Code Generation for Simulation and Control Applications

Mats Jirstrand and Johan Gunnarsson

MathCore AB
Mjärdevi Science Park
SE-583 30 Linköping
Sweden

Third International Mathematica Symposium 1999,
23-25 August, RISC, Linz, Austria

Abstract

The use of *Mathematica* in combination with *MathCode C++* is illustrated in a context of modeling of dynamical systems and design of controllers. The symbolic tools are used to derive a set of nonlinear differential equations using Euler-Lagrange equations of motion. The model is converted to C++ using *MathCode C++*, which produces an efficient implementation of the large expressions used in the model. The exported code is used for simulations, which illustrates that *Mathematica* in combination with *MathCode C++* can be used to do accurate and powerful simulations of nonlinear systems. Controller synthesis is performed where the resulting controller is exported to C++ and run externally. The applications presented are a seesaw/pendulum process and aerodynamics of a fighter aircraft.

1 Introduction

Control system design is a discipline where advanced mathematics is applied to real world problems ranging from paper and pulp manufacturing, aircraft, and CD players to bio-chemical processes, logistics, and financial applications. *Mathematica* is very powerful for dealing with the mathematics in these control problems and the application package Control System Professional (CSP) implements many useful features. When it comes to nonlinear systems the combination of the symbolic and numeric capabilities of *Mathematica* makes modeling and control system design in this environment very attractive. The notebook concept for documentation of different trade-offs and decisions during the design process also contributes to making *Mathematica* suitable for this kind of work.

The application package *MathCode C++* adds automatic C++ code generation to *Mathematica*, which can be used both to enhance performance of simulations of large systems and to generate stand-alone C++ code to be used in applications separately from *Mathematica*. Performance will be the issue in the seesaw/pendulum example below while the stand-alone feature is explored in the fighter aircraft example. The stand-alone code generation feature of *MathCode C++* makes it possible to design and prototype a controller in *Mathematica* and then "lift out" the resulting stand-alone code

to be used in the real control system. This minimizes the need of coding the control law manually from the algorithms designed in *Mathematica*.

In this document we will try to illustrate the use of *Mathematica* together with *MathCode C++* for modeling, control system design, and code generation. The document is organized as follows. In the rest of this section we give some basic facts about dynamic systems needed to formulate controller design problems. In Section 2 a nonlinear mechanical system is modeled and simulated using generated C++ code. In Section 3 controller design for an aircraft is done and in Section 4 we give some conclusions.

■ Preliminaries

Many dynamic systems can be modeled by a set of first order differential equations, see e.g. [1, 2]

$$\begin{aligned} \frac{dx[t]}{dt} &= f[x[t], u[t]] \\ y[t] &= g[x[t]] \end{aligned} \tag{1}$$

where $f : \tilde{N}^n \; \tilde{N}^m \; \tilde{N}^n$. Here $x[t] \; \tilde{N}^n$, $u[t] \; \tilde{N}^m$, and $y[t] \; \tilde{N}^p$ are called the *states*, *inputs*, and *outputs* of the system, respectively. Equation (1) is called the *state space form* of the equations describing the behavior of the system and is particularly useful for control system design. We will also use the dot-notation $\dot{x}$ for denoting differentiation w.r.t. time $\frac{dx}{dt}$. If the system is linear the state space form can be written as

$$\begin{aligned} \frac{dx[t]}{dt} &= A.x[t] + B.u[t], \\ y[t] &= C.x[t] \end{aligned} \tag{2}$$

where $A \; \tilde{N}^{n \; n}$, $B \; \tilde{N}^{n \; m}$, and $C \; \tilde{N}^{p \; n}$ are constant real matrices. If $f[0, 0] \; 0$ in (1) the system can be linearized around x=0, u=0. The A, B, and C matrices are then computed from f and g by taking partial derivatives as follows

$$\begin{aligned} A &= \frac{\partial f[x,u]}{\partial x} \Big|_{\substack{x=0, \\ u=0}} \\ B &= \frac{\partial f[x,u]}{\partial u} \Big|_{\substack{x=0, \\ u=0}} \\ C &= \frac{\partial g[x]}{\partial x} \Big|_{x=0} \end{aligned} \tag{3}$$

The linearized model is usually a good approximation whenever the states and input to the system are small. There exists a large number of control design methods for linear systems which makes it desirable to work with linear models if possible during controller synthesis. The performance of the resulting controller can then be studied in simulations where the linear model has been exchanged with the nonlinear one.

A linear state-feedback controller is a very simple type of controller where the control signal is a linear combination of the states that are assumed to be measurable. Hence, a linear state-feedback law has the form

```
u[t] = -L.x[t]
```
(4)

There exists many methods for computing the matrix L but a necessary condition is that the chosen L makes the closed loop system asymptotically stable, which essentially means that non-zero states converge to zero. More or less advanced methods makes it possible to compute L such that different trade offs between performance and robustness can be obtained, see e.g. [1, 4, 5].

A slightly more advanced controller is the linear dynamic controller, which includes an internal model of the system to be controlled. A so called observer is used to estimated the states $x[t]$ of the system from measured signals $y[t]$ and a linear

feedback law is used to compute the input to the system to be controlled. Using the notation $x[t]$ for estimated states the linear dynamic controller can be written as follows

$$\begin{aligned} \frac{d\hat{x}[t]}{dt} &= A_c . \hat{x}[t] + B_c . y[t], \\ u[t] &= -L . \hat{x}[t] \end{aligned} \tag{5}$$

Hence, a dynamic controller requires real time solving of a system of differential equations. In the seesaw/pendulum example below we will use a linear state-feedback controller of the form (1) and in the fighter aircraft example we will use the observer based controller of the form (5).

2 The Seesaw/Pendulum Process

In this section we will illustrate how the symbolic capabilities of *Mathematica* can be used to derive a model of a nonlinear system. Using *MathCode C++* the large symbolic expressions in the nonlinear model are then converted to C++ code and used to simulate the system very efficiently. The system is a laboratory process frequently used in control education known as the seesaw/pendulum process. One implementation of the seesaw/pendulum process has been developed by Quanser Consulting (http://www.quanser.com).

The System

The process consists of a seesaw, two carts called C_1 and C_2, two parallel tracks, an inverted pendulum, and a weight. Each cart can be driven by a DC motor controlled by an input voltage. Cart C_1 carries the weight and cart C_2 carries the inverted pendulum attached by a friction free joint. The carts can be moved along the tracks by controlling the input voltages of the DC motors.

We start by modeling the open loop system, i.e., without any feedback from measured signals. The forces F_1 and F_2 acting on each cart are chosen as inputs. We will use the Lagrangian methodology to obtain a nonlinear model of the system and then linearize it. The linearized model is used for control system design where the Control System Professional (CSP) application package is used. The closed loop system are then simulated both within *Mathematica* and by external code generated using *MathCode C++*.

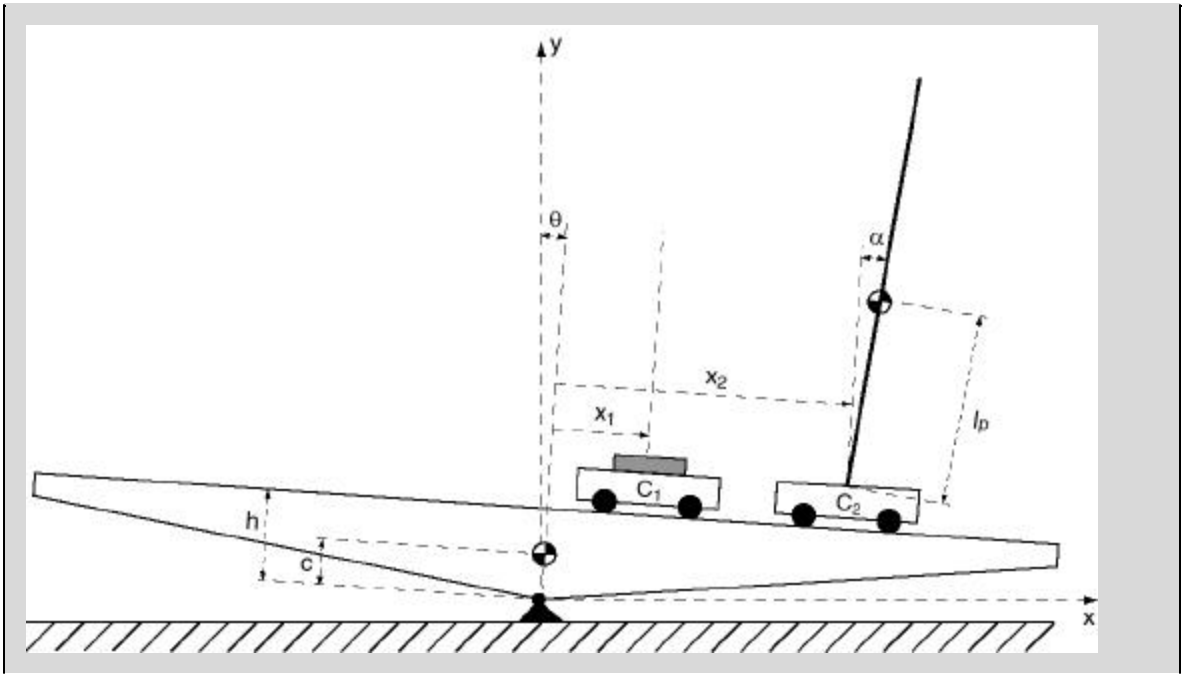

Figure 1. The Pendulum/Seesaw Process.

In Figure 1 we introduce names of variables and constants describing the process.

Variables:

x_1	translation of cart 1 from center of track
x_2	translation of cart 2 from center of track
	angle of seesaw with vertical
	angle of inverted pendulum with normal to track
F_1	force applied to cart 1
F_2	force applied to cart 2

Constants:

J	inertia of seesaw with track at height h
m_s	mass of seesaw with track
c	height of center of gravity of the seesaw from pivot point
h	height of track from pivot point
m_1	mass of cart 1 (weight cart, on the back track)
m_2	mass of cart 2 (pendulum cart, on the front track)
m_p	mass of pendulum
l_p	center of mass of pendulum rod (half of full length)
g	gravitational acceleration

The physical values of the above constants are stored as rules in *Mathematica* to be used later on in simulations.

```
physicalvalues := { J → 1.6, ms → 6.6, c → 0.06, h → 0.115,
  m1 → 0.48 + 0.38, m2 → 0.48, mp → 0.2, lp → 0.61/2 + 0.03, g → 9.81}
```

■ Modeling

Following the Lagrangian methodology [3] the system is divided in a number of subsystems whose potential energy and kinetic energy are computed in terms of the generalized coordinates introduced in Figure 1 above. The Lagrangian, which is the difference between the total kinetic and potential energy, can then be used to derive the equations of motion for the system.

Computation of the Lagrangian

Coordinates of the center of mass of the seesaw

```
x_s := c Sin[θ[t]]
y_s := c Cos[θ[t]]
```

The potential and kinetic energies of the seesaw

```
V_s := m_s g y_s
T_s := Simplify[1/2 J (∂_t θ[t])^2]
```

Coordinates of the center of track

```
x_c := h Sin[θ[t]]
y_c := h Cos[θ[t]]
```

Coordinates of cart 1

```
x_m1 := x_c + x_1[t] Cos[θ[t]]
y_m1 := y_c - x_1[t] Sin[θ[t]]
```

The potential and kinetic energies of cart 1

```
V_m1 := m_1 g y_m1
T_m1 := Simplify[1/2 m_1 ((∂_t x_m1)^2 + (∂_t y_m1)^2)]
```

Coordinates of cart 2

```
x_m2 := x_c + x_2[t] Cos[θ[t]]
y_m2 := y_c - x_2[t] Sin[θ[t]]
```

The potential and kinetic energies of cart 2

```
V_m2 := m_2 g y_m2
T_m2 := Simplify[1/2 m_2 ((∂_t x_m2)^2 + (∂_t y_m2)^2)]
```

Coordinates of the center of mass of the pendulum

```
x_p := x_m2 + l_p Sin[α[t] + θ[t]]
y_p := y_m2 + l_p Cos[α[t] + θ[t]]
```

The potential and kinetic energies of the pendulum

```
V_p := m_p g y_p
T_p := Simplify[1/2 m_p ((∂_t x_p)^2 + (∂_t y_p)^2)]
```

The total potential energy

```
V_tot := V_s + V_m1 + V_m2 + V_p
```

The total kinetic energy

```
T_tot := T_s + T_m1 + T_m2 + T_p
```

The Lagrangian of the system

```
L := T_tot - V_tot
```

The Lagrangian of the system becomes

```
Simplify[L]
```

$$\frac{1}{2}\Big(-2\,c\,g\,\text{Cos}[\theta[t]]\,m_s - 2\,g\,m_1\,(h\,\text{Cos}[\theta[t]] - \text{Sin}[\theta[t]]\,x_1[t]) -$$
$$2\,g\,m_2\,(h\,\text{Cos}[\theta[t]] - \text{Sin}[\theta[t]]\,x_2[t]) -$$
$$2\,g\,m_p\,(h\,\text{Cos}[\theta[t]] + \text{Cos}[\alpha[t] + \theta[t]]\,l_p - \text{Sin}[\theta[t]]\,x_2[t]) +$$
$$J\,\theta'[t]^2 + m_1\left((h^2 + x_1[t]^2)\,\theta'[t]^2 + 2\,h\,\theta'[t]\,x_1'[t] + x_1'[t]^2\right) +$$
$$m_2\left((h^2 + x_2[t]^2)\,\theta'[t]^2 + 2\,h\,\theta'[t]\,x_2'[t] + x_2'[t]^2\right) +$$
$$m_p\,(((h\,\text{Cos}[\theta[t]] - \text{Sin}[\theta[t]]\,x_2[t])\,\theta'[t] +$$
$$\text{Cos}[\alpha[t] + \theta[t]]\,l_p\,(\alpha'[t] + \theta'[t]) + \text{Cos}[\theta[t]]\,x_2'[t])\,\text{^}\,2 +$$
$$((h\,\text{Sin}[\theta[t]] + \text{Cos}[\theta[t]]\,x_2[t])\,\theta'[t] + \text{Sin}[\alpha[t] + \theta[t]]$$
$$l_p\,(\alpha'[t] + \theta'[t]) + \text{Sin}[\theta[t]]\,x_2'[t])\,\text{^}\,2)\Big)$$

Computation of the Equations of Motion

The equations of motion are given by the partial differential equations that L must satisfy

$$\frac{d}{dt}\frac{L}{q_i} - \frac{L}{q_i} = Q_i,\ i = 1, \ldots, n, \tag{6}$$

where L is the Lagrangian, $q_i,\ i = 1, \ldots, n$ are the generalized coordinates, and $Q_i,\ i = 1, \ldots, n$ are the generalized force associated with each coordinate. In this case the quadruple $\{q_1, q_2, q_3, q_4\}$ corresponds to $\{x_1, x_2, \ldots, \ldots\}$ and $Q_1 = F_1,\ Q_2 = F_2, Q_3 = Q_4 = 0$. We derive each equation and simplify it

```
eqn1 = Simplify[∂t (∂∂t x1[t] L) - ∂x1[t] L == F1];
eqn2 = Simplify[∂t (∂∂t x2[t] L) - ∂x2[t] L == F2];
eqn3 = Simplify[∂t (∂∂t θ[t] L) - ∂θ[t] L == 0];
eqn4 = Simplify[∂t (∂∂t α[t] L) - ∂α[t] L == 0];
```

These equations are coupled ordinary differential equations (ODEs) of second order in the generalized coordinates $x_1, \ldots, x_2, \ldots$. To rewrite these in standard state space form (a system of first order ODEs) we introduce the following states:

$$x = (x_1 \quad x_2 \quad \dot{x}_1 \quad \dot{x}_2).$$

The inputs to the system are

$$u = (F_1 \quad F_2).$$

We observe that the second order time derivatives of the positions and angles appears linearly which makes it easy to solve for these in terms the states. Second order time derivatives will *always* appear linearly in the equations derived from the partial differential equation (6) that the Lagrangian of the system has to satisfy. This is due to the fact that the Lagrangian only consists of at most first order time derivatives and the second order derivatives appear according to the chain rule when differentiating the term $\frac{L}{q_i}$ in (6) w.r.t. time.

Solve for second order derivatives

```
sol = Solve[ { eqn1, eqn2, eqn3, eqn4 },
   { ∂{t,2} x1[t], ∂{t,2} x2[t], ∂{t,2} θ[t], ∂{t,2} α[t] } ];
```

These solutions will become a part of the right hand sides of the differential equations used for simulating the system.

We introduce conversion rules to get rid of explicit time dependence. This makes some expressions look simpler and takes less space.

```
ssvariables := { x1[t] → x1, θ[t] → θ, x2[t] → x2, α[t] → α,
   ∂t x1[t] → dx1, ∂t θ[t] → dθ, ∂t x2[t] → dx2, ∂t α[t] → dα };
```

We compute the right hand side of the nonlinear state space form (1) of the equations

$x[t] \quad f[x[t], u[t]]$

which becomes very large! Observe that the first four entries in f correspond to pure integrations. This allow us to write the model as a system of first order differential equations as desired.

```
f = ( ∂t x1[t]
      ∂t θ[t]
      ∂t x2[t]
      ∂t α[t]
      ∂{t,2} x1[t]
      ∂{t,2} θ[t]
      ∂{t,2} x2[t]
      ∂{t,2} α[t] ) /. sol /. ssvariables // Flatten
```

The Linearized Model

Since we will design a controller using methods for linear systems we need to linearize the nonlinear state space model of the system around the origin.

$x \quad (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ u \quad (0\ 0) \qquad x[t] \quad A.x[t] \quad B.u[t].$

This can easily be obtained using the `Linearize` function in CSP

```
ss = Linearize[f, {x1, θ, x2, α},
        {{x1, 0}, {θ, 0}, {x2, 0},
     {α, 0}, {dx1, 0}, {dθ, 0}, {dx2, 0}, {dα, 0}},
        {{F1, 0}, {F2, 0}}] // Simplify;
Map[MatrixForm, ss]
```

$$\text{StateSpace}\left[\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -\frac{g\,h\,m_1}{J} & g-\frac{c\,g\,h\,m_s}{J} & -\frac{g\,h\,(m_2+m_p)}{J} & 0 & 0 & 0 & 0 & 0 \\ \frac{g\,m_1}{J} & \frac{c\,g\,m_s}{J} & \frac{g\,(m_2+m_p)}{J} & 0 & 0 & 0 & 0 & 0 \\ -\frac{g\,h\,m_1}{J} & g-\frac{c\,g\,h\,m_s}{J} & -\frac{g\,h\,(m_2+m_p)}{J} & -\frac{g\,m_p}{m_2} & 0 & 0 & 0 & 0 \\ -\frac{g\,m_1}{J} & -\frac{c\,g\,m_s}{J} & -\frac{g\,(m_2+m_p)}{J} & \frac{g\,(m_2+m_p)}{l_p\,m_2} & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{h^2}{J}+\frac{1}{m_1} & \frac{h^2}{J} \\ -\frac{h}{J} & -\frac{h}{J} \\ \frac{h^2}{J} & \frac{h^2}{J}+\frac{1}{m_2} \\ \frac{h}{J} & \frac{h}{J}-\frac{1}{l_p\,m_2} \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}\Big]$$

We extract the matrices from the linearization and substitute with the physical values of the parameters to get numerical entities

```
{A, B, C, D} = ss /. StateSpace → List /. physicalvalues;
```

```
A // MatrixForm
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -0.606381 & 9.53078 & -0.479464 & 0 & 0 & 0 & 0 & 0 \\ 5.27288 & 2.42797 & 4.16925 & 0 & 0 & 0 & 0 & 0 \\ -0.606381 & 9.53078 & -0.479464 & -4.0875 & 0 & 0 & 0 & 0 \\ -5.27288 & -2.42797 & -4.16925 & 41.4851 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
B // MatrixForm
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1.17106 & 0.00826563 \\ -0.071875 & -0.071875 \\ 0.00826563 & 2.0916 \\ 0.071875 & -6.14703 \end{pmatrix}$$

```
C // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
D // MatrixForm
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

The linear model of the system allow us to use standard methods for controller design.

■ Controller Design

The method for computing the feedback law requires the linear model to be controllable. This implies that the system can be stabilized and that the method will find a numerical instantiation of `L` such that this is obtained. To check controllability we use the `Controllable` command in CSP.

```
Controllable[ss /. physicalvalues]
```

```
True
```

Compute a full state feedback controller using the LQG-method (`LQRegulatorGains` is a CSP command).

```
L = LQRegulatorGains[
   ss /. physicalvalues,
   DiagonalMatrix[{1, 5., 1, 5., 0, 0, 0, 0}],
   0.1 IdentityMatrix[2] ];
L // MatrixForm
```

$$\begin{pmatrix} 26.7929 & 52.94 & 21.3676 & 11.8261 & 7.23982 & 18.3075 & 6.54028 & 2.2892 \\ -7.65183 & -26.7908 & -10.0775 & -27.7259 & -1.62889 & -8.77817 & -5.38841 & -4.54939 \end{pmatrix}$$

The control law to be used is $u[t]$ L.$x[t]$, where $x[t]$ are measurements of the states. This gives the following closed loop system to simulate $x[t]$ $f[x[t],$ L.$x[t]]$.

■ Simulation and Code Generation

Preliminaries

We store the states as a vector

```
xss := {x1, θ, x2, α, dx1, dθ, dx2, dα}
```

and close the feedback loop which gives the following right hand side of the state space equations

```
f1 = f /. Thread[{F1, F2} → -L.xss];
```

Change names of variables to only ASCII characters (needed for code generation)

```
Cvariables =
  {x1 → x1, θ → theta, x2 → x2, α → alpha,
   dx1 → dx1, dθ → dtheta, dx2 → dx2, dα → dalpha};
```

The right hand side of the closed loop state space equations (intended for code generation)

```
rhs = f1 /. physicalvalues /. Cvariables;
```

We define the function `ClosedLoopPendulumEquationsRHS`, which has the huge right hand side expression from above as its body. The type declarations `Real` and `→ Real[8]` below are the syntax in *MathCode C++* for providing type information. Explicit type declarations is needed to generate efficient C++ code.

```
ClosedLoopPendulumEquationsRHS[
    Real x1_, Real theta_, Real x2_, Real alpha_,
    Real dx1_, Real dtheta_, Real dx2_, Real dalpha_] → Real[8] = rhs;
```

Simulation of the Nonlinear Model within *Mathematica*

We define $f_{cl}[t]$ to be a short notation for the ClosedLoopPendulumEquationsRHS

```
fcl[t_] := ClosedLoopPendulumEquationsRHS[x1[t], theta[t],
  x2[t], alpha[t], dx1[t], dtheta[t], dx2[t], dalpha[t]]
```

The state vector

```
x[t_] := {x1[t], theta[t], x2[t],
   alpha[t], dx1[t], dtheta[t], dx2[t], dalpha[t]};
```

The differential equations for the closed loop system

```
deq := Thread[x'[t] == fcl[t]]
```

We need to specify some initial conditions for the simulation. Assume that cart 1 is positioned at -1, the seesaw is horizontal, cart 2 is positioned at +1 and the pendulum has a small deviation of 5.7° from the vertical axis. Furthermore, assume that the system is at rest at these positions (all time derivatives are zero).

```
initeq := Thread[x[0] == {-1, 0, 1, 0.1, 0, 0, 0, 0}]
```

Nonlinear System Simulation

We simulate the system using NDSolve.

```
dsolStandard = NDSolve[ deq ⋃ initeq, {x1, theta, x2,
     alpha, dx1, dtheta, dx2, dalpha}, {t, 0, 10} ]; // Timing
NDSolveStandardTime = First[%];

{0.952 Second, Null}
```

Linear System Simulation

```
dlineq := Thread[x'[t] == (A - B.L).x[t]] // Chop
```

We simulate the linear system using NDSolve.

```
dsollin = NDSolve[ dlineq ⋃ initeq,
   {x1, theta, x2, alpha, dx1, dtheta, dx2, dalpha}, {t, 0, 8} ];
```

Plot the result for the first 4 state variables from the nonlinear simulation together with the result for the linear system.

```
Map[
    Plot[#, {t, 0, 8}, PlotRange → All, DisplayFunction → Identity] &,
  {Part[Flatten[x[t] /. dsolStandard], Range[4]],
       Part[Flatten[x[t] /. dsollin], Range[4]]} // Transpose ];
```

```
Show[GraphicsArray[Partition[%, 2]], Display
       Function → $DisplayFunction];
```

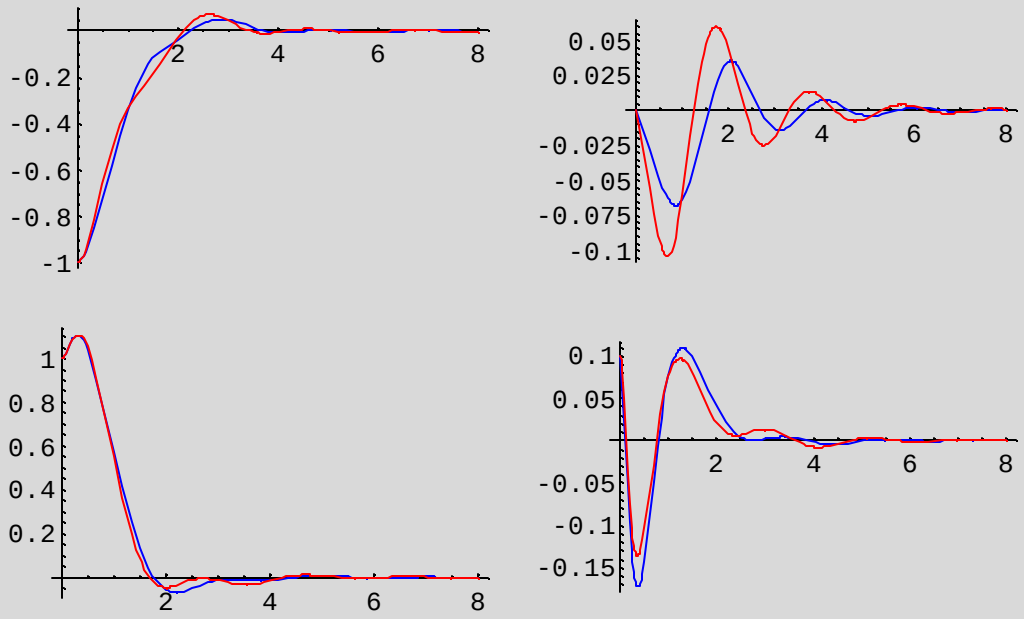

Figure 2. Initial value response for the linear (red) and nonlinear (blue) system.

We notice that the linear response differs significantly from the nonlinear one. The difference between the linear and nonlinear response will be much smaller if the initial values are chosen to be smaller since the linear model only is a good approximation for small values of the states and inputs.

External Simulation of the Nonlinear Model

To generate simulation code to be run outside *Mathematica* we have to provide a differential equation solver that can be compiled or linked when the executable program is generated. In this case we have chosen to implement a very simple Runge-Kutta solver i *Mathematica*, which will be included in the generated code. Another solution could be to use a solver from a publicly available software library like the solvers at www.netlib.org.

Code for External Simulation of the Nonlinear Model

Here follows an implementation of a state equation solver using the Runge-Kutta method

```
RK[Integer n_, Real h_, Real t0_, Real[_] startv_,
   Integer dimen_, Integer dimPlusOne_] → Real[n, dimPlusOne] :=
 Module[{
   Real[dimen] {x, k1, k2, k3, k4},
   Integer i,
   Real t,
   Real[n, dimPlusOne] res
  },
  t = t0;
  x = startv;
  res⟦1, 1⟧ = t;
  res⟦1, 2 | dimPlusOne⟧ = x;
  For[i = 1, i ≤ n - 1, i++,
   k1 = h ClosedLoopPendulumEquationsRHS[
      x⟦1⟧, x⟦2⟧, x⟦3⟧, x⟦4⟧, x⟦5⟧, x⟦6⟧, x⟦7⟧, x⟦8⟧];
   k2 = h ClosedLoopPendulumEquationsRHS[x⟦1⟧ + k1⟦1⟧ / 2,
      x⟦2⟧ + k1⟦2⟧ / 2, x⟦3⟧ + k1⟦3⟧ / 2, x⟦4⟧ + k1⟦4⟧ / 2, x⟦5⟧ + k1⟦5⟧ / 2,
      x⟦6⟧ + k1⟦6⟧ / 2, x⟦7⟧ + k1⟦7⟧ / 2, x⟦8⟧ + k1⟦8⟧ / 2];
   k3 = h ClosedLoopPendulumEquationsRHS[x⟦1⟧ + k2⟦1⟧ / 2,
      x⟦2⟧ + k2⟦2⟧ / 2, x⟦3⟧ + k2⟦3⟧ / 2, x⟦4⟧ + k2⟦4⟧ / 2, x⟦5⟧ + k2⟦5⟧ / 2,
      x⟦6⟧ + k2⟦6⟧ / 2, x⟦7⟧ + k2⟦7⟧ / 2, x⟦8⟧ + k2⟦8⟧ / 2];
   k4 = h ClosedLoopPendulumEquationsRHS[
                        x⟦1⟧ + k3⟦1⟧, x⟦2⟧ + k3⟦2⟧, x⟦3⟧ + k3⟦3⟧,
      x⟦4⟧ + k3⟦4⟧, x⟦5⟧ + k3⟦5⟧, x⟦6⟧ + k3⟦6⟧, x⟦7⟧ + k3⟦7⟧, x⟦8⟧ + k3⟦8⟧];
   x = x + 1 / 6 (k1 + 2 k2 + 2 k3 + k4);
   t = t + h;
   res⟦i + 1, 1⟧ = t;
   res⟦i + 1, 2 | dimPlusOne⟧ = x;
  ];
  res
 ]
```

We use RK in the following ODE solver

```
ndSolve[initvalue_, maxstepsize_, {x_, xmin_, xmax_}] :=
 Module[{n, dimen},
  n = IntegerPart[(xmax - xmin) / maxstepsize] + 1;
  dimen = Length[initvalue];
  res = RK[n, maxstepsize, xmin, initvalue, dimen, dimen + 1];
  Array[(Interpolation[AppendRows[
       Transpose[{res[[_, 1]]}], Transpose[{res[[_, # + 1]]}]]]) &, {8}]
 ]
```

To get a correct estimate of the time spent by the external code for simulating the system we define the following functions that repeats the external simulation n times.

```
RKRepeated[Integer n_, Real h_, Real t0_, Real[_] startv_, Integer dimen_,
   Integer dimPlusOne_, Integer loops_] → Real[n, dimPlusOne] :=
 Module[{Real[n, dimPlusOne] res},
  Do[res = RK[n, h, t0, startv, dimen, dimPlusOne], {loops}];
  res
 ]
```

```
ndSolveRepeated[
  initvalue_, maxstepsize_, {x_, xmin_, xmax_}, loops_] :=
 Module[{n, dimen},
  n = IntegerPart[(xmax - xmin) / maxstepsize] + 1;
  dimen = Length[initvalue];
  res =
   RKRepeated[n, maxstepsize, xmin, initvalue, dimen, dimen + 1, loops];
  Array[(Interpolation[AppendRows[Transpose[{res[[_, 1]]}],
       Transpose[{res[[_, # + 1]]}]]]) &, {8}]
 ]
```

Compilation

We are now ready to compile the package. The right hand side of the state equations are optimized using common subexpression elimination

```
CompilePackage[EvaluateFunctions → {ClosedLoopPendulumEquationsRHS}]
```

```
Successful compilation to C++: 3 function(s)
```

The corresponding code can be inspected in a text editor

```
Run["Notepad Global.cc"];
```

All necessary files for building binaries are now generated. The `MakeBinary` command can build either a standalone application using the code specified by an option `GenerateMainFileAndFunction` or a *MathLink* version which can be easily installed into *Mathematica*. We build the *MathLink* version

```
MakeBinary[];
```

To use the generated code for the simulation we only have to install the code using the `InstallCode` command from *MathCode C++*.

External Simulation

Uncompiled code (computations within *Mathematica*)

```
dsolUncompiled =
   ndSolve[{-1, 0, 1, 0.1, 0, 0, 0, 0}, 0.01, {t, 0., 10.}]; // Timing
ndSolveUncompiledTime = First[%];

{85.142 Second, Null}
```

We plot the result to verify the Runge-Kutta solver

```
Map[Plot[#[t], {t, 0, 8}, PlotRange → All, DisplayFunction → Identity,
    PlotStyle → Blue] &, Part[dsolUncompiled, Range[4]]];
```

```
Show[GraphicsArray[Partition[%, 2]],
  DisplayFunction → $DisplayFunction];
```

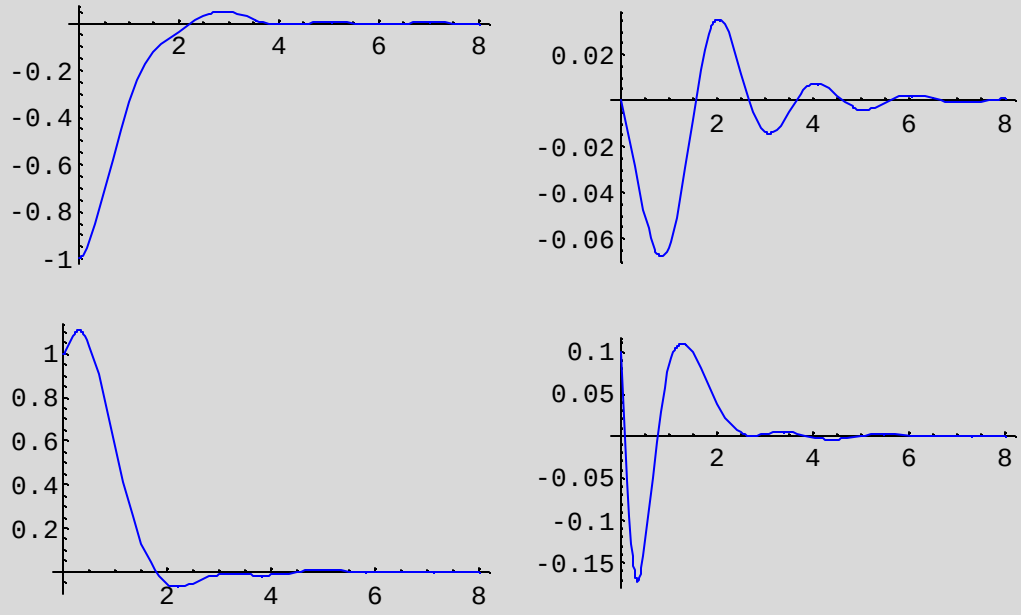

Figure 3. Initial value response for the nonlinear system computed internally by *Mathematica*.

Compiled code (external computations)

We install the executable code

```
InstallCode[];
```

```
Global is installed.
```

The external simulation is performed 200 times to get a accurate measure of its timing

```
dsolCompiled = ndSolveRepeated[
    {-1, 0, 1, 0.1, 0, 0, 0, 0}, 0.01, {t, 0., 10.}, 200]; // AbsTime
ndSolveCompiledTime = First[%] / 200;
```

```
{12.00000 Second, Null}
```

We plot the result to verify the compiled Runge-Kutta solver

```
Map[Plot[#[t], {t, 0, 8}, PlotRange → All, DisplayFunction → Identity,
    PlotStyle → Blue] &, Part[dsolCompiled, Range[4]]];
```

```
Show[GraphicsArray[Partition[%, 2]],
  DisplayFunction → $DisplayFunction];
```

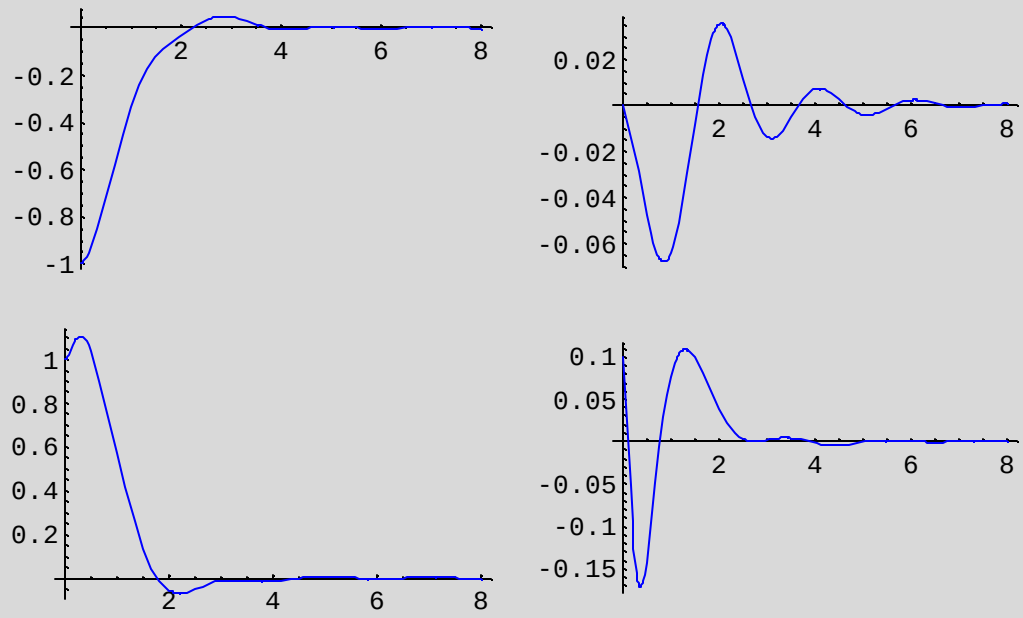

Figure 4. Initial value response for the nonlinear system computed by the external code.

The responses in Figure 3 and Figure 4 are identical.

Performance Comparisons

The difference in performance between the uncompiled and compiled version of the ODE solver is

```
ndSolveUncompiledTime / ndSolveCompiledTime
```

```
1419.03
```

A somewhat unfair comparison between the timings of the built-in ODE solver and the compiled solver gives

```
NDSolveStandardTime / ndSolveCompiledTime
```

```
15.8667
```

■ Clean-up

Uninstall the code and delete the temporary files.

```
UninstallCode[];
CleanMathCodeFiles[Confirm → False, CleanAllBut → {}];
```

Remove the functions for which we have generated code.

```
Remove[ClosedLoopPendulumEquationsRHS,
 RK, ndSolve, RKRepeated, ndSolveRepeated]
```

Unset the symbol *f* storing the right hand side of the state space equations.

```
f =.;
```

3 A Fighter Aircraft

In this section a controller for a linear system is designed using LQG techniques, see e.g. [1, 4]. The resulting filter is transformed into discrete form and C++ code is generated corresponding to the filter equations. A extremely simple simulation is used to compare the generated code with the uncompiled *Mathematica* functions.

■ The System

We will consider a linearization of a nonlinear aircraft dynamics for some specific flight conditions. We only study horizontal motions.

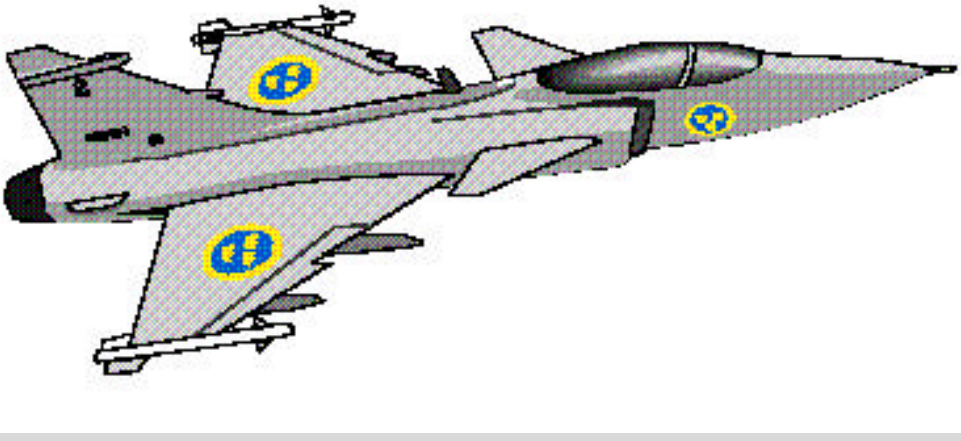

Figure 5. A fighter aircraft.

■ Modeling

Introduce the following notation:

	course angle
v_y	velocity across the aircraft in the y-direction
	roll angle
$_a$	aileron deflection
$_r$	rudder deflection

and choose the state vector x $\{v_y, p, r, \ , \ , \ _a, \ _r\}$, where p and r .

Notation for a Linear Time Invariant (LTI) system given in state-space form:

$$\begin{aligned} \dot{x} &= A\,x + B\,u + N\,w \\ z &= M\,x + D_z\,u \\ y &= C\,x + D_y\,u + v \end{aligned} \tag{7}$$

The following numerical instantiations of the system matrices comes from an example in [1].

```
A = {{-0.292,   8.13,    -202,    9.77,   0, -12.5,  17.1},
     {-0.152,  -2.54,   0.561, -0.0004,   0,  107,   7.68},
     {0.0364, -0.0678, -0.481,  0.0012,   0,  4.67, -7.98},
     {   0,       1,   0.0401,    0,      0,    0,     0},
     {   0,       0,      1,      0,      0,    0,     0},
     {   0,       0,      0,      0,      0,  -20,     0},
     {   0,       0,      0,      0,      0,    0,   -20}};

B = {{   0,    -2.15},
     {-31.7,  0.0274},
     {   0,    1.48},
     {   0,       0},
     {   0,       0},
     {  20,       0},
     {   0,      20}};

N = {{ 0.292,    0.001,    0.97,    0.0032},
     { 0.152,    -2.54,   0.552,  0.000043},
     {-0.0364, -0.0688, -0.456, -0.00012},
     {   0,       0,       0,       0},
     {   0,       0,       0,       0},
     {   0,       0,       0,       0},
     {   0,       0,       0,       0}};

M = C = {{0, 0, 0, 1, 0, 0, 0},
         {0, 0, 0, 0, 1, 0, 0}};

Dz = Dy = {{0, 0},
           {0, 0}};
```

■ Controller Design

Suppose that we have the following wind disturbance acting on the system:

```
w1[t_] := 10 t /; 0 ≤ t ∧ t ≤ 1
w1[t_] := 10 /; 1 < t ∧ t < 2
w1[t_] := -10 (t - 3) /; 2 ≤ t ∧ t ≤ 3
w1[t_] := 0 /; t > 3

w4[t_] := 10 /; 0 ≤ t ∧ t ≤ 1
w4[t_] := 0 /; 1 < t ∧ t < 2
w4[t_] := -10 /; 2 ≤ t ∧ t ≤ 3
w4[t_] := 0 /; t > 3
```

```
Plot[{w1[t], w4[t]}, {t, 0, 10},
  PlotLabel → "A wind disturbance", AxesLabel → {"t", "w1, w4"}];
```

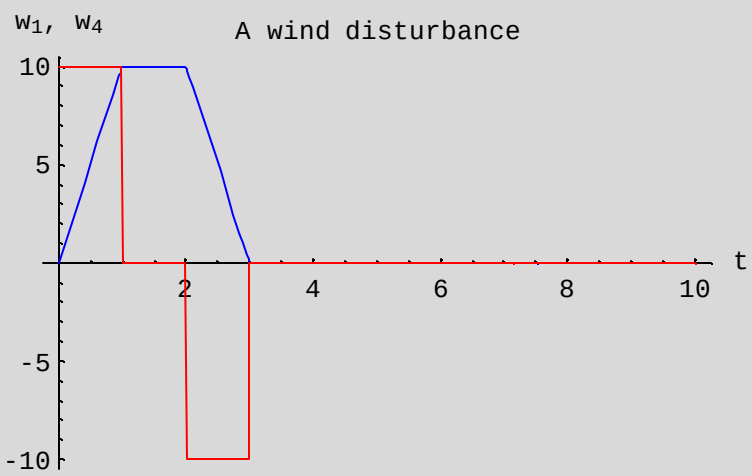

Figure 6. Some wind disturbances acting on the aircraft.

How will the system respond to these disturbances? We plot a simulation of the system response for 10 seconds.

```
plot1 = SimulationPlot[StateSpace[A, N, M, 0_{2×4}],
   {w1[t], 0, 0, w4[t]}, {t, 0, 10}, Sampled → Period[0.05],
   PlotRange → All, PlotStyle → {Blue, Red}, AxesLabel → {"t", "ϕ, ψ"}];
```

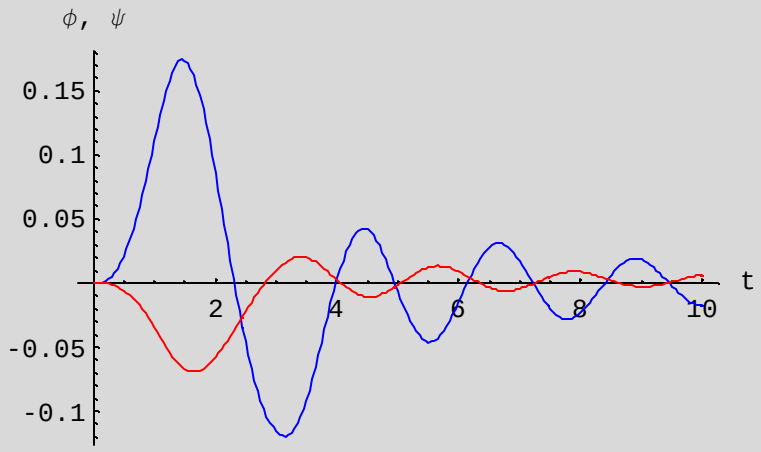

Figure 7. The response in roll angle, (blue) and course angle, (red) to the wind disturbances plotted in Figure 6.

An LQ State-Feedback Design

We compute a state-feedback, i.e., an L matrix, by minimizing a quadratic criterion with weight matrices Q_1 and Q_2. The quadratic criterion is given by

$$\min_L \int_0^\infty x^T[t]\, Q_1\, x[t] + u^T[t]\, Q_2\, u[t]\, dt \qquad (8)$$

We play around with the matrices Q_1 and Q_2 until we are satisfied with the result.

```
Q1 = I2;
Q2 = 0.1 I2;
```

```
L = LQOutputRegulatorGains[StateSpace[A, B, M, Dz], Q1, Q2]
```

```
{{-0.00425964, 0.442824, 0.155747, 3.14624,
  0.369764, 1.69149, 0.0369027}, {-0.00789519, 0.0587867,
  -0.836838, 0.305382, -3.14059, 0.142669, 0.26782}}
```

The closed loop eigenvalues are given by

```
A - B.L // Eigenvalues // TableForm
```

```
-21.8913
-20.0173
-9.49576 + 11.3361 i
-9.49576 - 11.3361 i
-2.35171 + 3.73879 i
-2.35171 - 3.73879 i
-0.178267
```

A necessary and sufficient condition for asymptotic stability of the closed loop system is that these eigenvalues belong to the left half plane.

We simulate the closed loop output response:

```
plot2 =
  SimulationPlot[StateSpace[A - B.L, N, M, 0_{2×4}], {w_1[t], 0, 0, w_4[t]},
   {t, 0, 10}, Sampled → Period[0.05], PlotRange → All, PlotStyle →
    {{Blue, Dashing[{.02, .02}]}, {Red, Dashing[{.02, .02}]}}];
```

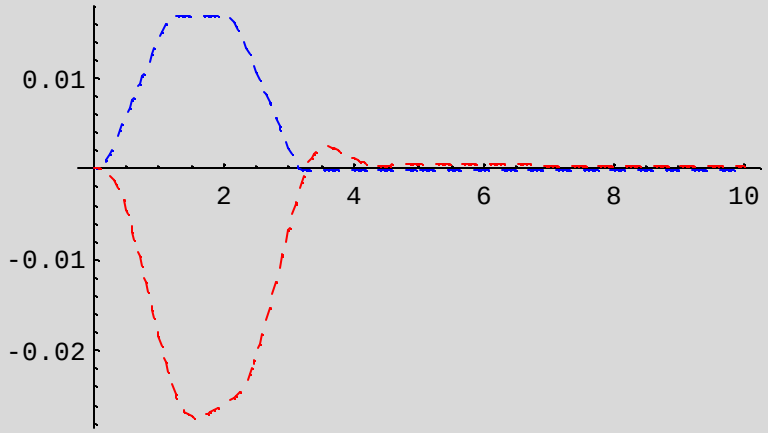

Figure 8. The response in roll angle, (blue) and course angle, (red) to the wind disturbances plotted in Figure 6 for the controlled system.

A comparison between the disturbance attenuation with (dashed) and without the controller:

```
Show[plot1, plot2];
```

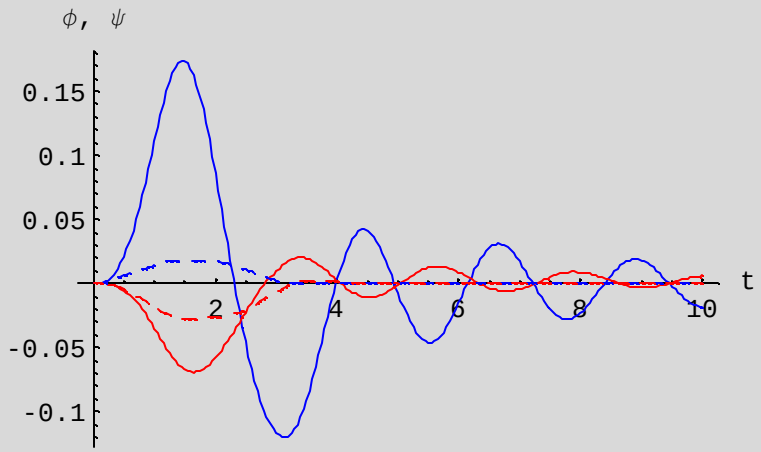

Figure 9. A comparison between the response in roll angle, (blue) and course angle, (red) to the wind disturbances plotted in Figure 6 for the controlled (dashed) and uncontrolled system, respectively.

Kalman Filter

Now suppose that the states cannot be measured. Here we will design a linear dynamic controller known as a Kalman filter which is used to estimate the states from measured signals.

Assume that the noise have the following covariance matrices:

```
R1 = 10^-2 I4;
R2 = 10^-6 I2;
```

```
K = LQEstimatorGains[StateSpace[A, N, C], R1, R2]
```

```
RiccatiSolve::meig :
 Matrix with multiple eigenvalues encountered. Result may be inaccurate.
```

```
{{-55.172, -1720.19}, {237.793, 15.4191}, {-1.66676, 34.8232},
 {21.7991, 0.502792}, {0.502792, 8.33028}, {0., 0.}, {0., 0.}}
```

We check the eigenvalues of the dynamic controller to verify that it is stable.

```
A - K.C // Eigenvalues // TableForm
```

```
-20.
-20.
-11.4064 + 11.4047 i
-11.4064 - 11.4047 i
-4.48281 + 5.19271 i
-4.48281 - 5.19271 i
-0.203896
```

A Dynamic Controller

The Kalman filter equations are

$$\dot{x}_e = A x_e + B u + K (y - C x_e) \tag{9}$$

and the computed state-feedback control law is of the form

$$u = -L x + L_r r. \tag{10}$$

if we add a reference signal *r*.

Using the separation principle [1, 4] we can use the estimated states in (9) instead of the true states and still get an optimal design. In that case (9) and (10) gives the following LQG controller

$$\begin{aligned} \dot{x}_e &= (A - B L - K C) x_e + B L_r r + K y \\ u &= -L x_e + L_r r \end{aligned} \tag{11}$$

Here we choose L_r such that the static gain from *r* to *z* becomes an identity matrix:

```
Lr = (M.(B.L - A)^-1.B)^-1; MatrixForm[Lr]
```

$$\begin{pmatrix} 3.15189 & 0.369764 \\ 0.371095 & -3.14059 \end{pmatrix}$$

We represent the controller given by (11) in state-space form

```
contKalmanFilter = StateSpace[A - B.L - K.C,
  BlockMatrix[(B.Lr  K)], -L, BlockMatrix[(Lr  0_2×2)]];
```

The continuous Kalman filter above is transformed to discrete time for implementation purposes

```
discKalmanFilter =
  Chop[ToDiscreteTime[contKalmanFilter, Sampled → Period[0.1]], 10^-8];
```

We define a simple test signal, which corresponds to a unit step in the reference signal for the roll angle, while keeping the reference signal for the course angle and the sensors for measuring the output signals and equal to zero.

```
ry = Table[{1, 0, 0, 0}, {10}]^T;
```

```
SimulationPlot[discKalmanFilter,
  ry, PlotRange → All, PlotStyle → {Blue, Red}];
```

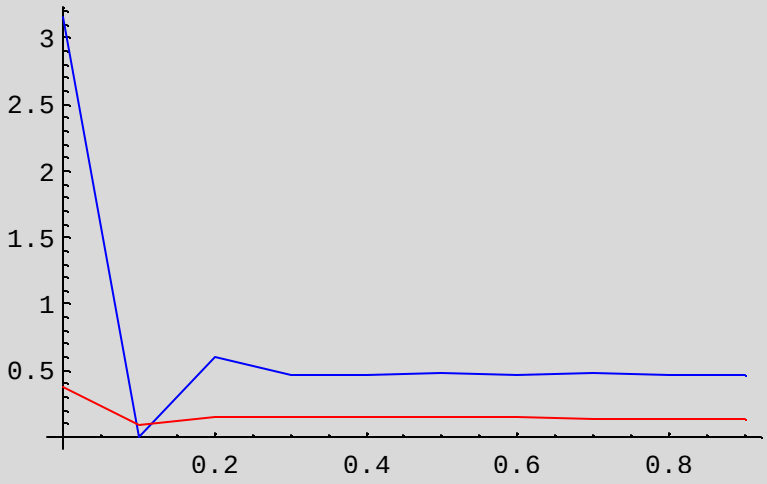

Figure 10. The controller outputs u_1 and u_2 when sensor signals are kept to zero and there is a unit step in the reference signal for the roll angle, .

■ Simulation and Code Generation

The system matrices of the discrete Kalman filter are declare as a global real constants (type information given to *Math-Code C++*)

```
Declare[
 Constant Real[_, _] Ac = discKalmanFilter[[1]],
 Constant Real[_] Bc = discKalmanFilter[[2]],
 Constant Real[_, _] Cc = discKalmanFilter[[3]],
 Constant Real Dc = discKalmanFilter[[4]]
]
```

Define the system functions (right hand sides of the controller equations).

```
f[Real[7] xe_, Real[4] in_] → Real[_] := Ac.xe + Bc.in;
g[Real[7] xe_, Real[4] in_] → Real[_] := Cc.xe + Dc. in;
```

We perform a simulation of the controller equations using these right hand sides. Initial condition:

```
x0 = {0, 0, 0, 0, 0, 0, 0};
```

FoldList can be used to compute an iteration $x_{n\ 1}$ $f[x_n, u_n]$, i.e., a discrete simulation:

```
xesim = FoldList[f, x0, ryT];
```

Remove the last value.

```
xesim = Drop[xesim, -1];
```

Compute the output from the calculated input and state

```
usim = MapThread[g, {xesim, ryT}];
```

A plot of output u_2 with samples instead of time on the x-axis:

```
ListPlot[(usimT)[[2]], PlotJoined → True, PlotStyle → Red];
```

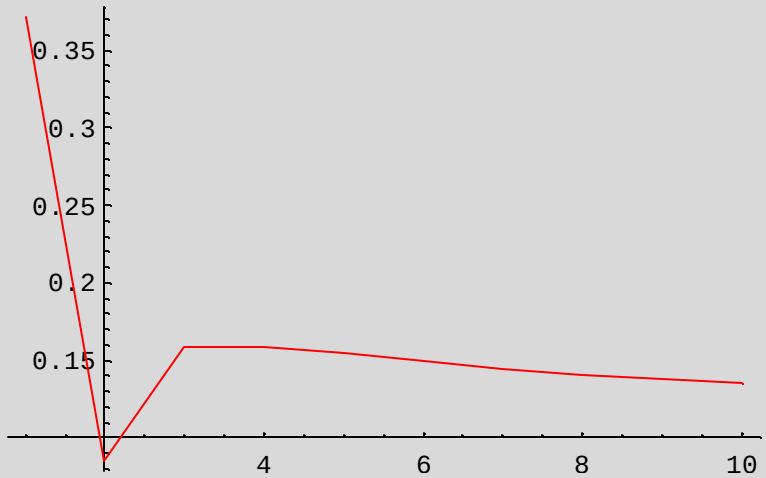

Figure 11. The controller output u_2 when sensor signals are kept to zero and there is a unit step in the reference signal for the roll angle, .

Create a loop that will be the main loop in the external code.

```
DiscreteSimulate[Real[n_, _] u_, Real[_] x0_] → Real[_, _] :=
  Module[
   {Real[_] x = x0,
    Real[n, 2] y = 0,
    Integer i},
   For[i = 1, i ≤ n, i++,
    y[[i]] = g[x, u[[i]]];
    x = f[x, u[[i]]]
   ];
   y
  ];
```

From construction, the function f and g can be expanded to remove the dot-products and to reduce the computational effort. This is especially good when the system matrices contain many zeroes. The expansion is achieved using the `EvaluateFunctions` option.

Generate the code and compile it. The functions f and g will be expanded and subexpression optimized before being code generated.

```
BuildCode[EvaluateFunctions → {f, g}]
```

```
Successful compilation to C++: 3 function(s)
```

We inspect the resulting C++ code

```
Run["notepad Global.cc"];
```

Install the compiled code into *Mathematica* and test if the same simulation result as in Figure 11 is achieved.

```
InstallCode[];
```

```
Global is installed.
```

```
usim2 = DiscreteSimulate[ry^T, x_0];
```

A comparison between controller outputs computed by *Mathematica* and by the external code. The maximum absolute error in the sequence

```
Max[Abs[usim - usim2]]
```

$$3.33067 \times 10^{-16}$$

A plot of the simulation when the compiled code is use.

```
ListPlot[(usim2^T)[[2]], PlotJoined → True, PlotStyle → Red];
```

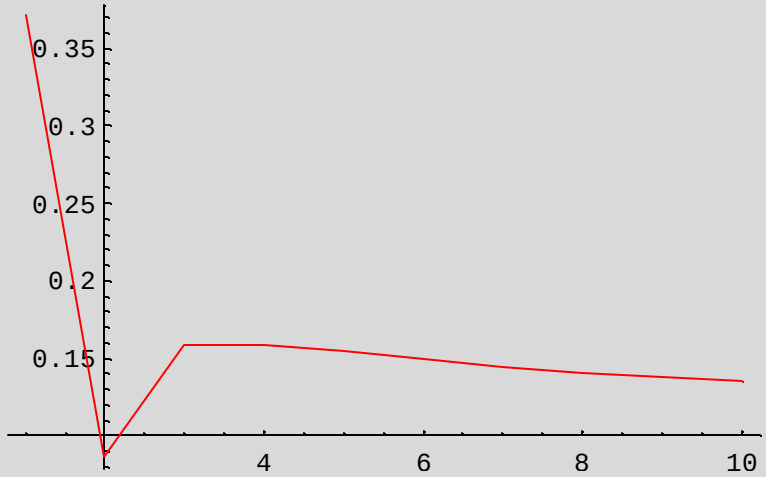

Figure 12. The same plot as in Figure 11 but computed by the external code.

We observe that the external simulation gives the same result as the simulation computed within *Mathematica*.

■ Clean-up

Uninstall the code and delete the temporary files.

```
UninstallCode[];
CleanMathCodeFiles[Confirm → False, CleanAllBut → {}];
```

Delete the temporary directory and all files it contains.

```
SetDirectory[ToFileName[".."]];
DeleteDirectory["Tmp", DeleteContents → True];
```

4 Conclusions

In this notebook we have demonstrated the use of *Mathematica* in modeling of dynamic systems and controller design. The symbolic capabilities of *Mathematica* are very useful for deriving dynamic equations according to the Lagrangian formalism. Furthermore, rewriting these equations in state-space form, suitable for controller design, is also easily done using `Solve`. A unique solution is always obtained since the second order derivatives to solve for always appear linearly in the equations derived from the partial differential equation (6) that the Lagrangian of the system has to satisfy.

We have also shown how efficient C++ code for simulation can be generated using the application package *MathCode C++*. The pendulum/seesaw example is only a toy example compared to many industrial applications, which motivates the efforts to be able to automatically generate C++ code that can be compiled and run ouside *Mathematica* for increased performance.

In the fighter aircraft example we illustrated how *Mathematica* can be used as a prototyping environment for controller design. First the model of the system is analyzed and different controllers can be evaluated. When a satisfactory solution has been found *MathCode C++* can be used to generate stand-alone C++ code that can be used in the real application.

References

[1] T. Glad and L. Ljung. *Reglerteori - Flervariabla och olinjära metoder*. Studentlitteratur, 1997.

[2] L. Ljung and T. Glad. *Modeling of Dynamic Systems*. Prentice Hall, 1994.

[3] H. Goldstein. *Classical Mechanics*. Addison-Wesley, second edition, 1980.

[4] J. M. Maciejowski. *Multivariable Feedback Design*. Electronic Systems Engineering Series. Addison-Wesley, 1989.

[5] K. Zhou, J. C. Doyle, and K. Glover. *Robust and Optimal Control*. Prentice Hall, 1996.

Initialization

In this section we load a number of add-on packages, define auxiliary functions and set up the environment for C++ code generation.

Load the Control System Professional (CSP) package:

```
Needs["ControlSystems`Master`"];
```

Load the *MathCode C++* package:

```
Needs["MathCode`"];
```

```
MathCode C++ ver  1.10 loaded.
```

Since a number of temporary files will be generated by *MathCode C++* we create a temporary directory and set this to the current working directory.

```
CreateDirectory["Tmp"];
SetDirectory[ToFileName[{".", "Tmp"}]];
```

This function will be used to measure the time for external processes. We cannot rely on the `Timing` command since it includes only CPU time spent in the *Mathematica* kernel. It does not include time spent in external processes connected via *MathLink*.

```
SetAttributes[AbsTime, HoldFirst];
AbsTime[x_] :=
  Module[{start, res},
   start = AbsoluteTime[];
   res = x;
   {(AbsoluteTime[] - start) Second, res}
  ];
```

Notation for colors:

```
Needs["Graphics`Colors`"]
```

Set default color list for multiple plots:

```
SetOptions[Plot, PlotStyle → {Blue, Red, Green, Cyan, Magenta, Yellow}];
```

Load my own package which gives improved notation for matrix operations:

```
Needs["MatrixNotation`"]
```

Turn of the real-time spell checking

```
Off[General::"spell1", General::"spell"]
```