Lab #4 for Geophysical Inverse Theory ES5523, Fall 2005, Univ. of WA.
TA presenting this lecture: Andy Ganse
Course professor: Ken Creager

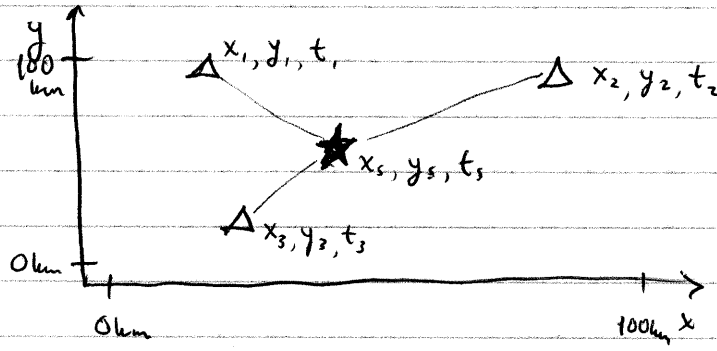
8 Nov 2005

Today we'll focus on a nonlinear parameter estimation problem to learn some concepts important to solving nonlinear inverse problems, but since this won't technically be an inverse problem today, we'll be able to conveniently skip issues such as resolution and regularization that come up in inversion. We can concentrate on those another time.

Now, if we have a nonlinear parameter estimation problem (or inverse problem as mentioned a moment ago), the methods we learned for linear problems won't cut it for finding our solution. But if the problem is not "too nonlinear", ie if it's "weakly nonlinear" enough, we can use those linear techniques in a series of steps and iterate our way to a solution. What do we mean by "weakly nonlinear"? That's most easily understood when we discuss objective surfaces in the next lab lecture. For now let us just assume there are some very useful problems out there (including travel time inversion of wave slownesses) which are nonlinear, but not so crazy that we can't use this iterative linearization procedure we'll discuss today. So now let's go ahead and launch into one variant of this technique, ~~while~~ demonstrating it on a really simple but still nonlinear source location problem.

By the way, in learning this technique we'll also introduce today a tool called "finite differences", which are the discrete approximation of derivatives. The simple demonstration problem here doesn't require them, but your own weakly nonlinear problem might need you to use these.

Our problem today is to estimate the time and x, y location of an earthquake from measurements of P-wave arrival times at a set of seismometer stations. The P-wave speed V_p here is constant everywhere in this problem so that the paths of the waves are straight.



assume 2D (no topography) and constant $V_p = 5.5$ km/s (so ray paths are straight)

In practice we might set the origin at one of the stations, since it's arbitrary, but I didn't do that here. It doesn't matter. Note too that we have 3 parameters we're estimating — the 2D location x_s, y_s of the source, and then also the source time t_s of the event. But on our map there we're only plotting 2 of the parameters. So recall from what we learned in Lab #2 and the textbook that this would affect the error ellipse we plot, since the 3D error ellipsoid for $\{x_s, y_s, t_s\}$ gets projected down onto the $\{x_s, y_s\}$ plane. No problem, we just snip the relevant code for that out of Lab #2. And we won't bother with that until we add to this lab problem in the next lab. anyway.

Our problem definition here is:
Given constant station locations x_i, y_i and P-wave arrival times t_i at those stations, we want to estimate the epicenter and source time $\{x_s, y_s, t_s\}$ of the earthquake.
So we're estimating a model vector $\underline{m} = \{x_s, y_s, t_s\}$.
And maybe we have say 7 stations, so $i = 1..7$.

The problem can be written like this, ie this is the "forward problem":

$$t_i = f_i(\underline{m})$$

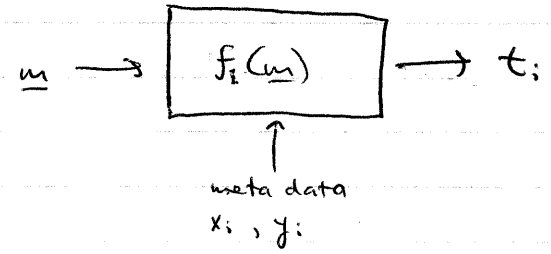
or equivalently:

$$t_i = f(\underline{m}; x_i, y_i)$$

where $\underline{m} = \{x_s, y_s, t_s\}$

and given the t_i's we estimate $\underline{m}$.

Where do the x_i and y_i come into play? They're part of the f_i. Sometimes we call the x_i and y_i "meta-data" — information that's part of the geometry, or maybe frequency, that we just stick into the problem. The difference between "data" t_i and "meta-data" such as x_i, y_i is that the meta-data are simply part of the forward problem function — they stay constant as the $\underline{m}$ gets changed to fit the t_i's. The x_i, y_i stay constant because the station locations don't change. We can draw the problem like this, as a "black box":



For this problem we can easily derive a mathematical expression for the $f_i(\cdot)$, and since this technique requires the derivatives of f_i (as we'll see shortly), we'll be able to easily derive expressions for those in this problem. But this "black box" representation of $f_i()$ there is handy, because in a more complicated problem $f()$ may be too complicated to derive derivative expressions for, or maybe $f()$ is actually an algorithm (computer program). In such cases we can still use this technique from today, by approximating the derivatives of $f()$ with finite differences, which treat $f()$ as a black box. Back to that in a moment.

We can find our $f_i()$ here by visual inspection of the problem geometry:

$$t_i = \text{arrival time for station } i = \text{travel time}_i + \text{source time} = \frac{\text{distance}_i}{\text{velocity}} + t_s =$$

$$= \sqrt{(x_s - x_i)^2 + (y_s - y_i)^2} \Big/ v_p + t_s$$

$$= f_i(x_s, y_s, t_s) = f_i(\underline{m}) \qquad \text{...where } \underline{m} = \{x_s, y_s, t_s\}$$

- remember v_p is constant here to keep things simple.
- notice that the t_i are linearly related to the t_s, but nonlinearly related to the x_s and y_s, so overall $f(\underline{m})$ is a nonlinear function in $\underline{m}$.

The t_i are the measurements of arrival times at the stations, so for 7 stations the are 7 arrival times and $i = 1..7$.
For this synthetic problem we must create our own measurements, so we choose some $\underline{m}$ as our $\underline{m}_{true}$ and use the station locations we made up to calculate our t_i's. In the <u>next</u> lab assignment which is a follow-on to this one, we'll do as we did in Lab #2 and add noise to our computed t_i's, and this will feed into our error ellipse calculation for our $\underline{m}$ estimate.
In that case we would calculate our t_i like:

$$t_i = \text{arrival time}_i + \text{noise} = f_i(\underline{m}) + t_{noise}$$

where $t_{noise} \sim N(0, \sigma_{noise})$
i.e., normally distributed with
mean = 0 and stdev = σ_{noise}

And then we could use these σ_{noise} when computing the error ellipsoid for $\underline{m}$.
But for today we'll skip the noise entirely, and concentrate on the linearization and finite differences.

Nonlinear parameter estimation by "local linearization":

Recall that a Taylor series expansion of some scalar function $f(x)$ lets us expand $f(x)$ about a point x_0 like this:

$$f(x) = f(x_0) + \frac{df(x_0)}{dx}(x - x_0) + \frac{1}{2}\frac{d^2f(x_0)}{dx^2}(x - x_0)^2 + \cdots$$

constant bias term — linear term — quadratic term

Let's rewrite this with a substitution to put it into a form more adaptable to our iterative scheme:

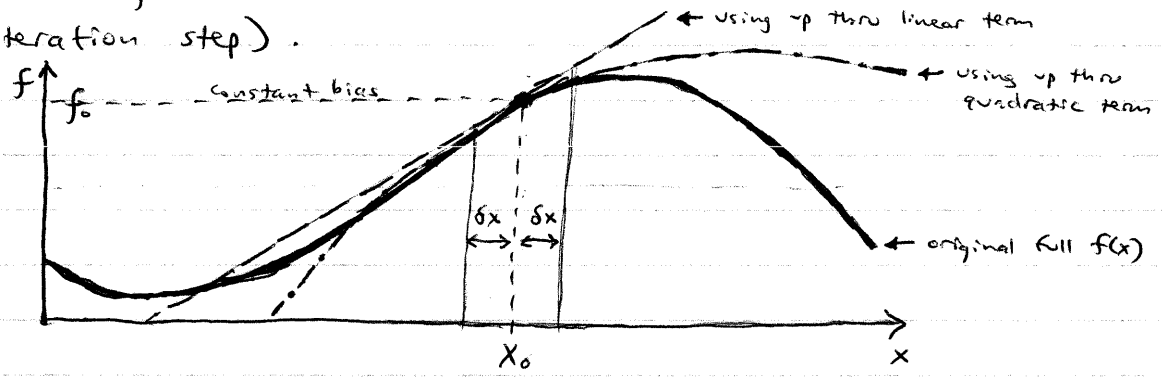
Let $x = x_0 + \delta x$

note these are the derivatives at the point x_0

$$f(x_0 + \delta x) = f(x_0) + \frac{df(x_0)}{dx}\delta x + \frac{1}{2}\frac{d^2f(x_0)}{dx^2}\delta x^2 + \cdots$$

See how that's like an update from an old x_0 to a newer estimate x using a δx (which is what will be solved for at each iteration step).

example of Taylor series expansion components:



If δx is small, we can justify dropping the higher order terms and approximating $f(x)$ by the linear terms. For δx to be small we can see that our old point x_0 must be close to our new point x. This is what we mean by "local linearization" — the linearity is only valid locally around x_0.

Well that Taylor series expansion business works on higher dimensional vector functions too:

$$f_i(\underline{m}) = f_i(\underline{m}_0 + \delta\underline{m}) = f_i(\underline{m}_0) + \frac{\partial f_i(\underline{m}_0)}{\partial m_j}\,\delta\underline{m} + \cdots^{0}$$

(dropping the terms of higher order than linear)

- $f_i(\underline{m})$: these are the measured data points d_i. In our problem today these are the arrival times t_i for the 7 stations that we calculated.
- $f_i(\underline{m}_0)$: data points predicted at $\underline{m}_0$
- $\frac{\partial f_i(\underline{m}_0)}{\partial m_j}$: this matrix of partial derivatives is often called the Jacobian matrix $\underline{\underline{J}}$, which is an annoying name because note there is also "the Jacobian" which is the determinant of $\underline{\underline{J}}$. Note that $\underline{\underline{J}}$ here is really $\underline{\underline{J}}(\underline{m}_0)$.

If we reorganize this locally linearized problem we'll see we have a linear relation between our model vector perturbation $\delta\underline{m}$ and our data residuals:

$$\text{data residuals } \delta d_i = f_i(\underline{m}) - f_i(\underline{m}_0) = d_i - f_i(\underline{m}_0)$$

... or the same thing as vectors:

$$\delta\underline{d} = \underline{d} - \underline{f}(\underline{m}_0)$$

($\underline{f}(\underline{m})$)

So from above we have:

$$\delta\underline{d} = \underline{d} - \underline{f}(\underline{m}_0) = \underline{\underline{J}}(\underline{m}_0)\,\delta\underline{m}$$

(i.e., $\underline{\underline{J}}$ here is a function of $\underline{m}_0$)

So now given a $\underline{m}_0$ we're back to our old linear problem, in which here we want to estimate $\delta\underline{m}$.
Once we have $\delta\underline{m}$, then we can add it to $\underline{m}_0$ to update our estimate and repeat the process.
When do we stop repeating the process? Generally you might specify some small tolerance value for $\delta\underline{m}$, say via norm $(\delta\underline{m})$.
But note that for each iteration you must recalculate the residuals $\delta\underline{d}$ based on $\underline{f}(\underline{m}_0)$, and the new derivatives $\underline{\underline{J}}(\underline{m}_0)$, since the $\underline{m}_0$ changed.

To start that iterative process we must have an initial $\underline{m}_0$ that we choose — we should try to pick an $\underline{m}_0$ that's close to where we think the solution is, because as we'll see in this lab, in a nonlinear problem this choice of $\underline{m}_0$ can yield different solutions depending on where we choose it.

We know how to compute the $\underline{f}(\underline{m}_0)$ each iteration, that's just the travel times per our equation for the t_i's. If we have a tractable analytic expression we can evaluate it to find the derivatives in the matrix $\underline{\underline{J}} = \frac{\partial f_i}{\partial m_j}$ (which depends on $\underline{m}_0$). But if our forward problem $f(m)$ is an algorithm (computer program) we may need to approximate those derivatives in $\underline{\underline{J}}$ by finite differences. We'll do both in this lab since it's simple here, so you can learn how to use finite diffs in your own projects.

By the way, another option that's faster and more accurate than finite differences for the algorithm case is called "automatic differentiation" (also sometimes called "algorithmic differentiation"). The catch is it requires certain high-quality/modern constraints on the form of your code, which — let's be honest — doesn't happen often in the scientific community! But that works for C, Fortran, and Matlab — check out www.autodiff.org. If you have a software background and also wrote your own forward code, this is ideal.

Okay, now for the finite differences...

Finite differences

These are simply numerical approximations to derivatives. A convenient property of these things is generally they let you treat your forward problem (which you're finding derivatives of) as a black box.

All they are is just the old definition of derivative but with the limit not all the way to zero. So instead of referring to:

$$\underline{\underline{J}}(\underline{m}_0) = \frac{\partial f_i(\underline{m}_0)}{\partial m_j} = \begin{bmatrix} \frac{\partial f_1(\underline{m}_0)}{\partial m_1} & \frac{\partial f_1(\underline{m}_0)}{\partial m_2} & \cdots \\ \frac{\partial f_2(\underline{m}_0)}{\partial m_1} & \ddots & \\ \vdots & & \end{bmatrix}$$

We'll instead refer to:

$$\underline{\underline{J}}(\underline{m}_0) = \frac{\Delta f_i(\underline{m}_0)}{\Delta m_j} = \begin{bmatrix} \frac{\Delta f_1(\underline{m}_0)}{\Delta m_1} & \frac{\Delta f_1(\underline{m}_0)}{\Delta m_2} & \cdots \\ \frac{\Delta f_2(\underline{m}_0)}{\Delta m_1} & \ddots & \\ \vdots & & \end{bmatrix}$$

Our $f_i()$ is a function of 3 arguments, m_1, m_2, m_3. We just perturb one argument at a time, so:

$$\frac{\Delta f_i(\underline{m}_0)}{\Delta m_1} = \frac{f_i(m_{01} + \Delta m_1, m_{02}, m_{03}) - f_i(m_{01}, m_{02}, m_{03})}{\Delta m_1}$$

where $\underline{m}_0 = \{m_{01}, m_{02}, m_{03}\}$.

That there gives you the first column of $\underline{\underline{J}}$; then you repeat for Δm_2 and Δm_3 for the 2nd & 3rd columns of $\underline{\underline{J}}$. Here's a quick snippet of Matlab code to compute that for you for an arbitrary length of m_0, not just 3, so you can use it for your own projects:

Note code corrections below – sorry about that!


```
% Matlab code for "forward differences":
m0 = [ 14 , 24 , 19 ]' ;  % simply use a longer vector for your projects
dm = [.001 , .001 , .01]' ;  % length of dm must equal length of m0
f0 = fwdprob(m0) ;
for j=1:length(m0)
    mask=zeros(length(m0)); mask(j,j)=1;
    J(:, j) = (fwdprob(m0 + mask * dm ) - f0 ) ./ dm(j) ;
end

% farther below you must have your forward problem in a function:
function outvector = fwdprob(invector)
   % Whatever computations using invector(1), invector(2), etc.
   % place output into outvector(1), outvector(2), etc.
   % If you're familiar with Matlab you can even use the "system"
   % statement in here to call C or Fortran programs at the commandline.
end
```

This type of finite difference is called "forward difference" (and notice from it you've computed both $f(m0)$ and J at $m0$). The "forward" name comes from noting in its definition that you're subtracting the f value at m_0 from the f value a step ahead of m_0 ($m_0 + \Delta m$). There's another similar finite diff called the "backward" diff, which uses a step behind m_0 ($m_0 - \Delta m$).

The forward and backward finite diffs are nice and efficient since you only compute fwdprob(m0) once but it's used in the whole matrix, and then you have f0 already computed for you too. That's nice if fwdprob() is slow to run.
However, these types of finite diffs aren't as accurate as they could be, because note they aren't centered over m_0.

If you want a more accurate finite difference there are "central differences", which are defined as:

$$\frac{\Delta f_i(\underline{m}_0)}{\Delta m_1} = \frac{f_i(m_{01} + \frac{1}{2}\Delta m_1, m_{02}, m_{03}) - f_i(m_{01} - \frac{1}{2}\Delta m_1, m_{02}, m_{03})}{\Delta m_1}$$

where again $\underline{m}_0 = \{m_{01}, m_{02}, m_{03}\}$.

The tradeoff is that there's a lot more fwdprob() evaluations there, and then if you want $f\emptyset$ = fwdprob($m\emptyset$) (which you will) you'll need to go and additionally compute that separately. I have a fwdprob() for example which takes 25sec to compute (it calls an old Fortran beast written before I was born!) and its $\underline{m}$ is length 180. So my using the forward diffs instead of central diffs saved a ton of time once I verified that they were accurate enough for my problem.

Speaking of which, how do we choose Δm? (I.e. the dm in the Matlab code snippet.)

★ Note the Aster/Borchers/Thurber text has a good discussion of this in §9.4 — Implementation Issues (of Nonlinear Regression problems). We want the Δm small enough so that the derivatives are approximated as accurately as possible. But we want it large enough so that it doesn't cause the difference between $f(m_0)$ and $f(m_0 + \Delta m)$ to be smaller than the accuracy of f(), causing wacky values. Those wacky values could look random, or, if the result of f() is written to a text file before read into Matlab, and say they're written to 6 decimal places, then if Δm is small enough to cause diffs only in the 7th decimal place, my finite diffs will be zero! (Because $f(m_0)$ and $f(m_0 + \Delta m)$ look the same.)

Aster/Borchers/Thurber recommends $\Delta m = \sqrt{\epsilon}$, where ϵ is accuracy of f().

That same implementation issues section also discusses what values make sense to use as the tolerance of $\underline{\delta m}$ to stop iterating at, as discussed back on page 6 of today's notes.

Okay, lastly, now here's exactly what to do for today's lab:

There's a Part A and a Part B to this assignment, and the only difference is the choices of station locations and initial guesses. You'll need to write a Matlab script to do Part A first anyway, so then Part B is simply rerunning your script again after specifying different x_i, y_i, and $\underline{m}_0$.

★ In your Matlab script:

1.) choose numbers for 7 station locations x_i, y_i and an $\underline{m}_0 = [x_{s0}, y_{s0}, t_{s0}]'$. (initial guess of location)

2.) analytically calculate (by hand) expressions for the derivatives matrix $\underline{\underline{J}}$ from the forward problem expression on page 4 (easy!). Also create finite difference code to approximate those derivatives. Calculate the 7x3 matrix $\underline{\underline{J}}$ (for 7 stations) both ways (ie put the analytic expression for derivs into Matlab too) and compare the matrices for some choice of $\underline{m}_0$. Are they close?

3.) Choose some numbers to fill in $\underline{m}_{true} = [x_s, y_s, t_s]'$.

4.) Use your fwdcode() function from the finite diffs to compute your synth data from your mtrue. Your synth data is a vector of the t_i's, the arrival times at each station. So your data vector will be of length 7.

5.) Calculate your $J = J(m_0)$ matrix. This must be recomputed every iteration.

6.) Calculate your residuals deltad = $t_i - f(m0)$, and note that now your locally linear equation will be deltad = J(m0)*dm

7.) Use the normal equation method from lab #2 to solve for dm.

8.) set vector m_new = m0 + dm

9.) plot the x_s and y_s components of m_new on your plot, as well as those for $m\emptyset$, and use the "hold on" option for plotting so that you can plot future iterations on there too. Also list out x_s, y_s, t_s at each iteration so at the end you have a table of them over your 5-8 iterations. You can use the "disp" and "num2str", or "fprintf" commands for this.

10.) now set $m\emptyset$ = m_new

11.) go back to step #5 and do this loop 5-8 times and see how the estimates of x_s, y_s, t_s improve. Generally we would look at the size of $\underline{dm}$ (eg norm(dm)) to decide when to stop iterating. But let's just do 5-8 iterations here.

for looping you can use the "for" command.

Part A:

Have your stations located in more of a "cloud" or "spread out" formation. For part A your initial guess $m\emptyset$ can be anywhere.

Part B:

Have your stations located all in a line in the middle of the map. Eg. with all having the same x=50km coordinate. This will cause ambiguity in your estimate of the source location, mirrored about that line. Compare runs of:

i.) an initial guess on the same side of the line as the true source location.

ii.) an initial guess on the opposite side of the line as the source.

So you'll turn in three plots, each showing the sequence of location estimates iteration-by-iteration along with the "true" source location; that's one plot for part A and two for B. But this should all be in a Matlab script, so it's no more work than for just part A.

See you next time! - end