

This document aims to outline the modifications we would need added to the configuration of models in Dolo yaml files in order for those config files to have the information necessary for them to be used for solving the kinds of models we currently solve in HARK.

1 The Problem - in Bellman Form

The usual analysis of dynamic stochastic programming problems packs a great many events (intertemporal choice, stochastic shocks, intertemporal returns, income growth, and more) into a small number of steps and variables. For the detailed analysis here, we will be careful to disarticulate everything that happens in the problem explicitly into separate steps so that each element can be scrutinized and understood in isolation.

We are interested in the behavior a consumer who begins stage t with a certain amount of ‘capital’ \mathbf{k}_t , which is immediately rewarded by a return factor R_t with the proceeds deposited in a bank account balance:

$$\mathbf{b}_t = \mathbf{k}_t R_t. \quad (1)$$

Simultaneously with the realization of the capital return, the consumer also receives noncapital income \mathbf{y}_t , which is determined by multiplying the consumer’s ‘permanent income’ \mathbf{p}_t by a transitory shock $\boldsymbol{\theta}_t$:

$$\mathbf{y}_t = \mathbf{p}_t \boldsymbol{\theta}_t \quad (2)$$

whose expectation is 1 (that is, before realization of the transitory shock, the consumer’s expectation is that actual income will on average be equal to permanent income \mathbf{p}_t).

The combination of bank balances \mathbf{b} and income \mathbf{y} define’s the consumer’s ‘market resources’ (sometimes called ‘cash-on-hand’, following Deaton (1992)):

$$\mathbf{m}_t = \mathbf{b}_t + \mathbf{y}_t, \quad (3)$$

which are available to be spent on consumption \mathbf{c}_t .

The consumer’s goal is to maximize discounted utility from consumption over the rest of a lifetime whose last stage is date T :

$$v_t(\mathbf{p}_t, \mathbf{m}_t) = \max_{\{\mathbf{c}\}_t^T} u(\mathbf{c}_t) + \mathbb{E}_t \left[\sum_{n=1}^{T-t} \beta^n \mathcal{L}_t^{t+n} \hat{\beta}_t^{t+n} u(\mathbf{c}_{t+n}) \right] \quad (4)$$

$$\begin{aligned} \mathbf{a}_t &= \mathbf{m}_t - \mathbf{c}_t \\ \mathbf{p}_{t+1} &= \mathcal{G}_{t+1} \mathbf{p}_t \Psi_{t+1} \\ \mathbf{y}_{t+1} &= \mathbf{p}_{t+1} \boldsymbol{\theta}_{t+1} \\ \mathbf{m}_{t+1} &= R \mathbf{a}_t + \mathbf{y}_{t+1} \end{aligned}$$

\mathcal{L}_t^{t+n}	: probability to \mathcal{L} ive until age $t + n$ given alive at age t
$\hat{\beta}_t^{t+n}$: age-varying discount factor between ages t and $t + n$
Ψ_t	: mean-one shock to permanent income
\beth	: time-invariant ‘pure’ discount factor

Transitory and permanent shocks are distributed as follows:

$$\Xi_s = \begin{cases} 0 & \text{with probability } \wp > 0 \\ \theta_s/\wp & \text{with probability } (1 - \wp), \text{ where } \log \theta_s \sim \mathcal{N}(-\sigma_\theta^2/2, \sigma_\theta^2) \end{cases} \quad (5)$$

$$\log \Psi_s \sim \mathcal{N}(-\sigma_\Psi^2/2, \sigma_\Psi^2)$$

where \wp is the probability of unemployment (and unemployment shocks are turned off after retirement). The problem can be normalized by the level of permanent income \mathbf{p} , yielding the resulting Bellman representation:

$$\begin{aligned} v_t(m_t) &= \max_{c_t} \quad u(c_t) + \beth \mathcal{L}_{t+1} \hat{\beta}_{t+1} \mathbb{E}_t[(\Psi_{t+1} \mathcal{G}_{t+1})^{1-\rho} v_{t+1}(m_{t+1})] \\ &\text{s.t.} \\ a_t &= m_t - c_t \\ m_{t+1} &= a_t \underbrace{\left(\frac{R}{\Psi_{t+1} \mathcal{G}_{t+1}} \right)}_{\equiv \mathcal{R}_{t+1}} + \theta_{t+1} \end{aligned}$$

Although our focus so far has been on the consumer’s consumption problem once m is known, for articulating the steps of the computational solution it will be useful to distinguish a sequence of three steps (we use the word ‘steps’ to capture the proposition that we are not really thinking of these as occurring at different moments in time – only that we are putting the things that happen within a given moment into an ordered sequence). The first step captures calculations that need to be performed before the shocks are realized, the middle step is after shocks have been realized but before the consumption decision has been made (this corresponds to the timing of v in the treatment above), and the final step captures the situation *after* the consumption decision has been made.

But even this breakdown of the problem is insufficient for the general case. As ? shows, some problems with multiple state or multiple control variables (or both) can be solved much more efficiently if the solution machinery lays the problem out as a sequence of single-control-variable choices. For example, one might have a consumption stage that results in some stock of assets, and then a portfolio choice stage that determines the allocation of those assets between investments in the risky and the safe asset.

We therefore want to allow for the problem in each period to have as many ‘stages’ as may be convenient for solving any particular model – say, a labor supply stage followed by a consumption stage followed by a portfolio choice stage.

We need to define notation for these three steps within a stage. We will use s^- as

a marker for the step before shocks have been realized, and s^+ as the indicator for the situation once the choice has been made, implicitly leaving the unmarked s to indicate the step in which the decision is made.

$$v_{s-}(k_s) = \mathbb{E}_{s-}[\overbrace{v_s(k_s \mathcal{R}_s + \theta_s)}^{=m_s}] \quad (6)$$

$$v_s(m_s) = u(c_s(m_s)) + v_{s+}(a_s) \quad (7)$$

$$v_{s+}(a_s) = \beta v_{t_{-}+1}(\underbrace{k_{s+1}}_{=a_s}) \quad (8)$$

If we turn off the mortality shocks ($\mathcal{L} = 1$) and family-size shocks, and redefine the pure time preference factor as $\beta = \beta$, the problem has first order condition

$$u'(c) = \beta R v'_{s+}(m - c). \quad (9)$$

2 Creating An Instance of a Model

The first thing to do in building a model would probably be to create an empty model framework, perhaps with a command like:

```
mymodl = create_empty_model('modelsetup.yaml')
```

where any information that was globally true for the model (like, maybe, its name, some links to external resources describing it, etc) would be encapsulated in ‘modelsetup.yaml’).

The next thing to do would be to define at least one ‘period’ in the model, where a period describes a set of things that are envisioned to happen in a single time step

```
mymodl = backward_create_a_period('periodsetup.yaml')
```

where any information that was globally true for the period would be defined.

We want to define a ‘period’ as a standalone environment that may consist of several stages, each of which may have several steps, but in which all steps and stages share a common set of resources, which include parameterizations, symbol definitions, calibrations, etc. These common resources would be defined in a standalone yaml file – say, `bufferstockperiod.yaml` which would get digested as the first thing to be done as the construction of the period commences. The file might look something like this:

```
symbols:
  statesBegOfStge: [k]
  statesMidOfStge: [m]
  statesEndOfStge: [a]
  controls: [c]
```

```

valuesBegOfStge: [vBegOfStge]
valuesMidOfStge: [vMidOfStge]
valuesEndOfStge: [vEndOfStge]
parameters: [beta          # time pref
             ,rho          # RRA
             ,Lambda       # survival probability
             ,R            # return factor
             ,G            # permanent income growth factor
             ,sigma_ltheta # std of tran shock
             ,sigma_lpsi   # std of perm shock
]

calibration:
  beta:      0.96
  G:         1.03
  rho:       2.0
  R:         1.04
  Lambda:    1.00
  sigma_ltheta: 0.1
  sigma_lpsi: 0.1
  min_m:     0.0
  max_m:     50.0
  min_a:     0.0
  max_a:     50.0
  min_k:     0.0
  max_k:     50.0

domain:
  k: [0.0, max_k]
  m: [0.0, max_m]
  a: [0.0, max_a]

exogenous: !Normal
  Sigma: [[sigma_ltheta^2      , 0. ],
          [0.                  , sigma_lpsi^2]]
  Mu:    [-(sigma_ltheta^2)/2 , (sigma_lpsi^2)/2]

options:
  grid: !Cartesian
  orders: [1000]

```

Next, one would construct at least one ‘stage’, which consists of a set of steps that instantiate the structure of the model. The task of defining the solution would then be to define what happens serially backwards as each step gets added.

So, for example, for the buffer stock consumption problem we would have a file that defines the consumption optimization step (call the file `coptstep.yaml`) of a problem, which would have an `equations` block that might look something like this

```
value: |
  vMidOfStge = (c^(1-rho))/(1-rho) + beta*Lambda*vEndOfStge

transition: |
  a = m - c

arbitrage: |
  c^(-rho) - beta*Lambda*Rfree*vEndOfStge.deriv['a']
```

where `vEndOfStge.deriv['a']` is a placeholder for whatever the correct notation would be to retrieve the derivative of `vEndOfStge` with respect to a and the assumption is that the equation defined as `arbitrage` presents an expression whose value should be zero at the optimum. The arbitrage equation therefore is equivalent to a rewritten version of our FOC (9):

$$u^c(c) - v_{s+}^a(m - c) = 0 \quad (10)$$

The result of processing this step should be the construction of the optimized value of c for each of the points in the state space defined as the input state space for each of the values of m constructed previously in the definition step, and construction of the `vMidOfStge` value function for the same states. (For the moment, we are not using EGM, because it is not clear how the syntax for EGM would work for dolo).

The next (earlier) step would compute the expectations that would contain the information necessary to construct the beginning-of-period value function. That file would need to define an operation that does not exist in dolo (as we currently understand it). That operation is basically to calculate the expectations of the value function (and other interesting objects) at the gridpoints already defined for the state variables.

The `equations` block of `vexp.yaml` might look something like this:

```
transition: |
  m = k*R/(G*exp(lpsi)) + exp(ltheta)

expectorate: |
  vBegOfStge = vMidOfStge
```

where the `expectorate` operation would be defined to construct `vBegOfStge` by calculating the expectations at the grid of points for k defined in the `stagesetup.yaml` file.

```
transition:
  a[stage-1] <-> k[stage]
  vEndOfStge[stage-1] <-> vBegOfStge[stage]
```

and possibly it would need to have further explicit information about things like how to map whatever grid exists for k in the current period to the corresponding grid for a in the prior period, and any information needed to map between the value functions.

With these files in place, the way to solve a period of the model might be something like this:

```
modl.add_period_preceding_existing_solutions = darkparse_period_setup('setupstage.y
modl.add_stage_to_current_period = darkparse_stagesetup('stagesetup.yaml')
modl.add_step_in_current_stage = darkparse('stagesetup.yaml')
modl.add_step_in_current_stage = darkparse('coptstep.yaml')
modl.add_step_in_current_stage = darkparse('expectorate.yaml')
modl.add_step_in_current_stage = darkparse('transit.yaml')
```

and at the end of this process the `modl` object would be ready to have another period added.

The payoff to all this work is that it should allow us to add components modularly. Suppose, for example, that we had the code to solve a portfolio choice problem of choosing the optimal share of risky versus safe assets; assuming that the risky return is realized at the same moment as the shocks to labor income (so that we can potentially allow the asset market return to be correlated with the labor supply decision – say, periods with bad stock market performance are also periods when unemployment is higher), that decision problem should be insertable into the structure above between the `transit` step and the `expectorate` step:

```
modl.add_period_preceding_existing_solutions()
modl.add_stage_to_current_period()
modl.add_step_to_current_stage = darkparse('setup.yaml')
modl.add_step_to_current_stage = darkparse('coptstep.yaml')
modl.add_step_to_current_stage = darkparse('expectorate.yaml')
modl.add_step_to_current_stage = darkparse('portfoliostep.yaml')
modl.add_step_to_current_stage = darkparse('transit.yaml')
```

(This would also require specification of the stochastic process for the rate of return; following the scheme sketched above, this would presumably be located in the `stagesetup.yaml` file, though possibly we might decide that it makes more sense to describe the stochastic shocks in the `expectorate.yaml` file that calculates the expectations over those shocks).

3 Maybe there should be two kinds of transitions

Arguably, the transition between periods is such a different phenomenon than within-period transitions that the best way to deal with it might be to declare that each ‘period’ is defined by a set of within-period steps, and a single transition equation that connects the period to its predecessor. In that case, the pseudocode for defining the solution to a period might look like this:

```
modl.add_period_preceding_existing_solutions()  
modl.add_step_in_current_period = darkparse('setup.yaml')  
modl.add_step_in_current_period = darkparse('coptstep.yaml')  
modl.add_step_in_current_period = darkparse('expectorate.yaml')  
modl.add_step_in_current_period = darkparse('portfoliostep.yaml')
```

followed by

```
modl.add_transition_to_prior_period = darktransparse('transit_to_prior_period.yaml')
```

.