Computational finance, take-home exam 2: theoretical part

Karl Albin Henriksson

October 2022

Exercise 4

By minimizing the root-mean-squared error of the difference between the observed prices and Heston model prices, the optimal Heston model parameters obtained were

$$\theta = 0.026065, \quad \kappa = 3.411271, \quad \sigma = 0.489289, \quad \rho = -0.603949, \quad V_0 = 0.018056.$$

Using these parameters, the RMSE was 1.205401. Since the maturities are given in days in the .mat-file, we needed to convert them into years. This can be be done more ways than one depending on what convention is used. In this example, we divided the maturities by 365. The code can be found below.

Code

```
clear all
close all

%% Load data, set parameters and initiate start guesses

data=cell2mat(struct2cell(load('Call_20050103.mat')));

call_prices=data(:,1);
strikes=data(:,2);
maturities=data(:,3)/365;

r=.015;
S=1202.10;

theta_init=.04;
kappa_init=1.5;
sigma_init=.3;
rho_init=-.6;
V_0_init=.0441;

%% Calibrate the Heston model by minimizing the RMSE of the difference between observed, and Heston prices.
```

```
LB = [-1, 0, 0, -1, 0];
UB = [1, 100, 10, 1, 10];
fun=@(x) sqrt(mean((Call_Heston_Vector(strikes, maturities, r, x(1), x(2), x(3), x(4), S, x(5))-call_prices).^2));
x_0=[theta_init kappa_init sigma_init rho_init V_0_init];
[x, fval]=fminsearchcon(fun,x_0,LB,UB);
fprintf(['Theta = %f\n','Kappa = %f\n','Sigma = %f\n'...
    ,'Rho = %f\n', 'V = %f\n\n'],x(1), x(2), x(3), x(4), x(5));
fprintf(['RMSE: %f'], fval);

%% Functions to be used

function vec=Call_Heston_Vector(strikes, maturities, r, theta, kappa, sigma, rho, S, V_0)
vec=zeros(length(strikes),1);
for i=1:length(strikes)
    vec(i,1)=Call_Heston(strikes(i), maturities(i), r, theta, kappa, sigma, rho, S, V_0);
end

end

function [x,fval,exitflag,output]=fminsearchcon(fun,x0,LB,UB,A,b,nonlcon,options,varargin)
% FMINSEARCHCON: Extension of FMINSEARCHBND with general inequality constraints
% usage: x=FMINSEARCHCON(fun,x0)
% usage: x=FMINSEARCHCON(fun,x0,LB)
% usage: x=FMINSEARCHCON(fun,x0,LB,UB)
% usage: x=FMINSEARCHCON(fun,x0,LB,UB,A,b)
% usage: x=FMINSEARCHCON(fun,x0,LB,UB,A,b,nonlcon)
% usage: x=FMINSEARCHCON(fun,x0,LB,UB,A,b,nonlcon,options)
% usage: x=FMINSEARCHCON(fun,x0,LB,UB,A,b,nonlcon,options,p1,p2,...)
% usage: [x,fval,exitflag,output]=FMINSEARCHCON(fun,x0,...)
%
% arguments:
%  fun, x0, options - see the help for FMINSEARCH
%
%       x0 MUST be a feasible point for the linear and nonlinear
%       inequality constraints. If it is not inside the bounds
%       then it will be moved to the nearest bound. If x0 is
%       infeasible for the general constraints, then an error will
%       be returned.
%
%  LB - lower bound vector or array, must be the same size as x0
%
%       If no lower bounds exist for one of the variables, then
%       supply -inf for that variable.
```

```
%
%        If no lower bounds at all, then LB may be left empty.
%
%        Variables may be fixed in value by setting the corresponding
%        lower and upper bounds to exactly the same value.
%
%  UB - upper bound vector or array, must be the same size as x0
%
%        If no upper bounds exist for one of the variables, then
%        supply +inf for that variable.
%
%        If no upper bounds at all, then UB may be left empty.
%
%        Variables may be fixed in value by setting the corresponding
%        lower and upper bounds to exactly the same value.
%
%  A,b - (OPTIONAL) Linear inequality constraint array and right
%        hand side vector. (Note: these constraints were chosen to
%        be consistent with those of fmincon.)
%
%        A*x <= b
%
%  nonlcon - (OPTIONAL) general nonlinear inequality constraints
%        NONLCON must return a set of general inequality constraints.
%        These will be enforced such that nonlcon is always <= 0.
%
%        nonlcon(x) <= 0
%
%
% Notes:
%
%  If options is supplied, then TolX will apply to the transformed
%  variables. All other FMINSEARCH parameters should be unaffected.
%
%  Variables which are constrained by both a lower and an upper
%  bound will use a sin transformation. Those constrained by
%  only a lower or an upper bound will use a quadratic
%  transformation, and unconstrained variables will be left alone.
%
%  Variables may be fixed by setting their respective bounds equal.
%  In this case, the problem will be reduced in size for FMINSEARCH.
%
%  The bounds are inclusive inequalities, which admit the
%  boundary values themselves, but will not permit ANY function
%  evaluations outside the bounds. These constraints are strictly
%  followed.
%
%  If your problem has an EXCLUSIVE (strict) constraint which will
%  not admit evaluation at the bound itself, then you must provide
```

```
%  a slightly offset bound. An example of this is a function which
%  contains the log of one of its parameters. If you constrain the
%  variable to have a lower bound of zero, then FMINSEARCHCON may
%  try to evaluate the function exactly at zero.
%
%  Inequality constraints are enforced with an implicit penalty
%  function approach. But the constraints are tested before
%  any function evaluations are ever done, so the actual objective
%  function is NEVER evaluated outside of the feasible region.
%
%
% Example usage:
% rosen = @(x) (1-x(1)).^2 + 105*(x(2)-x(1).^2).^2;
%
% Fully unconstrained problem
% fminsearchcon(rosen,[3 3])
% ans =
%    1.0000    1.0000
%
% lower bound constrained
% fminsearchcon(rosen,[3 3],[2 2],[])
% ans =
%    2.0000    4.0000
%
% x(2) fixed at 3
% fminsearchcon(rosen,[3 3],[-inf 3],[inf,3])
% ans =
%    1.7314    3.0000
%
% simple linear inequality: x(1) + x(2) <= 1
% fminsearchcon(rosen,[0 0],[],[],[1 1],.5)
%
% ans =
%    0.6187    0.3813
%
% general nonlinear inequality: sqrt(x(1)^2 + x(2)^2) <= 1
% fminsearchcon(rosen,[3 3],[],[],[],[],@(x) norm(x)-1)
% ans =
%    0.78633   0.61778
%
% Of course, any combination of the above constraints is
% also possible.
%
% See test_main.m for other examples of use.
%
%
% See also: fminsearch, fminspleas, fminsearchbnd
%
%
```

```
% Author: John D'Errico
% E-mail: woodchips@rochester.rr.com
% Release: 1.0
% Release date: 12/16/06

% size checks
xsize = size(x0);
x0 = x0(:);
n=length(x0);

if (nargin<3) || isempty(LB)
  LB = repmat(-inf,n,1);
else
  LB = LB(:);
end
if (nargin<4) || isempty(UB)
  UB = repmat(inf,n,1);
else
  UB = UB(:);
end

if (n~=length(LB)) || (n~=length(UB))
  error 'x0 is incompatible in size with either LB or UB.'
end

% defaults for A,b
if (nargin<5) || isempty(A)
  A = [];
end
if (nargin<6) || isempty(b)
  b = [];
end
nA = [];
nb = [];
if (isempty(A)&&~isempty(b)) || (isempty(b)&&~isempty(A))
  error 'Sizes of A and b are incompatible'
elseif ~isempty(A)
  nA = size(A);
  b = b(:);
  nb = size(b,1);
  if nA(1)~=nb
    error 'Sizes of A and b are incompatible'
  end
  if nA(2)~=n
    error 'A is incompatible in size with x0'
  end
end

% defaults for nonlcon
```

```
if (nargin<7) || isempty(nonlcon)
  nonlcon = [];
end

% test for feasibility of the initial value
% against any general inequality constraints
if ~isempty(A)
  if any(A*x0>b)
    error 'Infeasible starting values (linear inequalities failed).'
  end
end
if ~isempty(nonlcon)
  if any(feval(nonlcon,(reshape(x0,xsize)),varargin{:})>0)
    error 'Infeasible starting values (nonlinear inequalities failed)
        .'
  end
end

% set default options if necessary
if (nargin<8) || isempty(options)
  options = optimset('fminsearch');
end

% stuff into a struct to pass around
params.args = varargin;
params.LB = LB;
params.UB = UB;
params.fun = fun;
params.n = n;
params.xsize = xsize;

params.OutputFcn = [];

params.A = A;
params.b = b;
params.nonlcon = nonlcon;

% 0 --> unconstrained variable
% 1 --> lower bound only
% 2 --> upper bound only
% 3 --> dual finite bounds
% 4 --> fixed variable
params.BoundClass = zeros(n,1);
for i=1:n
  k = isfinite(LB(i)) + 2*isfinite(UB(i));
  params.BoundClass(i) = k;
  if (k==3) && (LB(i)==UB(i))
    params.BoundClass(i) = 4;
  end
```

```
end

% transform starting values into their unconstrained
% surrogates. Check for infeasible starting guesses.
x0u = x0;
k=1;
for i = 1:n
  switch params.BoundClass(i)
    case 1
      % lower bound only
      if x0(i)<=LB(i)
        % infeasible starting value. Use bound.
        x0u(k) = 0;
      else
        x0u(k) = sqrt(x0(i) - LB(i));
      end

      % increment k
      k=k+1;
    case 2
      % upper bound only
      if x0(i)>=UB(i)
        % infeasible starting value. use bound.
        x0u(k) = 0;
      else
        x0u(k) = sqrt(UB(i) - x0(i));
      end

      % increment k
      k=k+1;
    case 3
      % lower and upper bounds
      if x0(i)<=LB(i)
        % infeasible starting value
        x0u(k) = -pi/2;
      elseif x0(i)>=UB(i)
        % infeasible starting value
        x0u(k) = pi/2;
      else
        x0u(k) = 2*(x0(i) - LB(i))/(UB(i)-LB(i)) - 1;
        % shift by 2*pi to avoid problems at zero in fminsearch
        % otherwise, the initial simplex is vanishingly small
        x0u(k) = 2*pi+asin(max(-1,min(1,x0u(k))));
      end

      % increment k
      k=k+1;
    case 0
      % unconstrained variable. x0u(i) is set.
```

```
      x0u(k) = x0(i);

      % increment k
      k=k+1;
    case 4
      % fixed variable. drop it before fminsearch sees it.
      % k is not incremented for this variable.
  end

end
% if any of the unknowns were fixed, then we need to shorten
% x0u now.
if k<=n
  x0u(k:n) = [];
end

% were all the variables fixed?
if isempty(x0u)
  % All variables were fixed. quit immediately, setting the
  % appropriate parameters, then return.

  % undo the variable transformations into the original space
  x = xtransform(x0u,params);

  % final reshape
  x = reshape(x,xsize);

  % stuff fval with the final value
  fval = feval(params.fun,x,params.args{:});

  % fminsearchbnd was not called
  exitflag = 0;

  output.iterations = 0;
  output.funcount = 1;
  output.algorithm = 'fminsearch';
  output.message = 'All variables were held fixed by the applied bounds';

  % return with no call at all to fminsearch
  return
end

% Check for an outputfcn. If there is any, then substitute my
% own wrapper function.
if ~isempty(options.OutputFcn)
  params.OutputFcn = options.OutputFcn;
  options.OutputFcn = @outfun_wrapper;
end
```

```matlab

% now we can call fminsearch, but with our own
% intra-objective function.
[xu,fval,exitflag,output] = fminsearch(@intrafun,x0u,options,params);

% undo the variable transformations into the original space
x = xtransform(xu,params);

% final reshape
x = reshape(x,xsize);

% Use a nested function as the OutputFcn wrapper
  function stop = outfun_wrapper(x,varargin);
    % we need to transform x first
    xtrans = xtransform(x,params);

    % then call the user supplied OutputFcn
    stop = params.OutputFcn(xtrans,varargin{1:(end-1)});

  end

end % mainline end

% ======================================
% ========= begin subfunctions =========
% ======================================
function fval = intrafun(x,params)
% transform variables, test constraints, then call original function

% transform
xtrans = xtransform(x,params);

% test constraints before the function call

% First, do the linear inequality constraints, if any
if ~isempty(params.A)
  % Required: A*xtrans <= b
  if any(params.A*xtrans(:) > params.b)
    % linear inequality constraints failed. Just return inf.
    fval = inf;
    return
  end
end

% resize xtrans to be the correct size for the nonlcon
% and objective function calls
xtrans = reshape(xtrans,params.xsize);

% Next, do the nonlinear inequality constraints
```

```
if ~isempty(params.nonlcon)
  % Required: nonlcon(xtrans) <= 0
  cons = feval(params.nonlcon,xtrans,params.args{:});
  if any(cons(:) > 0)
    % nonlinear inequality constraints failed. Just return inf.
    fval = inf;
    return
  end
end

% we survived the general inequality constraints. Only now
% do we evaluate the objective function.

% append any additional parameters to the argument list
fval = feval(params.fun,xtrans,params.args{:});

end % sub function intrafun end

% ============================================
function xtrans = xtransform(x,params)
% converts unconstrained variables into their original domains

xtrans = zeros(1,params.n);
% k allows some variables to be fixed, thus dropped from the
% optimization.
k=1;
for i = 1:params.n
  switch params.BoundClass(i)
    case 1
      % lower bound only
      xtrans(i) = params.LB(i) + x(k).^2;

      k=k+1;
    case 2
      % upper bound only
      xtrans(i) = params.UB(i) - x(k).^2;

      k=k+1;
    case 3
      % lower and upper bounds
      xtrans(i) = (sin(x(k))+1)/2;
      xtrans(i) = xtrans(i)*(params.UB(i) - params.LB(i)) + params.LB
        (i);
      % just in case of any floating point problems
      xtrans(i) = max(params.LB(i),min(params.UB(i),xtrans(i)));

      k=k+1;
    case 4
      % fixed variable, bounds are equal, set it at either bound
```

```
        xtrans(i) = params.LB(i);
      case 0
        % unconstrained variable.
        xtrans(i) = x(k);

        k=k+1;
  end
end

end % sub function xtransform end







function P = Call_Heston(K, T, r, nu, kappa, sigma, rho, S, V)
% Call_Heston:  Compute the value of call option using the formula in
% Heston[1993], see also formula (6) in Albrecher et Al.[2006].
%
% USAGE: P = Call_Heston(K, T, r, nu, kappa, sigma, rho, S, V, modif);
%
% PARAMETERS:
%     Input:
%          K: strike price of the call option
%          T: maturity of the call option
%          r: risk free rate
%          nu, kappa, sigma: parameters of the Heston model
%          rho: correlation parameter between the stock and vol processes
%          S, V: initial stock price and volatility
%     Output:
%         P: price of the call option.

b1 = kappa-rho*sigma;
b2 = kappa;
u1 = 0.5;
u2 = -0.5;

x = log(S);
alpha = log(K); %log-strike

integrand = @(u) S *...
              real(exp(-1i*u*alpha) .* exp(C_CF(u, T, r, nu, kappa, sigma, rho, b1, u1)...
              + V * D_CF(u, T, sigma, rho, b1, u1) + 1i*u*x) ./ (1i*u))...
```

```
            - K * exp(-r*T) ...
            * real(exp(-1i*u*alpha) .* exp(C_CF(u, T, r, nu, kappa, sigma, rho, b2, u2) ...
            + V * D_CF(u, T, sigma, rho, b2, u2) + 1i*u*x) ./ (1i*u));

P = 0.5 * (S - K * exp(-r*T)) + 1/pi * quadgk(integrand, 0, 100);

end


function out = C_CF(u, t, r, nu, kappa, sigma, rho, bj, uj)

d = sqrt( (rho*sigma*u.*1i - bj).^2 - sigma^2 * (2*uj*u.*1i - u.^2) );
g = (bj - rho*sigma*u.*1i + d) ./ (bj - rho*sigma*u.*1i - d);

out = r*u*t.*1i + (nu * kappa) ./ sigma^2 * ...
    ( (bj - rho*sigma*u*1i + d) * t - 2*log( (1 - g.*exp(d*t)) ./ (1 - g)) );

end

function out = D_CF(u, t, sigma, rho, bj, uj)

d = sqrt( (rho*sigma*u.*1i - bj).^2 - sigma^2 * (2*uj*u.*1i - u.^2) );
g = (bj - rho*sigma*u.*1i + d) ./ (bj - rho*sigma*u.*1i - d);

out = (bj - rho*sigma*u*1i + d) ./ (sigma^2) .* ((1 - exp(d*t)) ./ (1 - g.*exp(d*t)));

end
```