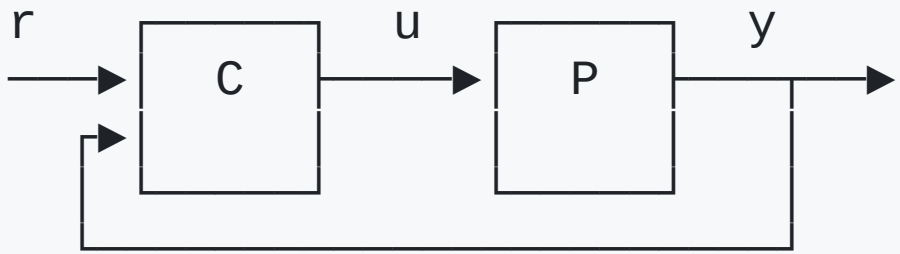
Analysis Points in Acausal Modeling Languages

MODPROD 2026 Workshop

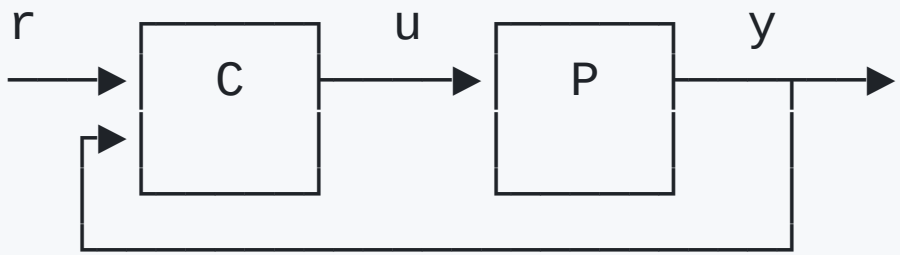
Fredrik Bagge Carlson, JuliaHub

The Problem: Linearizing Fully Connected Models



Challenge:

- Acausal models that can be simulated are fully connected
- Linearization requires the linearization inputs to be unconnected
- The connection equation **binds** the input variable



How do you compute the transfer function

- From `r` to `y`?
- From `u` to `y` without the feedback connection?
- From `u` to `y` with all connections intact

The Old Solution: Manual Model Modification

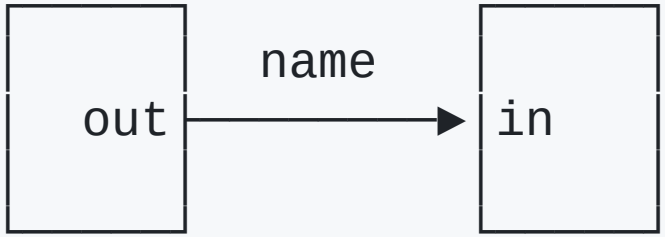
Unpleasant solution:

1. Manually break connections
2. Create separate model variant for linearization
3. Tedious and error-prone for complex systems
4. Hard to automate

What we need:

- Single model for both simulation and analysis
- Analysis-specific behavior without modifying structure
- Named connections for addressing

Analysis Points: Named Connections



Key concept: Analysis points "name" a connection

- **During simulation:** Act as if nothing is there (transparent)
- **During analysis:** Behave in analysis-specific ways
  - Connection can be broken to create unbound inputs
  - Connection destination can be perturbed with artificial input signals
  - Connection source can be measured for output

Analysis Points in Code

Dyad Syntax

```
u: analysis_point(controller.y, plant.u)
```

This name can be added *from anywhere* in the model hierarchy.

ModelingToolkit.jl Syntax

```
connect(controller.output, :u, plant.input)
```

Directionality matters: `output → input` (causal)

To make intent clear and prevent accidental errors

Hierarchical Models: Adding Analysis Points

Analysis points can be added **at any level** of model hierarchy:

This allows for addition of analysis points inside of library components, without modifying the library component.

Analysis Point Transformations (1/2)

Primitive Transformations

Transformation	Effect
Break	Removes connection, optionally adds input
GetInput	No modification, returns variable
PerturbOutput	Adds input `d_u`, optionally adds output
AddVariable	New variable with same type/size as analysis point

Analysis Point Transformations (2/2)

Derived Transformations

Transformation	Implementation
SensitivityTransform	`PerturbOutput(ap, true)`
Comp.Sens.Trans.	GetInput + AddVariable + PerturbOutput
LoopTransferTransform	GetInput + Break(add_input=true)

These combine primitive operations for specific analysis tasks.

Pre-packaged Analysis Functions

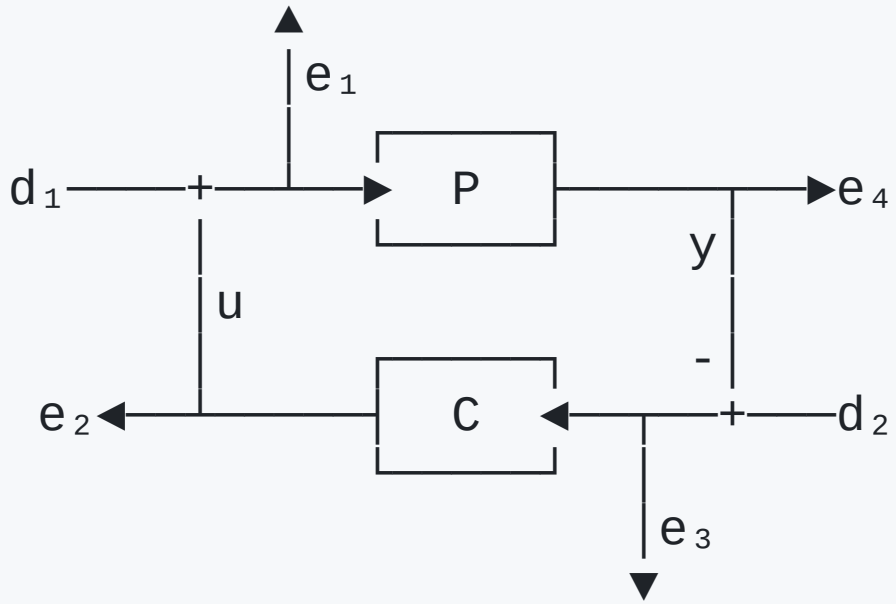
Function	Transformation	Computes
`linearize`	PerturbOutput + AddVariable	Transfer function between points
`get_sensitivity`	SensitivityTransform	Sensitivity $S(s)$
`get_comp_sensitivity`	Comp.Sens.Transform	Comp. sensitivity $T(s)$
`get_looptransfer`	LoopTransferTransform	Loop transfer $L(s) = P(s)C(s)$
`open_loop`	LoopTransferTransform	Nonlinear system with broken loop

All support: `loop_openings` parameter to break additional connections

Example: Analyze the inner loop in a cascade control system

Block Diagrams for Analysis Functions

Sensitivity Function $S(s)$

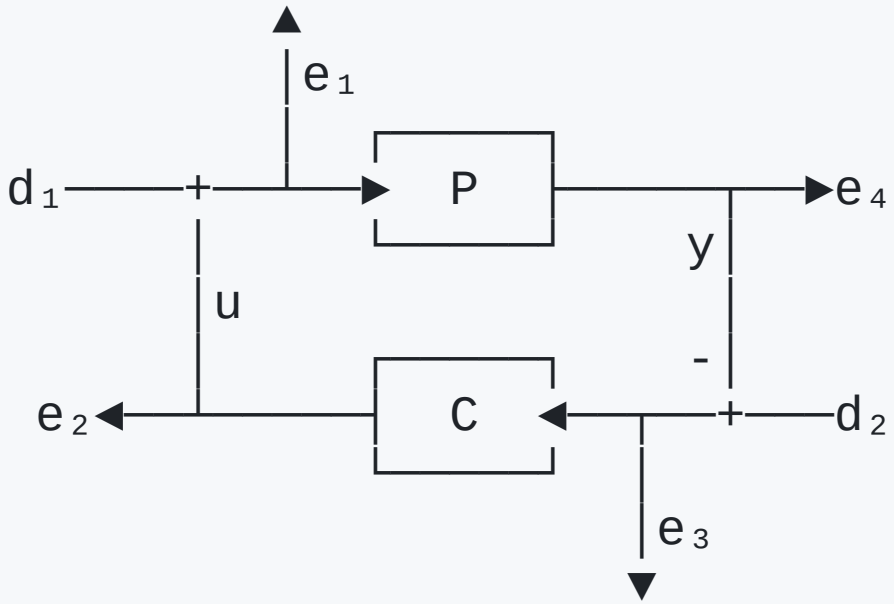



```
get_sensitivity(sys, sys.y)
```

$$S(s) = (I + P(s)C(s))^{-1} = e_3(s)/d_2(s)$$

Add the input variable `d2` and the output variable `e3`, i.e., the output appears *after* the perturbation.

Loop Transfer $L(s)$




```
get_looptransfer(sys, sys.y)
```

$$L(s) = P(s)C(s) = \frac{e_4}{d_2}$$

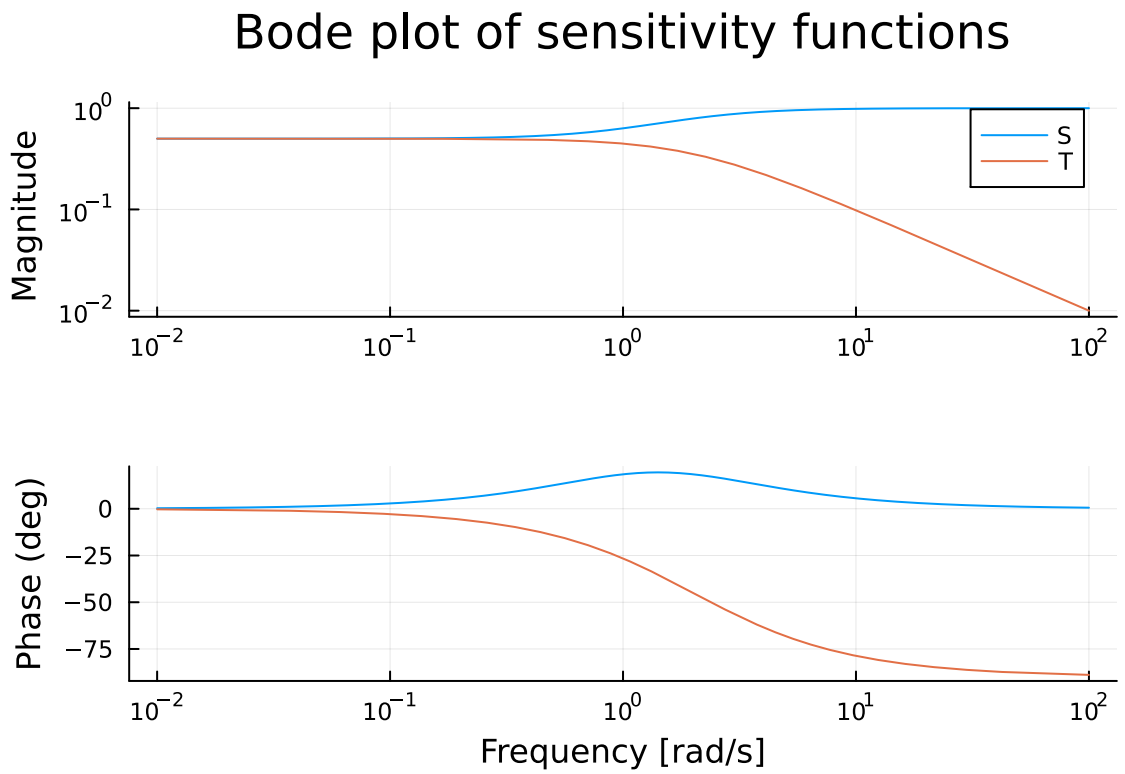
with connection `y` broken

Computing Sensitivity Functions (MTK)

```
using ModelingToolkit, ControlSystemsBase, Plots

@named P = FirstOrder(k=1, T=1)
@named C = Gain(-1)

eqs = [connect(P.output, :plant_output, C.input)
       connect(C.output, :plant_input, P.input)]
sys = System(eqs, t, systems=[P, C], name=:feedback)

# Compute sensitivity functions
matrices_S, _ = get_sensitivity(sys, sys.plant_input)
matrices_T, _ = get_comp_sensitivity(sys, sys.plant_input)

# Convert to linear state-space system objects
S = ss(matrices_S...)  # matrices_S = (A, B, C, D)
T = ss(matrices_T...)
```

Bode Plot of Sensitivity Functions

```
# Create Bode plot
bodeplot([S, T], lab=["S" "" "T" ""],
    plot_title="Bode plot of sensitivity Functions",
    margin=5Plots.mm)
```



Other Enabled features

- Construct state estimator from fully connected simulation model with connected disturbance sources
- Pre packaged analyses:

```
analysis DCMotorClosedLoopAnalysis
  extends DyadControlSystems.ClosedLoopAnalysis(
    measurement   = ["y"],
    control_input = ["u"],
    loop_openings = ["r"],
    wl = 1,
    wu = 1e4,
    duration = 2.0
  )
  model = DyadExampleComponents.TestDCMotorLoadControlled()
end
```


closed-loop analysis

Summary

Analysis points solve a fundamental challenge:

- Single model for simulation and analysis
- No manual model modification required

Key capabilities:

- Name connections in acausal models
- Transform connections for specific analyses
- Pre-packaged functions for common control analyses

Thank You

Questions?

MTK Documentation	Dyad Documentation
	
docs.sciml.ai/ModelingToolkit	help.juliahub.com/dyad