Model Design

Modeling Approaches

- Traditional state space approach
- Traditional signal-style block-oriented approach
- Object-oriented approach based on finished library component models
- Object-oriented flat model approach
- Object-oriented approach with design of library model components

Modeling Approach 1

Traditional state space approach

Traditional State Space Approach

- Basic structuring in terms of subsystems and variables
- Stating equations and formulas
- Converting the model to state space form:

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

Difficulties in State Space Approach

- 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.

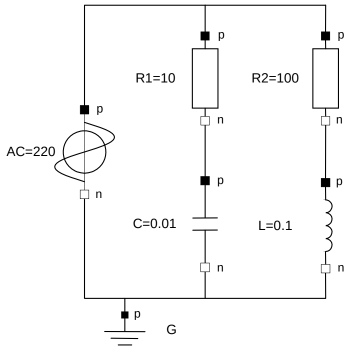
Modeling Approach 2

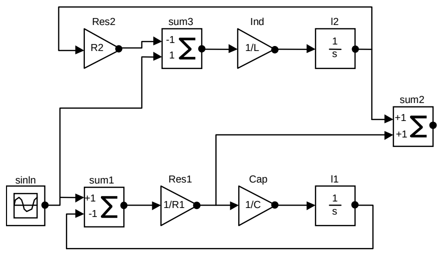
Traditional signal-style block-oriented approach

Physical Modeling Style (e.g Modelica) vs signal flow Block-Oriented Style (e.g. Simulink)

Modelica:
Physical model – easy to understand

Block-oriented:
Signal-flow model – hard to understand for physical systems

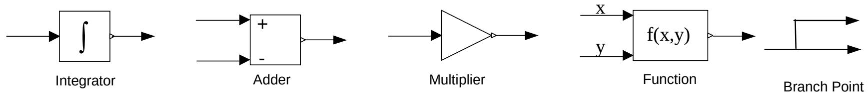




Traditional Block Diagram Modeling

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

Typical Block diagram model components:

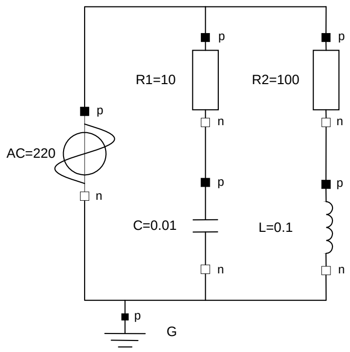


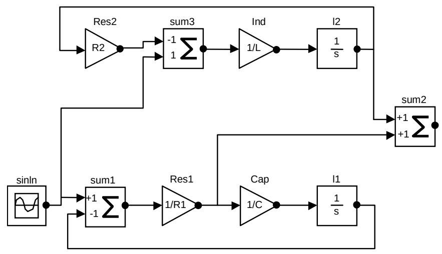
Simulink is a common block diagram tool

Physical Modeling Style (e.g Modelica) vs signal flow Block-Oriented Style (e.g. Simulink)

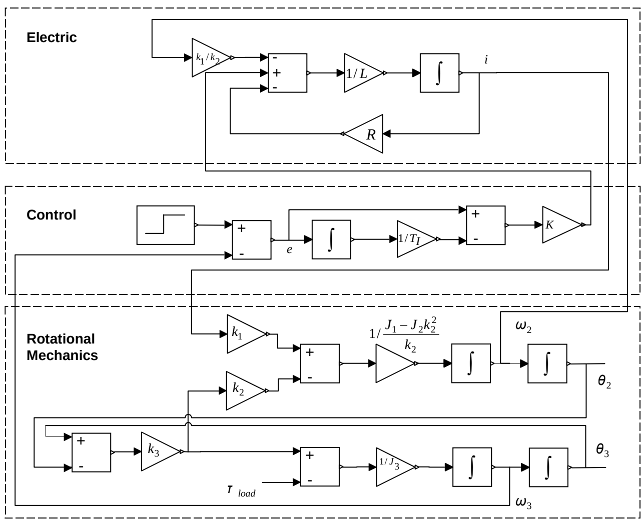
Modelica:
Physical model – easy to understand

Block-oriented:
Signal-flow model – hard to understand for physical systems





Example Block Diagram Models



Properties of Block Diagram Modeling

- - 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 modeling works well for control systems since they are signal-oriented rather than "physical"

Object-Oriented Modeling Variants

- Approach 3: Object-oriented approach based on finished library component models
- Approach 4: Object-oriented flat model approach
- Approach 5: Object-oriented approach with design of library model components

Object-Oriented Component-Based Approaches in General

- 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

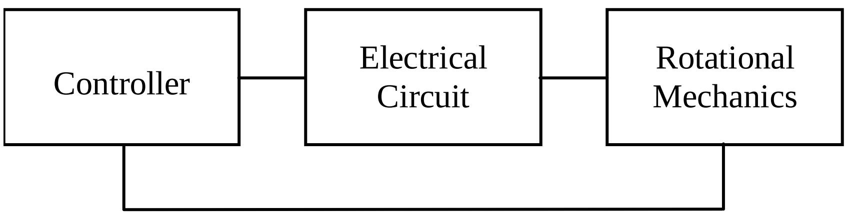
Top-Down versus Bottom-up Modeling

- 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.

Approach 3: Top-Down Object-oriented approach using library model components

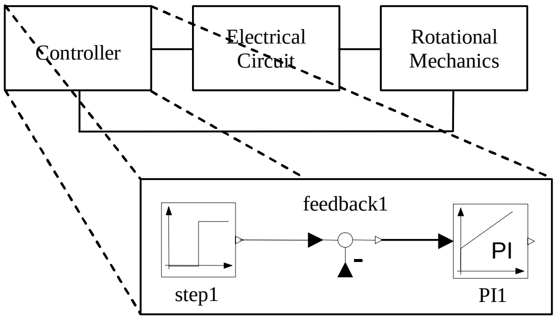
- Decompose into subsystems
- Sketch communication
- Design subsystems models by connecting library component models
- Simulate!

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



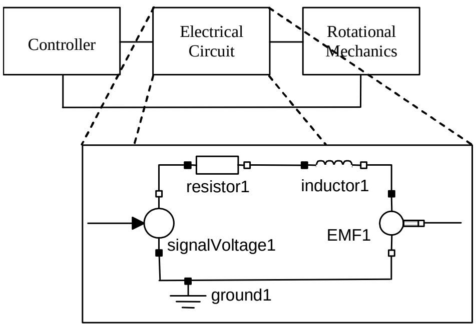
The DC-Motor servo subsystems and their connections

Modeling the Controller Subsystem



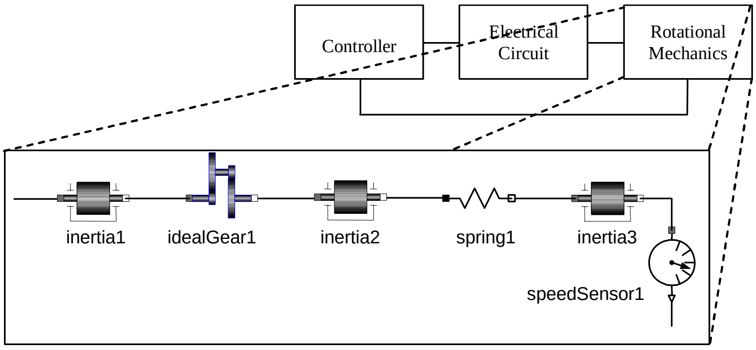
Modeling the controller

Modeling the Electrical Subsystem



Modeling the electric circuit

Modeling the Mechanical Subsystem

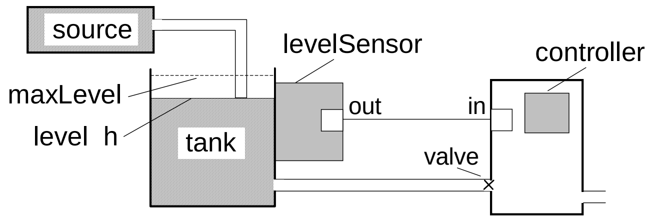


Modeling the mechanical subsystem including the speed sensor.

Object-Oriented Modeling from Scratch

- Approach 4: Object-oriented flat model approach
- Approach 5: Object-oriented approach with design of library model components

Example: OO Modeling of a Tank System



- The system is naturally decomposed into components

Object-Oriented Modeling

Approach 4: Object-oriented flat model design

Tank System Model FlatTank – No Graphical Structure

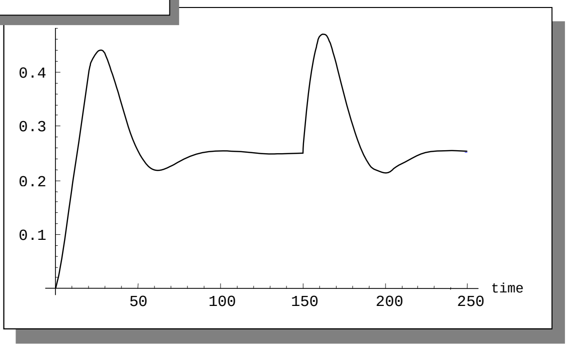
- No component structure
- Just flat set of equations
- Straight-forward but less flexible, no graphical structure

```
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)
```



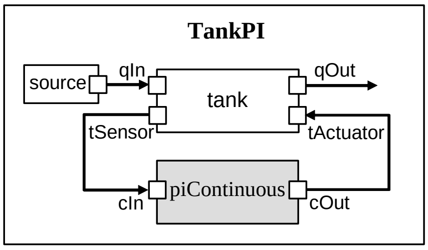
Object-Oriented Modeling

- Approach 5:
  Object-oriented approach with design of library model components

 Peter Fritzson

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;
```

 Peter Fritzson

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;
```

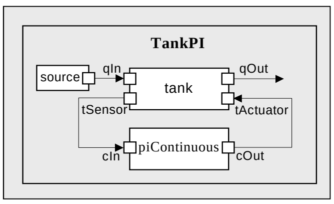
  pelab

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;
```

  pelab

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)

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

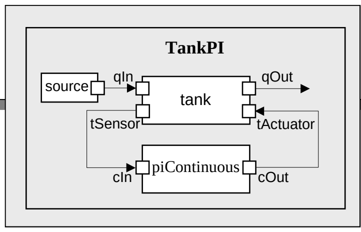
base class for controllers – to be defined

error – to be defined in controller base class

```
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;
```


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;
```

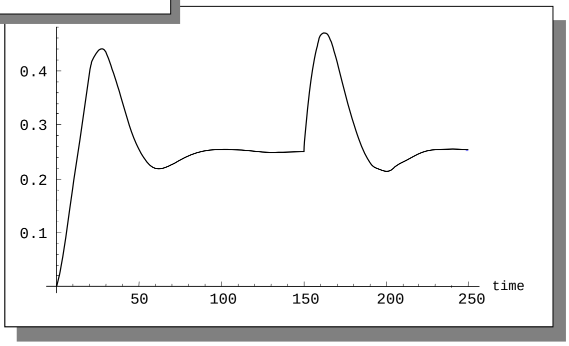


error = difference betwen reference level and actual tank level from cIn connector

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)
```

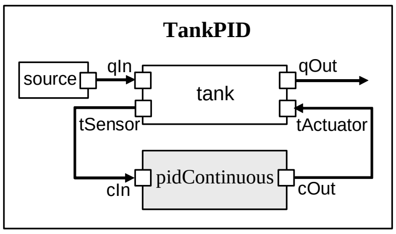


Flexibility of Component-Based Models

- 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)

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

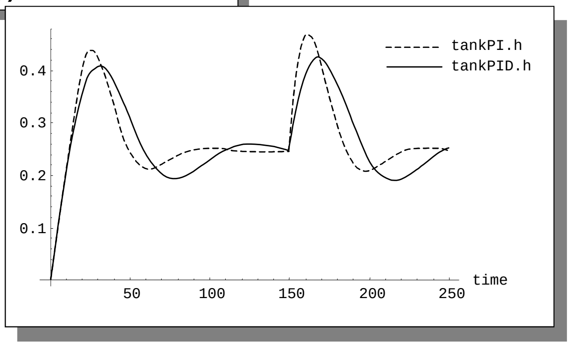
base class for controllers – to be defined

```
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

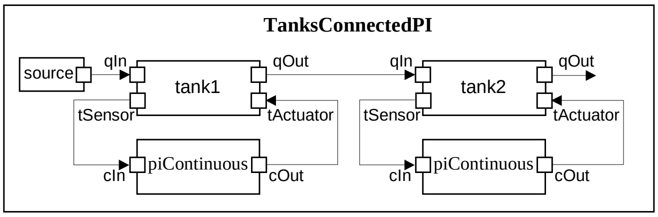
- TankPID with the PID controller gives a slightly different result compared to the TankPI model with the PI controller

```
simulate(compareControllers, stopTime=250)
plot({tankPI.h,tankPID.h})
```



Two Tanks Connected Together

- Flexibility of component-based models allows connecting models together

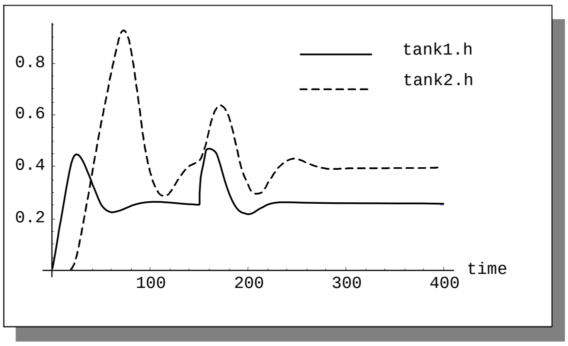



```
model TanksConnectedPI
  LiquidSource  source(flowLevel=0.02);
  Tank          tank1(area=1), tank2(area=1.3);;
  PIcontinuousController piContinuous1(ref=0.25), piContinuous2(ref=0.4);
equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tActuator,piContinuous1.cOut);
  connect(tank1.tSensor,piContinuous1.cIn);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



  pelab

Exchange: Either PI Continous 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;
```

  pelab

Exercises

- Replace the PIcontinuous controller by the PIdiscrete controller and simulate. (see also the book, page 461)
- Create a tank system of 3 connected tanks and simulate.

Principles for Designing Interfaces – i.e., Connector Classes

- Should be *easy* and *natural* to connect components
  - For interfaces to models of physical components it must be physically possible to connect those components
- Component interfaces to facilitate *reuse* of existing model components in class libraries
- Identify kind of interaction
  - If there is interaction between two *physical* components involving energy flow, a combination of one potential and one flow variable in the appropriate domain should be used for the connector class
  - If information or *signals* are exchanged between components, input/output signal variables should be used in the connector class
- Use composite connector classes if several variables are needed

Simplification of Models

- When need to simplify models?
  - When parts of the model are too complex
  - Too time-consuming simulations
  - Numerical instabilities
  - Difficulties in interpreting results due to too many low-level model details
- Simplification approaches
  - Neglect small effects that are not important for the phenomena to be modeled
  - Aggregate state variables into fewer variables
  - Approximate subsystems with very slow dynamics with constants
  - Approximate subsystems with very fast dynamics with static relationships, i.e. not involving time derivatives of those rapidly changing state variables

