Advanced Topics Macro I - Assignment 3

Andrea Titton & Alessandro Zona Mattioli

November 2020

Question 1

Both the one- and two-sided derivative method were implemented. First we acquired the machine epsilon as,

```
sq_ϵ = √ eps(Float64)
```

The two algorithms are then constructed as follows,

```
function constructtwosided(f::Function)::Function

    function f_prime(x::Float64)::Float64
        h = h_st(x)
        return (f(x + h) - f(x - h)) / 2h
    end

    function f_prime(x::Array{Float64,1})::Array{Float64,1}
        n = length(x)
        h = h_st(x)

        return [ f(x + h .* e(i, n)) - f(x - h .* e(i, n)) for i in
            1:n ] ./ 2h # Is this the best
            way?
    end

    return f_prime
end

function constructonesided(f::Function)::Function

    function f_prime(x::Float64)::Float64
        h = h_st(x)
        return (f(x + h) - f(x)) / h
    end

    function f_prime(x::Array{Float64,1})::Array{Float64,1}
        n = length(x)
        h = h_st(x)

        return [ f(x + h .* e(i, n)) - f(x) for i in 1:n ] ./ h
    end

    return f_prime
end
```

Note that the two functions distinguish between univariate and multivariate f by looking at the passed argument of the derivative.

We then use this functions to construct the derivatives for the utility,

```
x_grid = collect(range(-2π, 2π, length=100))
```

```
u(c) = - 1 / c
u_p(c) = 1 / c^2

for (name, derivative) in techniques

    df = derivative(f)

    eval_deriv = df.(x_grid)

    error = norm.(analy_der .- eval_deriv)

    errors[name] = multivariate ? sum(error, dims=2) : error
    evals[name] = eval_deriv

end
```

We can plot (Figure 1) the error of the numerical derivative against the analytical one for both methods.

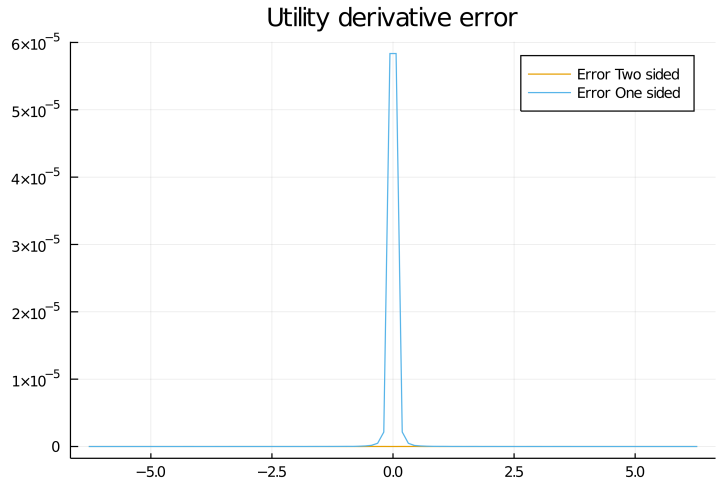

Figure 1: $u(c)$ error between numerical and analytical

The error appears to be particularly high for the one-sided approximation method as $c \to 0$. This is expected since, $u'(c) \xrightarrow{c \to 0} \infty$. On the other hand, the two sided method is more stable.

We then proceed to evaluate the production function, rewriting it as, $F(K, N, H) = G(K, N, H)^{1/\sigma}$. Such that $\nabla F = \frac{1}{\sigma} G(K, N, H)^{(1-\sigma)/\sigma} \cdot \nabla G$.

The production function is then,

```

const μ = 0.05
```

```
const σ = 0.6
const λ = 0.25
const ρ = 0.2

G(K, N, H) = μ * N^σ + (1 - μ) * (λ * K^ρ + (1 - λ) * H^ρ)^(σ / ρ)

F(K::Float64, N::Float64, H::Float64) = G(K, N, H)^(1 / σ)
function F(v::Array{Float64})
    K, N, H = v
    return F(K, N, H)
end

"""
The analytical derivative of the production function
"""
function F_p(K::Float64, N::Float64, H::Float64)

    outer = (1 / σ) * G(K, N, H)^((1 - σ) / σ)
    outer_kh = (σ * (1 - μ) / ρ) * (λ * K^ρ + (1 - λ) * H^ρ)^(-1 + σ
         / ρ)

    return outer * [
        outer_kh * λ * ρ * K^(ρ - 1),
        σ * μ * N^(σ - 1),
        outer_kh * (1 - λ) * H^(ρ - 1)
    ]

end
```

Then we can easily evaluate the function at,

```
grid = range(0.01, .99, length=10)
mul_grid = collect.(Iterators.product(grid, grid, grid))
```

The result of the evaluation is not presented here (since it is a $10 \times 10 \times 10$ matrix) but the evaluation is retrieved running test-diff.jl. Or by simply doing,

```
∇ F = constructtwosided(F).(grid)
```

Question 2

We start by solving the model analytically, using a value function representation:

$$V(k) = \max_{k'} \log(e^z \cdot k^\alpha - k') + \beta \cdot V(k') \tag{1}$$

Similarly to the slides, we guess that $V(k) = a + b \cdot \log(k)$ and this allows us to

retrieve:

$$\begin{aligned} k'(k) &= \alpha\beta \cdot e^z k^\alpha \\ c(k) &= (1-\alpha\beta) \cdot e^z k^\alpha \end{aligned} \tag{2}$$

We will use these functions also to compute the Euler errors, as we explain later. We now move to the second method, laying down the Lagrangean:

$$\Lambda = \sum_t \beta^t \left[log(c_t) + \lambda_t (e^{z_t} k_t^\alpha - c_t - k_{t+1}) \right] \tag{3}$$

and deriving the FOC:

$$\begin{aligned} FOC_{c_t} &: \ \frac{1}{c_t} = \lambda_t \\ FOC_{k_t} &: \ \lambda_t = \beta \lambda_{t+1} \alpha e^{z_t} k_t^{\alpha-1} \\ &\Rightarrow \frac{1}{c_t} = \beta\alpha e^{z_t} E\left[\frac{k_{t+1}^{\alpha-1}}{c_{t+1}} \right] \end{aligned} \tag{4}$$

where the last result is the the Euler equation. Therefore our model is characterized by:

$$\begin{aligned} c_t + k_{t+1} &= e^{z_t} k_t^\alpha \\ z_{t+1} &= \rho z_t + \sigma \epsilon_t \\ \frac{1}{c_t} &= \beta\alpha e^{z_t} E\left[\frac{k_{t+1}^{\alpha-1}}{c_{t+1}} \right] \end{aligned} \tag{5}$$

We now solve for the values of k_t and c_t in steady state, i.e. $(\bar{k}, \bar{c})$. First, we define:

$$\begin{aligned} \xi_t = \sigma\varepsilon_t \ &: \ \xi_t \sim N(0, \sigma^2) \\ \exp(z_t) = y_t \ &: \ E(y_t) = \exp\left(\frac{\sigma^2}{2(1-\rho^2)} \right) = \bar{y} \end{aligned} \tag{6}$$

Therefore from the Euler equation and budget constraint in steady state we can derive:

$$\begin{aligned} \frac{1}{\bar{c}} &= \beta\alpha\bar{y}\frac{\bar{k}^{\alpha-1}}{\bar{c}} \Rightarrow \bar{k} = (\alpha\beta\bar{y})^{\frac{1}{1-\alpha}} \\ \bar{c} &= \bar{y}\bar{k}^{\alpha} - \bar{k} \end{aligned} \tag{7}$$

Once we have derived the variables in steady state, we log-linearize the characterizing equations. In general, we denote:

$$\hat{x} = \ln\left(\frac{x}{\bar{x}}\right) \approx \frac{x-\bar{x}}{\bar{x}} \Rightarrow x = \bar{x}\cdot\exp\left(\hat{x}\right) \tag{8}$$

Therefore the log-linearized budget constraint becomes:

$$\begin{aligned} \bar{c}\cdot\exp(\hat{c}_t) + \bar{k}\cdot\exp(\hat{k}_{t+1}) &= \bar{y}\cdot\bar{k}^{\alpha}\exp(\hat{y}_t + \alpha\cdot\hat{k}_t) \\ \bar{c}\cdot\hat{c}_t + \bar{k}\cdot\hat{k}_{t+1} &= \bar{y}\cdot\bar{k}^{\alpha}(\hat{y}_t + \alpha\cdot\hat{k}_t) \end{aligned} \tag{9}$$

Concerning the Euler equation, if we assume perfect foresight:

$$\begin{aligned} \bar{c}\cdot\exp(-\hat{c}_t) &= \alpha\beta\cdot\bar{y}\cdot\exp(\hat{y}_t)\cdot\bar{c}\cdot\exp(-\hat{c}_{t+1})\cdot\bar{k}^{\alpha-1}\exp((\alpha-1)\cdot\hat{k}_{t+1}) \\ \hat{c}_{t+1} - \hat{c}_t &= \hat{y}_t + (\alpha-1)\cdot\hat{k}_{t+1} \end{aligned} \tag{10}$$

Otherwise, without assuming perfect foresight and including some approximation error in the expected value:

$$E[\hat{c}_{t+1} + (1-\alpha)\hat{k}_{t+1}] = \hat{y}_t + \hat{c}_t \tag{11}$$

Finally, concerning the evolution of technology:

$$\begin{aligned} \bar{y}\cdot\exp(\hat{y}_{t+1}) &= \rho\cdot\bar{y}\exp(\rho\hat{y}_t)\cdot\bar{xi}\cdot\exp(\hat{xi}_t) \\ \hat{y}_{t+1} &= \rho\hat{y}_t + \hat{\xi}_{t+1} \end{aligned} \tag{12}$$

Putting the equations together we can represent the system in matrix form and solve:

$$\begin{bmatrix} 1 & 1-\alpha \\ 0 & \bar{k} \end{bmatrix} \begin{bmatrix} \hat{c}_{t+1} \\ \hat{k}_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\bar{c} & \bar{y} \cdot \bar{k}^{\alpha}\alpha \end{bmatrix} \begin{bmatrix} \hat{c}_t \\ \hat{k}_t \end{bmatrix} + \begin{bmatrix} 1 \\ \bar{y} \cdot \bar{k}^{\alpha} \end{bmatrix} \hat{y}_{t+1}$$
$$A \cdot \hat{x}_{t+1} = B \cdot \hat{x}_t + C \cdot \hat{y}_{t+1} \tag{13}$$
$$\hat{x}_{t+1} = A^{-1} \cdot B \cdot \hat{x}_t + A^{-1} \cdot C \cdot \hat{y}_{t+1}$$

An alternative method for this point is given by the implicit function theorem, which we also test. We rewrite the problem as a system of functional equations $F(k_t, y_t)$:

$$\begin{cases} 0 &= c(k_t, y_t) + k(k_t, y_t) - y_t \cdot k_t^{\alpha} \\ 0 &= E\left[1 - \dfrac{c(k_t, y_t)}{c(k(k_t, y_t), y_t^{\rho} \cdot \xi)} \cdot \alpha\beta y_t k(k_t, y_t)^{\alpha-1}\right] \end{cases} \tag{14}$$

Then we move on to compute $F_k(k_t, y_t)$:

$$\begin{aligned} F_{k1} &= c_k + k_k - \alpha \cdot y_t \cdot k_t^{\alpha-1} \\ F_{k2} &= \alpha\beta \cdot k(k_t, y_t)^{\alpha-1} \cdot \frac{-c_k \cdot c(k(k_t, y_t), y_t^{\rho} \cdot \xi_t) + k_k \cdot c_k \cdot c(k_t, y_t)}{c(k(k_t, y_t), y_t^{\rho} \cdot \xi)^2} \cdot y_t - \\ &\quad - \frac{c(k_t, y_t)}{c(k(k_t, y_t), y_t^{\rho} \cdot \xi)} \cdot \alpha(\alpha-1) \cdot \beta \cdot k(k_t, y_t)^{\alpha-2} \cdot y_t \end{aligned} \tag{15}$$

and $F_y(k_t, y_t)$:

$$\begin{aligned} F_{y1} &= c_y + k_y - k_t^{\alpha} \\ F_{y2} &= \frac{(c_k \cdot k_y + \rho \cdot c_k y_t^{\rho-1} \cdot \xi_t) \cdot c(k_t, y_t) - c_y \cdot c(k(k_t, y_t), y_t^{\rho} \cdot \xi)}{(c(k(k_t, y_t), y_t^{\rho} \cdot \xi)^2} \cdot \alpha\beta y_t \cdot k_t^{\alpha-1} - \\ &\quad - \frac{c(k_t, y_t)}{c(k(k_t, y_t), y_t^{\rho} \cdot \xi)} \alpha\beta k_t^{\alpha-1} \end{aligned} \tag{16}$$

Now if we evaluate the derivatives in the point $(\bar{k}, 0, 0)$ we retrieve a system of four equations that we can solve for c_k, k_k, c_y, k_y, which we can in turn use to approximate the policy functions using a first order Taylor expansion. After some derivations we obtain the following set of conditions:

$$
\begin{aligned}
0 &= c_k \cdot k_k + (\rho - 1) \cdot c_y - \bar{c} \\
0 &= c_y + k_y - \bar{k}^{\alpha} \\
0 &= c_k + k_k - \alpha \bar{k}^{\alpha - 1} \\
0 &= (k_k - 1) \cdot c_k \cdot \bar{k}^{\alpha - 1} - k_k^{\alpha - 2} \cdot (\alpha - 1) \cdot \bar{c}
\end{aligned}
\tag{17}
$$

We will plug in these conditions in the computer so that we are finally able to plot the policy function and the Euler equation error.

Concerning the quadratic approximation method, we rely on the Julia equivalent of Dynare: Dolo. More in general, we execute all computations via the run_three.jl script and all files for the computations and graphs are stored in the week-three folder of our repository. For Question 2 the main file is stoch_growth.jl. It first simulates a commong path for z_t and then starts comparing policy functions and Euler error:

```
1
2  function compare_methods(model; bounds=[0.01, 0.7], n_steps=500)
3      quad_eee, tab = pert_eee(model; bounds=bounds, n_steps=n_steps)
4      k_space = tab[:k]
5      quad_policy = tab[:c]
6
7      y_ss, k_ss, c_ss = analytical_ss(model)
8      analy_policy = analytical_policy(model)[1].(k_space, y_ss)
9      analy_eee = analytical_eee(model).(k_space, y_ss)
10
11     c_imp_pol, k_imp_pol = implicit_policy(model)
12     c_imp_eee = eee(c_imp_pol, k_imp_pol, model).(k_space, y_ss)
13
14     plot(title="Euler equation errors", dpi=800, xaxis="k", yaxis="
           EEE(k)")
15
16     plot!(k_space, analy_eee, label="Analytical")
17     plot!(k_space, quad_eee, label="Quad. pert.")
18     plot!(k_space, c_imp_eee, label="Imp. Fn. Th.")
19
20
21     savefig("plot_path/sgm_comp/eee_comparison.png")
22
23     plot(title="Policy", dpi=800, xaxis="k", yaxis="c(k)")
24     plot!(k_space, analy_policy, label="Analytical")
25     plot!(k_space, quad_policy, label="Quad. pert.")
26     plot!(k_space, c_imp_pol.(k_space, y_ss), label="Imp. Fn. Th.")
27
28     savefig("plot_path/sgm_comp/policy_comparison.png")
29
30 end
```

Where analytical solutions (steady states, policy functions and Euler errors) are sourced from the solve-sgm/analytical.jl:

```
function analytical_policy(model)
    parameters = model.calibration.flat

    α, β = parameters[:alpha], parameters[:beta]
    ρ, σ = parameters[:rho], parameters[:sigma]

    k_policy(k, y) = α * β * y * k^α
    c_policy(k, y) = y * k^α - k_policy(k, y)

    return c_policy, k_policy
end

function analytical_ss(model)
    parameters = model.calibration.flat

    α, β = parameters[:alpha], parameters[:beta]
    ρ, σ = parameters[:rho], parameters[:sigma]

    y = exp(0.5 * σ^2 / (1 - ρ^2))
    k = (α * β * y)^(1 / (1 - α))
    c = y * k^α - k

    return y, k, c

end

function analytical_eee(model)
    parameters = model.calibration.flat

    α, β = parameters[:alpha], parameters[:beta]
    ρ, σ = parameters[:rho], parameters[:sigma]

    c_p, k_p = analytical_policy(model)

    f(k, y) = β * (α * y * k_p(k, y)^(α - 1) ) / c_p(k_p(k, y), y)

    return (k, y) -> log10(abs(1 - f(k, y)))
end
```

And the solutions for the log-linearized system and implicit function method are sourced from solve-sgm/log.jl" and solve-sgm/eee.jl. Below we report the implicit_policy function and the general eee function that we use also to plot policy and Euler errors:

```

function implicit_policy(model)
    parameters = model.calibration.flat

    α, β = parameters[:alpha], parameters[:beta]
```

```
    ρ, σ = parameters[:rho], parameters[:sigma]

    y_ss, k_ss, c_ss = analytical_ss(model)


    function f!(F, x)
        x, y, z, w = x
        F[1] = x * y + (ρ - 1) * z - c_ss
        F[2] = z + w - k_ss^α
        F[3] = x + y - α * k_ss^(α - 1)
        F[4] = (y - 1) * x * k_ss^(α - 1) - y^(α - 2) * (α - 1) *
            c_ss
    end

    init_x = [0.5, 0.5, 0.5, 0.5]
    res = mcpsolve(f!, [0., 0., 0., 0.], [Inf, Inf, Inf, Inf], init_x
        )
    c_k, k_k, c_y, k_y = res.zero

    c_policy(k, y) = max(c_ss + c_k * (k - k_ss) / k_ss + c_y * (y -
        y_ss) / y_ss, 0.01)
    k_policy(k, y) = max(k_ss + k_k * (k - k_ss) / k_ss + k_y * (y -
        y_ss) / y_ss, 0.01)


    return c_policy, k_policy
end

function eee(c_p, k_p, model)

    parameters = model.calibration.flat

    α, β = parameters[:alpha], parameters[:beta]
    ρ, σ = parameters[:rho], parameters[:sigma]

    f(k, y) = β * (α * y * k_p(k, y)^(α - 1) ) / c_p(k_p(k, y), y)

    return (k, y) -> log10(abs(1 - f(k, y)))
end
```

For the quadratic perturbation method we compile the models/sgm.yaml file with the main equations, steady states and parameters of the model. We solve it and retrieve policy function and Euler errors in solve-sgm/perturbation.jl:

```

function pert_eee(model; n_steps=1_000, bounds=[.01, 1.])
    res = time_iteration(model; verbose=false)

    m = model.calibration[:exogenous]
    x = model.calibration[:controls]
    p = model.calibration[:parameters]
    ss = model.calibration[:states]
```

```
9
10      f = eval(Dolang.gen_generated_gufun(model.factories[:arbitrage]))
11
12      tab = tabulate(model, res.dr, :k, bounds, ss, m; n_steps=n_steps)
13      c = tab[:c]
14      k_space = tab[:k]
15
16      eee = zeros(size(k_space))
17
18      for (i, k) in enumerate(k_space)
19
20          euler = f(m, [k], x, m, [k], x, p)
21
22          rel_error = 1 - (euler[1] + 1) / c[i]
23
24          eee[i] = log10(abs(rel_error))
25
26      end
27
28      return eee, tab
29
30  end
```

Finally we plot the policy functions and Euler errors in Figure 2 and Figure 3, respectively. We can see that the third method delivers a much better approximation of the policy function, while the second method is mainly reliable around the steady state (approx at $k = 0.18$).

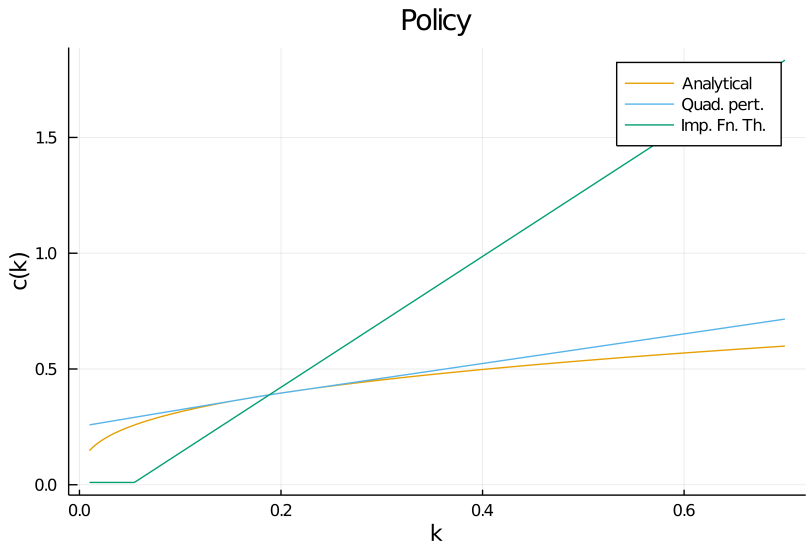

Figure 2: Policy functions for analytical solution, implicit function and quadratic approximation.

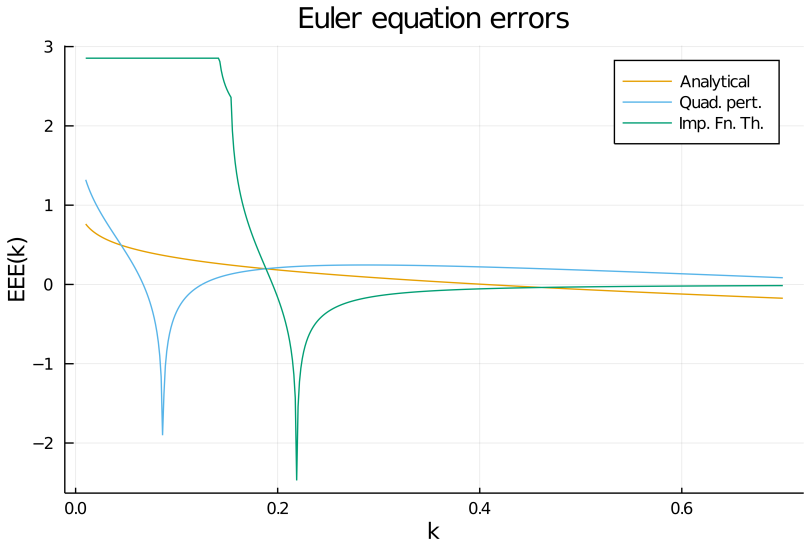

Figure 3: Euler equation errors for analytical solution, implicit function and quadratic approximation.

Finally we plot the simulation of k for a given path of z. We managed to get meaningful representations for the analytical solutions and the quadratic perturbation, and we present the results in Figure 5. The analytical solution clearly follows z much more closely, while the policy function obtained via the quadratic approximation fluctuates around the steady state. The code that implements the graph is again in stoch-growth.jl in the function perfect_foresight_simulation:

```
1
2  function perfect_foresight_simulation(model; T=200)
3      ts = collect(1:T)
4      shocks = Dict(:z => construct_shock(model; T=T))
5
6      # Dolo.jl simulations
7      sol_ref = perfect_foresight(model, shocks, T=T, complementarities
           =false, verbose=false)
8      z = sol_ref[1, 1:end - 1]
9      k = sol_ref[2, 1:end - 1]
10     c = sol_ref[3, 1:end - 1]
11
12     k1 = k[1]
13
14     # Analytical simulation
15     c_p_analytical, k_p_analytical = analytical_policy(model)
16     tab_analytical = simulate_shock(c_p_analytical, k_p_analytical,
           k1, z)
17     k_analytical = tab_analytical[:, 2]
18
```

```
    # Implicit policy simulation
    c_p_imp, k_p_imp = loglin_policy(model)
    tab_imp = simulate_shock(c_p_imp, k_p_imp, k1, z)
    k_imp = tab_imp[:, 2]

    plot(title="Simulation", xaxis="T", dpi=800)
    plot!(ts, k[2:end], yaxis="k", label="Quad. pert.", color=:blue,
        linewidth=1)
    plot!(ts, k_analytical[2:end], label="Analytical", color=:red,
        linewidth=1)


    plot!(ts, NaN .* ts, label="z", color=:black, alpha=0.5) # :(
    plt = twinx()
    plot!(plt, ts, z[2:end], label="z", color=:black, alpha=0.5,
        legend=false)

    savefig("plot_path/simulation/simulation.png")

end
```

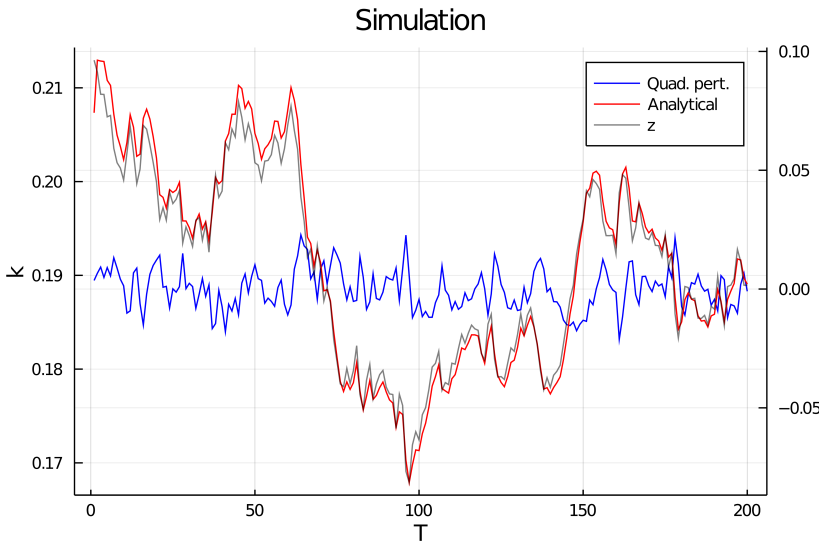

Figure 4: Simulated path for k according to method 1 and 3, given a path for z.