Non-Linear Optimization

CS522 – Spring 2005

Tibor Janosi

Time vs. Simplicity in Matlab

```
>> for N = [100, 500]
  tic; for i = 1:N; for j = 1:N; a(i, j) = 17; end; end; toc
  tic; b(N, N) = 17; for i = 1:N; for j = 1:N; b(i, j) = 17; end; end; toc
  tic; c(1:N, 1:N) = 17; toc
  clear all;
end
```

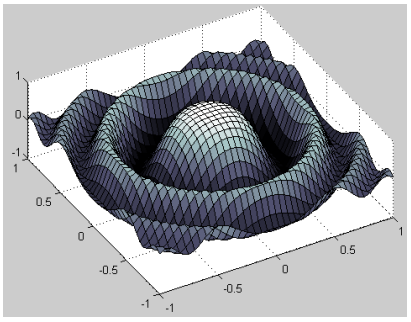
	N = 100	N = 500
	0.080	5.578
	0.040	1.192
	0	0.010

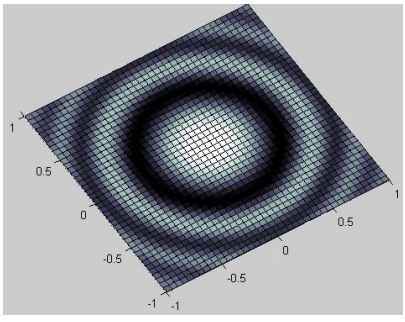
Two simple solutions to a trivial program can have a difference in speed of 1 to 550! What if you perform the loop a million times? **Let Matlab's implicit loops do the work whenever possible, i.e. use vectorization.** (Note: These time values depend on the characteristics of, and the current state of your computer.)

Non-Linear Optimization

- Must have access to optimization toolbox. The CSUGLAB should have it, your home version might not.
- Two functions of interest: **lsqcurvefit** and **lsqnonlin**.
- Non-linear optimization is as much an art as a science. One needs to adjust parameters (e.g. convergence criteria) to balance the time spent on a computation and the quality of the results.
- Global optimum (minimum) not guaranteed!

Local Minima Are a Big Problem





Optimizers at Work

```
% Generate a regular grid of values.
[x y] = meshgrid(-2:.1:2);

% Reshape the matrices
x = reshape(x, [prod(size(x)), 1]);
y = reshape(y, [prod(size(y)), 1]);

% Distance from origin in the XY plane
r = sqrt(x .* x + y .* y);

% Compute values.
vals = exp(-r) .* cos(10 * r);

% Add some random, normally distributed error.
vals = vals + .1 * randn(size(vals));

options = optimset('Display', 'iter');

% One could define lower and upper bounds for parameters.
lb = []; ub = [];

% lsqcurvefit(fun, x0, xdata, ydata)
[params, sse, res] = lsqcurvefit(@hat, [0, 1], [x y], vals, lb, ub, options);

params
```

```
function v = hat(params, xdata)
  x = xdata(:, 1);
  y = xdata(:, 2);
  r = sqrt(x .* x + y .* y);
  v = exp(-params(1)*r).*cos(params(2)*r);
```

Optimizers at Work (2)

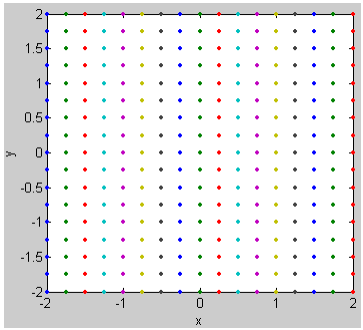
- Step 1: Acquire “measurement” data.
- Step 2: Implement model.
- Step 3: Specify option values and initial values for parameters.
- Step 4: Run regression.
- Step 5: Process results (e.g. save in file, plot values).

Step 1: Acquire Data

- Our example is artificial: we "synthesize" data by implementing an exact mathematical model and perturbing it with noise.

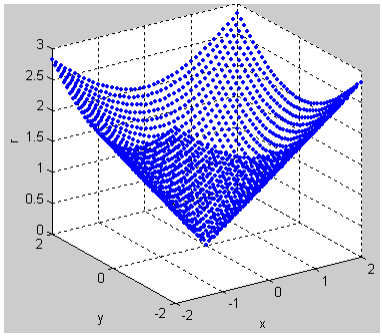
```
>> % Generate a regular grid of values.
>> [x y] = meshgrid(-2:.25:2);
>> plot(x, y, '.');
>> xlabel('x')
>> ylabel('y')
```

For this plot only, we changed .1 to .25.



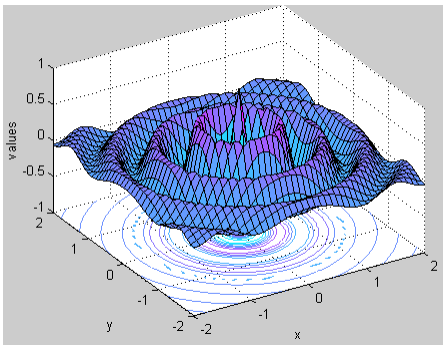
Step 1: Acquire Data (2)

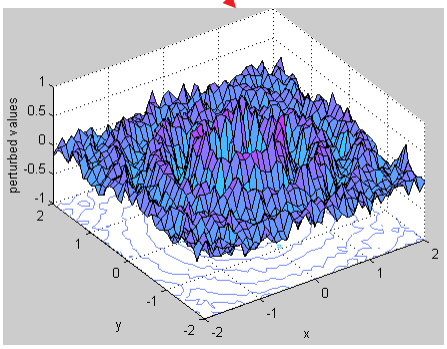
```
>> size(x)

ans =
    41    41

>> prod(size(x))

ans =
        1681

% Reshape the matrices
x = reshape(x, [prod(size(x)), 1]);
y = reshape(y, [prod(size(y)), 1]);

% Distance from origin in the XY plane
r = sqrt(x .* x + y .* y);
```



Step 1: Acquire Data (3)

```
% Compute values.
vals = exp(-r) .* cos(10 * r);

% Add some random, normally distributed error.
vals = vals + .1 * randn(size(vals));
```





Step 2: Implement Model

```
function v = hat(params, xdata)
  x = xdata(:, 1);
  y = xdata(:, 2);
  r = sqrt(x .* x + y .* y);
  v = exp(-params(1)*r).*cos(params(2)*r);
```

- Args appear in the first function argument, in the **same order** in which initial parameter values will be given.
- **xdata**: points where the model (function) is evaluated. In general, xdata will be a multicolumn array; each column corresponds to one independent coordinate.

Step 2: Implement Model (2)

- Use vectorization!

```
function v = hat(params, xdata)
  x = xdata(:, 1);
  y = xdata(:, 2);
  r = sqrt(x .* x + y .* y);
  v = exp(-params(1)*r).*cos(params(2)*r);
```

- Bad solution:

```
function v = hat(params, xdata)
  x = xdata(:, 1);
  y = xdata(:, 2);
  for i = 1:size(x, 1)
    r = sqrt(x(i) .* x(i) + y(i) .* y(i));
    v(i) = exp(-params(1)*r).*cos(params(2)*r);
  end
```

Step 2: Implement Model (3)

- Third argument of model function can be used to transmit values and arguments that are needed for the computation, but are not variable arguments, nor points where the model is evaluated.
- This must be done in conjunction with the use of an additional argument in **lsqcurvefit** (or a similar function).

Step 3: Specify Option Values

- Use **optimset** to set options, or to get their default values. Use **optimget** to retrieve option values. Not all options are used by all optimization functions (see docs).

```
% Shows the progress of the optimization;
% this is a good way to get visual feedback.
options = optimset('Display', 'iter');
```

- Check out: **MaxFunEvals**, **MaxIter**, **TolFun**, **TolX**.

Step 4: Run Regression

```
% One could define lower and upper bounds for parameters.
lb = []; ub = [];

% lsqcurvefit(fun, x0, xdata, ydata)
[params, sse, res, flag] = lsqcurvefit(@hat, [0, 1], [x y], vals, lb, ub, options);
```

- [0 1] = initial values for the parameters
- [x y] = points where the function is evaluated
- vals = function values (measurements, observations)
- lb, ub = lower and upper bounds for args (not used here)
- options = … well, options
- params = value of parameters at the **LOCAL** optimum
- sse = sum of squared errors (residuals = model – measurement)
- res = vector of residuals
- flag = conveys information on the success of the minimization

Step 4: Run Regression (2)

```
                                         Norm of      First-order
 Iteration  Func-count     f(x)          step          optimality   CG-iterations
     1          4         539.204             1           634           0
     2          7         225.218      0.582754           108           1
     3         10         142.936      0.802229          21.7           1
     4         13         112.564       1.33095          5.47           1
     5         16         94.6193       3.77522          1.19           1
     6         19         87.3107            10        0.0685           1
     7         22         87.1573            10          0.13           1
     8         25         86.4093           2.5         0.143           1
     9         28         83.5992             5         0.555           1
    10         31         83.5992            10         0.555           1
    11         34         79.0935           2.5           1.4           0
    12         37         17.8228             5           7.8           1
    13         40         16.6546      0.135853         0.985           1
    14         43         16.6398     0.0137327        0.0182           1
    15         46         16.6398    0.000296995      0.000276           1
Optimization terminated successfully:
 Relative function value changing by less than OPTIONS.TolFun
params =
    0.99796815257395 -10.01438336970244
```

The true parameter values are 1 and 10!

Step 4: Run Regression (3)

- What if the model contains parameters that are not subject to optimization?
- Say, we fix the coefficient in the exponential, so that p1 **does not** change during optimization, while p2 **does** change.

```
function v = hat(p2, xdata, p1)
  x = xdata(:, 1);
  y = xdata(:, 2);
  r = sqrt(x .* x + y .* y);
  v = exp(-p1*r).*cos(p2*r);
```

```
[params, sse, res] = lsqcurvefit(@hat, 7, [x y], vals, lb, ub, options, 1);
```

- 7 = initial value for p2, 1 = **fixed** value for p1.
- The values passed in the 8th argument of **lsqcurvefit** can be arbitrarily complex; in particular they can be **arrays of cells** or **structs**.
- Run this with the same options as before, and you get p2 = 6!!! (Note: might not be exactly reproducible due to random noise in the data.)

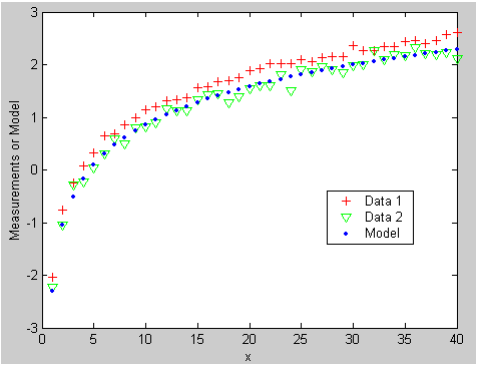
Step 4: Run Regression (4)

- Q: So how do we deal with local minima?
- A: Start close to solution (easy to say); try several algorithms; try several starting points.

```
best = +Inf;
[p1 p2] = meshgrid(-1:.1:+1, 0:10);
for i = 1:size(p1, 1)
  for j = 1:size(p2, 2)
    [params, sse, res] = lsqcurvefit(@hat, [p1 p2], [x y], vals, lb, ub, options);
    if(sse < best)
      best = sse;
      params = [p1 p2];
    end
  end
end
```

Step 4: Run Regression (5)

- **Warning**: There is more to this; one must (well, should) perform careful error analysis.



Step 5: Process Results

- It is often practical to save results to a file.
- If you want to pass results to other programs, or to let humans use them, create text files (often, CSVs): **fopen**, **fprintf**, **fclose** (also, see **diary**).
- If you want to load results back into Matlab for later processing, use **save** and **load**.
- It is a **good** idea to create self-documenting files. Include info on date, program version, parameter values, in short, anything that can give meaning to the results themselves.

Step 5: Process Results (2)

```
>> date = datestr(NOW);
>> comment = 'First version of the non-linear optimization.';
>> save 'd:/backup.mat' date comment options params -mat
>> clear all
>> comment
??? Undefined function or variable 'comment'.

>> load 'd:/backup.mat'
>> comment

comment =
First version of the non-linear optimization.

>> date

date =
22-Feb-2005 05:15:32
```

Computing Yields

```
price = 108.75;              % Bond (dirty) price.
now = datenum('02/22/2005'); % Settlement date.
mat = datenum('02/16/2010'); % Maturity date.
[cf cfd] = cfamounts(0.1175, now, mat); % Cash flows and (serial) dates.
cf = cf(2:end);                         % Eliminate accumulated interest.
cfd = yearfrac(now, cfd(2:end), 0);     % Convert time intervals to years.
peps = .0001;                           % Price error tolerance (.01 cent/$100).
ylb = 0; yub = .50;                     % Bounds for yield.
yield = [];                             % Initial value, use to test for failure.

for i = 1:50                            % Must converge in at most 50 iterations.
  ymid = .5 * (ylb + yub);              % Yield corresponding to midpoint.
  pmid = sum(cf .* exp(-cfd * ymid));   % Value of bond for mid-point yield.
  if(abs(pmid - price) <= peps)         % Are we "close" to observed price?
    yield = ymid;                       % Yes; we found the answer.
    break;                              % Leave loop.
  end
  if(pmid > price)                      % No; must restrict interval and go on.
    ylb = ymid;                         % Price too high, yield too low.
  else
    yub = ymid;                         % Price too low, yield too high.
  end
end
```

Computing Yields (2)

- Procedure on previous page returns yield = 0.09327363967896.
- We have a predefined function for yields:

```
>> bndyield(108.75, 0.1175, now, mat)
ans =
   0.09503922671516
```

- The two yields differs by 18 basis points.
- Are the two results significantly different?
- Think: How much money you must invest so that the 18 basis points difference in interest pays your salary?

Computing Yields (3)

- Bond maturity = 5 years.
- Amount invested = I.
- Your salary = $150,000/yr. (anybody interested?)
- 750,000 = I*exp(0.0932*5)*(exp(.0018*5)-1)
- I = $52,000,000
- Not a huge amount by institutional standards.
- Why does the difference arise?

Computing Yields (4)

- Well, we have a set of (semi) arbitrary parameters in our procedure. We could try to adjust them.
- Maybe we made an error.
- Maybe we are using a **different** notion of yield.
- Use `type bndyield` to list the function; see what it does.

Cells

- The fundamental datatype is the **array**; they are type-homogeneous.
- **Cell arrays** store heterogeneous data.

```
>> a{1} = 3;
>> a{2} = 'alpha';
>> a
a =
    [3]    'alpha'
```

- Often used to store strings of uneven length:

```
>> s{1} = 'short >';
>> s{2} = '< long string';
>> s{:}
ans = short >
ans = < long string

>> [s{:}]
ans =
short >< long string
```

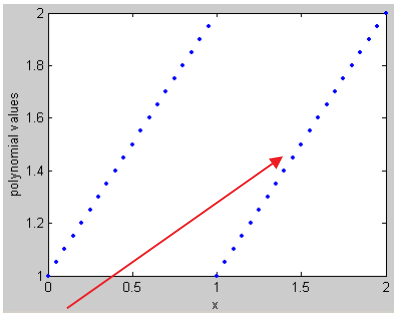
Structures

- For data that belongs together, but it is not practical to store it in a single (cell) array.

```
>> s.date = NOW;
>> s.maturity = '02/16/2005';
>>
>> s(2).date = '02/22/2005';
>> s(2).maturity = '10/10/2008';
>>
>> s.maturity

ans = 02/16/2005
ans = 10/10/2008

>> {s.maturity}

ans = '02/16/2005'    '10/10/2008'

>> [s.maturity]

ans = 02/16/200510/10/2008
```

Piecewise Polynomials

```
>> pp = mkpp([0 1 2], [1 1; 1 1]);
>> xx = 0:.01:2;
>> plot(xx, ppval(pp, xx))
>> plot(xx, ppval(pp, xx), '.')
```



pp = mkpp(breaks, coeffs)
[defines polynomials]
ppval(pp, x)
[evaluates polynomials]

f(x)=x+1, 0<=x<1 (!!)

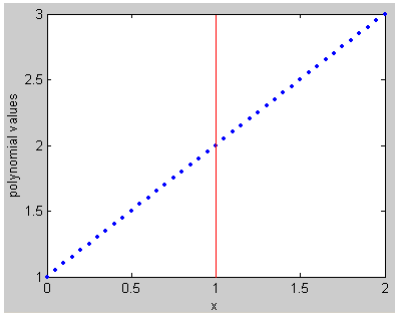
Given breaks b_1, b_2, …, b_n, if $b_i<=x<b_{i+1}$, then the i^{th} polynomial will be evaluated, but the argument will be $x-b_i$. Polynomial coefficients must be adjusted accordingly.

Piecewise Polynomials (2)

•Let us change the polynomial on the second interval, so that it corresponds to x + 1 on 1<=x<2.
x + 1 = ((x – 1) + 1) + 1 = (x – 1) + 2 => x + 2

```
>> pp = mkpp([0 1 2], [1 1; 1 2]);
>> xx = 0:.01:2;
>> plot(xx, ppval(pp, xx))
>> plot(xx, ppval(pp, xx), '.')
```

•If you rewrite your polynomials appropriately for each interval, mkpp and ppval save a lot of work.



Back to Forward Rates…

- We can rewrite the system of equations to further simplify it, by redefining the polynomials so that on interval $[t_i, t_{i+1}]$ their argument varies between 0 and t_{i+1}-t_i.
- This makes the solution immediately usable by **mkpp** and **ppval**.
- Another advantage: the difference in magnitude between various coefficients is decreases.

The Example, Revisited

$f(t) = a_{i4}t^4 + a_{i3}t^3 + a_{i2}t^2 + a_{i1}t + a_{i0}, 0 \le t \le t_i - t_{i-1}$
$t_0 = 0 < t_1 = 1 < t_2 = 2$

	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_{24}	a_{23}	a_{22}	a_{21}	a_{20}		RHS
f'(0)			2								=	
f''(0)		6									=	
p(1)	$\frac{1}{5}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{2}$	1						=	$\log \frac{1}{p_1}$
f(1)	1	1	1	1	1					-1	=	
f'(1)	4	3	2	1					-1		=	
f'(1)	12	6	2					-2			=	
f''(1)	24	6					-6				=	
p(2)						$\frac{1}{5}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{2}$	1	=	$\log \frac{p_1}{p_2}$
f'(2)						12	6	2			=	
f''(2)						24	6				=	

$$\int_0^{t_2-t_1} a_{24}t^4 dt = \frac{1}{5}\, t^5 \Big|_0^{t_2-t_1} a_{24} = \frac{1}{5}[(2-1)^5 - 0]a_{24}$$

So, Which Curve Is It?

End	**Conditions at** $t = t_e$
"fixed"	$f(t_e)$, $f'(t_e)$ given
"free"	$f''(t_e) = f'''(t_e) = 0$
"supported"	$f(t_e)$-given, $f''(t_e) = 0$
"other"	$f'(t_e)$-given, $f'''(t_e) = 0$

- Using non-linear regression we could implement any condition; unknown parameters could be discovered similarly to the p_i's.
- For our purposes, however, we will let the curve to be "free."

Homework

- Clean the data; check values and summarize values that you drop.
- Use non-linear regression for both Svensson (S) and smoothest curves (SC).
- S&SC:You might want to store the info about bonds in an array of structures (1 bond = one array element). The bonds are the "points" where you compute your function.
- SC: Choose a small number of knot points, corresponding to the underlying leftover bond maturity distribution.

Homework (2)

- SC: Your parameters are the p_i's in the knot points.
- S&SC: Choose "reasonable" initial values for parameters. You might want to cover a range of parameters, either by exploring it systematically, or by sampling it.
- S&SC: Regression parameters are critical. Precise results often require a long time, especially if you need to experiment.