Dynamical Movement Primitives Tutorial

Freek Stulp

March 5, 2014

This document was generated by doxygen, so the formatting is not always optimized for pdflatex.

Contents

Dynamical Systems

Introduction

Let a *state* be a vector of real numbers. A dynamical system consists of such a state and a rule that describes how this state will change over time; it describes what future state follows from the current state. A typical example is radioactive decay, where the state x is the number of atoms, and the rate of decay is $\frac{dx}{dt}$ proportional to x: $\frac{dx}{dt} = -\alpha x$. Here, α is the 'decay constant' and $\dot{x}$ is a shorthand for $\frac{dx}{dt}$. Such an evolution rule describes an implicit relation between the current state $x(t)$ and the state a short time in the future $x(t + dt)$.

If we know the initial state of a dynamical system, e.g. $x_0 \equiv x(0) = 4$, we may compute the evolution of the state over time through *numerical integration*. This means we take the initial state x_0, and iteratively compute subsequent states $x(t + dt)$ by computing the rate of change $\dot{x}$, and integrating this over the small time interval dt. A pseudo-code example is shown below for $x_0 \equiv x(0) = 4$, $dt = 0.01s$ and $\alpha = 6$.

```
alpha=6; // Decay constant
dt=0.01; // Duration of one integration step
x=4.0;   // Initial state
t=0.0;   // Initial time
while (t<1.5) {
  dx = -alpha*x;  // Dynamical system rule
  x = x + dx*dt;  // Project x into the future for a small time step dt (Euler
       integration)
  t = t + dt;     // The future is now!
}
```

This procedure is called "integrating the system", and leads the trajectory plotted below (shown for both $\alpha = 6$ and $\alpha = 3$.

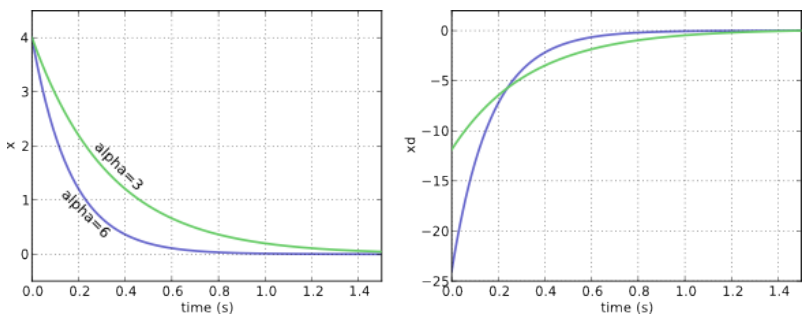

Figure 1: Evolution of the exponential dynamical system.

The evolution of many dynamical systems can also be determined analytically, by explicitly solving

the differential equation. For instance, $N(t) = x_0 e^{-\alpha t}$ is the solution to $\dot{x} = -\alpha x$. Why? Let's plug $x(t) = x_0 e^{-\alpha t}$ into $\frac{dx}{dt} = -\alpha x$, which leads to $\frac{d}{dt}(x_0 e^{-\alpha t}) = -\alpha(x_0 e^{-\alpha t})$. Then derive the left side of the equations, which yields $-\alpha(x_0 e^{-\alpha t}) = -\alpha(x_0 e^{-\alpha t})$. QED. Note that the solution works for arbitrary x_0. It should, because the solution should not depend on the initial state.

Implementation. *In the object-oriented implementation of this module, all dynamical systems inherit from the abstract DynamicalSystem class. The analytical solution of a dynamical system is computed with DynamicalSystem::analyticalSolution, which takes the times* `ts` *at which the solution should be computed, and returns the evolution of the system as* `xs` *and* `xds`*.*

A system's differential equation is implement in the function DynamicalSystem::differentialEquation, which takes the current state `x`*, and computes the rates of change* `xd`*. The functions DynamicalSystem::integrateStart() and DynamicalSystem::integrateStep() are then used to numerically integrate the system as follows (using the example plotted above):*

```
// Make exponential system that decays from 4 to 0 with decay constant 6, and
       tau=1.0
double alpha = 6.0;                // Decay constant
double tau = 1.0;                  // Time constant
VectorXd x_init(1); x_init << 4.0; // Initial state (a 1D vector with the value
       4.0 inside using Eigen comma initializer)
VectorXd x_attr(1); x_attr << 0.0; // Attractor state
DynamicalSystem* dyn_sys = new ExponentialSystem(tau, x_init, x_attr, alpha);

Eigen::VectorXd x, xd;
dyn_sys->integrateStart(x,xd); // Start the integration
double dt = 0.01;
for (double t=0.0; t<1.5; t+=dt)
{
  dyn_sys->integrateStep(dt,x,x,xd);          // Takes current state x,
       integrates system, and writes next state in x, xd
  cout << t << " " << x << " " << xd << endl; // Output current time, state and
       rate of change
}
delete dyn_sys;
```

Remark. Both analyticalSolution and differentialEquation functions above are const, i.e. they do not change the DynamicalSystem itself.

Plotting. *If you save the output of a dynamical in a file with format (where D is the dimensionality of the system, and T is the number of time steps)*

```
x^0_0 x^1_0 .. x^D_0   xd^0_0 xd^1_0 .. xd^D_0   t_0
x^0_1 x^1_1 .. x^D_1   xd^0_1 xd^1_1 .. xd^D_1   t_1
   :     :       :         :      :       :       :
x^0_T x^1_T .. x^D_T   xd^0_T xd^1_T .. xd^D_T   t_T
```

you can plot this output with

```
python dynamicalsystems/plotting/plotDynamicalSystem.py file.txt
```

Demos. *A demonstration of how to initialize and integrate an ExponentialSystem is in demoExponentialSystem.cpp*

A more complete demonstration including all implemented dynamical systems is in demoDynamicalSystems.cpp. If you call the resulting executable with a directory argument, e.g.

```
./demoDynamicalSystems /tmp/demoDynamicalSystems
```

it will save results to file, which you can plot with for instance:

```
python plotDynamicalSystem.py /tmp/demoDynamicalSystems/ExponentialSystem/
      results_rungekutta.txt
python plotDynamicalSystem.py /tmp/demoDynamicalSystems/ExponentialSystem/
      results_euler.txt
```

Different test can be performed with the dynamical system. The test can be chosen by passing further argument, e.g.

```
./demoDynamicalSystems /tmp/demoDynamicalSystems rungekutta euler
```

will integrate the dynamical systems with both the Runge-Kutta and simple Euler method. The available tests are:

- *"rungekutta" - Use 4th-order Runge-Kutta integration (more accurate, but more calls of - DynamicalSystem::differentialEquation)*
- *"euler" - Use simple Euler integration (less accurate, but faster)*
- *"analytical" - Use the analytical solution instead of numerical integration*
- *"tau" - Change tau before doing numerical integration*
- *"attractor" - Change the attractor state during numerical integration*
- *"perturb" - Perturb the state during numerical integration*

To compare for instance the analytical solution with the Runge-Kutta integration in a plot, you can do

```
python plotDynamicalSystemComparison.py /tmp/demoDynamicalSystems/
      ExponentialSystem/results_analytical.txt  /tmp/demoDynamicalSystems/ExponentialSystem/
      results_rungekutta.txt
```

Properties and Features of Linear Dynamical Systems

Convergence towards the Attractor. In the limit of time, the dynamical system for exponential decay will converge to 0 (i.e. $x(\infty) = x_0 e^{-\alpha\infty} = 0$). The value 0 is known as the *attractor* of the system. For simple dynamical systems, it is possible to *proove* that they will converge towards the attractor.

Suppose that the attractor state in our running example is not 0, but 1. In that case, we change the attractor state of the exponential decay to x^g (g=goal) and define the following differential equation:

$$\dot{x} = \quad -\alpha(x - x^g) \quad \text{with attractor } x^g$$

This system will now converge to the attractor state x^g, rather than 0.

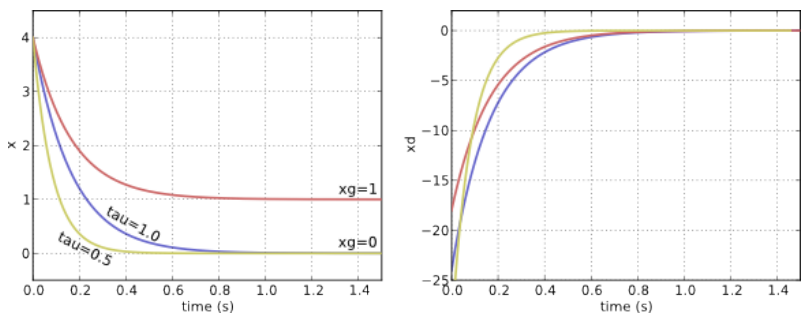

Figure 2: Changing the attractor state or time constant.

Robustness to Perturbations. Another nice feature of dynamical systems is their robustness to perturbations, which means that they will converge towards the attractor even if they are perturbed. The figure below shows how the perturbed system (cyan) converges towards the attractor state just as the unperturbed system (blue) does.

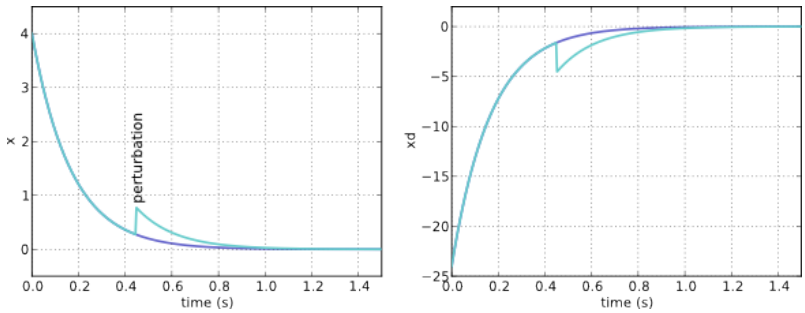

Figure 3: Perturbing the dynamical system.

Changing the speed of convergence: The time constant. The rates of change computed by the differential equation can be increased or decreased (leading to a faster or slower convergence) with a *time constant*, which is usually written as follows:

$$\begin{aligned} \tau\dot{x} = & \quad -\alpha(x - x^g) \\ \dot{x} = & \quad (-\alpha(x - x^g))/\tau \end{aligned}$$

Remark. For an exponential system, decreasing the time constant τ has the same effect as increasing α. For more complex dynamical systems with several parameters, it is useful to have a separate parameter that changes only the speed of convergence, whilst leaving the other parameters the same.

Multi-dimensional states. The state x need not be a scalar, but may be a vector. This then represents a multi-dimensional state, i.e. $\tau\dot{\mathbf{x}} = -\alpha(\mathbf{x} - \mathbf{x}^g)$. In the code, the size of the state vector $dim(\mathbf{x}) \equiv dim(\dot{\mathbf{x}})$ of a dynamical system is returned by the function DynamicalSystem::dim()

Autonomy. Dynamical system that do not depend on time are called *autonomous*. For instance, the formula $\dot{x} = -\alpha x$ does not depend on time, which means the exponential system is autonomous.

Implementation. *The attractor state and time constant of a dynamical system are usually passed to the constructor. They can be changed afterwards with with DynamicalSystem::set_attractor_state and DynamicalSystem::set_tau. Before integration starts, the initial state can be set with DynamicalSystem::set_initial_state. This influences the output of DynamicalSystem::integrateStart, but not DynamicalSystem::integrateStep.*

Further (first order) linear dynamical systems that are implemented in this module is a SigmoidSystem (see `http://en.wikipedia.org/wiki/Exponential_decay` *and* `http://en.wikipedia.org/wiki/Sigmoid_function`*), as well as a dynamical system that has a constant velocity (TimeSystem), so as to mimic the passing of time (time moves at a constant rate per time ;-)*

$$\begin{aligned}
\dot{x} &= -\alpha(x - x^g) && \textit{exponential decay/growth} \\
\dot{x} &= \alpha x(\beta - x) && \textit{sigmoid} \\
\dot{x} &= 1/\tau && \textit{constant velocity (mimics the passage of time)}
\end{aligned}$$

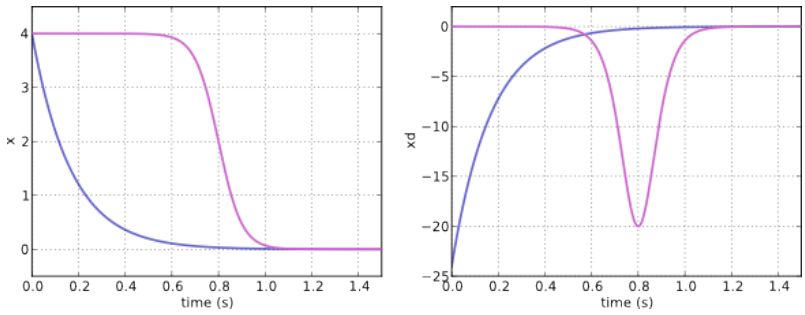

Figure 4: Exponential (blue) and sigmoid (purple) dynamical systems.

Second-Order Systems

The **order** of a dynamical system is the order of the highest derivative in the differential equation. For instance, $\dot{x} = -\alpha x$ is of order 1, because the derivative with the highest order ($\dot{x}$) has order 1. Such a system is known as a first-order system. All systems considered so far have been first-order systems, because the derivative with the highest order, i.e. $\dot{x}$, has always been of order 1.

Spring-Damper Systems. An example of a second order system (which also has terms $\ddot{x}$) is a spring-damper system (see http://en.wikipedia.org/wiki/Damped_spring-mass-_system), where k is the spring constant, c is the damping coefficient, and m is the mass:

$$\begin{aligned}
m\ddot{x} &= -kx - c\dot{x} && \text{spring-damper (2nd order system)} \\
\ddot{x} &= (-kx - c\dot{x})/m
\end{aligned}$$

Critical Damping. A spring-damper system is called critically damped when it converges to the attractor as quickly as possible without overshooting, as the red plot in http://en.-wikipedia.org/wiki/File:Damping_1.svg. This happens when $c = 2\sqrt{mk}$.

Rewriting one 2nd Order Systems as two 1st Order Systems. For implementation purposes, it is more convenient to work only with 1st order systems. Fortunately, we can expand the state x into two components $x = [y\ z]^T$ with $z = \dot{y}$, and rewrite the differential equation as follows:

$$\begin{bmatrix} \dot{z} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} (-ky - cz)/m \\ z \end{bmatrix}$$

With this rewrite, the left term contains only first order derivatives, and the right term does not contain any derivatives. This is thus a first order system. Integrating such an expanded system is done just as one would integrate a dynamical system with a multi-dimensional state:

Implementation. *The constructor DynamicalSystem::DynamicalSystem immediately converts second order systems into first order systems with an expanded state.*

The function DynamicalSystem::dim() returns the size of the entire state vector $x = [y\ z]$*, the function DynamicalSystem::dim_orig() return the size of only the* y *component. The attractor and initial state must always have the size returned by DynamicalSystem::dim_orig().*

Dynamical Movement Primitives

Introduction

The core idea behind dynamical movement primitives (DMPs) is to represent movement primitives as a combination of dynamical systems (please read Dynamical Systems Module, if you haven't already done so). The state variables of the main dynamical system $[\mathbf{y}\dot{\mathbf{y}}\ddot{\mathbf{y}}]$ then represent trajectories for controlling, for instance, the 7 joints of a robot arm, or its 3D end-effector position. The attractor state is the end-point or *goal* of the movement.

The key advantage of DMPs is that they inherit the nice properties from linear dynamical systems (guaranteed convergence towards the attractor, robustness to perturbations, independence of time, etc) whilst allowing arbitrary (smooth) motions to be represented by adding a non-linear forcing term. This forcing term is often learned from demonstration, and subsequently improved through reinforcement learning.

DMPs were introduced in [Ijspeert et al., 2002], but in this section we follow largely the notation and description in [Ijspeert et al., 2013], but at a slower pace.

Historical remark. Recently, the term "dynamicAL movement primitives" is preferred over "dynamic movement primitives". The newer term makes the relation to dynamicAL systems more clear, and avoids confusion about whether the output of "dynamical movement primitives" is in kinematic or dynamic space (it is usually in kinematic space).

Remark. This documentation and code focusses only on discrete movement primitives. For rythmic movement primitives, we refer to [Ijspeert et al., 2013].

Basic Point-to-Point Movements: A Critically Damped Spring-Damper System

At the heart of the DMP lies a spring-damper system, as described in Spring-Damper Systems. In DMP papers, the notation of the spring-damper system is usually a bit different:

$m\ddot{y} =$	$-ky - c\dot{y}$	spring-damper system, "traditional notation"
$m\ddot{y} =$	$c(-\frac{k}{c}y - \dot{y})$	
$\tau\ddot{y} =$	$\alpha(-\beta y - \dot{y})$	with $\alpha = c, \quad \beta = \dfrac{k}{c}, \quad m = \tau$
$\tau\ddot{y} =$	$\alpha(-\beta(y - y^g) - \dot{y})$	with attractor y^g
$\tau\ddot{y} =$	$\alpha(\beta(y^g - y) - \dot{y})$	typical DMP notation for spring-damper system

In the last two steps, we change the attractor state from 0 to y^g, where y^g is the goal of the movement.

To avoid overshooting or slow convergence towards y^g, we prefer to have a *critically damped* spring-damper system for the DMP. For such systems $c = 2\sqrt{mk}$ must hold, see Critical Damping. In our notation this becomes $\alpha = 2\sqrt{\alpha\beta}$, which leads to $\beta = \alpha/4$. This determines the value of β for a given value of α in DMPs. The influence of α is illustrated in the first figure in Introduction.

Rewriting the second order dynamical system as a first order system (see Rewriting one 2nd Order Systems as two 1st Order Systems) with expanded state $[y\ z]$ yields:

$$\begin{bmatrix} \dot{z} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} (\alpha(\beta(y^g - y) - z))/\tau \\ z/\tau \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} 0 \\ y_0 \end{bmatrix} \text{and attr. state} \begin{bmatrix} 0 \\ y^g \end{bmatrix}$$

Arbitrary Smooth Movements: the Forcing Term

The representation described in the previous section has some nice properties in terms of - Convergence towards the Attractor , Robustness to Perturbations , and Autonomy, but it can only represent very simple movements. To achieve more complex movements, we add a time-dependent forcing term to the spring-damper system. The spring-damper systems and forcing term are together known as a *transformation system*.

$$\begin{bmatrix} \dot{z} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} (\alpha(\beta(y^g - y) - z) + f(t))/\tau \\ z/\tau \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} 0 \\ y_0 \end{bmatrix} \text{and attr. state} \begin{bmatrix} ? \\ y^g \end{bmatrix}$$

The forcing term is an open loop controller, i.e. it depends only on time. By modifying the acceleration profile of the movement with a forcing term, arbitrary smooth movements can be achieved. The function $f(t)$ is usually a function approximator, such as locally weighted regression (LWR) or locally weighted projection regression (LWPR), see Function Approximation Module. The graph below shows an example of a forcing term implemented with LWR with random weights for the basis functions.

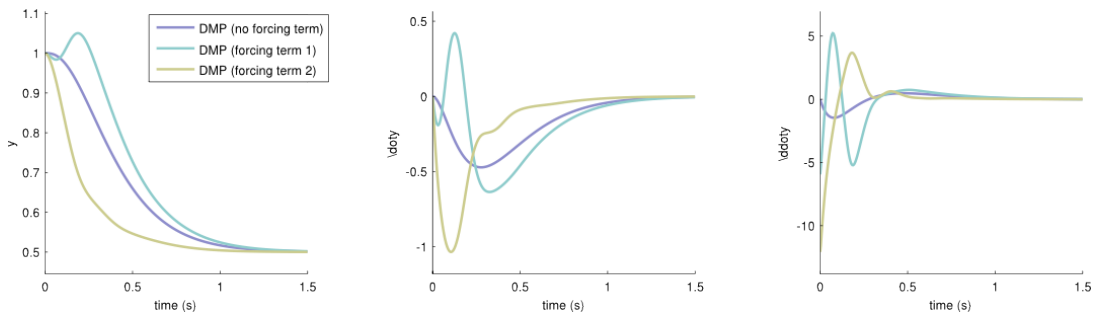

Figure 5: A non-linear forcing term enable more complex trajectories to be generated (these DMPs use a goal system and an exponential gating term).

Ensuring Convergence to 0 of the Forcing Term: the Gating System. Since we add a forcing term to the dynamical system, we can no longer guarantee that the system will converge towards x^g; perhaps the forcing term continually pushes it away x^g (perhaps it doesn't, but the point is that we cannot *guarantee* that it *always* doesn't). That is why there is a question mark in the attractor state in the equation above. To guarantee that the movement will always converge towards the attractor x^g, we need to ensure that the forcing term decreases to 0 towards the end

of the movement. To do so, a gating term is added, which is 1 at the beginning of the movement, and 0 at the end. This gating term itself is determined by, of course, a dynamical system. In [Ijspeert et al., 2002], it was suggested to use an exponential system. We add this extra system to our dynamical system by expanding the state as follows:

$$\dot{x} = \begin{bmatrix} \dot{z} \\ \dot{y} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} (\alpha_y(\beta_y(y^g - y) - z) + x \cdot f(t))/\tau \\ z/\tau \\ -\alpha_x x/\tau \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} 0 \\ y_0 \\ 1 \end{bmatrix} \text{ and attr. state} \begin{bmatrix} 0 \\ y^g \\ 0 \end{bmatrix}$$

Ensuring Autonomy of the Forcing Term: the Phase System. By introducing the dependence of the forcing term $f(t)$ on time t the overall system is no longer autonomous. To achieve independence of time, we therefore let f be a function of the state of an (autonomous) dynamical system rather than of t. This system represents the *phase* of the movement. [Ijspeert et al., 2002] suggested to use the same dynamical system for the gating and phase, and use the term *canonical system* to refer this joint gating/phase system. Thus the phase of the movement starts at 1, and converges to 0 towards the end of the movement, just like the gating system. The new formulation now is (the only difference is $f(x)$ instead of $f(t)$):

$$\begin{bmatrix} \dot{z} \\ \dot{y} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} (\alpha_y(\beta_y(y^g - y) - z) + x \cdot f(x))/\tau \\ z/\tau \\ -\alpha_x x/\tau \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} 0 \\ y_0 \\ 1 \end{bmatrix} \text{ and attr. state} \begin{bmatrix} 0 \\ y^g \\ 0 \end{bmatrix}$$

Todo

Discuss goal-dependent scaling, i.e. $f(t)s(x^g - x_0)$?

Multi-dimensional Dynamic Movement Primitives. Since DMPs usually have multi-dimensional states (e.g. one output $\mathbf{y}_{d=1\ldots D}$ for each of the D joints), it is more accurate to use bold fonts for the state variables (except the gating/phase system, because it is always 1D) so that they represent vectors:

$$\begin{bmatrix} \dot{\mathbf{z}} \\ \dot{\mathbf{y}} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} (\alpha_y(\beta_y(\mathbf{y}^g - \mathbf{y}) - \mathbf{z}) + x \cdot f(x))/\tau \\ \mathbf{z}/\tau \\ -\alpha_x x/\tau \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{z}_0 \\ 1 \end{bmatrix} \text{ and attr. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}^g \\ 0 \end{bmatrix}$$

So far, the graphs have shown 1-dimensional systems. To generate D-dimensional trajectories for, for instance, the 7 joints of an arm or the 3D position of its end-effector, we simply use D transformation systems. A key principle in DMPs is to use one and the same phase system for all of the transformation systems, to ensure that the output of the transformation systems are synchronized in time. The image below show the evolution of all the dynamical systems involved in integrating a multi-dimensional DMP.

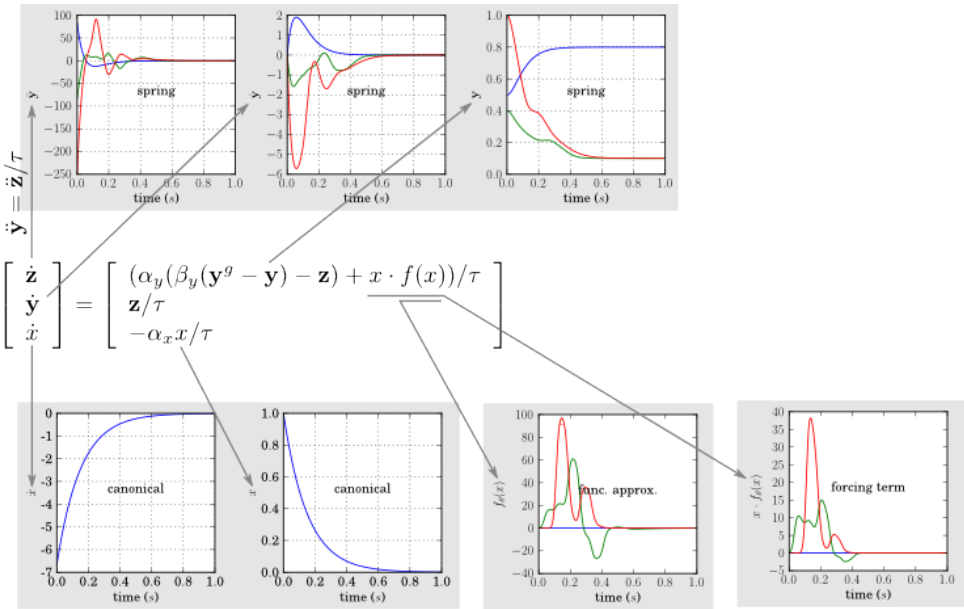

Figure 6: The various dynamical systems and forcing terms in multi-dimensional DMPs.

Implementation. *Since a Dynamical Movement Primitive is a dynamical system, the Dmp class derives from the DynamicalSystem class. It overrides the virtual function DynamicalSystem-::integrateStart(). Integrating the DMP numerically (Euler or 4th order Runge-Kutta) is done with the generic DynamicalSystem::integrateStep() function. It also implements the pure virtual function DynamicalSystem::analyticalSolution(). Because a DMP cannot be solved analytically (we cannot write it in closed form due to the arbitrary forcing term), calling Dmp::analyticalSolution() in fact performs a numerical Euler integration (although the linear subsystems (phase, gating, etc.) are analytically solved because this is faster computationally).*

Alternative Systems for Gating, Phase and Goals

Gating: Sigmoid System. A disadvantage of using an exponential system as a gating term is that the gating decreases very quickly in the beginning. Thus, the output of the function approximator $f(x)$ needs to be very high towards the end of the movement if it is to have any effect at all. This leads to scaling issues when training the function approximator.

Therefore, sigmoid systems have more recently been proposed [Kulvicius et al., 2012] as a gating system. This leads to the following DMP formulation (since the gating and phase system are no longer shared, we introduce a new state variable v for the gating term:

$$\begin{bmatrix} \dot{\mathbf{z}} \\ \dot{\mathbf{y}} \\ \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} (\alpha_y(\beta_y(\mathbf{y}^g - \mathbf{y}) - \mathbf{z}) + v \cdot f(x))/\tau \\ \mathbf{z}/\tau \\ -\alpha_x x/\tau \\ -\alpha_v v(1 - v/v_{\text{max}}) \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}_0 \\ 1 \\ 1 \end{bmatrix} \text{ and attr. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}^g \\ 0 \\ 0 \end{bmatrix}$$

where the term v_{max} is determined by τ

Phase: Constant Velocity System. In practice, using an exponential phase system may complicate imitation learning of the function approximator f, because samples are not equidistantly

spaced in time. Therefore, we introduce a dynamical system that mimics the properties of the phase system described in [Kulvicius et al., 2012], whilst allowing for a more natural integration in the DMP formulation, and thus our code base. This system starts at 0, and has a constant velocity of $1/\tau$, which means the system reaches 1 when $t = \tau$. When this point is reached, the velocity is set to 0.

$$\dot{x} = \begin{array}{l} 1/\tau \text{ if } x < 1 \\ 0 \text{ if } x > 1 \end{array}$$

This, in all honesty, is a bit of a hack, because it leads to a non-smooth acceleration profile. - However, its properties as an input to the function approximator are so advantageous that we have designed it in this way (the implementation of this system is in the TimeSystem class).

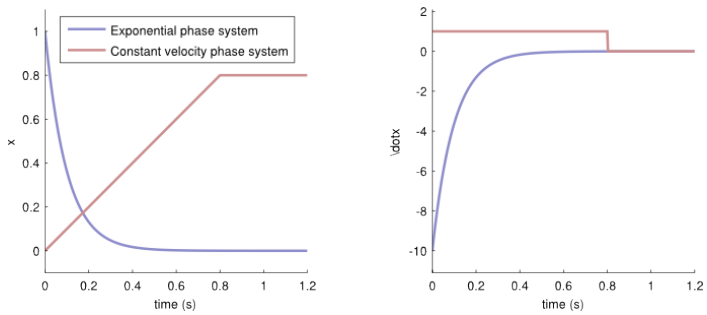

Figure 7: Exponential and constant velocity dynamical systems as the 1D phase for a dynamical movement primitive.

With the constant velocity dynamical system the DMP formulation becomes:

$$\begin{bmatrix} \dot{\mathbf{z}} \\ \dot{\mathbf{y}} \\ \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} (\alpha_y(\beta_y(\mathbf{y}^g - \mathbf{y}) - \mathbf{z}) + v \cdot f(x))/\tau \\ \mathbf{z}/\tau \\ 1/\tau \\ -\alpha_v v(1 - v/v_{\text{max}}) \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}_0 \\ 0 \\ 1 \end{bmatrix} \text{ and attr. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}^g \\ 1 \\ 0 \end{bmatrix}$$

Zero Initial Accelerations: the Delayed Goal System. Since the spring-damper system leads to high initial accelerations (see the graph to the right below), which is usually not desirable for robots, it was suggested to move the attractor of the system from the initial state y_0 to the goal state y^g *during* the movement [Kulvicius et al., 2012]. This delayed goal attractor y^{g_d} itself is represented as an exponential dynamical system that starts at y_0, and converges to y^g (in early versions of DMPs, there was no delayed goal system, and y^{g_d} was simply equal to y^g throughout the movement). The combination of these two systems, listed below, leads to a movement that starts and ends with 0 velocities and accelerations, and approximately has a bell-shaped velocity profile. This representation is thus well suited to generating human-like point-to-point movements, which have similar properties.

$$\begin{bmatrix} \dot{\mathbf{z}} \\ \dot{\mathbf{y}} \\ \dot{\mathbf{y}}^{g_d} \\ \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} (\alpha_y(\beta_y(\mathbf{y}^{g_d} - \mathbf{y}) - \mathbf{z}) + v \cdot f(x))/\tau \\ \mathbf{z}/\tau \\ -\alpha_g(\mathbf{y}^g - \mathbf{y}^{g_d}) \\ 1/\tau \\ -\alpha_v v(1 - v/v_{\text{max}}) \end{bmatrix} \quad \text{with init. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}_0 \\ \mathbf{y}_0 \\ 0 \\ 1 \end{bmatrix} \quad \text{and attr. state} \begin{bmatrix} \mathbf{0} \\ \mathbf{y}^g \\ \mathbf{y}^g \\ 1 \\ 0 \end{bmatrix}$$

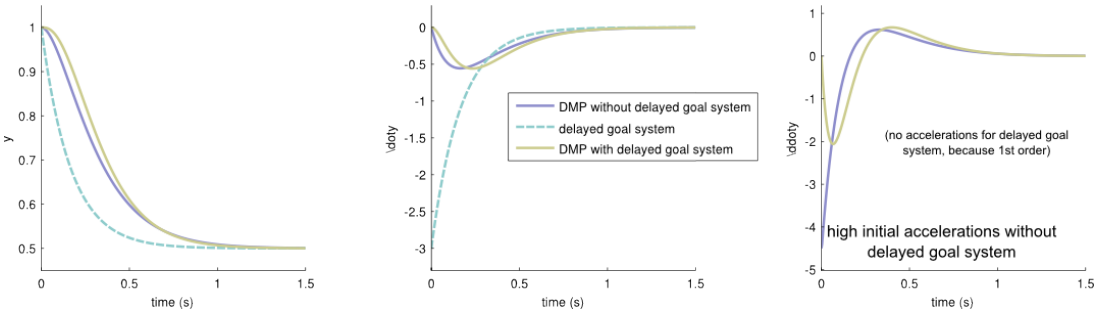

Figure 8: A first dynamical movement primitive, with and without a delayed goal system (left: state variable, center: velocities, right: accelerations.

In my experience, this DMP formulation is the best for learning human-like point-to-point movements (bell-shaped velocity profile, approximately zero velocities and accelerations at beginning and start of the movement), and generates nice normalized data for the function approximator without scaling issues (an exact empirical evaluation is on the stack...). The image below shows the interactions between the spring-damper system, delayed goal system, phase system and gating system.

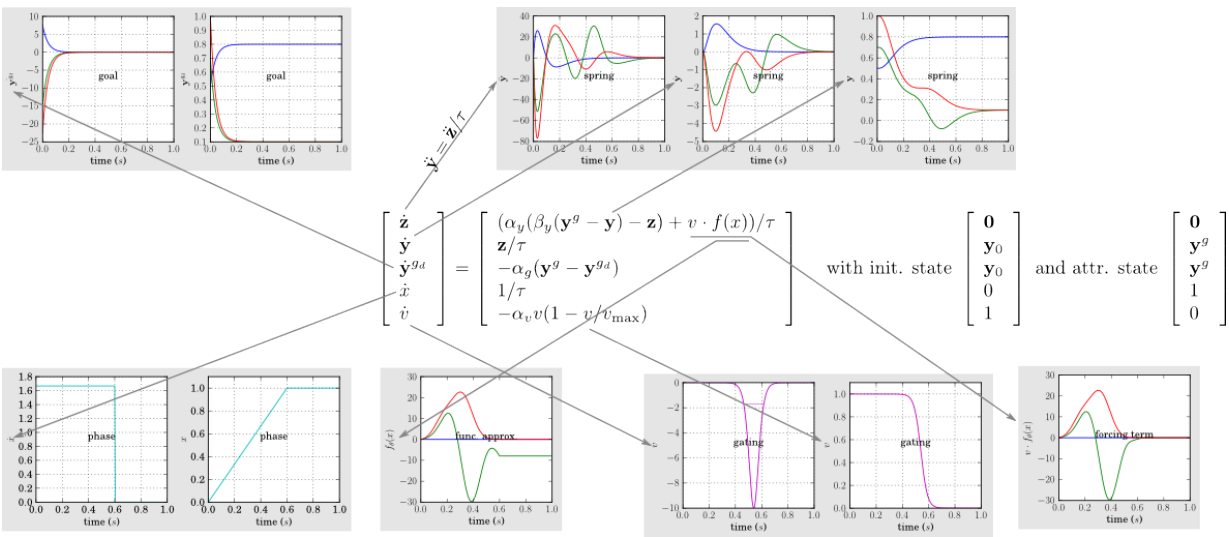

Figure 9: The various dynamical systems and forcing terms in multi-dimensional DMPs.

Known Issues

Todo

Known Issues

- Scaling towards novel goals

Summary

The core idea in dynamical movement primitives is to combine dynamical systems, which have nice properties in terms of convergence towards the goal, robustness to perturbations, and independence of time, with function approximators, which allow for the generation of arbitrary (smooth) trajectories. The key enabler to this approach is to gate the output of the function approximator with a gating system, which is 1 at the beginning of the movement, and 0 towards the end.

Further enhancements can be made by making the system autonomous (by using the output of a phase system rather than time as an input to the function approximator), or having initial velocities and accelerations of 0 (by using a delayed goal system).

Multi-dimensional DMPs are achieved by using multi-dimensional dynamical systems, and learning one function approximator for each dimension. Synchronization of the different dimensions is ensure by coupling them with only *one* phase system.

18

Bibliography

[Ijspeert et al., 2013] Ijspeert, A., Nakanishi, J., Pastor, P., Hoffmann, H., and Schaal, S. (2013). Dynamical Movement Primitives: Learning attractor models for motor behaviors. *Neural Computation*, 25(2):328–373. 11

[Ijspeert et al., 2002] Ijspeert, A. J., Nakanishi, J., and Schaal, S. (2002). Movement imitation with nonlinear dynamical systems in humanoid robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 11, 13

[Kulvicius et al., 2012] Kulvicius, T., Ning, K., Tamosiunaite, M., and Wörgötter, F. (2012). Joining movement sequences: Modified dynamic movement primitives for robotics applications exemplified on handwriting. *IEEE Transactions on Robotics*, 28(1):145–157. 14, 15

20

Function Approximation for the Forcing Term

(Please note that the the tutorial on Dynamical Movement Primitives is now basically over. The rest is more about documentation of the code.)

Function Approximation

This module implements a set of function approximators, i.e. supervised learning algorithms that are trained with demonstration pairs input/target, after which they make predictions for new inputs. For simplicity, this module implements only batch learning (not incremental), and does not allow trained function approximators to be retrained.

The two main functions are FunctionApproximator::train, which takes a set of inputs and corresponding targets, and FunctionApproximator::predict, which makes predictions for novel inputs.

MetaParameters and ModelParameters. In this module, algorithmic parameters are called - MetaParameters, and the parameters of the model when the function approximator has been trained are called ModelParameters. The rationale for this is that an untrained function approximator can be entirely reconstructed if its MetaParameters are known; this is useful for saving to file and making copies. A trained function approximator can be compeletely reconstructed given only its ModelParameters.

The life-cycle of a function approximator is as follows:

1. **Initialization:** The function approximator is initialized by calling the constructor with the MetaParameters. Its ModelParameters are set to NULL, indicating that the model is untrained.

2. **Training:** FunctionApproximator::train is called, which performs the conversion: train : MetaParameters $\times$ Inputs $\times$ Targets $\mapsto$ ModelParameters

3. **Prediction:** FunctionApproximator::predict is called, which performs the conversion: predict : ModelParameters $\times$ Input $\mapsto$ Output

Remark. FunctionApproximator::train in Step **2**. may only be called once. If you explicitly want to retrain the function approximator with novel input/target data call FunctionApproximator::reTrain() instead.

Remark. During the initialization, ModelParameters may also be passed to the constructor. This means that an already trained function approximator is initialized. Step **2**. above is thus skipped.

Changing the ModelParameters of a FunctionApproximator. The user should not be allowed to set the ModelParameters of a trained function approximator directly. Hence, FunctionApproximator::setModelParameters is protected. However, in order to change the values in-

side the model parameters (for instance when optimizing them), the user may call ModelParameters::getParameterVectorSelected and ModelParameters::setParameterVectorSelected it inherits these functions from Parameterizable). These take a vector of doubles, check if the vector has the right size, and get/set the ModelParameters accordingly.

Function approximators often have different types of model parameters. For instance, the model parameters of Locally Weighted Regression (FunctionApproximatorLWR) represent the centers and widths of the basis functions, as well as the slopes of the line segments. If you only want to get/set the slopes when calling ModelParameters::getParameterVectorSelected and -ModelParameters::setParameterVectorSelected, you must use ModelParameters::setSelectedParameters(const std::set<std::string>& selected_values_labels), for instance as follows:

```
std::set<std::string> selected;
selected.insert("slopes");
model_parameters.setSelectedParameters(selected);
Eigen::VectorXd values;
model_parameters.getParameterVectorSelected(values);
// "values" now only contains the slopes of the line segments

selected.clear();
selected.insert("centers");
selected.insert("slopes");
model_parameters.setSelectedParameters(selected);
model_parameters.getParameterVectorSelected(values);
// "values" now contains the centers of the basis functions AND slopes of the
       line segments
```

The rationale behind this implementation is that optimizers (such as evolution strategies) should not have to care about whether a particular set of model parameters contains centers, widths or slopes. Therefore, these different types of parameters are provided in one vector without semantics, and the generic interface is provided by the Parameterizable class.

Classes that inherit from Parameterizable (such as all ModelParameters and FunctionApproximator subclasses, must implement the pure virtual methods Parameterizable-::getParameterVectorAll Parameterizable::setParameterVectorAll and Parameterizable::getParameterVectorMask. Which gets/sets all the possible parameters in one vector, and a mask specifying the semantics of each value in the vector. The work of setting/getting the selected parameters (and normalizing them) is done in the Parameterizable class itself. This approach is a slightly longer run-time than doing the work in the subclasses, but it leads to more legible and robust code (less code duplication).

Black-box Optimization with Evolution Strategies

This module implements several evolution strategies for the optimization of black-box cost functions.

Black-box in this context means that no assumptions about the cost function can be made, for example, we do not have access to its derivative, and we do not even know if it is continuous or not.

The evolution strategies that are implemented are all based on reward-weighted averaging (aka probablity-weighted averaging), as explained in this paper/presentation: http://icml.-cc/discuss/2012/171.html

The basic algorithm is as follows:

```
x_mu = ??; x_Sigma = ?? // Initialize multi-variate Gaussian distribution
while (!halt_condition) {

    // Explore
    for k=1:K {
        x[k]     ~ N(x_mu,x_Sigma)    // Sample from Gaussian
        costs[k] = costfunction(x[k]) // Evaluate sample
    }

    // Update distribution
    weights = costs2weights(costs) // Should assign higher weights to lower
       costs
    x_mu_new = weights^T * x; // Compute weighted mean of samples
    x_covar_new = (weights .* x)^T * weights // Compute weighted covariance
       matrix of samples

    x_mu = x_mu_new
    x_covar = x_covar_new
}
```

Implementation

The algorithm above has been implemented as follows (see runEvolutionaryOptimization() and demoEvolutionaryOptimization.cpp):

```
int n_dim = 2; // Optimize 2D problem

// This is the cost function to be optimized
CostFunction* cost_function = new CostFunctionQuadratic(VectorXd::Zero(n_dim));

// This is the initial distribution
DistributionGaussian* distribution = new DistributionGaussian(VectorXd::Random(
      n_dim),MatrixXd::Identity(n_dim))
```

```
// This is the updater which will update the distribution
double eliteness = 10.0;
Updater* updater = new UpdaterMean(eliteness);

// Some variables
MatrixXd samples;
VectorXd costs;

for (int i_update=1; i_update<=n_updates; i_update++)
{

    // 1. Sample from distribution
    int n_samples_per_update = 10;
    distribution->generateSamples(n_samples_per_update, samples);

    // 2. Evaluate the samples
    cost_function->evaluate(samples,costs);

    // 3. Update parameters
    updater->updateDistribution(*distribution, samples, costs, *distribution);

}
```

CostFunction vs Task/TaskSolver. When the cost function has a simple structure, e.g. cost = x^2 it is convenient to implement the function x^2 in CostFunction::evaluate(). In robotics however, it is more suitable to make the distinction between a task (e.g. lift an object), and an entity that solves this task (e.g. your robot, my robot, a simulated robot, etc.). For these cases, the CostFunction is split into a Task and a TaskSolver, as follows:

```
CostFunction::evaluate(samples,costs) {
  TaskSolver::performRollouts(samples,cost_vars)
  Task::evaluate(cost_vars,costs)
}
```

The idea here is that the TaskSolver uses the samples to perform a rollout (e.g. the samples represent the parameters of a policy which is executed) and computes all the variables that are relevant to determining the cost (e.g. it records the forces at the robot's end-effector, if this is something that needs to be minimized)

Some further advantages of this approach:

- Different robots can solve the exact same Task implementation of the same task.
- Robots do not need to know about the cost function to perform rollouts (and they shouldn't)
- The intermediate cost-relevant variables can be stored to file for visualization etc.
- The procedures for performing the roll-outs (on-line on a robot) and doing the evaluation/updating/sampling (off-line on a computer) can be seperated, because there is a separate TaskSolver::performRollouts function.

When using the Task/TaskSolver approach, the runEvolutionaryOptimization process is as follows (only minor changes to the above):

```
int n_dim = 2; // Optimize 2D problem

// This is the cost function to be optimized
CostFunction* cost_function = new CostFunctionQuadratic(VectorXd::Zero(n_dim));

// This is the initial distribution
DistributionGaussian* distribution = new DistributionGaussian(VectorXd::Random(
      n_dim),MatrixXd::Identity(n_dim))

// This is the updater which will update the distribution
```

```
double eliteness = 10.0;
Updater* updater = new UpdaterMean(eliteness);

// Some variables
MatrixXd samples;
VectorXd costs;

for (int i_update=1; i_update<=n_updates; i_update++)
{

    // 1. Sample from distribution
    int n_samples_per_update = 10;
    distribution->generateSamples(n_samples_per_update, samples);

    // 2A. Perform the roll-outs
    task_solver->performRollouts(samples,cost_vars);

    // 2B. Evaluate the samples
    task->evaluate(cost_vars,costs);

    // 3. Update parameters
    updater->updateDistribution(*distribution, samples, costs, *distribution);

}
```