx),   un(nx),    res(nx)
         integer        ipiv(nx)
         real*8         dx,         dt,        tol
         integer        mxiter,     rc

         real*8         dtm1,       hdxm1,     qdxm1,
     &                  hdxm3,      qdxm3,
     &                  t0,        t1,         t2,         t3
         integer        iter ,      i,         j

         real*8         nrmu,       nrmdu
         integer        info

         logical        ltrace_newt
         parameter    ( ltrace_newt = .false. )
```

The Code

```
c-----------------------------------------------------------
c        'rc' is return code, 0 for success, -1 if linear
c        solve fails, -2 if Newton iteration does not
c        converge.
c-----------------------------------------------------------
         rc    = 0

c-----------------------------------------------------------
c        Define some useful constants.
c-----------------------------------------------------------
         dtm1  = 1.0d0 / dt
         hdxm1 = 0.5d0 / dx
         qdxm1 = 0.5d0 * hdxm1
         hdxm3 = 0.50d0 / (dx**3)
         qdxm3 = 0.25d0 / (dx**3)
```

The Code

```
c-----------------------------------------------------------
c        B E G I N   N E W T O N   L O O P
c-----------------------------------------------------------
         do iter = 1 , mxiter
c-----------------------------------------------------------
c           Set up Jacobian and evaluate residuals ...
c-----------------------------------------------------------

c-----------------------------------------------------------
c           Left boundary conditions (Dirichlet).
c-----------------------------------------------------------
            i = 1
            do j = 1 , 3
               if( i .eq. j ) then
                  jacb(5 + i - j,j) = 1.0d0
               else
                  jacb(5 + i - j,j) = 0.0d0
               end if
               res(i) = 0.0d0
            end do

            i = 2
            do j = 1 , 4
               if( i .eq. j ) then
                  jacb(5 + i - j,j) = 1.0d0
               else
                  jacb(5 + i - j,j) = 0.0d0
               end if
               res(i) = 0.0d0
            end do
```

The Code

```
c------------------------------------------------------------
c           Interior equations.
c------------------------------------------------------------
            do i = 3 , nx - 2
               t0 = dtm1 * (unp1(i) - un(i))
               t1 = 1.0d0 + 6.0d0 * (unp1(i) + un(i))
               t2 = qdxm1 * ( unp1(i+1) - unp1(i-1) +
     &                        un  (i+1) - un  (i-1) )
               t3 = qdxm3 * ( unp1(i+2) - 2.0d0 * unp1(i+1) +
     &                        2.0d0 * unp1(i-1) - unp1(i-2) +
     &                        un  (i+2) - 2.0d0 * un  (i+1) +
     &                        2.0d0 * un  (i-1) - un  (i-2) )
               res(i) = t0 + t1 * t2 + t3
               j = i - 2
               jacb(5 + i - j,j) = -qdxm3
               j = i - 1
               jacb(5 + i - j,j) = -qdxm1 * t1 + hdxm3
               j = i
               jacb(5 + i - j,j) = dtm1 + 6.0d0 * t2
               j = i + 1
               jacb(5 + i - j,j) = +qdxm1 * t1 - hdxm3
               j = i + 2
               jacb(5 + i - j,j) = +qdxm3
            end do
```

The Code

```
c-------------------------------------------------------------
c           Right boundary conditions (Dirichlet).
c-------------------------------------------------------------
            i = nx - 1
            do j = nx - 3 , nx
               if( i .eq. j ) then
                  jacb(5 + i - j,j) = 1.0d0
               else
                  jacb(5 + i - j,j) = 0.0d0
               end if
               res(i) = 0.0d0
            end do

            i = nx
            do j = nx - 2 , nx
               if( i .eq. j ) then
                  jacb(5 + i - j,j) = 1.0d0
               else
                  jacb(5 + i - j,j) = 0.0d0
               end if
               res(i) = 0.0d0
            end do
```

The Code

```
c-----------------------------------------------------------
c           Solve linear system.
c-----------------------------------------------------------
            call dgbsv( nx, 2, 2, 1, jacb, ldjacb, ipiv,
     &                  res, nx, info )

c-----------------------------------------------------------
c           Update unknown vector if solution linear solve
c           succeeded, otherwise return.
c-----------------------------------------------------------
            if( info .eq. 0 ) then
               call dvvs(unp1,res,unp1,nx)
               if( iter .eq. 1 ) then
                  nrmu = dvnrm2(unp1,nx)
               end if
               nrmdu = dvnrm2(res,nx)
               if( ltrace_newt ) then
                  write(0,*) iter, drelabs(nrmdu,nrmu,1.0d-10)
               end if
c-----------------------------------------------------------
c              Return if convergence achieved.
c-----------------------------------------------------------
               if( drelabs(nrmdu,nrmu,1.0d-10) .le. tol )
     &            return
            else
               write(0,*) 'update: dgbsv() failed, info = ',
     &                    info
               rc = -1
               return
            end if
```

The Code

```
c-----------------------------------------------------------
c        E N D   N E W T O N   L O O P
c-----------------------------------------------------------
         end do

c-----------------------------------------------------------
c        Non-convergence return.
c-----------------------------------------------------------
         write(0,*) 'update: Newton iteration failed'
         rc = -2
         return

      end
```

References

[1] Choptuik, MW. Problem handout for Phys 381C, UT Austin, 1997