Discrete Least-Squares Rational Approximation by Full-Newton Iteration

Carlos F. Borges

A Full Newton non-linear least-squares code for discrete least-squares rational approximation. This code implements the algorithm described in the paper:


All are welcome to use this code as they wish. I only ask that you cite the paper above if you do.

Usage:

\[ \text{[alpha]} = \text{dlsqrat}(t,y,p,q,\text{alpha}) \]

Inputs:

- \( t, y \) are the data points.
- \( p, q \) are the degrees of the numerator and denominator.
- \( \text{alpha} \) (optional) is the starting guess

Outputs:

- \( \text{alpha} \) contains the denominator coefficients starting with \( \text{alpha}_1 \)
- \( c \) contains the numerator coefficients starting with \( \text{c}_0 \)

Please note that the polynomial coefficients are generated in ascending order so if you want to use Matlab's polyval routine to evaluate things you need to flip the \( c \) vector, and you need to flip the \( \text{alpha} \) vector and then append a 1. Here is a code fragment you can use to view the results of the fit:

```matlab
cla;
pplot(t,y,'b.'); hold on
    tt = linspace(min(t),max(t),1000)';
    yy = polyval(flipud(c),tt)./polyval([flipud(alpha); 1],tt);
    plot(tt,yy); hold off;
```

Copyright (c) 2008 by Carlos F. Borges. All rights reserved.

Contents
function [alpha, c] = dlsqrat(t,y,p,q,alpha)
% begin dlsqrat

% Set the convergence tolerance.
TOLERANCE = 10^(-12);

% N is the Vandermonde that will be used to evaluate the numerator.
N = zeros(length(t),p+1);
N(:,1) = ones(length(t),1);
for k=2:p+1
    N(:,k) = N(:,k-1).*t;
end

% M is the Vandermonde that will be used to evaluate the denominator.
M = zeros(length(t),q);
M(:,1) = t;
for k=2:q
    M(:,k) = M(:,k-1).*t;
end

% If we are not given an initial guess then generate one.
if nargin < 5
    tmp_pade = [N -diag(y)*M] \ y;
    alpha = tmp_pade(p+2:end);
end

% Construct the model matrix and compute ancillary quantities.
update(alpha);

for iter=1:100

% Update the error.
old_err = err;

% Compute the Jacobian and the Hessian.
Tmp1 = diag(Py.*D)*M;
Tmp2 = Q'*diag((Py-r).*D)*M;
J = Tmp1 - Q'*Tmp2;
H = M'*diag((Py-2*r).*D)*Tmp1 - Tmp2'*Tmp2;

% Compute the gradient.
gradient = J'*r;

% Compute the Cholesky factorization of H.
[R, not_PD] = chol(H);
% If H is not positive definite then regularize and factor
if not_PD
    R = chol(H - 1.2*min(eig(H))*eye(q));
end

% Compute the Newton step.
delta = -R \ (R' \ gradient);
% Use stepsize control to take a step.
step_control;
% Convergence testing
if err > old_err
    disp('Failed to find descending step length.');
    break;
else
    alpha = new_alpha;
    rel_err = abs(old_err - err)/old_err;
    if rel_err <= TOLERANCE
        break;
    end
end
% End convergence testing.
end  %End of main loop.

% Compute the coefficients of the numerator.
c = (diag(D)*N)
y;

% Generate an error message if the algorithm failed to converge.
if rel_err > TOLERANCE
    disp('Algorithm did not converge.);
end

%XXXXXXXXXXXXXXXXX Subroutines
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
function update(alpha)
    % Updates the model matrix and computes ancillary quantities.
    D = 1./(1+M*alpha);         % Compute the denominator.
    [Q R] = qr(diag(D)*N,0);    % Compute the QR factorization of A =
    D*N
    Py = Q*(Q'*y);              % Compute the projection of y onto the
    range of A.
    r = y - Py;                 % Compute the residual.
    err = r'*r;                 % Compute the current squared error.
end

function step_control
    % This function implements stepsize control using a simple
    % backtracking scheme from Dennis & Schnabel.

    % Try taking a full step.
    new_alpha = alpha + delta;

    % Update the model.
    update(new_alpha);

    % If a full step does not sufficiently reduce the error then we
    % use a backtracking line-search method for step-size control.
    % This involves minimizing a function f(lambda) that interpolates
    % the
    % computed error (and its derivatives) at different values of
    % lambda.
    f0 = old_err;
    fprime = gradient'*delta;
    steptol = f0 + .0001*fprime;
    if err > steptol
errs(1) = err; lams(1) = 1; % We'll need this if further refinement is necessary.

% We start with a quadratic model at f(0), f'(0), and f(1)
% and will take the larger of the computed step or 1/10.
lambda = max([-fprime/(2*(err - f0 - fprime)) .1]);

new_alpha = alpha + lambda*delta;
% Update the model matrix and compute ancillary quantities.
update(new_alpha);

% If this doesn't work then we loop with a cubic model at
% f(0), f'(0), f(lambda), and f(lam2) where the last two are errors
% at
% the last two lambda that were tried.
steptol = f0 + .0001*fprime*lambda;
while err > steptol

% Push the current lambda and error to the top of the lams
and errs
% stacks.
lams = [lambda; lams(1)]; errs = [err; errs(1)];
rhs = (errs - fprime*lams - [f0 ; f0])./(lams.*lams);
ab = [lams [1 ; 1]]\rhs;
lambda = (-ab(2)+sqrt(ab(2)*ab(2) -
3*ab(1)*fprime))/(3*ab(1));

% It is still important to make certain that the new
% lambda
% progresses quickly but not too quickly. So if lambda is
less
% than lam2/10 we just use lam2/10, and if it is larger
% than lam2/2 then we use lam2/2.
if lambda < lams(1)/10
    lambda = lams(1)/10;
end
if lambda > lams(1)/2
    lambda = lams(1)/2;
end

new_alpha = alpha + lambda*delta;
% Update the model matrix and compute ancillary quantities.
update(new_alpha);
steptol = f0 + .0001*fprime*lambda;
end
end
end
Input argument "t" is undefined.

Error in ==> dlsqrat at 58
N = zeros(length(t),p+1);

% End of function.

References