Homework I, ECON 8210, Fall 2024

Author: Jiacheng Li

Created: Oct. 19
Updated: Oct. 29

1. Github

Please see the link: https://github.com/realjiachengli/ECON_8210_repository

2. Integration

The goal is to compute the integral:

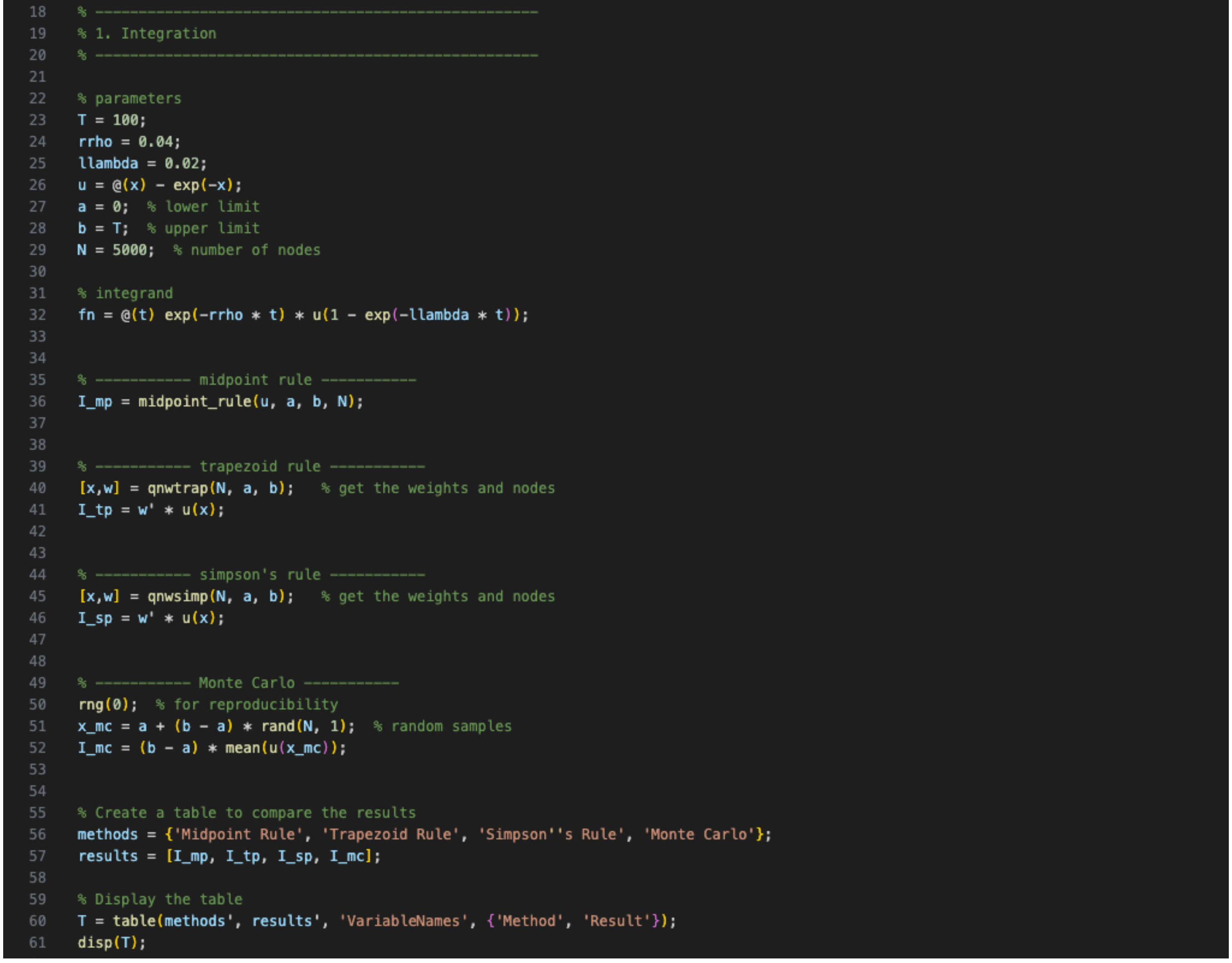
$$\int_0^T e^{-\rho t} u\left(1 - e^{-\lambda t}\right) dt$$

with $T = 100, \rho = 0.04, \lambda = 0.02, u(c) = -e^c$ using the following methods:

- Midpoint quadrature
- Trapezoid quadrature
- Simpson rule
- Monte Carlo method

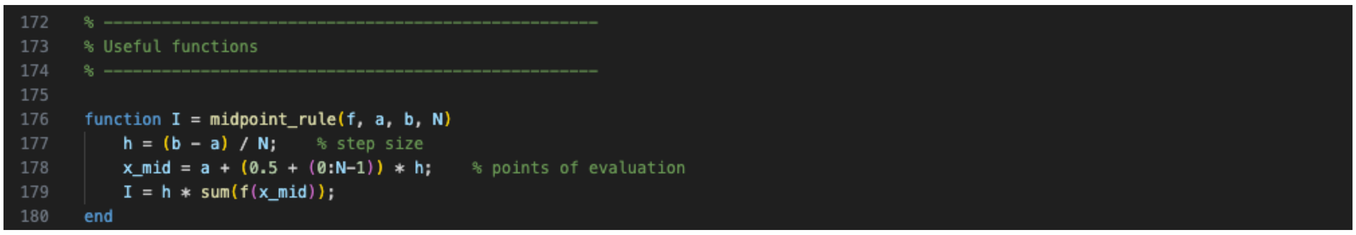
I implement each of the methods in Matlab and report the codes and results below. For Trapezoid and Simpson rule quadratures, I used the `compecon` toolbox of Miranda and Fackler (2002).

Codes



```
% ---------------------------------------------------
% 1. Integration
% ---------------------------------------------------

% parameters
T = 100;
rrho = 0.04;
llambda = 0.02;
u = @(x) - exp(-x);
a = 0;  % lower limit
b = T;  % upper limit
N = 5000;  % number of nodes

% integrand
fn = @(t) exp(-rrho * t) * u(1 - exp(-llambda * t));


% ----------- midpoint rule -----------
I_mp = midpoint_rule(u, a, b, N);


% ----------- trapezoid rule -----------
[x,w] = qnwtrap(N, a, b);   % get the weights and nodes
I_tp = w' * u(x);


% ----------- simpson's rule -----------
[x,w] = qnwsimp(N, a, b);   % get the weights and nodes
I_sp = w' * u(x);


% ----------- Monte Carlo -----------
rng(0);  % for reproducibility
x_mc = a + (b - a) * rand(N, 1);  % random samples
I_mc = (b - a) * mean(u(x_mc));


% Create a table to compare the results
methods = {'Midpoint Rule', 'Trapezoid Rule', 'Simpson''s Rule', 'Monte Carlo'};
results = [I_mp, I_tp, I_sp, I_mc];

% Display the table
T = table(methods', results', 'VariableNames', {'Method', 'Result'});
disp(T);
```

The midpoint method is implemented by the simple function:



```
% ---------------------------------------------------
% Useful functions
% ---------------------------------------------------

function I = midpoint_rule(f, a, b, N)
    h = (b - a) / N;     % step size
    x_mid = a + (0.5 + (0:N-1)) * h;     % points of evaluation
    I = h * sum(f(x_mid));
end
```

Results:

```
          Method            Result 
    __________________    ________

    {'Midpoint Rule' }    -0.99998
    {'Trapezoid Rule'}          -1
    {'Simpson's Rule'}          -1
    {'Monte Carlo'   }    -0.96001
```

The Monte Carlo integration uses 5000 random draws from the uniform. It is clear that this is the least accurate, while Midpoint Rule is the second to last worst performed. With sufficiently high number of nodes, the quadrature methods yield good performance for this simple problem.

3. Optimization: basic problem

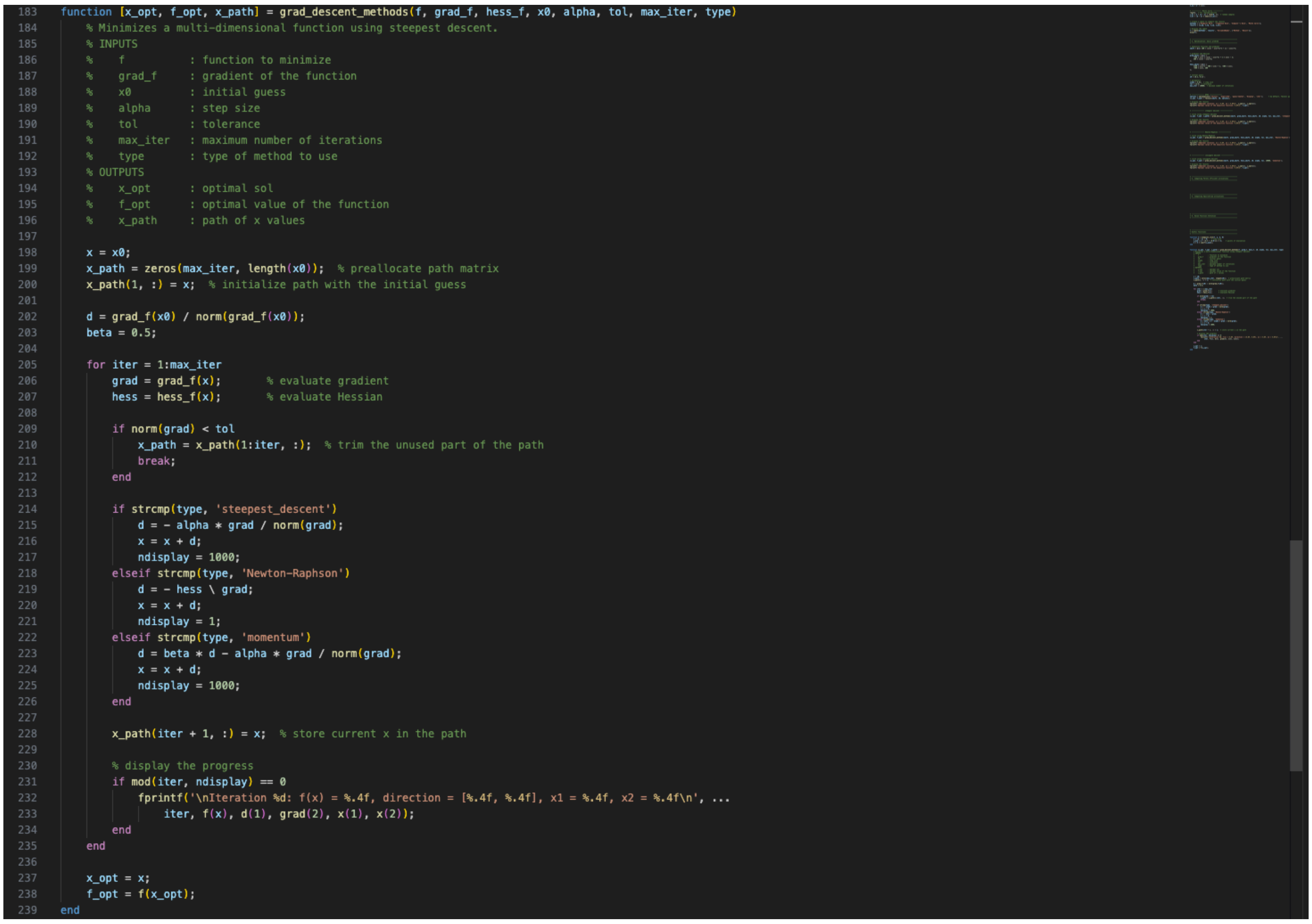
This exercise involves minimizing the classic Rosenbrock function:

$$\min_{x,y} 100\left(y - x^2\right)^2 + (1 - x)^2$$

using various direction-based methods, including:

- Newton-Raphson,
- BFGS (quasi-Newton),
- steepest descent,
- conjugate descent method (I used the momentum update method, where the update rule of direction follows: $v^{(k+1)} = \beta v^{(k)} - \alpha g^{(k)}$.)

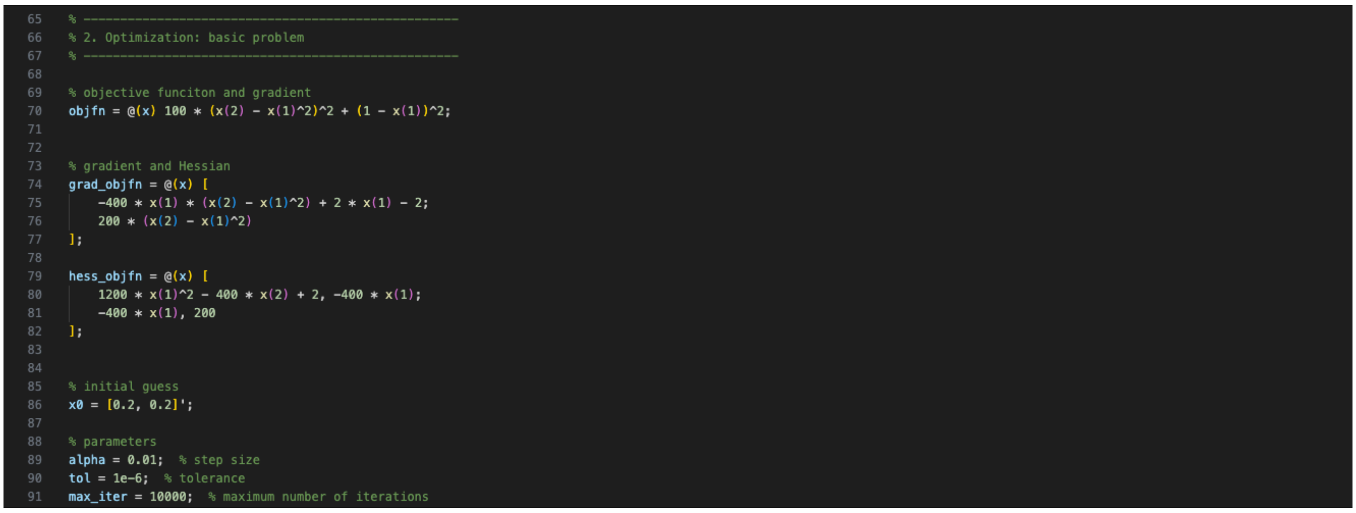
In the codes, I nest these methods in a single function:



```matlab
function [x_opt, f_opt, x_path] = grad_descent_methods(f, grad_f, hess_f, x0, alpha, tol, max_iter, type)
    % Minimizes a multi-dimensional function using steepest descent.
    % INPUTS
    %   f         : function to minimize
    %   grad_f    : gradient of the function
    %   x0        : initial guess
    %   alpha     : step size
    %   tol       : tolerance
    %   max_iter  : maximum number of iterations
    %   type      : type of method to use
    % OUTPUTS
    %   x_opt     : optimal sol
    %   f_opt     : optimal value of the function
    %   x_path    : path of x values

    x = x0;
    x_path = zeros(max_iter, length(x0));  % preallocate path matrix
    x_path(1, :) = x;  % initialize path with the initial guess

    d = grad_f(x0) / norm(grad_f(x0));
    beta = 0.5;

    for iter = 1:max_iter
        grad = grad_f(x);       % evaluate gradient
        hess = hess_f(x);       % evaluate Hessian

        if norm(grad) < tol
            x_path = x_path(1:iter, :);  % trim the unused part of the path
            break;
        end

        if strcmp(type, 'steepest_descent')
            d = - alpha * grad / norm(grad);
            x = x + d;
            ndisplay = 1000;
        elseif strcmp(type, 'Newton-Raphson')
            d = - hess \ grad;
            x = x + d;
            ndisplay = 1;
        elseif strcmp(type, 'momentum')
            d = beta * d - alpha * grad / norm(grad);
            x = x + d;
            ndisplay = 1000;
        end

        x_path(iter + 1, :) = x;  % store current x in the path

        % display the progress
        if mod(iter, ndisplay) == 0
            fprintf('\nIteration %d: f(x) = %.4f, direction = [%.4f, %.4f], x1 = %.4f, x2 = %.4f\n', ...
                iter, f(x), d(1), grad(2), x(1), x(2));
        end
    end

    x_opt = x;
    f_opt = f(x_opt);
end
```

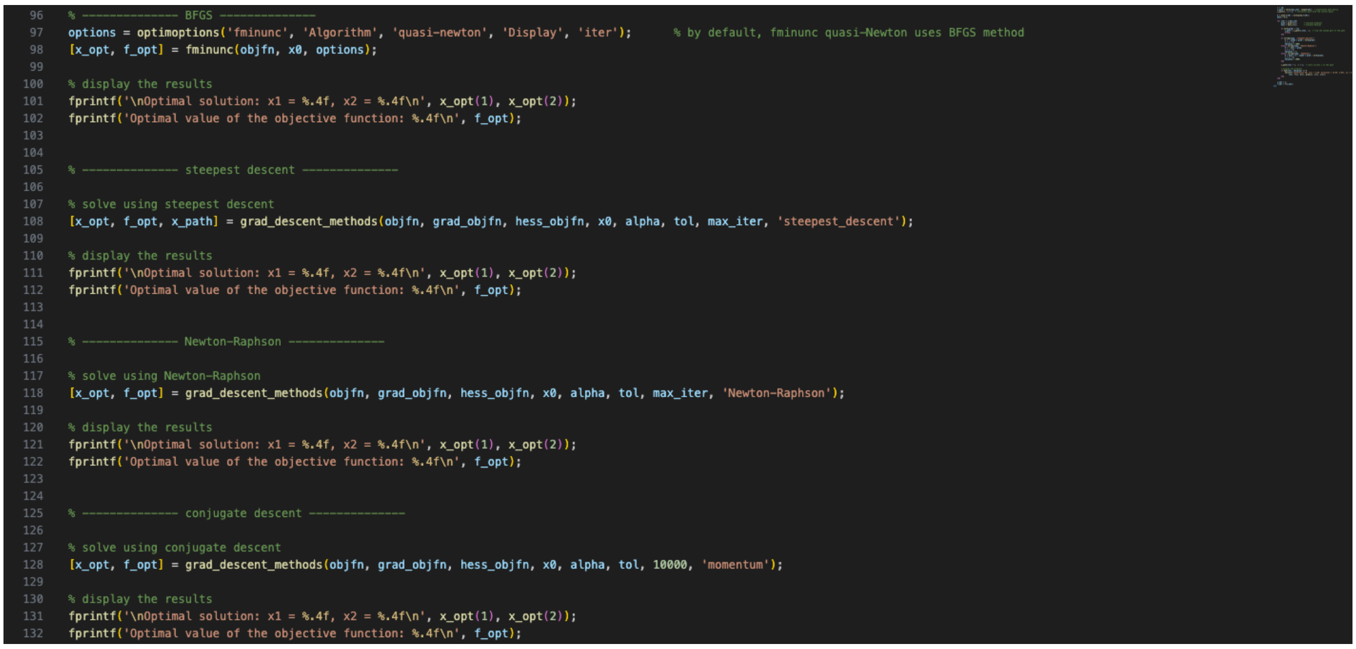
For BFGS, I use the built-in Matlab function `fminunc`, which implements BFGS as its default algorithm for quasi-Newton method.

The analytical gradient and Hessian are provided for steepest descent and Newton-Raphson.



```matlab
% -------------------------------------------------
% 2. Optimization: basic problem
% -------------------------------------------------

% objective funciton and gradient
objfn = @(x) 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;


% gradient and Hessian
grad_objfn = @(x) [
    -400 * x(1) * (x(2) - x(1)^2) + 2 * x(1) - 2;
    200 * (x(2) - x(1)^2)
];

hess_objfn = @(x) [
    1200 * x(1)^2 - 400 * x(2) + 2, -400 * x(1);
    -400 * x(1), 200
];


% initial guess
x0 = [0.2, 0.2]';

% parameters
alpha = 0.01;  % step size
tol = 1e-6;  % tolerance
max_iter = 10000;  % maximum number of iterations
```

Below are the implementations:



```
% ------------- BFGS -------------
options = optimoptions('fminunc', 'Algorithm', 'quasi-newton', 'Display', 'iter');     % by default, fminunc quasi-Newton uses BFGS method
[x_opt, f_opt] = fminunc(objfn, x0, options);

% display the results
fprintf('\nOptimal solution: x1 = %.4f, x2 = %.4f\n', x_opt(1), x_opt(2));
fprintf('Optimal value of the objective function: %.4f\n', f_opt);


% ------------- steepest descent -------------

% solve using steepest descent
[x_opt, f_opt, x_path] = grad_descent_methods(objfn, grad_objfn, hess_objfn, x0, alpha, tol, max_iter, 'steepest_descent');

% display the results
fprintf('\nOptimal solution: x1 = %.4f, x2 = %.4f\n', x_opt(1), x_opt(2));
fprintf('Optimal value of the objective function: %.4f\n', f_opt);


% ------------- Newton-Raphson -------------

% solve using Newton-Raphson
[x_opt, f_opt] = grad_descent_methods(objfn, grad_objfn, hess_objfn, x0, alpha, tol, max_iter, 'Newton-Raphson');

% display the results
fprintf('\nOptimal solution: x1 = %.4f, x2 = %.4f\n', x_opt(1), x_opt(2));
fprintf('Optimal value of the objective function: %.4f\n', f_opt);


% ------------- conjugate descent -------------

% solve using conjugate descent
[x_opt, f_opt] = grad_descent_methods(objfn, grad_objfn, hess_objfn, x0, alpha, tol, 10000, 'momentum');

% display the results
fprintf('\nOptimal solution: x1 = %.4f, x2 = %.4f\n', x_opt(1), x_opt(2));
fprintf('Optimal value of the objective function: %.4f\n', f_opt);
```

Results:

1. **BFGS**: as expected, BFGS performs pretty well even though it does not directly require analytical Hessian. It reaches the minimum $(1, 1)$ in only 20 iterations.

```
                                                        First-order 
 Iteration  Func-count       f(x)        Step-size       optimality
     0           3              3.2                           32
     1           9         0.548186      0.00428322         1.11
     2          12         0.540531               1         1.14
     3          15         0.469193               1         2.21
     4          21         0.287326        0.642908         2.81
     5          27         0.276637             0.5         4.05
     6          30         0.230649               1         3.24
     7          33           0.1434               1         1.58
     8          36         0.106483               1         4.39
     9          39        0.0425319               1        0.888
    10          45        0.0290431        0.399728         2.19
    11          48        0.0199143               1         1.98
    12          51       0.00731757               1        0.376
    13          57       0.00363958         0.44748         1.21
    14          60       0.00158309               1        0.813
    15          63      0.000272707               1       0.0172
    16          66      0.000148319               1        0.482
    17          69      4.37952e-06               1      0.00521
    18          72      2.20348e-07               1      0.00129
    19          75      1.72826e-10               1      0.00047
                                                        First-order 
 Iteration  Func-count       f(x)        Step-size       optimality
    20          78      2.04383e-11               1     1.08e-05

Local minimum found.
```

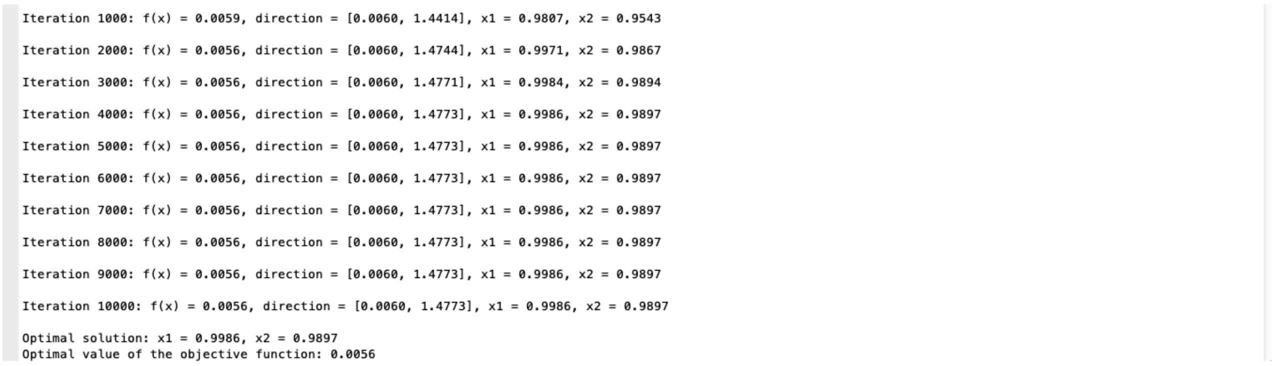
2. **Steepest descent**: no surprise that it performs quite slow. After 10000 iterations, it is close to but still hasn't reached the minimum (I set $\alpha = 0.01$ rather than searching for the optimal step size):

```
Iteration 1000: f(x) = 0.0213, direction = [-0.0086, -2.1341], x1 = 0.8927, x2 = 0.8067

Iteration 2000: f(x) = 0.0137, direction = [-0.0088, -2.1870], x1 = 0.9516, x2 = 0.9163

Iteration 3000: f(x) = 0.0127, direction = [-0.0089, -2.2061], x1 = 0.9717, x2 = 0.9552

Iteration 4000: f(x) = 0.0125, direction = [-0.0089, -2.2138], x1 = 0.9797, x2 = 0.9709

Iteration 5000: f(x) = 0.0125, direction = [-0.0089, -2.2171], x1 = 0.9831, x2 = 0.9775

Iteration 6000: f(x) = 0.0125, direction = [-0.0089, -2.2185], x1 = 0.9845, x2 = 0.9804

Iteration 7000: f(x) = 0.0125, direction = [-0.0089, -2.2191], x1 = 0.9852, x2 = 0.9816

Iteration 8000: f(x) = 0.0125, direction = [-0.0089, -2.2194], x1 = 0.9854, x2 = 0.9821

Iteration 9000: f(x) = 0.0125, direction = [-0.0089, -2.2195], x1 = 0.9855, x2 = 0.9824

Iteration 10000: f(x) = 0.0125, direction = [-0.0089, -2.2196], x1 = 0.9856, x2 = 0.9825

Optimal solution: x1 = 0.9856, x2 = 0.9825
Optimal value of the objective function: 0.0125
```

3. **Newton-Raphson**: this works the fastest, converging to the true minimum within 6 iterations.

```
Iteration 1: f(x) = 0.6820, direction = [-0.0258, 32.0000], x1 = 0.1742, x2 = 0.0297

Iteration 2: f(x) = 28.2124, direction = [0.7287, -0.1332], x1 = 0.9029, x2 = 0.2842

Iteration 3: f(x) = 0.0092, direction = [0.0009, -106.2130], x1 = 0.9038, x2 = 0.8169

Iteration 4: f(x) = 0.0085, direction = [0.0961, -0.0002], x1 = 1.0000, x2 = 0.9907

Iteration 5: f(x) = 0.0000, direction = [0.0000, -1.8487], x1 = 1.0000, x2 = 1.0000

Iteration 6: f(x) = 0.0000, direction = [0.0000, -0.0000], x1 = 1.0000, x2 = 1.0000

Optimal solution: x1 = 1.0000, x2 = 1.0000
Optimal value of the objective function: 0.0000
```

4. **Conjugate descent method** (momentum $\beta = 0.5, \alpha = 0.01$): there is some improvement relative to the steepest descent but not too much. Some distance remains after 10000 iterations, although it performs better than steepest descent.



```
Iteration 1000: f(x) = 0.0059, direction = [0.0060, 1.4414], x1 = 0.9807, x2 = 0.9543

Iteration 2000: f(x) = 0.0056, direction = [0.0060, 1.4744], x1 = 0.9971, x2 = 0.9867

Iteration 3000: f(x) = 0.0056, direction = [0.0060, 1.4771], x1 = 0.9984, x2 = 0.9894

Iteration 4000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Iteration 5000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Iteration 6000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Iteration 7000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Iteration 8000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Iteration 9000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Iteration 10000: f(x) = 0.0056, direction = [0.0060, 1.4773], x1 = 0.9986, x2 = 0.9897

Optimal solution: x1 = 0.9986, x2 = 0.9897
Optimal value of the objective function: 0.0056
```

4. Computing Pareto efficient allocatioins

Given endowment $e^i_j, i = 1, ..., m$ and $j = 1, ..., n$, the social planner solves

$$\begin{aligned}\max_{\{x^i_j\}_{\forall i, \forall j}} & \sum_{i=1}^{n} \lambda_i \sum_{j=1}^{m} \alpha_j \frac{(x^i_j)^{1+\omega^i_j}}{1+\omega^i_j} \\ \text{s.t.} & \sum_{i=1}^{n} x^i_j = \sum_{i=1}^{n} e^i_j \equiv \bar{e}_j \quad \forall j = 1, ..., m\end{aligned}$$

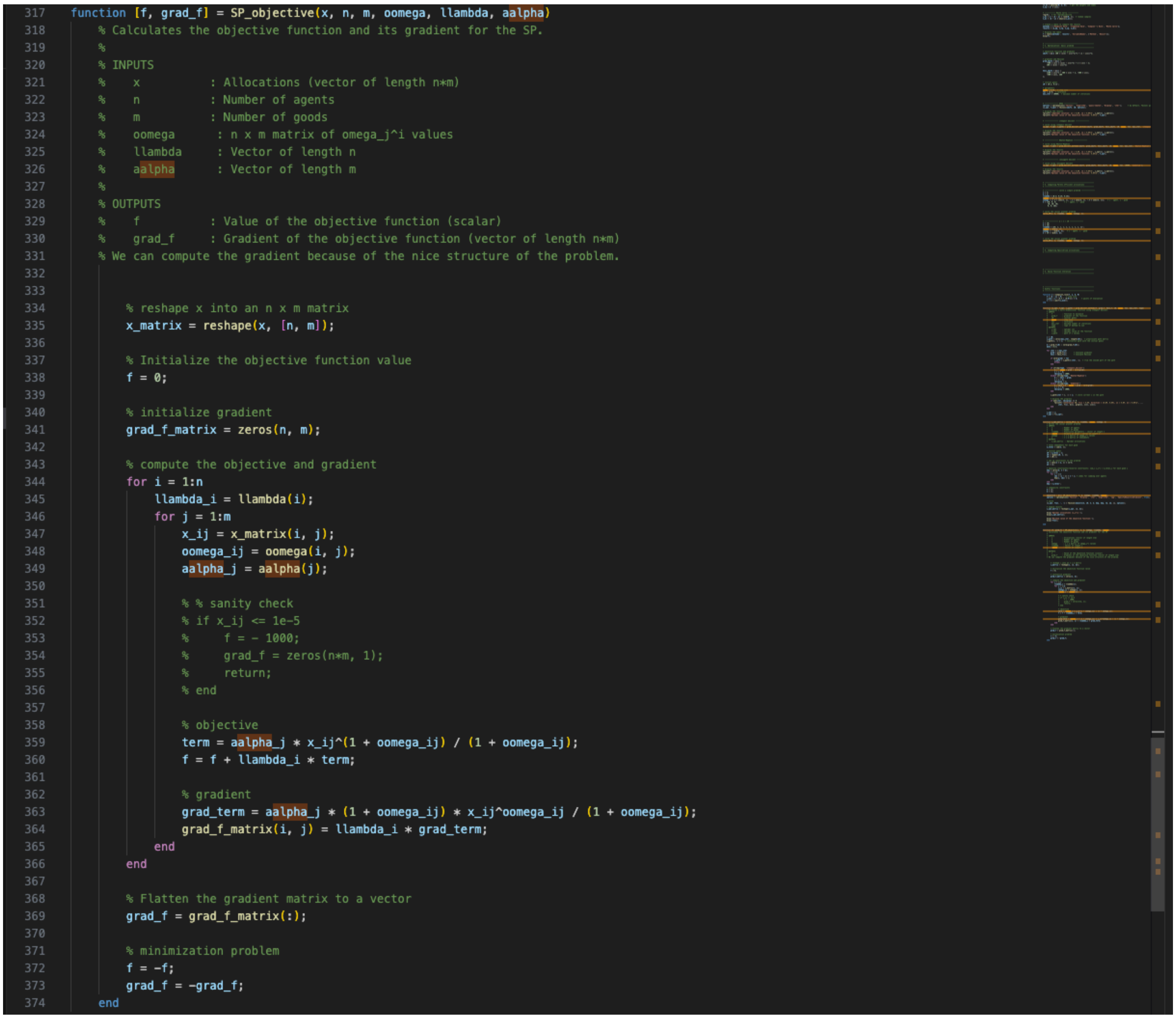
I solve this problem directly as an optimization problem in Matlab, using `fmincon`. In particular, notice that the individual-good specific nature of the problem allows us to **easily compute the gradient of the objective function analytically**. Notice that

$$f'_{i,j} = \lambda_i \alpha_j \frac{(x^i_j)^{\omega^i_j}}{\omega^i_j}$$

Thus, we can directly compute and feed the gradient at each evaluation to `fmincon`, which greatly accelerate the computation.

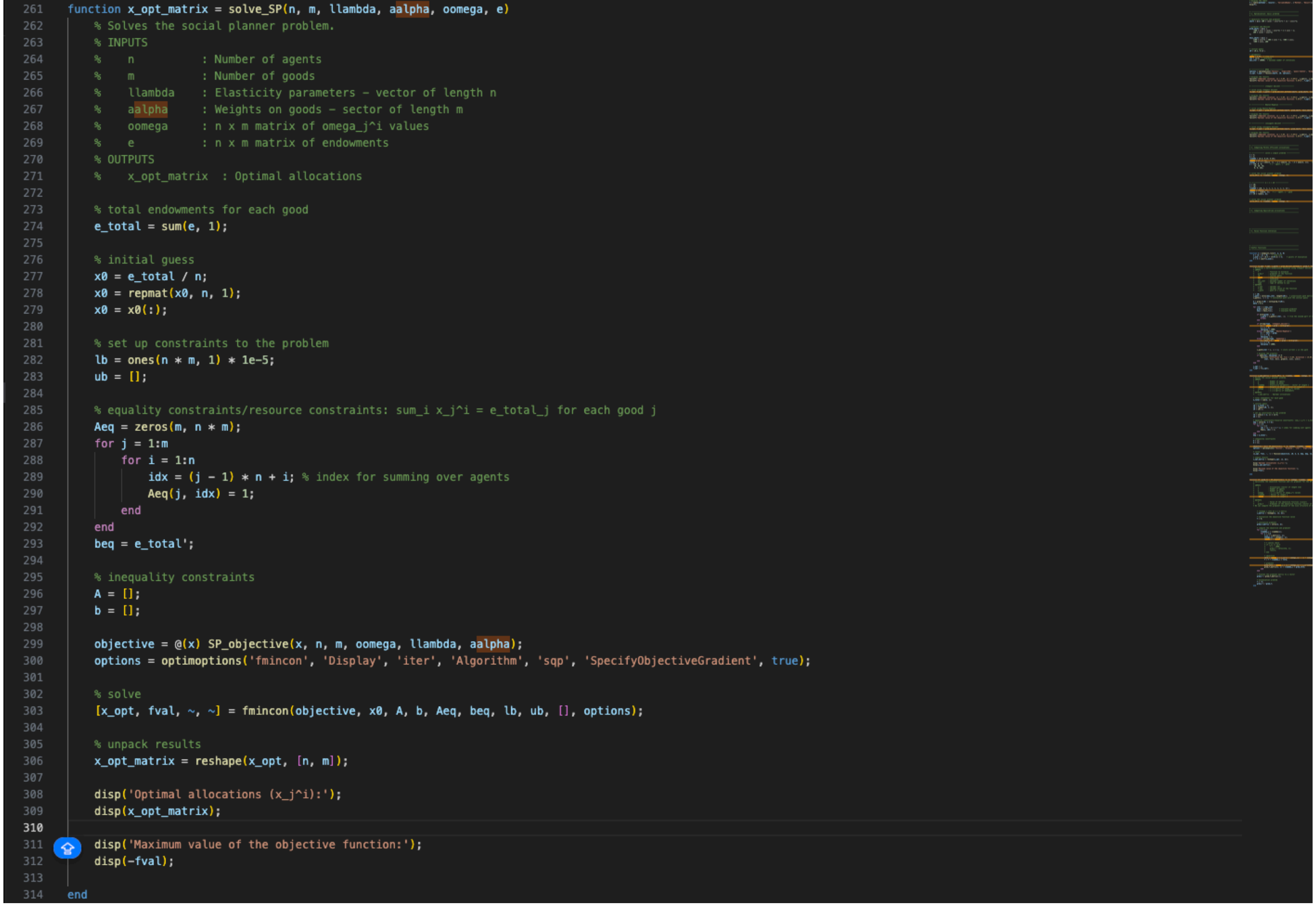
Codes

The objective function is evaluated in the following function:



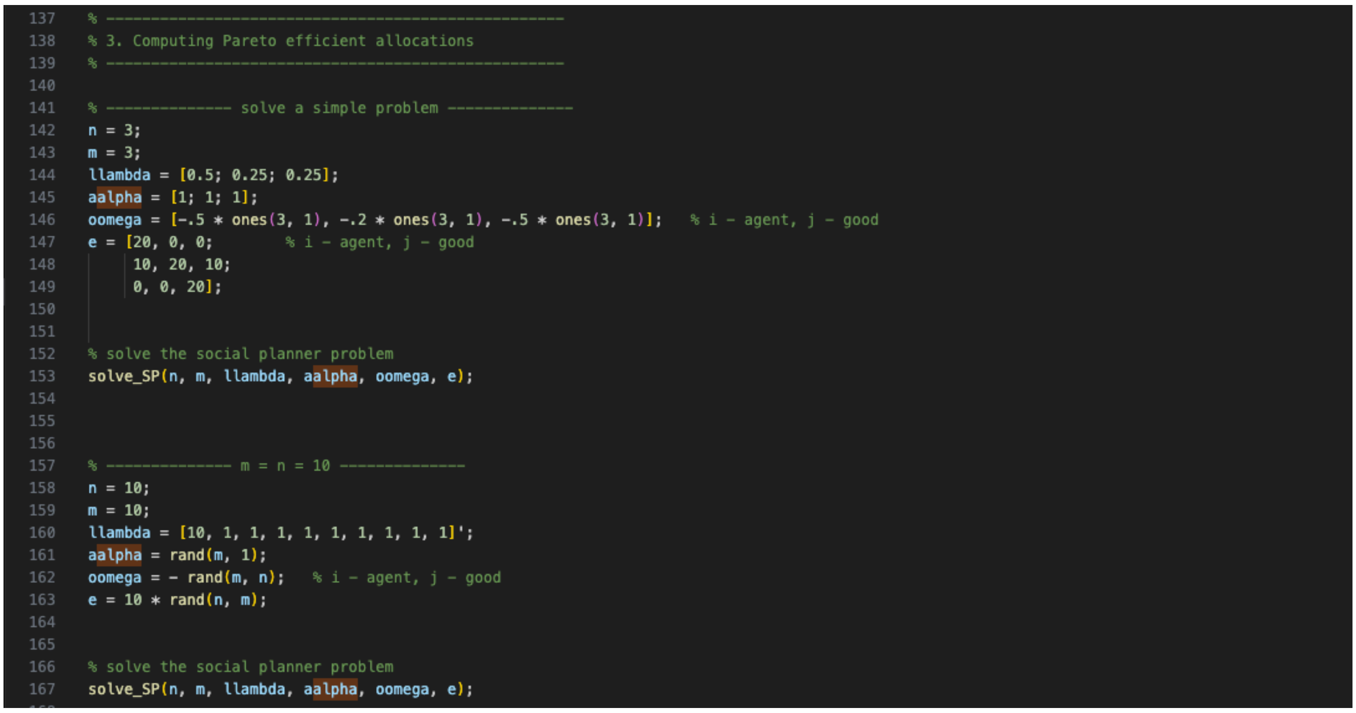
```
function [f, grad_f] = SP_objective(x, n, m, oomega, llambda, aalpha)
    % Calculates the objective function and its gradient for the SP.
    %
    % INPUTS
    %    x          : Allocations (vector of length n*m)
    %    n          : Number of agents
    %    m          : Number of goods
    %    oomega      : n x m matrix of omega_j^i values
    %    llambda     : Vector of length n
    %    aalpha      : Vector of length m
    %
    % OUTPUTS
    %    f          : Value of the objective function (scalar)
    %    grad_f     : Gradient of the objective function (vector of length n*m)
    % We can compute the gradient because of the nice structure of the problem.


        % reshape x into an n x m matrix
        x_matrix = reshape(x, [n, m]);

        % Initialize the objective function value
        f = 0;

        % initialize gradient
        grad_f_matrix = zeros(n, m);

        % compute the objective and gradient
        for i = 1:n
            llambda_i = llambda(i);
            for j = 1:m
                x_ij = x_matrix(i, j);
                oomega_ij = oomega(i, j);
                aalpha_j = aalpha(j);

                % % sanity check
                % if x_ij <= 1e-5
                %     f = - 1000;
                %     grad_f = zeros(n*m, 1);
                %     return;
                % end

                % objective
                term = aalpha_j * x_ij^(1 + oomega_ij) / (1 + oomega_ij);
                f = f + llambda_i * term;

                % gradient
                grad_term = aalpha_j * (1 + oomega_ij) * x_ij^oomega_ij / (1 + oomega_ij);
                grad_f_matrix(i, j) = llambda_i * grad_term;
            end
        end

        % Flatten the gradient matrix to a vector
        grad_f = grad_f_matrix(:);

        % minimization problem
        f = -f;
        grad_f = -grad_f;
    end
```

This following function takes a given set of physical parameters as input and report the optimal allocation:



```matlab
function x_opt_matrix = solve_SP(n, m, llambda, aalpha, oomega, e)
    % Solves the social planner problem.
    % INPUTS
    %    n           : Number of agents
    %    m           : Number of goods
    %    llambda     : Elasticity parameters - vector of length n
    %    aalpha      : Weights on goods - sector of length m
    %    oomega      : n x m matrix of omega_j^i values
    %    e           : n x m matrix of endowments
    % OUTPUTS
    %    x_opt_matrix  : Optimal allocations

    % total endowments for each good
    e_total = sum(e, 1);

    % initial guess
    x0 = e_total / n;
    x0 = repmat(x0, n, 1);
    x0 = x0(:);

    % set up constraints to the problem
    lb = ones(n * m, 1) * 1e-5;
    ub = [];

    % equality constraints/resource constraints: sum_i x_j^i = e_total_j for each good j
    Aeq = zeros(m, n * m);
    for j = 1:m
        for i = 1:n
            idx = (j - 1) * n + i; % index for summing over agents
            Aeq(j, idx) = 1;
        end
    end
    beq = e_total';

    % inequality constraints
    A = [];
    b = [];

    objective = @(x) SP_objective(x, n, m, oomega, llambda, aalpha);
    options = optimoptions('fmincon', 'Display', 'iter', 'Algorithm', 'sqp', 'SpecifyObjectiveGradient', true);

    % solve
    [x_opt, fval, ~, ~] = fmincon(objective, x0, A, b, Aeq, beq, lb, ub, [], options);

    % unpack results
    x_opt_matrix = reshape(x_opt, [n, m]);

    disp('Optimal allocations (x_j^i):');
    disp(x_opt_matrix);

    disp('Maximum value of the objective function:');
    disp(-fval);

end
```

I compute the optimal allocation in the two examples:

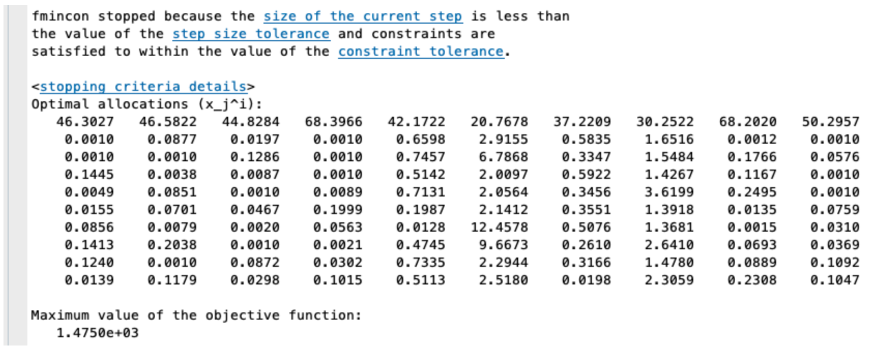


```matlab
% ---------------------------------------------------
% 3. Computing Pareto efficient allocations
% ---------------------------------------------------

% ------------- solve a simple problem -------------
n = 3;
m = 3;
llambda = [0.5; 0.25; 0.25];
aalpha = [1; 1; 1];
oomega = [-.5 * ones(3, 1), -.2 * ones(3, 1), -.5 * ones(3, 1)];   % i - agent, j - good
e = [20, 0, 0;        % i - agent, j - good
     10, 20, 10;
     0, 0, 20];


% solve the social planner problem
solve_SP(n, m, llambda, aalpha, oomega, e);



% ------------- m = n = 10 -------------
n = 10;
m = 10;
llambda = [10, 1, 1, 1, 1, 1, 1, 1, 1, 1]';
aalpha = rand(m, 1);
oomega = - rand(m, n);   % i - agent, j - good
e = 10 * rand(n, m);


% solve the social planner problem
solve_SP(n, m, llambda, aalpha, oomega, e);
```

Results

Even with $n = m = 10$ and a significant amount of heterogeneity (I assign random weights and elasticities to agents), the social planner's problem can be solved very fast (less than 1 sec). Here are the resulting allocations:

```
Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
Optimal allocations (x_j^i):
   20.0000   18.8235   20.0000
    5.0000    0.5882    5.0000
    5.0000    0.5882    5.0000

Maximum value of the objective function:
   20.3662
```



```
fmincon stopped because the size of the current step is less than
the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

<stopping criteria details>
Optimal allocations (x_j^i):
   46.3027   46.5822   44.8284   68.3966   42.1722   20.7678   37.2209   30.2522   68.2020   50.2957
    0.0010    0.0877    0.0197    0.0010    0.6598    2.9155    0.5835    1.6516    0.0012    0.0010
    0.0010    0.0010    0.1286    0.0010    0.7457    6.7868    0.3347    1.5484    0.1766    0.0576
    0.1445    0.0038    0.0087    0.0010    0.5142    2.0097    0.5922    1.4267    0.1167    0.0010
    0.0049    0.0851    0.0010    0.0089    0.7131    2.0564    0.3456    3.6199    0.2495    0.0010
    0.0155    0.0701    0.0467    0.1999    0.1987    2.1412    0.3551    1.3918    0.0135    0.0759
    0.0856    0.0079    0.0020    0.0563    0.0128   12.4578    0.5076    1.3681    0.0015    0.0310
    0.1413    0.2038    0.0010    0.0021    0.4745    9.6673    0.2610    2.6410    0.0693    0.0369
    0.1240    0.0010    0.0872    0.0302    0.7335    2.2944    0.3166    1.4780    0.0889    0.1092
    0.0139    0.1179    0.0298    0.1015    0.5113    2.5180    0.0198    2.3059    0.2308    0.1047

Maximum value of the objective function:
   1.4750e+03
```

5. Computing Equilibrium allocations

Now, we move on to compute the decentralized competitive equilibrium allocations in the same economy.

The individual decision problem gives the Lagrangian:

$$\mathcal{L}^i = \sum_{j=1}^{m} \alpha_j \frac{(x_j^i)^{1+\omega_j^i}}{1+\omega_j^i} + \lambda^i \left[\sum_{j=1}^{n} p_j e_j^i - \sum_{j=1}^{n} p_j x_j^i \right]$$

We can set the numeriare: $p_1 = 1$ and get the first-order conditions:

$$\alpha_j (x_j^i)^{\omega_j^i} = \lambda^i p_j \implies x_j^i = \left(\frac{\lambda^i p_j}{\alpha_j} \right)^{\frac{1}{\omega_j^i}}$$

where we substitute out consumptions to get a demand curve for each good j for each agent i.

The competitive equilibrium is characterized by

- m market clearing conditions:

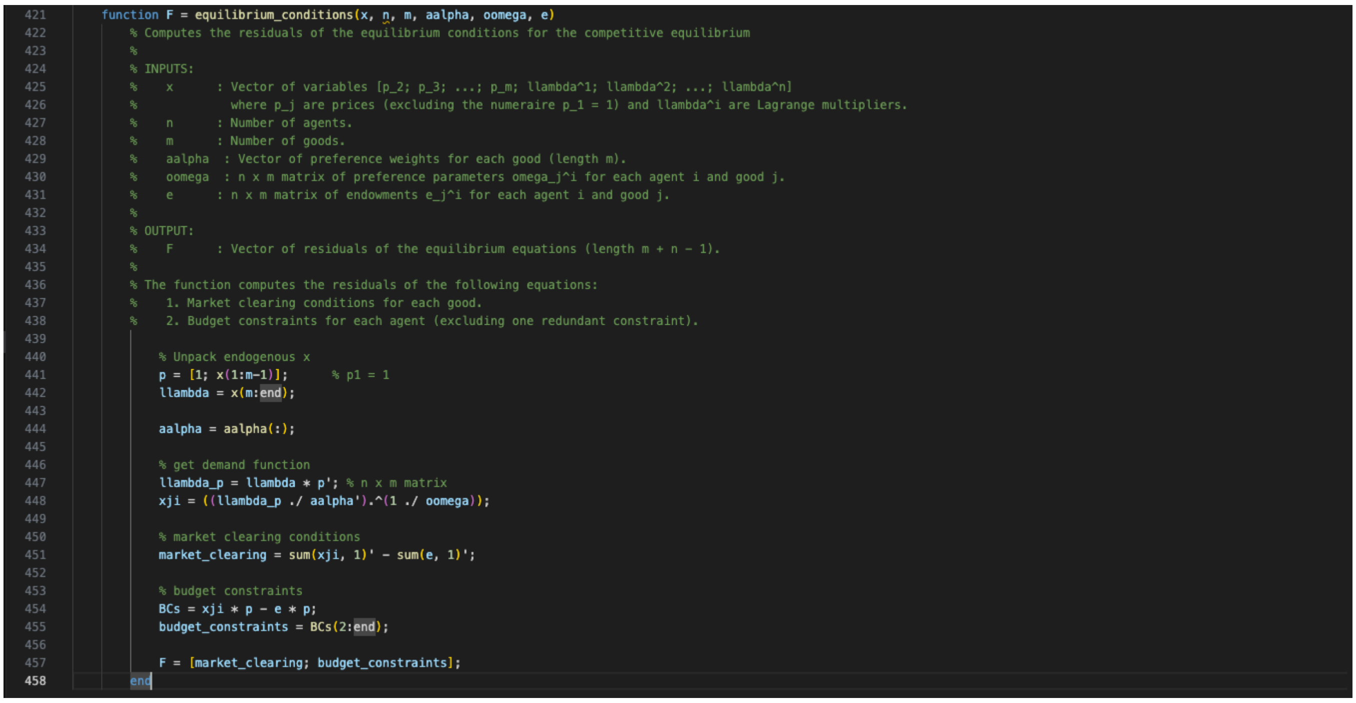
$$\sum_{i=1}^{n} \left(\frac{\lambda^i p_j}{\alpha_j} \right)^{\frac{1}{\omega_j^i}} = \sum_{i=1}^{n} e_j^i \quad \forall j = 1, ..., m$$

- $n-1$ budget constraint (the last one will be redundant):

$$\sum_{i=1}^{n} p_j e_j^i = \sum_{i=1}^{n} p_j x_j^i, \ \forall i = 2, ..., n$$

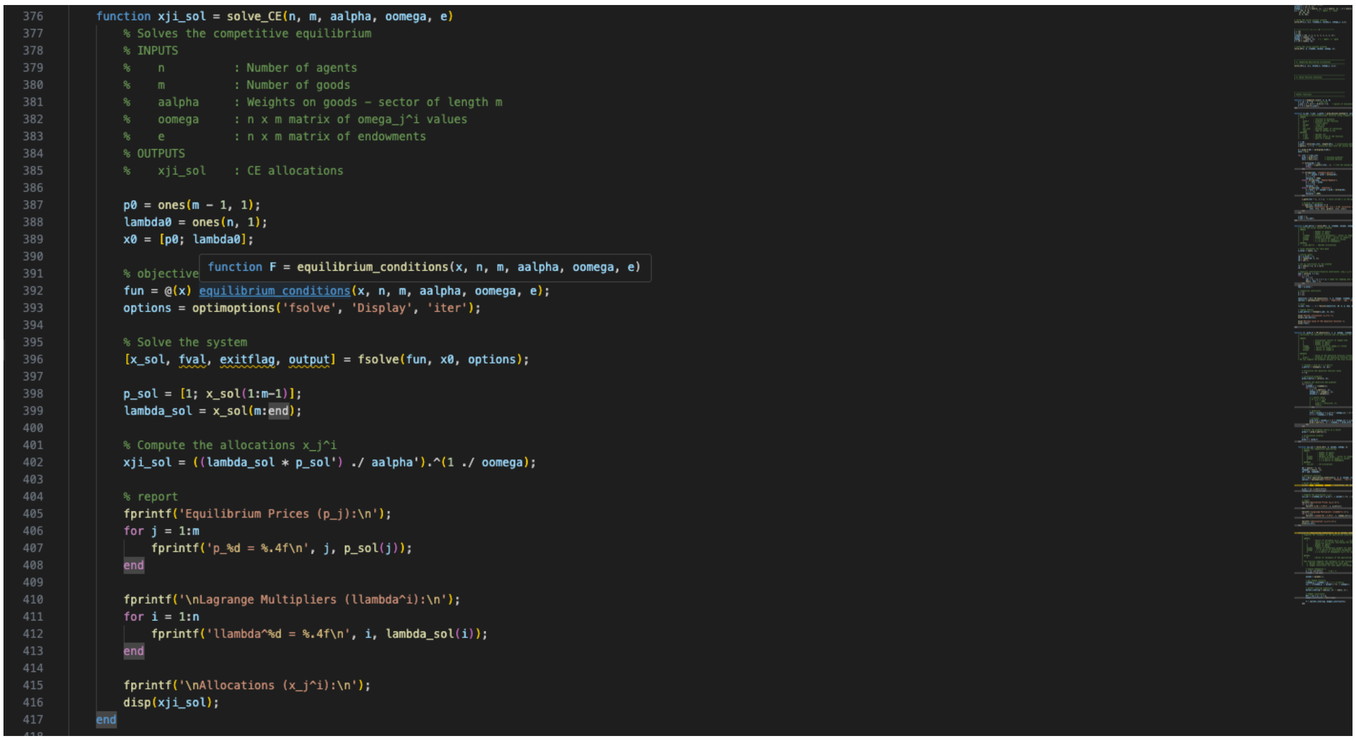
  in $m-1$ prices and n Lagrangian multipliers ($n+m-1$ variables in total).

Next, we code this non-linear system of equations into Matlab.



```matlab
function F = equilibrium_conditions(x, n, m, aalpha, oomega, e)
    % Computes the residuals of the equilibrium conditions for the competitive equilibrium
    %
    % INPUTS:
    %    x      : Vector of variables [p_2; p_3; ...; p_m; llambda^1; llambda^2; ...; llambda^n]
    %             where p_j are prices (excluding the numeraire p_1 = 1) and llambda^i are Lagrange multipliers.
    %    n      : Number of agents.
    %    m      : Number of goods.
    %    aalpha  : Vector of preference weights for each good (length m).
    %    oomega  : n x m matrix of preference parameters omega_j^i for each agent i and good j.
    %    e      : n x m matrix of endowments e_j^i for each agent i and good j.
    %
    % OUTPUT:
    %    F      : Vector of residuals of the equilibrium equations (length m + n - 1).
    %
    % The function computes the residuals of the following equations:
    %    1. Market clearing conditions for each good.
    %    2. Budget constraints for each agent (excluding one redundant constraint).

        % Unpack endogenous x
        p = [1; x(1:m-1)];      % p1 = 1
        llambda = x(m:end);

        aalpha = aalpha(:);

        % get demand function
        llambda_p = llambda * p'; % n x m matrix
        xji = ((llambda_p ./ aalpha').^(1 ./ oomega));

        % market clearing conditions
        market_clearing = sum(xji, 1)' - sum(e, 1)';

        % budget constraints
        BCs = xji * p - e * p;
        budget_constraints = BCs(2:end);

        F = [market_clearing; budget_constraints];
    end
```

And similarly, we pack the solver into a function:



```matlab
function xji_sol = solve_CE(n, m, aalpha, oomega, e)
    % Solves the competitive equilibrium
    % INPUTS
    %    n          : Number of agents
    %    m          : Number of goods
    %    aalpha     : Weights on goods - sector of length m
    %    oomega     : n x m matrix of omega_j^i values
    %    e          : n x m matrix of endowments
    % OUTPUTS
    %    xji_sol    : CE allocations

    p0 = ones(m - 1, 1);
    lambda0 = ones(n, 1);
    x0 = [p0; lambda0];

    % objective
    fun = @(x) equilibrium_conditions(x, n, m, aalpha, oomega, e);
    options = optimoptions('fsolve', 'Display', 'iter');

    % Solve the system
    [x_sol, fval, exitflag, output] = fsolve(fun, x0, options);

    p_sol = [1; x_sol(1:m-1)];
    lambda_sol = x_sol(m:end);

    % Compute the allocations x_j^i
    xji_sol = ((lambda_sol * p_sol') ./ aalpha').^(1 ./ oomega);

    % report
    fprintf('Equilibrium Prices (p_j):\n');
    for j = 1:m
        fprintf('p_%d = %.4f\n', j, p_sol(j));
    end

    fprintf('\nLagrange Multipliers (llambda^i):\n');
    for i = 1:n
        fprintf('llambda^%d = %.4f\n', i, lambda_sol(i));
    end

    fprintf('\nAllocations (x_j^i):\n');
    disp(xji_sol);
end
```

Let us solve the simple problem as in SP to see the results:

```
                                                Norm of      First-order   Trust-region
 Iteration  Func-count     ||f(x)||^2           step         optimality    radius
     0          6             3405                            1.21e+03            1
     1          7             3405                  1         1.21e+03            1
     2         13          2426.13               0.25         1.84e+03         0.25
     3         14          2426.13              0.625         1.84e+03        0.625
     4         20          1825.61            0.15625              787        0.156
     5         26          1372.88           0.390625         1.28e+03        0.391
     6         27          1372.88           0.976562         1.28e+03        0.977
     7         33          1079.64           0.244141         1.19e+03        0.244
     8         34          1079.64           0.610352         1.19e+03         0.61
     9         40          925.175           0.152588         3.14e+03        0.153
    10         46          714.488           0.152588              802        0.153
    11         52          217.052            0.38147         2.83e+03        0.381
    12         53          217.052           0.790428         2.83e+03        0.954
    13         59          112.746           0.197607         1.08e+03        0.198
    14         60          112.746           0.494018         1.08e+03        0.494
    15         66          65.4941           0.123504              353        0.124
    16         72          27.6928           0.308761         4.97e+03        0.309
    17         78         0.124771          0.0825842              295        0.309
    18         84      1.63256e-05         0.00780496             2.77        0.309
    19         90      1.26204e-12        0.000109296         0.000649        0.309
    20         96       6.2705e-26        2.99957e-08         2.45e-10        0.309

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
Equilibrium Prices (p_j):
p_1 = 1.0000
p_2 = 2.2745
p_3 = 1.0000

Lagrange Multipliers (llambda^i):
llambda^1 = 0.3700
llambda^2 = 0.2549
llambda^3 = 0.3700

Allocations (x_j^i):
    7.3053    2.3694    7.3053
   15.3893   15.2612   15.3893
    7.3053    2.3694    7.3053
```

Since the total endowment of the first and third goods are the same, and the agents assign the same weights and elasticities. They have the same prices in CE.

6. Value Function Iteration

6.1 Social planner

The original representative agent problem can be recast into the following social planner's problem with recursive formulation:

$$
\begin{aligned}
V(k, i; \tau, z) = \max_{c,l,i'} \ & \log c + 0.2 \log g - \frac{l^2}{2} + 0.97 \mathbb{E}_{\tau', z' \mid \tau, z} \left[V(k', i'; \tau', z') \right] \\
\text{s.t. } & c + i' = (1-\tau)(1-\alpha) e^z k^\alpha l^{1-\alpha} + \alpha e^z k^\alpha l^{1-\alpha} \\
& k' = 0.9k + \left(1 - 0.05 \left(\frac{i'}{i} - 1 \right)^2 \right) i'
\end{aligned}
$$

There are two things worth noting:

- One need an additional state variable i, which represents that last-period investment, to make the problem recursive.
- There is a question as to **whether one should endogenize the government balanced budget** $g = \tau w l$ into the decision problem.
    1. Because the question asks for solving the social planner's problem, I'm tempted to say yes. In that case, one just replaces the g in the value function with $\tau w l$, and agents will take it into account in their FOCs.
    2. If no, one solves the Bellan equation as if it is without this term $\log g$, since it does not induce any changes to decisions. Then, with the solved policy $l^\star$ at each grid point, we substitute in $g = \tau e^z (1-\alpha) k^\alpha (l^\star)^{1-\alpha}$ to the value function.

I will solve the problem **assuming the second case**, because it is more natural and consistent with a market equilibrium. Define for convenience the wedge on output: $\psi \equiv (1-\tau)(1-\alpha) + \alpha$. As a result, the problem becomes:

$$
\begin{aligned}
V\left(k, i; \tau, z\right) = \max_{c,l,i'} \; & \log c + \eta \log\left(g\right) - \frac{l^2}{2} + \beta \mathbb{E}_{\tau', z' | \tau, z}\left[V\left(k', i'; \tau', z'\right)\right] \\
\text{s.t. } & c + i' = \psi e^z k^\alpha l^{1-\alpha} \\
& k' = \left(1-\delta\right) k + \left(1 - \phi\left(\frac{i'}{i} - 1\right)^2\right) i'
\end{aligned}
$$

where $g = \tau(1-\alpha) e^z k^\alpha l^{1-\alpha}$, and I relabeled parameter values using parameter notations for generality.

6.2 Steady state

We can write the Lagrangian:

$$
\begin{aligned}
\mathcal{L} = & \log c + \eta \log(g) - \frac{l^2}{2} + \beta \mathbb{E}_{\tau', z' | \tau, z}\left[V\left(k', i'; \tau', z'\right)\right] \\
& + \lambda\left(\psi e^z k^\alpha l^{1-\alpha} - c - i'\right)
\end{aligned}
$$

This gives the first-order conditions:

$$
\begin{aligned}
\frac{1}{c} &= \lambda \\
l &= \lambda \psi (1-\alpha) e^z k^\alpha l^{-\alpha} \\
\beta \mathbb{E}_{\tau', z' | \tau, z}\left[V_i\left(k', i'; \tau', z'\right) + V_k\left(k', i'; \tau', z'\right) \frac{\partial k'}{\partial i'}\right] &= \lambda
\end{aligned}
$$

where

$$
\frac{\partial k'}{\partial i'} = 1 - \phi\left(\frac{i'}{i} - 1\right)^2 + 2\phi \frac{i'}{i}\left(\frac{i'}{i} - 1\right)
$$

And the envelop conditions:

$$
\begin{aligned}
V_i\left(k, i; \tau, z\right) &= \beta \mathbb{E}_{\tau', z' | \tau, z}\left[V_k\left(k', i'; \tau', z'\right) \frac{\partial k'}{\partial i}\right] \\
V_k\left(k, i; \tau, z\right) &= \lambda \psi \alpha e^z k^{\alpha-1} l^{1-\alpha} \\
& \quad + \beta(1-\delta) \mathbb{E}_{\tau', z' | \tau, z}\left[V_k\left(k', i'; \tau', z'\right)\right]
\end{aligned}
$$

where

$$
\frac{\partial k'}{\partial i} = -2\phi\left(\frac{i'}{i} - 1\right)\left(\frac{i'}{i}\right)^2
$$

In the steady state, we have

$$
V_k\left(\bar{k}, \bar{i}; \bar{\tau}, \bar{z}\right) = \frac{1}{1 - \beta(1-\delta)} \lambda \psi \alpha e^z \bar{k}^{\alpha-1} \bar{l}^{1-\alpha}
$$

$$
V_i\left(\bar{k}, \bar{i}; \bar{\tau}, \bar{z}\right) = 0
$$

As a result, the steady state is characterized by the following equations:

$$
\begin{aligned}
\frac{1}{\bar{c}} &= \bar{\lambda} \\
\bar{l} &= \bar{\lambda} \psi (1-\alpha) e^{\bar{z}} \bar{k}^\alpha \bar{l}^{-\alpha} \\
\psi \alpha e^{\bar{z}} \bar{k}^{\alpha-1} \bar{l}^{1-\alpha} &= 1/\beta - 1 + \delta
\end{aligned}
$$

These implies that the steady state consists of three equations in three unknowns $(\bar{l}, \bar{k}, \bar{c})$:

$$\bar{l} = \frac{\psi(1-\alpha)e^{\bar{z}}\bar{k}^{\alpha}\bar{l}^{-\alpha}}{\bar{c}}$$
$$1/\beta - 1 + \delta = \psi\alpha e^{\bar{z}}\bar{k}^{\alpha-1}\bar{l}^{1-\alpha}$$
$$\bar{c} + \delta\bar{k} = \psi e^{\bar{z}}\bar{k}^{\alpha}\bar{l}^{1-\alpha}$$

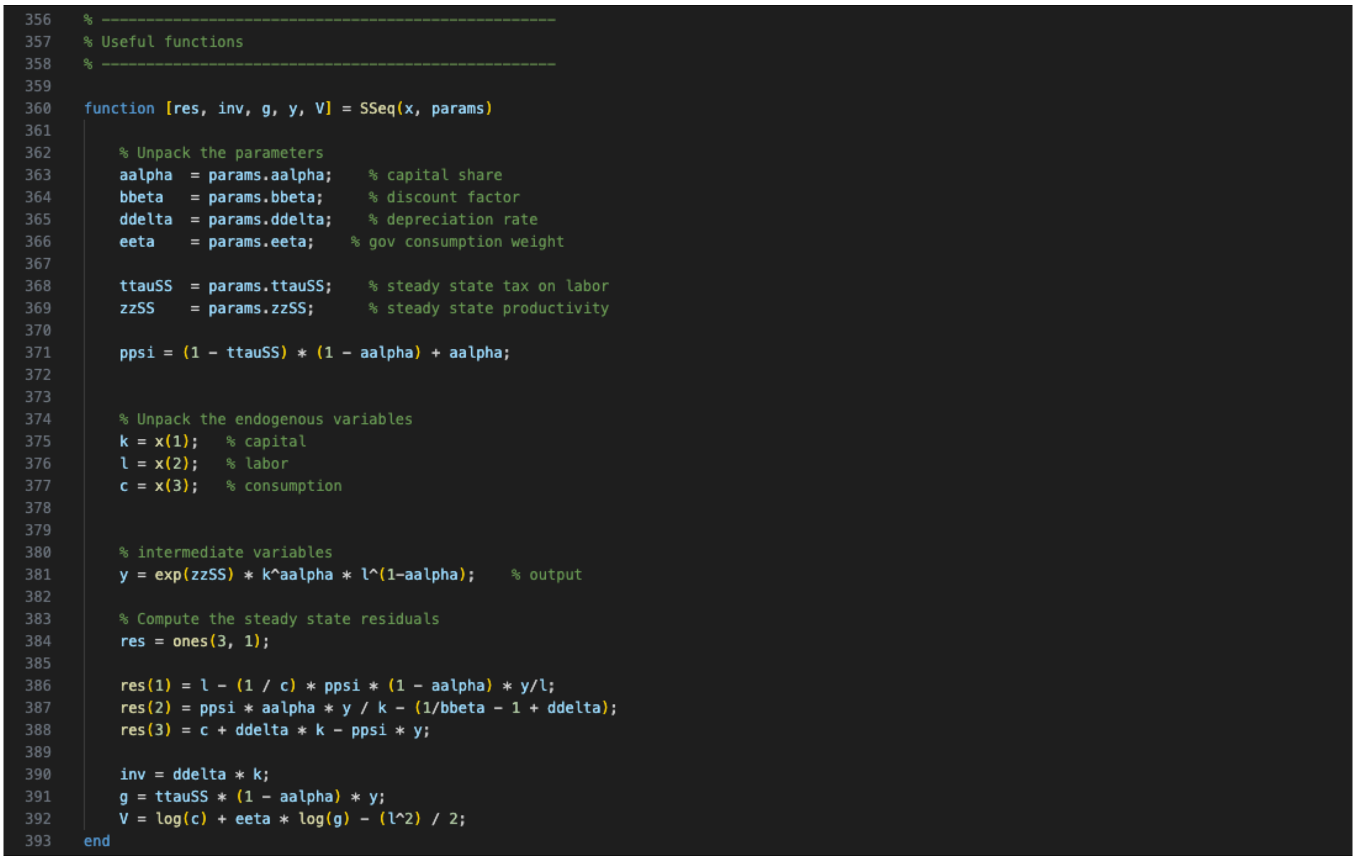
and we can then get other variables:

$$\bar{y} = e^{\bar{z}}\bar{k}^{\alpha}\bar{l}^{-\alpha}$$
$$\bar{i} = \delta\bar{k}$$
$$\bar{g} = \tau(1-\alpha)\bar{y}$$

Let us solve it in Matlab.

Codes

The following function solve for the steady state as a system of non-linear equations:



```matlab
% -------------------------------------------------
% Useful functions
% -------------------------------------------------

function [res, inv, g, y, V] = SSeq(x, params)

    % Unpack the parameters
    aalpha  = params.aalpha;     % capital share
    bbeta   = params.bbeta;      % discount factor
    ddelta  = params.ddelta;     % depreciation rate
    eeta    = params.eeta;     % gov consumption weight

    ttauSS  = params.ttauSS;     % steady state tax on labor
    zzSS    = params.zzSS;       % steady state productivity

    ppsi = (1 - ttauSS) * (1 - aalpha) + aalpha;


    % Unpack the endogenous variables
    k = x(1);    % capital
    l = x(2);    % labor
    c = x(3);    % consumption


    % intermediate variables
    y = exp(zzSS) * k^aalpha * l^(1-aalpha);    % output

    % Compute the steady state residuals
    res = ones(3, 1);

    res(1) = l - (1 / c) * ppsi * (1 - aalpha) * y/l;
    res(2) = ppsi * aalpha * y / k - (1/bbeta - 1 + ddelta);
    res(3) = c + ddelta * k - ppsi * y;

    inv = ddelta * k;
    g = ttauSS * (1 - aalpha) * y;
    V = log(c) + eeta * log(g) - (l^2) / 2;
end
```

The steady state is computed and put into a structure for future use:

```
% ---------------------------------------------------
% 1. Compute the steady state
% ---------------------------------------------------

% Define the parameters
params.aalpha = 0.33;
params.bbeta = 0.97;
params.ddelta = 0.1;
params.pphi = 0.05;
params.eeta = 0.2;
params.ttauSS = 0.25;
params.zzSS = 0;


% Initial guess for the endogenous variables [k, l, c]
x0 = [1, 0.5, 0.5]';

% Solve the steady state equations using fsolve
options = optimoptions('fsolve', 'Display', 'iter');
[x_ss, fval, exitflag] = fsolve(@(x) SSeq(x, params), x0, options);
[~, inv_ss, g_ss, y_ss, V_ss] = SSeq(x_ss, params);

% Display the results
k_ss = x_ss(1);
l_ss = x_ss(2);
c_ss = x_ss(3);

disp('Steady state values:');
disp(['Capital (k): ', num2str(k_ss)]);
disp(['Labor (l): ', num2str(l_ss)]);
disp(['Consumption (c): ', num2str(c_ss)]);
disp(['Investment (inv): ', num2str(inv_ss)]);
disp(['Output (y): ', num2str(y_ss)]);
disp(['Gov consumption (g): ', num2str(g_ss)]);
disp(['Value: ', num2str(V_ss)]);

ssvals = struct();
ssvals.k_ss = k_ss;
ssvals.l_ss = l_ss;
ssvals.c_ss = c_ss;
ssvals.inv_ss = inv_ss;
ssvals.y_ss = y_ss;
ssvals.g_ss = g_ss;
ssvals.V_ss = V_ss;
```

Results:

```
<stopping criteria details>
Steady state values:
Capital (k): 2.8609
Labor (l): 0.94646
Consumption (c): 0.84897
Investment (inv): 0.28609
Output (y): 1.3634
Gov consumption (g): 0.22837
Value: -0.90698
```

6.3 Value function iteration (fixed grid, multi-grid, stochastic grid, with policy acceleration)

To organize this section in a cleaner way, I solve the recursive problem using the basic Value Function Iteration methods with exogenous grids first. I relegate the Endogenous Grid Method to the next section.

I proceed with the following steps:

1. solve the problem with **value function iteration with fixed grids** (250 gridpoints for capital, 50 gridpoints for lagged investment).
2. accelerate the method by **switching between policy and value function iteration**. In particular, only solve the maximization problem once in ten iterations, while simply fixing the policy (decision) in the rest.
3. **Multigrid**: solve the problem on a coarser grid (100 gridpoints for capital) first, then on a finer grid (500 gridpoints for capital) with the solution in the first round as the initial guess for value function, and finally on 5000 gridpoints for capital in the last round.
4. **Stochastic grid**: in the spirit of Rust (1997), in each round of the value function iteration, I draw an uniform random sample (of size 200) from the capital gridpoints (500 in total) at each iteration,

solve the problem on those grid points, interpolate the value function, and iterate. I use a benchmark grid (the full grid) to evaluate convergence result (I re-interpolate the value function over the benchmark grid at the end of every iteration to compute the distance from the last round).

In all the methods except the first one, policy iteration acceleration (occasionally solving the optimal decision) steps are applied. In addition, I use parallelization in Matlab with 8 workers on my laptop to make my computation faster.

To solve the problem using Value Function Iteration, we can rescale the value function $\tilde{V} = (1-\beta)V$, and write the problem as:

$$\begin{aligned}\tilde{V}(k, i; \tau, z) = \max_{c,l} (1-\beta)\left[\log c + \eta \log(g) - \frac{l^2}{2}\right] + \beta \mathbb{E}_{\tau', z'|\tau, z}\left[\tilde{V}(k', i'; \tau', z')\right] \\ \text{s.t. } i' = \psi e^z k^\alpha l^{1-\alpha} - c \\ k' = (1-\delta)k + \left(1 - \phi(\frac{i'}{i} - 1)^2\right) i'\end{aligned}$$

To reduce the dimensionality of the decision problem (labor and investment), I use the following trick to reformulate the problem (making use of the static first-order condition for labor decision): notice that the labor decision is a static problem given the state variable, with FOC:

$$l = (1-\alpha)\psi e^z k^\alpha l^{-\alpha}$$

which implies optimal labor:

$$l^* = ((1-\alpha)\psi e^z k^\alpha)^{\frac{1}{1+\alpha}}$$

Thus, the problem becomes one that only involves a single choice variable:

$$\begin{aligned}\tilde{V}(k, i; \tau, z) = \max_{c} (1-\beta)\left[\log c + \eta \log(g) - \frac{l^{*2}}{2}\right] + \beta \mathbb{E}_{\tau', z'|\tau, z}\left[\tilde{V}(k', i'; \tau', z')\right] \\ \text{s.t. } c = \psi e^z k^\alpha l^{*1-\alpha} - i' \\ k' = (1-\delta)k + \left(1 - \phi\left(\frac{i'}{i} - 1\right)^2\right) i'\end{aligned}$$

Euler equation error

The Euler equation error can be defined by:

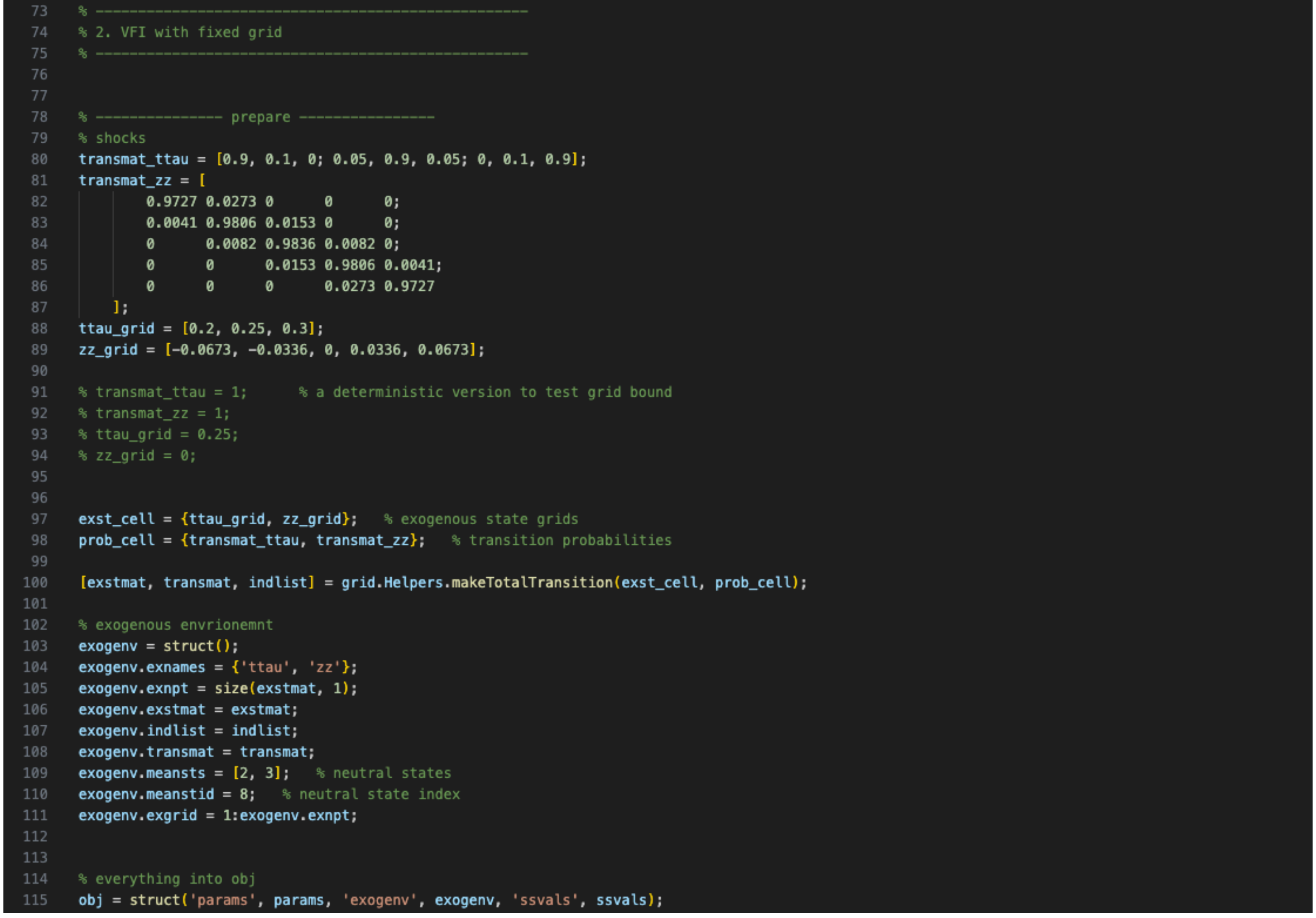
$$err = 1 - \frac{\beta \mathbb{E}_{\tau', z'|\tau, z}\left[V_i(k', i'; \tau', z') + V_k(k', i'; \tau', z')\frac{\partial k'}{\partial i'}\right]}{(1-\beta)1/c}$$

where $\frac{\partial k'}{\partial i'} = 1 - \phi\left(\frac{i'}{i} - 1\right)^2 + 2\phi\frac{i'}{i}\left(\frac{i'}{i} - 1\right)$.

We can compute the derivative of the value function with respect to i and k using finite difference and the interpolated value function. Alternatively, we could use the Envelop condition to evaluate the value function derivatives more precisely.

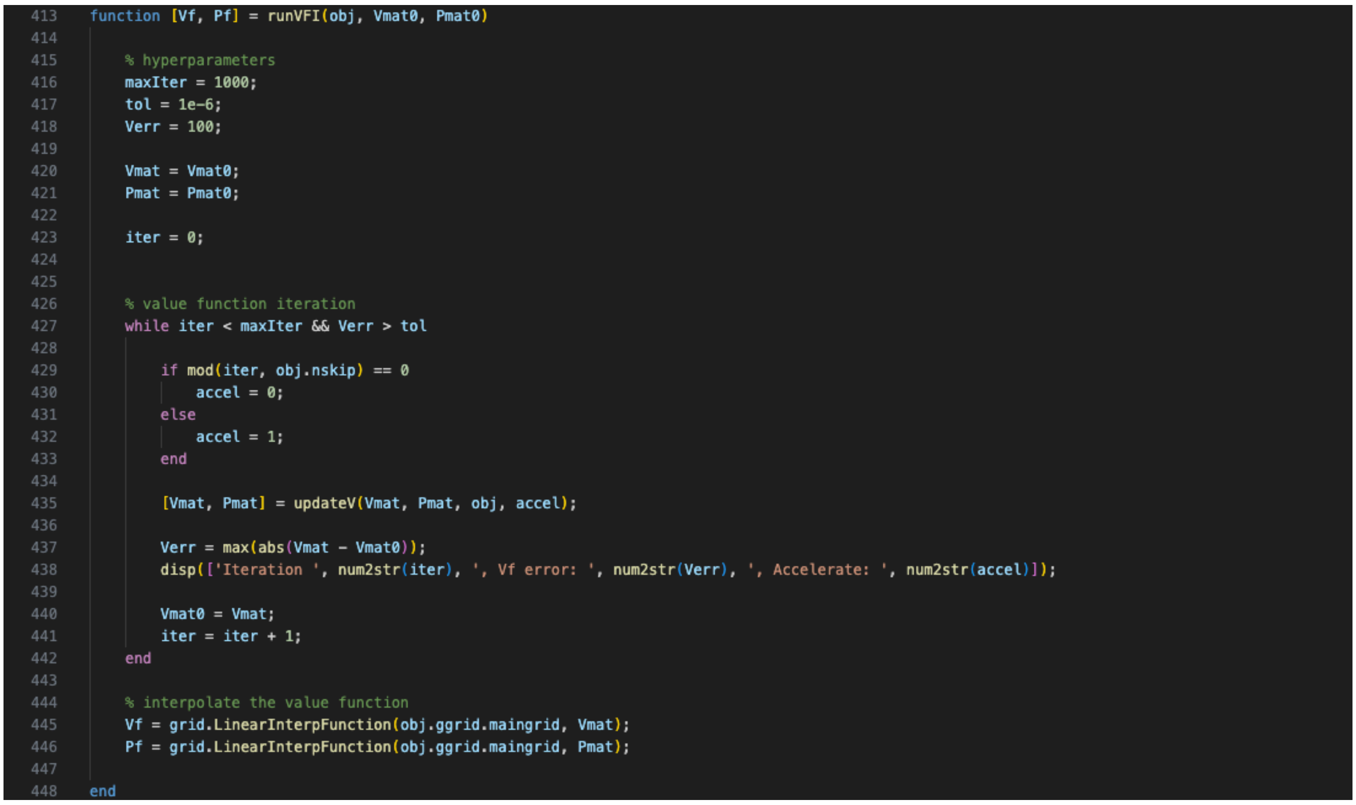
Codes

The parameters characterizing the exogenous state transition, about the physical environment, the steady state values, and the grid objects are inputted and stored in `obj` structure:



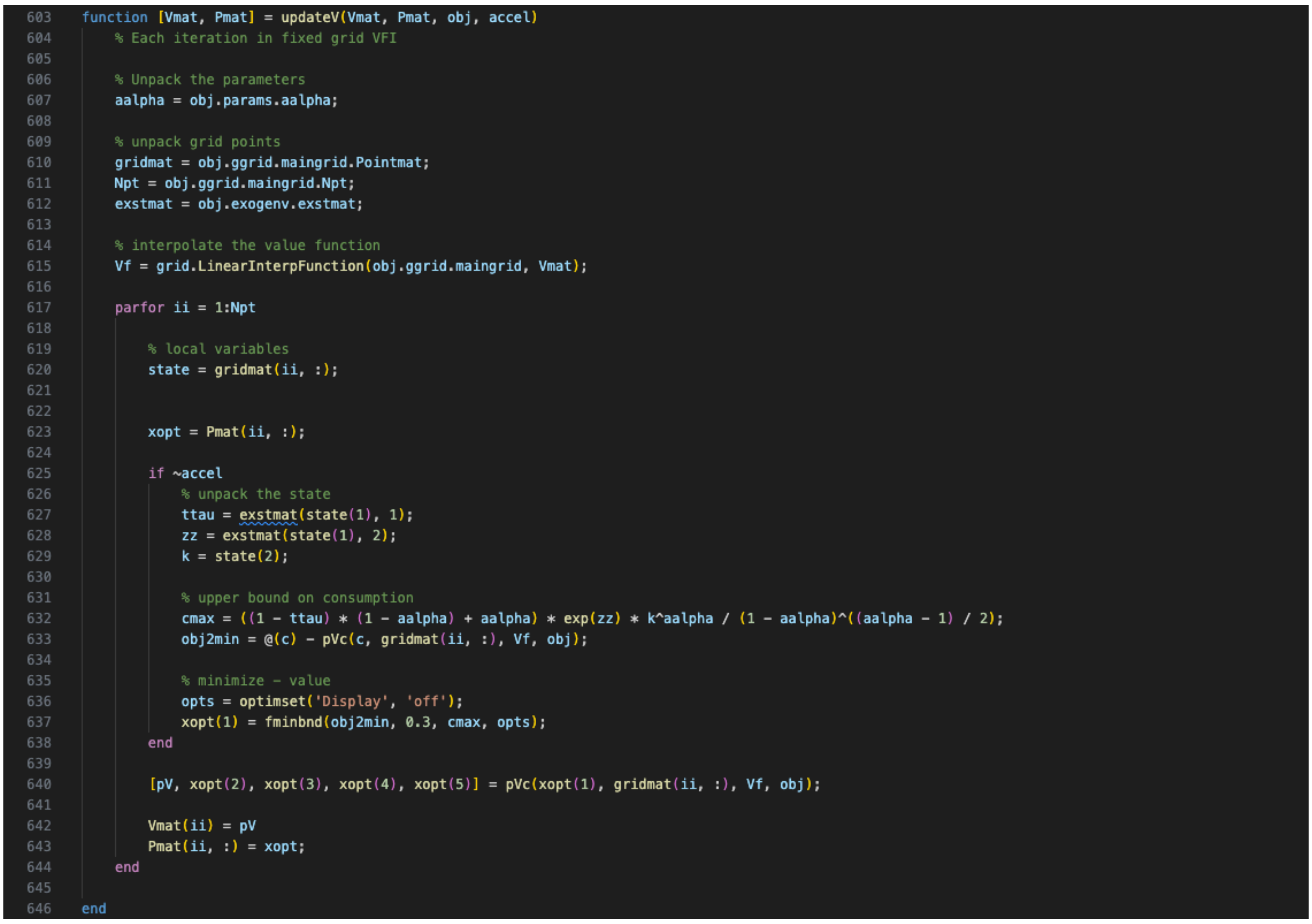
```matlab
% ----------------------------------------------------
% 2. VFI with fixed grid
% ----------------------------------------------------


% --------------- prepare ---------------
% shocks
transmat_ttau = [0.9, 0.1, 0; 0.05, 0.9, 0.05; 0, 0.1, 0.9];
transmat_zz = [
        0.9727 0.0273 0      0      0;
        0.0041 0.9806 0.0153 0      0;
        0      0.0082 0.9836 0.0082 0;
        0      0      0.0153 0.9806 0.0041;
        0      0      0      0.0273 0.9727
    ];
ttau_grid = [0.2, 0.25, 0.3];
zz_grid = [-0.0673, -0.0336, 0, 0.0336, 0.0673];

% transmat_ttau = 1;      % a deterministic version to test grid bound
% transmat_zz = 1;
% ttau_grid = 0.25;
% zz_grid = 0;


exst_cell = {ttau_grid, zz_grid};   % exogenous state grids
prob_cell = {transmat_ttau, transmat_zz};   % transition probabilities

[exstmat, transmat, indlist] = grid.Helpers.makeTotalTransition(exst_cell, prob_cell);

% exogenous envrionemnt
exogenv = struct();
exogenv.exnames = {'ttau', 'zz'};
exogenv.exnpt = size(exstmat, 1);
exogenv.exstmat = exstmat;
exogenv.indlist = indlist;
exogenv.transmat = transmat;
exogenv.meansts = [2, 3];   % neutral states
exogenv.meanstid = 8;   % neutral state index
exogenv.exgrid = 1:exogenv.exnpt;


% everything into obj
obj = struct('params', params, 'exogenv', exogenv, 'ssvals', ssvals);
```

The basic VFI procedure is coded as follows:



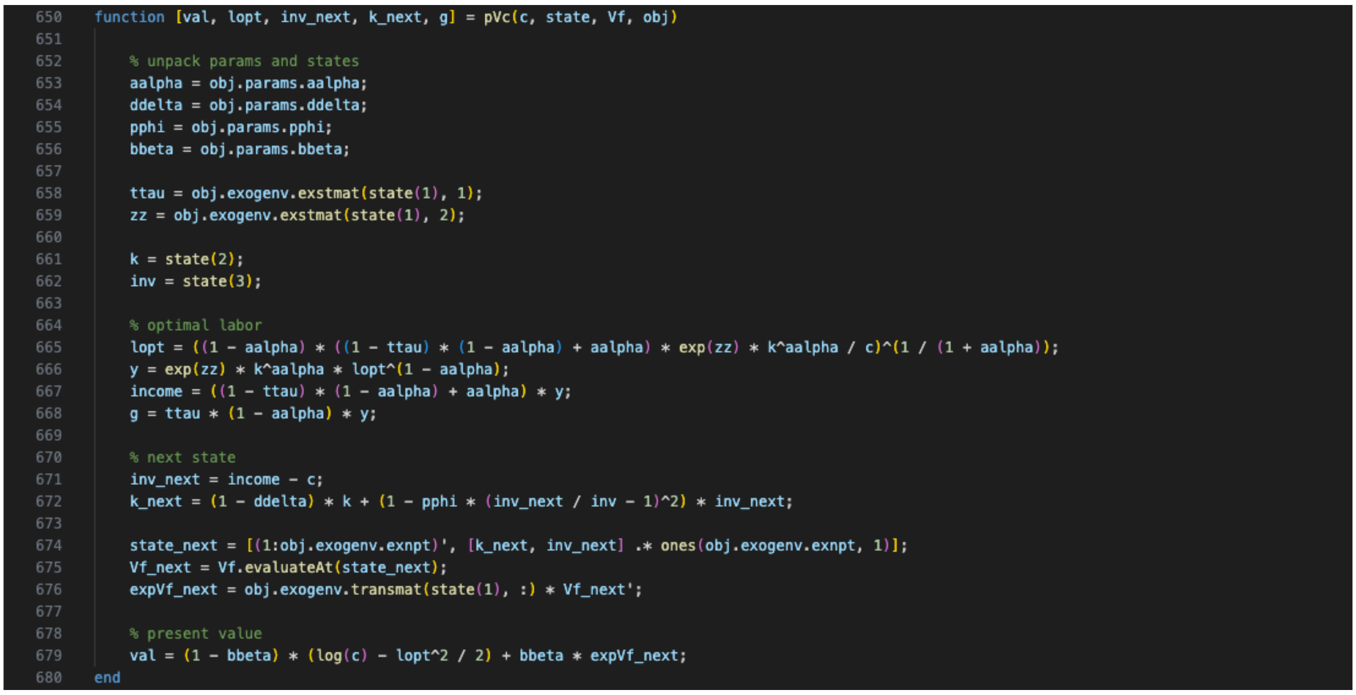
```matlab
function [Vf, Pf] = runVFI(obj, Vmat0, Pmat0)

    % hyperparameters
    maxIter = 1000;
    tol = 1e-6;
    Verr = 100;

    Vmat = Vmat0;
    Pmat = Pmat0;

    iter = 0;


    % value function iteration
    while iter < maxIter && Verr > tol

        if mod(iter, obj.nskip) == 0
            accel = 0;
        else
            accel = 1;
        end

        [Vmat, Pmat] = updateV(Vmat, Pmat, obj, accel);

        Verr = max(abs(Vmat - Vmat0));
        disp(['Iteration ', num2str(iter), ', Vf error: ', num2str(Verr), ', Accelerate: ', num2str(accel)]);

        Vmat0 = Vmat;
        iter = iter + 1;
    end

    % interpolate the value function
    Vf = grid.LinearInterpFunction(obj.ggrid.maingrid, Vmat);
    Pf = grid.LinearInterpFunction(obj.ggrid.maingrid, Pmat);

end
```

Each iteration updates V using the method we described:



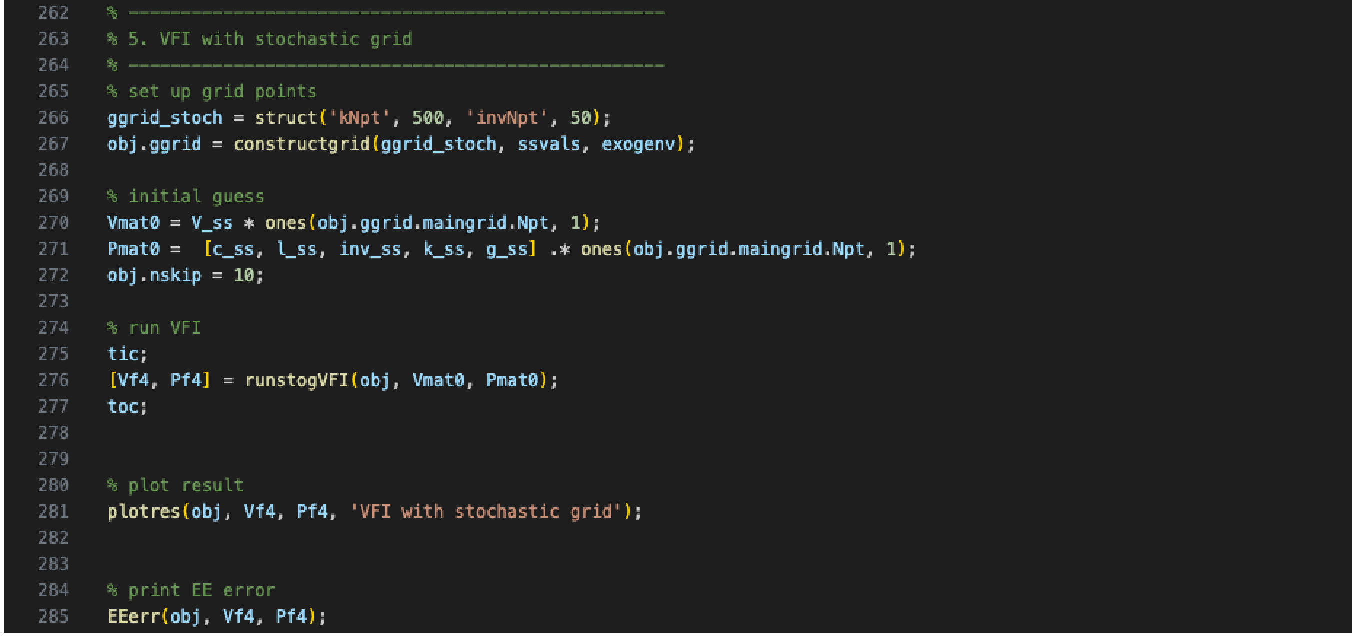
```matlab
function [Vmat, Pmat] = updateV(Vmat, Pmat, obj, accel)
    % Each iteration in fixed grid VFI

    % Unpack the parameters
    aalpha = obj.params.aalpha;

    % unpack grid points
    gridmat = obj.ggrid.maingrid.Pointmat;
    Npt = obj.ggrid.maingrid.Npt;
    exstmat = obj.exogenv.exstmat;

    % interpolate the value function
    Vf = grid.LinearInterpFunction(obj.ggrid.maingrid, Vmat);

    parfor ii = 1:Npt

        % local variables
        state = gridmat(ii, :);


        xopt = Pmat(ii, :);

        if ~accel
            % unpack the state
            ttau = exstmat(state(1), 1);
            zz = exstmat(state(1), 2);
            k = state(2);

            % upper bound on consumption
            cmax = ((1 - ttau) * (1 - aalpha) + aalpha) * exp(zz) * k^aalpha / (1 - aalpha)^((aalpha - 1) / 2);
            obj2min = @(c) - pVc(c, gridmat(ii, :), Vf, obj);

            % minimize - value
            opts = optimset('Display', 'off');
            xopt(1) = fminbnd(obj2min, 0.3, cmax, opts);
        end

        [pV, xopt(2), xopt(3), xopt(4), xopt(5)] = pVc(xopt(1), gridmat(ii, :), Vf, obj);

        Vmat(ii) = pV
        Pmat(ii, :) = xopt;
    end

end
```

The objective function to maximize (the present value given consumption choice):



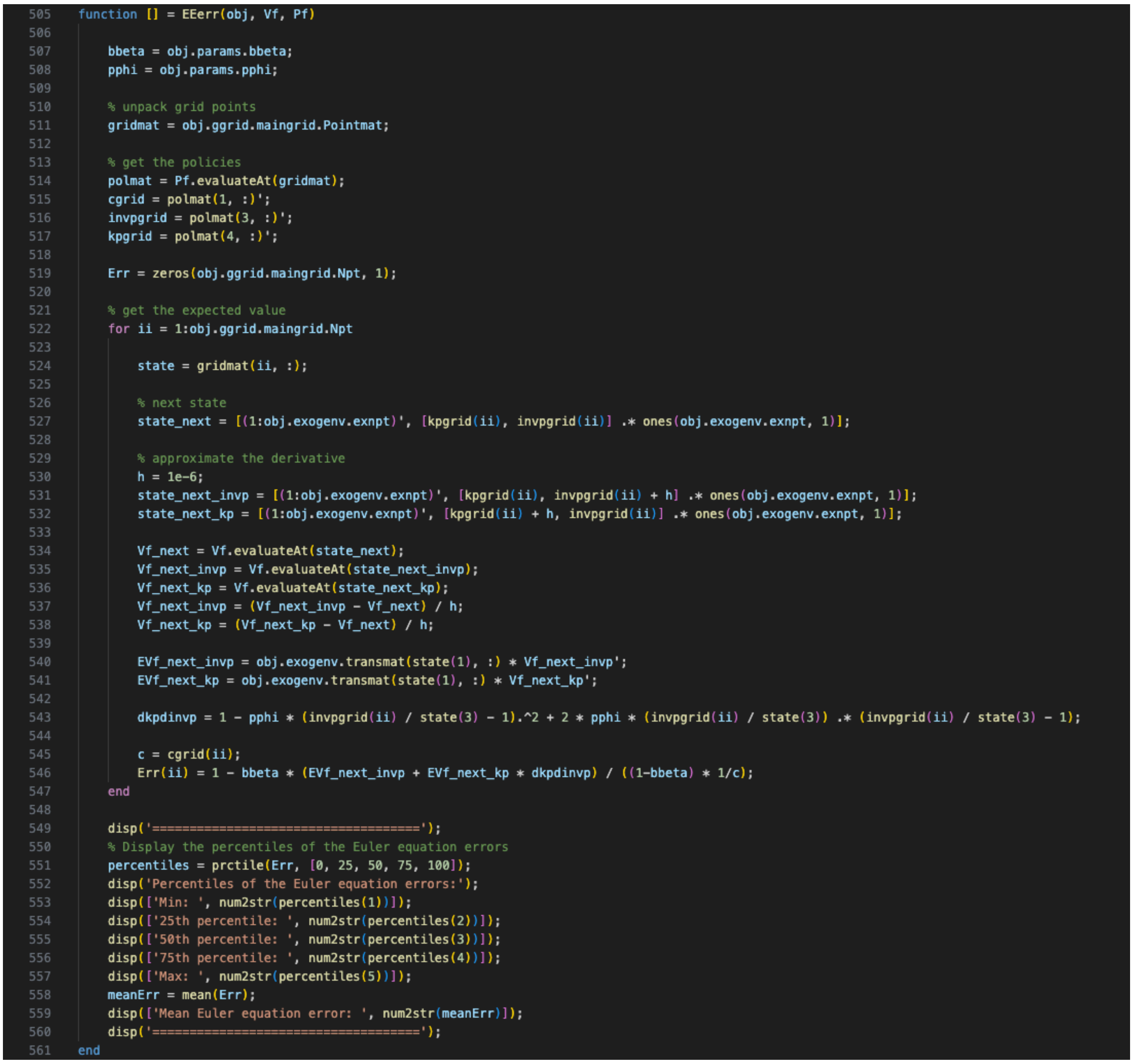
```matlab
function [val, lopt, inv_next, k_next, g] = pVc(c, state, Vf, obj)

    % unpack params and states
    aalpha = obj.params.aalpha;
    ddelta = obj.params.ddelta;
    pphi = obj.params.pphi;
    bbeta = obj.params.bbeta;

    ttau = obj.exogenv.exstmat(state(1), 1);
    zz = obj.exogenv.exstmat(state(1), 2);

    k = state(2);
    inv = state(3);

    % optimal labor
    lopt = ((1 - aalpha) * ((1 - ttau) * (1 - aalpha) + aalpha) * exp(zz) * k^aalpha / c)^(1 / (1 + aalpha));
    y = exp(zz) * k^aalpha * lopt^(1 - aalpha);
    income = ((1 - ttau) * (1 - aalpha) + aalpha) * y;
    g = ttau * (1 - aalpha) * y;

    % next state
    inv_next = income - c;
    k_next = (1 - ddelta) * k + (1 - pphi * (inv_next / inv - 1)^2) * inv_next;

    state_next = [(1:obj.exogenv.exnpt)', [k_next, inv_next] .* ones(obj.exogenv.exnpt, 1)];
    Vf_next = Vf.evaluateAt(state_next);
    expVf_next = obj.exogenv.transmat(state(1), :) * Vf_next';

    % present value
    val = (1 - bbeta) * (log(c) - lopt^2 / 2) + bbeta * expVf_next;
end
```

Finally, to run all these, a specfic grid is constructed. The initial guess for the value function (and policy function, though this is not consequential) is the steady state value and choice variables. One more thing to notice is that I include five variables of interest in the policy function object and update them during the iterations. They don't matter for the computation, but helps me to recover these variables implied by optimal policy for each state in an easier way.



```matlab
% ---------------------------------------------------
% 5. VFI with stochastic grid
% ---------------------------------------------------
% set up grid points
ggrid_stoch = struct('kNpt', 500, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_stoch, ssvals, exogenv);

% initial guess
Vmat0 = V_ss * ones(obj.ggrid.maingrid.Npt, 1);
Pmat0 =  [c_ss, l_ss, inv_ss, k_ss, g_ss] .* ones(obj.ggrid.maingrid.Npt, 1);
obj.nskip = 10;

% run VFI
tic;
[Vf4, Pf4] = runstogVFI(obj, Vmat0, Pmat0);
toc;


% plot result
plotres(obj, Vf4, Pf4, 'VFI with stochastic grid');


% print EE error
EEerr(obj, Vf4, Pf4);
```

The Euler equation errors are computed in the function:



```matlab
function [] = EEerr(obj, Vf, Pf)

    bbeta = obj.params.bbeta;
    pphi = obj.params.pphi;

    % unpack grid points
    gridmat = obj.ggrid.maingrid.Pointmat;

    % get the policies
    polmat = Pf.evaluateAt(gridmat);
    cgrid = polmat(1, :)';
    invpgrid = polmat(3, :)';
    kpgrid = polmat(4, :)';

    Err = zeros(obj.ggrid.maingrid.Npt, 1);

    % get the expected value
    for ii = 1:obj.ggrid.maingrid.Npt

        state = gridmat(ii, :);

        % next state
        state_next = [(1:obj.exogenv.exnpt)', [kpgrid(ii), invpgrid(ii)] .* ones(obj.exogenv.exnpt, 1)];

        % approximate the derivative
        h = 1e-6;
        state_next_invp = [(1:obj.exogenv.exnpt)', [kpgrid(ii), invpgrid(ii) + h] .* ones(obj.exogenv.exnpt, 1)];
        state_next_kp = [(1:obj.exogenv.exnpt)', [kpgrid(ii) + h, invpgrid(ii)] .* ones(obj.exogenv.exnpt, 1)];

        Vf_next = Vf.evaluateAt(state_next);
        Vf_next_invp = Vf.evaluateAt(state_next_invp);
        Vf_next_kp = Vf.evaluateAt(state_next_kp);
        Vf_next_invp = (Vf_next_invp - Vf_next) / h;
        Vf_next_kp = (Vf_next_kp - Vf_next) / h;

        EVf_next_invp = obj.exogenv.transmat(state(1), :) * Vf_next_invp';
        EVf_next_kp = obj.exogenv.transmat(state(1), :) * Vf_next_kp';

        dkpdinvp = 1 - pphi * (invpgrid(ii) / state(3) - 1).^2 + 2 * pphi * (invpgrid(ii) / state(3)) .* (invpgrid(ii) / state(3) - 1);

        c = cgrid(ii);
        Err(ii) = 1 - bbeta * (EVf_next_invp + EVf_next_kp * dkpdinvp) / ((1-bbeta) * 1/c);
    end

    disp('====================================');
    % Display the percentiles of the Euler equation errors
    percentiles = prctile(Err, [0, 25, 50, 75, 100]);
    disp('Percentiles of the Euler equation errors:');
    disp(['Min: ', num2str(percentiles(1))]);
    disp(['25th percentile: ', num2str(percentiles(2))]);
    disp(['50th percentile: ', num2str(percentiles(3))]);
    disp(['75th percentile: ', num2str(percentiles(4))]);
    disp(['Max: ', num2str(percentiles(5))]);
    meanErr = mean(Err);
    disp(['Mean Euler equation error: ', num2str(meanErr)]);
    disp('====================================');
end
```

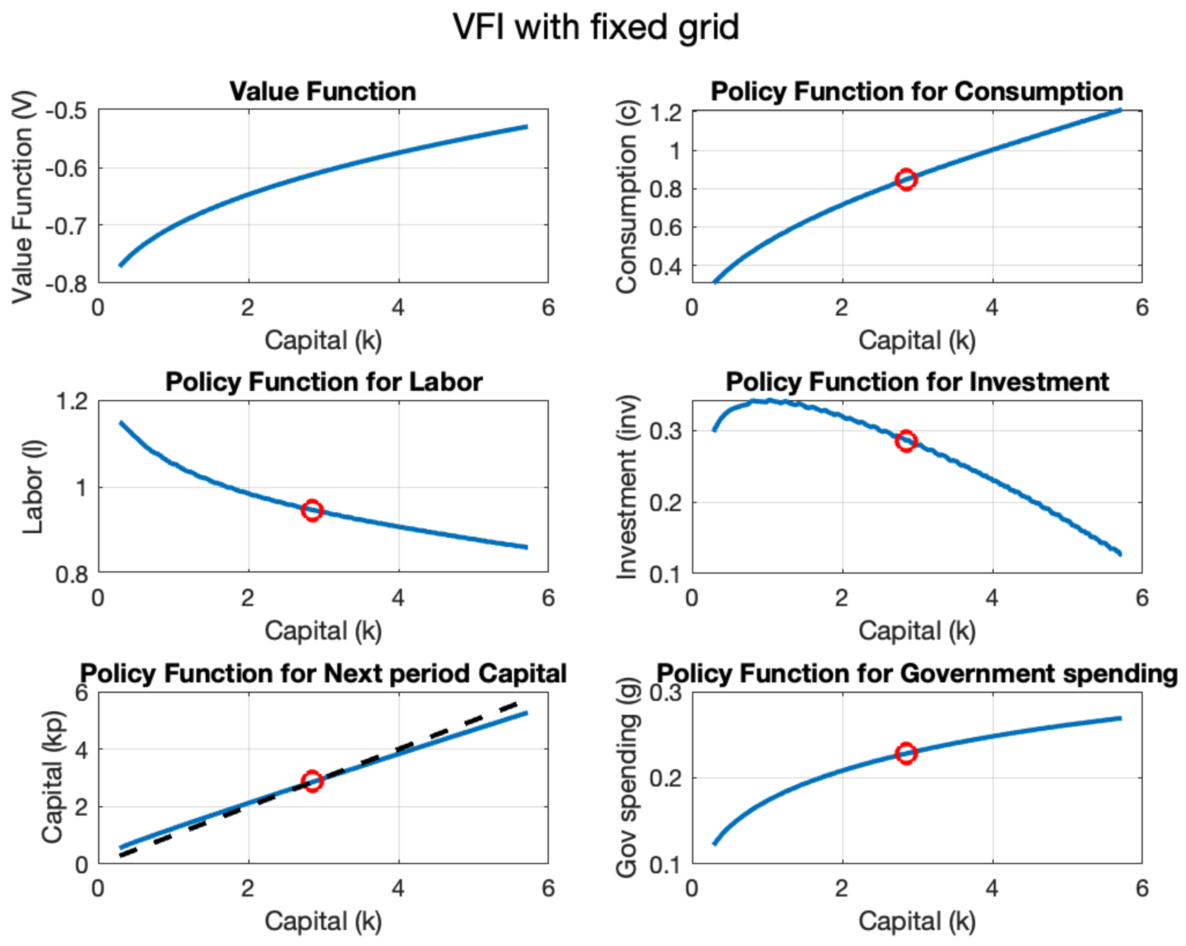
Results

The iterations look like below:

```
-----------------------------------
VFI with fixed grid
-----------------------------------
Iteration 0, Vf error: 0.028104, Accelerate: 0
Iteration 1, Vf error: 0.026108, Accelerate: 0
Iteration 2, Vf error: 0.024218, Accelerate: 0
Iteration 3, Vf error: 0.022426, Accelerate: 0
Iteration 4, Vf error: 0.020731, Accelerate: 0
Iteration 5, Vf error: 0.019124, Accelerate: 0
Iteration 6, Vf error: 0.017601, Accelerate: 0
Iteration 7, Vf error: 0.01616, Accelerate: 0
```

Acceleration means acceleration steps by just iterating previous policy are applied. The `Vf error` is a bit misnamed - they are distance between two consecutive value function matrices.

For a fixed grid, here are the optimal value and policy functions:



The Euler equation errors and running time are (4 parallel workers are used):

```
Elapsed time is 4037.886689 seconds.
====================================
Percentiles of the Euler equation errors:
Min: -1.0226
25th percentile: -0.079314
50th percentile: 0.0096814
75th percentile: 0.039667
Max: 0.059251
Mean Euler equation error: -0.050344
====================================
```

We can see that this is very slow.

VFI with fixed grid with acceleration (switching between VFI and PFI)

This is simply setting `obj.nskip = 10` in my code.

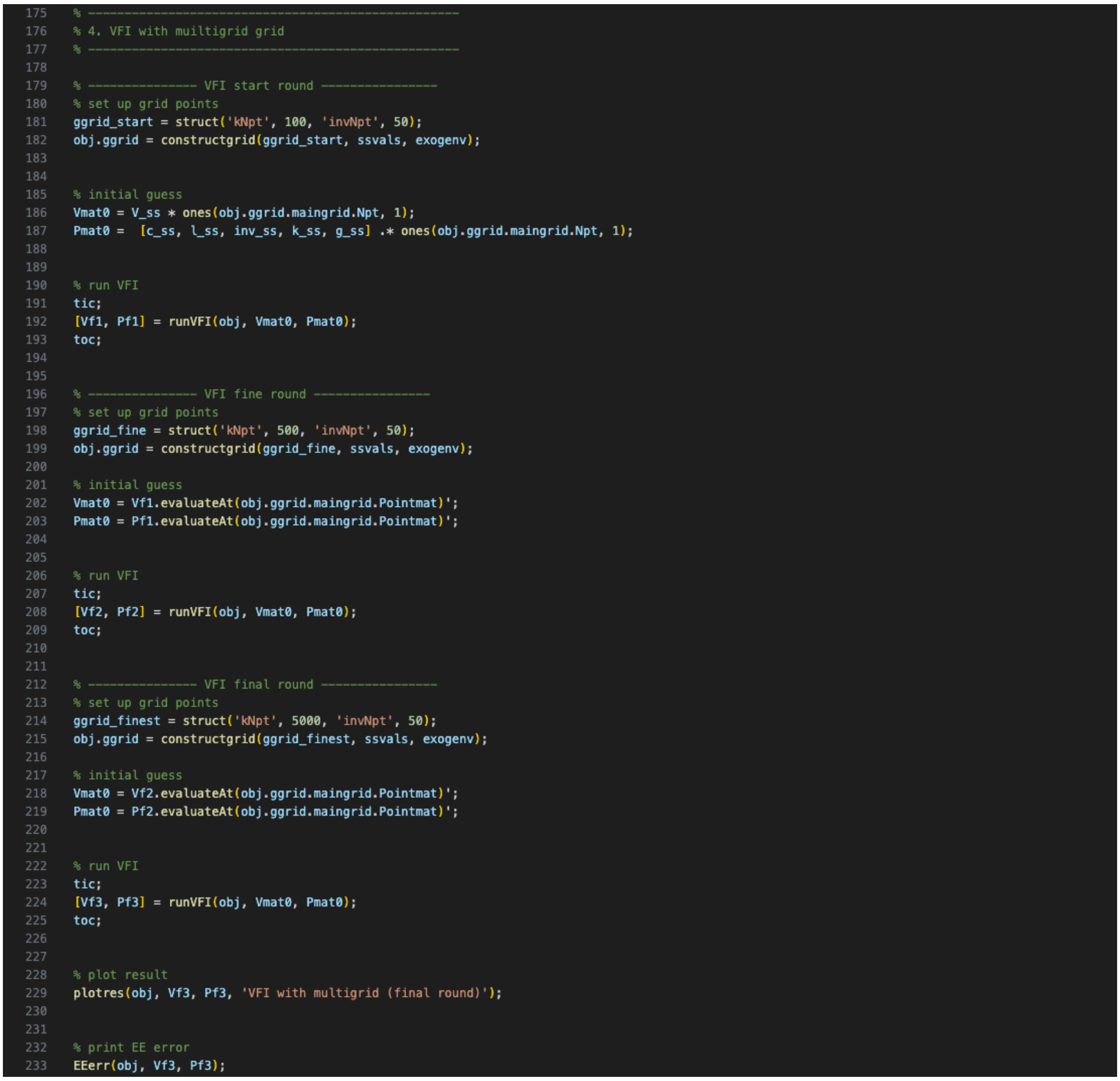
```
Iteration 0, Vf error: 0.028104, Accelerate: 0
Iteration 1, Vf error: 0.026108, Accelerate: 1
Iteration 2, Vf error: 0.024217, Accelerate: 1
Iteration 3, Vf error: 0.022426, Accelerate: 1
Iteration 4, Vf error: 0.02073, Accelerate: 1
Iteration 5, Vf error: 0.019123, Accelerate: 1
Iteration 6, Vf error: 0.017601, Accelerate: 1
Iteration 7, Vf error: 0.016159, Accelerate: 1
Iteration 8, Vf error: 0.014792, Accelerate: 1
Iteration 9, Vf error: 0.013498, Accelerate: 1
Iteration 10, Vf error: 0.019636, Accelerate: 0
Iteration 11, Vf error: 0.01842, Accelerate: 1
Iteration 12, Vf error: 0.01657, Accelerate: 1
Iteration 13, Vf error: 0.01036, Accelerate: 1
```

Results are the same, while convergence is much faster:

```
Iteration 290, Vf error: 0.001527, Accelerate: 0
Iteration 291, Vf error: 0.00010511, Accelerate: 1
Iteration 292, Vf error: 1.0013e-06, Accelerate: 1
Iteration 293, Vf error: 9.6999e-07, Accelerate: 1
Elapsed time is 647.149231 seconds.
```

VFI with multi-grid scheme

I implement the three rounds scheme as follows, where the last round uses 5000 grid points for capital.

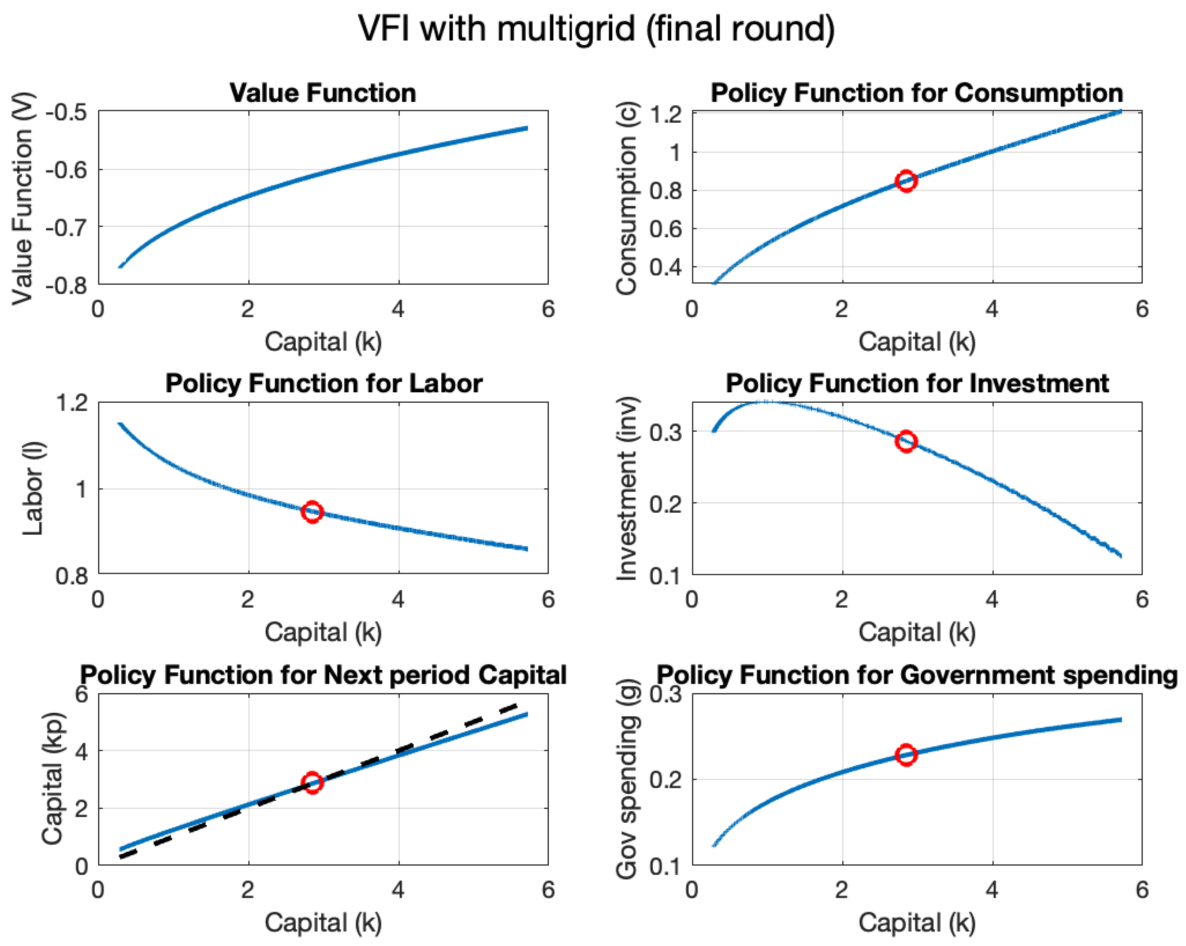


```
% --------------------------------------------------
% 4. VFI with muiltigrid grid
% --------------------------------------------------

% -------------- VFI start round ---------------
% set up grid points
ggrid_start = struct('kNpt', 100, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_start, ssvals, exogenv);


% initial guess
Vmat0 = V_ss * ones(obj.ggrid.maingrid.Npt, 1);
Pmat0 =  [c_ss, l_ss, inv_ss, k_ss, g_ss] .* ones(obj.ggrid.maingrid.Npt, 1);


% run VFI
tic;
[Vf1, Pf1] = runVFI(obj, Vmat0, Pmat0);
toc;


% -------------- VFI fine round ---------------
% set up grid points
ggrid_fine = struct('kNpt', 500, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_fine, ssvals, exogenv);

% initial guess
Vmat0 = Vf1.evaluateAt(obj.ggrid.maingrid.Pointmat)';
Pmat0 = Pf1.evaluateAt(obj.ggrid.maingrid.Pointmat)';


% run VFI
tic;
[Vf2, Pf2] = runVFI(obj, Vmat0, Pmat0);
toc;


% -------------- VFI final round ---------------
% set up grid points
ggrid_finest = struct('kNpt', 5000, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_finest, ssvals, exogenv);

% initial guess
Vmat0 = Vf2.evaluateAt(obj.ggrid.maingrid.Pointmat)';
Pmat0 = Pf2.evaluateAt(obj.ggrid.maingrid.Pointmat)';


% run VFI
tic;
[Vf3, Pf3] = runVFI(obj, Vmat0, Pmat0);
toc;


% plot result
plotres(obj, Vf3, Pf3, 'VFI with multigrid (final round)');


% print EE error
EEerr(obj, Vf3, Pf3);
```

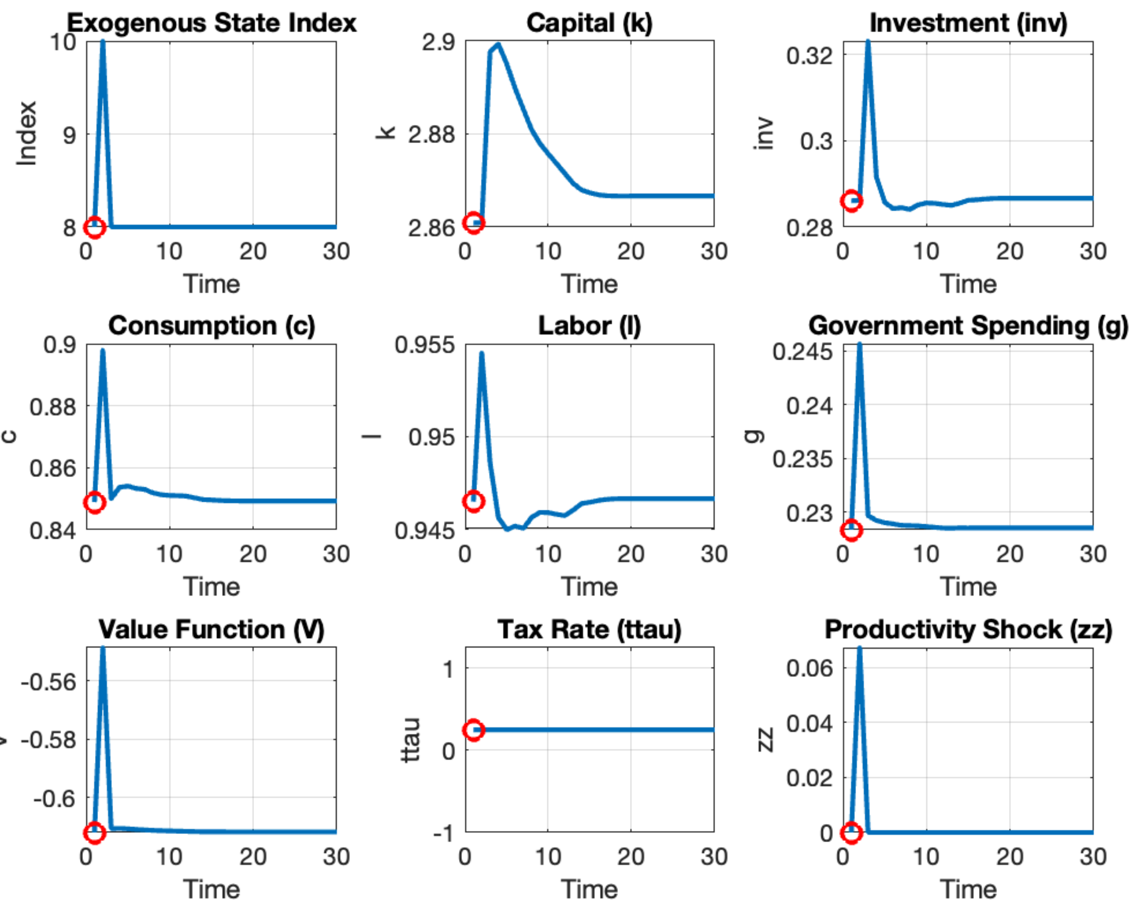
The final round only took 5 iterations to convergence:

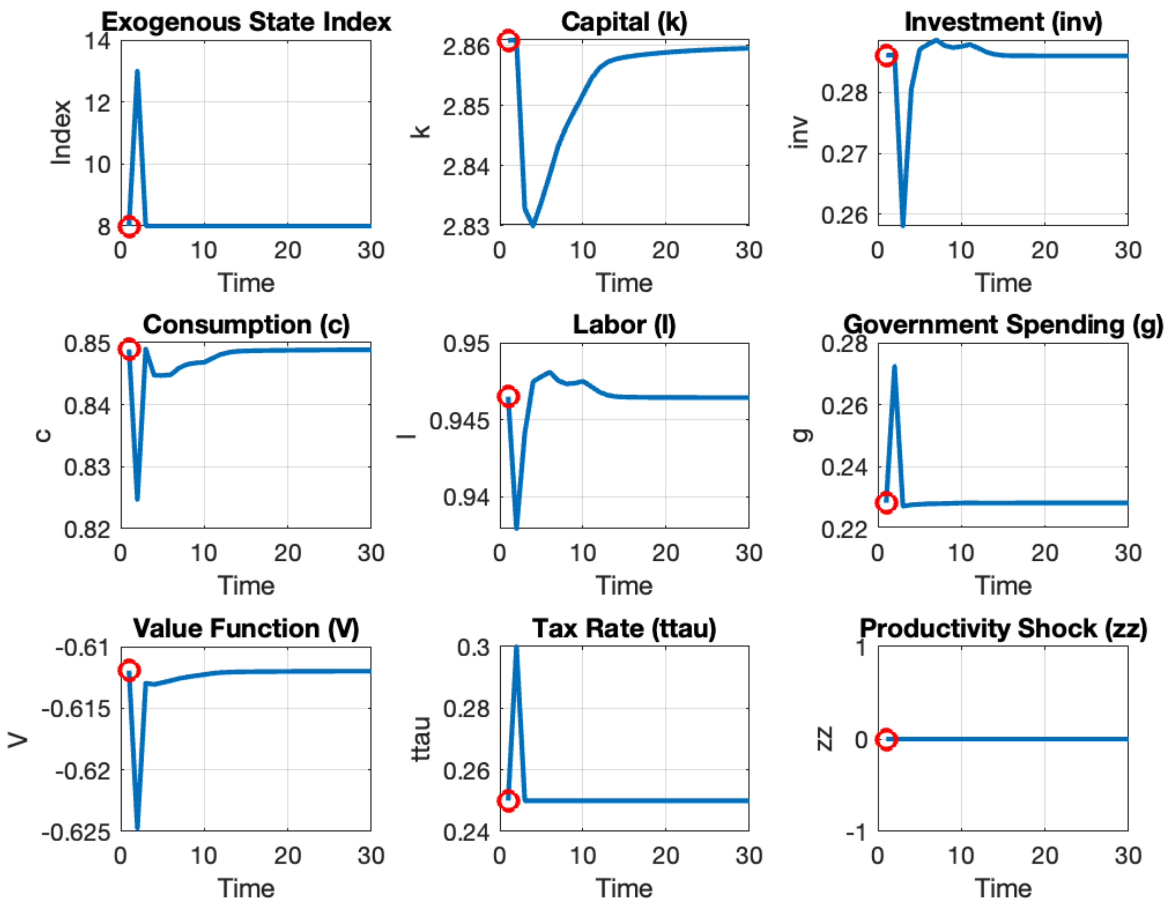
```
Elapsed time is 159.103841 seconds.
Iteration 0, Vf error: 2.7515e-05, Accelerate: 0
Iteration 1, Vf error: 4.1035e-06, Accelerate: 1
Iteration 2, Vf error: 2.4323e-06, Accelerate: 1
Iteration 3, Vf error: 1.6797e-06, Accelerate: 1
Iteration 4, Vf error: 1.3744e-06, Accelerate: 1
Iteration 5, Vf error: 1.211e-06, Accelerate: 1
Iteration 6, Vf error: 1.076e-06, Accelerate: 1
Iteration 7, Vf error: 9.7619e-07, Accelerate: 1
Elapsed time is 472.882931 seconds.
```

The results:



I plot the impulse response functions (IRFs) to a positive shock to tax and a positve shock to capital (while keeping them fixed at the mean values after the first shock period), given these optimal value and policy functions:



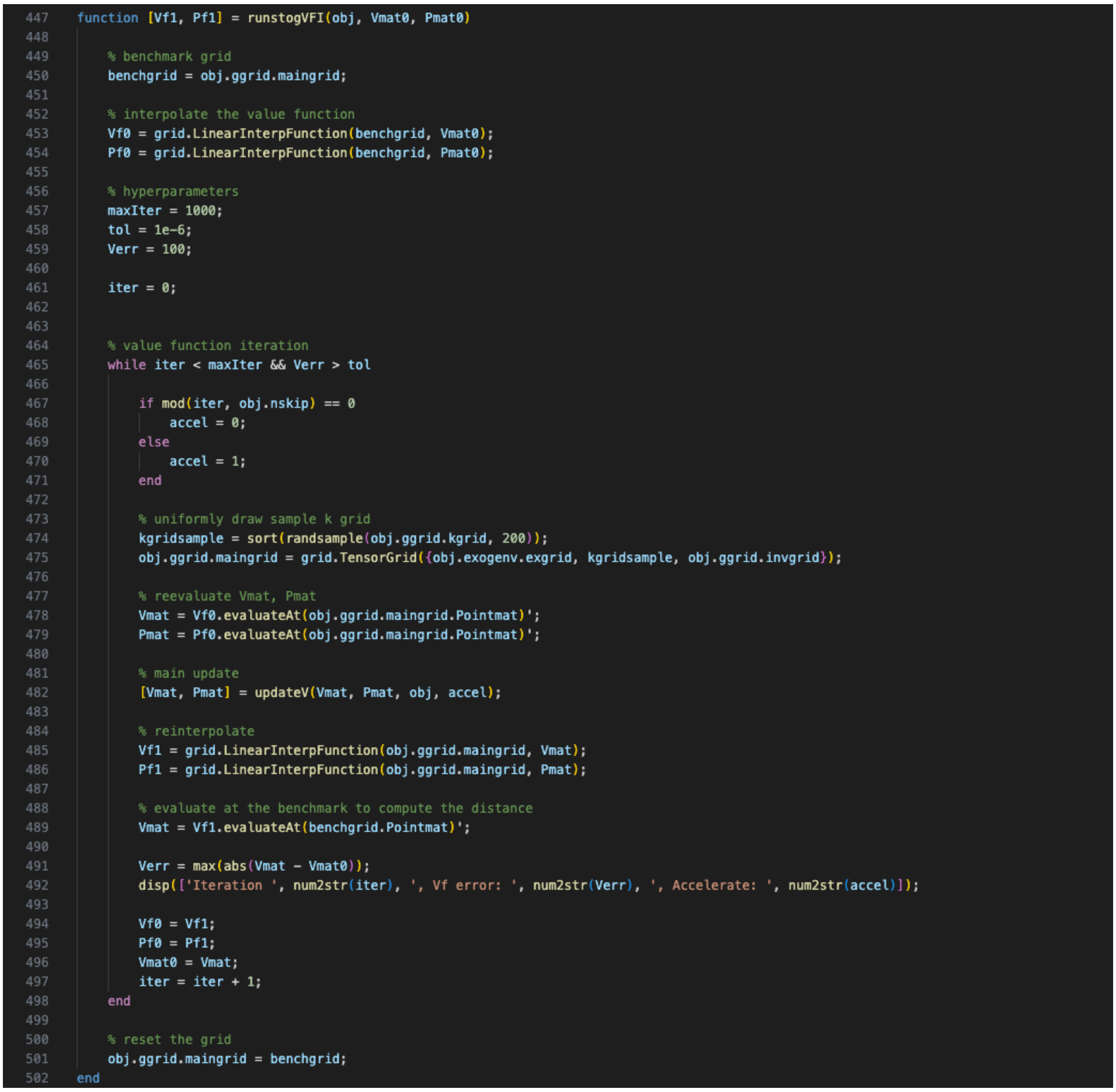


The way how variables move mostly follow our intuition.

VFI with stochastic grid

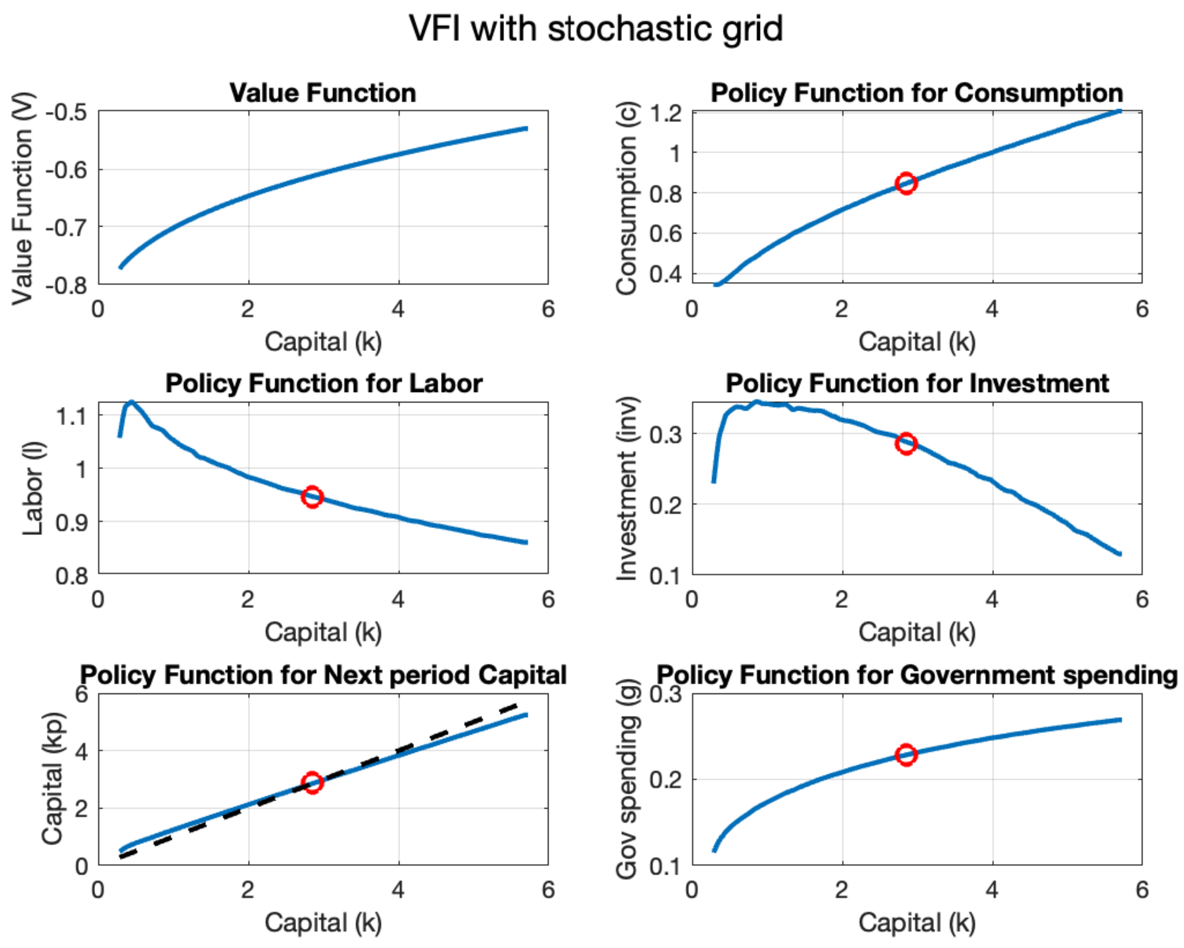
For the stochastic grid method, I uniformly draw a subsample (200) of capital grid points (combined with the entire investment and exogneous state grids) to solve and update the value function in each iteration. To make sure that we keep track of previous progress, the value function over unselected grid points are kept intact, while the rest are updated. After each iteration. The value function are re-interpolated and evaluated at a benchmark grid points to measure the convergence.

The codes are adapted as follows:



```
function [Vf1, Pf1] = runstogVFI(obj, Vmat0, Pmat0)

    % benchmark grid
    benchgrid = obj.ggrid.maingrid;

    % interpolate the value function
    Vf0 = grid.LinearInterpFunction(benchgrid, Vmat0);
    Pf0 = grid.LinearInterpFunction(benchgrid, Pmat0);

    % hyperparameters
    maxIter = 1000;
    tol = 1e-6;
    Verr = 100;

    iter = 0;


    % value function iteration
    while iter < maxIter && Verr > tol

        if mod(iter, obj.nskip) == 0
            accel = 0;
        else
            accel = 1;
        end

        % uniformly draw sample k grid
        kgridsample = sort(randsample(obj.ggrid.kgrid, 200));
        obj.ggrid.maingrid = grid.TensorGrid({obj.exogenv.exgrid, kgridsample, obj.ggrid.invgrid});

        % reevaluate Vmat, Pmat
        Vmat = Vf0.evaluateAt(obj.ggrid.maingrid.Pointmat)';
        Pmat = Pf0.evaluateAt(obj.ggrid.maingrid.Pointmat)';

        % main update
        [Vmat, Pmat] = updateV(Vmat, Pmat, obj, accel);

        % reinterpolate
        Vf1 = grid.LinearInterpFunction(obj.ggrid.maingrid, Vmat);
        Pf1 = grid.LinearInterpFunction(obj.ggrid.maingrid, Pmat);

        % evaluate at the benchmark to compute the distance
        Vmat = Vf1.evaluateAt(benchgrid.Pointmat)';

        Verr = max(abs(Vmat - Vmat0));
        disp(['Iteration ', num2str(iter), ', Vf error: ', num2str(Verr), ', Accelerate: ', num2str(accel)]);

        Vf0 = Vf1;
        Pf0 = Pf1;
        Vmat0 = Vmat;
        iter = iter + 1;
    end

    % reset the grid
    obj.ggrid.maingrid = benchgrid;
end
```

Here are the results:



This is taking relatively long time compared to previous methods. And the value function errors remain quite large:

```
Iteration 995, Vf error: 0.011776, Accelerate: 1
Iteration 996, Vf error: 0.014498, Accelerate: 1
Iteration 997, Vf error: 0.0014818, Accelerate: 1
Iteration 998, Vf error: 0.00030063, Accelerate: 1
Iteration 999, Vf error: 0.00054393, Accelerate: 1
Elapsed time is 1749.765292 seconds.
====================================
Percentiles of the Euler equation errors:
Min: -1.6393
25th percentile: -0.083923
50th percentile: 0.008592
75th percentile: 0.039023
Max: 0.071202
Mean Euler equation error: -0.052811
====================================
```

6.4 Value function iteration with an endogenous grid

To solve the problem using the generalized Endogenous Grid method following Barillas and Fernandez-Villaverde (2006), we follow the two steps:

1. Solve the VFI while fixing $l = \bar{l}$ using the endogenous grid method.
2. Solve the original VFI from the solution of step 1 as an initial guess.

To reformulate the problem, we define a new state variable

$$Y = \psi e^z k^\alpha \bar{l}^{1-\alpha}$$

The recasted problem is

$$\tilde{V}\left(Y, i; \tau, z\right) = \max_{i',l} \left(1-\beta\right) \left[\log\left(Y - i'\right) + \eta \log\left(g\right) - \frac{l^2}{2}\right] + \beta \mathbb{E}_{\tau', z' | \tau, z} \left[\tilde{V}\left(Y', i'; \tau', z'\right)\right]$$
$$\text{where } Y' = \psi e^{z'} k'^{\alpha} \bar{l}^{1-\alpha}$$
$$k' = (1-\delta)k + \left(1 - \phi\left(\frac{i'}{i} - 1\right)^2\right) i'$$

where we used the inversion:

$$Y = \psi e^{z} k^{\alpha} \bar{l}^{1-\alpha} \iff k = \left(\frac{Y}{\psi e^{z}}\right)^{1/\alpha} \bar{l}^{\frac{\alpha-1}{\alpha}}$$

Now, define

$$\hat{V}(Y, i', \tau, z) \equiv \mathbb{E}_{\tau', z' | \tau, z} \left[\tilde{V}\left(Y', i'; \tau', z'\right)\right]$$

where the RHS can be evaluated knowing today's state and i'.

We know the first-order condition:

$$\frac{1}{Y - i'} = \frac{\beta}{1-\beta} \frac{\partial \hat{V}(Y, i', \tau, z)}{\partial i'}$$

Algorithm of EGM

- To proceed, given a value function, on each grid point (τ, z, k, i):
  1. For each i', compute k' and evaluate $\hat{V}(Y, i', \tau, z)$;
  2. Approximate the derivate $\frac{\partial \hat{V}(Y, i'; \tau, z)}{\partial i'}$ using a local finite difference method;;
  3. Compute corresponding Y today, which also gives us

     $$k = \left(\frac{Y}{\psi e^{z} \bar{l}^{1-\alpha}}\right)^{\frac{1}{\alpha}}$$

     and thus we can evaluate $(1-\beta)\left[\log\left(Y - i'\right) + \eta \log\left(g\right) - \frac{\bar{l}^2}{2}\right]$. Finally,

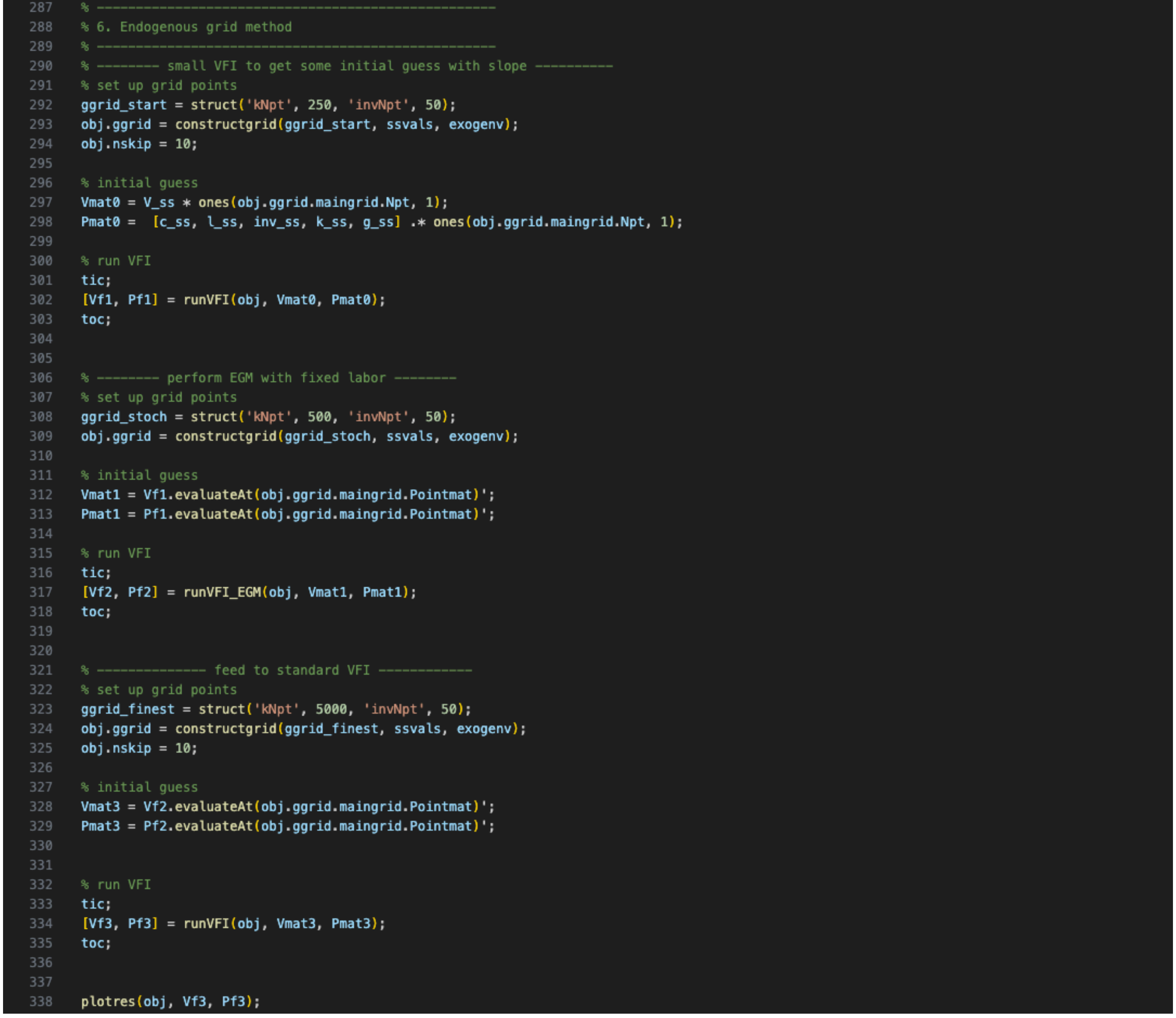
     $$\begin{aligned} \tilde{V}(k, \hat{i}; \hat{\tau}, \hat{z}) &= \tilde{V}\left(Y, \hat{i}; \hat{\tau}, \hat{z}\right) \\ &= (1-\beta)\left[\log\left(Y - i'(k)\right) + \eta \log\left(g\right) - \frac{\bar{l}^2}{2}\right] \\ &+ \beta \hat{V}(Y, i'(k); \tau, z) \end{aligned}$$

- Update the value function:
  1. Reinterpolate $\tilde{V}(k, \hat{i}, \hat{\tau}, \hat{z})$ over fixed $(\hat{i}, \hat{\tau}, \hat{z})$ and new endogenous grid points k.
  2. Evaluate the interpolated value function over the base grids and update the relevant range of values over the fixed $(\hat{i}, \hat{\tau}, \hat{z})$. (Do the same for policy function as well).

Codes

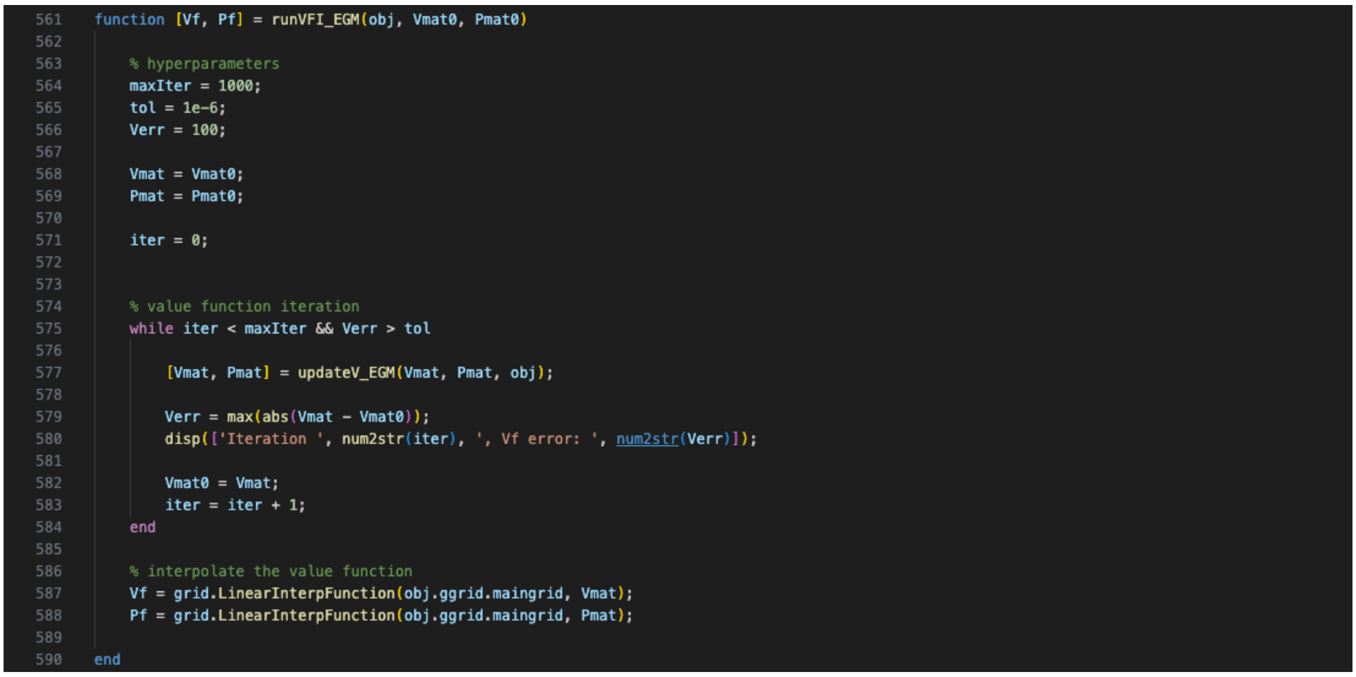
The outer implementation includes three main sections:

- Solving over a coarser grid to get a reasonable initial guess for value function,
- Perform the endogenous grid method with labor fixed at the steady state level,
- Now feed the policy function and value function back to standard VFI, which is expected to converge very fast.

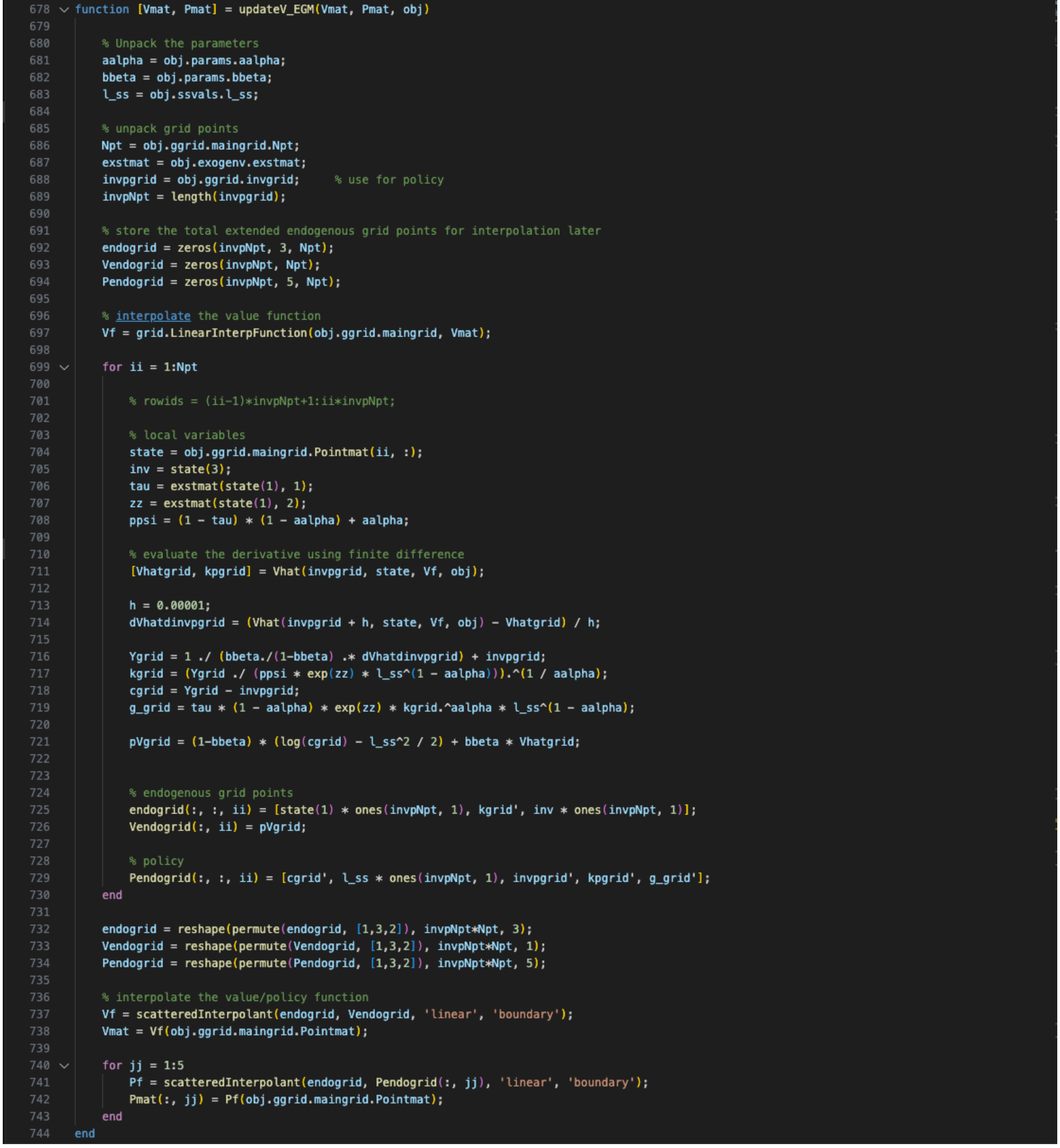


```
% --------------------------------------------------
% 6. Endogenous grid method
% --------------------------------------------------
% -------- small VFI to get some initial guess with slope ----------
% set up grid points
ggrid_start = struct('kNpt', 250, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_start, ssvals, exogenv);
obj.nskip = 10;

% initial guess
Vmat0 = V_ss * ones(obj.ggrid.maingrid.Npt, 1);
Pmat0 =  [c_ss, l_ss, inv_ss, k_ss, g_ss] .* ones(obj.ggrid.maingrid.Npt, 1);

% run VFI
tic;
[Vf1, Pf1] = runVFI(obj, Vmat0, Pmat0);
toc;


% -------- perform EGM with fixed labor --------
% set up grid points
ggrid_stoch = struct('kNpt', 500, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_stoch, ssvals, exogenv);

% initial guess
Vmat1 = Vf1.evaluateAt(obj.ggrid.maingrid.Pointmat)';
Pmat1 = Pf1.evaluateAt(obj.ggrid.maingrid.Pointmat)';

% run VFI
tic;
[Vf2, Pf2] = runVFI_EGM(obj, Vmat1, Pmat1);
toc;


% -------------- feed to standard VFI ------------
% set up grid points
ggrid_finest = struct('kNpt', 5000, 'invNpt', 50);
obj.ggrid = constructgrid(ggrid_finest, ssvals, exogenv);
obj.nskip = 10;

% initial guess
Vmat3 = Vf2.evaluateAt(obj.ggrid.maingrid.Pointmat)';
Pmat3 = Pf2.evaluateAt(obj.ggrid.maingrid.Pointmat)';


% run VFI
tic;
[Vf3, Pf3] = runVFI(obj, Vmat3, Pmat3);
toc;


plotres(obj, Vf3, Pf3);
```

Here are the run function and the updating procedure:

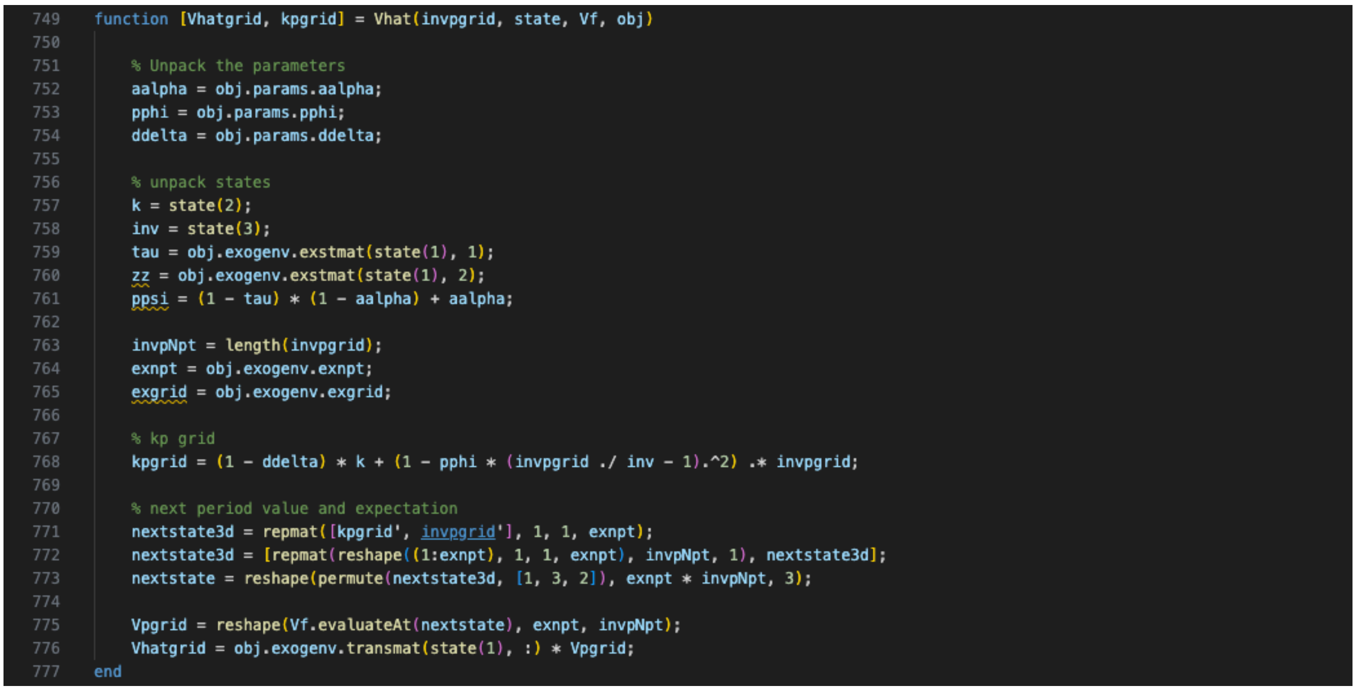


```
function [Vf, Pf] = runVFI_EGM(obj, Vmat0, Pmat0)

    % hyperparameters
    maxIter = 1000;
    tol = 1e-6;
    Verr = 100;

    Vmat = Vmat0;
    Pmat = Pmat0;

    iter = 0;


    % value function iteration
    while iter < maxIter && Verr > tol

        [Vmat, Pmat] = updateV_EGM(Vmat, Pmat, obj);

        Verr = max(abs(Vmat - Vmat0));
        disp(['Iteration ', num2str(iter), ', Vf error: ', num2str(Verr)]);

        Vmat0 = Vmat;
        iter = iter + 1;
    end

    % interpolate the value function
    Vf = grid.LinearInterpFunction(obj.ggrid.maingrid, Vmat);
    Pf = grid.LinearInterpFunction(obj.ggrid.maingrid, Pmat);

end
```



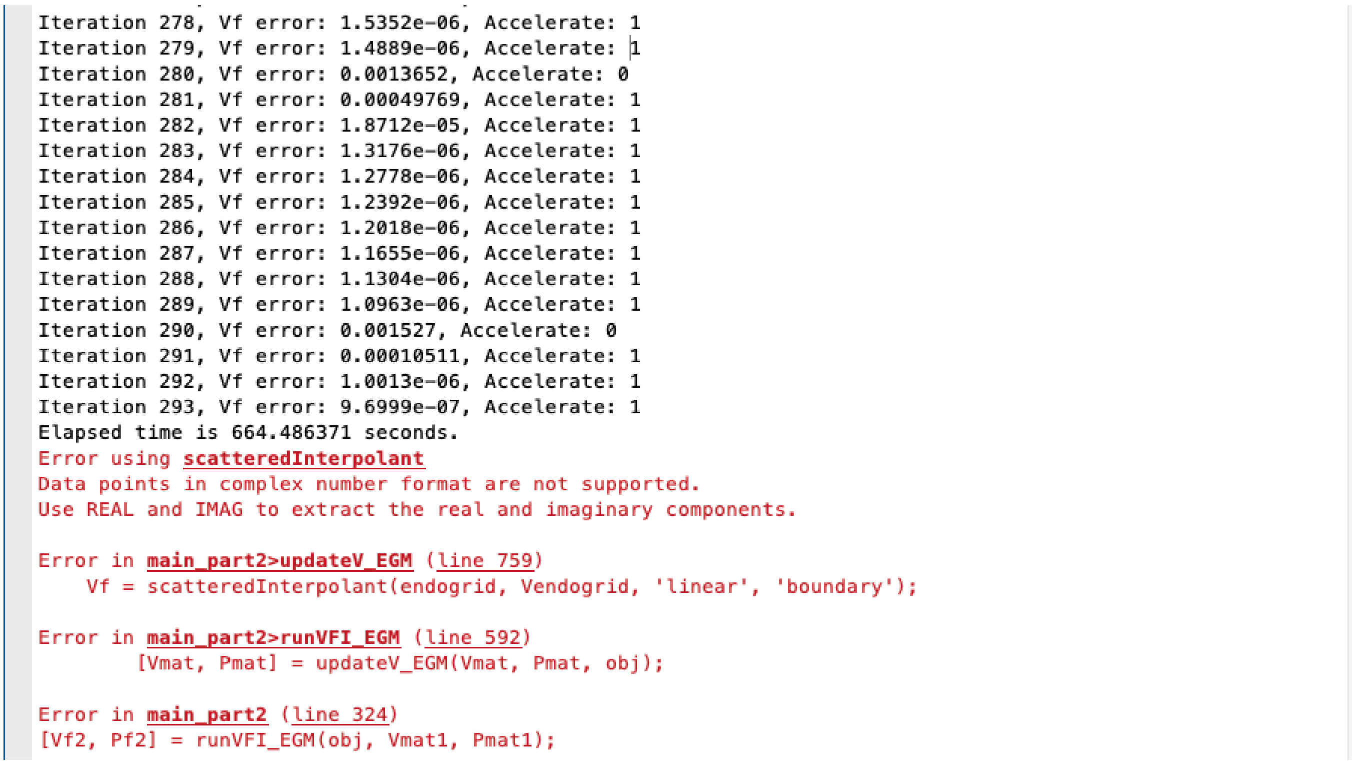
```matlab
function [Vmat, Pmat] = updateV_EGM(Vmat, Pmat, obj)

    % Unpack the parameters
    aalpha = obj.params.aalpha;
    bbeta = obj.params.bbeta;
    l_ss = obj.ssvals.l_ss;

    % unpack grid points
    Npt = obj.ggrid.maingrid.Npt;
    exstmat = obj.exogenv.exstmat;
    invpgrid = obj.ggrid.invgrid;     % use for policy
    invpNpt = length(invpgrid);

    % store the total extended endogenous grid points for interpolation later
    endogrid = zeros(invpNpt, 3, Npt);
    Vendogrid = zeros(invpNpt, Npt);
    Pendogrid = zeros(invpNpt, 5, Npt);

    % interpolate the value function
    Vf = grid.LinearInterpFunction(obj.ggrid.maingrid, Vmat);

    for ii = 1:Npt

        % rowids = (ii-1)*invpNpt+1:ii*invpNpt;

        % local variables
        state = obj.ggrid.maingrid.Pointmat(ii, :);
        inv = state(3);
        tau = exstmat(state(1), 1);
        zz = exstmat(state(1), 2);
        ppsi = (1 - tau) * (1 - aalpha) + aalpha;

        % evaluate the derivative using finite difference
        [Vhatgrid, kpgrid] = Vhat(invpgrid, state, Vf, obj);

        h = 0.00001;
        dVhatdinvpgrid = (Vhat(invpgrid + h, state, Vf, obj) - Vhatgrid) / h;

        Ygrid = 1 ./ (bbeta./(1-bbeta) .* dVhatdinvpgrid) + invpgrid;
        kgrid = (Ygrid ./ (ppsi * exp(zz) * l_ss^(1 - aalpha))).^(1 / aalpha);
        cgrid = Ygrid - invpgrid;
        g_grid = tau * (1 - aalpha) * exp(zz) * kgrid.^aalpha * l_ss^(1 - aalpha);

        pVgrid = (1-bbeta) * (log(cgrid) - l_ss^2 / 2) + bbeta * Vhatgrid;


        % endogenous grid points
        endogrid(:, :, ii) = [state(1) * ones(invpNpt, 1), kgrid', inv * ones(invpNpt, 1)];
        Vendogrid(:, ii) = pVgrid;

        % policy
        Pendogrid(:, :, ii) = [cgrid', l_ss * ones(invpNpt, 1), invpgrid', kpgrid', g_grid'];
    end

    endogrid = reshape(permute(endogrid, [1,3,2]), invpNpt*Npt, 3);
    Vendogrid = reshape(permute(Vendogrid, [1,3,2]), invpNpt*Npt, 1);
    Pendogrid = reshape(permute(Pendogrid, [1,3,2]), invpNpt*Npt, 5);

    % interpolate the value/policy function
    Vf = scatteredInterpolant(endogrid, Vendogrid, 'linear', 'boundary');
    Vmat = Vf(obj.ggrid.maingrid.Pointmat);

    for jj = 1:5
        Pf = scatteredInterpolant(endogrid, Pendogrid(:, jj), 'linear', 'boundary');
        Pmat(:, jj) = Pf(obj.ggrid.maingrid.Pointmat);
    end
end
```

IN particular, a new function `Vhat` is defined that takes both today's state and actions as given, to compute the expected next period value. The finite difference method is applied to approximate the partial derivative with respect to `i'`, that is, (`(Vhat(i' + h)-Vhat(i'))/h`). Here is the `Vhat` function:



```
function [Vhatgrid, kpgrid] = Vhat(invpgrid, state, Vf, obj)

    % Unpack the parameters
    aalpha = obj.params.aalpha;
    pphi = obj.params.pphi;
    ddelta = obj.params.ddelta;

    % unpack states
    k = state(2);
    inv = state(3);
    tau = obj.exogenv.exstmat(state(1), 1);
    zz = obj.exogenv.exstmat(state(1), 2);
    ppsi = (1 - tau) * (1 - aalpha) + aalpha;

    invpNpt = length(invpgrid);
    exnpt = obj.exogenv.exnpt;
    exgrid = obj.exogenv.exgrid;

    % kp grid
    kpgrid = (1 - ddelta) * k + (1 - pphi * (invpgrid ./ inv - 1).^2) .* invpgrid;

    % next period value and expectation
    nextstate3d = repmat([kpgrid', invpgrid'], 1, 1, exnpt);
    nextstate3d = [repmat(reshape((1:exnpt), 1, 1, exnpt), invpNpt, 1), nextstate3d];
    nextstate = reshape(permute(nextstate3d, [1, 3, 2]), exnpt * invpNpt, 3);

    Vpgrid = reshape(Vf.evaluateAt(nextstate), exnpt, invpNpt);
    Vhatgrid = obj.exogenv.transmat(state(1), :) * Vpgrid;
end
```

However, at some point, the implied endogneous grid start to jump out of the grid bound by too much, which eventually leads to negative consumption or capital and complex values. I'm still trying to make this code work.



```
Iteration 278, Vf error: 1.5352e-06, Accelerate: 1
Iteration 279, Vf error: 1.4889e-06, Accelerate: 1
Iteration 280, Vf error: 0.0013652, Accelerate: 0
Iteration 281, Vf error: 0.00049769, Accelerate: 1
Iteration 282, Vf error: 1.8712e-05, Accelerate: 1
Iteration 283, Vf error: 1.3176e-06, Accelerate: 1
Iteration 284, Vf error: 1.2778e-06, Accelerate: 1
Iteration 285, Vf error: 1.2392e-06, Accelerate: 1
Iteration 286, Vf error: 1.2018e-06, Accelerate: 1
Iteration 287, Vf error: 1.1655e-06, Accelerate: 1
Iteration 288, Vf error: 1.1304e-06, Accelerate: 1
Iteration 289, Vf error: 1.0963e-06, Accelerate: 1
Iteration 290, Vf error: 0.001527, Accelerate: 0
Iteration 291, Vf error: 0.00010511, Accelerate: 1
Iteration 292, Vf error: 1.0013e-06, Accelerate: 1
Iteration 293, Vf error: 9.6999e-07, Accelerate: 1
Elapsed time is 664.486371 seconds.
Error using scatteredInterpolant
Data points in complex number format are not supported.
Use REAL and IMAG to extract the real and imaginary components.

Error in main_part2>updateV_EGM (line 759)
    Vf = scatteredInterpolant(endogrid, Vendogrid, 'linear', 'boundary');

Error in main_part2>runVFI_EGM (line 592)
        [Vmat, Pmat] = updateV_EGM(Vmat, Pmat, obj);

Error in main_part2 (line 324)
[Vf2, Pf2] = runVFI_EGM(obj, Vmat1, Pmat1);
```