dolo Documentation

Release 0.4.9.2

Pablo Winant

January 21, 2015

Contents

CHAPTER 1

Index

1.1 Introduction: What is dolo ?

1.2 Installation

Dolo can be installed using the standard python command `pip`. You can choose between one of the three following options.

- To install last version from *PyPI* directories:

```
`pip install dolo`
```

- To install last version from github (requires git on your computer):

```
pip install git+https://github.com/albop/dolo.git
```

- To install *dolo* from your own clone of the dolo repository. Type from this directory:

```
pip install -e .
```

The option -e tells the installer to make symlinks so that the installed version is automatically updated when a new version is pulled.

1.3 Quick Tutorial

1.3.1 Solving the rbc model

This worksheet demonstrates how to solve the RBC model with the dolo library and how to generates impulse responses and stochastic simulations from the solution.

- This notebook is distributed with dolo in : `examples\notebooks\`. The notebook was opened and run from that directory.
- The model file is in : `examples\global_models\`

First we import the dolo library.

```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
from dolo import *
```

The RBC model is defined in a YAML file which we can read locally or pull off the web.

```
filename = ('https://raw.githubusercontent.com/EconForge/dolo'
            '/master/examples/models/rbc.yaml')

#filename='../models/rbc.yaml'

pcat(filename)
```

`yaml_import(filename)` reads the YAML file and generates a model object.

```
model = yaml_import(filename)
```

The model file already has values for steady-state variables stated in the calibration section so we can go ahead and check that they are correct by computing the model equations at the steady state.

```
model.residuals()
```

```
OrderedDict([('transition', array([  0.00000000e+00,   2.50466314e-13])), ('arbitrage', array([ -1.0
```

Printing the model also lets us have a look at all the model equations and check that all residual errors are 0 at the steady-state, but with less display prescision.

```
print( model )
```

```
Model object:
------------

- name: "RBC"
- type: "fga"
- file: "https://raw.githubusercontent.com/EconForge/dolo/master/examples/models/rbc.yaml

- residuals:

    transition
        1   : 0.0000 : z = (1-rho)*zbar + rho*z(-1) + e_z
        2   : 0.0000 : k = (1-delta)*k(-1) + i(-1)

    arbitrage
        1   : 0.0000 : 1 = beta*(c/c(1))**(sigma)*(1-delta+rk(1))    | 0 <= i <= inf
        2   : 0.0000 : w - chi*n**eta*c**sigma                   | 0 <= n <= inf

    auxiliary
        1   : 0.0000 : c = z*k**alpha*n**(1-alpha) - i
        2   : 0.0000 : rk = alpha*z*(n/k)**(1-alpha)
        3   : 0.0000 : w = (1-alpha)*z*(k/n)**(alpha)

    value
        1   : 0.0000 : V = log(c) + beta*V(1)
```

Next we compute a solution to the model using a second order perturbation method (see the source for the approximate_controls function). The result is a decsion rule object. By decision rule we refer to any object is callable and maps states to decisions. This particular decision rule object is a TaylorExpansion (see the source for the TaylorExpansion class).

```
dr_pert = approximate_controls(model, order=2)
```

```
There are 2 eigenvalues greater than 1. Expected: 2.
```

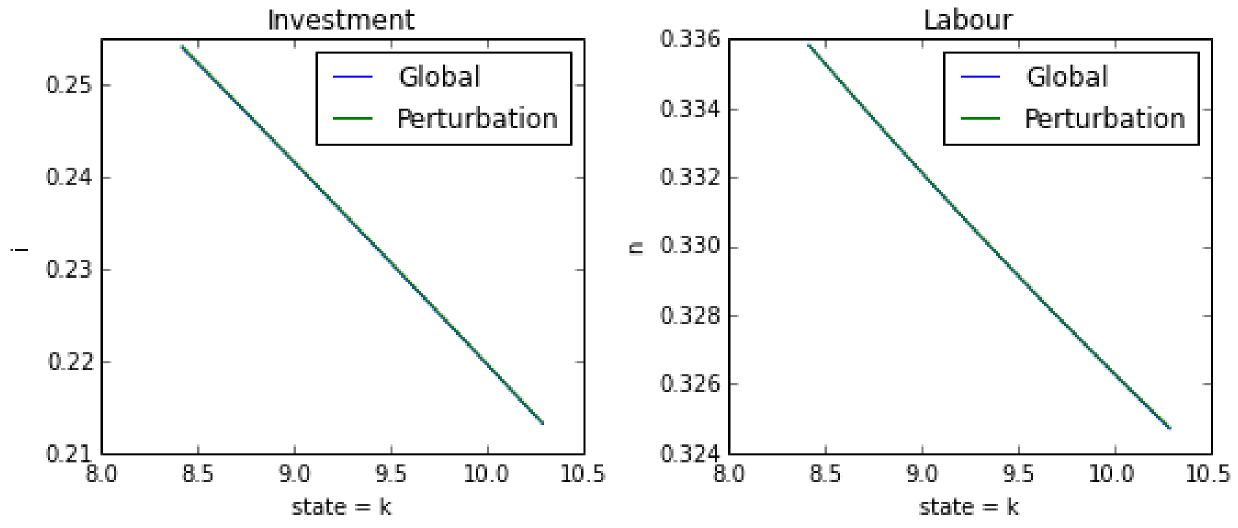
We now compute the global solution (see the source for the time_iteration function). It returns a decision rule object of type SmolyakGrid (see the source for the SmolyakGrid class).

```
dr_global = time_iteration(model, pert_order=1, smolyak_order=3)
```

1.3.2 Decision rule

Here we plot optimal investment and labour for different levels of capital (see the source for the plot_decision_rule function).

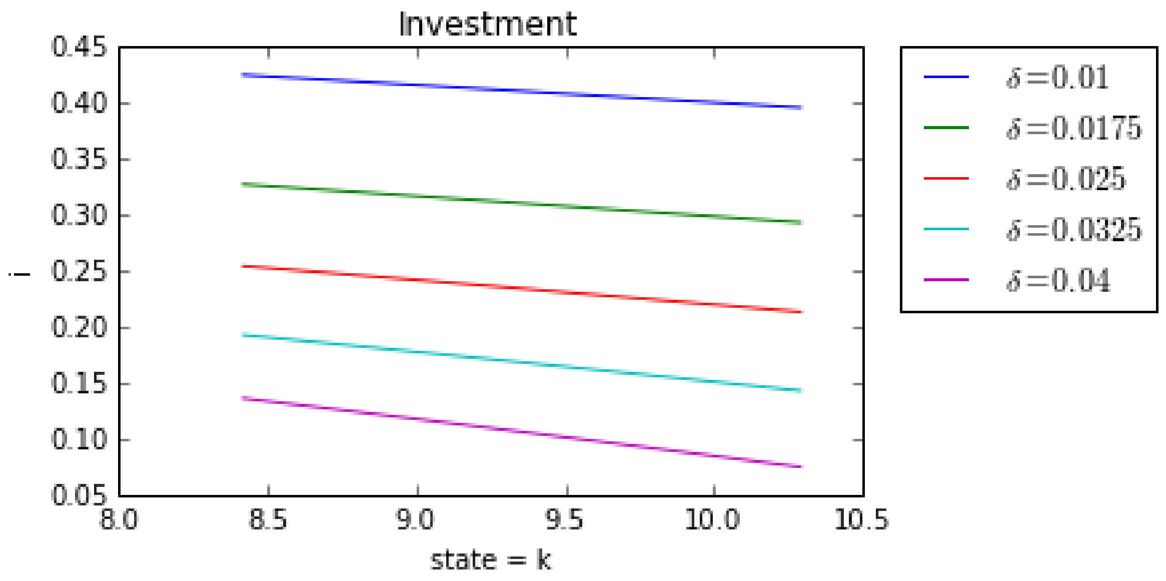
```
Decisionbounds = [dr_global.smin[1], dr_global.smax[1]]

figsize(8,3.5)

subplot(121)
plot_decision_rule(model, dr_global, 'k', 'i', label='Global', bounds=bounds)
plot_decision_rule(model, dr_pert, 'k', 'i', label='Perturbation', bounds=bounds)
ylabel('i')
title('Investment')
legend()

subplot(122)
plot_decision_rule(model, dr_global, 'k', 'n', label='Global', bounds=bounds)
plot_decision_rule(model, dr_pert, 'k', 'n', label='Perturbation', bounds=bounds)
ylabel('n')
title('Labour')
legend()

tight_layout()
show()
```



It would seem, according to this, that second order perturbation does very well for the RBC model. We will revisit this issue more rigorously when we explore the deviations from the model's arbitrage section equations.

Let us repeat the calculation of investment decisions for various values of the depreciation rate, δ. Note that this is a comparative statics exercise, even though the models compared are dynamic.

```
original_delta=model.calibration_dict['delta']

drs = []
delta_values = linspace(0.01, 0.04,5)
for val in delta_values:
    model.set_calibration(delta=val)
    drs.append(approximate_controls(model, order=2))


figsize(5,3)

for i,dr in enumerate(drs):
    plot_decision_rule(model, dr, 'k', 'i',
                       label='$\delta={}$'.format(delta_values[i]),
                       bounds=bounds)

ylabel('i')
title('Investment')
legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
show()

model.set_calibration(delta=original_delta)

There are 2 eigenvalues greater than 1. Expected: 2.
There are 2 eigenvalues greater than 1. Expected: 2.
There are 2 eigenvalues greater than 1. Expected: 2.
There are 2 eigenvalues greater than 1. Expected: 2.
There are 2 eigenvalues greater than 1. Expected: 2.
```



We find that more durable capital leads to higher steady state investment and slows the rate of convergence for capital (the slopes are roughly the same, which implies that relative to steady state capital investment responds stronger at higher δ, this in addition to the direct effect of depreciation).

1.3.3 Use the model to simulate

We will use the deterministic steady-state as a starting point.

```
s0 = model.calibration['states']
print(str(model.symbols['states'])+'='+str(s0))
```

```
['z', 'k']=[ 1.          9.35497829]
```

We also get the covariance matrix just in case. This is a one shock model so all we have it the variance of e_z.

```
sigma2_ez = model.covariances
sigma2_ez
```

```
array([[ 0.000256]])
```

Impulse response functions

Consider a 10% shock on productivity.

```
s1 = s0.copy()
s1[0] *= 1.1
print(str(model.symbols['states'])+'='+str(s1))
```

```
['z', 'k']=[ 1.1         9.35497829]
```

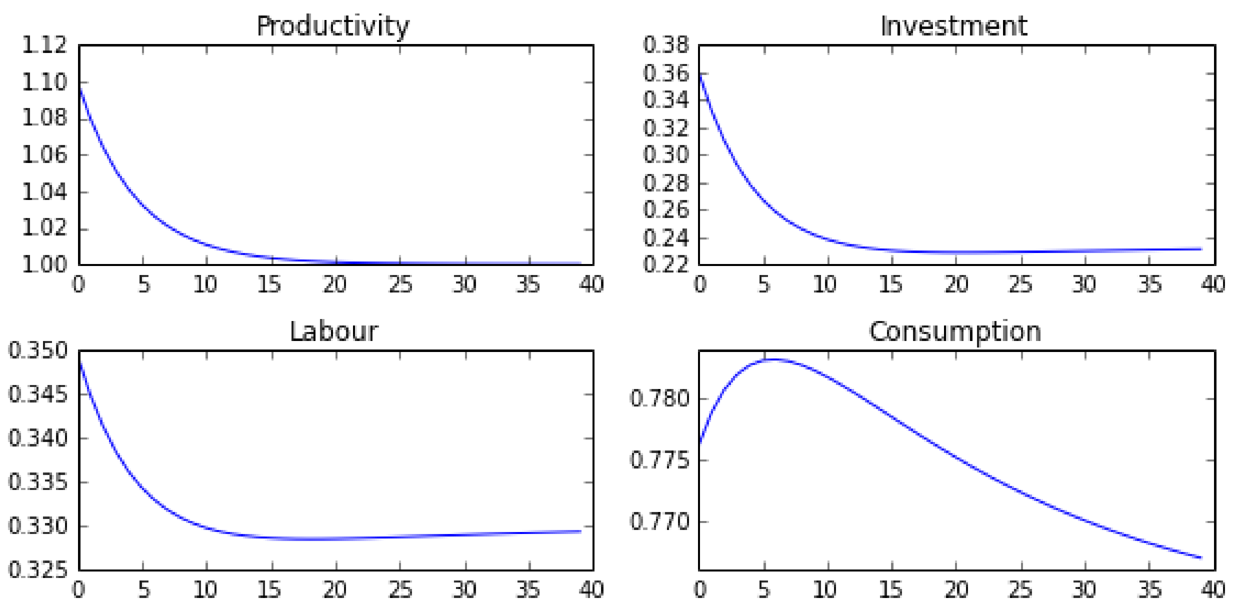
The `simulate` function is used both to trace impulse response functions and compute stochastic simulations. Choosing `n_exp>=1`, will result in that many "stochastic" simulations. With `n_exp = 0`, we get one single simulation without any stochastic shock (see the source for the simulate function). The output is a panda table of size $H \times n_v$ where n_v is the number of variables in the model and H the number of dates.

```
irf = simulate(model, dr_global, s1, n_exp=0, horizon=40 )
print(irf.__class__)
print(irf.shape)
```

```
<class 'pandas.core.frame.DataFrame'>
(40, 7)
```

Let us plot the response of consumption and investment.

```
figsize(8,4)
subplot(221)
plot(irf['z'])
title('Productivity')
subplot(222)
plot(irf['i'])
title('Investment')
subplot(223)
plot(irf['n'])
title('Labour')
subplot(224)
plot(irf['c'])
title('Consumption')

tight_layout()
```



Note that the plotting is made using the wonderful matplotlib library. Read the online tutorials to learn how to customize the plots to your needs (e.g., using latex in annotations). If instead you would like to produce charts in Matlab, you can easily export the impulse response functions, or any other matrix, to a `.mat` file.

```
irf_array = array( irf )
import scipy.io
scipy.io.savemat("export.mat", {'table': irf_array} )
```

Now Stochastic simulations

Now we run 1000 random simulations the result is an array of size $H \times n_{exp} \times n_v$ where - H the number of dates - n_{exp} the number of simulations - n_v is the number of variables

```
sim = simulate(model, dr_global, s0, n_exp=1000, horizon=40 )
print(sim.shape)
```

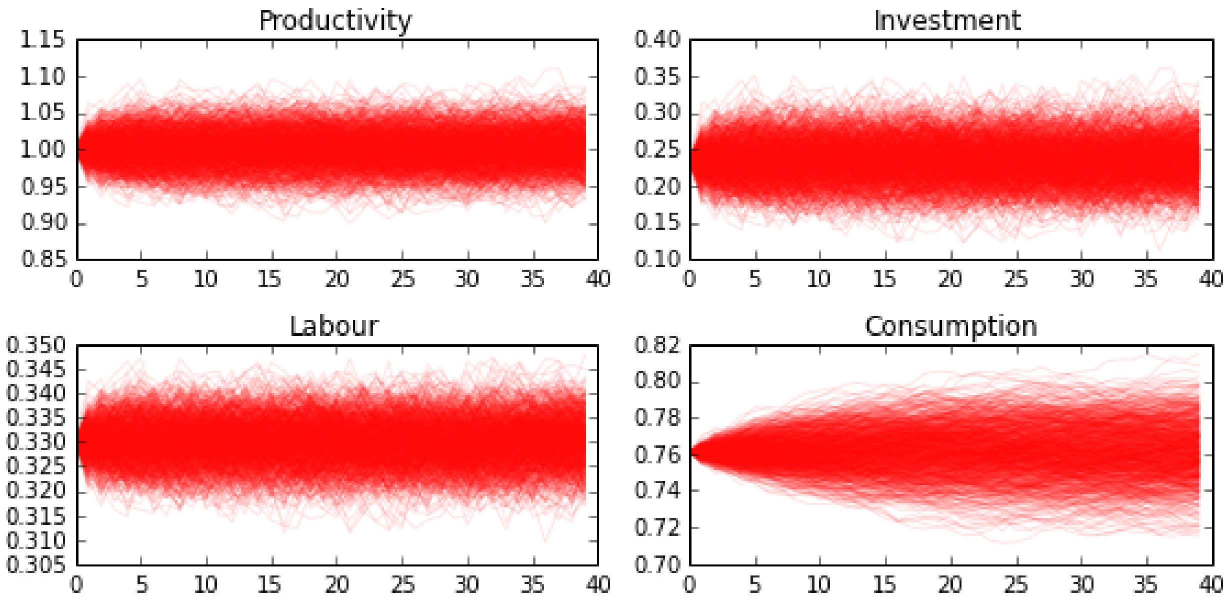
```
(40, 1000, 7)
```

```
model.variables
```

```
('z', 'k', 'i', 'n', 'c', 'rk', 'w')
```

We plot the responses of consumption, investment and labour to the stochastic path of productivity.

```
i_z = model.variables.index('z')
i_i = model.variables.index('i')
i_n = model.variables.index('n')
i_c = model.variables.index('c')
figsize(8,4)
for i in range(1000):
    subplot(221)
    plot(sim[:, i, i_z], color='red', alpha=0.1)
    subplot(222)
    plot(sim[:, i, i_i], color='red', alpha=0.1)
    subplot(223)
    plot(sim[:, i, i_n], color='red', alpha=0.1)
    subplot(224)
    plot(sim[:, i, i_c], color='red', alpha=0.1)

subplot(221)
title('Productivity')
subplot(222)
title('Investment')
subplot(223)
title('Labour')
subplot(224)
title('Consumption')

tight_layout()
```



We find that while the distribution of investment and labour converges quickly to the ergodic distribution, that of consumption takes noticeably longer. This is indicative of higher persistence in consumption, which in turn could be explained by permanent income considerations.

Descriptive statistics

The success of the RBC model is in being able to mimic patterns in the descriptive statistics of the real economy. Let us compute some of these descriptive statistics from our sample of stochastic simulations. First we compute growth rates:

```
dsim = log(sim[1:,:,:]/sim[:-1,:,:,])
print(dsim.shape)
```

```
(39, 1000, 7)
```

Then we compute the volatility of growth rates for each simulation:

```
volat = dsim.std(axis=0)
print(volat.shape)
```

```
(1000, 7)
```

Then we compute the mean and a confidence interval for each variable. In the generated table the first column contains the standard deviations of growth rates. The second and third columns contain the lower and upper bounds of the 95% confidence intervals, respectively.

```
table = column_stack([
    volat.mean(axis=0),
    volat.mean(axis=0)-1.96*volat.std(axis=0),
    volat.mean(axis=0)+1.96*volat.std(axis=0)  ])
table
```

```
array([[ 0.01667413,  0.01280193,  0.02054634],
       [ 0.00296542,  0.00175695,  0.00417388],
       [ 0.09196494,  0.06834055,  0.11558933],
       [ 0.01028367,  0.00788583,  0.01268152],
       [ 0.00313835,  0.00236476,  0.00391193],
       [ 0.02426923,  0.01861151,  0.02992694],
       [ 0.01303212,  0.01002955,  0.01603469]])
```

We can use the pandas library to present the results in a nice table.

```
model.variables
```

```
('z', 'k', 'i', 'n', 'c', 'rk', 'w')
```

```
import pandas
df = pandas.DataFrame(table, index=model.variables,
                      columns=['Growth rate std.',
                               'Lower 95% bound',
                               'Upper 95% bound' ])
pandas.set_option('precision', 4)
df
```

1.3.4 Error measures

It is always important to get a handle on the accuracy of the solution. The `omega` function computes and aggregates the errors for the model's arbitrage section equations. For the RBC model these are the investment demand and labor supply equations. For each equation it reports the maximum error over the domain and the mean error using ergodic distribution weights (see the source for the omega function).

```
ErrorErrorfrom dolo.algos.dtcscc.accuracy import omega

print("Perturbation solution")
err_pert = omega(model, dr_pert)
err_pert
```

```
Perturbation solution

Euler Errors:
- max_errors     : [ 0.00019241  0.00045583]
- ergodic        : [  1.37473238e-04   1.69920101e-05]
```

```
print("Global solution")
err_global=omega(model, dr_global)
err_global
```

```
Global solution
```

```
Euler Errors:
- max_errors     : [  1.38008607e-04   2.28991817e-06]
- ergodic        : [  1.32367122e-04   6.62075500e-07]
```

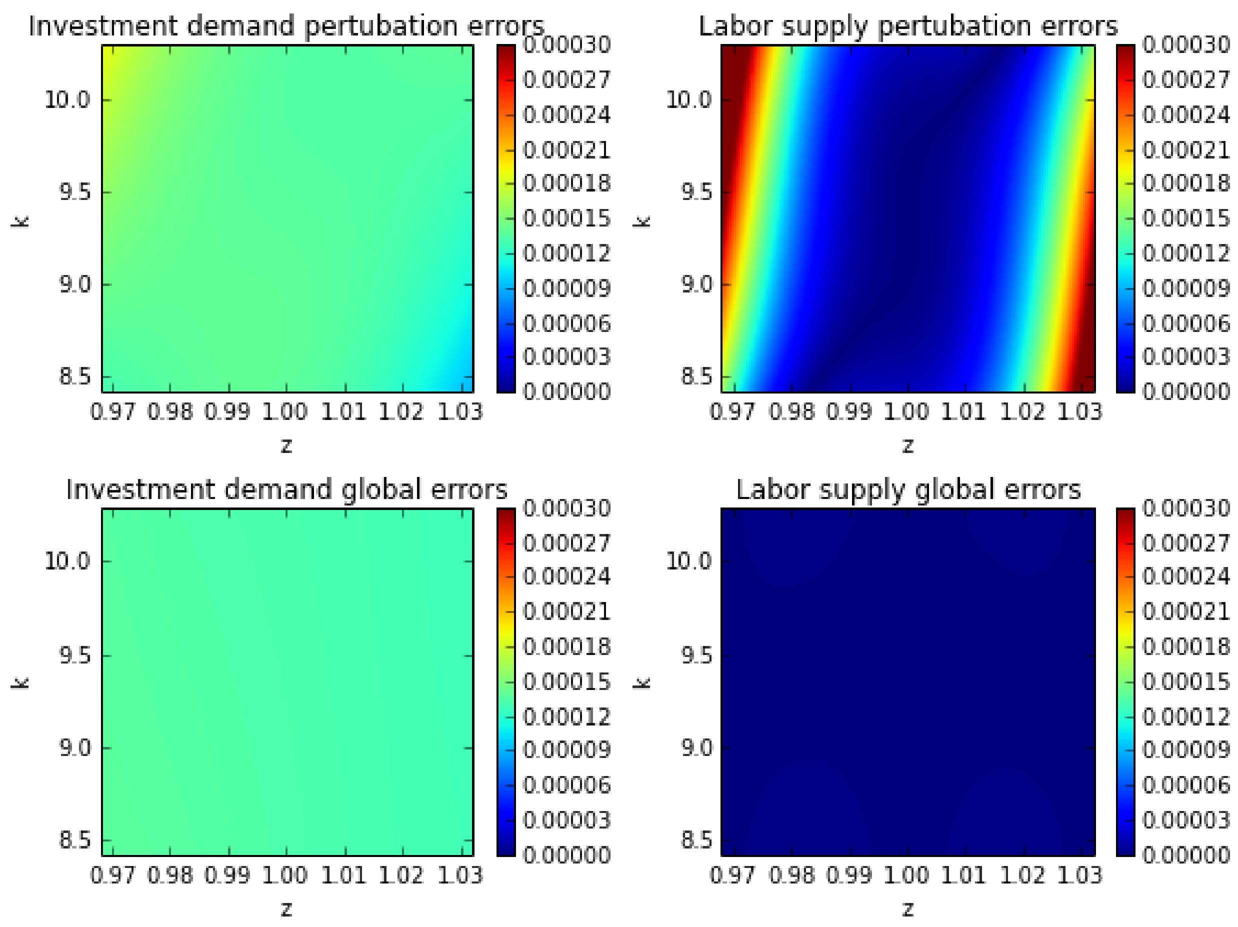
The result of `omega` is a subclass of `dict`. `omega` fills that dict with some useful information that the default print does not reveal:

```
err_pert.keys()
```

```
['domain', 'errors', 'densities', 'ergodic', 'max_errors', 'bounds']
```

In particular the domain field contains information, like bounds and shape, that we can use to plot the spatial pattern of errors.

```
a = err_pert['domain'].a
b = err_pert['domain'].b
orders = err_pert['domain'].orders
errors = concatenate((err_pert['errors'].reshape( orders.tolist()+[-1] ),
                      err_global['errors'].reshape( orders.tolist()+[-1] )),
                     2)

figure(figsize=(8,6))

titles=["Investment demand pertubation errors",
        "Labor supply pertubation errors",
        "Investment demand global errors",
        "Labor supply global errors"]

for i in range(4):

    subplot(2,2,i+1)
    imgplot = imshow(errors[:,:,i], origin='lower',
                     extent=( a[0], b[0], a[1], b[1]), aspect='auto')
    imgplot.set_clim(0,3e-4)
    colorbar()
    xlabel('z')
    ylabel('k')
    title(titles[i])

tight_layout()
```



1.4 Dolo modeling language

1.4.1 YAML format

Dolo describes *yaml* files to describe models.

1.4.2 Declaration of symbols

1.4.3 Declaration of equations

1.4.4 Calibration section

1.4.5 Shock specification

Normally distributed shocks

Markov chains mini-language

1.4.6 Options

Approximation space

1.5 Dolo Model Classification

1.5.1 Dynare models

A Dynare model is specified by a set of equations captured by one vector function f so that the vector of endogeneous states y_t satisfies

$$0 = E_t f(y_{t-1}, y_t, y_{t+1}, \epsilon_t)$$

where ϵ_t is a vector of exogenous variables following an i.i.d. series of shocks.

In this formulation, any variable appearing at date $t-1$ in the system (a predetermined variable) is part of the state-space as well as all shocks. Hence, the solution of the system is a function g such that:

$$y_t = f(y_{t-1}, \epsilon_t)$$

1.5.2 Discrete Time - Continuous States - Continuous Controls models (DTCSCC)

Solution

State-space is characterized by a vector s of continuous variables. The unknown vector of controls x is expressed by a function φ such that:

$$x = \varphi(s)$$

Transitions

```
- name: `transition`
- short name: `g`
```

Transitions are given by a function g such that at all times:

$$s_t = g(s_{t-1}, x_{t-1}, \epsilon_t)$$

where ϵ_t is a vector of i.i.d. shocks.

Value equation

```
- name: `value`
- short name: `v`
```

The (separable) value equation defines a value v_t as:

$$v_t = U(s_t, x_t) + \beta E_t \left[v_{t+1} \right]$$

In general the Bellman equations are completely characterized by the reward function U and the discount rate β.

Note that several values can be computed at once, if U is a vector function and β a vector of discount factors.

Generalized Value Equation

```
- name: `value_2`
- short name: `v_2`
```

A more general updating equation can be useful to express non-separable utilities or prices. In that case, we define a function U^* such that

$$v^*_t = U^*(s_t, x_t, v^*_t, s_{t+1}, x_{t+1}, v^*_{t+1})$$

This equation defines the vector of (generalized) values v^*

Boundaries

```
- name: `controls_lb` and `controls_ub`
- short name: `lb` and `ub`
```

The optimal controls must also satisfy bounds that are function of states. There are two functions $\underline{b}()$ and $\overline{b}()$ such that:

$$\underline{b}(s_t) \leq x_t \leq \overline{b}(s_t)$$

Euler equation

```
- name: `arbitrage` (`equilibrium`)
- short name: `f`
```

A general formulation of the Euler equation is:

$$0 = E_t[f(s_t, x_t, s_{t+1}, x_{t+1})]$$

Note that the Euler equation and the boundaries interact via "complentarity equations". Evaluated at one given state, with the vector of controls $x = (x_1, ..., x_i, ..., x_{n_x})$, the Euler equation gives us the residuals $r=(f_1, ..., f_i, ..., f_{n_x})$. Suppose that the i-th control x_i is supposed to lie in the interval $[\underline{b} * i, \overline{b} * i]$. Then one of the following conditions must be true:

- f_i = 0
- $f_i > 0$ and $x_i = \underline{b}_i$
- $f_i < 0$ and $x_i = \overline{b}_i$

By definition, this set of conditions is denoted by:

- $f_i = 0 perp underline{b}*i leq x*i leq overline{b}_i$

These notations extend to a vector setting so that the Euler equations can also be written:

$$0 = E_t[f(s_t, x_t, s_{t+1}, x_{t+1})] \perp \underline{b}(s_t) \leq x_t \leq \overline{b}(s_t)$$

Specifying the boundaries together with Euler equation, or specifying them as independent equations is (read: should be) equivalent. In any case, when the boundaries are finite and occasionally binding, some attention should be devoted to write the Euler equations in a consistent manner. In particular, note, that the Euler equations are order-sensitive.

The Euler conditions, together with the complementarity conditions typically come from the Kuhn-Tucker conditions associated with the maximization problem, but that is not true in general.

Expectations

```
- name: `expectation`
- short name: `h`
```

The vector of explicit expectations z_t is defined by a function h such that:

$$z_t = E_t \left[h(s_{t+1}, x_{t+1}) \right]$$

Generalized expectations

```
- name: `expectation_2`
- short name: `h_2`
```

The vector of generalized explicit expectations z_t is defined by a function $h^\star$ such that:

$$z_t = E_t \left[h^\star(s_t, x_t, \epsilon_{t+1}, s_{t+1}, x_{t+1}) \right]$$

Euler equation with explicit equations

```
- name: `arbitrage_2` (`equilibrium_2`)
- short name: `f_2`
```

If expectations are defined using one of the two preceding definitions, the Euler equation can be rewritten as:

$$0 = f(s_t, x_t, z_t) \perp \underline{b}(s_t) \leq x_t \leq \overline{b}(s_t)$$

Direct response function

```
- name: `direct_response`
- short name: `d`
```

In some simple cases, there a function $d()$ giving an explicit definition of the controls:

$$x_t = d(s_t, z_t)$$

Compared to the preceding Euler equation, this formulation saves computational time by removing to solve a nonlinear to get the controls implicitly defined by the Euler equation.

Terminal conditions

```
- name: `terminal_control`
- short name: `f_T`
```

When solving a model over a finite number T of periods, there must be a terminal condition defining the controls for the last period. This is a function f^T such that:

$$0 = f^T(s_T, x_T)$$

Terminal conditions

```
- name: `terminal_control_2`
- short name: `f_T_2`
```

When solving a model over a finite number T of periods, there must be a terminal condition defining the controls for the last period. This is a function $f^{T,\star}$ such that:

$$x_T = f^{T,\star}(s_T)$$

Auxiliary variables

```
- name: `auxiliary`
- short name: `a`
```

In order to reduce the number of variables, it is useful to define auxiliary variables y_t using a function a such that:

$$y_t = a(s_t, x_t)$$

When they appear in an equation they are automatically substituted by the corresponding expression in s_t and x_t.

1.5.3 Discrete Time - Mixed States - Continuous Controls models (DTMSCC)

The definitions for this class of models differ from the former ones by the fact that states are split into exogenous and discrete markov states, and endogenous continous states as before. Most of the definition can be readily transposed by replacing only the state variables.

State-space and solution

For this kind of problem, the state-space, is the cartesian product of a vector of “markov states” m_t that can take a finite number of values and a vector of “continuous states” s_t which takes continuous values.

The unknown controls x_t is a function φ such that:

$$x_t = \varphi(m_t, s_t)$$

Transitions

```
- name: `transition`
- short name: `g`
```

(m_t) follows an exogenous and discrete markov chain. The whole markov chain is specified by two matrices P, Q where each line of P is one admissible value for m_t and where each element $Q(i,j)$ is the conditional probability to go from state i to state j.

The continuous states s_t evolve after the law of motion:

$$s_t = g(s_{t-1}, x_{t-1}, \epsilon_t) : math : `where`\epsilon_t$$

where ϵ_t is a vector of i.i.d. shocks.

Boundaries

```
- name: `controls_lb`, `controls_ub`
- short name: `lb`, `ub`
```

The optimal controls must satisfy bounds that are function of states. There are two functions $\underline{b}()$ and $\overline{b}()$ such that:

$$\underline{b}(m_t, s_t) \leq x_t \leq \overline{b}(m_t, s_t)$$

Value Equation

```
- name: `value`
- short name: `v`
```

The (separable) Bellman equation defines a value v_t as:

$$v_t = U(m_t, s_t, x_t) \| \beta E_t \left[v_{t+1} \right]$$

It is completely characterized by the reward function U and the discount rate β.

Generalized Value Equation

```
- name: `value_2`
- short name: `v_2`
```

The generalized value equation defines a value $v^{\star}_t$ as:

$$: math : `v^{\star}_t = U^{\star}(m_t, s_t, x_t, v^{\star}, m_{t+1}, s_{t+1}, x_{t+1})`$$

Euler equation

```
- name: `arbitrage` (`equilibrium`)
- short name: `f`
```

Many Euler equations can be defined a function f such that:

$$0 = E_t \left(f(m_t, s_t, x_t, m_{t+1}, s_{t+1}, x_{t+1}) \right) \perp \underline{b}(m_t, s_t) \leq x_t \leq \overline{b}(m_t, s_t)$$

See discussion about complementarity equations in the Continuous States - Continuous Controls section.

Expectations

```
- name: `expectation`
- short name: `h`
```

The vector of explicit expectations z_t is defined by a function h such that:

$$z_t = E_t \left[h(m_{t+1}, s_{t+1}, x_{t1}) \right]$$

Generalized expectations

```
- name: `expectation_2`
- short name: `h_2`
```

The vector of generalized explicit expectations z_t is defined by a function $h^\star$ such that:

$$z_t = E_t \left[h^\star(m_t, s_t, x_t, m_{t+1}, s_{t+1}, x_{t+1}) \right]$$

Euler equation with explicit equations

```
- name: `arbitrage_2` (`equilibrium_2`)
- short name: `f_2`
```

If expectations are defined using one of the two preceding definitions, the Euler equation can be rewritten as:

$$0 = f(m_t, s_t, x_t, z_t) \perp \underline{b}(s_t) \leq x_t \leq \overline{b}(s_t)$$

Direct response function

```
- name: `direct_response`
- short name: `d`
```

In some simple cases, there a function $d()$ giving an explicit definition of the controls:

$$x_t = d(s_t, z_t)$$

Compared to the preceding Euler equation, this formulation saves computational time by removing to solve a nonlinear to get the controls implicitly defined by the Euler equation.

Direct states function

```
- name: `direct_states`
- short name: `d_s`
```

For some applications, it is also useful to have a function $d\star$ which gives the endogenous states as a function of the controls and the exogenous markov states:

$$s_t = d^\star(m_t, x_t)$$

Auxiliary variables

```
- name: `auxiliary`
- short name: `a`
```

In order to reduce the number of variables, it is useful to define auxiliary variables y_t using a function a such that:

$$y_t = a(m_t, s_t, x_t)$$

Terminal conditions

```
- name: 'terminal_control'
- short name: 'f_T'
```

When solving a model over a finite number T of periods, there must be a terminal condition defining the controls for the last period. This is a function f^T such that:

$$x_T = f^T(m_T, s_T)$$

Terminal conditions (explicit)

```
- name: 'terminal_control'
- short name: 'f_T_2'
```

When solving a model over a finite number T of periods, there must be a terminal condition defining the controls for the last period. This is a function $f^{T,\star}$ such that:

$$f^{T,\star}(m_T, s_T, x_T)$$

1.5.4 Misc

Variables

For DTCSCC and DTMSCC models, the following list variable types can be used (abbreviation in parenthesis): Required:

- `states` (`s`)
- `controls` (`x`) For DTCSCC only:
- `shocks` (`e`) For DTMSCC only:
- `markov_states` (`m`) Optional:
- `auxiliaries` (`y`)
- `values` (`v`)
- `values_2` (`v_2`)
- `expectations` (`z`)
- `expectations_2` (`z_2`)

Algorithms

Several algorithm are available to solve a model, depending no the functions that are specified.

	Dynare model	DTCSCC	DTMSCC
Perturbations	yes	(f,g)	no
Perturbations (higher order)	yes	(f,g)	no
Value function iteration		(v,g)	(v,g)
Time iteration		(f,g),(f,g,h)	(f,g),(f,g,h)
Parameterized expectations		(f,g,h)	(f,g,h)
Parameterized expectations (2)		(f_2,g,h_2)	(f_2,g,h_2)
Parameterized expectations (3)		(d,g,h)	(d,g,h)
Endogeneous gridpoints			(d,d_s,g,h)

1.5.5 Additional informations

calibration

In general, the models will depend on a series of scalar parameters. A reference value for the endogeneous variables is also used, for instance to define the steady-state. We call a "calibration" a list of values for all parameters and steady-state.

state-space

When a global solution is computed, continuous states need to be bounded. This can be done by specifying an n-dimensional box for them.

Usually one also want to specify a finite grid, included in this grid and the interpolation method used to evaluate between the grid points.

specification of the shocks

For DTCSCC models, the shocks follow an i.i.d. series of random variables. If the shock is normal, this one is characterized by a covariance matrix.

For DTMSCC models, exogenous shocks are specified by a two matrices P and Q, containing respectively a list of nodes and the transition probabilities.

Remarks

Some autodetection is possible. For instance, some equations appearing in `f` fonctions, can be promoted (or downgraded) to expectational equation, based on incidence analysis.

1.6 Solution algorithms

1.6.1 Algorithms for DTCSCC models

Steady-state

Perturbation

Perfect foresight

We consider an *fg* model, that is a model with in the form:

$$s_t = g\left(s_{t-1}, x_{t-1}, \epsilon_t\right)$$
$$0 = E_t\left(f\left(s_t, x_t, s_{t+1}, x_{t+1}\right)\right) \perp \underline{u} <= x_t <= \overline{u}$$

We assume that $\underline{u}$ and $\overline{u}$ are constants. This is not a big restriction since the model can always be reformulated in order to meet that constraint, by adding more equations.

Given a realization of the shocks $(\epsilon_i)_{i>=1}$ and an initial state s_0, the perfect foresight problem consists in finding the path of optimal controls $(x_t)_{t>=0}$ and the corresponding evolution of states $(s_t)_{t>=0}$.

In practice, we find a solution over a finite horizon $T > 0$ by assuming that the last state is constant forever. The stacked system of equations satisfied by the solution is:

$s_0 = s_0$	$f(s_0, x_0, s_1, x_1) \perp \underline{u} <= x_0 <= \overline{u}$
$s_1 = g(s_0, x_0, \epsilon_1)$	$f(s_1, x_1, s_2, x_2) \perp \underline{u} <= x_1 <= \overline{u}$
$s_T = g(s_{T-1}, x_{T-1}, \epsilon_T)$	$f(s_T, x_T, s_T, x_T) \perp \underline{u} <= x_T <= \overline{u}$

autofunction:: dolo.algos.dtcscc.perfect_foresight.find_steady_state

Time iteration

1.6.2 Algorithms for DTMSCC models

Time-iteration

1.6.3 Algorithms for Dynare models

Perturbation

1.7 Examples

How to get a global solution of the RBC model.

How to solve a the RBC model using dynare's statefree approach

How to compute the response to a tax under perfect foresight

Redo figures 11.9.1, 11.9.2, 11.9.3 from Ljunquvist and Sargent: RMT4 (perfect foresight exercise contributed by Spencer Lyon).

1.7.1 Other languages:

Solve the RBC model using Julia.

1.8 Frequently Asked Questions

1.9 Advanced topics

1.9.1 Custom model types

Recipes

Different kinds of solution algorithm are associated with several ways of writing the model. Dolo provides a simple way to define your own type of models. These models can then be translated into numerical models.

A custom definition of a model type is called a recipe. Recipes can be described in a YAML file or directly as a Python object. Let's inspect the file which define the `fga` type is used by the default time-iteration algorithm.

- First the `model_spec` key defines a short name for the recipe.
- Second, a list of symbol types is declared. Two types of symbols are automatically

added to that list : `shocks` and `parameters`.

- Third, all blocks of equation are described in the equation_type part by supplying a list of allowed symbol types, with the date at which they occur.

  Currently, two types of blocks are recognized by dolo:

  – regular blocks (like `arbitrage`) consist of a list of expressions. When an equal sign is supplied in the model file, the right hand side is substracted from the left hand side.

  – definition blocks (with `definition: True` as in `transition` and `auxiliary`) define the left hand side as a function of the right hand side. The left hand side must be a series of variable at date 0.

  See also:

  When an actual model is checked, dolo will check that these variables are defined in the same order as in the declaration header of the model file. It also allows for recursive definitions as long as the block of equations forms a triangular system of the left hand side variables.

Checking a model's validity

A recipe can be used inside Python to check the validity of a symbolic model. For instance the following code checks that the `rbc.yaml` file is valid.

```
model = yaml_import('rbc.yaml')
from dolo.symbolic.recipes import recipe_fga
from dolo.symbolic.validator import validate

validate( model, recipe_fga )
```

If the model is valid according the its definition, the last function will not do anything. If not an error is thrown.

Producing matlab/julia code using a custom recipe

Assuming a model and a recipe are respectively defined in files `model.yaml` and `recipe.yaml`, one can generate a compiled model for matlab `model.m` using the following command:

```
dolo-matlab model.yaml --recipe=recipe.yaml
```

The same can be down for Julia using:

```
dolo-matlab model.yaml --recipe=recipe.yaml
```

The resulting file will contain nested structures (or nested dictionaries depending on the target language) with the following content:

```
- symbols

    - val_1          ( cell array with names of symbols of type 1 )
    - val_2
    - parameters     ( names of parameters )
    - shocks         ( names of shocks )

- calibration

    - steady_state
            - val_1      ( steady_state values for symbols of type 1 )
            - val_2
    - parameters         ( numeric values for parameters )
    - sigma             ( covariance matrix )

- functions

    - fun1    ( function of type fun1 )
    - fun2
```

The generated function handlers (here `fun1` and `fun2`) have the following signature : `fun(arg1, arg2, ..., p)` where `arg_i` is the i-th argument defined in the recipe as `type_i`, and is expected to be a vertical array of size `N x n_i` where `n_i` is the number of variables of type `type_i` and `N` is the number of points at which the function must be evaluated. `p` is a one dimensional vector of parameter values.

See also:

For row-major languages, especially for Python, arrays are expected to be of the size `n_i x N` so that the function can still operate on contiguous chunks of memory when `N` is big.

1.9.2 Model object

Model object

Symbolic Model

1.9.3 Interpolation

There are many available methods to interpolate points when solving global models. *dolo* provides several of them. They can been used with exactly the same interface. By default, *dolo* uses smolyak interpolation.

smolyak

linear (delaunay tessellation)

multilinear

splines

1.9.4 Compilation

Compile a generic symbolic expression

1.9.5 Model conventions

This page describes various conventions used in dolo.

Model types

There are several ways to represent a DSGE model. Here we list some of them:

state-free approach

This approach is taken in Dynare and in most perturbation softwares. The model is specified by a vectorial function

$f(y_{t+1}, y_t, y_{t-1}, \epsilon_t) = 0$

with the restriction that ϵ_t and y_{t+1} cannot appear in the same equations. For this kind of models, the solver finds a solution g such that the law of motion of (y_t) is given by: $y_t = g\left(y_{t-1}, \epsilon_t\right)$

controlled process variants

We define several categories of models.

$$\begin{tikzcd}
\text{fgah} \arrow[d] & & \\
\text{fga} \arrow[dr] & & \text{fgh} \arrow[dl] \\
& \text{fg} &
\end{tikzcd}$$

***fg* model** With these versions, the state-space is chosen by the user. A law of motion for the state-space must be specified (depending on the controls and on the shocks). And optimality conditions must be given to pin down all the controls. The model is specified by giving g and f such that:

$s_t = g\left(s_{t-1}, x_{t-1}, \epsilon_t\right)$

$E_t\left[f\left(s_t, x_t, s_{t+1}, x_{t+1}\right)\right] = 0$

The solution is a function φ such that $x_t = \varphi(s_t)$.

***fga* model** In some cases, some variables can be directly expressed as a function of other variables. We call them *auxiliary* variables. Auxiliary variables are restricted to depend only on contemporaneous variables (controls or states). The model can be rewritten:

$a_t = a\left(s_t, x_t\right)$

$s_t = g\left(s_{t-1}, x_{t-1}, a_{t-1}, \epsilon_t\right)$

$E_t\left[f\left(s_t, x_t, a_t, s_{t+1}, x_{t+1}, a_{t+1}\right)\right]$

Clearly, by substituting the variables a everywhere, this type of model can be seen as an *fg* model. Hence when some algorithm is applicable to an *fg* model, it can be also be applied to an *fga* model.

not implemented ; fgh model A sub-variant of this specification let the user choose equations to define expectations. This is useful for PEA approaches. The model is specified by giving g, f and h such that:

$s_t = g\left(s_{t-1}, x_{t-1}, \epsilon_t\right)$

$f\left(s_t, x_t, z_t\right)$

$z_t = E_t h\left(s_{t+1}, x_{t+1}\right)$

Same remark as for *fga* model: is needed, *fgh* models can behave exactly as an *fg* model.

not implemented ; fgah model A sub-variant of this specification let the user choose equations to define expectations. This is useful for PEA approaches. The model is specified by giving g, f and h such that:

$a_t = a\left(s_t, x_t\right)$

$s_t = g\left(s_{t-1}, x_{t-1}, a_{t-1}, \epsilon_t\right)$

$f\left(s_t, x_t, a_t, z_t\right)$

$z_t = E_t h\left(s_{t+1}, x_{t+1}, a_{t+1}\right)$

fgah models can behave exactly as an *fg* model, or an *fga* model

Numerical Conventions

- A list of *N* points in a *d*-dimensional space is represented by a *d x N* matrix. In other words the last index always corresponds to the successive observations. This is consistent with default ordering in numpy: last index varies faster.

Note: When dolo is used to compile a model for matlab the convention is reversed: first dimension denotes the successive points. This is adapted to matlab's storage order (Fortran order).

- In order to evaluate many times a function $f : R^s \to R^x$ for a list of *N* points we consider the vectorized version:

$f : R^s x R^N \to R^x x R^N$ **. If *X* is the matrix of inputs and *F* the matrix of output, then for any i** *F[i,:]* is the image of *X[i,:]* by *f*.

- In particular, the implementation of the model definition follow this convention, as well as the solution of the model.