Essential numerical tools and perturbation analysis (1.b)

Recursive sequences

Pablo Winant

African Econometric Society Worskhop 14/12/23

Another Day in the Life of a Computational Economist

The Impossible Task"

A Recursive Sequence

Consider:

- a function

$$f : \mathbf{R}^n \rightarrow \mathbf{R}^n$$

- the recursive sequence (x_n) defined by $x_0 \in \mathbf{R^n}$ and

$$x_n = f(x_{n-1})$$

.

We want to compute a fixed point $\overline{x}$ of f

$$f(\overline{x}) = \overline{x}$$

and study its properties.

For a serious mathematical treatment, one needs assumptions on f, like continuity of differentiability, and on the metric space which contains x. It is not essential for today's discussion though.

Motivation

Some models are classically expressed in such a way.

Example: Solow Model

- capital accumulation: $k_t = (1-\delta)k_{t-1} + i_{t-1}$
- production: $y_t = k_t^\alpha$
- consumption: $c_t = (1-s)y_t$
- investment: $i_t = s y_t$ with $s \in \mathbf{R}$

⏩ after some calculations...

$$k_{t+1} = (1-\delta)k_t + (1-s)k_t^\alpha$$

$$k_t = f(k_{t-1}, s)$$

Code: Solow model

Let's code the solow model

First, we use a namedtuple to store the model parameters

```
p_array =  [0.96, 0.1, 0.3, 4.0]
```

```
# different options to code the model
p_array = [0.96, 0.1, 0.3, 4]
```

```
p_tuple =  (0.96, 0.1, 0.3, 4, "Baby 🦈")
```

```
p_tuple = (0.96, 0.1, 0.3, 4, "Baby 🦈")
```

```
p_dict =  Dict(:α ⇒ 0.3, :γ ⇒ 4, :δ ⇒ 0.1, :β ⇒ 0.96)
```

```
p_dict = Dict( :β=>0.96, :δ=>0.1, :α=>0.3, :γ=>4 )
```

```
0.3
```

```
p_dict[:α]
```

Namedtuples are great!

```
# elements from named tuples can be accessed in many different ways
# try to recover the value of $\alpha$
```

```
p0 =  (β = 0.96, δ = 0.1, α = 0.3, γ = 4)
```

```
p0 = (;  β=0.96, δ=0.1, α=0.3, γ=4 )
```

```
0.96
```

```
p0[1]
```

```
0.3
```

```
p0.α
```

```
(β = 0.96, δ = 0.1, α = 0.3, γ = 4)
```

```
# to avoid typing repetitive code, we can use keyword unpacking
# α = p0.α
# β = p0.β
 (;  β, δ, α, γ) = p0
```

```
0.3
```

```
α
```

```
# one can easily create another tuple with some changed parameters
# merge(p0, (;δ=0.2) )
```

```
(β = 0.96, δ = 0.2, α = 0.35, γ = 4)
```

```
merge(p0, (;δ=0.2, α=0.35))
```

Second, we write a function to compute the model transition

```
f (generic function with 1 method)
```

```
function f(k, p; s=0.5)

    # k: capital (float)
    # p: parameters (namedtuple)
    # s: saving rate (float)

    # unpack the tuple
    (;α, δ) = p

    # compute next state
    kn = k*(1-δ) + k^α*s

    return kn

end
```

```
0.8561261981781177
```

```
f(0.5, p0) #; s=0.2)
```

```
# to play with the parameters one can use a pluto slider
```

```
@bind s_sl Slider(0:0.01:0.5)
```

```
0.5231027156720612
```

```
f(0.5, p0; s=s_sl)
```

bonus: make a nice graph

```
md"__bonus: make a nice graph__"
```

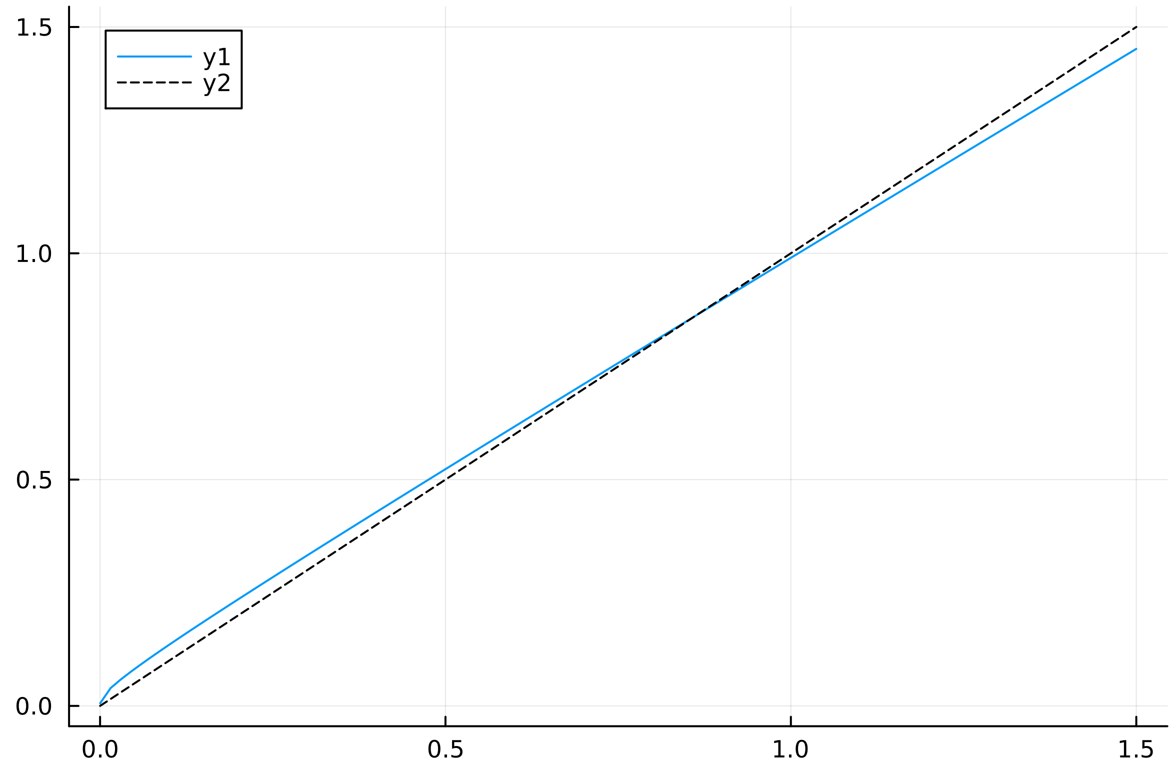
```
kvec = 0.0001:0.015150505050505:1.5
```

```
kvec = range(0.0001, 1.5; length=100)
```

```
k1vec =
 [0.00576862, 0.0393837, 0.0589189, 0.0766245, 0.093465, 0.109785, 0.125752, 0.14146, 0.156
```

```
k1vec = [f(k, p0; s=s_sl) for k in kvec]
```

```
#[1,2,3,4].^2
#(u->f(u,p0; s=s_sl)).(kvec)
```




```
begin
    pl0 = plot(kvec, k1vec)
    plot!(pl0, kvec, kvec; color=:black, linestyle=:dash)
end
```

Third we can simulate the model over T periods.

```
md"""__Third__ we can simulate the model over $T$ periods."""
```

```
simulate0 (generic function with 1 method)
```

```
function simulate0(k0, T, p; s=0.5)

    # simulation vector
    sim = [k0]

    for i ∈ 1:T     # same as for i in ... or for i=...
        # in Julia, intervals contain the lower and upper bound
        k1 = f(k0, p; s=s)

        # add new value to simulation vector
        push!(sim, k1)

        k0 = k1
    end

    return sim
end
```

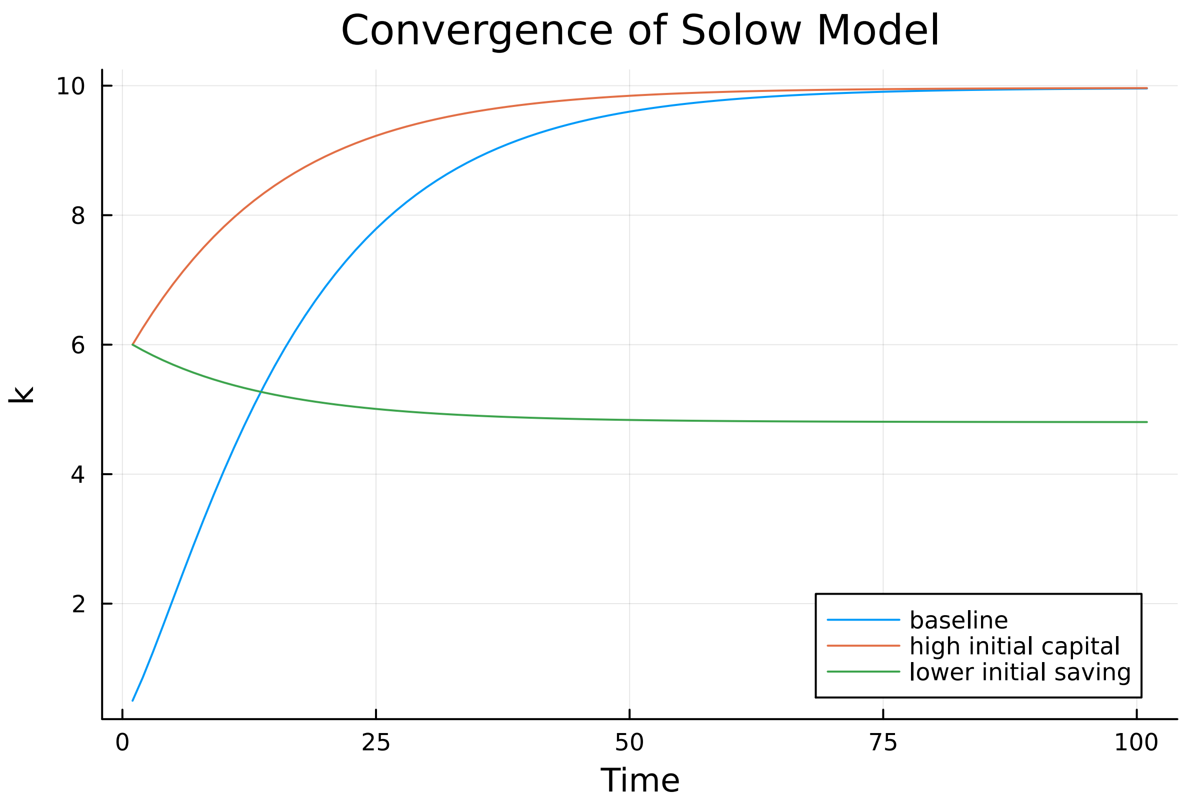
```
sim = simulate0(0.5, 100, p0;)

```

Fourth: look at what you've done!

What happens if capital is higher? If saving rate is lower?

```
using Plots
```




```
begin
    pl = plot(
        simulate0(0.5, 100, p0;);
        label="baseline", title="Convergence of Solow Model", xaxis="Time", yaxis="k"
    )
    plot!(pl, simulate0(6.0, 100, p0;); label="high initial capital");
    plot!(pl, simulate0(6.0, 100, p0;s=0.3); label="lower initial saving");

end
```

Local Analysis

Suppose there is a steady-state $\overline{x}$ such that $f(\overline{x}) = \overline{x}$.

Then stability of $\overline{x}$ is characterized by the derivative $f'(x)$.

In general $f'(\overline{x})$ is a matrix $L \in R^n \times R^n$. It is *defined* by the relation:
$f(\overline{x}+u) = f(\overline{x}) + L.u + o(u)$

The spectral radius $\rho(L)$ of matrix L is decisive:

- if strictly smaller than 1, then (x_n) is locally stable
- if strictly bigger than 1, f is locally unstable.
- if equal to 1, anything can happen (it depends on the problem)

> Tip
>
> The spectral radius of a matrix is equal to the norm of its biggest eigenvalue

▶ Informal (and incorrect) intuition.

Assessing Convergence

We are often interested in monitoring the speed of convergence for a given algorithm.

Since $\overline{x}$ is in general not known (would be too easy), the solution error $\epsilon_n = |x_n - \overline{x}|$ is not available.

We look instead at **successive approximation errors**:

$$\eta_n = |x_n - x_{n-1}|$$

When there is no more progress $\eta_n = 0$ implies $x_n = f(x_{n-1}) = x_{n-1}$ so that x_{n-1} is a fixed point.

In particular, we measure how quickly they decrease by measuring the **ratio of successive approximation errors**.

$$\lambda_n = \frac{|x_n - x_{n-1}|}{|x_{n-1} - x_{n-2}|}$$

When the ratio stays strictly below 1, that is if we know $\forall n > N,\ \lambda_n < \overline{\lambda} < 1$, the algorithm is converging properly.

A ratio oscillating around 1., or converging to 1. signals convergence problems.

▶ Some details

In practice

- Problem:
  - Suppose one is trying to find x solving the model $G(x) = 0$

  - An iterative algorithm provides a function f defining a recursive series x_{n+1}.
- The best practice consists in monitoring at the same time:
  - the success criterion: $\epsilon_n = |G(x_n)|$
    - have you found the solution?
  - the successive approximation errors $\eta_n = |x_{n+1} - x_n|$
    - are you making progress?
  - the ratio of successive approximation errors $\lambda_n = \frac{\eta_n}{\eta_{n-1}}$
    - what kind of convergence?
    - (if $|\lambda_n| < 1$: OK, otherwise: ❓)

Exercise 1

Modify the `simulate` function so that it prints convergence metrics.

```
md"""## Exercise 1

__Modify the `simulate` function so that it prints convergence metrics.__
"""
```

```
simulate1 (generic function with 1 method)
```

```
function simulate1(k0, T, p; s=0.5, verbose=true, τ_η=1e-8)

    # simulation vector
    sim = [k0]

    η0 = NaN
    for i ∈ 1:T    # same as for i in ... or for i=...
        # in Julia, intervals contain the lower and upper bound
        k1 = f(k0, p; s=s)

        # add new value to simulation vector
        push!(sim, k1)

        η = abs(k1 - k0)
        if  η < τ_η
            return (sim, i)
        end
        λ = η / η0

        if (verbose)
            println("Iteration $i: η=$η, λ=$λ")
        end

        k0 = k1
        η0 = η
    end

    error("No convergence")
    # return sim, -1
end
```

```
sim1 =
([0.5, 0.856126, 1.24775, 1.6573, 2.07339, 2.48832, 2.89675, 3.29501, 3.68054,   more ,9.9
```

```
sim1 = simulate1(0.5, 1000, p0;)
```

```
Iteration 64: η=0.009309643794537692, λ=0.9302948287002395
Iteration 65: η=0.00866051961990344, λ=0.930274005218640З
Iteration 66: η=0.008056488774672133, λ=0.9302546646458563
Iteration 67: η=0.007494441529111384, λ=0.9302366997236186
Iteration 68: η=0.006971479482977827, λ=0.9302200111773286
Iteration 69: η=0.006484901636151008, λ=0.9302045070899384
Iteration 70: η=0.006032191316540647, λ=0.9301901023314428
Iteration 71: η=0.005611003921380586, λ=0.9301767180352621
Iteration 72: η=0.005219155428868305, λ=0.9301642811156926
Iteration 73: η=0.004854611638256401, λ=0.9301527238304628
Iteration 74: η=0.004515478097721015, λ=0.9301419833745568
Iteration 75: η=0.004199990680785248, λ=0.9301320015050023
Iteration 76: η=0.003906506773642349, λ=0.9301227242037526
Iteration 77: η=0.0036334970372120523, λ=0.9301141013571704
Iteration 78: η=0.0033795377094776313, λ=0.9301060864688963
Iteration 79: η=0.0031433034152144046, λ=0.9300986363902
Iteration 80: η=0.0029235604518724756, λ=0.9300917110711248
Iteration 81: η=0.0027191605220053816, λ=0.9300852733398483
Iteration 82: η=0.0025290348841160437, λ=0.9300792886809345
Iteration 83: η=0.0023521888954558534, λ=0.9300737250518385
Iteration 84: η=0.0021876969216556574, λ=0.9300685526923561
Iteration 85: η=0.0020346975896270436, λ=0.930063743970155
Iteration 86: η=0.0018923893614264387, λ=0.9300592732177514
Iteration 87: η=0.0017600264081760741, λ=0.9300551165904926
Iteration 88: η=0.0016369147643757742, λ=0.9300512519423607
Iteration 89: η=0.0015224087440959266, λ=0.9300476586980302
Iteration 90: η=0.0014159076017250527, λ=0.9300443177406216
Iteration 91: η=0.0013168524210023236, λ=0.9300412113035861
Iteration 92: η=0.0012247232171169742, λ=0.9300383228856995
Iteration 93: η=0.001139036237560731, λ=0.9300356371475081
Iteration 94: η=0.0010593414484070252, λ=0.9300331398372594
Iteration 95: η=0.000985220193486569, λ=0.9300308177010205
```

No convergence

1. **error**(::String) @ *error.jl:35*
2. **var"#simulate1#3"**(::Float64, ::Bool, ::Float64, ::typeof(Main.var"workspace#4".simulate1), ::Float64, ::Int64, ::NamedTuple{(:β, :δ, :α, :γ), Tuple{Float64, Float64, Float64, Int64}}) @ *Other: 28*
3. **simulate1** @ *Other: 1* [inlined]
4. **top-level scope** @ *Local: 1* [inlined]

```
simulate1(0.5, 10, p0;) # no convergence in 10 iterations
```

```
Iteration 1: η=0.35612619817811775, λ=NaN
Iteration 2: η=0.391621340097257б, λ=1.0996701228405186
Iteration 3: η=0.4095533545053651, λ=1.0457891656354943
Iteration 4: η=0.41609227881014355, λ=1.0159659888823906
Iteration 5: η=0.4149244351803487, λ=0.9971933061744516
Iteration 6: η=0.4084351186841366, λ=0.9843602450325889
Iteration 7: η=0.3982534326197662, λ=0.9750714725580578
Iteration 8: η=0.3855336812688326, λ=0.9680611632967949
Iteration 9: η=0.3711145335897159, λ=0.962599512365141
Iteration 10: η=0.3556156742911982, λ=0.9582369918294485
```

Convergence Rates

Convergence of sequence x_n towards $x^\star$ can be classified as:

- linear

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^\star|}{|x_n - x^\star|} = \mu \in R^+$$

- superlinear:

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^\star|}{|x_n - x^\star|} = 0$$

- quadratic:

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^\star|}{|x_n - x^\star|^{2}} = \mu \in R^+$$

Linear convergence is also called geometric convergence. It is slooooooow.

Improve Convergence

There are tricks to improve convergence rate.

Define a new function g with the same steady-state as f

Tip

Consider the following iteration:

$$x_{n+1} = (1-\lambda) x_n + \lambda f(x_n)$$

Parameter λ is the learning rate:

- acceleration: $\lambda>1$
- dampening: $\lambda<1$

$$g(x) = (1-\lambda)x + \lambda f(x)$$

```
tip(md"""Consider the following iteration:

$$x_{n+1} = (1-\lambda) x_n + \lambda f(x_n)$$

Parameter $\lambda$ is the learning rate:
- acceleration: $\lambda>1$
- dampening: $\lambda<1$

$g(x) = (1-\lambda)x + \lambda f(x)$
""")

```

Tip

Suppose f is differentiable $\mathbf{R} \rightarrow \mathbf{R}$.

Define: $g(x) = x - \frac{f(x)-x}{f'(x)-1}$

Function g corresponds to the Newton iterations. It If $f'(\overline{x}) \neq 0$, it converges *quadratically*.

Exercise 2 (Bonus)

Suppose the goal is to compute the steady-state.

Propose a way to accelerate convergence of the simulate1 function.

```
md"""## Exercise 2 (Bonus)

Suppose the goal is to compute the steady-state.

Propose a way to accelerate convergence of the simulate1 function.


"""
```

```
simulate2 (generic function with 1 method)
```

```
# Your code
function simulate2(k0, T, p; s=0.5, verbose=true)
end

```

Table of Contents