Modeling Approaches

- State-space Approach
  - Example: Pendulum
- Block Diagram Approach
  - Example: Pendulum
- Component-Oriented Approach
  - Example: Pendulum
- Exercise: Tank with Controller
- Exercise: DC Motor with Controller

Modeling Approaches

State-space Approach

State-space Approach

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

$x \in \mathbb{R}^n$	state vector
$u \in \mathbb{R}^p$	input vector
$y \in \mathbb{R}^q$	output vector
$A \in \mathbb{R}^{n \times n}$	state matrix
$B \in \mathbb{R}^{n \times p}$	input matrix
$C \in \mathbb{R}^{q \times n}$	output matrix
$D \in \mathbb{R}^{q \times p}$	feedthrough matrix

- continuous time-invariant
- linear system in terms of states and inputs

State-space Approach

$$\dot{x} = f(x, u)$$
$$y = h(x, u)$$

$x \in \mathbb{R}^n$	state vector
$u \in \mathbb{R}^p$	input vector
$y \in \mathbb{R}^q$	output vector
$f: (x, y) \rightarrow \mathbb{R}^n$	state equation
$h: (x, y) \rightarrow \mathbb{R}^q$	output equation

- continuous time-invariant
- nonlinear system in terms of states and inputs

State-space Approach

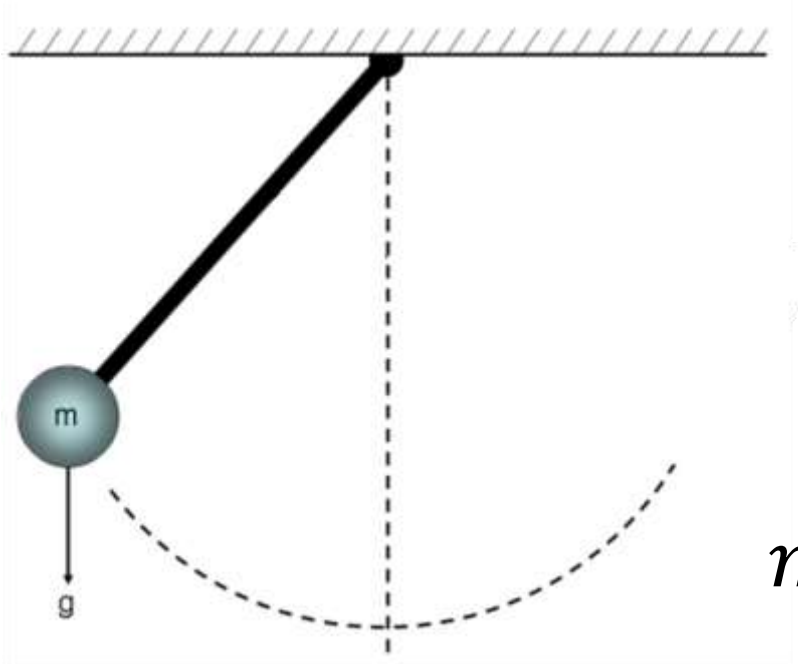
$$\dot{x} = f(t, x, u)$$
$$y = h(t, x, u)$$

$t \in \mathbb{R}$	time
$x \in \mathbb{R}^n$	state vector
$u \in \mathbb{R}^p$	input vector
$y \in \mathbb{R}^q$	output vector
$f: (x, y) \rightarrow \mathbb{R}^n$	state equation
$h: (x, y) \rightarrow \mathbb{R}^q$	output equation

- continuous time-variant
- nonlinear system in terms of states and inputs

State-space Approach

- Classic example: pendulum

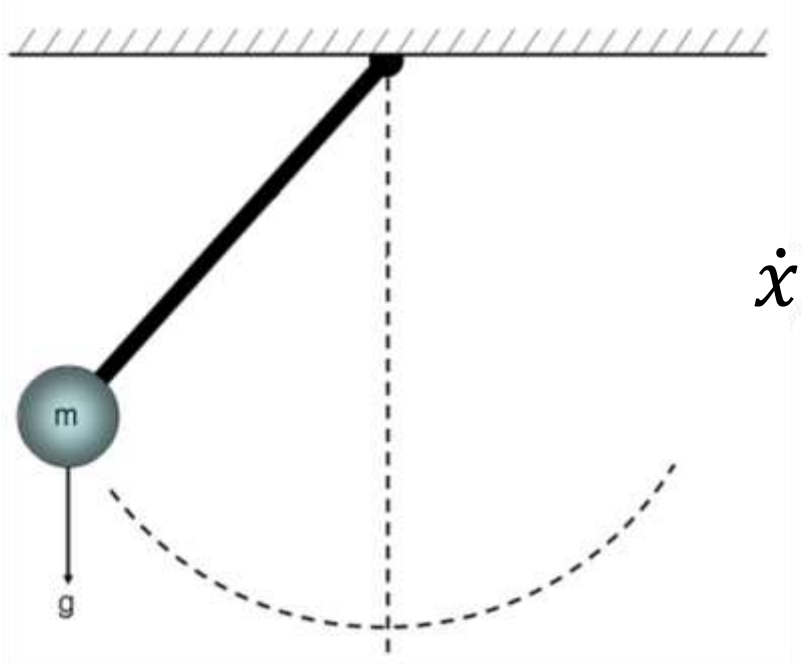


m	mass
l	length
g	acceleration of gravity
k	damping coefficient

$$ml^2\ddot{\theta} = -mgl\sin\theta - kl\dot{\theta}$$

State-space Approach

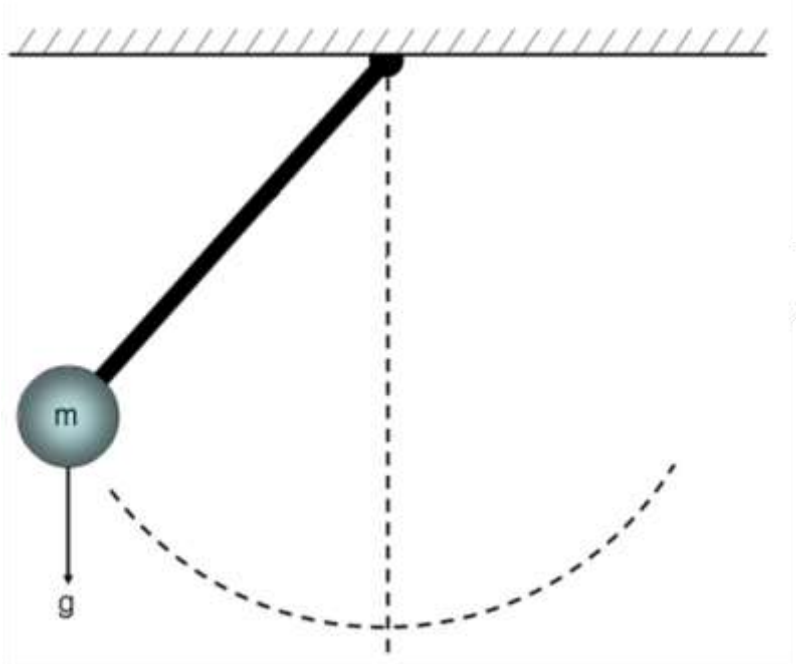
- Classic example: pendulum



$$\dot{x} = \begin{pmatrix} x_2 \\ -\frac{g}{l}\sin x_1 - \frac{k}{ml}x_2 \end{pmatrix}$$

State-space Approach

- Classic example: pendulum



$$A = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l}\cos x_1 & -\frac{k}{ml} \end{bmatrix}$$

State-space Approach

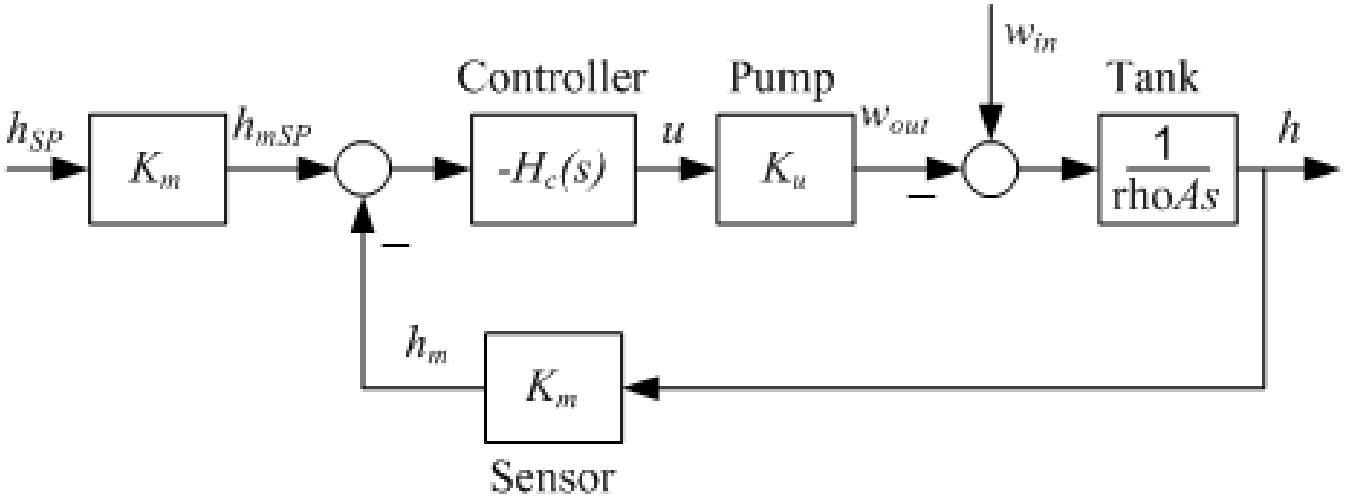
- (-) No graphical representation
- (-) The system decomposition does not correspond to the "natural" physical system structure
- (-) Breaking down into subsystems is difficult if the connections are not of input/output type.
- (-) Two connected state-space subsystems do not usually give a state-space system automatically.
- (+) Easy to handle for computer systems from the previous century, before symbolic transformations for equation systems became efficient enough

Modeling Approaches

Block Diagram Approach

Block Diagram Approach

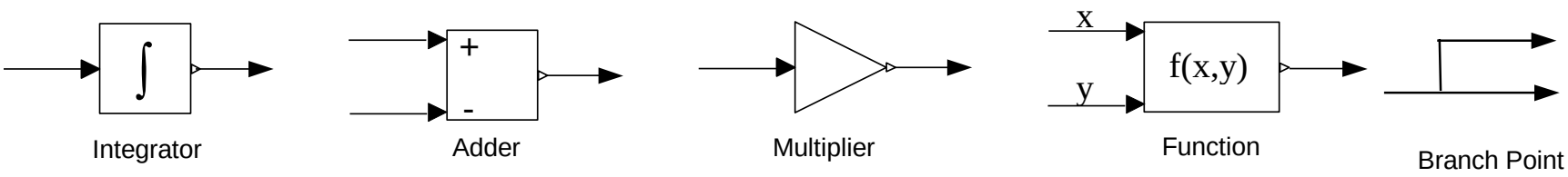
- Graphical modelling
- Signal-flow model
- Fixed input/output dependencies
- Usually used in control engineering



Block Diagram Approach

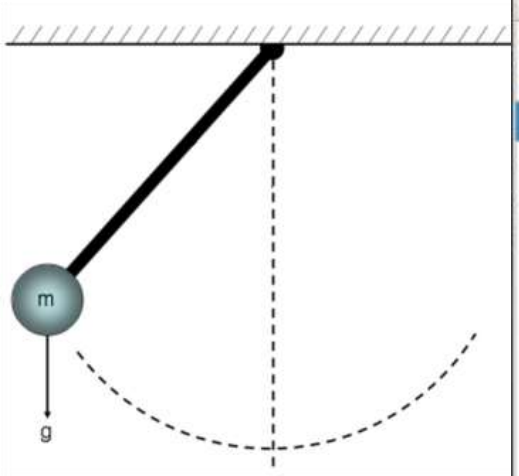
- Special case of model components: the causality of each interface variable has been fixed to either *input* or *output*

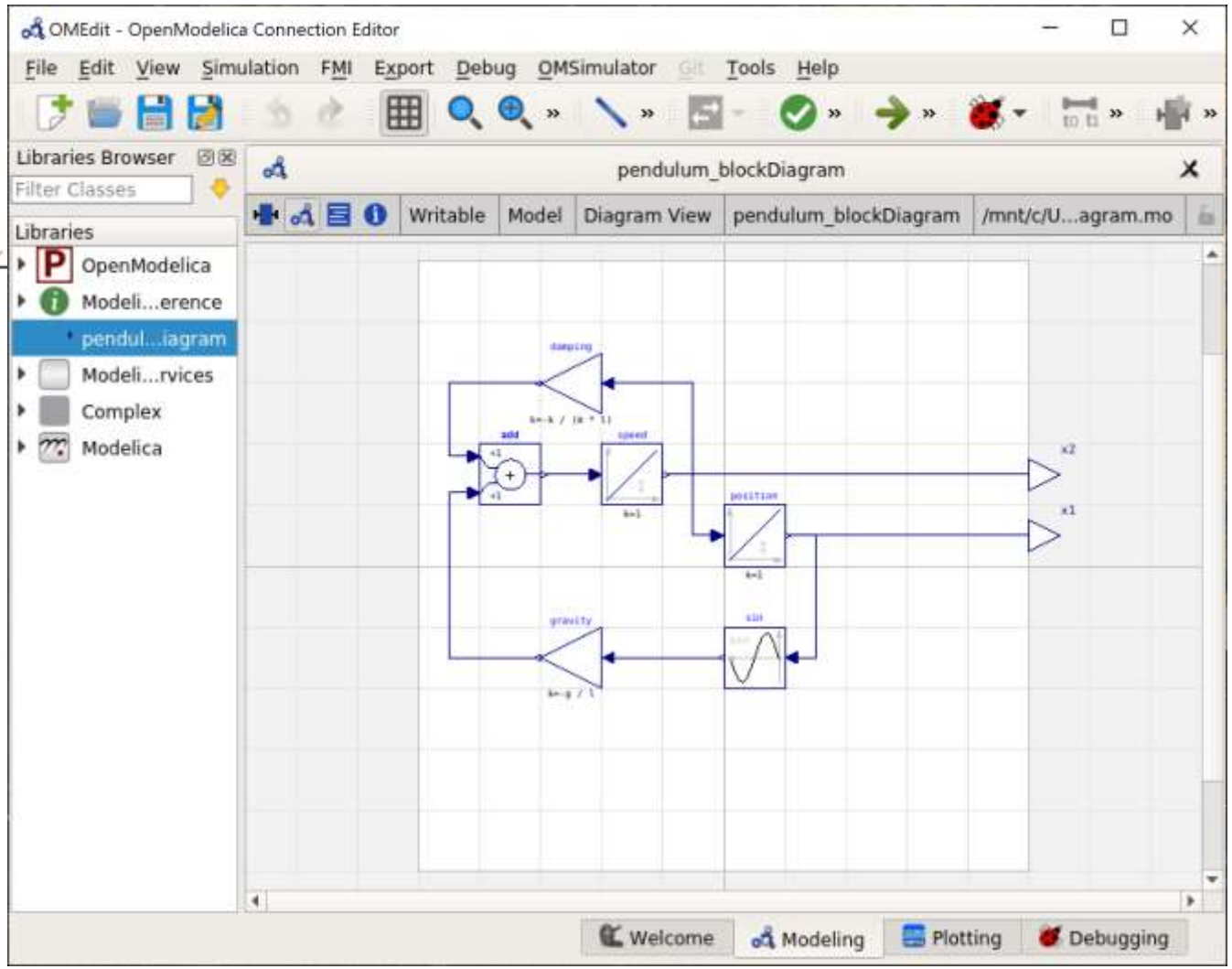
Typical Block diagram model components:



- Conceptual equivalent to FMUs

Block Diagram Approach





Block Diagram Approach

- (-) The system decomposition topology does not correspond to the "natural" physical system structure
- (-) Hard work of manual conversion of equations into signal-flow representation
- (-) Physical models become hard to understand in signal representation
- (-) Small model changes (e.g. compute positions from force instead of force from positions) requires redesign of whole model
- (+) Block diagram modelling works well for control systems since they are signal-oriented rather than "physical“
- (+) Graphical modelling

Modeling Approaches

Component-Oriented Approach

Component-Oriented Approach

- Define the system briefly
  - What kind of system is it?
  - What does it do?
- Decompose the system into its most important components
  - Define communication, i.e., determine interactions
  - Define interfaces, i.e., determine the external ports/connectors
  - Recursively decompose model components of “high complexity”
- Formulate new model classes when needed
  - Declare new model classes.
  - Declare possible base classes for increased reuse and maintainability

Top-Down versus Bottom-up Modelling

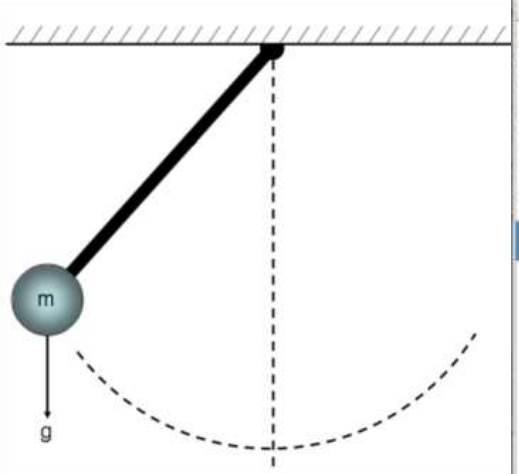
- **Top Down**: Start designing the overall view. Determine what components are needed.

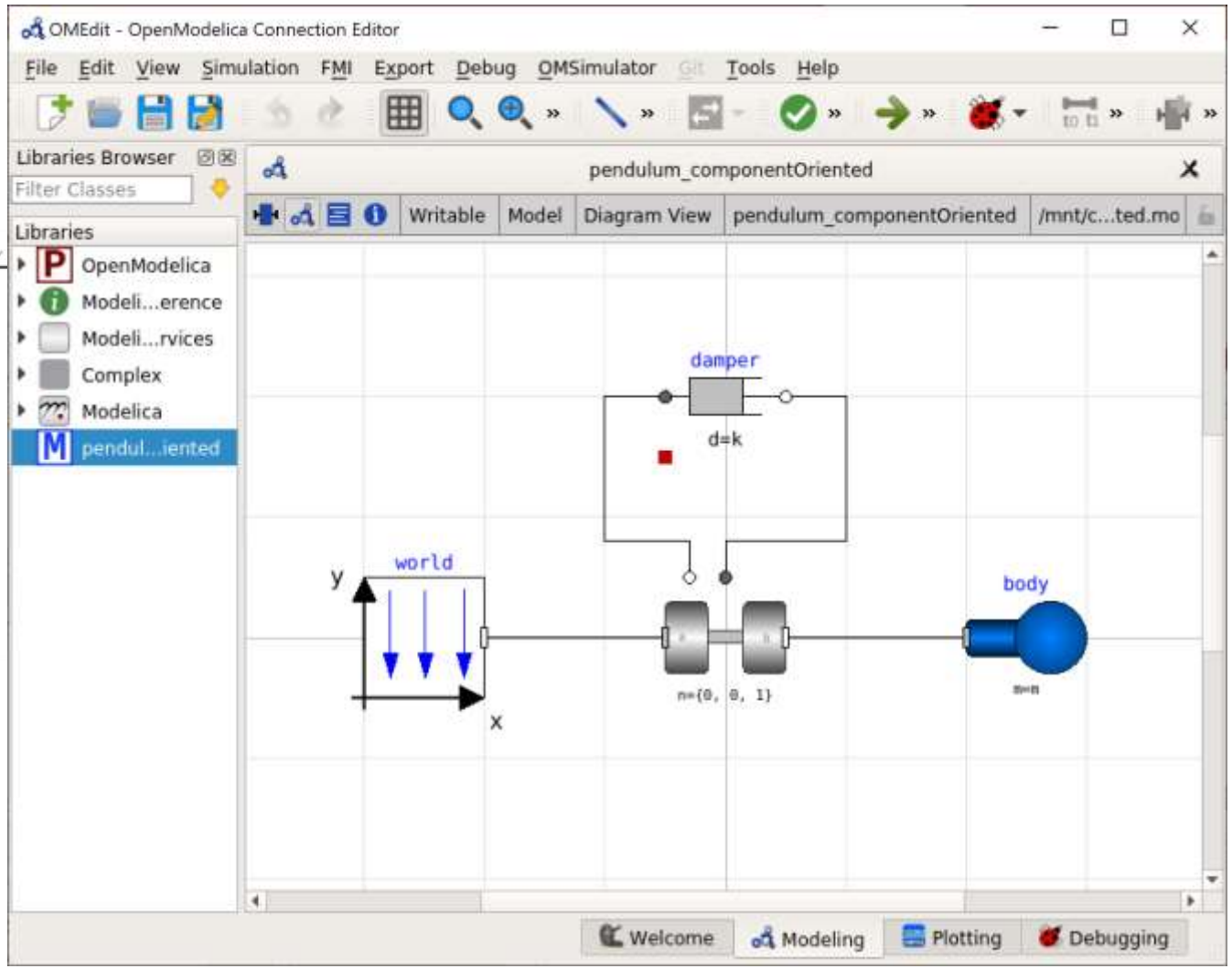
- **Bottom-Up**: Start designing the components and try to fit them together later.

Using Library Model Components

- **Decompose** into subsystems
- Sketch **communication**
- Design **subsystems** models by connecting library component models
- Simulate!

Component-Oriented Approach



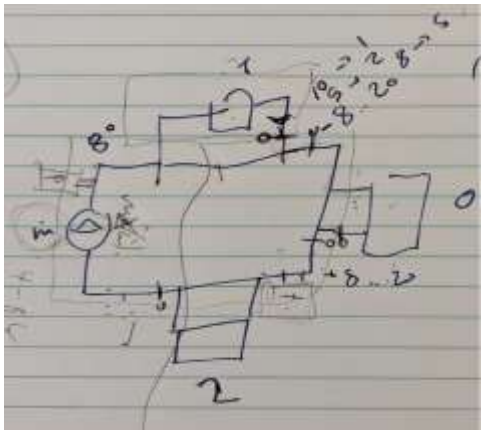


Component-Oriented Approach

- (-) Huge system of equations
- (+) Works well for control systems since physical models can be inverted and linearized
- (+) Graphical modelling
- (+) The system decomposition correspond to the "natural" physical system structure
- (+) Easy to connect two systems which each other
- (+) High reusability of components, because of acausal and object-oriented modelling
- (-) Sometimes difficult to debug

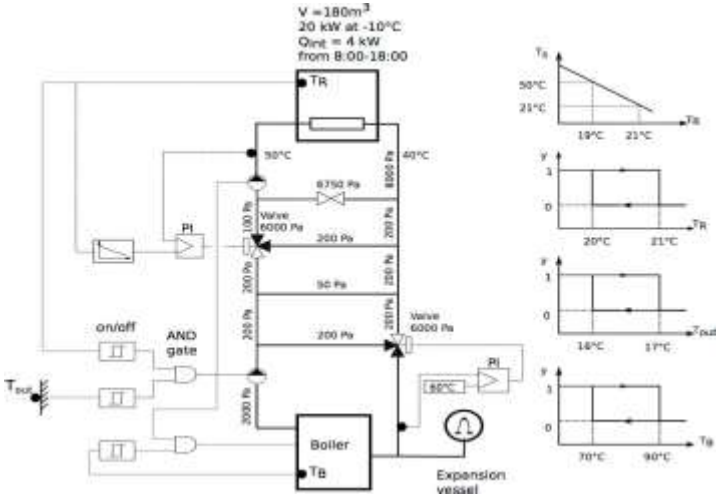
Advice for Building Large System Models

1. **Understand the problem**:
    1. What question do you want to answer?
    2. Know what you want to model.
        1. Draw system schematics.
        2. Identify control input.
        3. Draw the control loops.
        4. Determine the control sequences.



2. **Compartmentalize**: Split the system into subcomponents that can be tested in isolation.

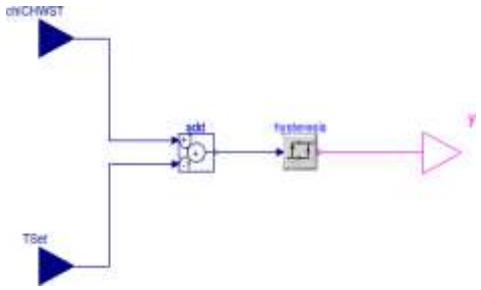
3. **Implement**: Now, and only now, start implementing in software.
    1. Document and build test cases as you go along.
       Errors are easy to detect in small models, but hard in large models. If you add unit tests, you make sure what has been tested remains intact as the model evolves.
    2. Assemble the subcomponents to build the full model.



Slide content acknowledgement: LBL Buildings intro tutorial M. Wetter et al

Advice for Building Large System Models
How do you debug a large system model?

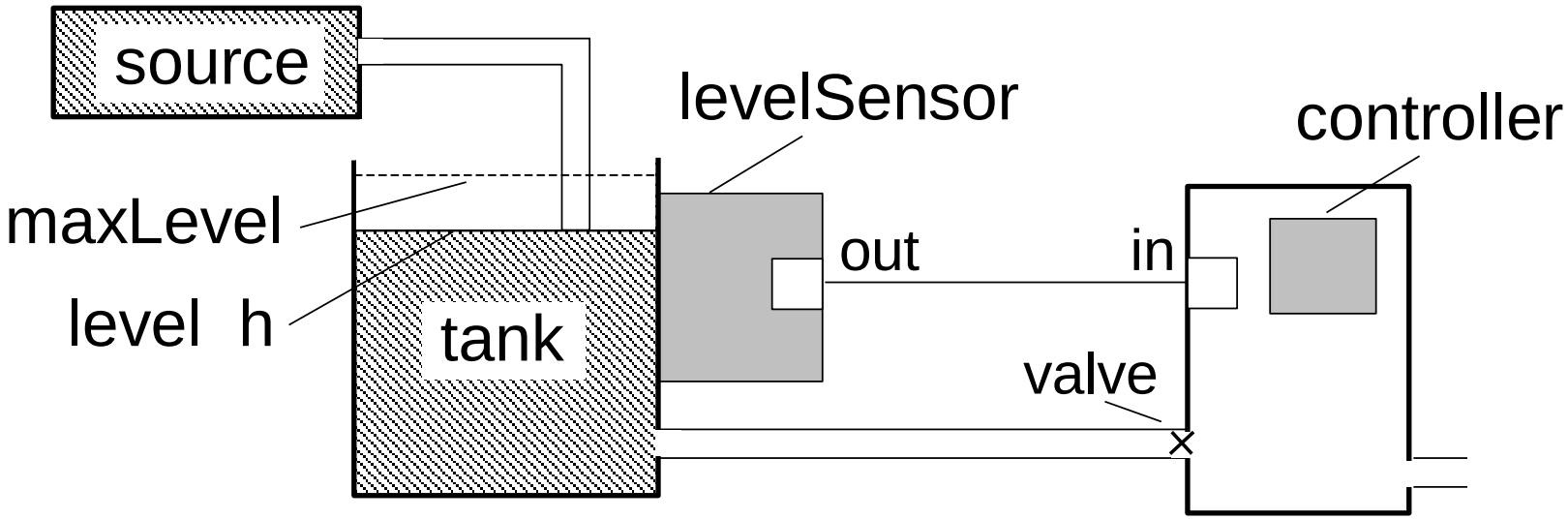
- **Split** the model into **small models** — or better, architect the large model from the beginning to be based on smaller models
- **Test** the smaller models for well known conditions.
- **Add** smaller models to unit tests.
- The OpenModelica debugger can be used to locate some bugs, and to find dependencies on variables



Slide content acknowledgement: LBL Buildings intro tutorial M. Wetter et al

Exercise: Tank with Controller

Exercise: Tank with Controller



Tank System Model FlatTank – No Graphical Structu

- No component structure
- Just flat set of equations
- Straight-forward but less flexible, no graphical representation

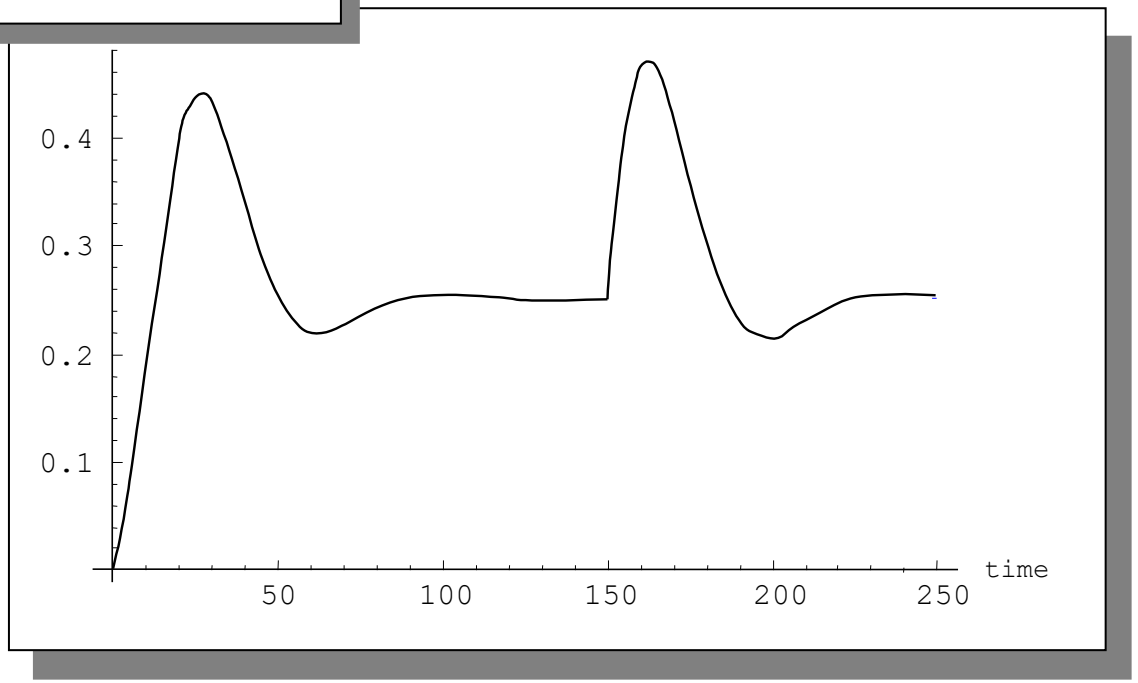
```
model FlatTank
  // Tank related variables and parameters
  parameter Real flowLevel(unit="m3/s")=0.02;
  parameter Real area(unit="m2")        =1;
  parameter Real flowGain(unit="m2/s") =0.05;
  Real           h(start=0,unit="m")    "Tank level";
  Real           qInflow(unit="m3/s")   "Flow through input valve";
  Real           qOutflow(unit="m3/s") "Flow through output valve";
  // Controller related variables and parameters
  parameter Real K=2                     "Gain";
  parameter Real T(unit="s")= 10         "Time constant";
  parameter Real minV=0, maxV=10;     // Limits for flow output
  Real           ref = 0.25  "Reference level for control";
  Real           error       "Deviation from reference level";
  Real           outCtr      "Control signal without limiter";
  Real           x;          "State variable for controller";
equation
  assert(minV>=0,"minV must be greater or equal to zero");//
  der(h) = (qInflow-qOutflow)/area;   // Mass balance equation
  qInflow  = if time>150 then 3*flowLevel else flowLevel;
  qOutflow = LimitValue(minV,maxV,-flowGain*outCtr);
  error  = ref-h;
  der(x) = error/T;
  outCtr = K*(error+x);
end FlatTank;
```

Simulation of FlatTank System

- Flow increase to flowLevel at time 0
- Flow increase to 3*flowLevel at time 150

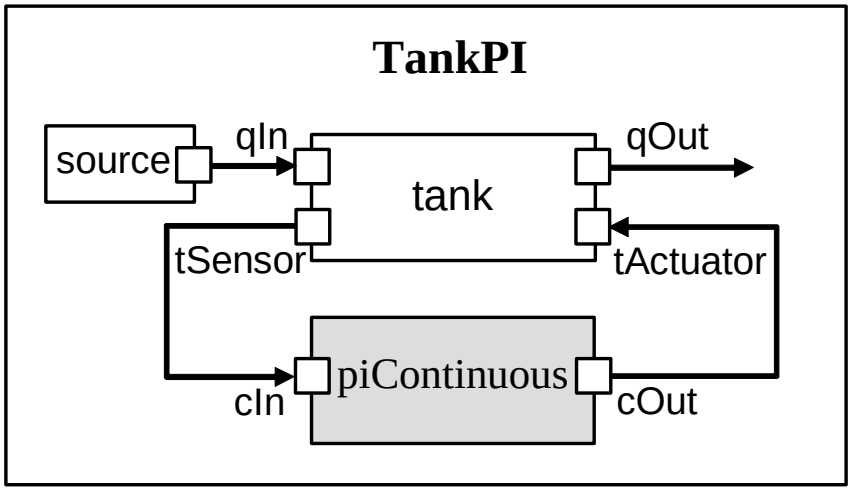
```
simulate(FlatTank, stopTime=250)

plot(h, stopTime=250)
```



Object Oriented Component-Based Approach
Tank System with Three Components

- Liquid source
- Continuous PI controller
- Tank




```
model TankPI
  LiquidSource           source(flowLevel=0.02);
  PIcontinuousController piContinuous(ref=0.25);
  Tank                   tank(area=1);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, piContinuous.cOut);
  connect(tank.tSensor, piContinuous.cIn);
end TankPI;
```

Tank model

- The central equation regulating the behavior of the tank is the mass balance equation (input flow, output flow), assuming constant pressure

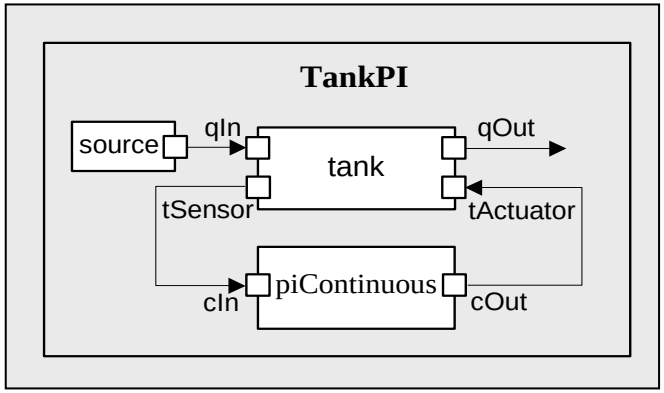
```
model Tank
  ReadSignal  tSensor   "Connector, sensor reading tank level (m)";
  ActSignal   tActuator "Connector, actuator controlling input flow";
  LiquidFlow  qIn       "Connector, flow (m3/s) through input valve";
  LiquidFlow  qOut      "Connector, flow (m3/s) through output valve";
  parameter Real area(unit="m2")        = 0.5;
  parameter Real flowGain(unit="m2/s") = 0.05;
  parameter Real minV=0, maxV=10; // Limits for output valve flow
  Real h(start=0.0, unit="m") "Tank level";
equation
  assert(minV>=0,"minV - minimum Valve level must be >= 0 ");//
  der(h)       = (qIn.lflow-qOut.lflow)/area;   // Mass balance
equation
  qOut.lflow  = LimitValue(minV,maxV,-flowGain*tActuator.act);
  tSensor.val = h;
end Tank;
```

Connector Classes and Liquid Source Model for Tank System

```
connector ReadSignal "Reading fluid level"
  Real val(unit="m");
end ReadSignal;

connector ActSignal  "Signal to actuator
 for setting valve position"
  Real act;
end ActSignal;

connector LiquidFlow  "Liquid flow at inlets or outlets"
  Real lflow(unit="m3/s");
end LiquidFlow;
```




```
model LiquidSource
  LiquidFlow qOut;
  parameter  flowLevel = 0.02;
equation
  qOut.lflow = if time>150 then 3*flowLevel else flowLevel;
end LiquidSource;
```

Continuous PI Controller for Tank System

- error = (reference level – actual tank level)
- T is a time constant
- x is controller state variable
- K is a gain factor

$$\frac{dx}{dt} = \frac{error}{T}$$

$$outCtr = K * (error + x)$$

Integrating equations gives Proportional & Integrative (PI)

$$outCtr = K * (error + \int \frac{error}{T} dt)$$

base class for controllers – to be defined

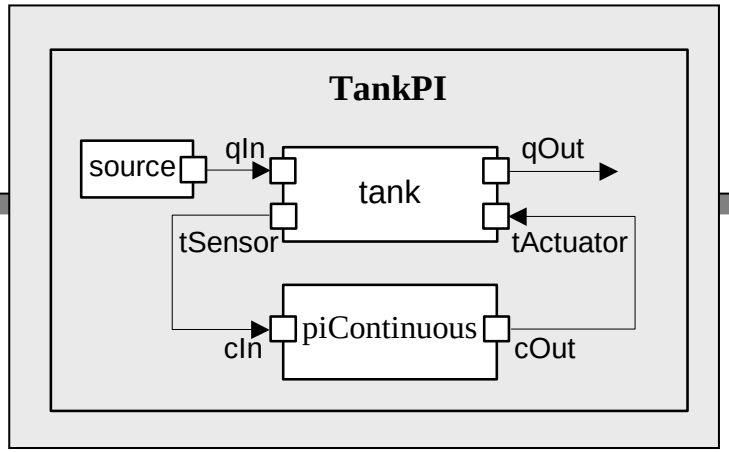
```
model PIcontinuousController
  extends BaseController(K=2,T=10);
  Real  x  "State variable of continuous PI controller";
equation
  der(x) = error/T;
  outCtr = K*(error+x);
end PIcontinuousController;
```

error – to be defined in controller base class

The Base Controller – A Partial Model

```
partial model BaseController
  parameter Real Ts(unit="s")=0.1
    "Ts - Time period between discrete samples - discrete sampled";
  parameter Real K=2              "Gain";
  parameter Real T=10(unit="s")  "Time constant - continuous";
  ReadSignal     cIn              "Input sensor level,  connector";
  ActSignal      cOut             "Control to actuator, connector";
  parameter Real ref              "Reference level";
  Real           error            "Deviation from reference level";
  Real           outCtr           "Output control signal";
equation
  error    = ref-cIn.val;
  cOut.act = outCtr;
end BaseController;
```

error = difference betwen reference level and actual tank level from cIn connector

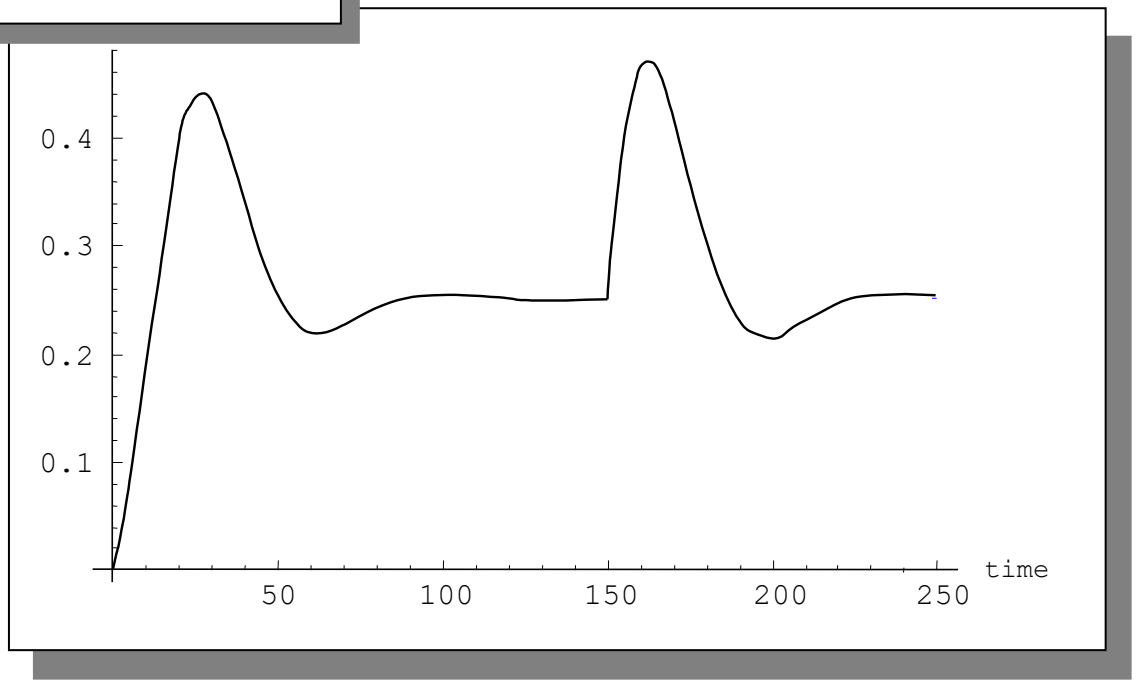


Simulate Component-Based Tank System

- As expected (same equations), TankPI gives the same result as the flat model FlatTank

```
simulate(TankPI, stopTime=250)

plot(h, stopTime=250)
```



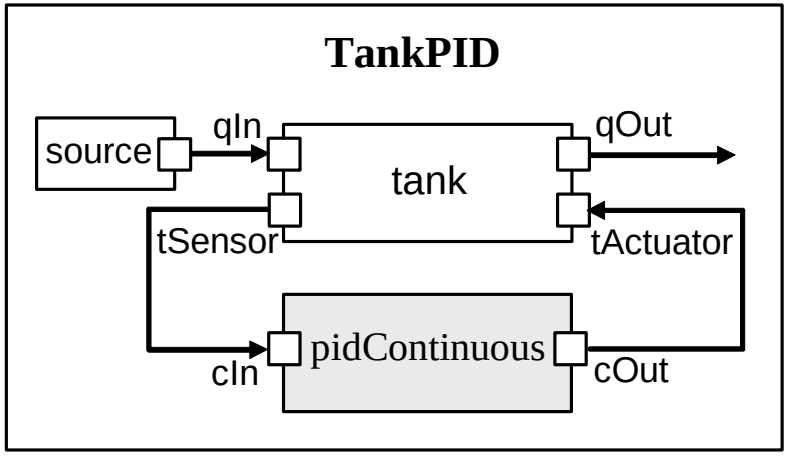
Flexibility of Component-Based Models

- Exchange of components possible in a component-based model

- Example:
  Exchange the PI controller component for a PID controller component

Tank System with Continuous PID Controller Instead of Continuous PI Controller

- Liquid source
- Continuous PID controller
- Tank




```
model TankPID
  LiquidSource              source(flowLevel=0.02);
  PIDcontinuousController pidContinuous(ref=0.25);
  Tank                      tank(area=1);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, pidContinuous.cOut);
  connect(tank.tSensor, pidContinuous.cIn);
end TankPID;
```

Continuous PID Controller

- error = (reference level – actual tank level)
- T is a time constant
- x, y are controller state variables
- K is a gain factor

$$\frac{dx}{dt} = \frac{error}{T}$$

$$y = T\frac{d\,error}{dt}$$

$$outCtr = K*(error + x + y)$$

Integrating equations gives Proportional & Integrative & Derivative(PID)

$$outCtr = K*(error + \int \frac{error}{T} dt + T\frac{d\,error}{dt})$$

base class for controllers – to be defined

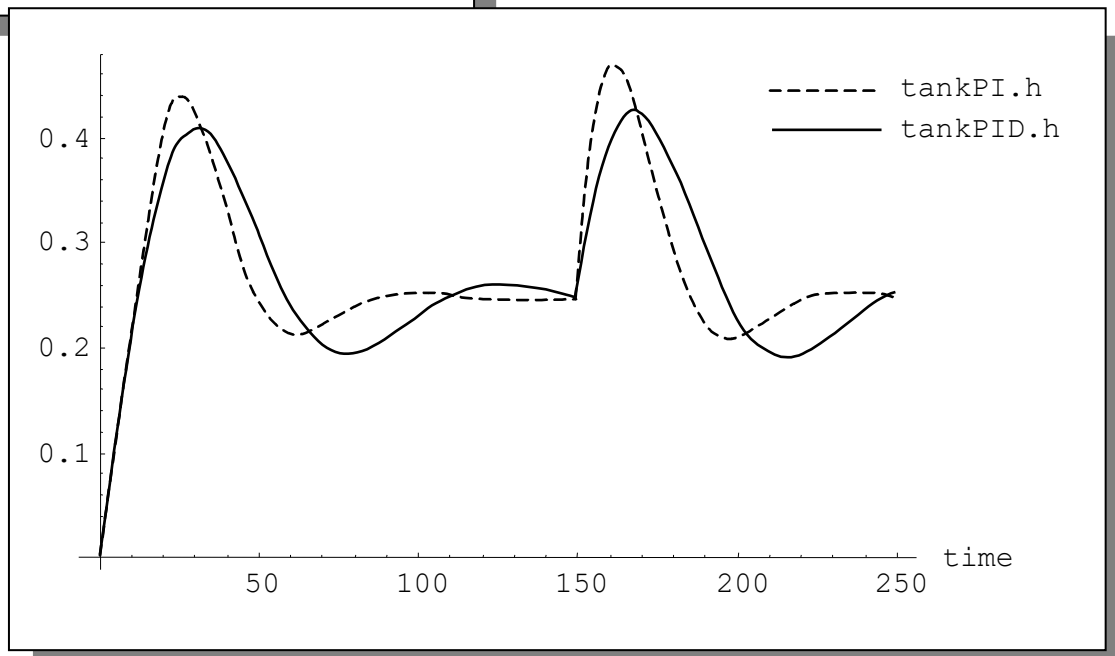
```
model PIDcontinuousController
  extends BaseController(K=2,T=10);
  Real  x; // State variable of continuous PID controller
  Real  y; // State variable of continuous PID controller
equation
  der(x) = error/T;
  y      = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```

Simulate TankPID and TankPI Systems

- TankPID with the PID controller gives a slightly different result compared to the TankPI model with the PI controller

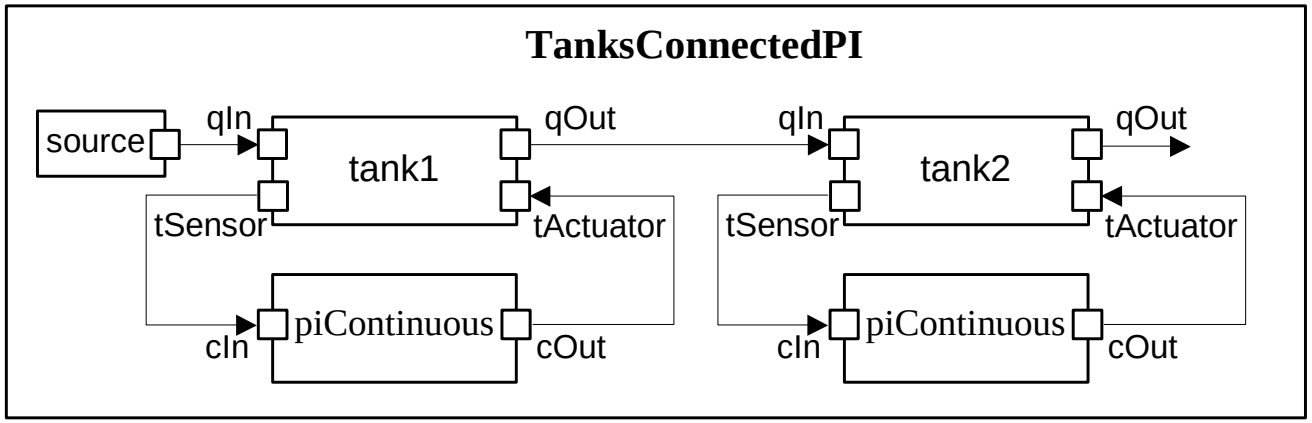
```
simulate(compareControllers, stopTime=250)

plot({tankPI.h,tankPID.h})
```



Two Tanks Connected Together

- Flexibility of component-based models allows connecting models together



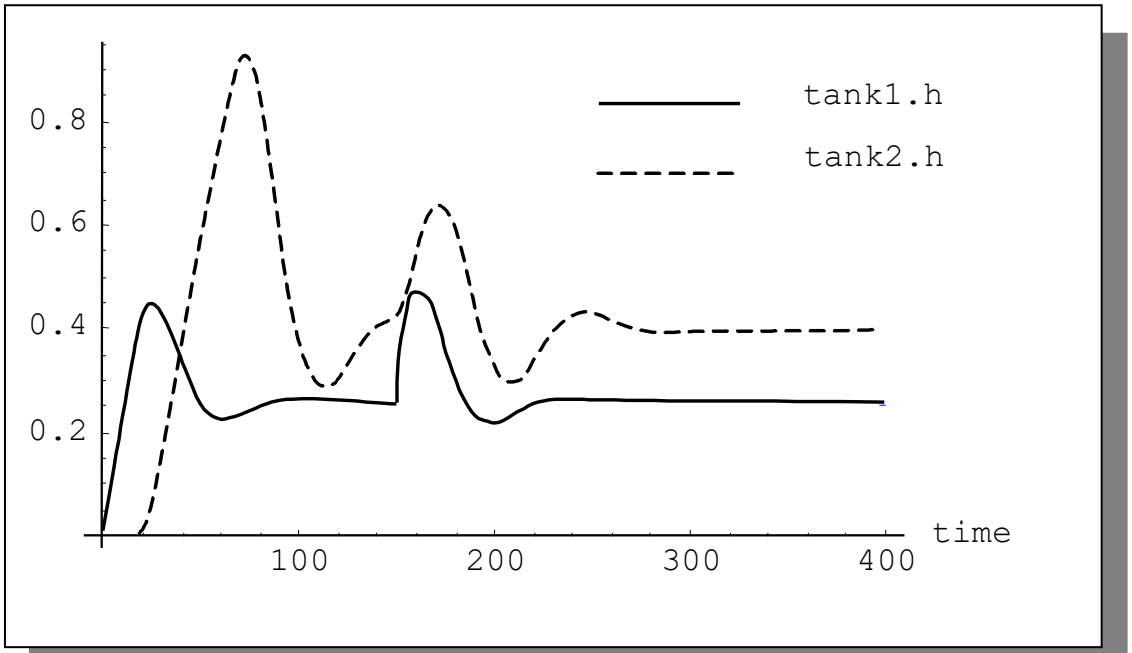

```
model TanksConnectedPI
  LiquidSource  source(flowLevel=0.02);
  Tank          tank1(area=1), tank2(area=1.3);;
  PIcontinuousController piContinuous1(ref=0.25), piContinuous2(ref=0.4);
equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tActuator,piContinuous1.cOut);
  connect(tank1.tSensor,piContinuous1.cIn);
  connect(tank1.qOut,tank2.qIn);
  connect(tank2.tActuator,piContinuous2.cOut);
  connect(tank2.tSensor,piContinuous2.cIn);
end TanksConnectedPI;
```

Simulating Two Connected Tank Systems

- Fluid level in tank2 increases after tank1 as it should
- Note: tank1 has reference level 0.25, and tank2 ref level 0.4

```
simulate(TanksConnectedPI, stopTime=400)

plot({tank1.h,tank2.h})
```



Exchange: Either PI Continuous or PI Discrete Controller

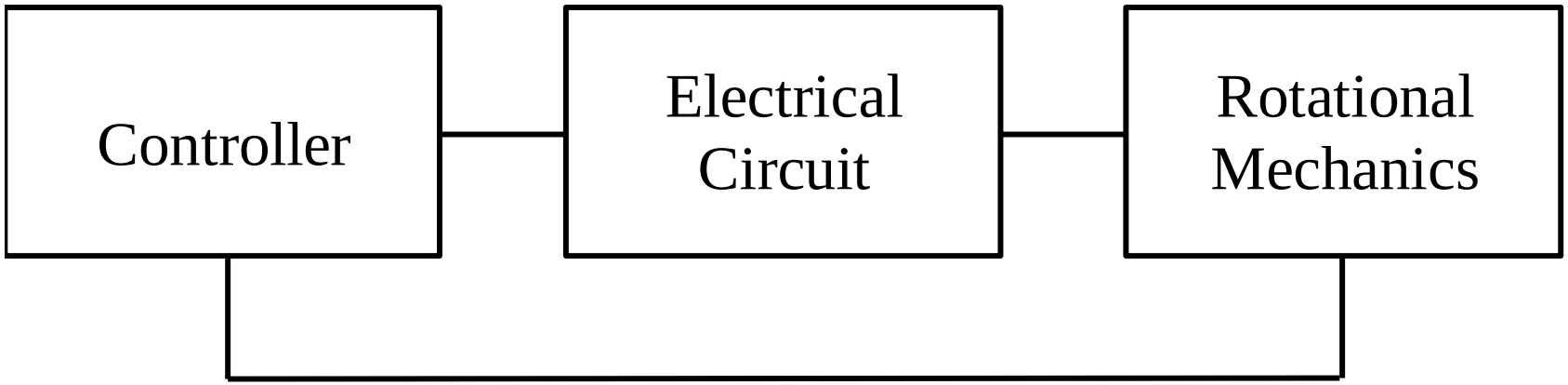
```
partial model BaseController
  parameter Real Ts(unit = "s") = 0.1   "Time period between discrete samples";
  parameter Real K = 2                  "Gain";
  parameter Real T(unit = "s") = 10     "Time constant";
  ReadSignal cIn                        "Input sensor level, connector";
  ActSignal  cOut                       "Control to actuator, connector";
  parameter Real ref                    "Reference level";
  Real error                            "Deviation from reference level";
  Real outCtr                           "Output control signal";
equation
  error = ref - cIn.val;
  cOut.act = outCtr;
end BaseController;
```

```
model PIDcontinuousController
  extends BaseController(K = 2, T = 10);
  Real  x;
  Real  y;
equation
  der(x) = error/T;
  y      = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```

```
model PIdiscreteController
  extends BaseController(K = 2, T = 10);
  discrete Real x;
equation
  when sample(0, Ts) then
    x = pre(x) + error * Ts / T;
    outCtr = K * (x+error);
  end when;
end PIdiscreteController;
```

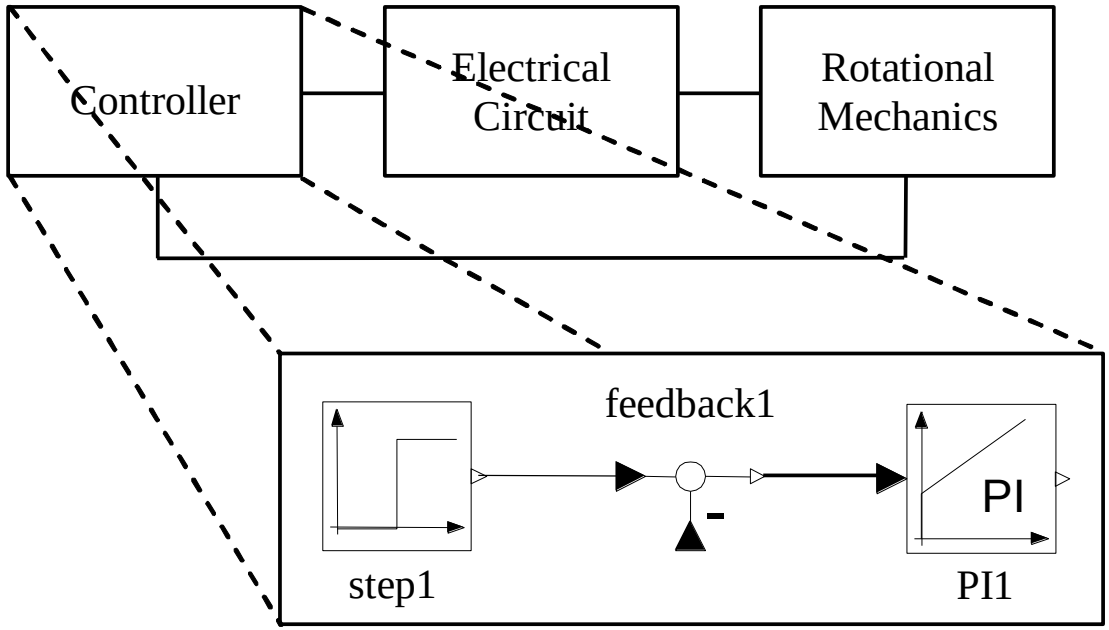
Exercise: DC Motor with Controller

Decompose into Subsystems and Sketch Communication – DC-Motor Servo Example



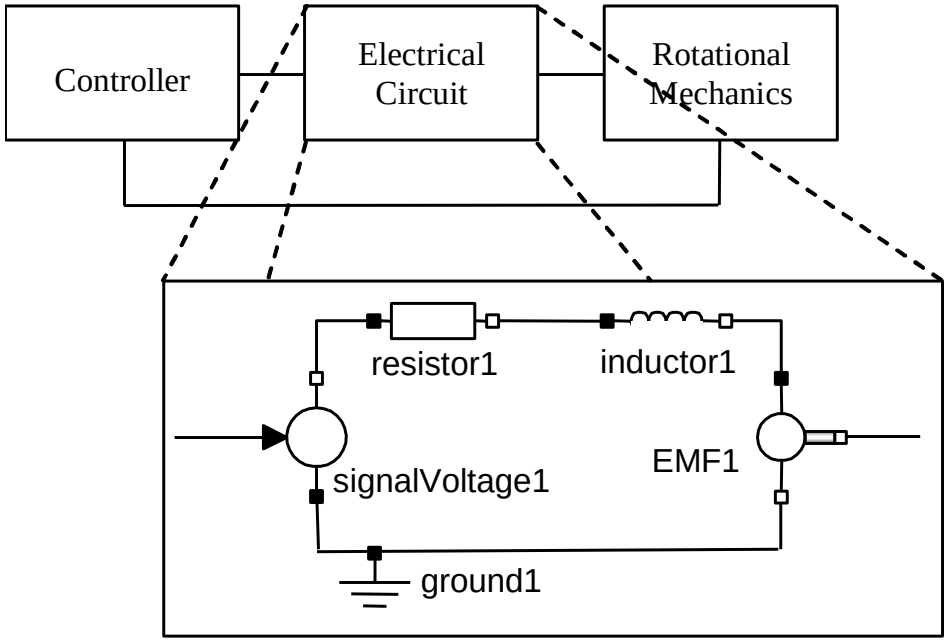
The DC-Motor servo subsystems and their connections

Modeling the Controller Subsystem



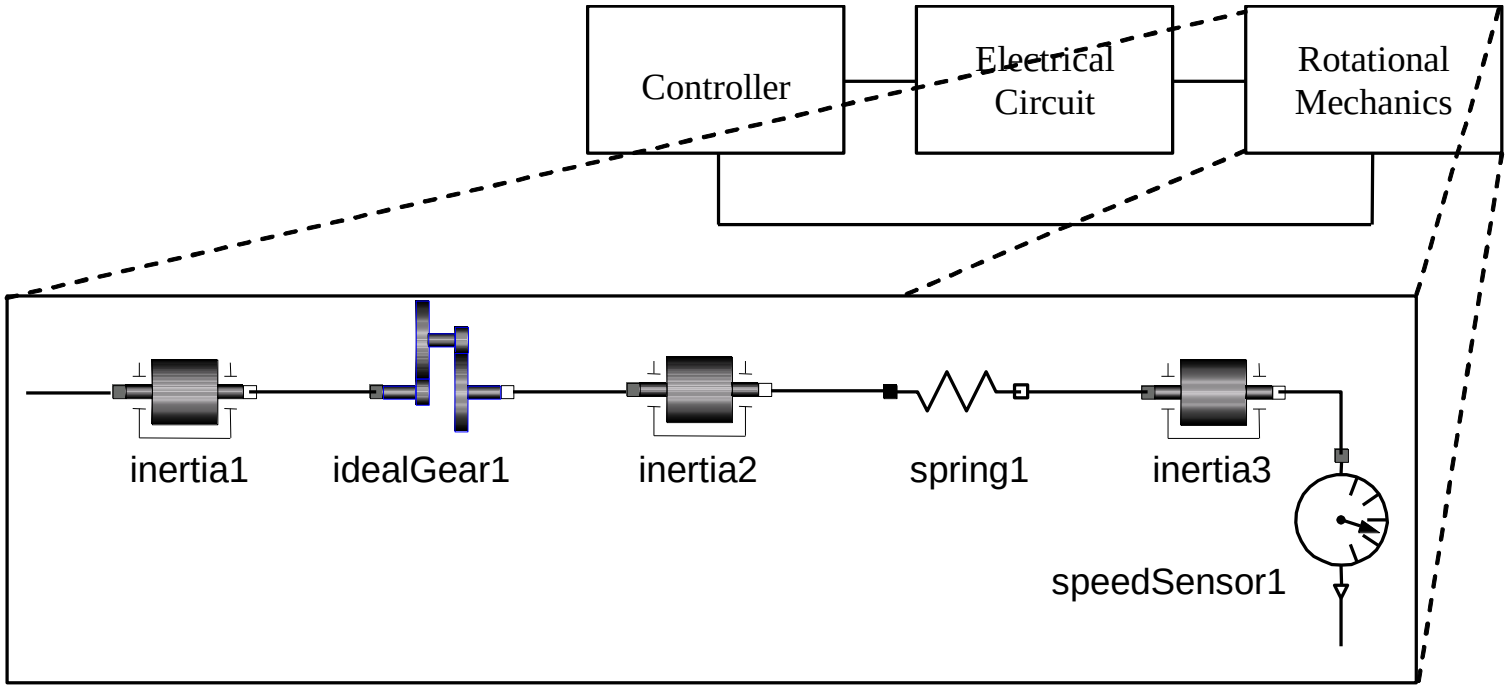
Modeling the controller

Modeling the Electrical Subsystem



Modeling the electric circuit

Modeling the Mechanical Subsystem



Modeling the mechanical subsystem including the speed sensor.