dynoptim a Scilab toolbox for discrete time optimal control

S.Berthaud, J.P. Chancelier, M. Cohen de Lara

March 30, 2001

Contents

1 Introduction

The dynoptim package is a Scilab toolbox for solving discrete optimal control problems with possibly state constraints of the form $\underline{x}_t \leq x_t \leq \overline{x}_t$. It involves Scilab code and C coded functions. Scilab code consists of macros which control the steps of each algorithm: a descent method for `dynoptim` which solves the unconstrained case and an Uzawa algorithm for `dynoptimsc` which solves the state constrained case. A set of C coded Scilab primitives are used to compute the cost function and its derivatives with respect to the control. The control problems to be solved are coded in Scilab through the definition of a set of functions and the C code which computes the cost and its derivatives can access to these Scilab function by internally calling Scilab interpreter.

This toolbox is a first step in developing a set of tools for deterministic and stochastic control in Scilab. It could be improved by adding translation facilities of the Scilab coded problems to C coded function for improving running efficiency. And, since the user must provide the derivatives of the functions which specify his problem, the use of the toolbox could be simplified by adding the possibility to differentiate Scilab code automatically.

2 Discrete optimal control problems

The `dynoptim` function aims at solving discrete time optimal control problem (with possibly bounds constraints on the control) by a descent method. `dynoptim` in this case will build an "oracle" which will be devoted to the computation of the gradient of the objective function. Then the standard `optim` primitive of Scilab will perform a gradient algorithm using this "oracle".

Problems to be solved are of the following type:

$$\min_u J(x,u) = \sum_0^{T-1} L(x_t,u_t,t) + \Phi(x_T,T) \tag{1}$$

$$ub_t \leq u_t \leq uh_t, \quad t = 0,\dots T-1 \tag{2}$$

$$x_{t+1} = F(x_t,u_t,t), \quad t \in \{0,\dots,T-1\}, \quad x_0 \quad \text{given} \tag{3}$$

where $X_t \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$. For a given control trajectory $(u_0,\dots,u_{T-1})$, since x_0 is known, the dynamics can be used to compute the state trajectory. Thus we can consider that the value function J only depends on u (and we shall note it $\hat{J}(u)$). Thus the `optim` function will have for a given set of Scilab functions (i.e L, F, Φ and their derivatives) to compute the value function and its derivative as a function depending on u only.

A classical way to compute the derivative of $\hat{J}$ is to use an auxiliary variable called the adjoint state, named Λ, which can be computed given a control and the associated state by solving a backward equation:

$$\left\{\begin{array}{rcl} \forall t & \in & \{T-1,\dots,0\} \\ \Lambda_{T-1} & = & (\frac{d\Phi}{dx}(x_T))', \\ \Lambda_{t-1} & = & (\frac{\partial F}{\partial x}(u_t,x_t,t))'\Lambda_t + (\frac{\partial L}{\partial x}(u_t,x_t,t))'. \end{array}\right. \tag{4}$$

In the sequel $\frac{\partial f}{\partial x}$ will denote the matrix whose (i,j) entry is $\frac{\partial f_i}{\partial x_j}$ and we will denote the transpose of x as x'. Using adjoint state the derivatives of $\hat{J}$ can be computed as follows:

$$\frac{\partial J}{\partial u_k} = \Lambda_k' \frac{\partial F}{\partial u}(x_k,u_k,k) + \frac{\partial L}{\partial u}(x_k,u_k,k) \tag{5}$$

we first start the proof with an easy lemma

Lemme 1 *Let dx_t be the solution of the discrete dynamics:*

$$\left\{\begin{array}{rcl} dx_{k+1} & = & \alpha_k dx_k + \beta_k du_k \\ dx_0 & = & 0 \end{array}\right. \tag{6}$$

and G defined as follows

$$G = \sum_{k=0}^{T-1} \delta_k dx_k + \delta_T dx_T \tag{7}$$

Then, introducing the discrete backward dynamics:

$$\left\{ \begin{array}{rcl} y_{k-1} & = & \alpha'_k y_k + \delta'_k \\ y_{T-1} & = & \delta'_T, \end{array} \right. \tag{8}$$

G can be rewritten as follows

$$G = \sum_{k=0}^{T-1} \left(y'_k \beta_k du_k \right). \tag{9}$$

proof:

$$\begin{aligned} G &= \sum_{k=1}^{T-1} (\delta_k dx_k) + \delta_T dx_T = \sum_{k=1}^{T-1} (y_{k-1} - \alpha'_k y_k)' dx_k + \delta'_T dx_T \\ &= \sum_{k=1}^{T-1} (y'_{k-1} dx_k - y'_k \alpha_k dx_k) + \delta_T dx_T = \sum_{k=0}^{T-2} y'_k dx_{k+1} - \sum_{k=1}^{T-1} y'_k \alpha_k dx_k + \delta_T dx_T \\ &= \underbrace{y'_0 dx_1}_{y'_0 \beta_0 du_0} + \sum_{k=1}^{T-2} y'_k \underbrace{(dx_{k+1} - \alpha_k dx_k)}_{\beta_k du_k} + \underbrace{-y'_{T-1} \alpha_{T-1} dx_{T-1} + \delta_T dx_T}_{y'_{T-1}(\beta_{T-1} du_{T-1})} \\ &= \sum_{k=0}^{T-1} y'_k \beta_k du_k \end{aligned}$$

□

A a small variation of control will lead to the following cost variation

$$\begin{array}{rcl} \Delta J & = & J(u + \delta u) - J(u) \\ & = & \sum_{t=0}^{T-1} \frac{\partial L}{\partial x}(u_t, x_t, t) \delta x_t + \frac{\partial L}{\partial u}(u_t, x_t, t) \delta u_t + \frac{d\Phi}{dx}(x_T, T) \delta x_T \end{array} \tag{10}$$

where δx_t is the solution of the system

$$\left\{ \begin{array}{rcl} \delta x_{t+1} & = & \frac{\partial F}{\partial x}(x_t, u_t, t) \delta x_t + \frac{\partial F}{\partial u}(x_t, u_t) \delta u_t \\ \delta x_0 & = & 0 \end{array} \right. \tag{11}$$

The previous lemma applied to the previous equations gives the result using :

$$\begin{aligned} \alpha_k &= \frac{\partial F}{\partial x}(x_k, u_k, k), \quad k = 0, \ldots, T-1 \\ \beta_k &= \frac{\partial F}{\partial u}(x_k, u_k, k), \quad k = 0, \ldots, T-1 \\ \delta_k &= \frac{\partial L}{\partial x}(x_k, u_k, k), \quad k = 0, \ldots, T-1 \\ \delta_T &= \frac{\partial \Phi}{\partial x}(x_T, T) \end{aligned}$$

3 The dynoptim primitive of the control toolbox

We will call *primitive* a function callable from Scilab but not written in Scilab language and we will call *macro* a Scilab coded function.

Since the computation of the derivatives of $\hat{J}$ is quite time consuming and since we will have to compute it a lot of time in the course of a descent algorithm we have chosen to write the corresponding code in C. The cost and cost derivative computation will be performed by a primitive called `doc_J`.

```
[J,grd,state]=doc_J(u,x0)
```

Note that the functions which characterize the optimal control problem do not appear in the calling sequence of `doc_J`. `doc_J` performs the computation for a current problem and we shall see in a next paragraph how to specify a current problem.

The whole computation is organized in the macro `dynoptim`

```
[uopt,xopt,vopt,gradopt]=dynoptim(L,f,phi,ub,u0,uh,X0 [,stop])
```

which initialize the current problem using the parameters `L`, `f`, `phi` (which are string matrices) then iterate a descent method using `optim` and the oracle function `doc_J`.

In the C coded function `doc_J` the functions which characterize the problem to be solved have canonical names. For example, the discrete dynamics evolution is obtained by calling the C function `f` with is coded as:

```
static void f(double* cmd,double* etat,int* temps,
       double* res,int* n_cmd,int* n_etat)
{
  int pass;
  pass=sciex3(f_name,cmd,etat,temps,res,n_cmd,n_etat,n_etat);
}
```

Since the dynamics of the program to be solved is coded by a Scilab macro the function `f` must find the dynamics macro name, convert C arguments to Scilab data, call the Scilab interpreter and convert the Scilab results back to C data types.

The first step is easy. As pointed out the relevant names are transmitted to `optim` as arguments and during its initialization phase `optim` copies the Scilab names which describe the current optimal control problem into C global string variables. This is performed by a C interface which code a Scilab primitive called `doc_setf`.

The next step is more tricky and the heart of the code is coded in `sciex3`:

```
#define MySciString(ibegin,name,mlhs,mrhs) \
  if( ! C2F(scistring)(ibegin,name,mlhs,mrhs,strlen(name))) \
       { longjmp(Jcenv,-1);return 0; }

int sciex3(char fct[nlgh+1],double* cmd,double* etat,int* temps,
```

```
   double* res,int* n_cmd,int* n_etat, int* n_res)
{
  static int l6,l7,l8;  /* we will use stack poision 6,7 and 8 */
  /* we use three arguments and we return one result */
  static int ibegin=6,mlhs=1,mrhs=3;
  int i,un=1;

  CreateVar(6,"d",n_cmd,&un,&l6);
  for(i=0;i < *n_cmd;i++) stk(l6)[i]=cmd[i];

  CreateVar(7,"d",n_etat,&un,&l7);
  for(i=0;i < *n_etat;i++) stk(l7)[i]=etat[i];

  CreateVar(8,"d",&un,&un,&l8); stk(l8)[0]=*temps;

  /* execute la fonction fct */

  MySciString(&ibegin,fct,&mlhs,&mrhs);

  for(i=0;i<n_res[0];i++) res[i]=stk(l6)[i];
  return 0;
}
```

The code is in fact not so complex. When the `doc_J` function is called from Scilab it is called from an interface (called `InterfaceJ`) and, as explained in the course on interfacing, a set of functions can be used inside the interface to deal with Scilab stack variables. Let first take a look at the interface of `doc_J`:

```
int interfaceJ(char* fname)
{
  static int ierr, static int m1,n1,l1,m2,n2,l2,m3,n3,l3,m4,n4,l4,m5,n5,l5;

  CheckRhs(2,2); CheckLhs(0,3);

  GetRhsVar(1,"d",&m1,&n1,&l1); /* u(.) command */
  GetRhsVar(2,"d",&m2,&n2,&l2); /* x(0) initial state */
  /* output variables */
  n3=1; m3=1;  CreateVar(3,"d",&m3,&n3,&l3);  /*  J */
  n4=n1; m4=m1;  CreateVar(4,"d",&m4,&n4,&l4); /* grad J*/
  n5=(n1+1); m5=m2; CreateVar(5,"d",&m5,&n5,&l5); /* X opt */

  if (( returned_from_longjump = setjmp(Jcenv)) != 0 )
    {
      Scierror(999,"%s: Internal error \r\n",fname);
      return 0;
```

```
    }

  J_GRAD(stk(l1),&n1,&m1,&m2,stk(l2),stk(l5),stk(l4),stk(l3),0);

  LhsVar(1)=3; /* we return 3,4,5 */
  LhsVar(2)=4;
  LhsVar(3)=5;
  return 0;
}
```

Five variables are used in the stack. Two are used to get arguments and three are created to store the return values of `doc_J`. Thus, when we enter the execution of the C function `J_GRAD` we are allowed to use stack positions from position 6 to the end of the stack to perform local computation.

This possibility is used in `sciex3`. Stack position 6,7 and 8 are used to create Scilab variables and to fill them with C data. For example, the statement

```
  CreateVar(6,"d",n_cmd,&un,&l6);
  for(i=0;i < *n_cmd;i++) stk(l6)[i]=cmd[i];
```

creates a Scilab scalar matrix of size `n_cmd`$\times 1$ at position 6 on the stack and fills that matrix with the C array `cmd`.

Then we call the Scilab interpreter through the Scilab primitive `scistring`. `scicstring` performs the evaluation of a Scilab function call given the function name and using arguments stored in the stack at given position. When `scicstring` returns, the results of the Scilab computation are also left on the stack at the same starting position as the given arguments.

Thus the simple statement

```
  for(i=0;i<n_res[0];i++) res[i]=stk(l6)[i];
```

will copy back the result of a Scilab call into the C array `res`

There's a last thing to be explained. The `setjmp` and `longjmp` calls are just here to deal with potential Scilab errors during the Scilab evaluation. In case of errors we want to immediately quit the `J_GRAD` function and return at Scilab level with an error.

The mechanism that we have explained here for the function `f` is used for calling all the functions which characterize our control problem `f`, `f_etat`, `f_cmd`, `phi`, `phi_etat`, `L`, `L_etat`, `L_cmd`.

Since the `dynoptim` functionality is built as a dynamically loaded interface. It is also possible to modify the code and provide C coded cost functions and dynamics. This can be easily done since no modification on the `J_GRAD` function are involved.

3.1 A small example

$$\min_u J(x,u) = u_0^2 + 1.2 x_1' Q x_1 \tag{12}$$

$$-10 \leq u_0 \leq 10 \tag{13}$$

$$x_{t+1} = Ax_T + Bu_t \quad x_0 = (1,2)' \tag{14}$$

This problem can be coded as follows:

- instantaneous cost function and its time and state derivatives

```
deff('y=L(u,x,t)','y=u^2')
deff('y=L_cmd(u,x,t)','y=2*u')
deff('y=L_etat(u,x,t)','y=[0,0]')
```

- system dynamics and its derivatives

```
deff('y=f(u,x,t)','y=B*u+A*x')
deff('y=f_etat(u,x,t)','y=A')
deff('y=f_cmd(u,x,t)','y=B')
```

- final cost and its derivatives (we assume here that `Q` is symmetric)

```
deff('y=phi(x,t)','y= 0.5* x''*Q*x')
deff('y=phi_etat(x,t)','y= Q*x')
```

And the `dynoptim` computation as follows

```
exec SCI/contrib/dynoptim/loader.sce // we load the package

x0=[1;2]; // initial state

// control bounds and initial guess
// ub <= u <= uh and u0 initial value for u
ub= -10; // lower bound (could be \verb+- %inf+)
uh= +10; // lower bound (could be \verb-+ %inf-)
u0= 0;   // intial guess

Q=diag([1,2]);
B=[3;4];
A=[1,2;3,4];

// coding relevant macros names for dynoptim
nom_L=["L","L_cmd","L_etat"];
nom_f=["f","f_cmd","f_etat"];
nom_phi=["phi","phi_etat"];

// calling dynoptim

[Uopt,Eopt,Jopt,Gradopt]=dynoptim(nom_L,nom_f,nom_phi,ub,u0,uh,x0);
```

```
// here we check the results ...

a=1+ 0.5*B'*Q*B ;
b=0.5*( x0'*A'*Q*B + B'*Q*A*x0);
c=0.5* x0'*A'*Q*A*x0 ;
// solve u^2 a + u*b + c ;
uopt = -b/(2*a);
deff('y=cost(u)','y=L(u,x0,0) + phi(f(u,x0,0))')
jopt=cost(uopt) ;

if abs(uopt-Uopt) > 1.e-2 then pause,end
if abs(jopt-Jopt) > 1.e-2 then pause,end
```

4 Taking into account state constraints

In order to take into account state constraints of the form $\underline{x_t} \leq x_t \leq \overline{x}_t$ in the problems exposed in the previous sections we will use augmented Lagrangian techniques.

4.1 Augmented Lagrangian for inequality constraints

We will briefly expose the Augmented Lagrangian techniques for a standard optimization problem and then we will apply it to optimal control problems with state constraints.

In order to solve

$$\min_u \mathcal{J}(u), \tag{15}$$

with

$$\underline{u} < u < \overline{u}, \tag{16}$$

we introduce the associated augmented Lagrangian:

$$L_c(u,p) = \mathcal{J}(u) + \frac{c}{2}\|\text{proj}_{(\mathbb{R}^+)^m}\left(\theta(u) + \frac{p}{c}\right)\|^2 - \frac{1}{2c}\|p\|^2. \tag{17}$$

where $\theta(u)$ is the following function

$$\theta(u) = \begin{pmatrix} u_0 - \overline{u_0} \\ \underline{u_0} - u_0 \\ \vdots \\ u_{p-1} - \overline{u_{p-1}} \\ \underline{u_{p-1}} - u_{p-1} \end{pmatrix}. \tag{18}$$

p is the dimension of u and in this case $\theta(u)$ is of dimension $m = 2p$.

Standard algorithms can be used to solve the Lagrangian problem and we will use an Uzawa algorithm whose steps are described in table 1.

Step 1	Initialize (u, p) with (u^0, p^0) ; and k is set to zero: $k = 0$.
Step 2	solve $\min_u \left(L(u, p^k)\right)$, let u^k be a solution.
Step 3	update p as follows: $p^{k+1} = \left(1 - \frac{\rho}{c}\right) p^k + \frac{\rho}{c}\text{proj}_{(\mathbb{R}^+)^m}\left(p^k + c\theta(u^{k+1})\right)$.
Step 4	If $\|u^{k+1} - u^k\| + \|p^{k+1} - p^k\|$ is small enough, stop else go to step 2.

Table 1: Uzawa Algorithm

Note that the convergence of the Uzawa algorithm depends on the choice of two parameters ρ and c.

- To increase the speed of gradient step 3 one can use large ρ and c.
- But minimisation step 2 can become difficult for large c.

4.2 Augmented Lagrangian for state constraints

We want to solve:

$$\min_u J(x, u) = \sum_0^{T-1} L(x_t, u_t, t) + \Phi(x_T, T) \tag{19}$$

$$ub_t \leq u_t \leq uh_t, \quad t = 0, \ldots T-1 \tag{20}$$

$$x_{t+1} = F(x_t, u_t, t), \quad t \in \{0, \ldots, T-1\}, \quad x_0 \tag{21}$$

$$\underline{x}_t \leq x_t \leq \overline{x}_t, \quad t \in \{0, \ldots, T-1\}\text{given} \tag{22}$$

taking into account state constraints through augmented Lagrangian techniques.

We will show here that the step 2 of Uzawa algorithm can be expressed in this case as an unconstrained optimal control problem. We associate to the state constraint $\underline{x} \leq x \leq \overline{x}$ the constraint vector function $\theta(x)$:

$$\theta(x) = \begin{pmatrix} x_0 - \overline{x_0} \\ \underline{x_0} - x_0 \\ \vdots \\ x_{n-1} - \overline{x_{n-1}} \\ \underline{x_{n-1}} - x_{n-1} \end{pmatrix}. \tag{23}$$

where n is here the state dimension. Note that since x can be expressed as a function of u. The state constraint vector $\theta(x)$ can be considered as a function of u (which will be denoted also $\theta(u)$).

We define now the augmented Lagrangian in the usual way:

$$L_c(u,p) = \sum_0^{T-1} L(x_t,u_t,t) + \Phi(x_T,T) \ + \frac{c}{2}\|\text{proj}_{(\mathbb{R}^+)^m}\left(\theta(u)+\frac{p}{c}\right)\|^2 - \frac{1}{2c}\|p\|^2,$$

This expression can be developed using the definition of $\theta(u)$ and this leads to:

$$\begin{aligned} L_c(u,p) \ = \ & \sum_0^{T-1}\Bigg(L(x_t,u_t,t) + \frac{c}{2}\Bigg(\max(0,x_t-\overline{x}+\frac{p^{(2t)}}{c})^2 \\ & + \max(0,\underline{x}-x_t+\frac{p^{(2t+1)}}{c})^2\Bigg) - \frac{1}{2c}(p^{(2t)}+p^{(2t+1)})\Bigg) \\ & + \Bigg(\Phi(x_T,T) + \frac{c}{2}\Bigg(\max(0,x_T-\overline{x}+\frac{p^{(2T)}}{c})^2 \\ & + \max(0,\underline{x}-x_T+\frac{p^{(2T+1)}}{c})^2\Bigg) - \frac{1}{2c}(p^{(2T)}+p^{(2T+1)})\Bigg), \end{aligned} \tag{24}$$

which can be simplified as

$$L_c(u,p) = \sum_{t=0}^{T-1} L_p(x_t,u_t,t) + \phi_p(x_T,T). \tag{25}$$

where $\phi_p(u,p)$ and L_p are defined as follows

$$\begin{aligned} l_p(x_t,u_t,t) \ = \ & L(x_t,u_t,t) \\ + \ & \frac{c}{2}\left(\max(0,x_t-\overline{x}+\frac{p^{(2t)}}{c})^2 + \max(0,\underline{x}-x_t+\frac{p^{(2t+1)}}{c})^2\right) \\ - \ & \frac{1}{2c}\left(p^{(2t)}+p^{(2t+1)}\right) \end{aligned} \tag{26}$$

$$\begin{aligned} \phi_p(x_T,T) \ = \ & \phi(x_T,u_T,T) \\ + \ & \frac{c}{2}\left(\max(0,x_T-\overline{x}+\frac{p^{(2T)}}{c})^2 + \max(0,\underline{x}-x_T+\frac{p^{(2T+1)}}{c})^2\right) \\ - \ & \frac{1}{2c}\left(p^{(2T)}+p^{(2T+1)}\right), \end{aligned} \tag{27}$$

Thus step 2 of Uzawa algorithm for a fixed p is a standard discrete control problem without state constraints which can be solved by `dynoptim`

4.3 dynoptimsc

The Scilab macro `dynoptimsc` was developed to solve state constraints control problems using Uzawa algorithm. As pointed out in the previous paragraphs The step 2 of the algorithm can be performed by `dynoptim`. The constraints are taken into account through modified version of the cost functions L_p and ϕ_p. A new primitive which is devoted to the computation of $\theta(u)$ is also added. The algorithm implemented in Scilab fig:dynoptimsc is presented in Figure 1.

5 Starts model

We present here an economical model belonging to the starts family models (Cired). A country has to find a new policy which will lead to a reduction of its emissions of green house effect gazes (for example CO_2 emissions) at a specified time horizon. Since the application of a new policy induces economical costs, the country wants to find an admissible policy which will minimize a given cost function. The cost function is taken as an actualized sum of instantaneous costs

$$J(a) = \sum_{t=0}^{T-1} C(a_t, a_{t-1}, t) \frac{1}{(1+\rho)^t}. \tag{28}$$

Where the given instantaneous cost is expressed as

$$C(a_t, a_{t-1}, t) = \alpha \overline{E_t} {a_t}^{\nu} \lambda(t) \gamma(a_t, a_{t-1}).$$

Here a_t is the control and admissible controls must keep the concentration of CO_2 (the state vector noted here M_t) below a specified level.

$$M_t \leq \overline{M}. \tag{29}$$

The state evolution is given by

$$\left\{ \begin{array}{lcl} M_{t_{|t=0}} & = & M_0 \\ M_t & = & M_{t-1} + \Delta t \Big(\beta \overline{E}_{t-1}(1 - a_{t-1}) - \sigma(M_{t-1} - M_{-\infty}) \Big). \end{array} \right. \tag{30}$$

where E_t are given values. This dynamics is based on the fact that CO_2 emissions are partially absorbed by oceans and earth and partially stored in the atmosphere. The problem is a discrete time problem, the step size is one year and the time horizon is $T = 12$.

Many variations on the above problem can be specified according to different economical modeling hypothesis and the availability of a Scilab toolbox for easily testing modification (i.e by only modifying Scilab code) in the cost or in the dynamics specifications was of great interest.

The previous problem can be easily coded in Scilab as follows

- constants and given functions

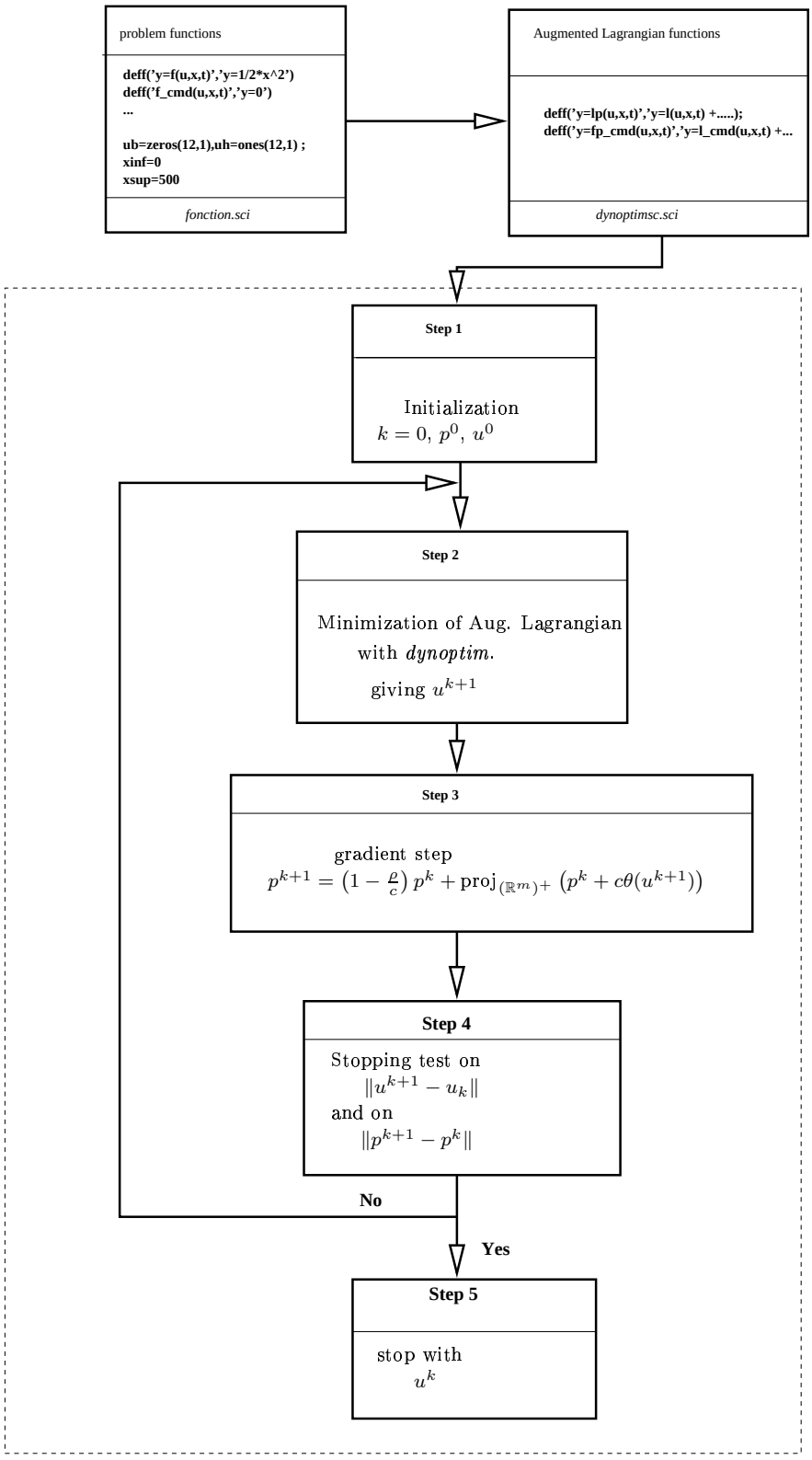

Figure 1: The Scilab macro dynoptimsc

```
E=[5.9623,6.998,8.4363,9.9111,11.018,12.126,13.233,14.541,...
15.848,17.156,18.463,19.771];

alpha=1000; // cost of backstop technology
nu=3;
theta=0.0000526;
M0=360;     // initial concentration
Minf=274;   //  preindustrial concentration
rho=0.05;   // actualization cost
sigma=0.01; // rate of sink absorption
delta=10;   // step size
beta=0.38;  // rate of atmospheric absorption
e=0.0001
Gamma=0.005;

deff('y=Lambda(t)','y=0.25+0.75*exp(-0.01*delta*t)');
deff( 'y=gama(v)','y=1/(2*Gamma*delta)*(v+sqrt(v*v+e))+1');
deff( 'y=gama_v(v)','y=1/(2*Gamma*delta)*(1+v/sqrt(v*v+e))');

deff('y=psi(u1,u2)','y=u2-u1-delta*Gamma');
deff('y=psi_u1(u1,u2)','y=-1')
deff('y=psi_u2(u1,u2)','y=1')
```

- cost function and related derivatives

```
deff('y=L(a,x,t)','y=(alpha*E(t+1)*a^nu*Lambda(t)*gama(psi(x(1),a)))/(1+rho)^(delta*t)')

deff('y=L_cmd(a,x,t)','y=alpha*E(t+1)*(nu*a^(nu-1)*Lambda(t)*gama(psi(x(1),a))+a^nu*Lamb

deff('y=L_etat(a,x,t)','y(1)=-alpha*E(t+1)*a^nu*Lambda(t)*gama_v(psi(x(1),a))/(1+rho)^(d
```

- system dynamics and related derivatives

```
deff('y=f(a,x,t)','y(1)=a,y(2)=x(2)+10*(beta*E(t+1)*(1-a)-sigma*(x(2)-Minf))');
// Warning f_etat must return the matrix M such that
// M(i,j) = dfi/dxj
deff('y=f_etat(a,x,t)','y=zeros(2,2),y(1,1)=0,y(2,1)=0,y(1,2)=0,y(2,2)=1-sigma*delta');
deff('y=f_cmd(a,x,t)','y(1)=1,y(2)=-beta*E(t+1)*delta');
```

- final cost and its derivatives

```
deff('y=phi(x,t)','y=0');
deff('y=phi_etat(x,t)','y(1)=0,y(2)=0');
```

- calling `dynoptimsc`

```
x0=[0;360];  // initial state x
xinf=[0;0];  // state constraints
xsup=[1;450];

// bounds and initial value on the control  a
T=12
ab=zeros(1,T);
ah=zeros(1,T)+1;
a0=rand(1,T);

// names
nom_L=["L","L_cmd","L_etat"];
nom_f=["f","f_cmd","f_etat"];
nom_phi=["phi","phi_etat"];

[Uopt,Eopt,Jopt,Gradopt]=dynoptimsc(nom_L,nom_f,nom_phi,ab,a0,ah,xinf,x0,xsup)
//,'lag',k=zeros(2,1)+1,150,40);

// graphics
years=1990+10*(0:T-1);
xset('window',0);
xbasc()
plot2d2(years,Uopt,2)
xstring(2010,0.5,"time evolution of control")

Emax=xsup(2)*ones(1,T);

xset('window',1);
xbasc()
plot2d2(years,Eopt(2,1:T),5)
plot2d2(years,Emax(1:T),6)
xstring(2030,390,"time evolution of second component of state" )
```