

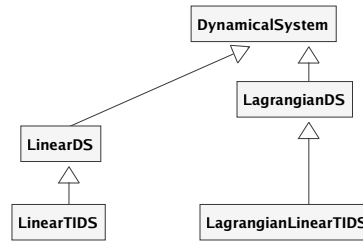
Dynamical Systems formulations in Siconos.

F. P rignon

For Kernel version 1.1.4
May 24, 2006

1 Class Diagram

There are five possible formulations for dynamical systems in Siconos, three for first order systems and two for second order Lagrangian systems. The main class is `DynamicalSystem`, all other derived from this one, as shown in the following diagram:



2 General non linear first order dynamical systems

→ **class** *DynamicalSystem*

This is the top class for dynamical systems. All other systems classes are derived from this one.

A general dynamical systems is described by the following set of n equations, completed with initial conditions:

$$\dot{x} = f(x, t) + T(x)u(x, t) + r \quad (1)$$

$$x(t_0) = x_0 \quad (2)$$

- x : state of the system - Vector of size n .
- $f(x, t)$: sometimes called vector field - Vector of size n .
- $u(x, t)$: control term - Vector of size $uSize$.
- $T(x)$: $n \times uSize$ matrix, related to control term.
- r : input due to non-smooth behavior - Vector of size n .

The Jacobian matrix, $\nabla_x f(x, t)$, of f according to x , $n \times n$ square matrix, is also a member of the class.

Initial conditions are given by the member x_0 , vector of size n . This corresponds to x value when simulation is starting, ie after a call to `strategy->initialize()`.

There are plug-in functions in this class for f , its Jacobian, $jacobianXF$, u and T . All of them can handle a vector of user-defined parameters:

Required parameters: n , x_0 , f (as a plug-in), its Jacobian.

Main functions of the class:

- *computeRhs(time)*: to compute right hand side of equation (1) (saved in member rhs)
- *computeJacobianXRhs(time)*: to compute the Jacobian according to x of the right-hand side.
- *computeF(time)*: to compute $f(x, t)$ and save it into rhs member. Plug-in function must be set with *setComputeFFunction(pluginPath, functionName)*.
- *computeJacobianXF(time)*: to compute $\nabla_x f(x, t)$, and save it into member *JacobianXF*. Plug-in function must be set with *setComputeJacobianXFFunction(pluginPath, functionName)*.

With *pluginPath* the name of the library that contains your plugin and *functionName* the name you give to the function. (for example if you defined your plug-in for f in the file *MyPlugin.cpp* and call the function *computeMyF*, then *pluginPath* = "MyPlugin.so" and *functionName* = "computeMyF".) Warning: the name of the plugin file must end with the string "Plugin", and the length of the string before "Plugin" must not exceed 6 letters.

The signature of each function (ie the number and type of arguments) must be exactly the same as the one given in *Kernel/src/plugin/DefaultPlugin.cpp* for the corresponding function.

3 First order linear dynamical systems → class *LinearDS*

Derived from *DynamicalSystem*, described by the set of n equations and initial conditions:

$$\dot{x} = A(t)x(t) + Tu(t) + b(t) + r \quad (3)$$

$$x(t_0) = x_0 \quad (4)$$

With:

- $A(t)$: $n \times n$ matrix, state independent but possibly time-dependent.
- $b(t)$: Vector of size n , possibly time-dependent.

Other variables are those of *DynamicalSystem* class.

A and B have corresponding plug-in functions.

Links with *vectorField* and its Jacobian are:

$$f(x, t) = A(t)x(t) + b(t) \quad (5)$$

$$jacobianXF = \nabla_x f(x, t) = A(t) \quad (6)$$

Main functions of the class:

- *computeRhs(time)*: to compute $Ax + b + Tu$ (saved in member rhs)
- *computeJacobianXRhs(time)*: compute $A(t)$.
- *computeA(time)*: to compute $A(t)$. Plug-in function must be set with *setComputeAFunction(pluginPath, functionName)*.
- *computeB(time)*: to compute $b(t)$. Plug-in function must be set with *setComputeBFunction(pluginPath, functionName)*.

4 First order time-invariant linear dynamical systems → class *LinearTIDS*

Derived from *DynamicalSystem*, described by the set of n equations and initial conditions:

$$\dot{x} = Ax(t) + Tu(t) + b + r \quad (7)$$

$$x(t_0) = x_0 \quad (8)$$

With:

- $A(t)$: $n \times n$ constant matrix
- $b(t)$: constant vector of size n

Other variables are those of *DynamicalSystem* class.

Links with *vectorField* and its Jacobian are:

$$f(x, t) = Ax(t) + b \quad (9)$$

$$jacobianXF = \nabla_x f(x, t) = A \quad (10)$$

Main functions of the class:

- *computeRhs(time)*: to compute $Ax + b + Tu$ (saved in member *rhs*)
- *computeJacobianXRhs(time)*: compute $A(t)$.

5 Second order non linear Lagrangian dynamical systems → class *LagrangianDS*

Lagrangian second order non linear systems are described by the following set of $nDof$ equations + initial conditions:

$$M(q)\ddot{q} + NNL(\dot{q}, q) + F_{Int}(\dot{q}, q, t) = F_{Ext}(t) + p \quad (11)$$

$$q(t_0) = q0 \quad (12)$$

$$\dot{q}(t_0) = velocity0 \quad (13)$$

With:

- $M(q)$: $nDof \times nDof$ matrix of inertia.
- q : state of the system - Vector of size $nDof$.
- \dot{q} or *velocity*: derivative of the state according to time - Vector of size $nDof$.
- $NNL(\dot{q}, q)$: non linear terms, time-independent - Vector of size $nDof$.
- $F_{Int}(\dot{q}, q, t)$: time-dependent linear terms - Vector of size $nDof$.
- $F_{Ext}(t)$: external forces, time-dependent BUT do not depend on state - Vector of size $nDof$.
- p : input due to non-smooth behavior - Vector of size $nDof$.

The following Jacobian are also member of this class:

- *jacobianQFInt* = $\nabla_q F_{Int}(t, q, \dot{q})$ - $nDof \times nDof$ matrix.
- *jacobianVelocityFInt* = $\nabla_{\dot{q}} F_{Int}(t, q, \dot{q})$ - $nDof \times nDof$ matrix.

- $\text{jacobianQNNL} = \nabla_q \text{NNL}(q, \dot{q})$ - $nDof \times nDof$ matrix.
- $\text{jacobianVelocityNNL} = \nabla_{\dot{q}} \text{NNL}(q, \dot{q})$ - $nDof \times nDof$ matrix.

There are plug-in functions in this class for F_{int} , F_{Ext} , M , NNL and the four Jacobian matrices. All of them can handle a vector of user-defined parameters.

Call `computeOperator(...)` to compute value for operator = F_{Int} , F_{Ext} , $Mass$, NNL , $JacobianQF_{Int}$, $JacobianVelocityF_{Int}$, $JacobianQNNL$, $JacobianVelocityNNL$. For any of them, link with plug-in function must be set using `setComputeOperatorFunction(pluginPath, pluginName)`.

Links with first order dynamical system are:

$$n = 2nDof \quad (14)$$

$$x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix} \quad (15)$$

$$f(x, t) = \begin{bmatrix} \dot{q} \\ M^{-1}(F_{Ext} - F_{Int} - NNL) \end{bmatrix} \quad (16)$$

$$\nabla_x f(x, t) = \begin{bmatrix} 0_{nDof \times nDof} & I_{nDof \times nDof} \\ \nabla_q(M^{-1})(F_{Ext} - F_{Int} - NNL) - M^{-1}\nabla_q(F_{Int} + NNL) & -M^{-1}\nabla_{\dot{q}}(F_{Int} + NNL) \end{bmatrix} \quad (17)$$

$$r = \begin{bmatrix} 0_{nDof} \\ p \end{bmatrix} \quad (19)$$

$$u(x, \dot{x}, t) = u_L(\dot{q}, q, t) \text{ (not yet implemented)} \quad (20)$$

$$T(x) = \begin{bmatrix} 0_{nDof} \\ T_L(q) \end{bmatrix} \text{ (not yet implemented)} \quad (21)$$

$$(22)$$

With 0_n a vector of zero of size n , $0_{n \times m}$ a $n \times m$ zero matrix and $I_{n \times n}$, identity $n \times n$ matrix.

Warning: control terms (Tu) are not fully implemented in Lagrangian systems. This will be part of future version.

6 Second order linear and time-invariant Lagrangian dynamical systems \rightarrow class *LagrangianLinearTIDS*

$$M\ddot{q} + C\dot{q} + Kq = F_{Ext}(t) + p \quad (23)$$

With:

- C : constant viscosity $nDof \times nDof$ matrix
- K : constant rigidity $nDof \times nDof$ matrix

And:

$$F_{Int} = C\dot{q} + Kq \quad (24)$$

$$NNL = 0_{nDof} \quad (25)$$

7 How to handle parameters in plug-in functions for Dynamical Systems

7.1 User management

All plug-in in `DynamicalSystem` class, or in its derived classes, have a possible user-defined parameter argument, usually the last one in the list (see `DefaultPlugin.cpp` file). This argument consists in a pointer

to SimpleVector, which can be defined by user in the following way:

In the main input file of your sample (python or cpp),

- first declare and define a SimpleVector
- assign this SimpleVector to the DynamicalSystem plug-in, using one of the two following:
 - `setParameter(yourVector, "id")`, where `"id"` is the plug-in name (see table 1 below for the list of available ids for each class). In that case, `yourVector` is a SimpleVector (not a pointer!). This will set `yourVector` values as an input list for parameters in plug-in function `"id"`.
 - `setParameterPtr(yourVector, "id")`, where `"id"` is the plug-in name (see table 1 below for the list of available ids for each class). Here, `yourVector` is a pointer to SimpleVector. This will link parameters vector in plug-in function `"id"` to `yourVector`. (Warning: this means that any change to one of them (`yourVector` and `parameter`) will affect the other).
- then in the corresponding `yourPlugin.cpp` file, the variable `param` corresponds to the vector you defined.

Example: suppose that you defined a LagrangianDS named `lds`, and want to set two parameters in the external forces, say `mu` and `lambda`. Then cpp input file looks like:

// In the main file:

```
double mu;
double lambda;
```

```
// ... give mu and lambda the required value
// ... declare and built your dynamical system
DynamicalSystem * lds = new LagrangianDS(...)
// === First way, with setParameter function ===
// declare and built a SimpleVector of size 2
SimpleVector parameters(2);
parameters(0) = mu;
parameters(1) = lambda;
```

```
lds->setParameter(parameters, "fExt");
```

// In this case, if parameters values are change after this step, this won't affect param values inside the dynamical system.

```
// === Second way, with setParameterPtr function ===
// declare and built a pointer to SimpleVector of size 2
SimpleVector * parameters = new SimpleVector(2);
(*parameters)(0) = mu;
(*parameters)(1) = lambda;
```

```
lds->setParameter(parameters, "fExt");
```

// Warning: in that case, from this point any change in parameters will affect param value in the dynamical system.
 //

// Then in the plug-in file, you have access to the parameter values:

```
extern "C" void computeFExt(const unsigned int&sizeOfq, const double *time, double *fExt, double
*param)
```

```

{
  for(unsigned int i=0; i<sizeOfq;++i)
    fExt[i] = cos(param[1]*time) + param[0];

  // this means that Fext = cos(lambda t) + mu
}

```

Warning: there is no relation between the name you give to your plug-in function (computeFExt in previous example) and the name given when you call `setParameter(..., id)` ("Fext" in previous example).

7.2 List of plug-in parameters id for Dynamical Systems

Parameter id in ...	Corresponding operator
DynamicalSystem class:	
"f" "jacobianXF" "u" "T"	$f(x, t)$ $\nabla_x f(x, t)$ $u(x, t)$ $T(x)$
LinearDS class:	
"A" "b"	$A(t)$ $b(t)$
LagrangianDS class:	
"mass" "fExt" "fInt" "NNL" "jacobianQFInt" "jacobianVelocityFInt" "jacobianQNNL" "jacobianVelocityNNL" "uL" "TL"	$M(q)$ $F_{Ext}(t)$ $F_{Int}(\dot{q}, q, t)$ $NNL(\dot{q}, q)$ $\nabla_q F_{Int}(\dot{q}, q, t)$ $\nabla_{\dot{q}} F_{Int}(\dot{q}, q, t)$ $\nabla_q NNL(\dot{q}, q)$ $\nabla_{\dot{q}} NNL(\dot{q}, q)$ $u_L(q, t)$ $T_L(q)$
LagrangianLinearTIDS class:	
"fExt"	$F_{Ext}(t)$

Table 1: List of available parameters id for plug-in functions in Dynamical Systems.

Mind that any item present in a class is also available in its derived classes.

Warning: if you create a new system using the copy constructor, any existing parameters in plug-in functions will also be copied. This means that no link between pointer for SimpleVector parameters will remain. In that case it may be better to re-set properly your parameters for the new created dynamical

system.