Themes:
Linear equations, least squares, non-linear equations; Error estimates, polynomials
Sensitivity analysis:

EXS 3.8
Compute sensitivity of (*a* and) *b* with respect to k1, k2, atot and btot
Eliminate *a* and form the polynomial equation. Solve by roots, compute sensitivities db/dk1, db/dk2, db/datot and db/dbtot, by analytic differentiation of roots wrt coefficients and by numerical differencing.

```
% EXS 3.8
clear all
format compact
k1 = 0.12; k2 = 0.22;
atot = 0.60; btot = 1.0;
% a = atot/(1+k1b+k2b^2) = (btot-b)/(k1 b + 2k2 b^2)
% atot(k1 b + 2k2 b^2)-(btot-b)(1+k1b+k2b^2) = 0
% - btot +
%  b(atot k1 +1 -btot k1) +
%  b^2 (2 k2 atot -btot k2 + k1) +
%  b^3 k2
c = [k2,(2*k2*atot-btot*k2+k1),(atot*k1+1-btot*k1),-btot ];
z = roots(c);
i = find(imag(z)==0); % there is only one real root
x  = real(z(i))
x0 = x;
%
% Analytical differentiation
%==================================
% dp/dz* dz/dk + dp/dk = 0
% dz/dk = -dp/dk / dp/dz;
dc      = polyder(c);
dpdz    = polyval(dc,x);
%
dcdk1   = [0, 1          , atot-btot, 0 ];
dcdk2   = [1, 2*atot-btot, 0        , 0 ];
dcdatot = [0, 2*k2       , k1       , 0 ];
dcdbtot = [0, -k2        ,-k1       ,-1 ];
% dp/dk:
dpdk1   = polyval(dcdk1  ,x);
dpdk2   = polyval(dcdk2  ,x);
dpdatot = polyval(dcdatot,x);
dpdbtot = polyval(dcdbtot,x);
% dz/dk:
dzdk1   = -dpdk1  /dpdz
dzdk2   = -dpdk2  /dpdz
dzdatot = -dpdatot/dpdz
dzdbtot = -dpdbtot/dpdz
%
% Experimental evaluation; change parameters by h
%================================
h = 1e-4; % how large?
par0 = [k1 k2 atot btot];
npar = length(par0);
dxdp = zeros(4,1);
for k = 1:npar
    par = par0;
```

```
    par(k) = par(k)+h;
    k1   = par(1);
    k2   = par(2);
    atot = par(3);
    btot = par(4);
    c = [k2,(2*k2*atot-btot*k2+k1),(atot*k1+1-btot*k1),-btot ];
    z = roots(c);
    i = find(imag(z)==0);
    x = real(z(i));
    dxdp(k) = (x-x0)/h;
end
dxdp
```

EXS 4.2

EXS 4.6 – note that normal equations become diagonal. mention FFT för periodic data.

EXS 4.13

Different linear formulations:

1. ln xj – ln x0 + k tj = 0 (the obvious)
   Compute residual vector r and estimate measurement error variance squared as $\mathbf{r}^T\mathbf{r}$/sqrt(*m*-*n*) n = #measurements (5), m = #parameters (2)

The diff. eqn implies x(tk+1)/x(tk) =exp(-k(tk+1 – tk)) so
Note equidistant table tk+1 – tk = h

2. lnxk+1 – lnxk = –k h

Solve normal equations exactly (one unknown!):
n khat = -1/h sum (lnxk+1 – lnxk) = 1/h ln (xN/x0)
Is this a good idea (only x0 and x4 are used)

To use more data,

3. lnx4 – lnx2 = – k 2 h,
   lnx3 – lnx1 = – k 2 h
   Solve normal eq's exactly again.
   4 k h = ln ((x4 x3) / (x2 x1))
   Difference?

Now use differential equation dx/dt = -kx and use difference approx.

4. (xk+1 - xk-1)/(2h) + k xk = 0 k = 1,2,3
   2h k = sum((xk+1 - xk-1)*xk) / sum(xk ^2)

..exact solution …discuss error sources

Interpolation by polynomials

EXS 5.3 – note linearity of polynomial wrt data.

- mention matlab `polyfit, polyval, polyder`
- show matlab code for evaluating the Newton form of polynomial

like Horner's scheme.

- Show also how to compute derivative from the recursion.
- Challenge – compute coeffs in "standard" polynomial from `(x,c)` for Newton

Exempel (finns inte i PP) (sorry, in swedish)

I Lab 3 får ni programmera polynom-interpolation med Newtons ansats ovan. Men interpolationen kan också göras med polyfit, dock med polynomet representerat på annat sätt. Man kan kontrollera sitt program mot polyfit genom att konvertera Newton-koefficienterna cj till monom-koefficienterna aj

$$p_n(x) = \sum_{i=1}^{n+1} a_i x^{n+1-i} = \sum_{i=0}^{n} c_i \prod_{k=1}^{i} (x - x_k)$$

Antag att vi representerar polynom med koefficienterna {ci}, i = 0,…,n och {xi}, i = 1,…,n+1. Ge en algoritm för beräkning av pn(x) som använder n multiplikationer, och en algoritm för beräkning av derivatan dpn/dx.

Lösning

Om man programmerar formeln ovan som den står går det åt ungefär n2/2 multiplikationer. Men produkten ck(x–x1) (x–x2)…(x–xk) kan ju skrivas cktk där tk räknas ut rekursivt i slingan som adderar termerna, t ex såhär (obs. vi lagrar ck i c(k+1)). Interpolationspunkterna ligger i X.

```
t = ones(size(x)); p = 0;
for k = 1:n+1
   p = p + c(k)*t;
   t = t.*(x-X(k));
end
```

Programmet refererar till X(n+1) (men den används aldrig) så därför låter vi X(n+1) ingå i datastrukturen.

Man kan spara ytterligare en multiplikation per varv genom att evaluera högstagradstermen först,

pn(x) = p0, där
pn = cn+1
pk–1 = ck-1 + (x–xk) pk, k = n,n–1,…,1

Derivatan kan beräknas samtidigt genom att man "deriverar" rekursionsformeln. Inför p'n(x) = q0, där qk är derivatan av mellanresultaten pk, k = n,n–1,…

pn = cn+1, qn = 0
pk–1 = ck–1 + (x–xk) pk,
qk–1 = pk + (x–xk) qk, k = n,n–1,…,1