Dynamic optimization and differential games with applications to economics

Yuji Itaya

Ph.D. in Engineering
Professor
Department of Information Management
Asahi University
Hozumi, Motosu, Gifu, 501-0296, Japan
E-mail: itaya@alice.asahi-u.ac.jp

ABSTRACT

In IMS'97 we showed several packages for: (1)solving simultaneous equations in real domain; (2)obtaining necessary conditions of constrained static optimization problems; (3)solving simultaneous nonlinear equations approximately; (4)determining signs of expressions; (5)solving differential equations approximately; (6)solving boundary-value problems approximately; and (7)solving dynamic optimization problems approximately. We also presented applications of these packages to urban economic models[1].

However, applications are limited in the package for solving dynamic optimization problems, since there exist many problems for which the package cannot be applied. In this paper, we revise and expand it so that it can be widely used.

By further expanding the package above, we can solve differential games. We first show a solution of simple differential games with Mathematica and view the optimal trajectories. We then present packages for differential games by upgrading the dynamic optimization package, and show some applications to economic problems.

1. Introduction

Continuous-time dynamic optimization models play a vital role in analyzing time-dependent activities in an economic system. A system is represented by differential equations with respect to time, and its objective is given by integration over time.

Pontryagin's maximum principle are often employed to solve dynamic optimization problems. The principle states that the solutions must satisfy boundary-value differential equations, which in some cases can be solved by the *Mathematica* `DSolve` command with both initial and terminal conditions.

Unfortunately in many cases, however, analytical solutions cannot be obtained, thus we carry out calculations to obtain a numerical solution, which may be undesirable because of obscuring relationships between parameters and solutions. Sometimes, approximate solutions will suffice to investigate the relations.

As described in our paper [1], we developed the commands which solves models approximately. We have upgraded some of the packages and show them in this paper.

When several agents are involved in a model and they act independently or cooperatively, continuous-time dynamic optimization models of these agents are regarded as differential games. Each agent, or player, has its own objective to be maximized. We can use our packages to obtain approximate solutions of differential games and demonstrate their utilization in this paper.

This paper is organized as follows: first, we present a simple economic differential game model; we then obtain its Pareto-optimal exact solution; we describe an open-loop Nash equilibria solution and get it approximately; we show the usage, applicability and advantages of our packages which solve models approximately.

2. Simple economic model

Following references [2] and [3], we consider two communities that carry out a joint project, for example, serving public goods. They make each continuous contribution $x_1(t)$ and $x_2(t)$ over time t. Let $K(t)$ be the total contribution, which accumulates over time as

$$\frac{dK(t)}{dt} = -\delta K(t) + x_1(t) + x_2(t), \quad K(0) = K_0 \tag{1}$$

where δ is a constant decay rate of the contribution and K_0 is an initial contribution at time 0.

It is assumed here that the instantaneous utility of each community is

$$f(K(t)) - C(x_i(t)), \quad i = 1, 2. \tag{2}$$

$f(K)$ is a continuous monotonically increasing production function of contribution, and is assumed to be

$$f(K) = aK - bK^2, \quad K < \frac{a}{2b}. \tag{3}$$

$C(x)$ is a continuous monotonically increasing cost function, and we assume that

$$C(x) = \frac{cx^2}{2}, \quad c > 0. \tag{4}$$

We assume that each community maximizes its discounted utility, i.e.,

$$J_i(x_1(t), x_2(t)) = \int_0^T [f(K(t)) - C(x_i(t))]\, e^{-rt}\, dt, \quad i = 1, 2. \tag{5}$$

where r is a constant discount rate. The model is described as a linear quadratic differential game by these assumptions.

3. Dynamic optimization -- Pareto optimal solution

In this section we obtain a Pareto optimal solution, which has the property that each community cannot increase its utility without degrading the other's. We will first get the exact solution, and then an approximate solution. We will show that in some cases we cannot get the exact solution but an approximate one.

■ 3.1 the exact solution

Pareto optimal solutions can be obtained by setting the joint instantaneous utility to be maximized as the weighted sum of each instantaneous utility, i. e.,

$$F(K(t), x_1, x_2) = \sigma(f(K(t)) - C(x_1(t))) + (1-\sigma)(f(K(t)) - C(x_2(t))),\ 0 \le \sigma \le 1 \qquad (6)$$

where σ is a weight. The Pareto optimal model is described as dynamic optimization and can be solved by Pontryagin's maximum principle[4],[5].

According to the maximum principle, necessary conditions for optimality are

$$H = -F(K(t), x_1, x_2) + \phi(-\delta K(t) + x_1(t) + x_2(t)) \qquad (7)$$

$$\frac{\partial H}{\partial x_1} = 0 \qquad (8)$$

$$\frac{\partial H}{\partial x_2} = 0 \qquad (9)$$

$$\frac{dK(t)}{dt} = \frac{\partial H}{\partial \phi} \qquad (10)$$

$$\frac{d\phi(t)}{dt} = r\phi(t) - \frac{\partial H}{\partial K} \qquad (11)$$

$$K(0) = K_0 \qquad (12)$$

$$\phi(T) = 0 \qquad (13)$$

H is a Hamiltonian, $\phi(t)$ is a costate variable, and the last condition is a transversality condition. Since there are an initial and terminal condition, we can see that we must solve the boundary-value differential equations.

We obtain a Pareto optimal solution with *Mathematica* by setting

$$a = 4$$
$$b = 1$$
$$c = 100$$
$$r = 0.05$$
$$\delta = 0.01$$
$$K_0 = 1$$
$$T = 10$$
$$\sigma = 0.5$$

for simplicity as follows.

First we define the production and cost functions.

```
f[K_] := a K - b K^2 /. a -> 4 /. b -> 1;
```

```
c[x_] := c x^2 / 2 /. c -> 100;
```

Then we define the objective function and system differential equation.

```
F[{K_}, {x_, y_}] :=
 σ (f[K] - c[x]) + (1 - σ) (f[K] - c[y]) /. σ -> 1/2;

g[{K_}, {x_, y_}] := -δ K + x + y /. δ -> 0.01;
```

Constructing the Hamiltonian gives

```
H = -F[{K}, {x1, x2}] + ϕ g[{K}, {x1, x2}]
```

$$\frac{1}{2}\left(-4\,K+K^2+50\,x_1^2\right)+\phi\left(-0.01\,K+x_1+x_2\right)+\frac{1}{2}\left(-4\,K+K^2+50\,x_2^2\right)$$

and maximizing H with respect to x_1 and x_2 yields

```
xSol = Solve[{D[H, x1] == 0, D[H, x2] == 0}, {x1, x2}][[1]]
```

$$\left\{x_1\to-\frac{\phi}{50},\ x_2\to-\frac{\phi}{50}\right\}$$

Forming the boundary-value problem, we have

```
deq = {K'[t] == (D[H, ϕ] /. xSol /. K -> K[t] /. ϕ -> ϕ[t]),
      ϕ'[t] == (r ϕ - D[H, K] /. K -> K[t] /. ϕ -> ϕ[t]),
      K[0] == K0, ϕ[T] == 0} /. r -> 0.05 /. K0 -> 1 /.
  T -> 10
```

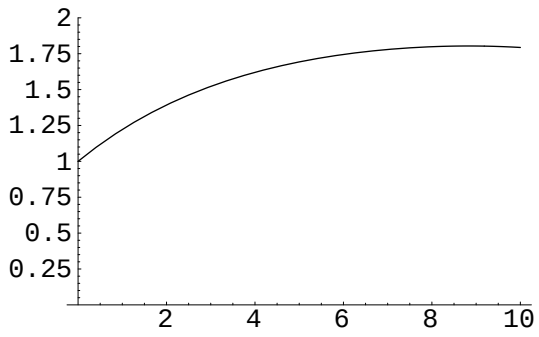
$$\left\{K'[t]==-0.01\,K[t]-\frac{\phi[t]}{25},\ \phi'[t]==4-2\,K[t]+0.06\,\phi[t],\ K[0]==1,\ \phi[10]==0\right\}$$

and solve it.

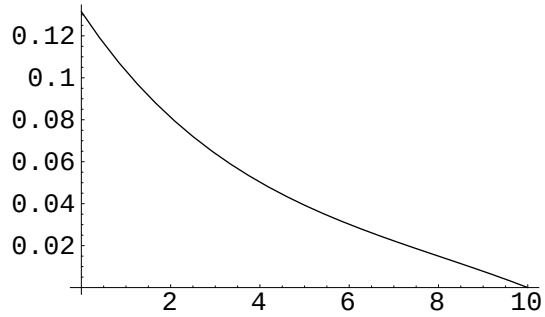
```
dSol = DSolve[deq, {K[t], ϕ[t]}, t];
```

Plotting a graph of $K(t)$ and $x_1(t)$ gives

```
Plot[Release[K[t] /. dSol], {t, 0, 10}, PlotRange -> {0, 2}];
```



```
Plot[Release[x1 /. (xSol /. ϕ -> (ϕ[t] /. dSol))], {t, 0, 10}];
```



The graph of x_2 is the same as that of x_1, since we have assumed that $\sigma = \frac{1}{2}$.

We then try to obtain the exact solution when K_0 and T are not specified and regarded as parameters. Hence, we delete **K₀->1** and **T->10** above and repeat the similar calculation.

```
deqT = {K'[t] == (D[H, ϕ] /. xSol /. K -> K[t] /. ϕ -> ϕ[t]), ϕ'[t] ==
    (r ϕ - D[H, K] /. K -> K[t] /. ϕ -> ϕ[t]), K[0] == K0, ϕ[T] == 0} /.
  r -> 0.05
```

$$\left\{K'[t] == -0.01\,K[t] - \frac{\phi[t]}{25},\ \phi'[t] == 4 - 2\,K[t] + 0.06\,\phi[t],\ K[0] == K_0,\ \phi[T] == 0\right\}$$

```
DSolve[deqT, {K[t], ϕ[t]}, t]
```

$$\text{DSolve}\left[\left\{K'[t] == -0.01\,K[t] - \frac{\phi[t]}{25},\ \phi'[t] == 4 - 2\,K[t] + 0.06\,\phi[t],\ K[0] == K_0,\ \phi[T] == 0\right\},\ \{K[t], \phi[t]\}, t\right]$$

`DSolve` does not return a solution. Apparently we cannot obtain the exact solution of the boundary-value problem with the `DSolve` command.

■ 3.2 an approximate solution

Our scheme here is that we will obtain a solution approximated by a polynomial of an arbitrary degree *n*. It is expected that a solution is closer to the exact one as *n* is larger, although it will take longer computation time. To solve differential equations with boundary values, we have developed packages for solving simultaneous nonlinear equations approximately and for solving differential equations approximately.

■ 3.2.1 approximate solution of simultaneous nonlinear equations

We cannot always obtain solutions of simultaneous nonlinear equations with the *Mathematica* command `Solve`, even if there exists the exact analytical solutions. Newton's method is often used to get numerical solutions. The method can be also used to obtain a symbolic solution approxiamted by a polynomial of parameters.

Consider simultaneous nonlinear equations $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{p})=\boldsymbol{0}$ where $\boldsymbol{f}$ is a nonlinear vector function, $\boldsymbol{x}$ is a variable vector, $\boldsymbol{p}$ is a parameter vector. To solve $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{p})=\boldsymbol{0}$ with respect to $\boldsymbol{x}$, by Newton's method we repeat the following calculation:

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{\partial f(x_n, p)}{\partial x}}. \tag{14}$$

To determine an initial $\boldsymbol{x}_0$, with an approximate value $\boldsymbol{p}_0$ of $\boldsymbol{p}$, we use Newton's method again to solve $\boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{p}_0)=\boldsymbol{0}$.

We have developed the approximateSolve package to carry out the above calculation. First, we read the package:

```
<< approximate`Solve`
```

```
?approximateSolve
```

```
approximateSolve[f,var,var0,prm,degree] solves systems of nonlinear
  equations approximately.  The solution is approximated by degree-th
  degree polynomial of prm. f is a list of nonlinear equations, var is a
  list of variables, var0 is a list of initial values for var, and prm is
  a list of parameters with the corresponding approximate values.
  If the first element of the output is False, no solution is obtained.
```

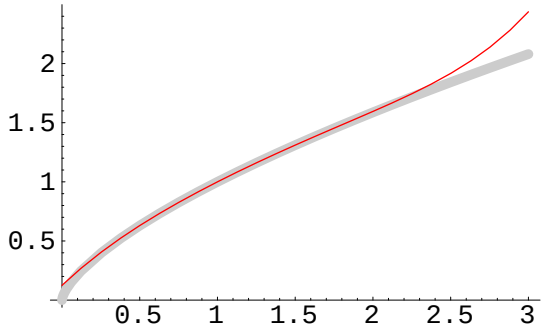
To solve $x^3 - p^2=0$ around $p=1$ with the approximation by a polynomial of degree 5, execute the following command:

```
approximateSolve[{x^3 - p^2 == 0}, {x}, {1.}, {{p, 1.}}, 5]
```

```
{x → 0.124829 + 1.24829 p - 0.624143 p^2 + 0.356653 p^3 -
    0.124829 p^4 + 0.0192044 p^5}
```

To see how close the approximate solution is to the exact one, we plot the exact solution $p^{\frac{2}{3}}$ (the gray thick line) and the approximate solution (red line).

```
Plot[{p^(2/3), Release[x /. %]}, {p, 0, 3},
 PlotStyle -> {{RGBColor[0.8, 0.8, 0.8], Thickness[0.02]},
   RGBColor[1, 0, 0]}];
```



For another example, we show a solution of a little more complicated equations:

```
approximateSolve[{x - y - 2 p^2 == 0, x + y - 4 p == 0}, {x, y},
{1., 1.}, {{p, 1.}}, 2]
```

```
{x → 0. + 2. p + p^2, y → 0. + 2. p - p^2}
```

See Appendix for the code of approximateSolve.

■ 3.2.2 approximate solution of simultaneous differential equations

To obtain a solution of $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{g}(\boldsymbol{x}, t)$ approximately, we use Taylor's expansion

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \frac{d\boldsymbol{x}}{dt}(t-t_0) + \frac{1}{2}\frac{d^2\boldsymbol{x}}{dt^2}(t-t_0)^2 + \frac{1}{3!}\frac{d^3\boldsymbol{x}}{dt^3}(t-t_0)^3 + \cdots \tag{15}$$

and substitute $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{g}(\boldsymbol{x}, t)$, $\frac{d^2\boldsymbol{x}}{dt^2} = \frac{d\boldsymbol{g}(\boldsymbol{x}, t)}{dt} = \frac{\partial \boldsymbol{g}(\boldsymbol{x}, t)}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{dt} + \frac{\partial \boldsymbol{g}(\boldsymbol{x}, t)}{\partial t}$, etc.

The approximateDSolve package carries out the expansion. First, we read the package:

```
<< approximate`DSolve`
```

```
? approximateDSolve
```

```
approximateDSolve[expr,y,y0,x,x0,degree] solves an ordinary differential
  equation approximately. The differential equation must be given by dy/
  dx = expr. with an initial condition y0=y[x0]. approximateDSolve[
  {expr1,expr2,...},{y1,y2,...},{y10,y20,...},x,x0,degree] solves
  a system of ordinary differential equations approximately. The
  system must be given by dy1/dx = expr1, dy2/dx = exp2,... with
  initial conditions y10 = y1[x0], y20 = y2[x0],... The solution
  is approximated by a degree-th degree polynomial of x.
```

To solve $\frac{dx(t)}{dt} = x(t)$ and $x(0) = p$ with a polynomial of degree 5, execute the following command:

```
approximateDSolve[x, x, p, t, 0, 5]
```

$$\left\{x \to p + p\,t + \frac{p\,t^2}{2} + \frac{p\,t^3}{6} + \frac{p\,t^4}{24} + \frac{p\,t^5}{120}\right\}$$

It is easy to show that the approximate solution is a good approximation to the exact one, since the exact solution is $x(t) = p\,e^t$, and its Taylor expansion is

```
Series[p Exp[t], {t, 0, 5}]
```

$$p + p\,t + \frac{p\,t^2}{2} + \frac{p\,t^3}{6} + \frac{p\,t^4}{24} + \frac{p\,t^5}{120} + O[t]^6$$

To solve $\frac{dy_1}{dt} = -2x(t)y_2(t)$, $\frac{dy_2}{dt} = 2x(t)y_1(t)$, $y_1(0) = 1$, and $y_2(0) = 0$ by a polynomial of degree 10, we execute the following command:

```
approximateDSolve[{- 2 x y2, 2 x y1}, {y1, y2}, {1, 0}, x, 0, 10]
```

$$\left\{y1 \to 1 - \frac{x^4}{2} + \frac{x^8}{24},\ y2 \to x^2 - \frac{x^6}{6} + \frac{x^{10}}{120}\right\}$$

The exact solution is $y_1(t) = \cos x^2$ and $y_2(t) = \sin x^2$, and the approximation is good. However, we cannot get the exact solution with the `DSolve` command since

```
DSolve[
 {y1'[x] == -2 x y2[x], y2'[x] == 2 x y1[x], y1[0] == 1, y2[0] == 0},
 {y1[x], y2[x]}, x]
```

```
- Solve::verif : Potential solution {C[1] → 0, C[2] → Indeterminate}
    cannot be verified automatically. Verification may require use of limits.
- Solve::ifun : Inverse functions are being used by Solve, so some
    solutions may not be found.
- Solve::verif : Potential solution {C[1] → 0, C[2] → Indeterminate}
    cannot be verified automatically. Verification may require use of limits.
- Solve::ifun : Inverse functions are being used by Solve, so some
    solutions may not be found.
- Solve::verif : Potential solution {C[1] → 0, C[2] → Indeterminate}
    cannot be verified automatically. Verification may require use of limits.
- General::stop :
   Further output of Solve::verif will be suppressed during this calculation.
- Solve::ifun : Inverse functions are being used by Solve, so some
    solutions may not be found.
- General::stop :
   Further output of Solve::ifun will be suppressed during this calculation.
```

```
{{y2[x] → Indeterminate, y1[x] → Indeterminate}}
```

See Appendix for the code of approximateDSolve.

■ 3.2.3 boundary-value problem

Consider boundary-value differential equations, $\frac{d\boldsymbol{x}_1}{dt} = \boldsymbol{g}_1(\boldsymbol{x}_1, \boldsymbol{x}_2, t)$, $\frac{d\boldsymbol{x}_2}{dt} = \boldsymbol{g}_2(\boldsymbol{x}_1, \boldsymbol{x}_2, t)$, $\boldsymbol{x}_1(0) = \boldsymbol{x}_{10}$, and $\boldsymbol{x}_2(T) = \boldsymbol{x}_{2T}$. Using approximateSolve and approximateDSolve, we can obtain an approximate solution. First, with additional initial conditions $\boldsymbol{x}_2(0) = \boldsymbol{x}_{20}$, we solve differential equations by approximateDSolve to get $\boldsymbol{x}_1(t)$ and $\boldsymbol{x}_2(t)$. Then, we solve $\boldsymbol{x}_2(T) = \boldsymbol{x}_{2T}$ to obtain $\boldsymbol{x}_{20}$ by approximateSolve, and substitute $\boldsymbol{x}_{20}$ back to $\boldsymbol{x}_2(t)$.

The approximateBVPSolve package solves boundary-value differential equations. First, we read the package:

```
<< approximate`BVPSolve`
```

```
? approximateBVPSolve
```

```
approximateBVPSolve[exprs,{y1,y2},{y10,y2T},x,x0,xT,degree] solves a
  system of ordinary differential equations approximately with boundary-
  value conditions. The system must be given by dy1/dx=expr1, dy2/dx=
  expr2,..., with initial conditions y10 (=y1[x0]) and terminal conditions
  y2T (=y2[xT]). y1, y2, y10, and y2T are lists. The solution is
  approximated by degree-th degree polynomial of x. Parameters and their
  approximated values can be specified with approximateBVPSolve[
  exprs,{y1,y2},{y10,y2T},{{p1,p10},{p2,p20},...},x,x0,xT,degree],
  which solves a system of ordinary differential equations as a
  degree-th polynomial of parameters approximately.
```

For solving approximately $\frac{dx_1}{dt} = 2t$, $\frac{dx_2}{dt} = x_1(t)$, $\frac{dx_3}{dt} = x_2(t)$, $x_1(0) = x_{10}$, $x_2(T) = x_{2T}$, and $x_3(T) = x_{3T}$ around $x_{10} = 1$, $x_{2T} = 1$, $x_{3T} = 2$, and $T = 10$ with a polynomial of degree 2, we execute the following command:

```
approximateBVPSolve[
   {2 t, x1, x2}, {{x1}, {x2, x3}}, {{x10}, {x2T, x3T}},
  {{x10, 1.}, {x2T, 1.}, {x3T, 2.}, {T, 10.}},
   t, 0, T, 4] // ExpandAll // N //
Chop
```

```
{x1 → t^2 + x10, x2 → 0.333333 t^3 - 0.333333 T^3 + t x10 - 1. T x10 + x2T,
 x3 → 0.0833333 t^4 - 0.333333 t T^3 + 0.25 T^4 + 0.5 t^2 x10 -
   1. t T x10 + 0.5 T^2 x10 + t x2T - 1. T x2T + x3T}
```

The exact solution can be obtained with the following `DSolve` command, which shows the good approximation.

```
DSolve[{x1'[t] == 2 t, x2'[t] == x1[t],
    x3'[t] == x2[t], x1[0] == x10, x2[T] == x2T, x3[T] == x3T},
   {x1[t], x2[t], x3[t]}, t] // ExpandAll //
 N
```

```
{{x1[t] → t^2 + x10,
  x2[t] → 0.333333 t^3 - 0.333333 T^3 + t x10 - 1. T x10 + x2T,
  x3[t] → 0.0833333 t^4 - 0.333333 t T^3 + 0.25 T^4 + 0.5 t^2 x10 -
    1. t T x10 + 0.5 T^2 x10 + t x2T - 1. T x2T + x3T}}
```

See Appendix for the code of approximateBVPSolve.

■ 3.2.4 an approximate solution of dynamic optimization

Now that everything is ready, we can solve dynamic optimization problems. The dynamicMax package performs the calculation using the maximum principle. We first read the package:

```
<< approximate`dynamicMax`
```

```
? dynamicMax
```

```
dynamicMax[f,g,x,u,u0,x0,t,t0,tT,r,param,n] solves a dynamic maximization
  problem.  The objective function is given by the integral of f[x,u] Exp[
  -r t] from t0 to tT, and the constraints are given by dx/dt = g[x,u],
  where x is a state vector, u is an input vector, u0 is a vector of
  estimated values for u, x0 is an initial state vector, r is a discount
  rate, and param is a list of parameters and their estimated
  values.  The solution is approximated by a polynomial of degree n.
```

Consider a dynamic maximization problem such as

$$\int_0^{\mathrm{T}} [x(t)^2 + u(t)^2]\, \mathrm{e}^{-r\,t}\, dt \longrightarrow \max_{u(t)} \tag{16}$$

$$\text{subject to } \frac{\mathrm{d}x(t)}{\mathrm{d}t} = u(t), \text{ and } x(0) = x_0. \tag{17}$$

To solve the above when $r = 0$, we execute the following command :

```
objective[{x_}, {u_}] := -(x^2 + u^2) / 2;
g[{x_}, {u_}] := u;
dynamicMax[
  objective, g, x, u, 1., x0, t, 0, T, 0, {{T, 10.}, {x0, 1.}}, 3] //
 Chop
```

$$\left\{x \to x0 + \frac{t^2\, x0}{2} + t\,(-0.482863\, x0 - 0.264619\, T\, x0 - 0.00443446\, T^2\, x0 + 0.000108447\, T^3\, x0) + \frac{1}{6}\, t^3\,(-0.482863\, x0 - 0.264619\, T\, x0 - 0.00443446\, T^2\, x0 + 0.000108447\, T^3\, x0),\right.$$
$$\left. u \to -0.482863\, x0 + t\, x0 + \frac{t^3\, x0}{6} - 0.264619\, T\, x0 - 0.00443446\, T^2\, x0 + 0.000108447\, T^3\, x0 - \frac{1}{2}\, t^2\,(0.482863\, x0 + 0.264619\, T\, x0 + 0.00443446\, T^2\, x0 - 0.000108447\, T^3\, x0)\right\}$$

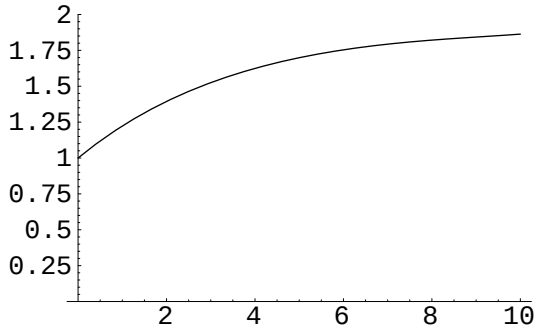
- **3.2.5 an approximate Pareto optimal solution**

We obtain an approximate Pareto optimal solution of our economic model by dynamicMax. The following command solves our model as a polynomial of degree 7.

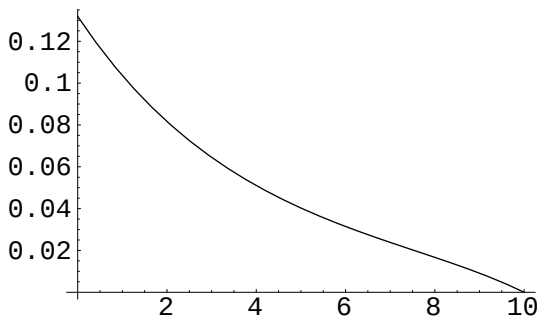
```
Psol =
 dynamicMax[F, g, K, {x1, x2}, {1., 1.}, 1, t, 0, 10, 0.05, {}, 7];
```

As described below, by plotting the optimal trajectories of both $K(t)$ and $x_1(t)$ and comparing them with those of the exact solution, we can see that the solution is well approximated.

```
Plot[Release[K /. Psol], {t, 0, 10}, PlotRange -> {0, 2}];
```




```
Plot[Release[x1 /. Psol], {t, 0, 10}];
```



Contrary to the exact calculation, when T is not specified, an approximate solution, albeit represented by a long expression, is obtained, as can be seen below.

```
PsolT = dynamicMax[F, g, K, {x1, x2}, {1., 1.}, 1, t, 0, T,
  0.05, {{T, 10}}, 7];
```

```
K /. PsolT // Short
```

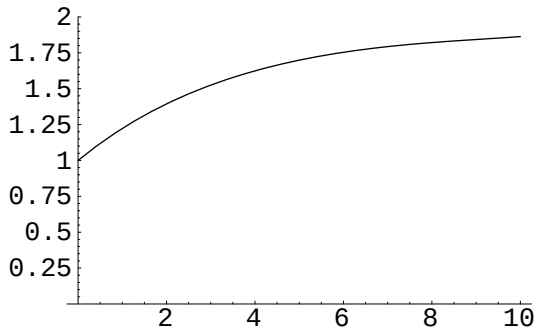
$$1 + t\left(-0.01 + \frac{1}{25}\ (\ll 1\gg)\right) + \ll 4\gg + \ll 1\gg + \frac{t^7\ (\ll 1\gg)}{5040}$$

```
x1 /. PsolT // Short // Chop
```

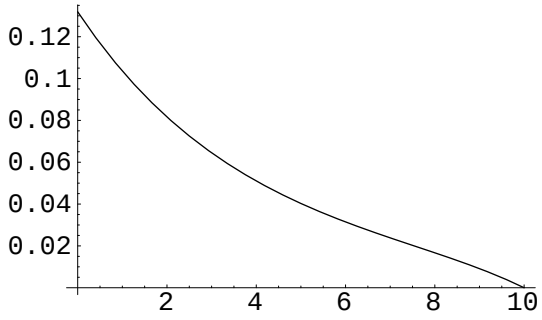
$$\frac{1}{50}\ (-1.57908 + \ll 24\gg)$$

Substituting T=10 to the solution and plotting the trajectories show the credibility of the solution.

```
Plot[Release[K /. PsolT /. T -> 10], {t, 0, 10},
 PlotRange -> {0, 2}];
```




```
Plot[Release[x1 /. PsolT /. T -> 10], {t, 0, 10}];
```



4. Differential games

■ 4.1 open-loop Nash equilibria solution and its necessary conditions

There are several equilibria solutions in differential games: open-loop Nash, feedback Nash, open-loop Stackelberg, and feedback Stackelberg equilibria solutions[6],[7]. In this paper, we investigate open-loop Nash equilibria solutions.

Open-loop Nash equilibria (x_1^*, x_2^*) state that

$$J_1(x_1, x_2^*) \leq J_1(x_1^*, x_2^*) \tag{18}$$

$$J_2(x_1^*, x_2) \leq J_2(x_1^*, x_2^*), \tag{19}$$

which means that the contribution of each community is the optimal response to the contribution by the other community, and vice versa, and that each community has no intention to change its contribution if neither does the other community.

Necessary conditions for open-loop Nash solutions are [6],[7],[8]

$$H_1 = -[f(K(t)) - C(x_1(t))] + \phi_1(-\delta K(t) + x_1(t) + x_2(t)) \tag{20}$$

$$H_2 = -[f(K(t)) - C(x_1(t))] + \phi_2(-\delta K(t) + x_1(t) + x_2(t)) \tag{21}$$

$$\frac{\partial H_1}{\partial x_1} = 0 \tag{22}$$

$$\frac{\partial H_2}{\partial x_2} = 0 \tag{23}$$

$$\frac{dK(t)}{dt} = \frac{\partial H_1}{\partial \phi_1} = \frac{\partial H_2}{\partial \phi_2} \tag{24}$$

$$\frac{d\phi_1(t)}{dt} = r\phi_1(t) - \frac{\partial H_1}{\partial K} \tag{25}$$

$$\frac{d\phi_2(t)}{dt} = r\phi_2(t) - \frac{\partial H_2}{\partial K} \tag{26}$$

$$K(0) = K_0 \tag{27}$$

$$\phi_1(T) = 0 \tag{28}$$

$$\phi_2(T) = 0 \tag{29}$$

■ 4.2 exact solution

The calculations of the previous subsection give the following exact solution.

```
H1 = f[K[t]] - c[x1[t]] + ϕ1[t] g[{K[t]}, {x1[t], x2[t]}];
H2 = f[K[t]] - c[x2[t]] + ϕ2[t] g[{K[t]}, {x1[t], x2[t]}];

xolNashSol =
 Solve[{D[H1, x1[t]] == 0, D[H2, x2[t]] == 0}, {x1[t], x2[t]}];
```

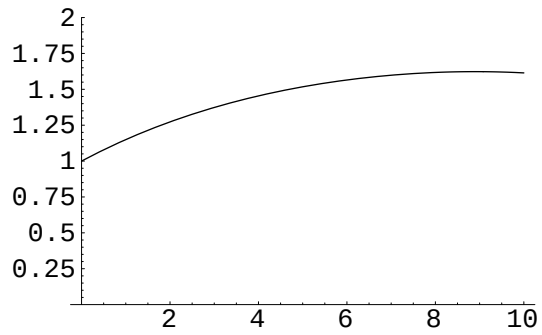
```
deq =
 {φ1'[t] == r φ1[t] - D[H1, K[t]], φ2'[t] == r φ2[t] - D[H2, K[t]],
     K'[t] == g[{K[t]}, {x1[t], x2[t]}],
     K[0] == 1, φ1[T] == 0, φ2[T] == 0} /.
   xolNashSol[[1]] /. r -> 0.05 /.
  T -> 10;

exOlNashSol = DSolve[deq, {K[t], φ1[t], φ2[t]}, t];
```

Plotting the solution shows

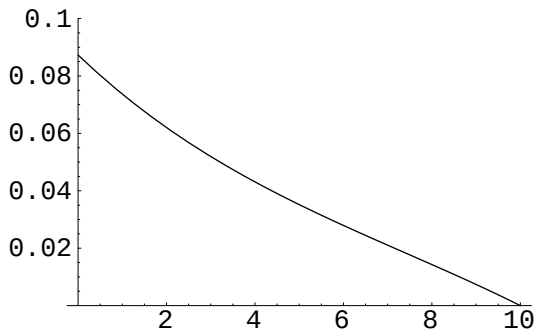
```
phi1 = φ1[t] /. exOlNashSol;
phi2 = φ2[t] /. exOlNashSol;

Plot[Release[K[t] /. exOlNashSol], {t, 0, 10},
 PlotRange -> {0, 2}];
```




```
xt =
 ((x1[t] /. xolNashSol[[1]]) /. φ1[t] -> (φ1[t] /. exOlNashSol) /.
   φ2[t] -> (φ2[t] /. exOlNashSol));

Plot[Release[
  (x1[t] /. xolNashSol[[1]]) /. φ1[t] -> (φ1[t] /. exOlNashSol) /.
   φ2[t] -> (φ2[t] /. exOlNashSol)], {t, 0, 10},
 PlotRange -> {0, 0.1}];
```



When *T* and *r* are not specified, we cannot obtain the exact solution with our computer probably due to limited memory capacity (128MB) as shown below.

```
deqTr =
 {ϕ1'[t] == r ϕ1[t] - D[H1, K[t]], ϕ2'[t] == r ϕ2[t] - D[H2, K[t]],
   K'[t] == g[{K[t]}, {x1[t], x2[t]}],
   K[0] == 1, ϕ1[T] == 0, ϕ2[T] == 0} /.
  xolNashSol[[1]];

exOlNashSolTr = DSolve[deqTr, {K[t], ϕ1[t], ϕ2[t]}, t];
```

```
Out of memory.  Exiting.
```

■ 4.3 approximate solution

Obviously, the necessary conditions for Nash equilibria solutions constitute boundary-value differential equations, an approximate solution of which can be obtained by our approximateBVPSolve command.

The open-loopNash package solves the necessary conditions.

```
<< approximate`openloopNash`
```

```
? approximateOpenloopNash
```

```
approximateOpenloopNash[f1,f2,r,g,x,{u1,u2},{u10,u20},x0,t,t0,tT,param,n]
  obtains a Nash equilibria solution of differential games. The objective
  functions of each player are given by the integral of f1[x,u] Exp[-r t]
  and f2[x,u] Exp[-r t] from t0 to tT, and the constraints are given
  by dx/dt = g[x,u], where x is a state vector, u1 is a strategy
  for player 1, u2 is a strategy for player 2, x0 is an initial
  state, and param is a list of parameters and their estimated
  values.  The solution is approximated by a polynomial of degree n.
```

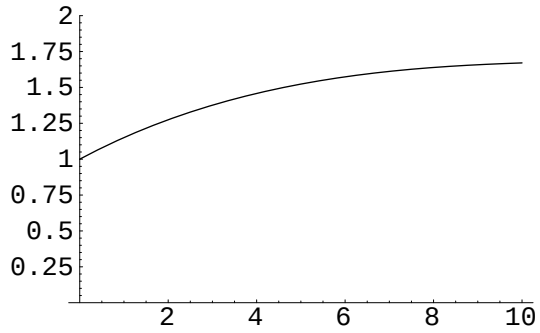
Executing the `approximateOpenloopNash` command for our model gives

```
olNashSol = approximateOpenloopNash[f[K] - c[x1], f[K] - c[x2],
   0.05, g[{K}, {x1, x2}], K, {x1, x2}, {1, 1}, 1, t, 0, 10, {}, 5] //
  ExpandAll
```

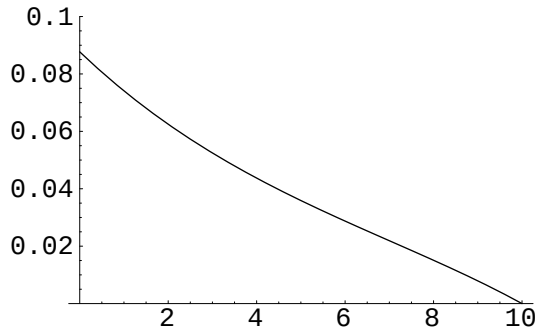
```
{K → 1 + 0.165401 t - 0.015565 t^2 +
    0.000859795 t^3 - 0.0000419141 t^4 + 1.32624 × 10^-6 t^5,
 x1 → 0.0877003 - 0.014738 t + 0.00121187 t^2 -
    0.0000795292 t^3 + 3.10604 × 10^-6 t^4 - 1.30384 × 10^-7 t^5,
 x2 → 0.0877003 - 0.014738 t + 0.00121187 t^2 - 0.0000795292 t^3 +
    3.10604 × 10^-6 t^4 - 1.30384 × 10^-7 t^5}
```

We plot the Nash equilibria trajectories for $K(t)$ and x_1 as follows:

```
Plot[Release[K /. olNashSol], {t, 0, 10}, PlotRange -> {0, 2}];
```




```
Plot[Release[x1 /. olNashSol], {t, 0, 10}, PlotRange -> {0, 0.1}];
```



Even when *r* and *T* are not specified, we still obtain an approximate solution as can be seen below.

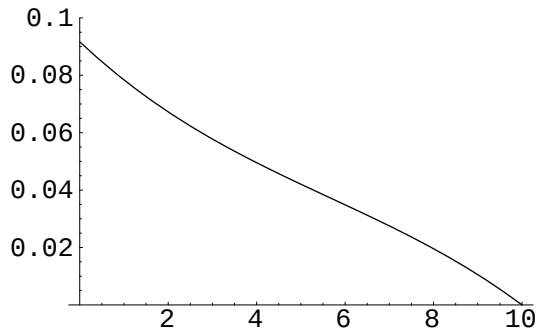
```
olNashSolT = approximateOpenloopNash[
   f[K] - c[x1], f[K] - c[x2], r, g[{K}, {x1, x2}], K,
   {x1, x2}, {1, 1}, 1, t, 0, T, {{T, 10}, {r, 0.05}}, 3] //
  ExpandAll;

Short[olNashSolT[[2]], 5]
```

```
x1 → 0.0243784 + 0.0249347 r + <<126>> + 0.000207377 r^6 t^3 T^3
```

We also see that the solution above coincides with the earlier one when we substitute *T* with 10.

```
Plot[Release[x1 /. olNashSolT /. T -> 10 /. r -> 0.05],
 {t, 0, 10}, PlotRange -> {0, 0.1}];
```



5. Conclusion

In this paper, we show the packages which solve dynamic maximization problems. We also present open-loop Nash equilibria solutions in differential games. There are other kinds of equilibria solution in differential games, which we will pursue in the near future.

Appendix. Packages

Below are the codes of our packages, which are still on development stages.

■ approximateSolve

```
(* :Name: approximateSolve *)

(* :Version: 2.0 beta2 *)

(* :Title: an approximate solution of systems of nonlinear equations *)

(* :Author: Yuji Itaya *)

(* :Context: approximate`Solve` *)

(* :History: Version 2.0 beta2 June 9, 1999;
    -precision -> -precision+3 *)
(* :History: Version 2.0 beta June  5, 1999;
    bug fix for prm=={} *)
(* :History: Version 2.0alpha May 15, 1999;
    change and improve the algorithm *)
(* :History: Version 1.0 February 25, 1997;
    rewrite the algorithm *)
(* :History: Version 0.1 June 25, 1995 *)

(* :Copyright: (c)1995, 1997, 1999, Yuji Itaya *)

(* :Examples:
   approximateSolve[{x^3-p^2==0},{x},{1.},{{p,1.}},5]
   approximateSolve[x^3-p^2==0,x,1.,{{p,1.}},5]
   approximateSolve[{x^3-p^2}=={0},{x},{1.},{{p,1.}},5]
   approximateSolve[{x y-p^2+1},{x},{1.},{{p,2.},{y,1.}},4,debug->On]
   approximateSolve[{x y-p^2,x+y-3p},{x,y},{1.,2.},{{p,1}},7]
   approximateSolve[{a
b-p^2,a+b-3p,z-a-b^2},{a,b,z},{1.,2.,5.},{{p,1}},2,debug->On]
   approximateSolve[1+p2+2x-p1  x^2==0,x,0.,{{p1,1},{p2,2}},5]
   approximateSolve[Exp[x]-p1-p2==0,x,1.,{{p1,1},{p2,1}},4]
   approximateSolve[{x-y-2p^2==0,x+y-4p==0},{x,y},{1.,1.},{{p,1.}},2]
*)

BeginPackage["approximate`Solve`"]

approximateSolve::usage="approximateSolve[f,var,var0,prm,degree] solves
systems of nonlinear equations approximately.  The solution is
approximated by degree-th degree polynomial of prm. f is a list of
nonlinear equations, var is a list of variables, var0 is a list of
initial values for var, and prm is a list of parameters with the
corresponding approximate values.  If the first element of the output is
False, no solution is obtained."

debug::usage="debug is an option; debug->On for debugging."

(* error messages *)
approximateSolve::inapproEq="Inappropriate equations."
```

```
approximateSolve::inapproVar="Inappropriate variable or initial value."

approximateSolve::notFindInitPoint="Cannot find the initial point."

approximateSolve::singularPoint="Attain a singular point; cannot
continue."

approximateSolve::inapproArg="Inappropriate arguments."

Options[approximateSolve]={debug->Off};

Begin["`Private`"]

(* Hessian matrix *)
Hessian[f_,x_]:=Transpose[Map[D[f,#]&,x]];

(* Implementing Newton's method; f is a function, x is a variable, and
x0 is an initial point. *)
newtonMethod[f_,x_,x0_]:=
  Module[{h=Hessian[f,x],precision=Precision[x0],det},
    If[precision===0,precision=$MachinePrecision];
    If[precision===Infinity,precision=$MachinePrecision];
    det=SetPrecision[  Abs[Det[h]]/.Thread[Rule[x,x0]],precision+10];
    If[NumberQ[det] && (det<10^(-precision+3)),
      Return[{False,(x-Inverse[h].f)/.Thread[Rule[x,x0]]}],
      Return[{True,(x-Inverse[h].f)/.Thread[Rule[x,x0]]}]];
  ];

(* check if list_ is a list of equations *)
eqListQ[list_]:=Apply[And,Map[(Head[#]===Equal)&,list]];

findRoot[fl_List==fr_List,x_,x0_,opt___]:=
  Module[{},
    If[Not[VectorQ[fl] && VectorQ[fr] && (Length[fl]==Length[fr]) ],
      Message[approximateSolve::inapproEq];
      Return[{False,Thread[Rule[x,Indeterminate]]}];
    ];
    Return[findRoot[fl-fr,x,x0,opt]];
  ];

findRoot[fl_==fr_,x_,x0_,opt___]:=
  Module[{},
    If[Not[(Length[x]==0) && (Length[x0]==0) ],
      Message[approximateSolve::inapproVar];
      Return[{False,Thread[Rule[x,Indeterminate]]}];
    ];
    Return[findRoot[{fl-fr},{x},{x0},opt]];
  ];

findRoot[f_?eqListQ,x_List,x0_List,opt___]:=
  Module[{},
    Return[findRoot[Map[(#[[1]]-#[[2]])&,f],x,x0,opt]];
  ];

(* finding a root of a function f by Newton's method *)
findRoot[f_List,x_List,x0_List,opt___]:=
  Module[

{precision=Max[Map[If[Precision[#]==Infinity,$MachinePrecision,Precision[
#]]&,x0]],
```

```
       xOld,xNew,nonsingular=True,fNew,iteration=0,debugging},
    (* If the argument is not correct, an error message is displayed,
and x->Indeterminate *)
    If[Not[VectorQ[f] && VectorQ[x] &&
        VectorQ[x0]&&(Length[f]==Length[x])&&(Length[x]==Length[x0]) ],
      Message[approximateSolve::inapproArg];
      Return[{False,Thread[Rule[x,Indeterminate]]}];
    ];
    If[precision==0,precision=$MachinePrecision];
    If[precision==Infinity,precision=$MachinePrecision];
    Options[findRoot]={debug->Off};
    debugging=debug/.{opt}/.Options[findRoot];
    xOld=SetPrecision[x0,precision+20];
    xNew=SetPrecision[x0,precision+20];
    fNew=SetPrecision[f/.Thread[Rule[x,xOld]],precision+20];
    (* successively implementing Newton's method until the value of the
function is small *)
    While[(iteration<=$RecursionLimit)  &&
        Apply[Or,Map[Abs[#]>10^(-precision+3) &,fNew]],
      iteration++;
      xOld=xNew;
      {nonsingular,xNew}=newtonMethod[f,x,xOld];
      If[Not[nonsingular],Break[]];
      fNew=SetPrecision[f/.Thread[Rule[x,xNew]],precision+20];
      If[(debugging==On)||(debugging==All),
         Print["No. ", iteration," trial: At ",Thread[Rule[x,xNew]],
            ", the value of the function is ", fNew]];
      ];
    If[Apply[Or,Map[Abs[#] >10^(-precision+3) &,fNew]] ||
Not[nonsingular],
       {False,Thread[Rule[x,N[xNew,precision]]]},
       {True, Thread[Rule[x,N[xNew,precision]]]}]
  ];

findRoot[False,x_,x0_,opt___]:=
  Module[{},
    Message[approximateSolve::inapproEq];
    Return[{False,Thread[Rule[x,Indeterminate]]}];
  ];

approximateSolve[fl_List==fr_List,var_List,var0_List,prm_List,degree_,opt
___]:=
  Module[{},
    If[Not[VectorQ[fl] && VectorQ[fr] && (Length[fl]==Length[fr])],
      Message[approximateSolve::inapproEq];
      Return[{False,Thread[Rule[var,Indeterminate]]}];
    ];
    Return[approximateSolve[fl-fr,var,var0,prm,degree,opt]];
  ];

approximateSolve[fl_==fr_,var_,var0_,prm_,degree_,opt___]:=
  Module[{},
    If[Not[(Length[var]==0) && (Length[var0]==0) ],
      Message[approximateSolve::inapproVar];
      Return[{False,Thread[Rule[var,Indeterminate]]}];
    ];
    Return[approximateSolve[{fl-fr},{var},{var0},prm,degree,opt]];
  ];

approximateSolve[f_?eqListQ,var_List,var0_List,prm_List,degree_,opt___]:=
```

```
  Module[{},

Return[approximateSolve[Map[(#[[1]]-#[[2]])&,f],var,var0,prm,degree,opt]]
;
  ];

approximateSolve[f_List,var_List,var0_List,prm_List,degree_,opt___]:=

Module[{i,j,func,x0,sol,g,t,d,com,comm,poly,param,dummy,xOld,xNew,prmDegr
ee,
         iteration,nonsingular,debugging},
    debugging=debug/.{opt}/.Options[approximateSolve];
    If[prm==={},param={{dummy,0}},param=prm];
    (* finding a root of f(var)==0 when prm is set as given. *)
    If[debugging===All,

{nonsingular,x0}=findRoot[(f/.Map[Apply[Rule,#]&,param]),var,var0,opt],

{nonsingular,x0}=findRoot[(f/.Map[Apply[Rule,#]&,param]),var,var0]];
    If[Not[nonsingular],
      Message[approximateSolve::notFindInitPoint];
      Return[Thread[Rule[var,Indeterminate]]]];
    If[(debugging===On) || (debugging===All),
      Print[x0," satisfies
",(f/.Map[Apply[Rule,#]&,param]),"==",Table[0,{Length[var]}]]];
    xOld:=var/.x0;
    If[degree<=0,Return[Thread[Rule[var,xOld]]]];
    prmDegree=Transpose[
Append[Transpose[param],Table[degree,{Length[param]}]]];
    (* successively implementing Newton's method *)
    For[iteration=1,iteration<=(degree+1)/2,iteration++,
     {nonsingular,xNew}=newtonMethod[f,var,xOld];
     If[Not[nonsingular],
       Message[approximateSolve::singularPoint];
       Return[Thread[Rule[var,Indeterminate]]]];
     xNew=Expand[Normal[Series[xNew,Apply[Sequence, prmDegree]]]];
     If[(debugging===On) || (debugging===All),
       Print["No. ", iteration," trial: ",xNew]];
     xOld=xNew;
    ];
    Return[Thread[Rule[var,xNew]]];
  ];

approximateSolve[False,var_,var0_,prm_,degree_,opt___]:=
  Module[{},
    Message[approximateSolve::inapproEq];
    Return[{False,Thread[Rule[x,Indeterminate]]}];
  ];

End[];(* end of the private context *)

EndPackage[] (* end of the package "approximate`Solve" context *)
```

■ approximateDSolve

```
(* :Name: approximateDSolve *)

(* :Version: 1.0 *)
```

```
(* :Title: approximate solution of systems of ordinary differential
equations
 *)

(* :Author: Yuji Itaya *)

(* :Context: approximate`DSolve` *)

(* :Requirements: No special system requirements *)

(* :History: Version 1.1 February 28, 1997;
    change the output format;
    add some comments and error messages
*)
(* :History: Version 1.0 March 9, 1995 *)

(* :Warning:
*)

(* :Copyright: (c) 1995, 1997, Yuji Itaya *)

(* :Examples:
    approximateDSolve[x,x,p,t,0,5]
    approximateDSolve[{x,2},{x,y},{1,0},t,0,5]
      Exact solution: x=Exp[t],y=2t
    approximateDSolve[{- 2 x y2, 2 x y1},{y1, y2},{1,0},x,0,16]
      Exact solution: y1=Cos[x^2],y2=Sin[x^2]
*)

BeginPackage["approximate`DSolve`"]

approximateDSolve::usage=
"approximateDSolve[expr,y,y0,x,x0,degree] solves an ordinary
differential equation approximately. The differential equation must be
given by dy/dx = expr. with an initial condition y0=y[x0].
approximateDSolve[{expr1,expr2,...},{y1,y2,...},{y10,y20,...},x,x0,degree
] solves a system of ordinary differential equations approximately. The
system must be given by dy1/dx = expr1, dy2/dx = exp2,... with initial
conditions y10 = y1[x0], y20 = y2[x0],... The solution is approximated
by a degree-th degree polynomial of x."

(* Error Messages *)
approximateDSolve::neq="The numbers of expressions, independent
variables, and initial conditions are not equal."

approximateDSolve::indv="The independent variable `1` must be a symbol."

approximateDSolve::depv="The dependent variable `1` must be a symbol or
a list of symbols."

approximateDSolve::deg="The degree `1` must be an integer."

Begin["`Private`"]

approximateDSolve[exprs_List,y_List,y0_List,x_Symbol,x0_,degree_Integer]:
=
  Module[{sol,r1,r2},
    If[!(Length[exprs]==Length[y]==Length[y0]),
       (* if the number of elements in exprs, y, and y0 are not equal *)
       Message[approximateDSolve::neq];Return[]];
```

```
    If[Length[Cases[y,_Symbol]]!=Length[y],
      (* if y is not a list of symbols *)
       Message[approximateDSolve::depv,y];Return[]];
    r1=Union[{x->x0},Thread[Rule[y,y0]]]; (* initial conditions *)
    r2=Thread[Rule[D[y,x,NonConstants->y],exprs]];
        (* a transformation rule for derivate of variables;
           replace dy/dx with the corresponding expr *)
    sol=y0+(x-x0) (exprs/.r1);  (* first degree approximation *)
    (* successive approximation to a higher degree *)
    Thread[y->
             Nest[{#[[1]] (x-x0)/#[[3]],D[#[[2]],x,
                  NonConstants->y]/.r2,#[[3]]+1,#[[4]]+(#[[2]]/.r1)
#[[1]]}&,

{(x-x0)^2/2,D[exprs,x,NonConstants->y]/.r2,3,sol},degree-1]
                  [[4]]]
  ];

(* syntax check *)
approximateDSolve[exprs_,y_,y0_,x_,x0_,degree_]:=
  Module[{},
    If[(Head[exprs]===List)&&(Head[y]===List)&&(Head[y0]===List),
      (* if all of exprs, y, and y0 are lists *)
      If[!(Length[exprs]==Length[y]==Length[y0]),
        (* if the number of elements in exprs, y, and y0 are not equal *)
        Message[approximateDSolve::neq]],
      (* if all of exprs, y, and y0 are not lists *)
      If[(Head[exprs]=!=List)&&(Head[y]=!=List)&&(Head[y0]=!=List),
        (* if all of exprs, y, and y0 are symbols *)
        Return[approximateDSolve[{exprs},{y},{y0},x,x0,degree]],
        (* if some of exprs, y, and y0 are lists and some are symbols *)
        Message[approximateDSolve::neq]]];
    If[Head[x]=!=Symbol,Message[approximateDSolve::indv,x]];
    If[Head[y]==List,
       (* if y is a list *)
       If[Length[Cases[y,_Symbol]]!=Length[y],
         (* if y is not a list of symbols *)
         Message[approximateDSolve::depv,y]],
       (* if y is not a list *)
       If[Head[y]=!=Symbol,
         (* if y is not a symbol *)
         Message[approximateDSolve::depv,y]]];
    If[Head[degree]=!=Integer,
         (* if degree is not an integer *)
         Message[approximateDSolve::deg,degree]];
  ]

End[]; (* end the private context *)

EndPackage[] (* end the package "approximate`DSolve`" context *)
```

■ approximateBVPSolve

```
(* :Name: BVPSolve *)

(* :Version: 2.0 alpha *)

(* :Title: approximate solution of systems of ordinary differential
equations
```

```
           with boundary-value conditions *)

(* :Author: Yuji Itaya *)

(* :Context: approximate`BVPSolve` *)

(* :Requirements: No special system requirements *)

(* :History: Version 2.0 alpha May 18, 1999;
   use estimate values as initial
   values for the approximateSolve *)
(* :History: Version 1.1 March 1, 1997;
   add some comments and error messages *)
(* :History: Version 1.0 June 25, 1995 *)

(* :Warning:
*)

(* :Copyright: (c) 1995, 1997, 1999 Yuji Itaya *)

(* :Examples:
  approximateBVPSolve[{2t,x1,x2},{{x1},{x2,x3}},{{x10},{x2T,x3T}},
     {{x10,1.},{x2T,1.},{x3T,2.},{T,10.}},t,0,T,2]
  approximateBVPSolve[{- 2 t x2, 2 t x1},{{x1}, {x2}},{{x10},{x2T}},
     {{x10,1.},{x2T,10.},{T,100.}},t,0,T,3]
  approximateBVPSolve[{2t, x1, x2},{{x1}, {x2,x3}},{{x10},{x2T,x3T}},
     {{x10,1.},{x2T,10.},{T,100.},{x3T,1.}},t,0,T,3]
  approximateBVPSolve[{2t, x1, x2 p},{{x1}, {x2,x3}},{{x10},{x2T,x3T}},
     {{p,1},{x10,1.},{x2T,10.},{T,100.},{x3T,1.}},t,0,T,3,debug->On]
*)

BeginPackage["approximate`BVPSolve`",{"approximate`Solve`","approximate`D
Solve`"}]

approximateBVPSolve::usage="approximateBVPSolve[exprs,{y1,y2},{y10,y2T},x
,x0,xT,degree] solves a system of ordinary differential equations
approximately with boundary-value conditions. The system must be given
by dy1/dx=expr1, dy2/dx=expr2,..., with initial conditions y10 (=y1[x0])
and terminal conditions y2T (=y2[xT]). y1, y2, y10, and y2T are lists.
The solution is approximated by degree-th degree polynomial of x.
Parameters and their approximated values can be specified with
approximateBVPSolve[exprs,{y1,y2},{y10,y2T},{{p1,p10},{p2,p20},...},x,x0,
xT,degree], which solves a system of ordinary differential equations as
a degree-th polynomial of parameters approximately. "

debug::usage="debug is an option; debug->On for debugging."

(* error messages *)
approximateBVPSolve::y1neq="The numbers of elements in `1` and `2` are
not equal."

approximateBVPSolve::y2neq="The numbers of elements in `1` and `2` are
not equal."

approximateBVPSolve::exprneq="The numbers of expressions `1` and
dependent variables `2` `3` are not equal."

approximateBVPSolve::degint="Degree `1` is not an integer."

approximateBVPSolve::ind="The independent variable `1` is not a symbol."
```

```
Options[approximateBVPSolve]={debug->Off};

Begin["`Private`"]

(* Extract estimate values of varList from parameter list, prmlist *)

initialValues[prmList_,varList_]:= varList/.Map[Apply[Rule,#]&,prmList];
(*initialValues[prmList_,varList_]:=
  Transpose[Map[Cases[prmList,{#,_}][[1]]&,varList]][[2]];*)

approximateBVPSolve[exprs_List,{y1_List,y2_List},{y10_List,y2T_List},para
m_List,x_,x0_,xT_,degree_,opt___]:=
  Module[{sol,dsol,y20v,y20,y2degree,i},
    debugging=debug/.{opt}/.Options[approximateBVPSolve];
    y2degree=Length[y2];

If[Head[x]=!=Symbol,Message[approximateBVPSolve::ind,x];Return[Thread[Rul
e[Join[y1,y2],Indeterminate]]]];

If[Head[degree]=!=Integer,Message[approximateBVPSolve::degint,degree];Ret
urn[Thread[Rule[Join[y1,y2],Indeterminate]]]];

If[Length[y1]=!=Length[y10],Message[approximateBVPSolve::y1neq,y1,y10];Re
turn[Thread[Rule[Join[y1,y2],Indeterminate]]]];

If[Length[y2]=!=Length[y2T],Message[approximateBVPSolve::y2neq,y2,y2T];Re
turn[Thread[Rule[Join[y1,y2],Indeterminate]]]];
    If[Length[exprs]=!=(Length[y1]+Length[y2]),

Message[approximateBVPSolve::exprneq,exprs,y1,y2];Return[Thread[Rule[Join
[y1,y2],Indeterminate]]]];
    y20v=Table[y20[i],{i,y2degree}];
    dsol=approximateDSolve[exprs,Join[y1,y2],Join[y10,y20v],x,x0,degree];
    If[(debugging===On) || (debugging===All),
      Print["Intermediate result --a solution of differentail equations:
",dsol] ];
    If[debugging===All,

sol=approximateSolve[((y2/.dsol)/.x->xT)-y2T,y20v,initialValues[param,y2T
],param,degree,debug->All],

sol=approximateSolve[((y2/.dsol)/.x->xT)-y2T,y20v,initialValues[param,y2T
],param,degree]];
    If[(debugging===On) || (debugging===All),
      Print["Intermediate result --a solution of initial values:
",sol] ];
    Return[dsol/.sol];
];

approximateBVPSolve[exprs_List,{y1_List,y2_List},{y10_List,y2T_List},x_,x
0_,xT_,degree_,opt___]:=
  approximateBVPSolve[exprs,{y1,y2},{y10,y2T},{},x,x0,xT,degree,opt]

End[] (* end of the private context *)

EndPackage[] (* end of the package "approximate`BVPSolve`" context *)
```

■ approximateDynamicMax

```
(* :Name: dynamicMax *)

(* :Version: 2.0 alpha 2 *)

(* :Title: approximate solution of dynamic optimization problem with
           an integral objective function to be maximized subject to
           differential equations. *)

(* :Author: Yuji Itaya *)

(* :Context: approximate`dynamicMax` *)

(* :Requirements: No special system requirements *)

(* :History: Version 2.0 alpha 2 June 10, 1999;
   include a discount rate as a parameter  *)
(* :History: Version 2.0 alpha May 25, 1999;
   revised the algorithm *)
(* :History: Version 1.1 March 5, 1997;
   extended to multivariable systems;
   added some comments and error messages *)
(* :History: Version 1.0 July 8, 1995 *)

(* :Warning:
*)

(* :Copyright: (c) 1995, 1997, 1999 Yuji Itaya *)

(* :Examples:
****** Example 1: single-input single-state without parameters ******
objective[{x_},{u_}]:=-(x^2+u^2)/2;
g[{x_},{u_}]:=u;
dynamicMax[objective,g,x,u,1.,x0,t,0,T,0,{{T,10},{x0,1}},2]

****** Example 2: single-input single-state with a parameter w ******
objective[{x_},{u_}]:=-(x^2+w u^2)/2;
g[{x_},{u_}]:=u;
dynamicMax[objective,g,x,u,1.,x0,t,0,T,0.05,{{T,10},{x0,1},{w,1}},2]

****** Example 3: single-input single-state ******
objective[{x_},{u_}]:=-(x^2+u^2)/2;
g[{x_},{u_}]:= u;
dynamicMax[objective,{g},{x},{u},{1.},{x0},t,0,T,0,{{T,10},{x0,1}},3]

****** Example 4: single-input multiple-state ******
objective[{x1_,x2_},{u1_}]:=-(x1^2+x2^2+u1^2)/2;
g1[{x1_,x2_},{u1_}]:=u1;
g2[{x1_,x2_},{u1_}]:=u1;
dynamicMax[objective,{g1,g2},{x1,x2},u1,1.,{x10,x20},t,0,
    T,0,{{T,10},{x10,1},{x20,1}},2]

****** Example 5: multiple-input single-state ******
objective[{x_},{u1_,u2_}]:=-(x^2+u1^2+u2^2)/2;
g[{x_},{u1_,u2_}]:=u1+u2;
dynamicMax[objective,g,x,{u1,u2},{1.,1.},x0,t,0,T,0,{{T,10},{x0,1}},2]
```

```
****** Example 6: multiple-input multiple-state ******
objective[{x1_,x2_},{u1_,u2_}]:=-(x1^2+x2^2+u1^2+u2^2)/2;
g1[{x1_,x2_},{u1_,u2_}]:=u1;
g2[{x1_,x2_},{u1_,u2_}]:=u2;
dynamicMax[objective,{g1,g2},{x1,x2},{u1,u2},{1.,1.},{x10,x20},t,0,
  T,0,{{T,10},{x10,1},{x20,1}},3]

****** Example 7: multiple-input multiple-state with a parameter a ******
objective[{x1_,x2_},{u1_,u2_}]:=-(x1^2+x2^2+u1^2+u2^2)/2;
g1[{x1_,x2_},{u1_,u2_}]:=a u1;
g2[{x1_,x2_},{u1_,u2_}]:=u2;
dynamicMax[objective,{g1,g2},{x1,x2},{u1,u2},{1.,1.},{x10,x20},t,0,
  T,0.05,{{a,-1},{T,10},{x10,1},{x20,2}},2]

*)

BeginPackage["approximate`dynamicMax`",{"approximate`Solve`","approximate
`BVPSolve`"}]

dynamicMax::usage="dynamicMax[f,g,x,u,u0,x0,t,t0,tT,r,param,n] solves a
dynamic maximization problem.  The objective function is given by the
integral of f[x,u] Exp[-r t] from t0 to tT, and the constraints are
given by dx/dt = g[x,u], where x is a state vector, u is an input
vector, u0 is a vector of estimated values for u, x0 is an initial state
vector, r is a discount rate, and param is a list of parameters and
their estimated values.  The solution is approximated by a polynomial of
degree n."

debug::usage="debug is an option; debug->On for debugging."

Options[approximateSolve]={debug->Off};

(* error messages *)
dynamicMax::objf="The object function `1` must not be a list."

dynamicMax::gxneq="The numbers of `1`, `2` and `3` are not equal."

dynamicMax::nomax="The control variables that maximize the Hamiltonian
cannot be obtained."

dynamicMax::syntxerr="Syntax error!
Syntax: dynamicMax[f,g,x,u,u0,x0,t,t0,tT,r,param,n]"

dynamicMax::degint="Degree `1` must be an integer and greater than 0."

Begin["`Private`"]

(* Extract estimate values of varList from parameter list, prmlist *)
initialValues[prmList_,varList_]:= varList/.Map[Apply[Rule,#]&,prmList];

(* multiple inputs and multiple states *)
dynamicMax[f_,g_List,x_List,u_List,u0_List,x0_List,t_,t0_,tT_,r_,param_Li
st,degree_,opt___]:=
  Module[{phiv,xtv,utv,H,tt,uu,s,s1,i,xdphid,paramAppend,debugging},

If[(Head[degree]=!=Integer)||(degree<=0),Message[dynamicMax::degint,degre
e];Return[]];
    If[Head[f]===List,Message[dynamicMax::objf,f];Return[]];

If[Not[Length[g]===Length[x]===Length[x0]],Message[dynamicMax::gxneq,g,x,
```

```
x0];Return[]];
    debugging=debug/.{opt}/.Options[approximateSolve];
    phiv=Table[ToExpression["phi"<>ToString[i]],{i,Length[g]}];
    xtv=Table[ToExpression["xt"<>ToString[i]],{i,Length[x]}];
    utv=Table[ToExpression["ut"<>ToString[i]],{i,Length[u]}];
    H=-f[xtv,utv]+phiv.Map[#[xtv,utv]&,g]; (* Hamiltonian *)
    If[(debugging===On) || (debugging===All),
    Print["Hamiltonian:
",H/.Join[{tt->t},Thread[xtv->x],Thread[utv->u]]] ];
    (* maximize H with respect to control variables *)
    paramAppend=Join[param,Transpose[{phiv,Table[1.,{Length[g]}]}]];
    If[debugging===All,

s=approximateSolve[Map[D[H,#]&,utv],utv,u0,paramAppend,degree,debug->All]
,
      s=approximateSolve[Map[D[H,#]&,utv],utv,u0,paramAppend,degree]
    ];
    If[(debugging===On) || (debugging===All),
      Print["Optimal input vector:
",s/.Join[{tt->t},Thread[xtv->x],Thread[utv->u]]]];
    If[(Head[s]=!=List || s=={}),Message[dynamicMax::nomax];Return[]];
    (* the derivates of the state and costate variables with respect to
time *)
    xdphid=Join[Map[(#[xtv,utv])&,g],r phiv+Map[(-D[H,#])&,xtv]]/.s;
    If[debugging===All,

s1=approximateBVPSolve[xdphid,{xtv,phiv},{x0,Table[0,{Length[g]}]},param,
tt,t0,tT,degree,debug->All],

s1=approximateBVPSolve[xdphid,{xtv,phiv},{x0,Table[0,{Length[g]}]},param,
tt,t0,tT,degree]
    ];
    If[(debugging===On) || (debugging===All),
      Print["Solution of the boundary value problem:
",s1/.Join[{tt->t},Thread[xtv->x],Thread[utv->u]]]  ];
    Join[
      Select[s1,(MemberQ[xtv,#[[1]]])&],
      Thread[utv->((utv/.s)/.s1)]
    ]/.Join[{tt->t},Thread[xtv->x],Thread[utv->u]]
  ]

(* unListQ checks if the Head of g is not List *)
unListQ[g_]:=UnsameQ[Head[g],List];

(* single input and single state *)
dynamicMax[f_,g_?unListQ,x_?unListQ,u_?unListQ,u0_?unListQ,x0_?unListQ,t_
,t0_,tT_,r_,param_List,degree_,opt___]:=
  dynamicMax[f,{g},{x},{u},{u0},{x0},t,t0,tT,r,param,degree,opt];

(* multiple inputs and single state *)
dynamicMax[f_,g_?unListQ,x_?unListQ,u_List,u0_List,x0_?unListQ,t_,t0_,tT_
,r_,param_List,degree_,opt___]:=
  dynamicMax[f,{g},{x},u,u0,{x0},t,t0,tT,r,param,degree,opt];

(* single inputs and multiple state *)
dynamicMax[f_,g_List,x_List,u_?unListQ,u0_?unListQ,x0_List,t_,t0_,tT_,r_,
param_List,degree_,opt___]:=
  dynamicMax[f,g,x,{u},{u0},x0,t,t0,tT,r,param,degree,opt];

(* when no parameter list is given *)
```

```
dynamicMax[f_,g_,x_,u_,u0_,x0_,t_,t0_,tT_,r_,degree_,opt___]:=
  dynamicMax[f,g,x,u,u0,x0,t,t0,tT,r,{},degree,opt]


(* syntax error, otherwise *)
dynamicMax[___]:=
  (Message[dynamicMax::syntxerr];Return[];)

End[]

EndPackage[]
```

■ approximateOpenloopNash

```
(* :Name: openloopNash *)

(* :Version: 0.5 *)

(* :Title: approximate open-loop Nash equilibria solution of dynamic
games *)

(* :Author: Yuji Itaya *)

(* :Context: approximate`openloopNash` *)

(* :Requirements: No special system requirements *)

(* :History: Version 0.5 June 10, 1999 *)

(* :Warning:
*)

(* :Copyright: (c) 1999 Yuji Itaya *)

BeginPackage[
  "approximate`openloopNash`",{"approximate`BVPSolve`"}]

approximateOpenloopNash::usage=
"approximateOpenloopNash[f1,f2,r,g,x,{u1,u2},{u10,u20},x0,t,t0,tT,param,n
] obtains a Nash equilibria solution of differential games. The
objective functions of each player are given by the integral of f1[x,u]
Exp[-r t] and f2[x,u] Exp[-r t] from t0 to tT, and the constraints are
given by dx/dt = g[x,u], where x is a state vector, u1 is a strategy for
player 1, u2 is a strategy for player 2, x0 is an initial state, and
param is a list of parameters and their estimated values.  The solution
is approximated by a polynomial of degree n."

(* debug::usage="debug is an option; debug->On for debugging."

Options[approximateOpenloopNash]={debug->Off}; *)

(* error messages *)
approximateOpenloopNash::error="Error: `1`"

Begin["`Private`"]

approximateOpenloopNash[f1_,f2_,r_,g_,x_,{u1_,u2_},{u10_,u20_},x0_,t_,t0_
,tT_,
    param_,degree_,opt___]:=
```

```
    Module[{debugging,H1,H2,phi1,phi2,xsol,deq,bvpsol},
        H1=-f1+phi1 g;
        H2=-f2+phi2 g;
        xsol=Solve[{D[H1,u1]==0,D[H2,u2]==0},{u1,u2}];
        deq={g,r phi1-D[H1,x],r phi2-D[H2,x]}/.xsol[[1]];

bvpsol=approximateBVPSolve[deq,{{x},{phi1,phi2}},{{x0},{0,0}},param,t,t0,
tT,degree];
        Flatten[{bvpsol[[1]],xsol[[1]]/.bvpsol}]
    ];

End[]

EndPackage[]
```

References

[1] Y. Itaya: Analyzing and simulating economic systems, with applications to urban economics, pp. 255-262, V. Keränen and P. Mitic ed., **Innovation in Mathematics**, Computatonal Mechanics Publications (1997).
[2] C. Freshtman and S. Nitzan: Dynamic voluntary provision of public goods, pp. 1057-1067, **European Economic Review** 35 (1991).
[3] A. Shibata and Y. Takeda: Differential games in Economics, pp. 1-22, **Journal of Economics-- *The Keizaigaku Zasshi*** 98 (1997) (in Japanese).
[4] M. D. Intriligator: **Mathematical Optimization and Economic Theory**, Prentice-Hall (1971).
[5] A. C. Chiang: **Elements of Dynamic Optimization**, McGraw-Hill (1992).
[6] T. Basar, G. J. Olsder: **Dynamic Noncooperative Game Theory**, Academic Press (1982).
[7] M. L. Petit: **Control Theory and Dynamic Games in Economic Policy Analysis**, Cambridge University Press (1990).
[8] M. I. Kamien and N. L. Schwartz: **Dynamic Optimization** 2nd ed., North-Holland (1991).