

NelwinDecomp for Maple V.4: Introduction

24 Sep 1996 — rev. 5 Oct 1999
Esben Sloth Andersen, esa@business.auc.dk

WARNING 1: The program do not work with Maple V!
WARNING 2: You have to press <enter> on all procedures
to allow the system to function properly.
REMARK: In some cases you get an error message. If the
text following the error message includes "oops!" or
similar expressions, then the error message is the result
of a deliberately wrong Maple statement. See the text
about how to react.

[-] The problem of decomposition in Maple V.4

[-] Introduction

The basic heuristic in problem solving is "divide and conquer". Unless we divide a complex simulation model into simple and relatively independent components, we are likely to run into serious trouble. Maple's primary means of decomposition are procedures (also called functions). In the following we see how a Nelson and Winter model can be defined by means of a fairly large number of procedures (in contrast to SimpleNelwin where everything was included in a single procedure). The procedure that (directly and indirectly) reintegrates all the components is called StructuredNelwin.

One problem with this decomposition is that at the beginning of a Maple procedure, we have to declare which global variables we want to change. All other variables that are changed within the procedure are assumed to be local. This means that Maple makes a distinction between items that can only be accessed from within a given procedure (local or private items) and items that can be accessed without restrictions (global or public items). Global items survive during a whole Maple session while local items disappear when control is returned from the procedure in which they existed.

In the present worksheet we shall make a compromise between decomposition of programs and easy access to simulation data by placing most variables in a single table.

We start by setting the overall system in an adequate manner:

```
> restart;  
interface(prettyprint=1);  
printlevel := 3;  
printlevel := 3
```

[-] A (huge) table as a single global variable

In an evolutionary model like Nelwin and extensions thereof there is a very large number of variables, constants, etc. that we want to access and/or modify from different proceduress. We also want to have access to these items for data analysis and for saving to the harddisk (so that we can later inspect the data without having to do time consuming reruns of the model).

To simplify these uses of global items we shall try out the following solution: all global items are stored in a (huge) table declared as global. Maple tables can be considered as a generalisation of the array structure. It holds an indexed collection of elements. When we create an item, we give its indexes and its value. Later the element can be retrieved by referring to its indexes. Many types of indexes can be used.

The table is controlled by a table monitor (# 1) that stores the data in the table and administers access to it.

```
> `index/monitor1` := proc(indices,table)  
if not (type(op(1,indices),string) or  
type(op(1,indices),indexed)) then ERROR(`wrong first  
part of the index`);  
fi;  
if (nargs = 2) then
```

```

    if (assigned(table[op(indices)])) then
table[op(indices)];
    else ERROR(`not assigned`);
    fi;
else table[op(indices)] := op(args[3..nargs]);
fi;
end:

```

Since the names of most of the variables and constants have to be enhanced with the name of the global table, we want this name to be as invisible as possible and not to be confused with any other names. We shall apply the name "_" (underscore).

We now define a procedure for initialising the table with the name "_" and the index function "monitor". Then we call the initialisation procedure.

```

> CreateTable := proc()
  global _;
  _ := table(monitor1);
end:
> CreateTable();
table(monitor1, [
    ])

```

The functioning of the global table

We now see how we can store and retrieve information. The first part of the index has to be a name (letter or mix of letters and integers). Only information that has been stored can be retrieved.

```

> _[1,1,1] := 89.7;
Error, (in index/monitor1) wrong first part of the index
executing statement: ERROR(`wrong first part of the index`)
index/monitor1 called with arguments: [1, 1, 1], table, [89.7]

```

Oops! The problem here was that the first part of the index was not a name. Let us make it correctly:

```

> _[K,1,1] := 89.7;
                                _[K, 1, 1] := 89.7

```

That was OK. But let us now try to make the error indirectly:

```

> K := 1;
                                K := 1
> _[K,1,1];
Error, (in index/monitor1) wrong first part of the index
executing statement: ERROR(`wrong first part of the index`)
index/monitor1 called with arguments: [1, 1, 1], table

```

Still the problem is that the first part of the index received by the Maple engine is not a name. Let us, therefore, restore the name "K":

```

> K := 'K';
                                K := K
> _[K,1,1] := 89.7;
                                _[K, 1, 1] := 89.7
> _[K,1,1];
                                89.7
> _[K,2,1];
Error, (in index/monitor1) not assigned
executing statement: ERROR(`not assigned`)
index/monitor1 called with arguments: [K, 2, 1], table
> _[K,2,1] := 89.7;
                                _[K, 2, 1] := 89.7
> _[K,2,1];
                                89.7
> _[K,1,2] := 93.8;
                                _[K, 1, 2] := 93.8

```

```

> for i from 1 to 2 do
  print(_[K,1,i]);
od;
                                89.7
                                93.8

```

[Now we can write a procedure that initialises the variables K and A for N firms:

```
[ > Initialise := proc(N)
  global _;
  local i;
  for i from 1 to N do
    _[A,i,1] := 0.16;
    _[K,i,1] := 89.70;
  od;
  lprint(`Initialisation OK`);
end:
[ > Initialise(4);
Initialisation OK
[ > print(_);
table(monitor1, [
    (K, 4, 1) = 89.70
    (A, 2, 1) = .16
    (K, 1, 2) = 93.8
    (K, 3, 1) = 89.70
    (A, 3, 1) = .16
    (K, 2, 1) = 89.70
    (A, 4, 1) = .16
    (K, 1, 1) = 89.70
    (A, 1, 1) = .16
  ])
```

[Remark that one value has survived from our earlier definition ((K, 1, 2) = 93.8). If we want to avoid such surprises, we have to clear the table (by a redefinition: _ := table(monitor);)

A little on random numbers

[Let us look at the results of this procedure as an example of random number generation. To secure that the result can be reproduced, we start by setting the "_seed", i.e. the system variable governing the outcome of the (pseudo) random number generation. The _seed is normally set to the last random number that has been generated by Maple's internal system. But the user can define an initial value for _seed. This initial value generates at deterministic sequence of numbers that behave as if they were produced randomly. This means that although the sequence can be taken to represent the behaviour of a random variable, the sequence can also be reproduced by giving the original _seed to Maple's engine. In other words, since Maple's generation of random numbers is strictly deterministic, each run with the same initial _seed value will give exactly the same result, ceteris paribus (i.e. with no parameter change, etc.).

```
[ > rand();
                                         427419669081
[ > rand();
                                         321110693270
[ > rand();
                                         343633073697
[ > _seed := 2:
[ > rand();
                                         854839338162
[ > _seed := 2:
[ > rand();
                                         854839338162
```

```
[ > _seed := 27:
[ > rand();
540331065308
[ > _seed := 27:
[ > rand();
540331065308
```

Apart from the rand function, Maple's functions for producing random numbers are placed in the random package within the stats package. This means that you can either call these functions by a long name or - after calling the package - by a short name. Let us consider Maple's function for generating a sequence of random numbers distributed according to a normal distribution with defined mean ($\mu=2$) and standard deviation ($\sigma=.5$). We start by the long name. Then we load the random package and make a few tries with the short name.

```
[ > stats[random,normald[2,0.5]]();
2.440536751
[ > with(stats[random]);
[beta, binomiald, cauchy, chisquare, discreteuniform, empirical, exponential, fra
laplaced, logistic, lognormal, negativebinomial, normald, poisson, studentst,
weibull]
[ > normald[2,0.5]();
2.155756361
[ > _seed := 27:
[ > normald[2,0.5]();
1.513856966
```

```
[ > _seed := 27:
[ > normald[2,0.5]();
1.513856966
```

Finally we ask for five random numbers determined by the same _seed. Remark that the first is the same as before, while the others are (pseudo) randomly distributed.

```
[ > _seed := 27:
[ > normald[2,0.5](5);
1.513856966, 1.732695730, 1.247211154, 2.270571276, 2.961489731
```

We also need random numbers distributed according to the Poisson distribution (with the parameter lambda, here $\lambda=3$).

```
[ > poisson[3](5);
5.0, 3.0, 2.0, 1.0, 0
```

Finally we try in vain to obtain a series of uniformly distributed random numbers from 1 to 6 (like throwing a dice).

```
[ > uniform[1..6](5);
Error, (in stats/parms/functions/uniform) invalid types in sum
executing statement: b := a+1
locals defined as: a = 1. .. 6., b = b, status = status, x = x
stats/parms/functions/uniform called with arguments: 5, [1 .. 6], false
stats/functions/random called with arguments: [uniform[1 .. 6]], [5],
uniform[1 .. 6]
stats[random,uniform[1 .. 6]] called with arguments: 5, topname(uniform[1
.. 6])
uniform[1 .. 6] called with arguments: 5
```

Oops! Here we encountered a grave error in the Maple system. Luckily most of Maple's system is accessible to users, so we can correct the error immediately (do not think about the details!).

Correcting a bug in Maple's random package

Unfortunately there is a bug in the Maple function for creating a uniform distribution. Therefore, we have to include a corrected version of a helping procedure relating to the uniform function. The new version of the procedure (``stats/parms/functions/uniform``) secures that the uniform distribution in Maple's random function package works correctly. Just press <enter>. Do not bother about the details.

```
[ > `stats/parms/functions/uniform` := proc(xvalue, given,
xcheck)
```

```

local a, b, status, x;
option system, remember,
`Copyright (c) 1993 by Waterloo Maple Inc.
Bug corrected by ESA, 1 Sep 1996. Another bug concerning
one real argument is still present. In the one-argument
case, only integers can be used`;

if nops(given) = 0 then a := evalf(0); b := evalf(1)
elif nops(given[]) = 2 then
  a := evalf(op(1, given[])); b := evalf(op(2, given[]))
elif nops(given) = 1 then a := evalf(op(1, given[])); b :=
a + 1
else
  status := 'abort';
  RETURN(status,
    [requires two parameters (no parameters defaults to
0,1), received`, given])
fi;
if not type([a, b], list(numeric)) then
  status := 'FAIL';
  RETURN(status, [unable to find evalf() of the bounds
`, a, b])
fi;
if not (a < b) then
  status := 'abort';
  RETURN(status, [right bound must be greater than left
bound, received`, a, b])
fi;
if not xcheck then x := xvalue
else
  x := evalf(xvalue);
  if not type(x, numeric) then
    status := 'FAIL';
    RETURN(status, [unable to find evalf() of
argument,
received`, xvalue])
  fi
fi;
status := 1;
status, [x, a, b]
end:

```

[Now we try again with our dice:

```
[ > uniform[1..6](5);
2.860190768, 3.619231213, 1.287895212, 2.938127176, 5.667342603
```

[-] The elements of NelwinDecomp

[-] Introduction

[In the following formulation of functions and procedures, the classic Nelson and Winter model family has been formulated according to the paradigm of decomposed programming that combines elements of structured programming and object-oriented programming.

[The core of the program (or system of functions and procedures) is the transformation of the state variables of a firm - its physical capital (K) and its (reciprocal) productivity (

$$A = \frac{Q}{K}.$$

[The mechanism of transformation from one state to the next includes both elements within

and beyond the individual firm, and the state transformation includes both deterministic and probabilistic elements. Let us start to explore these elements.

Creating a global table and an individual firm

We start by making a slight change in the functions for monitoring a table and for creating the global table. Then we call CreateTable function:

```
> `index/monitor2` := proc(indices,table)
  if not (type(op(1,indices),string) or
  type(op(1,indices),indexed)) then ERROR(`wrong first part
  of the index`); fi;
  if (nargs = 2) then table[op(indices)];
  else table[op(indices)] := op(args[3..nargs]);
  fi;
  end:
> CreateTable := proc()
  global _;
  _ := table(monitor2);
  _[newFirm] := 0;
  RETURN(`Table OK`);
  end:
> CreateTable();
                                     Table OK
> print(_);
table(monitor2, [
  newFirm = 0
])
```

Then we define a procedure for creating firms and make our first firm:

```
> CreateFirm := proc(capital,productivity)
  global _;
  local firmKey;
  firmKey := _[newFirm] + 1;
  _[newFirm] := firmKey;
  _[K,firmKey,1] := capital;
  _[A,firmKey,1] := productivity;
  RETURN(`Firm created. FirmKey`, firmKey);
  end:
> CreateFirm(91.33,.16);
                                     Firm created. FirmKey, 1
> print(_);
table(monitor2, [
  (K, 1, 1) = 91.33
  (A, 1, 1) = .16
  newFirm = 1
])
```

We now make another firm to allow for Schumpeterian competition:

```
> CreateFirm(91.33,.16);
                                     Firm created. FirmKey, 2
```

Output decision

The firm is now ready for making its first decision. This concerns output. The decision-making rule is simply that the firm i produces as much as possible in any given period (t). The output is $Q_{i,t}$ or, in relation to the global table, $_{[Q,i,t]}$.

```
> FirmOutput := proc(i,t)
  global _;
  _[Q,i,t] := _[A,i,t]*_[K,i,t];
```

```

end:
> FirmOutput(1,1);
14.6128
> FirmOutput(2,1);
14.6128

```

Industry-level supply and market price

Let us now consider what happens in the short run at the market level. In the Nelson and Winter model this is quite simple (because our analytical interest is concentrated on other issues). First we find total supply. Then we find the market price in a market with unit elasticity of demand:

```

> Supply := proc(t)
  global _;
  local i;
  _[TQ,t] := add(_[Q,i,t], i=1.. _[newFirm]);
end:
> Supply(1);
29.2256
> Price := proc(t)
  global _;
  _[P,t] := _[Revenue]/_[TQ,t];
end:

```

Before we can find the price, we need to define the parameter "Revenue":

```

> _[Revenue] := 67;
_[Revenue] := 67
> Price(1);
2.292510676

```

Firm-level profits

We can now calculate the firm's profits per unit of capital.

```

> Profit := proc(i,t)
  global _;
  local costs, revenue;
  costs := _[c] + _[r_in] + _[r_im];
  revenue := _[P,t]*_[A,i,t];
  _[pi,i,t] := revenue - costs;
end:

```

This function depends on three parameters. We give them values taken from Nelson and Winter - but not adapted to the two-firm case. This means that we should not expect the model to be in an initial equilibrium.

```

> _[c] := .16;
_[r_in] := 0.00112;
_[r_im] := 0.0223;
_[c] := .16
_[r_in] := .00112
_[r_im] := .0223
> Profit(1,1); Profit(2,1);
.1833817082
.1833817082

```

We see that each firm makes a profit in the initial period. However, let us make a couple of new firms of the same size as the two previous ones, and consider the consequence for profits:

```

> CreateFirm(91.33,.16); CreateFirm(91.33,.16);
Firm created. FirmKey, 3
Firm created. FirmKey, 4
> FirmOutput(3,1); FirmOutput(4,1);
14.6128
14.6128
> Supply(1);
58.4512
> Price(1);
1.146255338

```

```
> for i from 1 to 4 do Profit(i,1); od;
-.0000191459
-.0000191459
-.0000191459
-.0000191459
```

With the set-up of new firms, the capacity of the industry fits the fixed level of revenue in a way that profits become practically equal to zero.

Search for and choice of technology

The simulation of innovation and imitation depends on Maple's random number generators (see the section on random numbers).

A firm can obtain a new level of productivity (a new A) either in a direct (innovative) or in an indirect (imitative) way. The chances of obtaining success depends on the firm's level of innovative R&D and imitative R&D.

```
> Innovation := proc(i,t)
  global _;
  local innoResearch;
  innoResearch := _[r_in]*_[K,i,t];
  _[A_in,i,t] := InnoResult(i,t,innoResearch);
end;
```

The Innovation procedure depends on a procedure (InnoResult) that we have not defined yet. Furthermore, we have not specified the size of r_n . Therefore, it is still too early to call the procedure. Or, rather, we obtain a symbolic result instead of a numeric result:

```
> Innovation(1,1);
InnoResult(1, 1, .1022896)
```

Before we define the missing procedure and parameter, we define how a firm makes an Imitation:

```
> Imitation := proc(i,t)
  global _;
  local imiResearch;
  imiResearch := _[r_im]*_[K,i,t];
  _[A_im,i] := ImiResult(i,t,imiResearch);
end;
```

Whether the results of the firm's innovative and imitative search are successful or unsuccessful, they can be used in determining the technology applied in the next period.

```
> TechnoChoice := proc(i,t)
  global _;
  _[A,i,t+1] := max(_[A,i,t], _[A_in,i,t], _[A_im,i,t]);
end;
```

"Techno space" for innovation and industry-level imitation

To determine innovations and imitations we need to be sure that the random package is loaded. Furthermore, we define the `_seed`:

```
> with(stats[random]): _seed := 1;
_seed := 1
```

The space in which firms search for innovations can be constructed in different ways. Major alternatives depict science-based search and cumulative-technology search. Let us put both alternatives into the definition of the procedure "InnoResult".

```
> InnoResult := proc(i,t,effort)
  global _;
  local lambda, draws, mean,lnResult;
  lambda := _[d_in]*effort;
  draws := poisson[lambda]();
  if draws > 0 then
    if _[searchType] = cumulative then
      mean := ln(_[A,i,t]);
      lnResult := normald[mean,_[sigma_in]]();
      _[A_in,i,t] := exp(lnResult);
    else
```

```

        mean := ln(_[A_mean,t]);
        lnResult := normald[mean,_[sigma_in]]();
        _[A_in,i,t] := exp(lnResult);
    fi;
else _[A_in,i,t] := 0;
fi;
end:

```

[The imitation procedure is simpler:

```

> ImiResult := proc(i,t,effort)
global _;
local lambda, draws;
lambda := _[d_im]*effort;
draws := poisson[lambda]();
if draws > 0 then
    _[A_im,i,t] := _[A_max,t];
else _[A_im,i,t] := 0;
fi;
end:

```

[We need to specify some parameters before these procedures can be called by the firms:

```

> _[searchType] := cumulative;
                                _[searchType] := cumulative
> _[d_in] := 0.3;
                                _[d_in] := .3
> _[d_im] := 0.3;
                                _[d_im] := .3
> _[sigma_in] := 0.2;
                                _[sigma_in] := .2

```

[We also need to find the firm with the leading technology:

```

> Maximum := proc(t)
global _;
_[A_max,t] := max(seq(_[A,i,t], i=1.._[newFirm]));
end:
> Maximum(1);
                                .16

```

Firms' innovation and imitation

[The firms can now try the innovative and imitative lotteries. We start with firm #1 in period 1:

```

> Innovation(1,1);
                                0

```

[We were unlucky. However, with another seed we obtain an innovation:

```

> _seed := 21: Innovation(1,1);
                                .1860056542
> Imitation(1,1);
                                .16
> Innovation(2,1); Imitation(2,1);
                                0
                                .16
> Innovation(3,1); Imitation(3,1);
                                0
                                0
> Innovation(4,1); Imitation(4,1);
                                0
                                .16

```

[In this sequence there are three firms that obtain an imitation (with a boring value) and one firm that obtains an innovation. This is reflected in the TechnoChoice of the firms:

```

> TechnoChoice(1,1); TechnoChoice(2,1);
TechnoChoice(3,1); TechnoChoice(4,1);
                                .1860056542
                                .16
                                .16
                                .16

```

[-] New Capital

Now we come to the investment process. The ability to invest depends on the profits in the present period. Banks can improve the situation but are also using profits as their criteria. The willingness to invest depends on the expected mark-up over costs in the next period (i.e. possibly using the new technology in the estimate).

```
> MaxInvest := proc(i,t)
  global _;
  if _[pi,i,t] <= 0 then _[loans,i] := 0;
  else _[loans,i] := _[b]*_[pi,i,t];
  fi;
  _[I_max,i,t] := _[delta] + _[pi,i,t] + _[loans,i];
end:

> DesiredInvest := proc(i,t)
  global _;
  _[s,i,t] := _[Q,i,t]/_[TQ,t];
  _[rho,i] := _[c]/(_[P,t]*_[A,i,t+1]);
  _[I_des,i,t] := _[delta] + 1
                - _[eta]/(_[eta] - _[s,i,t])*_[rho,i];
end:

> ChangeCapital := proc(i,t)
  global _;
  local Inv;
  _[constraint,i,t] := min(_[I_des,i,t], _[I_max,i,t]);
  Inv := max(0, _[constraint,i,t]);
  _[K,i,t+1] := _[K,i,t]*(Inv + 1 - _[delta]);
end:
```

Before we can run these procedures we need to define some new parameters:

```
> _[b] := 1.0;
                                     _[b] := 1.0

> _[delta] := 0.03;
                                     _[delta] := .03

> _[eta] := 1;
                                     _[eta] := 1

> MaxInvest(1,1); DesiredInvest(1,1); ChangeCapital(1,1);
                                     .0299808541
                                     .029421419
                                     91.27715820

> MaxInvest(2,1); DesiredInvest(2,1); ChangeCapital(2,1);
                                     .0299808541
                                     -.133207960
                                     88.5901
```

For the first firm the innovation means that the firm wants to invest, but investment is smaller than depreciation so capital becomes smaller. In the second case mark-up is considered insufficient. Therefore no investment takes place, and capital decreases by the depreciation.

[-] Parameters

We have defined the parameters successively. Here is a procedure that contains them all. It makes parameter change easier:

```
> Parameters := proc()
  global _;
  _[b] := 1 ;
  _[c] := 0.16 ;
  _[delta] := 0.03 ;
  _[Revenue] := 67 ;
  _[d_im] := 0.3 ;
  _[d_in] := 0.3 ;
  _[eta] := 1 ;
  _[phi] := 0.05 ;
  _[r_im] := 0.00112 ;
```

```

_[r_in]      := 0.0223 ;
_[searchType]:= cumulative;
_[sigma_in] := 0.2    ;
end:

```

[-] The evolution of an industry in t periods

[Now we can define a procedure that uses the specification of the individual decisions etc. to simulate the evolution of an industry:

```

> NelwinDecomp := proc(n,T,seed)
  global _, _seed;
  local i,t;
  with(stats[random]);
  _seed := seed;
  _[newFirm] := 0;
  CreateTable();
  Parameters();
  for i from 1 to n do
    CreateFirm(91.33,.16);
  od;
  for t from 1 to T do
    for i from 1 to n do
      FirmOutput(i,t);
    od;
    Supply(t);
    Price(t);
    Maximum(t);
    for i from 1 to n do
      Innovation(i,t);
      Imitation(i,t);
      TechnoChoice(i,t);
      Profit(i,t);
      MaxInvest(i,t);
      DesiredInvest(i,t);
      ChangeCapital(i,t);
    od;
  od;
end:

```

[-] Model run and data analysis

[-] The time-consuming simulation excersises

[We start by running the model for a relative low number of firms and periods. The reason is that Maple is very slow compared to e.g. C++. Make the run and study the time it takes at your computer system. (REMARK: At the moment the parameters are adapted to 4 firms.) In our first call we use the arguments: firms = 4, periods = 4, seed = 1.

```

> NelwinDecomp(4,4,1);

```

83.35282919

[-] Procedures for data plotting

[To ease the analysis of the results of evolutionary simulations, Andersen et al. (DRUIDIC) have made a series of helping procedures. Here are a couple for plotting of time series data.

[-] Procedure that plots a time series for a firm-level variable

[This procedure draws time series data for the firms of the industry. In order to allow simple comparisons across time series for different variables, each firm is given a specific colour that is the same in all plots. (The procedure can easily be extended with new features, like plotting over specific ranges of firms and periods, etc.)

```

> DrawVariable := proc(var,firmno,periods)
  global peri,plotdata,text;

```

```

local colour,tt,t,n,a,Num,Per;
option `ES Andersen, Aalborg University, 1 Sep 96`;
if nargs = 3 then
    Num := firmno;
    Per := periods;
else
    Num := _[N];
    Per := _[T];
fi;
peri := seq(t, t = 1..Per);
colour := seq('COLOUR'(HUE,n/Num), n=1..Num);
a:=NULL;
for n from 1 to Num do
    a := a, 'CURVES'([seq(MergeLists([peri[tt]],

[_[var,n,tt]]),tt=1..Per)],colour[n],THICKNESS(2));
od;

text := cat(`Variable `,var,` for `, Num, ` firms and
`, Per, ` periods.`);
plotdata := 'PLOT'(a, 'TITLE'(text), AXESSTYLE(NORMAL),
AXESTICKS(5,5));
plotdata;

end:

```

Function that helps the creation of a PLOT data structure

This procedure performs a subtask for the DrawVariable procedure, namely to create data points.

```

> MergeLists := proc(list1, list2)
local LookUpItem,i;
option `ES Andersen, Aalborg University, 24 Aug 96`;

LookUpItem := proc(list1, list2, seriesnumber)
local listnumber;
if type(seriesnumber, even) then
    listnumber := seriesnumber/2;
    RETURN(list1[listnumber]);
else
    listnumber := (seriesnumber-1)/2;
    RETURN(list2[listnumber]);
fi;
end;

if not nops(list1) = nops(list2) then
    ERROR(`the number of elements in lists must be the
same`);
fi;
[seq(LookUpItem(list1, list2, i),
i = 2..nops(list1)*2 + 1)];
end:

```

Legend for plots

```

> Legend := proc(N)
global periods,plotdata,text;
local colour,i,n,a,b;
option `ES Andersen, Aalborg University, 28 Aug 96`;

colour := seq('COLOUR'(HUE,n/N), n=1..N);
a:=NULL; b := NULL;

```

```

for n from 1 to N do
  a := a, 'CURVES'([[0,N-n],[10,N-n]],
    colour[n],THICKNESS(2));
  b := b, 'TEXT'([11,N-n],`Firm
`.n,FONT(TIMES,ROMAN,12));
od;

text := cat(`Legend for `,N,` firms`);
plotdata := 'PLOT'(a, b, 'TITLE'(text),
AXESSTYLE(NONE));
plotdata;

end:

```

Plots of core firm variables

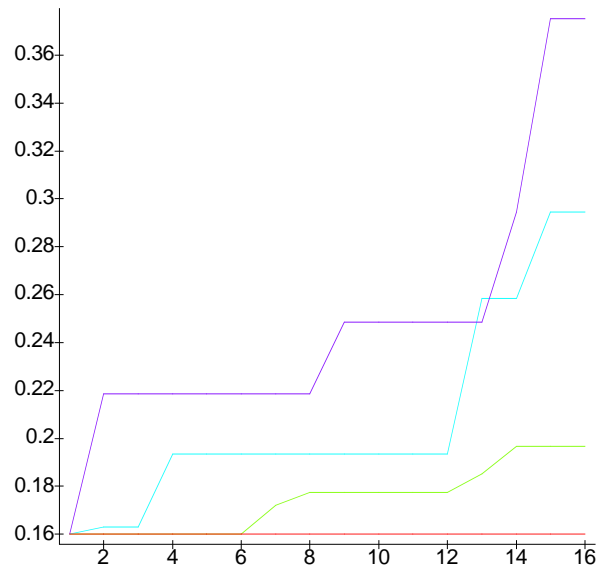
[Let us make a new simulation and analyse the results. We call "StructuredNelwin" with the arguments: firms = 4, periods = 16, seed = 17.

[> **NelwinDecomp(4,16,17):**

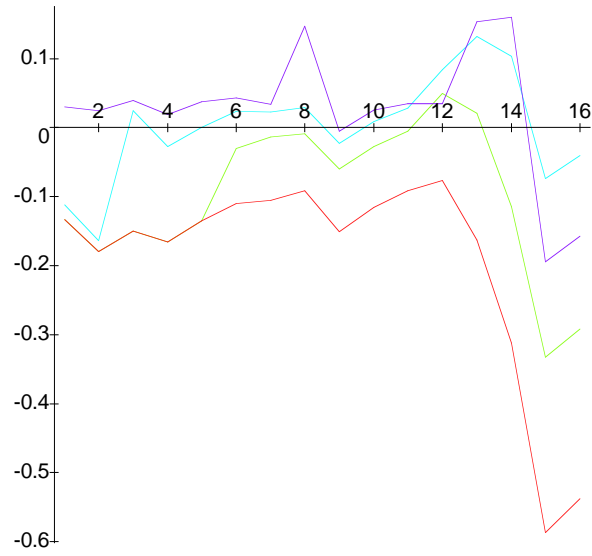
[We now use the plot procedures to draw core variables for firms. To get a plot for all firms and all periods, simply give a variable name as argument.

[> **DrawVariable(A,4,16); DrawVariable(constraint,4,16);**

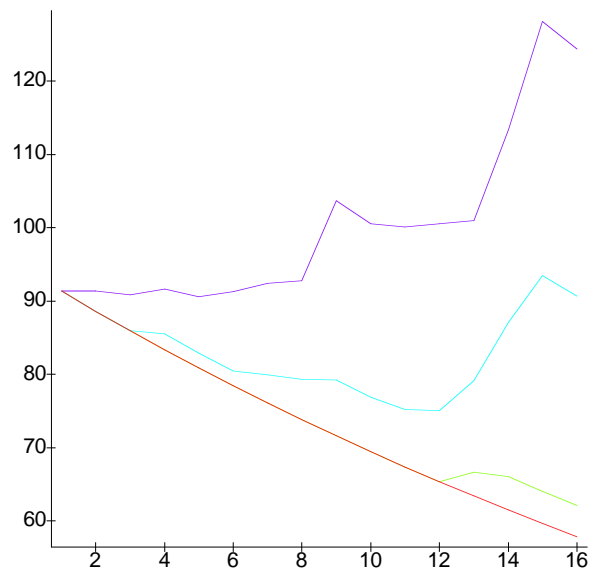
Variable A for 4 firms and 16 periods.



Variable constraint for 4 firms and 16 period

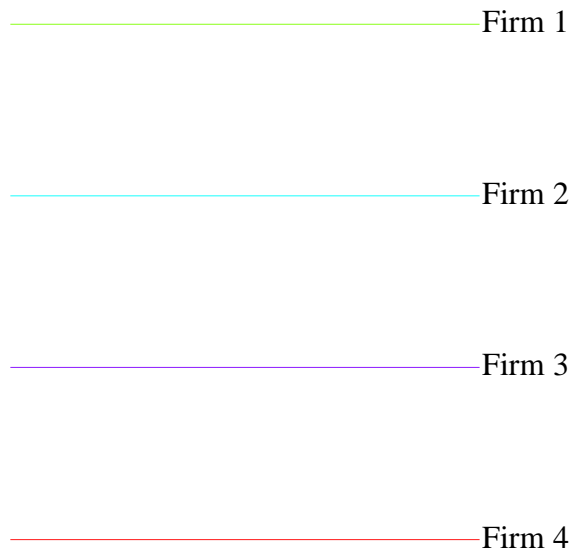


```
> DrawVariable(K,4,16);  
Variable K for 4 firms and 16 periods.
```



```
> Legend(4);
```

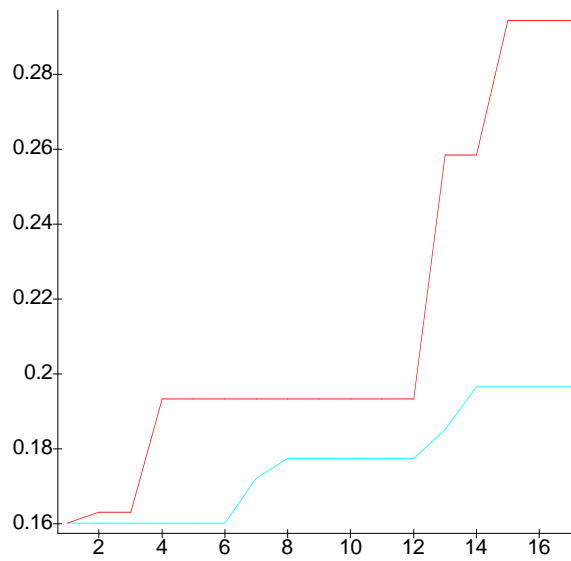
Legend for 4 firms



The DrawVariable procedure allows us to focus on certain firms. For the state variables we can also get the T+1'th value:

```
> DrawVariable(A,2,17);
```

Variable A for 2 firms and 17 periods.



>