Basic Numerical Concepts

Felix Wellschmied

UC3M

Macroeconomics III

Motivation

Who Needs Numerical Methods

- Macro economists.
- Micro economists: Dynamic games and dynamic contracts.
- Applied economists: Estimate (non-)parametric models.
- Econometricians: Bootstrapping and simulations.

A Simple Growth Model

$$
\begin{aligned}
V(k,z) &= \max_{c,k'} \Big\{ ln(c) + \beta \mathbb{E} V(k',z') \Big\} \\
c &= y - i \\
k' &= (1-\delta)k + i, \quad 0 \leq \delta \leq 1 \\
y &= zk^{\alpha} \\
z' &= P(z).
\end{aligned}
$$

Goal: Find the policy function (and value function) $k_{t+1} = \phi(k_t, z_t)$.

The "most basic" macro model, yet analytical solution only with $\delta = 1$.

Algorithm Solving the Model

1. Discretize a grid for the state k and z.
2. Guess the (continuous and concave) value function $V^0(k, z)$.
3. Solve $V^n(k, z) = \max_{c,k'} \left\{ ln(c) + \beta \mathbb{E} V^{n-1}(k', z') \right\}$.
4. Replace last iteration guess by new solution $V^{n-1} = V^n$.
5. Iterate until $|V^n - V^{n-1}| < crit$.

Limits to VFI

This is great, but many problems are more complex.

- Household has assets, a_t, and housing, h_t, and decides a_{t+1}, h_{t+1}.
- It earns its productivity $exp(z_t)$.
- Log productivity follows a Markov chain: $P_{jk}(z_{t+1} = z^j | z_t = z^k)$.
- $c_t + a_{t+1} + h_{t+1} = a_t + h_t + exp(z_t)$.

Limits to VFI II

$$V(a, h, z) = \max_{c,a',h'} \left\{ U(c, h) + \beta \mathbb{E} V(a', h', z') \right\}$$

- Two endogenous dynamic state variables a_t and h_t.
- One exogenous state variable z_t.
- Assume I discretize $N_a = 1000$, $N_h = 1000$, $N_z = 5$, these are 5,000,000 state combinations with 1,000,000 choices.
- 5,000,000,000,000 computations of $U(c, h) + \beta V(a', h', z')$ and finding 5,000,000 times the maximum for one update of V!

Two Controls

Concepts

- Consider a problem with one state variable (size $N1$) and two controls (sizes $N1$ and $N2$).
- We could construct two grids, one for each control.
- For each iteration of the value function we need to solve $\forall N1$, $N1\ X\ N2$ possible choices.
- Sometimes, first-order conditions suggest something simpler.

Two Controls

Neo-classical growth model with labor l:

$$V(k,z) = \max_{c,k',l}\left\{\frac{\left(c^{\theta}(1-l)^{1-\theta}\right)^{1-\tau}}{1-\tau} + \beta \mathbb{E} V(k',z')\right\}$$

$$c + k' = zk^{\alpha} l^{1-\alpha} + (1-\delta)k$$

$$ln(z') = \rho ln(z) + \epsilon'$$

Find $\phi_c(k,z), \phi_l(k,z)$. The first order conditions imply:

$$\frac{c}{1-l} = \frac{\theta}{1-\theta}(1-\alpha)zk^{\alpha} l^{-\alpha}$$

Two Controls

Neo-classical growth model with labor l:

$$V(k,z) = \max_{c,k',l}\left\{\frac{\left(c^{\theta}(1-l)^{1-\theta}\right)^{1-\tau}}{1-\tau} + \beta\mathbb{E}V(k',z')\right\}$$
$$c + k' = zk^{\alpha}l^{1-\alpha} + (1-\delta)k$$
$$ln(z') = \rho ln(z) + \epsilon'$$

Find $\phi_c(k,z), \phi_l(k,z)$. The first order conditions imply:

$$\frac{c}{1-l} = \frac{\theta}{1-\theta}(1-\alpha)zk^{\alpha}l^{-\alpha}$$

Knowing optimal policy $\phi_c(k,z)$, this is a non-linear root finding problem in l.

Two Controls II

- One way to solve the problem is:

    1. Guess optimal policy for labor, $\phi_l(k, z)$.
    2. Solve for optimal policy for consumption $c = \phi_c(k, z)$.
    3. Solve FOC for optimal $\phi_l(k, z)$.
    4. Iterate until convergence.

- For step (3) we need a **root-finding** algorithm.

Newton-Raphson Method for Root Finding

- Newton method uses first order approximation to function.
- First order approximation around guess x_0:

  $$f(x) \approx f(x_0) + f'(x_0)(x - x_0).$$

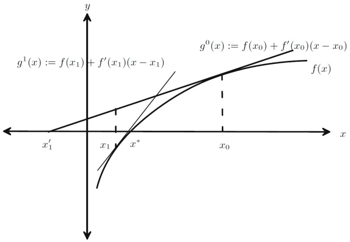
- Setting $f(x) = 0$ and solving for x gives new guess:

  $$x' = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

  The tangent intersects the x-axis.

- This requires **numerical differentiation** (in one second)!

Modified Newton-Raphson Method



- When the objective function is close to flat around x^0, the linear approximation may lead to a poor prediction.
- Function may not be defined at x'.

  Reformulating the problem is often possible.

- The Modified Newton-Raphson Method updates slowly $\lambda \in [0, 1]$:
  $x' = x_0 - \lambda \frac{f(x_0)}{f'(x_0)}$.

Multivariate case

The method can be extended straightforward to the multivariate case:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \Leftrightarrow \begin{cases} 0 = f^1(x_1, ..., x_n) \\ ... \\ 0 = f^n(x_1, ..., x_n) \end{cases}$$

Define the Jacobian:

$$\mathbf{J}(\mathbf{a}) = \begin{bmatrix} f_1^1 & f_2^1 & f_3^1 & \dots & f_n^1 \\ f_1^2 & f_2^2 & f_3^2 & \dots & f_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_1^n & f_2^n & f_3^n & \dots & f_n^n \end{bmatrix}, \quad f_j^i = \frac{\partial f^i(\mathbf{x})}{\partial x_j}$$

Multivariate case II

If $\mathbf{J}(\mathbf{x})$ is Libschitz (sufficient: continuous differentiable), then approximate

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x_0})(\mathbf{x} - \mathbf{x}_0),$$

with solution

$$\mathbf{x}' = \mathbf{x}_0 - \lambda \mathbf{J}(\mathbf{x}_0)^{-1}\mathbf{f}(\mathbf{x}_0).$$

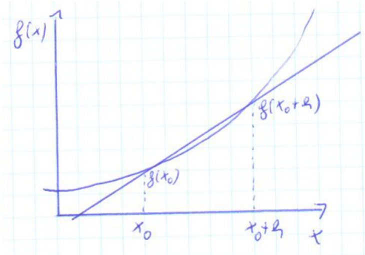
Numerical Differentiation

For this algorithm, we need to compute

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

- Simplest method called one sided approximation:

  $f'(x) \approx \frac{f(x+h)-f(x)}{h}$. Slope error proportional to h

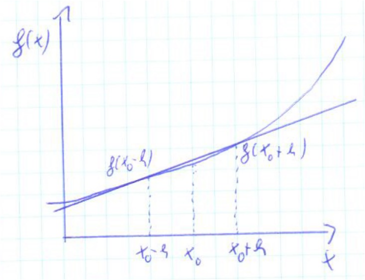


Numerical Differentiation II

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Two sided approximation:

  $f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$. Slope error proportional to h^2.



Numerical Differentiation III

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Five point method:

  $$f'(x) \approx \frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}.$$

  Slope error proportional to h^4.

Alternatives to "Standard" VFI

Methods Relying on FOCs

Projection Method

Consider the Neo-classical growth model without labor:

$$
\begin{aligned}
c_t^{-\gamma} &= \mathbb{E}\Big\{\beta c_{t+1}^{-\gamma}(\alpha z_{t+1} k_{t+1}^{\alpha-1} + (1-\delta))\Big\} \\
c_t + k_{t+1} &= z_t k_t^{\alpha} + (1-\delta)k_t \\
ln(z_{t+1}) &= \rho ln(z_t) + \epsilon_{t+1} \\
\epsilon_{t+1} &\sim N(0, \sigma^2)
\end{aligned}
$$

Rational expectation solution:

$$
\begin{aligned}
c_t &= \mathbf{c}(k_t, z_t) \\
k_{t+1} &= \mathbf{k}(k_t, z_t)
\end{aligned}
$$

Reformulating the Problem

$$\mathbf{c}(k_t, z_t)^{-\gamma} = \mathbb{E}\Big\{\beta \mathbf{c}\left(k_{t+1}, z_{t+1}\right)^{-\gamma}\left(\alpha z_{t+1} k_{t+1}^{\alpha-1} + (1-\delta)\right)\Big\}$$

Substitute in the budget constraint:

$$\mathbf{c}(k_t, z_t)^{-\gamma} - \mathbb{E}\Big\{\beta \mathbf{c}\left(z_t k_t^{\alpha} + (1-\delta)k_t - \mathbf{c}(k_t, z_t), z_{t+1}\right)^{-\gamma}$$
$$\left(\alpha z_{t+1}\left(z_t k_t^{\alpha} + (1-\delta)k_t - \mathbf{c}(k_t, z_t)\right)^{\alpha-1} + (1-\delta)\right)\Big\} = 0$$

Which is at each grid point k_i, z_i a root-finding problem in optimal consumption.

Idea of Projection Methods

Idea, **approximate** policy **function** by a known function:

$$\mathbf{c}(k_t, z_t) \approx P_n(k_t, z_t; \nu_n).$$

Idea of Projection Methods

Idea, **approximate** policy **function** by a known function:

$$\mathbf{c}(k_t, z_t) \approx P_n(k_t, z_t; \nu_n).$$

- Usually, P_n of polynomial class.
- Euler equation needs to hold at each grid point i.

Projection Method II

Substituting $\mathbf{c}(k_i, z_i) \approx P_n(k_i, z_i; \nu_n)$:

$$e(k_i, z_i; \nu_n) = P_n(k_i, z_i; \nu_n)^{-\gamma} - \mathbb{E}\Big\{\beta P_n(k', z'; \nu_n)^{-\gamma}(\alpha z' k'^{\alpha-1} + (1-\delta))\Big\}$$

Inserting budget constraint and law of motion:

$$\begin{aligned} e(k_i, z_i; \nu_n) = {} & P_n(k_i, z_i; \nu_n)^{-\gamma} - \\ & \mathbb{E}\Big\{\beta P_n(z_i k_i^{\alpha} + (1-\delta)k_i - P_n(k_i, z_i; \nu_n), exp(\rho ln(z_i) + \epsilon'); \nu_n)^{-\gamma} \\ & [\alpha exp(\rho ln(z_i) + \epsilon')(z_i k_i^{\alpha} + (1-\delta)k_i - P_n(k_i, z_i; \nu_n))^{\alpha-1} + (1-\delta)]\Big\} \end{aligned}$$

Projection Method III

Approximating integral by J Hermite Gaussian quadrature nodes:

$$e(k_i, z_i; \nu_n) = P_n(k_i, z_i; \nu_n)^{-\gamma} -$$

$$\sum_{j=1}^{J} \left[\beta \frac{\omega_j}{\sqrt{\pi}} P_n(z_i k_i^{\alpha} + (1-\delta)k_i - P_n(k_i, z_i; \nu_n), exp(\rho ln(z_i) + \sqrt{2}\sigma\xi_j); \nu_n)^{-\gamma} \right.$$

$$\left. [\alpha exp(\rho ln(z_i) + \sqrt{2}\sigma\xi_j)(z_i k_i^{\alpha} + (1-\delta)k_i - P_n(k_i, z_i; \nu_n))^{\alpha-1} + (1-\delta)] \right]$$

This can be solved for ν_n at each grid point to minimize $e(k_i, z_i; \nu_n)$.

Projection Method IV

- We have to fix k_i, z_i.

  Chebyshev nodes have good convergence properties.

- We have to find the parameters ν_n.

  Collacation ($M = N$): Use a function solver to solve for $e(k_i, z_i; \nu_n) = 0$ at all grid points.

  Galerkin ($M > N$), **minimize** $e(k_i, z_i; \nu_n)$.
  For example, Gauss-Newton algorithm.

- The latter requires to evaluate $(\overline{X}'\overline{X})^{-1}$.

  Chebyshev polynomial avoids multicollinearity.

Function approximation

Assume you want to approximate $g(x)$ by a known function $f(x)$:

$$g(x) \approx f(x).$$

In our case: $\mathbf{c}(k, z) \approx P_n(k, z; \nu_n)$.

Function approximation

Assume you want to approximate $g(x)$ by a known function $f(x)$:

$$g(x) \approx f(x).$$

In our case: $\mathbf{c}(k,z) \approx P_n(k,z;\nu_n)$.

- (One-dimensional) Polynomials:

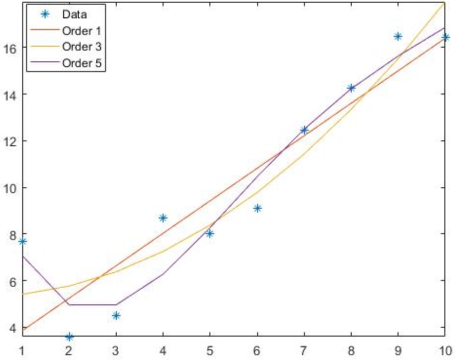
  $$f(x) = \nu_0 T_0(x) + \nu_1 T_1(x) + \nu_2 T_2(x) ... + \nu_n T_n(x)$$

  Weierstrass Theorem: A continuous, real valued function on a bounded interval can be approximated arbitrary well by a polynomial.

- **Splines** are an alternative:

  Piecewise polynomial functions.

Increasing Polynomial Order



$$f(x) = \nu_0 + \nu_1 x + \nu_2 x^2 + ... + \nu_n x^n.$$

Chebychev Polynomial

One important type has the basis function:

$$
\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \\
g(x) &\approx \sum_{j=1}^{n} \nu_j T_j(x)
\end{aligned}
$$

- Defined on the interval $[-1, 1]$, but we can always transform a continuous function.

  If space $S = [a, b]$ map into $[-1, 1]$ by $2\frac{s-a}{b-a} - 1$.

Why Use Chebychev Polynomial?

- Chebychev polynomials help avoid multicollinearity

  $\int_a^b T_i(x) T_j(x) w(x) dx = 0.$

- This is helpful when evaluating $(\overline{X}'\overline{X})^{-1}$.

Chebyshev Nodes

In projection methods, we usually create the grid using Chebychev nodes. n Chebychev nodes are the roots to the n^{th} Chebyshev basis function:

$$T_n(x) = 0$$

For example, to create $n = 3$ Chebychev nodes:

$$T_3(x) = 4x^3 - 3x = 0$$
$$x = [-\sqrt{3/4} \quad 0 \quad \sqrt{3/4}].$$

Why Use Chebyshev Nodes?

Chebyshev nodes can also be useful outside projection methods. In structural modeling, we are often free to choose nodes at which to approximate:

$$V(a) \approx f(a) \ \forall a \in \mathcal{A}$$

$\mathcal{A}$ could be a linear grid of length n in $[\underline{a}, \overline{a}]$. It can also be the n^{th} Chebyshev nodes in $[\underline{a}, \overline{a}]$.

Chebychev nodes have desirable convergence properties given an initial coefficient guess ν_n^0!

Numerical Integration

We need to know $\mathbb{E}\mathbf{c}(k_{t+1}, \rho ln(z_i) + \epsilon')^{-\gamma}$, where $\epsilon' \sim N(\mu, \sigma^2)$. Generally, in economics, we often need to calculate:

$$\int_a^b f(x)dx$$

- An integral is an infinite object.
- We need to calculate a finite approximation.

Numerical Integration II

Numerical integration replaces the integral by a finite sum:

$$\int_a^b f(x)dx \approx \sum_{j=1}^{J} \omega_j f(\xi_j)$$

- ξ_j is the node j at which we evaluate the function.
- ω_j is the weight for node j.
- This gives $2J$ free parameters.

Gauss-Legendre

Let us start with the following problem:

$$\int_{-1}^{1} f(x) \approx \sum_{j=1}^{J} \omega_j f(\xi_j)$$

Idea: Choose ξ_j, ω_j such that approximation is accurate for functions

that can be approximate by polynomials of degree $2J-1$.

$$\int_{-1}^{1} x^i dx = \sum_{j=1}^{J} \omega_j \xi_j^i, \quad i = 0, 1, ...2J-1.$$

- Yields $2J$ equations in $2J$ unknowns.
- Note, the choices of ξ and ω do not depend on f! Only the evaluations $f(\xi_j)$ do.

Gauss-Hermite

Now assume a function $g(x)$ can be approximated by polynomial, and we can write

$$f(x) = g(x)W(x)$$

Gauss-Hermite uses $W(x) = e^{-x^2}$ and domain is the real line:

$$\int_{-\infty}^{\infty} x^i e^{-x^2} dx = \sum_{j=1}^{J} \omega_j \xi_j^i, \quad i = 0, 1, ...2J - 1.$$

So we approximate:

$$\int_{-\infty}^{\infty} g(x) e^{-x^2} dx \approx \sum_{j=1}^{J} \omega_j g(\xi_j).$$

Expectations of a Normally Distributed Variable

We want to compute $\mathbb{E}(g(x))$, where $x \sim N(\mu, \sigma^2)$:

$$\mathbb{E}(g(x)) = \int_{-\infty}^{\infty} \frac{g(x)}{\sigma\sqrt{2\pi}} exp\Big(- \frac{(x-\mu)^2}{2\sigma^2}\Big) dx$$

Define auxiliary variable $y = \frac{(x-\mu)}{\sqrt{2}\sigma}$, with $x = h(y) = \sqrt{2}\sigma y + \mu$. Now use

integration by substitution:

$$\int_a^b g(x)dx = \int_{h^{-1}(a)}^{h^{-1}(b)} g(h(y))h'(y)dy \quad \text{with } x = h(y).$$

Expectations of a Normally Distributed Variable II

$$
\begin{aligned}
\mathbb{E}(g(x)) &= \int_{-\infty}^{\infty} \frac{g(x)}{\sigma\sqrt{2\pi}} exp\Big(- \frac{(x-\mu)^2}{2\sigma^2}\Big) dx \\
&= \int_{-\infty}^{\infty} \frac{g(\sqrt{2}\sigma y + \mu)}{\sigma\sqrt{2\pi}} exp\Big(- y^2 \Big) \sigma\sqrt{2} dy \\
&= \int_{-\infty}^{\infty} \frac{g(\sqrt{2}\sigma y + \mu)}{\sqrt{\pi}} exp\Big(- y^2 \Big) dy
\end{aligned}
$$

So, we have:

$$
\mathbb{E}(g(x)) \approx \sum_{j=1}^{J} \frac{\omega_j}{\sqrt{\pi}} g(\sqrt{2}\sigma\xi_j + \mu)
$$

Gauss-Newton Method

We need to find coefficients ν_n to minimize $e(k_i, z_i; \nu_n)$. One possible algorithm is the Gauss-Newton method which uses an approximation to the SSR norm. Consider the general formulation where we have outcomes, y_i, (LHS of Euler equation) and a function mapping points, x_i, (our grid) into outcomes (RHS of Euler equation). Thus,

$$\min_{\gamma}\{\sum_{i=1}^{N}(y_i - f(x_i, \gamma))^2\}.$$

$$\gamma = \begin{bmatrix} \gamma_1 \\ ... \\ \gamma_p \end{bmatrix}$$

We want to minimize the sum of squared residual $r_i = y_i - f(x_i, \gamma)$.

Gauss-Newton Method II

Consider the simpler first order approximation around γ_s:

$$r(x_i, \gamma) \approx r(x_i, \gamma_s) + [\nabla r(x_i, \gamma_s)]'(\gamma - \gamma_s)$$

$$\underset{\gamma}{min}\{\sum_{i=1}^{N}(r_i - [\nabla r(x_i, \gamma_s)]'(\gamma_s - \gamma))^2\}.$$

Where $\nabla r(x_i, \gamma_s)$ is the derivative of the residual with respect to γ_j, a $N \ X \ p$ matrix.

Gauss-Newton Method III

- Let $\overline{\gamma} = (\gamma_s - \gamma)$
- The problem has the solution: $\overline{\gamma} = (\nabla r(x_i, \gamma_s)' \nabla r(x_i, \gamma_s))^{-1} \nabla r(x_i, \gamma_s)' r(x_i, \gamma_s)$.
- It follows that the next guess is $\gamma_{s+1} = \gamma_s - \overline{\gamma}$.
- The algorithm requires $\nabla r(x_i, \gamma_s)$

  Sometimes (polynomials) known analytically, use it!

  Otherwise, use numerical differentiation.

Extensions

- Several extensions exist which deal with:

  Exploiting second derivatives (Hessian).

  Non-smooth functions (simplex methods).

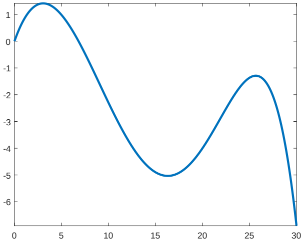
  Constrained non-linear programming.

A Simpler Approach

Alternatively, we can also iterate on γ until convergence:

1. Construct a grid X.
2. We will approximate $u'(c(X))$ but we could just as well approximate $c(X)$.
3. Guess an initial γ_0.
4. Compute the right-hand side, RHS, of the Euler equation given γ_0.
5. The FOC requires that $u'(c_t) = RHS$.
6. Given as norm SSR, the optimal γ satisfies $(X'X)^{-1}X'RHS$.
7. Check for convergence and update $\gamma_0 = \lambda\gamma_0 + (1-\lambda)\gamma$.

Global vs. Local Solutions



- Minimizers are usually designed to find a local minimum.
- So called genetic algorithms aim at finding the global minimum:

  Find a local minimum, try other starting values and recompute local minimum.

  Pattern search, *simulated annealing*.

PM with Multiple States

- We need to approximate $F(X) : [-1, 1]^L \rightarrow \mathbb{R}$.
- Polynomial function for L state variables (z, k in our case).
- We can use the Tensor product of Chebyshev polynomials:

  $P_n(X; \nu_n) = \sum_{l1=0}^{n} \cdots \sum_{L=0}^{n} \nu_{l1,\ldots,L} T_{l1}(x_1) *** T_L(x_L)$

  If basis is orthogonal in a norm, tensor product is orthogonal in the product norm.

- Number of grid points growth exponentially in number of dimensions.

  Smolyak's algorithm: Number of grid points growth polynomially in number of dimensions.

Smolyak's Algorithm

Sparse grid methods reduce computational burden.

- Idea is to choose those grid points from the Tensor grid that are important.
- In practice, Smolyak's algorithm has been found useful.
- Judd et al. (2014) provide a comprehensive discussion.

The Idea in two Dimensions

- The algorithm relies on nested sets of points: $S_i \subset S_{i+1}\ \forall i$.
- The extrema of the Chebychev-polynomialal is one class of these sets.
- Suppose we use $i1 = i2 = 3$ for our $d = 2$ dimensions. This yields a $5\ X\ 5$ tensor grid.
- Smolyak's rule is to select only those points from the sets for which $d \leq i1 + i2 \leq d + \mu$.
- μ is an accuracy parameter.

The Idea in two Dimensions

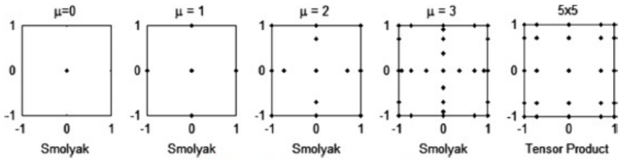


Fig. 1. Smolyak grids versus a tensor-product grid.

We need to interpolate our multidimensional function on this sparse grid. See Judd et al. (2014) for a discussion.

Algorithm for PM

1. Guess coefficients of $P_n(X; \nu_n)$.
2. For each state, compute today's decisions.
3. Using the budget constrained, compute the implied states tomorrow.
4. Use $P_n(X; \nu_n)$ to compute tomorrow's decisions (RHS of Euler eq.).
5. Compute implied today's consumption decisions, $\bar{y} = RHS^{-1/\gamma}$.
6. Compute implied coefficients by $(\overline{X}'\overline{X})^{-1}\overline{X}'\bar{y}$.
7. Check convergence of coefficients and update.

Algorithm for PM II

1. The previous algorithm is called fixed-point algorithm.
2. Uses current guess of $P_n(X;\nu_n)$ to compute LHS and RHS of FOC.
3. Convenient because no solver needed. But convergence is tricky.
4. Alternatively, use time-iteration algorithm.
5. Use $P_n(X;\nu_n)$ to compute tomorrow's policies.
6. Solve for optimal policy today to solve FOCs (a non-linear problem), $\bar{y} = RHS^{-1/\gamma}$.
7. Compute implied coefficients by $(\overline{X}'\overline{X})^{-1}\bar{y}$.

Methods not Relying on FOCs

Projection Methods

Projection methods can also deal with borrowing constraints. Consider the Aiyagary economy:

$$
\begin{aligned}
V(a,\epsilon) = \max_{c,a'} & \Big\{ U(c) + \beta \mathbb{E} V(a',\epsilon') \Big\} \\
c + a' &= \epsilon + a(1+r) \\
a' &\geq \underline{a} \\
\pi_{jk}&(\epsilon' = \epsilon^j | \epsilon = \epsilon^k)
\end{aligned}
$$

With solution

$$
c_{it} = \begin{cases} \beta(1+r)\mathbb{E}_t c_{it+1} & \text{if } a_{t+1} \geq \underline{a} \\ \epsilon + a(1+r) - \underline{a} & \text{otherwise.} \end{cases}
$$

Algorithm

1. Guess coefficients of $C(X) = P_n(X; \nu_n)$.
2. For each state, compute today's decisions.
3. If $a_{t+1} < \underline{a}$ replace $c_{it} = \epsilon + a(1+r) - \underline{a}$.
4. Use $P_n(X; \nu_n)$ to compute tomorrow's decisions (RHS of Euler eq.).
5. Compute implied today's consumption decisions, $\bar{y} = RHS^{-1/\gamma}$.
6. If $a_{t+1} < \underline{a}$ replace $c_{it} = \epsilon + a(1+r) - \underline{a}$.
7. Compute implied coefficients by $(\overline{X}'\overline{X})^{-1}\bar{y}$.
8. Check convergence of coefficients and update.

Off-Grid Choices

1. Define a grid, g_n, for your dynamic state with N points.
2. Define a second grid, g_m, for possible choices with $M > N$ points.
3. Some points of g_m are not part of g_n. **Interpolation** needed:

   If we know $V(x_1)$ and $V(x_2)$, what is $V(x_0)$ with $x_1 < x_0 < x_2$.

   Usually we use **splines** for this.
4. Super quick: interpolation base points and interpolation weights stay constant.

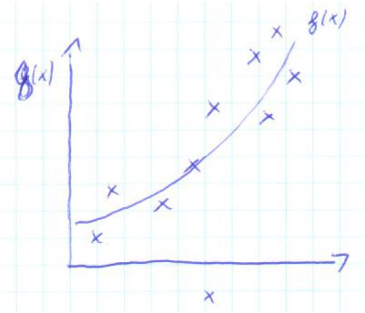
Spline Approximation I

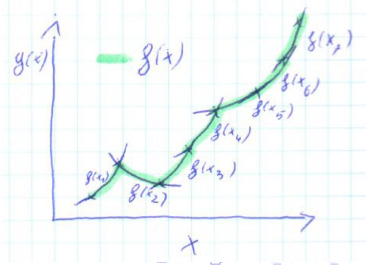
Before considering the specific issue of interpolation, consider general idea of splines. Think of spline approximation as again replacing an unknown function $f(x)$ by a know function $g(x)$.

- Polynomials assume $g(x) \approx f(x)\ \forall x \in [\underline{x}, \overline{x}]$.

- Splines fit polynomials for different regions of $[\underline{x}, \overline{x}]$: $[x_1, x_2]$,$[x_2, x_3]$,...

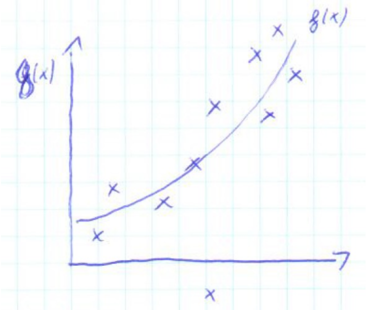
  By using $N-1$ splines, we assure $f(x_i) = g(x_i)$.

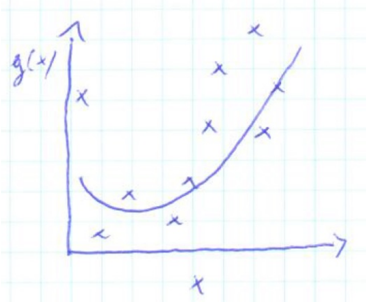




Spline Approximation II

- This *local* approach assures that a change in $x >> x_i$ does little to $f(x_i)$.

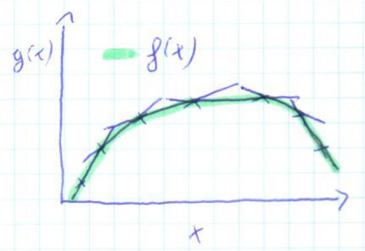




Different Splines

Simplest is a polynomial of order one which is called piecewise linear spline. For $x \in [x_i, x_{i+1}]$:

$$f(x) = f(x_i) + (x - x_i)\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}.$$



Different Splines II

The function is non-differentiable at the nodes. To avoid this, use cubic splines:

$$f(x) = a_i + b_i x + c_i x^2 + d_i x^3.$$

- With n-segments, $4n$ unknowns.

Different Splines II

The function is non-differentiable at the nodes. To avoid this, use cubic splines:

$$f(x) = a_i + b_i x + c_i x^2 + d_i x^3.$$

- With n-segments, $4n$ unknowns.
- $f(x_i) = g(x)\ \forall x_i$.
- assure differentiability.
- assure 2nd derivative.
- 2 free parameters left.

Interpolation

- Spline approximation gives us function defined on $\mathbb{R}$. Interpolation requires only specific points.

- One dimension:

  I know $V(x_1)$ and $V(x_2)$.

  I want to know $V(x_0)$ where $x_1 < x_0 < x_2$.

  Use a function $V(x_0) \approx f(x_1, x_2, x_0, V(x_1), V(x_2))$

- $V(x)$ needs to be continuous and monotone between grid points.

- Idea easily extended to n-dimensions:

  Denote by $X_0^n = [x_0^1, ..., x_0^n]$.

  $V(X_0^n) \approx f(X_1^n, X_2^n, X_0^n, V(X_1^n), V(X_2^n))$

Linear Interpolation

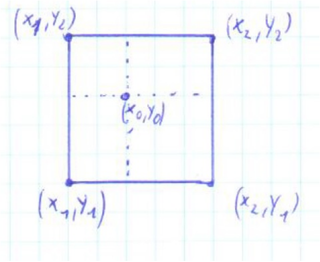
- Simplest function is linear interpolation:

  One dimension: $V(x_0) = V(x_1) + \frac{V(x_2)-V(x_1)}{x_2-x_1}(x_0 - x_1)$

- The resulting linear spline approximation is not differentiable.

- Linear interpolation, by far the fastest!

Bilinear Interpolation



Define $d = \frac{1}{(x_2 - x_1)(y_2 - y_1)}$

$$\begin{aligned} V(x_0, y_0) = d[&V(x_1, y_1)(x_2 - x_0)(y_2 - y_0) + V(x_2, y_1)(x_0 - x_1)(y_2 - y_0) \\ &+ V(x_1, y_2)(x_2 - x_0)(y_0 - y_1) + V(x_2, y_2)(x_0 - x_1)(y_0 - y_1)] \end{aligned}$$

Spline Interpolation

- When function is non-linear, more accurate functions available.
- As seen, cubic splines (Cubic Hermite Splines) assure first two derivatives at $V(x_1)$ and $V(x_2)$.
- In theory, can be extended to higher order derivatives.

Tsao and Tsitsiklis (1991) Multigrid

1. Solve the model on a curse grid, yielding V^0.
2. Increase number of grid points in each dimension by factor 2.
3. Obtain initial guess of value function by interpolating using V^0.
4. Decrease critical value by factor of 2.
5. Perform value function iteration to obtain V^1.
6. Repeat until desired grid size.

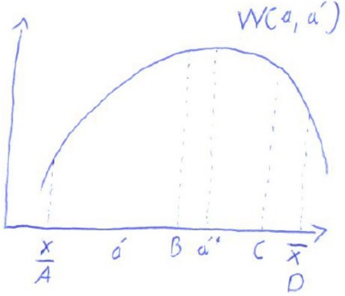
Golden Section Search

Consider again a simple household problem:

$$V(a,z) = \max_{c,a'} \Big\{ U(c) + \beta \mathbb{E} V(a',z') \Big\}$$
$$c + a' = z + a(1+r)$$
$$\underline{x} \leq a' \leq \overline{x}.$$

- We know $W(a, a', z) = U(a') + \beta \mathbb{E}_z V(a', z')$ is concave.
- Find the maximum over a concave function in interval $[\underline{x'}, \overline{x'}]$.

Golden Section Search



- We know a'^{*} is between $[A, D]$.
- Assume we evaluate $W(a, B)$ and $W(a, C)$

$W(a, B) > W(a, C)$ so $a'^{*} \in [A, C]$.

Otherwise, $a'^{*} \in [B, D]$.

Only one new function evaluation.

Golden Section Search II

- How to choose B, C?
- Find the maximum with minimum function evaluations.
- Choose intervals to have same length: $\overline{AC} = \overline{BD}$.
- Assure that: $p := \frac{\overline{AC}}{\overline{AD}} = \frac{\overline{A_1 C_1}}{\overline{A_1 D_1}}$.

  $p = \frac{\sqrt{5}-1}{2} \approx 0.618$.

Golden Section Search Algorithm

1. Set $A = \underline{x}$, $D = \overline{x}$. Compute:

   $$B = pA + (1-p)D, \quad C = (1-p)A + pD.$$

2. If $W(a, B) > W(a, C)$, replace D by C and C by B. Compute:

   $$B = pA + (1-p)D.$$

3. Iterate until $|A - D| < crit$.

B, C may be off grid points. Interpolation needed!

Endogenous Grid Points (EGM)

Consider the Aiyagari economy, where households face an exogenous borrowing constraint

$$V(a,\epsilon) = \max_{c,a'}\Big\{U(c) + \beta\mathbb{E}V(a',\epsilon')\Big\}$$

$$c + a' = \epsilon + a(1+r)$$

$$a' \geq \underline{a}$$

$$\pi_{jk}(\epsilon' = \epsilon^j | \epsilon = \epsilon^k)$$

Endogenous Grid Points II

The first order condition implies

$$U'(\mathbf{c}(a_t,\epsilon_t)) = (1+r)\beta\sum_{j=1}^{N}\pi(\epsilon_{t+1}|\epsilon_t)U'(\mathbf{c}(a_{t+1},\epsilon_{t+1}))$$

$$U'(\mathbf{c}(a_t,\epsilon_t)) - (1+r)\beta\sum_{j=1}^{N}\pi(\epsilon_{t+1}|\epsilon_t)U'(\mathbf{c}(a_t+\epsilon_t-\mathbf{c}(a_t,\epsilon_t),\epsilon_{t+1})) = 0$$

- This is (again) a root finding problem in optimal policy $c(a,\epsilon)$.
- Carroll (2006) insight: If we knew $c(a_{t+1},\epsilon_{t+1})$, simply a linear equation.

E.g., $c = \left((1+r)\beta\sum_{j=1}^{N}\pi(\epsilon_{t+1}|\epsilon_t)c(a_{t+1},\epsilon_{t+1})^{-\gamma}\right)^{-1/\gamma}$.

Endogenous Grid Points Algorithm

1. Construct a grid of assets today, $a \in A$, and tomorrow $\mathbf{a} \in \mathcal{A}$ with $\mathbf{a}_1 = \underline{a}$.
2. Guess the policy function $c(\mathbf{a}, \epsilon)$.
3. Solve $B(\mathbf{a}, \epsilon) = (1 + r)\beta \sum_{j=1}^{N} \pi(\epsilon'|\epsilon)U'(c(\mathbf{a}, \epsilon'))$.
4. Solve for implied consumption today $c(\tilde{a}, \epsilon) = B(\mathbf{a}, \epsilon)^{-1/\gamma}$.

Endogenous Grid Points Algorithm

1. Construct a grid of assets today, $a \in A$, and tomorrow $\mathbf{a} \in \mathcal{A}$ with $\mathbf{a}_1 = \underline{a}$.
2. Guess the policy function $c(\mathbf{a}, \epsilon)$.
3. Solve $B(\mathbf{a}, \epsilon) = (1 + r)\beta \sum_{j=1}^{N} \pi(\epsilon'|\epsilon)U'(c(\mathbf{a}, \epsilon'))$.
4. Solve for implied consumption today $c(\tilde{a}, \epsilon) = B(\mathbf{a}, \epsilon)^{-1/\gamma}$.
5. From budget constraint: $\tilde{a} = \frac{c+\mathbf{a}-\epsilon}{1+r}$.

Endogenous Grid Points Algorithm

1. Construct a grid of assets today, $a \in A$, and tomorrow $\mathbf{a} \in \mathcal{A}$ with $\mathbf{a}_1 = \underline{a}$.
2. Guess the policy function $c(\mathbf{a}, \epsilon)$.
3. Solve $B(\mathbf{a}, \epsilon) = (1 + r)\beta \sum_{j=1}^{N} \pi(\epsilon'|\epsilon)U'(c(\mathbf{a}, \epsilon'))$.
4. Solve for implied consumption today $c(\tilde{a}, \epsilon) = B(\mathbf{a}, \epsilon)^{-1/\gamma}$.
5. From budget constraint: $\tilde{a} = \frac{c+\mathbf{a}-\epsilon}{1+r}$.
6. For $a \leq \tilde{a}(1)$: $c = \epsilon + a(1 + r) - \underline{a}$.

Endogenous Grid Points Algorithm

1. Construct a grid of assets today, $a \in A$, and tomorrow $\mathbf{a} \in \mathcal{A}$ with $\mathbf{a}_1 = \underline{a}$.
2. Guess the policy function $c(\mathbf{a}, \epsilon)$.
3. Solve $B(\mathbf{a}, \epsilon) = (1+r)\beta \sum_{j=1}^{N} \pi(\epsilon'|\epsilon) U'(c(\mathbf{a}, \epsilon'))$.
4. Solve for implied consumption today $c(\tilde{a}, \epsilon) = B(\mathbf{a}, \epsilon)^{-1/\gamma}$.
5. From budget constraint: $\tilde{a} = \frac{c+\mathbf{a}-\epsilon}{1+r}$.
6. For $a \leq \tilde{a}(1)$: $c = \epsilon + a(1+r) - \underline{a}$.
7. Interpolate $c(a, \epsilon)$ on $c(\tilde{a}, \epsilon)$.
8. Replace initial guess and iterate until convergence.

Endogenous Grid Points Value Function

- Sometimes, we are not only interested in the policy, but also the value function.
- We can use the insight of EGM, to iterate on the value function.

$$\frac{\partial V(a,\epsilon)}{\partial a'} = \frac{\partial U(c)}{\partial c}\frac{\partial c}{\partial a'} + \beta\frac{\partial \mathbb{E}V(a',\epsilon')}{\partial a'} = 0$$

$$U'(c) = \beta\frac{\partial \mathbb{E}V(a',\epsilon')}{\partial a'}$$

Endogenous Grid Points Value Function II

1. Construct a grid of assets today, $a \in A$, and tomorrow $\mathbf{a} \in \mathcal{A}$.
2. Guess the expected value function tomorrow $\hat{V}(\mathbf{a}, \epsilon) = \beta \sum_{j=1}^{N} \pi(\epsilon'|\epsilon) V(\mathbf{a}, \epsilon')$.
3. Solve $B(\mathbf{a}, \epsilon) = \frac{\hat{V}(\mathbf{a}, \epsilon')}{\partial \mathbf{a}}$.
4. Solve for implied consumption today $c(\tilde{a}, \epsilon) = B(\mathbf{a}, \epsilon)^{-1/\gamma}$.
5. From budget constraint: $\tilde{a} = \frac{c + \mathbf{a} - \epsilon}{1 + r}$.
6. For $a \leq \tilde{a}(1)$: $c = \epsilon + a(1 + r) - \underline{a}$.
7. Interpolate $c(a, \epsilon)$ on $c(\tilde{a}, \epsilon)$.
8. From budget constraint: $a'(a, \epsilon) = (1 + r)a - c(a, \epsilon) + \epsilon$.
9. Obtain $\hat{V}(a', \epsilon)$ by interpolating on $\hat{V}(\mathbf{a}, \epsilon)$.
10. Update value function: $V(a, \epsilon) = U(c) + \hat{V}(a', \epsilon)$.

Endogenous Grid Points, Two Choices

Barillas and Fernandez-Villaverde (2007) study problem similar to:

$$
\begin{aligned}
V(a,z) &= \max_{c,a',l}\left\{\frac{\left(c^{\theta}(1-l)^{1-\theta}\right)^{1-\tau}}{1-\tau}+\beta\mathbb{E}V(a',z')\right\}\\
z' &= \rho z+\epsilon'\\
a'+c &= (1+r)a+l\exp(z)\\
a' &\geq 0
\end{aligned}
$$

Endogenous Grid Points, Two Choices II

First order condition for asset next period:

$$\theta \frac{\left(c^{\theta}(1-l)^{1-\theta}\right)^{1-\tau}}{c} = \beta \frac{\partial \mathbb{E}\{V(a', z')\}}{\partial a'} := \hat{V}$$

This can be solved for consumption today:

$$c_t = \left[\frac{\hat{V}}{\theta(1-l_t)^{(1-\theta)(1-\tau)}}\right]^{\frac{1}{\theta(1-\tau)-1}}$$

Thus, as before, knowing $\hat{V}$ (and l_t) yields a solution for consumption today.

Endogenous Grid Points, Two Choices III

First order condition for labor implies:

$$\frac{1-\theta}{\theta}\frac{c_t}{1-l_t} = z_t$$

Knowing consumption, we can solve for labor.

Endogenous Grid Points, Two Choices Algorithm

1. Guess optimal policy for labor: $\phi_l(a, z)$.
2. Solve the EGM algorithm for $\phi_c(a, z)$.
3. Solve for $\phi_l(a, z)$ and update policy.
4. Iterate until convergence.

Consider again a simple household problem:

$$
\begin{aligned}
V(a,\epsilon) &= \max_{c,a'}\Big\{U(c) + \beta\mathbb{E}V(a',\epsilon')\Big\} \\
c + a' &= \epsilon + a(1+r) \\
a' &\geq \underline{a} \\
\pi_{jk}&(\epsilon' = \epsilon^j | \epsilon = \epsilon^k)
\end{aligned}
$$

Take an asset grid of 5000 points and a productivity grid of 3 points the problem takes:

- 147 seconds to solve on an $i7-10700$ 2.9 GH processor when written with loops.
- , for reasons explained below, 25 seconds when fully vectorized.

Parallelizing Your Code

- Many loop operations can be done simultaneously, instead of sequentially:

  Solve value function at each grid point.

  Simulate a Markov process.

- There are two broad types of parallizations:

  Computer has several cores (local).

  Server has several computers (cluster).

Matlab Parallelizing

Using several cores:
parpool('local',6)
parfor $i = 1 : 10$
$f(i) = VFI(i)$;
end
poolobj = gcp('nocreate');
delete(poolobj);

Using a cluster:
parpool('Name',22, 'AttachedFiles', {'VFI.m' 'FOC.m'})
parfor $i = 1 : 10$
$f(i) = VFI(i)$;
end
poolobj = gcp('nocreate');
delete(poolobj);

Efficiency of Parallelizing

The speed gain is significantly below $1/N$:

- It can be even considerably slower than non-paralization.
- As memory needs to be passed to each worker at the same time, you may run into memory issues.
- Parallization creates overhead communication between Matlab and the different cores.
- Often, the efficiency loss is smallest when every single computation takes time.
- Because how things are organized on the RAM, it can matter over which dimension you loop.
- My computer has 8 cores. Using 6, computation time drops from 147 seconds to 69 seconds.

Going beyond Matlab

Interpreted Languages

Matlab is what is called an interpreted language:

- What does $B = sum(A)$ mean in *Matlab*?

  Reads the expression.

  Checks what A is (one or more dimensions?)

  Check, what $sum()$ does for this type of argument.

  Check if B exists or if it needs to be created.

- This is why loops are slow in *Matlab*.

This is different from compiled languages. Two famous exaples are Fortran and C++:

- What does $B = sum(A)$ mean in *Fortran*?

- At execution time, the compiler has translated this statement into machine code.

  It has determined what A is.

  It has made sure, A is a data type that $sum()$ can be applied to.

  It has made sure that B has been declared and can contain the result of $sum(A)$.

  The computer than just executes instruction by instruction.

Compiled Code in *Matlab*

- Some *Matlab* functions are compiled code.
- *Matlab* provides possibility to include your own compiled code as .mex functions.

  Either *C++* or *Fortran*.

  Unfortunately the documentation is poor.
- This provides the opportunity to outsource computational expensive routines.
- While keeping the advantages of *Matlab*.
- Debugging is tedious.

Compiled Code in *Matlab* II

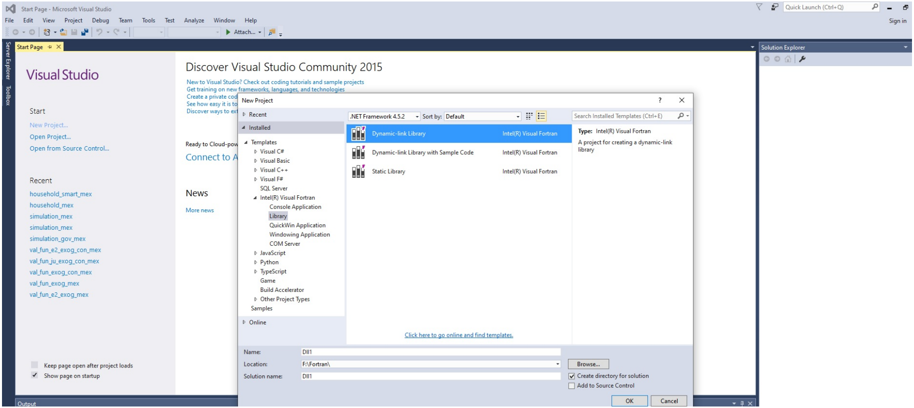
- Here, I show you how to use *Windows Visual Studio* together with an *Intel* compiler.

- There are also free of charge compilers (*Windows Visual Studio Community* is free of charge).

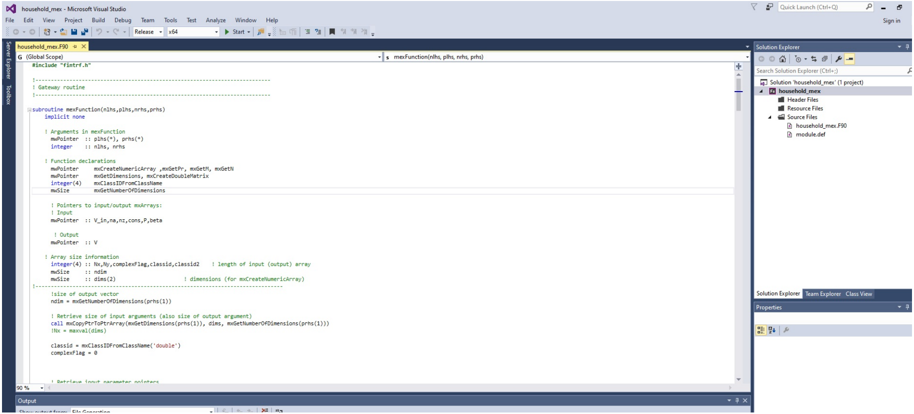
- Linux systems (Ubuntu) have compilers already installed

  Our cluster runs on Ubuntu!

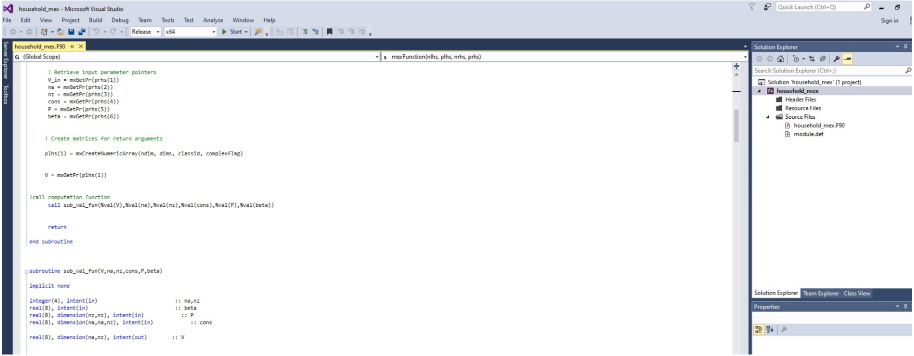
Visual Studio I



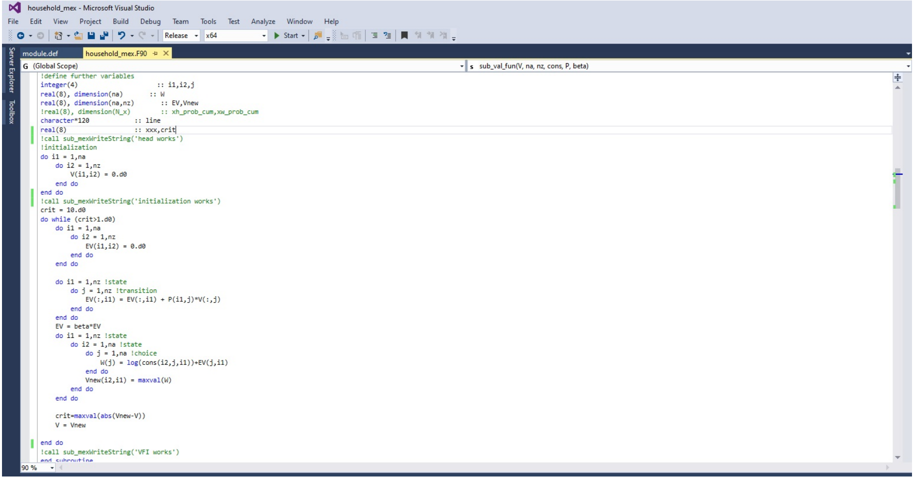
Visual Studio II



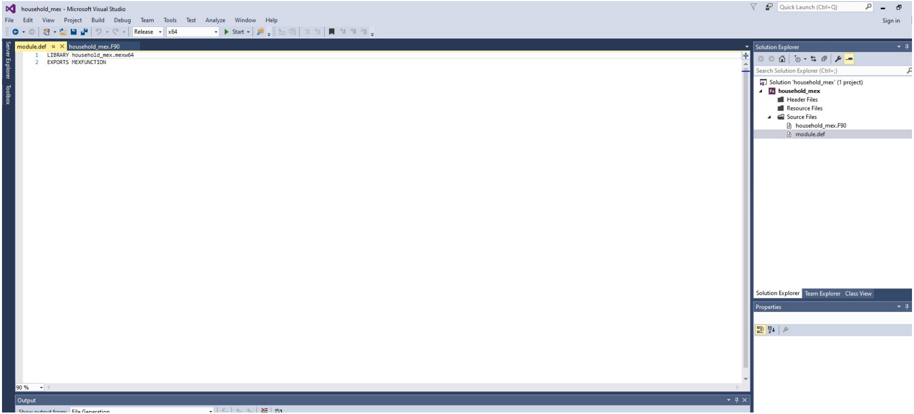
Visual Studio III



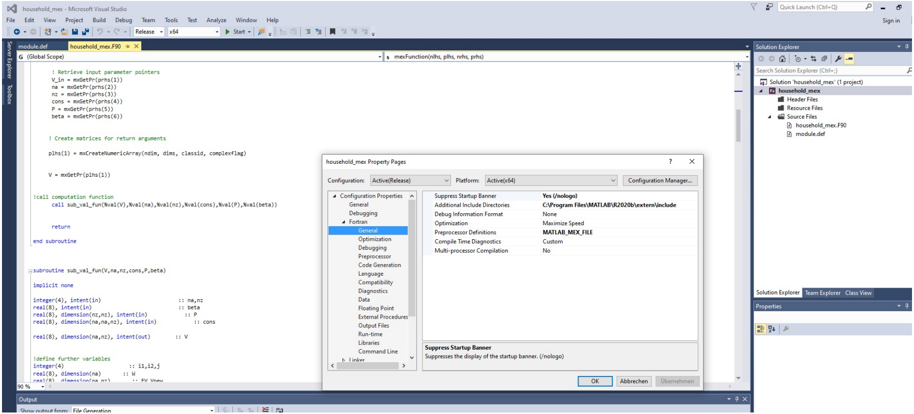
Visual Studio IV



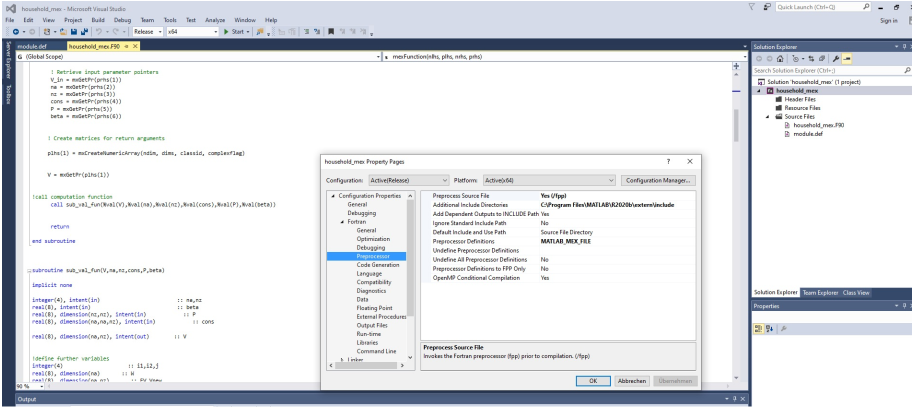
Visual Studio V



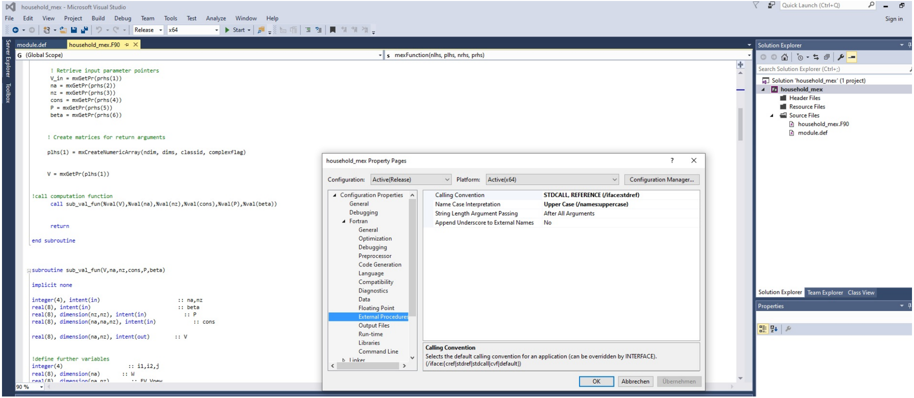
Visual Studio VI



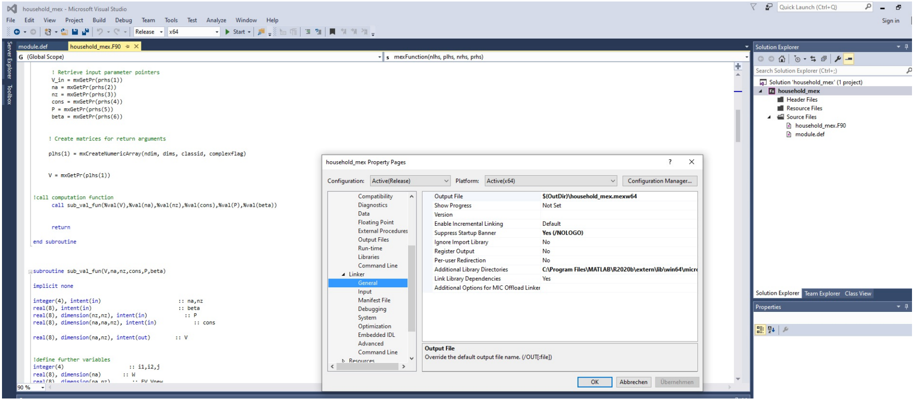
Visual Studio VII



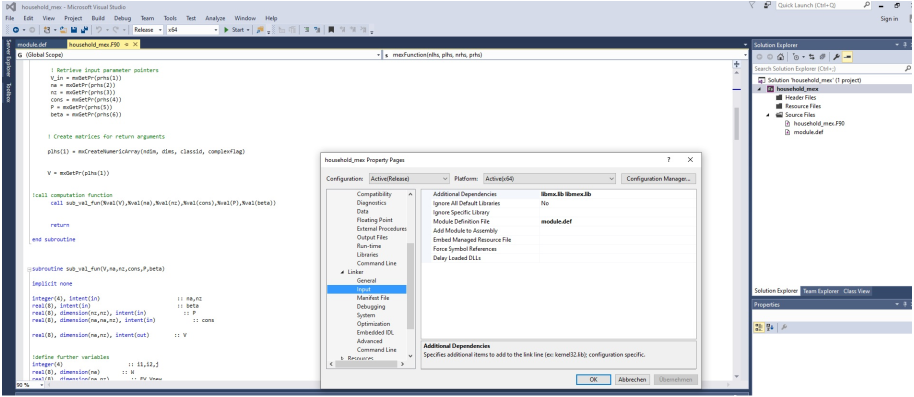
Visual Studio IIX



Visual Studio IX



Visual Studio X



Mex-file Computation Time

- Solving the household problem with a mex-file takes 27 seconds.
- Much faster than the 147 seconds in *Matlab*.
- It is still slower than the 25 seconds from the fully vectorized version in *Matlab*. The reason is communication cost.
- However, full vectorization is often not feasible:
  - Monte Carlo simulations.
  - Large state spaces imply huge matrices stretching the RAM. A $10000X10000$ matrix is already 3.9 GB with *double* precision and 2.6 with *single* precision.
- Non-paralized code is easier to read.
- Non-paralized code can save on non-necessary computations.

Saving on Non-Necessary Computations

- In our problem, most computations are not necessary.
- We know the policy function is monotone and the return function is concave.
- In *Matlab* a non-paralized smart code takes 0.16 seconds.
- a mex-file takes 0.04 seconds.
- These speed gains are extreme due to the regularity of the problem but you often know (or suspect) something about your problem.

From the CPU to the GPU

- So far, we ask our computer to solve the problem on the computer processing unit (CPU).
- CPU's are designed to solve complex problems.
- It turns our, simpler problems can be more efficiently handled by the graphical processing unit (GPU).
- A GPU has a large amount of cores but only limited memory.
- I have a *NVIDIA GeForce RTX 3060*. This GPU has 3584 cores with 12GB RAM.
- Hence, the GPU is only useful for tasks that can be paralized.

From the CPU to the GPU

- *CUDA* allows you to write your own programs based on *C++* as .cu files.
- You can embed these in *Matlab* as .mex files (Matlab: mexcuda) or .ptx files (Visual Studio).
- This, however, requires some advanced programing knowledge.
- The VFI-toolkit does it for you for a particular class of problems.
- With my *NVIDIA GeForce RTX 3060*, the earlier problem takes 3 seconds (down from 147 with the CPU).

More on GPU Code

- Only 1024 threads can access what is called "shared memory" posing a limit to evaluate $\max(abs(Vnew - Vold))$. Hence, $\max(abs(Vnew - Vold))$ needs to be evaluated on the Host. When Matlab is the host, this produces overhead.
- It must be possible to paralize the function. This implies, you cannot exploit the monotonicity of the policy function.
- In the present case, we can still exploit concavity of the value function.

Summary of Speed

- 147 seconds with for loops in *Matlab*.
- 69 seconds with parfor loop and 6 workers in *Matlab*.
- 25 seconds with vectorization in *Matlab*.
- 27 seconds with *Fortan* mex-file.
- 3 seconds with the VFI-toolkit (GPU).
- 0.37 seconds with smart code on the GPU.
- 0.16 seconds with smart code in *Matlab*.
- 0.04 seconds with smart code and a *Fortan* mex-file.

More on GPU Programming and Overhead

- When working with the GPU, passing information between the "Host" and the "Device" creates overhead costs. Also Matlab creates overhead costs.
- Hence, you want to write the CUDA code as "complete" as possible.
- To understand the role of overhead, the next slide shows speeds when I decrease the asset grid size to 330 (but decrease the convergence criteria). I.e., every function evaluation is more simple but we do more.

Summary of Speed with fewer Grid Points

- 9.65 seconds with for loops in *Matlab*.
- 9.82 seconds with parfor loop and 6 workers in *Matlab*.
- 2.14 seconds with vectorization in *Matlab*.
- 1.82 seconds with *Fortan* mex-file.
- 2.01 seconds with the VFI-toolkit (GPU).
- 0.17 seconds with smart code in *Matlab*.
- 0.03 seconds with smart code and a *Fortan* mex-file.
- 0.001 seconds with smart code and "complete" code on the GPU.
- There is a trade-off between paralization and overhead!

Accuracy of Numerical Approximation

Accuracy of Numerical Approximation

We would like to assess the accuracy of numerial solutions. One possibility are normalized Euler equation errors:

$$EE = \frac{u'(c_t) - \beta \mathbb{E} R_{t+1} u'(c_{t+1})}{u'(c_t)}$$

In the Neo-classical growth model:

$$EE(k_t, z_t) = 1 - \frac{(\beta \mathbb{E}(\alpha Z_{t+1} \phi(k_t, z_t)^{\alpha-1} + 1 - \delta) u'(c_{t+1}))^{-1/\gamma}}{c_t}$$

- The error is defined at each grid point k_i, z_j.
- It has a natural interpretation:

  If $EE_{i,j} = 0.01$, the agent makes a 1$ mistake for every 100$ spend.

Dynamic Euler Equation Error

- Euler equation error are a one period ahead error.
- But (small) errors may accumulate over time.

Simulate two time series with T periods:

1. Simulate the series using policy function for consumption.
2. Simulate an alternative series:

   Compute rhs of Euler equation using numerical integration (g).

   Solve for $c_t = g^{-1/\gamma}$.

   Solve for $k_{t+1} = z_t k_t^{\alpha} + (1-\delta)k_t - c_t$.
3. Compare the two series.

References

Barillas, F. and J. Fernandez-Villaverde (2007): "A Generalization of the Endogenous Grid Method," *Journal of Economic Dynamics and Control*, 31, 2698–2712.

Carroll, C. D. (2006): "The Method of Endogenous Gridpoints for Solving Dynamic Stochastic Optimization Problems," *Economics Letters*, 91, 312–320.

Judd, K., L. Maliar, S. Maliar, and R. Valero (2014): "Smolyak Method for Solving Dynamic Economic Models: Lagrange Interpolation, Anisotropic Grid and Adaptive Domain," *Journal of Economic Dynamics and Control*, 44, 92–123.

Tsao, C.-S. and J. Tsitsiklis (1991): "An Optimal One-Way Multigrid Algorithm for Discrete Time Stochastic Control," *IEEE Transaction on Automatic Control*, 36, 898–914.