Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Institute for Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme und Regelungstechnik

User FAQ – DPM Function

Wednesday, June 19, 2013

Who should use the dpm function?

You should have profound skills in Matlab programming, Numerics, Systems Theory and you should be familiar with the concept of Dynamic Programming [1]. This tool intended to be used by *skilled* users only. The function does not contain sophisticated error handling. Before starting to use the dpm function you should have read the corresponding publication [2] and understand the examples provided on our webpage. **If you encounter any problems with the dpm function, please consult this document *before* contacting IDSC.** Most of the problems that occur can be easily solved, and it is likely that other users have had the same problem before. We will keep this User-FAQ as well as the programming examples on the webpage up to date, as we find more problems and implementation related issues.

What Matlab Version and Toolboxes do I need?

The newest posted version of the dpm function (v1.1.0) work for Matlab 2007a and later. There are no special toolboxes required.

What kind of problem can be solved with the dpm function?

The dpm function is intended for solving optimal control problems (OCP), but any other problem that can be reformulated as a dynamic programming problem should work as well.

OCPs with *fixed final time and free or fixed final state* work well for both *continuous and/or discrete states and inputs*. If the final state is free you are likely to use a *final cost term* (see below), if your final state is free and there is only one state then you are likely to use the *boundary line method* (see below).

If the final time is free you need to introduce time as an additional state and use a final cost term that penalizes the use of time.

I want to add a final cost term to the cost function

If you want to add a *final cost term* to the cost function, you need to use the `options.gN{1}` structure. The dimensions of the `gN{1}` structure need to be `grd.Nx{1} x...x grd.Nx{n}`. In this case the final constraint is typically relaxed to a larger interval or even the whole state-space. Set `grd.XN{.}` accordingly.

I want to use a boundary method

Just set `options.BoundaryMethod` to `Line` for problems with one state variable, or to `LevelSet` for problems with an arbitrary number of states. The boundary method is to be used for *fixed final state* problems only. Specify the final state constraint in `grd.XN{.}`. The benefit of using this method is that interpolation errors due to a high gradient in the cost at the border of the feasible

region can be eliminated. The result is higher accuracy in both total cost and deviation of the final state. Furthermore in most cases the discretization can be chosen much coarser and therefore computation time can be saved. Read [3] and [4] for more information.

How to choose `MyInf`?

`Options.MyInf` is used instead of `Inf` for penalizing infeasible states. Thus `MyInf` needs to be larger than the maximum value in the cost-to-go that can ever occur. If you deal with a fixed final state problem but do not use the boundary line Method, it is beneficial to choose the value of `MyInf` as low as possible but still larger than the worst-case cost. In many cases the worst-case cost can be approximated quite easily or even calculated analytically. At least it is very easy to check the maximum of the cost-to-go a posteriori. This way, interpolation errors due to a high gradient in the cost-to-go, can be eliminated to some extent. If your problem contains only one state, consider using the boundary line method.

How many state variables can the dpm function handle?

Up to 5, in the current version. But since you know about dynamic programming, you are aware that the computation time increases exponentially with the number of states, so…

Depending on your problem you might be able to find a different formulation, such that you can reduce the number of states. If you are able to do so you can save an exponential amount of time, so it is really worth a try.

How fine discretization should I use?

When solving discrete problems the answer to this question will be trivial. If you are concerned about continuous problems you will have to discretize time, control and state space. The solution will converge to the “true” global optimal solution as the discretization is increased. Generally there is no rule to find out how fine the discretization has to be. It is up to the user to trade off computational time against quality of the solution. Best way to do so is to start with a very fine grid representing the “optimal” solution. Then compare solutions found with a coarser discretization to the “optimal” one.

I want to solve an optimal control problem with free final time

This class of problems is not very easy to solve with dynamic programming if not intractable, since the length of the problem is not known a priori. There is a general way around this problem, but you might be able to find a better way around this problem for your specific problem. The general way is the following: You need to introduce time as an additional state to the system and use a final cost term (`options.gN{1}`) that penalizes the use of time. You might also find a different formulation of the problem such that you can reduce the number of states. If you are able to do so you can save an exponential amount of time, so it is really worth a try.

I get an error that I should check the model function

The model should be formulated such that all vectors have the right dimensions; this is the most common problem for most users (use element-wise operations like "`.^`" in the model file). Another often-occurring problem is that your model code should not produce `NaN` values. Instead of working with `NaN`, use the infeasibility matrix `I`.

The dpm tells me to make sure the problem is feasible

If you get the error message

```
DPM:Forward simulation error      Make sure the problem is feasible.
```

this means that either all input-state combinations are infeasible at a certain time-step, or that the cost-to-go function has a value larger than `options.MyInf`. First try to increase the value of `options.MyInf` to make sure the error was not the latter one. If you still get the same error at the same time-step, a good practice to track it down is to insert the following command at the end of your model:

```
if numel(find(I==0))==0
    keyboard
end
```

This will stop your code at the time-step where the error takes place. You then have the possibility to see where the infeasible input-state combinations originate from.

If you are 100% sure the problem is feasible, check whether your discretization is fine enough.

I want to add more functionality to the code, am I allowed to do that?

Sure, however, any commercialization of this code is neither intended nor allowed.

What changed since the last version?

- We included a `Verbose`-option that prints to the command window what the dpm function is doing at the moment. Note that `Verbose` is much faster than `Waitbar`.
- We removed the use of `roundn.n`, which is part of a Matlab toolbox.
- We included the LevelSet Method [4].
- We included some consistency checks to reduce errors.
- We made the new version backwards compatible to previous versions.
- We changed the `options` structure to include the following fields
    - `Waitbar` (on/**off**) – Turns the waitbar on/off.
    - `Verbose` (**on**/off) – Turns verbose on/off
    - `Warnings` (**on**/off) – Turns the warnings on/off
    - `SaveMap` (**on**/off) – Specifies whether cost-to-go is saved (requires memory)

  - `MyInf` (**1e5**) – A value, larger than the highest cost-to-go that can occur in the problem
  - `Minimize` (**1**/0) – Minimizes/Maximizes the Cost-Function
  - `InputType` (**c**/d) – Continuous/Discrete Inputs, string of length(grd.U)
  - `gN` – final cost term
    `size(OPTIONS.gN{1})=[GRD.Nx{1},GRD.Nx{2}...GRD.Nx{.}]`
  - `BoundaryMethod` (none/Line/LevelSet) – Boundary Method
    - The following options apply only if `BoundaryMethod = Line`
    - `FixedGrid` (**1**/0) – Fix/Adjust grid in backward calculation
    - `Iter` (**10**) – maximum number of iterations when inverting model
    - `Tol` (**1e-8**) – minimum tolerance when inverting model
- (Format: `Name` (**default**/alternative) – description)

References

[1] D. P. Bertsekas, Dynamic Programming and Optimal Control, Vol. I and II, ISBN 1-886529-08-6, 2005

[2] O. Sundström and L. Guzzella, A Generic Dynamic Programming Matlab Function, in Proceedings of the 18th IEEE International Conference on Control Applications, pages 1625-1630, Saint Petersburg, Russia, 2009

[3] O. Sundström, D. Ambühl and L. Guzzella, On Implementation of Dynamic Programming for Optimal Control Problems with Final State Constraints, in Proceedings of Les Rencontres Scientifiques de l'IFP: Advances in Hybrid Powertrains, 2008

[4] P. Elbert, S. Ebbesen and L. Guzzella, Implementation of Dynamic Programming for n-Dimensional Optimal Control Problems with Final State Constraints, in IEEE Transactions on Control Systems Technology, 2012

License