Direct methods - Bocop

P. Martinon
Inria Saclay, CMAP Ecole Polytechnique

03/09/2018
CIMICON - Toulouse
Optimal Control Problem: formulation

Variables:
- $y(\cdot)$ state (describe the system, subject to ODE)
- $u(\cdot)$ control (action on the system)
- $z(\cdot)$ algebraic variables (similar to controls)
- $\pi$ scalar parameters

\[
\begin{align*}
\text{Min } J(\cdot) &= \int_0^T l(t, y(t), u(t))\,dt \\
&\quad + g_0(T, y(0), y(T)) \\
\dot{y}(t) &= f(t, y(t), u(t)) \\
\Phi_l &\leq \Phi(y(0), y(T)) \leq \Phi_u \\
g_l &\leq g(t, y(t), u(t)) \leq g_u
\end{align*}
\]

(OCP)
Optimal Control Problem: functions

- **Objective:** $\text{Min } J(\cdot)$
  Can include running cost and/or final cost (Lagrange, Mayer, Bolza forms).

- **Dynamics:** $\dot{y} = f(t, y, u)$
  ODE for state dynamics, may include *delay* terms depending on past state/control.

- **Boundary conditions:** $\Phi_l \leq \Phi(y(0), y(T)) \leq \Phi_u$
  Constraints on initial and final states, including periodicity.
  Also scalar constraints involving parameters.

- **Path constraints:** $g_l \leq g(t, y, u) \leq g_u$
  Running constraints on state and/or control at all times:
  basic bounds, pure constraints, mixed constraints.
Optimal Control Problem: methods

**HJB (cf Dynamic Programing) (global)**
Bellman’s optimality principle: value function $V$ solution of a PDE. From $V$ reconstruct optimal trajectories. Full discretization: $t, y, u$
+ global, switches, stochastic, feedback control
- costly (state dimension), free $T$, periodicity, state constraints

**Indirect methods (local)**
Pontryagin’s Maximum Principle: control elimination (maximize Hamiltonian). Boundary Value Problem; shooting or collocation
+ fast, accurate
- sensitive to adjoint initialization, singular / constrained arcs

**Direct methods (direct transcription) (local)**
Time discretization: Nonlinear Programming; interior point or SQP
+ handy for singular/constrained arcs, parameters, model Id
- accuracy can be limited by discretization / structure
Direct method: time discretization

Time discretization: \((t_i)_{i=0..N}\), usually uniform with step \(h\)
New state and control variables: \((y_i, u_i) = X\)
OCP is reformulated in terms of unknown \(X\)

- objective: final cost \(g_0(t_N, y_0, y_N)\), running cost as sum of terms \(l(t_i, y_i, u_i)\)
- boundary conditions: \(\Phi(X_0, X_N)\)
- path constraints: \(g(t_i, y_i, u_i), i = 0 \ldots N\)
- dynamics (ex:Euler): \(y_{i+1} = y_i + h f(t_i, y_i, u_i), i = 0 \ldots N - 1\)

This gives a discretized problem (NLP)

\[
\begin{align*}
\min \ F(X) \\
C_{LB} \leq C(X) \leq C_{UB}
\end{align*}
\]

\(\rightarrow\) solve (NLP) as approximation of (OCP)
Note: KKT for NLP tends to PMP for OCP when \(h \rightarrow 0\)
Direct method: Runge Kutta formulas

General RK formula: \( s \) stages and coefficients \( a_{ij}, b_i, c_i \).

Conditions: \( \sum_{i=1}^{s} b_i = 1 \) and \( c_i = \sum_{j=1}^{i-1} a_{ij} \)

Butcher form

\[
\begin{array}{cccc}
  c_1 & a_{11} & a_{12} & \ldots & a_{1s} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  c_s & a_{s1} & a_{s2} & \ldots & a_{ss} \\
\end{array}
\begin{array}{c}
  b_1 \\
  b_2 \\
  \vdots \\
  b_s \\
\end{array}
\]

The formula for one step at time \( t_\ell \) is

\[
y(t_{\ell+1}) = y(t_\ell) + h \sum_{i=1}^{s} b_i k_i^\ell, \quad k_i^\ell = f(t_\ell + c_i h, y(t_\ell) + h \sum_{j=1}^{s} a_{ij} k_j^\ell)
\]

The \( k_i^\ell \) are either part of \( X \), or computed (harder for implicit case).

Remark: \( A \) is strictly lower triangular for explicit methods.

Example: explicit/implicit euler, RK4, Gauss, Lobatto...

Properties: RK formulas can be symmetric (time reversible), symplectic (invariant conservation, such as energy)
Direct method: state and control variables

**State variables**: 2 approaches
- 'Sequential': set $X = (u_i)$ only, recompute state $y_i$ by RK formula
  Smaller problem, but requires solving for implicit RK methods...
- 'Simultaneous': set $X = (y_i, k_j^i, u_i)$, enforce RK formulas as equality constraints at each time step and stages.
  Easy implementation of implicit RK formulas (better properties).
  Simultaneous approaches seem more effective overall.

**Control variables**
In RK formula dynamics $f$ is evaluated at time stages $t_\ell + c_i h$.
Control variables are actually at stages, ie $u_i^j$ (same as $k_j^i$).
Consequence: path constraints $g$ at stages too, with state
$y(t_\ell) + h \sum_{j=1}^{s} a_{ij} k_j^\ell$

Note: $C(X)$ structure gives strongly sparse derivatives.
Direct method: solving the NLP

2 popular families of methods: SQP and IP.

**Sequential Quadratic Programming (SQP)**
At each iterate $X^k$:
- linearization of constraints, quadratic objective with Lagrangian term
- solve the resulting QP problem for $X^{k+1}$

**Interior point (IP)**
Constrained problem $(P) \ min F(X), C(X) \geq 0$
Penalize constraints with barrier functions: unconstrained problem $(P_\mu) \ min F(X) - \mu \sum_{i=1}^{m} \log C_i(X)$
When $\mu \to 0$, solution of $(P_\mu) \to$ solution of $(P)$. 
Direct method: derivatives

Both SQP and IP solvers typically require
- gradient of the objective $F(Z)$
- hessian of the lagrangian $L(Z, \lambda) = F(Z) + \lambda C(Z)$

- **Finite differences**
  Allows black boxes, but requires step choice

- **Automatic differentiation**
  - source generation (**TAPENADE**)
    Generated code can be modified. Or may require to be fixed ...
  - operator overloading (**ADOL-C, CPPAD**)
    No additional code, but some limitations (no if...then...)

NB. Take advantage of hessian sparsity for numerical efficiceny!
In practice: tips and tricks

- running cost
- free final time
- constraints: bounds, mixed/pure, periodicity
- control structure: bang, singular, constrained arcs
- scalar parameters and model identification
- delay problems
- time discretization tradeoffs
- definition / differentiability domains
- NLP starting guess
- OS, compiler and library quirks
Running cost
Reformulate as final cost with an additional state variable
\[ \text{Min } x_{obj}(T) \text{ with } \dot{x}_{obj} = 1 \text{ and } x_{obj}(0) = 0 \]

Free final time
Renormalize time to \([0, 1]\) and multiply dynamics by \(T\).
\(T\) is an additional parameter of \(X\) to be optimized.
NB. Remember to bound \(T \geq \epsilon > 0\)! Upper bound \(T \leq M\) may be useful for non minimum-time problems.

Constraints
- Periodicity: boundary conditions on \((X_0, X_N)\), part of \(C(X)\)
- Bounds for state, control, parameters: bounds for components \(X_i\)
- Mixed constraints \(g(x, u)\): transcripted automatically in \(C(X)\)
- Pure state constraints \(h(x)\): idem. No additional work.
Control structure:

No structure assumptions or junction conditions

Control variables are components of \( X \) just like state variables. **Bang arcs**: appear naturally in solution (up to discretization)
**Singular arcs**: likewise (junctions may exhibit glitch points)
**Constrained arcs**: likewise (possible oscillations on the boundary)

See Fuller, Goddard, regulator examples
Optimize scalar parameters
Scalar parameters are simply added to the unknown $X$.
Example: free final time $T$, design parameters ...
No need to define a constant state variable.
Parameters can have bounds and more general constraints (defined as boundary conditions).

Model identification
System model may include parameters which values are not known.
Provided a set of experimental measures for a given control.
Solve OCP with fixed control and free parameters, minimizing the error between trajectory and measures (e.g. mean square error).
See example Jackson

Algebraic variables
Additional unknown $z(t)$, may appear in dynamics / constraints.
In practice, behaves like a control.
Time discretization
Direct methods solve a **discretized** approximation of OCP.
- higher order gives more accuracy (if sufficient smoothness !)
Drawback: larger problem size and cpu time; risk of numerical instabilities.
- smaller time step increases accuracy, but also size and cpu time.
Advice: start with low order (implicit midpoint), play a bit with time step.

Delay problems
Dynamics and/or constraints depend on 'past' states or controls.
\[ \dot{y}(t) = f(t, y(t), u(t), y(t - \tau), u(t - \theta)) \]
Direct (simultaneous) method: \( X \) contains states \( y_i \) for all times \( t_i \).
'Past' values can be approximated by interpolation.
Technical limitation: fixed final time.
Remark: 'future' values could be interpolated as well...
Build or execution errors
Try to understand compiler errors and warnings. Use debug build for more information. Careful with indexes, function definition / differentiability domains. Do not hesitate to reformulate the problem.

Platform quirks
Depending on OS, compiler, libraries: different results? *same source code → different executable code.* A properly formulated problem should give the same solution regardless of the platform: differences indicate a non robust code!
NLP convergence issues
Check functions definition and differentiability domains.
Typical traps: $1/x$, $\sqrt{x}$, $Ln \ x$, $|x|$, $min(x, y)$...
Visualize arguments, add constraints if necessary.
Changing the starting guess may help.

NLP starting guess
NLP solver requires a starting guess $X^0$ (iterative algorithm).
$X^0$ needs **not** to be admissible for constraint $C$.
In particular $X^0$ does not have to satisfy the ODE!
That said, try to avoid completely unrealistic values...
Bocop: toolbox for optimal control problems

C/C++, linux/win/mac, GUI, EPL license

- Deterministic problems, model identification, delay systems
- Direct transcription method
  - Generalized Runge Kutta
  - interior point solver Ipopt
  - sparse automatic differentiation
- Robust and handy, but local method
Second package in the Bocop toolbox

- Stochastic problems, switched systems
- Global method, feedback control
- HJB method with Semi-Lagrangian (cf Dyn. Prog.)
- Limitations on state dimension
Bocop: problem files

A problem is defined in Bocop by

4 functions (C/C++)
- criterion.tpp: objective $J$ (as final cost)
- dynamics.tpp: system dynamics $f$
- boundarycond.tpp: boundary conditions $\phi$
- pathcond.tpp: path constraints $g$

3 definition files (text)
- problem.def: general definition and settings
- problem.bounds: bounds for $Z$ and $C(Z)$
- problem.constants: constant parameters
- BOCOP: displays general messages (build, NLP iterations, ...)
- Definition: general definition and settings, bounds, constants
- Starting point: defines initial guess for NLP solver
- Optimization: settings for OCP and NLP
- Visualization: optimal solution (variables, constraints, multipliers)
Bocop: Goddard problem

1D rocket ascent with maximal final altitude

\[
\begin{aligned}
\text{max } & m(T) \\
\dot{r} &= v \\
\dot{v} &= -\frac{1}{r^2} + \frac{1}{m(T_{\text{max}})}(T_{\text{max}}u - D(r, v)) \\
\dot{m} &= -bu \\
u(\cdot) &\in [0, 1] \\
r(0) &= 1, \ v(0) = 0, m(0) = 1, \\
r(T) &= 1.01 \\
D(r(\cdot), v(\cdot)) &\leq C \\
T &\text{ free}
\end{aligned}
\]

Drag: \( D(r, v) = Av^2\rho(r) \) with \( \rho(r) = e^{-k(r-r_0)} \).

Parameters \( b = 7, T_{\text{max}} = 3.5, A = 310, k = 500 \) and \( r_0 = 1 \).

Typical solution is bang/singular with structure \( B^+SB^- \)
// Function for the dynamics of the problem
// dy/dt = dynamics(y,u,z,p)

#include "header_dynamics"
{
    // DYNAMICS FOR GODDARD PROBLEM
    // dr/dt = v
    // dv/dt = (Thrust(u) - Drag(r,v)) / m - grav(r)
    // dm/dt = -b*|u|

    double Tmax = constants[0];
    double A = constants[1];
    double k = constants[2];
    double r0 = constants[3];
    double b = constants[4];

    Tdouble r = state[0];
    Tdouble v = state[1];
    Tdouble m = state[2];

    state_dynamics[0] = v;
    state_dynamics[1] = (thrust(control[0],Tmax) - drag(r,v,A,k,r0)) / m - grav(r);
    state_dynamics[2] = - b * control[0];
}
// Function for the criterion of the problem
// Min criterion(z)

// Tdouble variables correspond to values that can change during optimization:
// states, controls, algebraic variables and optimization parameters.
// Values that remain constant during optimization use standard types (double,\
// int, ...).

#include "header_criterion"
{
    // CRITERION FOR GODDARD PROBLEM
    // MAXIMIZE FINAL MASS
    criterion = -final_state[2];
}
// Function for the initial and final conditions of the problem
// lb <= Phi(t0, y(t0), tf, y(tf), p) <= ub

// Tdouble variables correspond to values that can change during optimization:
// states, controls, algebraic variables and optimization parameters.
// Values that remain constant during optimization use standard types (double, int, ...).

#include "header_boundarycond"
{

    // INITIAL CONDITIONS FOR GODDARD PROBLEM
    // r0 = 1  v0 = 0  m0 = 1
    // MODELED AS 1 <= r0 <= 1, etc
    boundary_conditions[0] = initial_state[0];
    boundary_conditions[1] = initial_state[1];
    boundary_conditions[2] = initial_state[2];

    // FINAL CONDITIONS FOR GODDARD PROBLEM
    // rf >= 1.01 MODELED AS 1.01 <= rf
    boundary_conditions[3] = final_state[0];

}
Listing 4: goddard/pathcond.tpp

```cpp
// Function for the path constraints of the problem
// a <= g(t,y,u,z,p) <= b

// Tdouble variables correspond to values that can change during optimization:
// states, controls, algebraic variables and optimization parameters.
// Values that remain constant during optimization use standard types (double, int, ...).

#include "header_pathcond"
{
    // CONSTRAINT ON MAX DRAG FOR GODDARD PROBLEM
    // Drag <= C ie Drag - C <= 0

    double A = constants[1];
    double k = constants[2];
    double r0 = constants[3];
    double C = constants[5];

    Tdouble r = state[0];
    Tdouble v = state[1];

    path_constraints[0] = drag(r, v, A, k, r0) - C;
}
```

Remember to set bounds values!
Auxiliary functions:
- declared in dependencies.hpp
- defined in dependencies.tpp if using Tdouble variables
- defined in dependencies.cpp if not using Tdouble variables

Listing 5: goddard/dependencies.hpp

```
// function for the goddard drag
template <class Tdouble> Tdouble drag(const Tdouble, const Tdouble, const double, const double, const double);

// function for gravity
template <class Tdouble> Tdouble grav(const Tdouble);

// function for goddard thrust
template <class Tdouble> Tdouble thrust(const Tdouble, const double);
```
Listing 6: goddard/dependencies.tpp

```cpp
#include <cmath>

// FUNCTION FOR GODDARD DRAG
// \( \text{drag} = 310 \ v^2 \ exp \left(-500(r-1)\right) \)
template <class Tdouble> Tdouble drag(Tdouble r, Tdouble v, double A, double k, double r0)
{
    Tdouble drag = A * v * v * exp(-k*(fabs(r)-r0));
    return drag;
}

// FUNCTION FOR GRAVITY
// \( g = \frac{1}{r^2} \)
template <class Tdouble> Tdouble grav(Tdouble r)
{
    Tdouble grav = 1e0 / r / r;
    return grav;
}

// FUNCTION FOR THRUST (GODDARD)
// \( T = u \times T_{\text{max}} \)
template <class Tdouble> Tdouble thrust(Tdouble u, double Tmax)
{
    Tdouble thrust = u * Tmax;
    return thrust;
}
```
Bocop: Goddard problem (definition)

General definition (dimension, time discretization, ...).
Set lower and upper bounds for $Z$ and $C(Z)$
Bocop: Goddard problem (initialization)

Starting guess for NLP solver (not necessarily feasible)
Bocop: Goddard problem (optimization)

Settings for optimization (initial guess, NLP solver, model identification...)

Ipopt options:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_iter</td>
<td>1000</td>
</tr>
<tr>
<td>print_level</td>
<td>5</td>
</tr>
<tr>
<td>tol</td>
<td>1.0e-14</td>
</tr>
<tr>
<td>output_file</td>
<td>result.out</td>
</tr>
<tr>
<td>mu_strategy</td>
<td>adaptive</td>
</tr>
</tbody>
</table>

Parameter Identification

- Method: Least Squares
- Number of observation files
- Separator: Choose one observation file
## Bocop: Goddard problem (execution)

**Iterations from NLP solver (Ipopt)**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objective (scaled)</th>
<th>Objective (unscaled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>20</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>21</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>22</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>23</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>24</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>25</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>26</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
<tr>
<td>27</td>
<td>6.3413096e-01</td>
<td>6.3413096e-01</td>
</tr>
</tbody>
</table>

**Number of Iterations:** 27

**Objective:**
- Scaled: 6.3413096e-01
- Unscaled: 6.3413096e-01

**Infeasibility:**
- Scaled: 7.247559047530219e-13
- Unscaled: 7.247559047530219e-13

**Constraint Violation:**
- Scaled: 3.031348342169517e-14
- Unscaled: 5.662137425583984e-14

**Complementarity:**
- Scaled: 6.085134924984513e-15
- Unscaled: 6.085134924984513e-15

**Overall NLP Error:**
- Scaled: 7.247559047530219e-13
- Unscaled: 7.247559047530219e-13

**Number of Evaluations:**
- Objective: 62
- Objective Gradient: 28
- Equality Constraint: 62
- Inequality Constraint: 63
- Equality Constraint Jacobian: 28
- Inequality Constraint Jacobian: 28
- Lagrangian Hessians: 27

**CPU Times:**
- Ipopt (w/o Jacobian evaluations): 0.176
- Ipopt (w/ Jacobian evaluations): 0.120

**EXIT:** Solved to Acceptable Level.

<table>
<thead>
<tr>
<th>Objective Value</th>
<th>Time taken</th>
<th>Ipopt solver returns 1: solved to acceptable level</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6.341313e-01</td>
<td>0.13s</td>
<td></td>
</tr>
</tbody>
</table>
Bocop: Goddard problem (visualization)

Visualize optimal trajectory (state, control, constraints, multipliers)
Links between OCP methods: Goddard problem

Solution for local and global methods.
Links between OCP methods: principle

All 3 classes of methods should give the same (primal) solution. Additionally,

\[ \text{direct method multiplier for dynamics} \]
\[ \quad = \]
\[ \text{costate for PMP in indirect method} \]
\[ \quad = \]
\[ \text{value function gradient in HJB} \]

Ex: goddard (singular arc, unconstrained)

<table>
<thead>
<tr>
<th>Method</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct $\lambda$</td>
<td>$(-50.9150, -1.8569, -0.7050)$</td>
</tr>
<tr>
<td>Indirect $p$</td>
<td>$(-50.9281, -1.9412, -0.6933)$</td>
</tr>
</tbody>
</table>
A few useful references

- *Practical methods for optimal control using nonlinear programming*, Betts.
- *Nonlinear Programming*, Biegler.