Large Scale System Modeling:

Structuring model libraries
and numerical best practices

Hubertus Tummescheit

With material from Luca Bertuccioli and Lars Pedersen

UTRC

Outline

- Motivation: model & library requirements
- Model architecture:
  - design patterns for object oriented, equation based modeling
- Numerical design patterns
  - Non-linear equations, ODE, DAE, PDE
  - Bounds checking, Scaling
  - Smoothness
  - Stiffness
- Best Practices for Equation-based Model Development (version control, testing, …)
  - Documentation
  - Naming conventions
  - Testing

Model and model library requirements

Homework reading

- Robust, versatile models for system design
  - Easy to initialize, fast execution, failures give meaningful errors
  - Tested in all corners of validity range (testable models)
  - Models adaptable to simulation and sizing optimization
  - Documentation
    - easy to read and understand models
    - complete reference & description exists
    - naming conventions “self-documenting models”).
  - Fully documented range of validity
    - Range of permissible operation is parameter, e.g. 25% – 105% nominal power
  - If more than 1 model needed per component to fulfill all the requirements, these models must be exchangeable (e.g. same degrees of freedom for simulation)
  - Clear definition of what information the model needs to provide in design/sizing, off-design or dynamic operation

Library requirements for numerics and connectivity

Homework reading

- Full understanding of all numerical issues
  - Avoid causing change of mathematical model structure by connection
  - Non-linear equation systems: occurrence, complexity, scaling (usually not an issue in good Modelica libraries)
  - Hybrid issues: smoothness, unavoidable discontinuities, connectivity issues, chattering
- Ability to include existing legacy models
  - Using FORTRAN or C-code, dll’s for part of the model
    - Existing legacy code reuse (property functions, terrain)
    - Handle problems outside of the range of the modeling platform
- Fully modular: no restrictions on model connectivity
  - Subsystems models have same interfaces & connectivity & parameters independent of level of detail

Library requirements

Homework reading

calibration and data consistency

- Must be able to calibrate models using existing data
  - Calibration from experiments
  - Calibration of subsystem models from detailed component models
- Achieving data consistency with other applications must be easy: facilities to import and export data.
  - Static data exchange: before execution
  - Dynamic data exchange: during model execution (performance issue)
- Easy to maintain and add on to libraries
  - Documented, stable interfaces
- Interface to post-processing
  - Define minimum, extensible variable set for reporting
- Use available standards where possible (e.g. S-function interface)

Library architecture

Homework reading

anticipate changes in model requirements

- Define appropriate interfaces (“ports” in gPROMS, “connectors” in Dymola
- Identify common units of reuse
- Use hierarchical sub-libraries for larger projects
- Explore availability of numerically robust legacy codes that could be interfaced
  - gPROMS: “Foreign Objects”
  - Modelica/Dymola: “external functions” and “external objects”
- Use consistent parameterizations and user-interface guidelines throughout the library – higher ease of use and fewer errors by users
- Use encapsulation of model details as much as possible
  - Use of “protected” variables in Modelica
  - Encapsulate using sub-models with well-defined interfaces, all other variables should be protected
- Ensure interoperability with related libraries by having compatible interfaces – use Modelica standard libraries whenever possible
- Use inheritance for model parts that are common to many library elements (“extends” in Modelica)
- Do not sacrifice easy of use and easy understanding for sophisticated reuse

Model architecture

Homework reading

design interfaces and sub-models for modularity and reuse

- Design principle: write models that are easy to use by non-experts!
- Structure of each model needs to be consistent with library structure
- Consistent parameterizations for all related models
- If possible, base on parent class with common interfaces/UI-design (Modelica)
- Document all assumptions and range of validity/applicability
- Define and document allowable degrees of freedom (steady-state models) and boundary & initial conditions (PDE, dynamic models)
- Interaction with other models only via defined, visible interfaces (no “hidden” control-like actions within models)
- Encapsulate functional entities that are used by many components and that might be changed to improve reuse: property calculations for fluids, coordinate transformations in mechanical systems that depend on the parameterization of the orientation.
- Every model needs to have one or more associated test-cases that
  - Demonstrate robustness of the model in the whole operating range
  - Provides a simple verification example

Why Design Patterns?

- Recurring problems
- Canned solutions
- Easy to communicate

- 80 - 90 % of support requests in dynamic modeling answered by a numerical design pattern

Why Design Patterns?

- Some design patterns are trivial
  - – but often missed
- Using an object-oriented language is no silver bullet to obtain code reuse.
- Coding style is an important element

Object-Oriented Modeling is not Object-Oriented Programming

- Static inheritance code structure similar
- static data structures very similar
- almost identical notation

- run time behavior very different
- run time data structures different
- different semantics

Semantics of OOM and OOP

- Object-oriented Modeling
  - differential algebraic equations
  - discrete time events
  - difference equations
  - equations from connecting subsytems
  - structure at run-time is static

⇓ result is one large, hybrid DAE
the complete behavior is used all the time

Homework reading

Semantics of OOM and OOP

- Object-oriented Programming
  - message passing between objects
  - methods operate on encapsulated data
  - objects created and destroyed
  - dynamic run-time structure

⇓ Many executions of a program work if errors are in rarely used methods

Encapsulation

- No encapsulation of operations in OOM!
- data access can be restricted
- parameters can be encapsulated

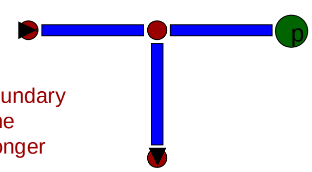
Encapsulation

Example of bad encapsulation properties in OOM:

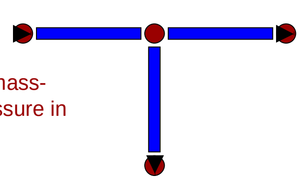
Network of incompressible flows.

2 Types of boundary conditions:

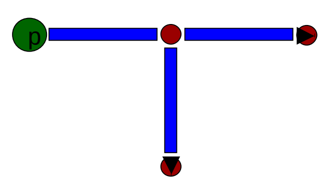
- pressure given
- flow rate given



1 unphysical boundary condition and the problem is no longer well-posed



With 3 given mass-flows, the pressure in the system is undefined



Library Design in OOM and OOP

- Divide-and-conquer strategy of OOP works equally in OOM
- Code design strategies of OOP apply also to OOM:
  - Using only single inheritance leads to too many classes
  - component aggregation often a more flexible design
  - Multiple inheritance useful for *mix-in behavior*
- Interaction via equations has no direction of information flow. Making sure that a unit works in all configurations is more difficult than debugging and testing signal-based models.

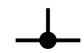
Structural Design Patterns

- Enable code reuse
- Based on Modelica semantics
- Details are important to avoid rewriting code
- Some patterns hold for all engineering domains
  - used in almost all Modelica Libraries
- Other patterns hold only for specific domains

Homework reading

Models of Connection Nodes

- Small - neglect mass and extent



$$\frac{d}{dt}(m \cdot v) = \sum F_i \quad \xrightarrow{m=0} \quad \sum F_i = 0, \quad r_i = r_j$$

- Kirchhoff current and voltage laws

$$\sum I_i = 0, \quad V_i = V_j$$

- Small - neglect volume and losses

$$\frac{d}{dt}(\rho \cdot V) = \sum f_{in} - \sum f_{out} \quad \xrightarrow{V=0} \quad \sum \pm f_i = 0, \quad p_i = p_j$$

- Two basic laws
  - Variables are equal
  - Variables sum to zero

Modelica syntax:
flow-prefix

Homework reading

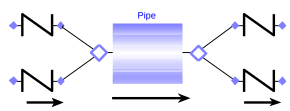
Use Flow Semantics

- *Use Modelica flow semantics for transport of conserved quantities for all physical connectors between subsystems.*

Homework reading

Use Flow Semantics

- Some engineering domains use it traditionally (Electrical, Mechanical), others not
- Natural for bi-directional interaction
- Often omitted for convection models



  - the simplest model for flow splitters and junctions in ThermoFluid makes use of flow semantics

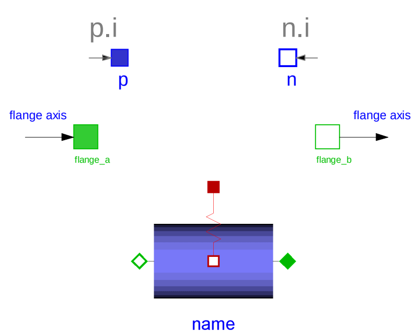
Connectors Sets

- *Provide models with typical configurations in base classes. Implement the physics inside them in derived classes.*

Electrical Library

Translational 1D Mechanics Library

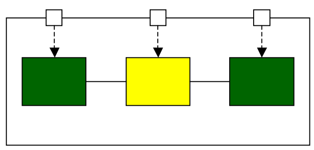
ThermoFluid Library



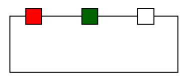
Modelica Class Parameters

- Redeclare **component**
- Individually change class
- Keep connections and parameters
- Checking for consistency

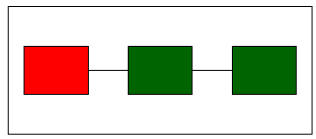
Class declaration



Component declaration

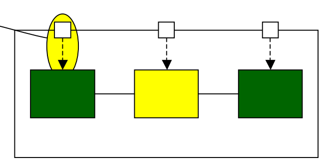


Expansion

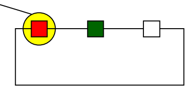


Modelica realization

```
class C
  replaceable GreenClass   comp1(p1=5);
  replaceable YellowClass comp2;
  replaceable GreenClass   comp3;
  connect(…);
end C;
```



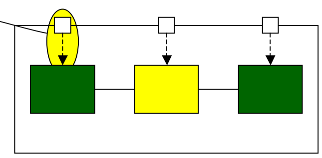
```
class C2 =
  C(redeclare RedClass    comp1,
    redeclare GreenClass comp2);
```



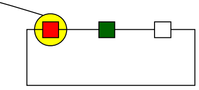
Condition: replacement class is type -compatible to original class!

Modelica realization

```
class C
  replaceable GreenClass  comp1(p1=5);
  replaceable YellowClass comp2;
  replaceable GreenClass  comp3;
  connect(…);
end C;
```

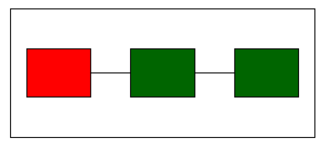


```
class C2 =
  C(redeclare RedClass   comp1,
    redeclare GreenClass comp2);
```



Equivalent to

```
class C
  RedClass   comp1(p1=5);
  GreenClass comp2;
  GreenClass comp3;
  connect(…);
end C;
```



Records for type-compatibility

- Type-compatible: at least the same components, may have extra ones
- May be arbitrary hierarchical structure
- *Collect all components that define a replaceable type in a record. Let all models inherit from that record.*

Records for type-compatibility

- Used in ThermoFluid for fluid properties. Same class with other properties used for Water, CO_2 and R134a.

Mathematical Structure and Model Robustness

- Equation types
  - Non-linear steady-state, ODE, index-1 DAE, High index DAE, PDE,
- Robustness issues non-linear equations
  - Scaling (also for ODE, DAE)
  - Bounds checking
  - Coordinate transformations
  - Infinite derivatives
  - Smoothness
- High-index DAE
- Discretization for PDE
- Hybrid discrete-continuous models

Equation types

- Equation types (often combined)
  - Non-linear steady-state
    - Can be very difficult to solve
    - Non-convexity of equations & multiple solutions are common pitfalls
  - Mixed continuous-discrete
    - Even more difficult to solve
    - Avoid if caused by piecewise discontinuous correlations that should be continuous on physical reasoning

Equation types

- Equation types (continued)
  - ODE, index-1 DAE
    - Scaling of time, stiffness, choice of frequency range of “interesting dynamics”
  - High index DAE
    - Depending on capabilities of environment, circumvent (gPROMS) or utilize (Dymola) with proper coding rules (all code needs to be symbolically differentiable!)
  - PDE
    - Choice of time & length scales of interest, spatial resolution & frequency range matter

Solver types and implications for coding guidelines

Write equations that play nicely with all solver types that might be applied to model!

- Newton-based
  - Smooth, bounded gradients needed (0 and inf are both bad)
- Combined continuous-discrete methods
  - Remove non-physical discontinuities
  - Make unavoidable discontinuities visible to solver, this may be problematic with discontinuities in external code

Solver types and implications for coding guidelines

Homework reading

- Bounds checking for equation solving
  - Often needed due to limited ranges of validity and polynomial approximations
  - Too tight bounds can cause failure
  - Too loose bounds may result in final solutions outside the permissible range.
- Optimization
  - Smooth, convex problems are much easier to solve than others → models are often adapted to optimization needs
  - Often necessary to allow loose bounds and check validity for result as post-condition
- Real-time adapted solvers
  - Fixed step time for guaranteed max. computation time
  - Available in Matlab and Dymola, not gPROMS
  - Errors have to be checked against off-line reference trajectories

Homework reading

Scaling of equations / variables

- **Ideally try to scale variables so that typical magnitude is 1**
  - Not always practical to do this
- **In gPROMS, easier to scale equations**
  - Given the original equation: LHS = RHS
  - Introduce a scaling factor f_{scale} that is a parameter in your model
  - Rewrite equation as: LHS / f_{scale} = RHS / f_{scale}, such that the new terms on either side of the equation are approximately 1
- **In Dymola/Modelica use nominal attribute**
  - **E.g. SIunits.Pressure p(nominal = 1.0e5)**

Homework reading

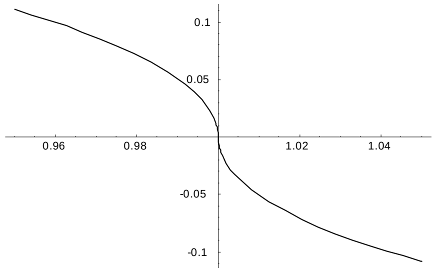
Scaling of equations / variables

- **Always try to scale your mass, energy and momentum equations**
  - Don’t hard-code a scaling factor, rather introduce a parameter
  - A different value for the factor may be needed depending on the model application (e.g. test-tube or full-size plant)
  - Either method (variable scaling or equation scaling) can be used, depending on modeling tool.

Singularities: Root function Example

- Textbook form of turbulent flow resistance

$$\dot{m} - k\,sign(\Delta p)\sqrt{\rho\, abs(\Delta p)} = 0$$



Infinite derivative at origin

Inflection Singularity

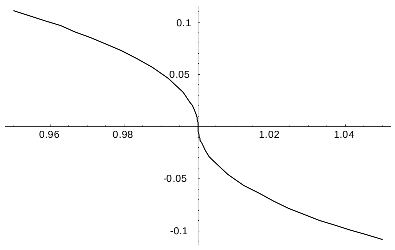
- Infinite derivative causes trouble with Newton-Raphson type solvers:
  Solutions are obtained from following iteration:

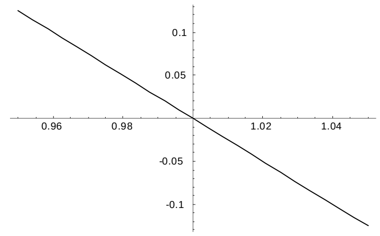
$$z^{j+1} = z^j + \frac{f(z^j)}{\frac{\partial f(z^j)}{\partial z^j}} \approx \Delta z^j + \frac{f(z^j)}{\frac{\Delta f(z^j)}{\Delta z^j}}$$

For $\frac{\partial f(z^j)}{\partial z^j} \to \infty$, the step size goes to 0.

This is called inflection problem

Root function remedy



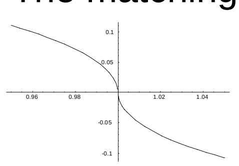


- Replace singular part with non-singular substitute
  - result should be qualitatively correct (quantitative values often unknown)
  - the function should be C^1 continuous

Homework reading

Root function remedy: How-To

- Define region where original correlation is replaced by approximation, e.g. 0.01*mdot0
- Simplify the function at that point inserting the limit
- Take the symbolic derivative of the function at that point
- Compute polynomial coefficients to match both function and derivative at connection point → 4 equations for 4 unknowns
- The matching point may be a parameter



$$f(x) = a + bx + cx^2 + dx^3$$
$$f'(x) = b + 2cx + 3dx^2$$

Homework reading

Log-mean Temperature

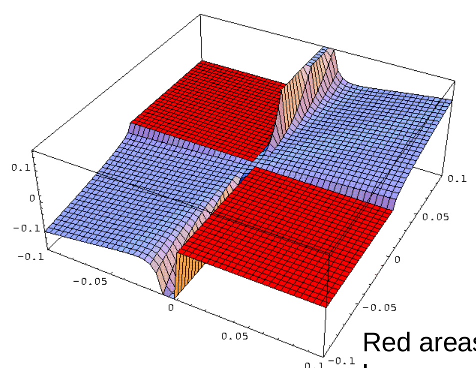
Log-mean Temperature $$\Delta T_{lm} = \frac{\Delta T_1 - \Delta T_2}{\ln(\Delta T_1 / \Delta T_2)}$$

- Invalid for all $\mathrm{sign}(\Delta T_1) \times \mathrm{sign}(\Delta T_2) < 0$
- numerical singularities for

$$\Delta T_1 = \Delta T_2,\ \Delta T_1 \to 0,\ \Delta T_2 \to 0$$

Homework reading

Singularities



Red areas:
log-mean invalid

Log-mean Temperature

Homework reading

Log-mean Temperature

For $\Delta T_1 - \Delta T_2 < 0.01$ K use:

$$\Delta T_{lm} = 0.5(\Delta T_1 + \Delta T_2) \times \left(1 - \frac{1}{12}\frac{(\Delta T_1 - \Delta T_2)^2}{\Delta T_1 \Delta T_2}\left[1 - \frac{1}{2}\frac{(\Delta T_1 - \Delta T_2)^2}{\Delta T_1 \Delta T_2}\right]\right)$$

Beware: this fixes only 1 of the 3 singularity problems! → My suggestion: don’t use it for dynamic modeling!

Homework reading

Non-linear scaling example: Equilibrium of dissociation of Hydrogen

$$\frac{1}{2}\mathrm{H}_2 \longleftrightarrow \mathrm{H}$$

$$x_{\mathrm{H}} = \frac{\sqrt{x_{\mathrm{H}_2}}}{\sqrt{\mathrm{p}}}\exp(k)$$

$$k = 2.6727 - \frac{11.247}{T} - 0.0743T + 0.4317\log(T) + 0.002407T^2$$

Take the log of the equation and variables:

$$\log(x_{\mathrm{H}}) = \frac{1}{2}\left(\log(x_{\mathrm{H}_2}) - \log(p)\right) + k$$

Homework reading

Non-linear scaling example: Equilibrium of dissociation of Hydrogen

$$\frac{1}{2}H_2 \longleftrightarrow H$$

Effect of log at 280K: ratio of $\frac{x_{H_2}}{x_H} = 1.3\times 10^{75}$

ratio of $\frac{\log(x_{H_2})}{\log(x_H)} = 172$

Experiences tested with several non-linear solvers:

22 simultaneous, similar equilibrium reactions
exp form: solvable if T > 1200 K
log form: solvable if T > 250 K

Scaling of equations / variables example collection

1. **Enforce positive values, e.g.**

   $x = y^2 \; if \;$ x is called in $\log(x)$

   $x = \exp(y)$ instead of $y = \log(x)$

   $(x^2)^{1/2}$ instead of $|x|$ or $x^{0.9}$ gets $(x^2)^{0.45}$

2. **linearize expressions (it may help the solver + insure unique solution)**

$$x = y * z \Rightarrow \log(x) = \log(y) + \log(z)$$

Homework reading

Simplifications & avoiding singularities

1. **Decompose & simplify complicated statements**

$$z = \frac{f(x)}{g(y)} \Rightarrow$$

$$y_1 = g(y),\ x_1 = f(x) \Rightarrow z * y_1 = x_1$$

Continue by taking logarithm

2. **Avoid singularities**

$$\log(x) \Rightarrow \log\left((x^2 + 0.000001)^{1/2}\right)$$

3. **Eliminate Denominators (see example above)**

Homework reading

Bounds checking

- Applicable to Dymola and gPROMS
  - Use physically reasonable bounds (1e6 instead of 1e90)
  - Very important for correlations with limited range of validity and ill-behaved function values outside
  - Bounds can also be used to protect from offending terms: introduce
    $\omega =$ offending term
    as separate variable and put bounds on it
  - Sometimes bounds have to be loosened: difficult to find one overall solution
- Proper bounds checking may require non-trivial transformations (see compressor example using a coordinate transformation)

Smoothing

Homework reading

- Piece-wise and discontinuous function approximations which should be continuous for physical reasons should be smoothened.
- Truly discontinuous functions (e.g. step-functions) and functions with discontinuous need to have proper “crossing function” (auto-generated for certain expressions)

```
var1 = if f > f*
  then expression1 else expression2;
```

Modelica, variant 1

```
if f > f* then
  equation1;
else
  equation2;
end if;
```

Modelica, variant 2

Smoothing example: Heat transfer equations

Homework reading

- Convective heat transfer with flow perpendicular to a cylinder. Two Nusselt numbers for laminar and turbulent flow

$$Nu_{lam} = 0.664 Re^{1/2} Pr^{1/3}$$

$$Nu_{turb} = \frac{0.037 Re^{0.8} Pr}{1 + 2.443 Re^{-0.1} (Pr^{2/3} - 1)}$$

For *Re* < 10 we have to take care of the root function singularity as well!

- Combine as:

$$Nu = 0.3 + \sqrt{Nu_{lam}^2 + Nu_{turb}^2}$$

$$10 < Re < 10^7, \quad 0.6 < Pr < 1000$$

Smoothing

Homework reading

- **Often it is useful to smooth discontinuities**
  - e.g. a distributed model with IF statements may be slow to integrate because of repeated re-initializations as the profiles in the equipment shift
  - Smoothened step demonstrated below is a standard trick for smoothing

Original	Smoothed
	Introduce λ: 0 if $f > f^*$, one if $f < f^*$ and with
IF $f > f^*$ THEN	smooth transfer from zero and one
Equation1 = 0	
ELSE	$\lambda = (1 + \tanh(\beta(f - f^*)))/2$
Equation2 = 0	
END	Equation1 * λ + Equation2 * $(1 - \lambda)$ = 0

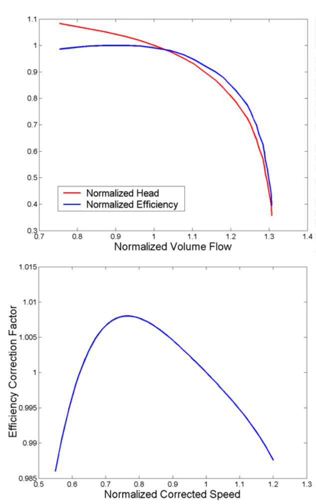
Controls sharpness of transition

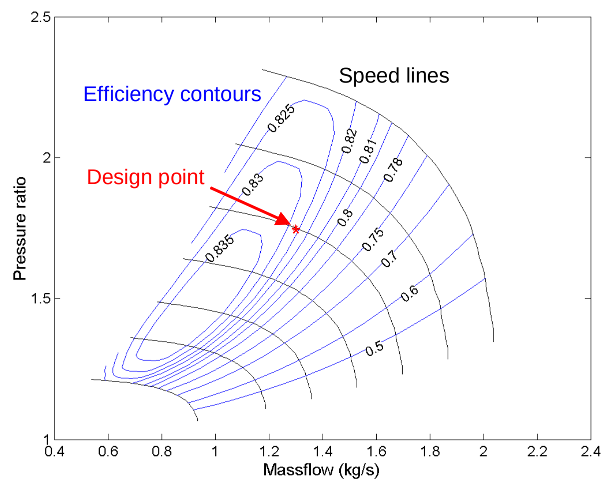
Example: Compressor map scaling

Homework reading

Map inherently has:

- Non-linear boundaries
- Steep slope wrt physical variables



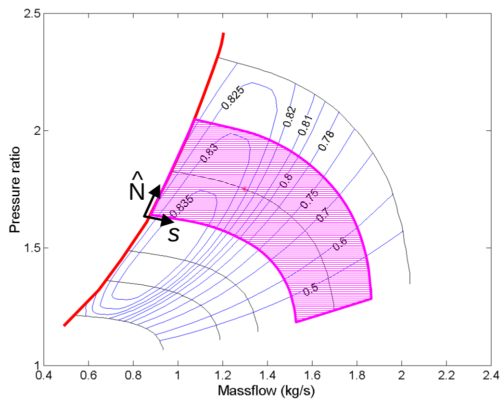


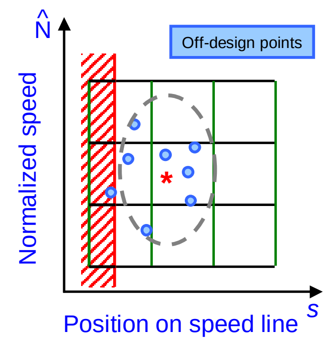
Variable transformation

Homework reading

Define intrinsic variables resulting in:

- Well-defined linear bounds
- More even slopes along characteristics

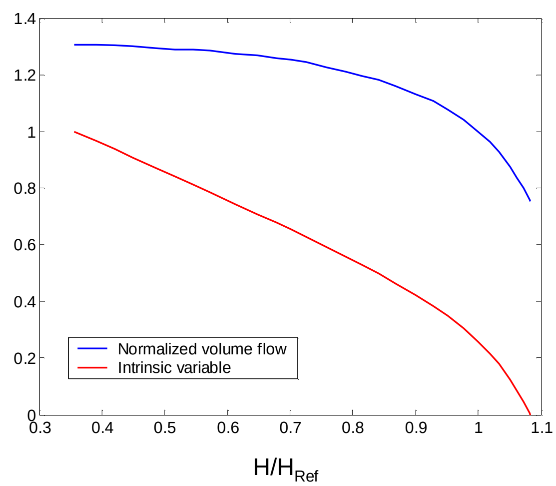




Intrinsic variable benefits

Homework reading

- slopes along characteristics have smaller, numerically beneficial range



Bounds: Compressor example

Homework reading

Possible pitfalls of polynomial fit:

η ⬇

⮟

Req'd compressor power input ⬆

⮟

Req'd turbine power output ⬆

⮟

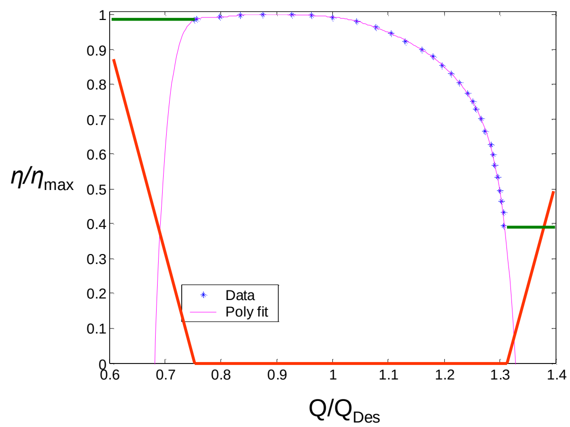
Turbine exit temp ⬇

⮟

-iterate over temp

⮟

Solver failure!



General problem: optimizers may evaluate use models outside the proper range during the search for the optimum

Bounds on an ideal gas radial turbine

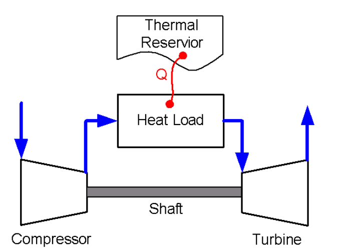
Homework reading

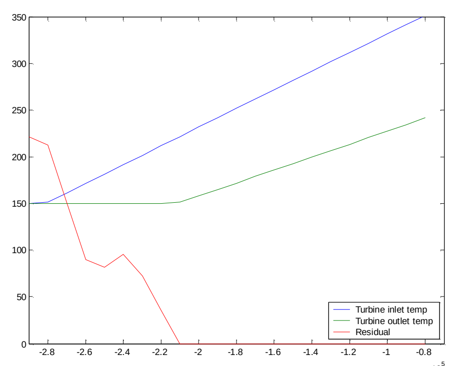
Turbine:
Efficiency bound $0.05 < \eta < 1$

Properties:
T > 150 K

If T < 150 K, property calculation truncated at T = 150 K value

Result ➡ non-convex, non-physical residual





Radial turbine efficiency calculation

Homework reading

Specific speed:

$Ns = \Omega\, Q^{0.5} / H_{ad}^{0.75}$

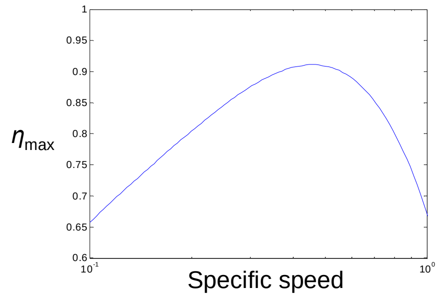
Q Exit volume flow rate

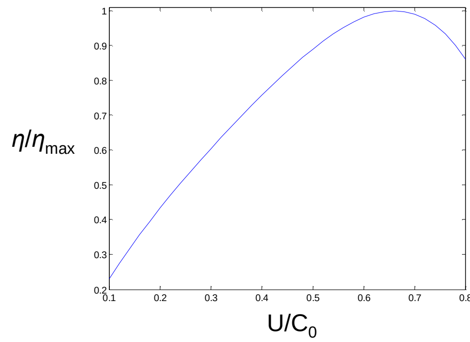
H_{ad} Adiabatic head

C_0 Spouting velocity

U Tip velocity

$C_0 = (2\, H_{ad})^{0.5}$





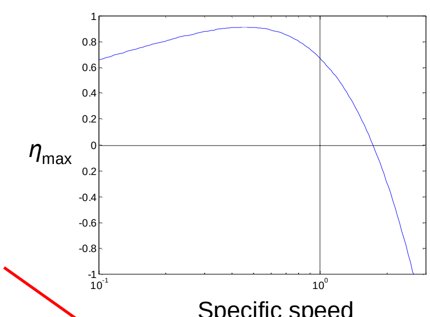
Radial turbine efficiency calculation when beyond bounds

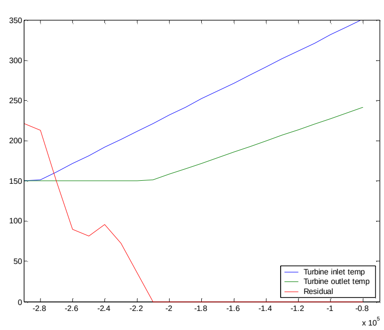
Homework reading

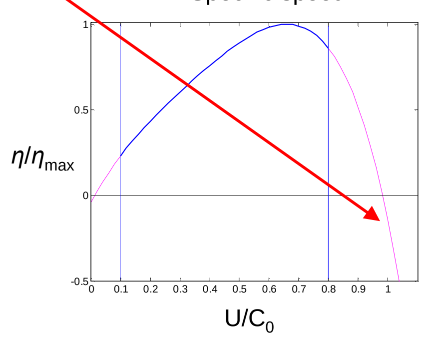
$Ns = \Omega\, Q^{0.5} / H_{ad}^{0.75}$

$C_0 = (2\, H_{ad})^{0.5}$

Negative efficiency!





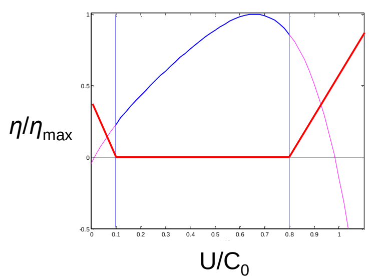


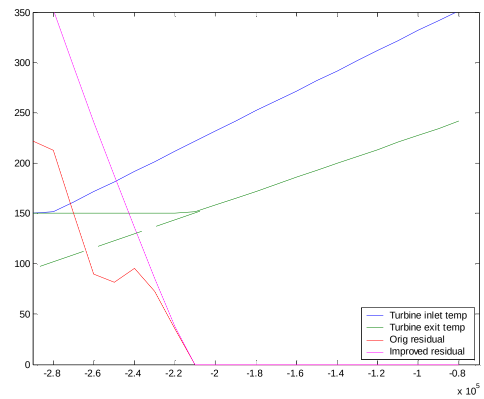
A possible solution

Homework reading

Truncate temperature,
but extrapolate enthalpy

Bound Ns, U/C_0





General rules:

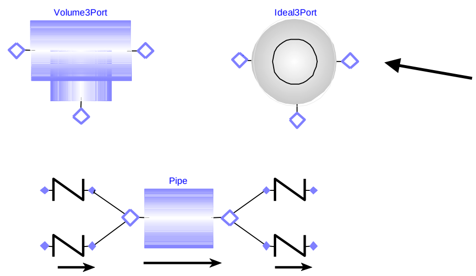
- apply bounds to input variable rather than derived quantity
- ensure bound implementation robustly produces physical output for current & surrounding components
- test, test and test again

Avoiding Stiffness

- Models with largely varying time scales of the dynamics are called *stiff* and cause slow-down of the numerical solvers
- Different remedies are possible:
  - Make a problem non-stiff by using a steady-state assumption (singular perturbation technique) when the fast dynamics of the system are not of interest.
  - Make a problem non-stiff by approximating very slow dynamics with constants.
  - Degrease the stiffness by making very fast time constants slower, but keep them approx. 10 times faster than the fastest time scale of interest

Stiffness caused by Flow Splitters and Joints (small volumes)

Homework reading



Zero-volume assumption in ideal 3-ports is a singular perturbation

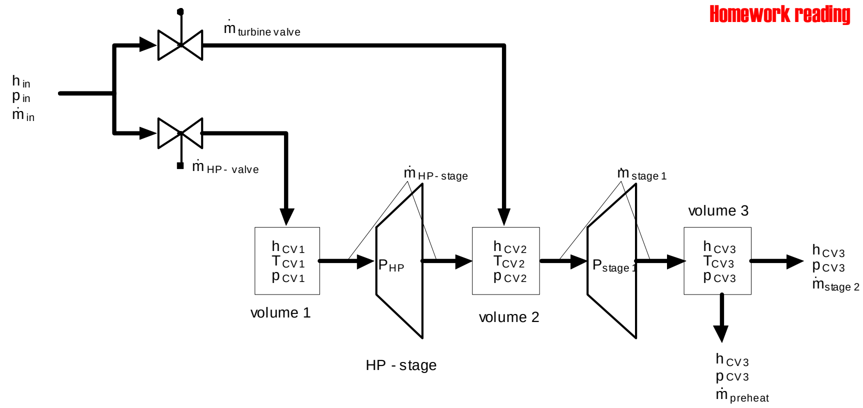
- Many more examples of singular perturbation:
  - chemical equilibrium
  - static instead of dynamic momentum balance

Time scale considerations

- Render a problem less stiff by making the time constants of the fast dynamics slower.
- Different solution for same problem: when to use which solution?
  - singular perturbation leads to nasty non-linear equation system
  - we are not interested in the fast dynamics
  - even after changing the time constants, the fast dynamics are much faster than other, interesting dynamics.

Easing Stiffness: power plant example

Homework reading

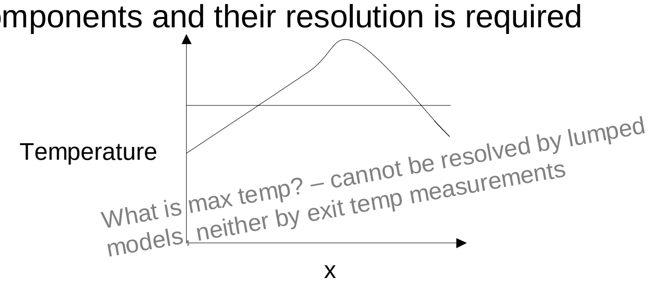


- Very small volumes at tap-off locations cause stiffness
  - Dynamics of interest: power reserve (several seconds)
  - making volumes larger by a factor of 100 does not influence the dynamics of interest (and the dynamics are still much faster than dynamics of interest)

When to go for distributed Modeling?

Homework reading

- When non-linear fields are possible inside the components and their resolution is required



- When Underlying Physics is well-understood – will require physics based parameterization (Enabler)

Example : In a vaporizer, if it is known exit gas is saturated, then X_{water} (out) = X_{sat} for lumped model but for distribution of X in the reactor, need to evaluate heat and mass transfer

Best Practices for model development

- Model development
- Documentation
- Naming Conventions (self-documenting models)
- Team development: version control
- Testing practices
- Robustness metrics

Model Development

- Always assume that somebody else needs to understand and be able to continue work on the model in any phase of the development: document and test while developing (or before), never after the fact.
- Start with simplest models that produces plausible results and **test it**
- Add complexity one step at a time and **test at every step**
- Add changes/difficulties and **test them one at a time** (it is very hard to detect and treat combined difficulties)
- Start with explicit, simple forms, advance towards implicit, numerically challenging ones and **test at every step**
- Separate the difficulties - e.g., at different stages or in modules
- Build hierarchies of models as in the real systems build modules hierarchically starting with lowest level modules which provide outputs as elements for higher modules and **test every time** models are combined
- Adapt models to the needs of the task to be solved, simple models may be sufficient for many tasks

Documentation:

- Provide plenty of documentation, in the code and as separate document, provide templates and guidelines for both
- 50% (or more) of the total text should be documentation
- Standardize on documentation in model header
  - What-how-why of model, assumptions, name, data, test cases, validity range, application areas, qualification level
- Standardize on minimal set of info that has to be added for each model revision, e.g.
  - Author
  - Date of change
  - Reason for change, requester
  - What has been changed
  - How it is demonstrated that the revision fixed a problem (name of test model, verification data set if applicable)

Naming Conventions

- Structuring and naming conventions are influenced by the modeling platform. Models in Modelica are usually structured in small components. Long, self-explaining names help the model reader to understand models, e.g. `discharge_saturation_temperature`
- Use standardized order of terms, e.g. as in `refrigerant_inlet_temperature`
- In Modelica: short names are fine when the meaning is clear from the context, dot-notation can play similar role as long name, e.g: `refrigerant_pipe.inlet.T`
  in addition, variables should always be documented at declaration with a string: `parameter Real eta “total turbine efficiency”`

Team Development

- For any project that involves 2 or more persons, use version control (CVS or similar)
- Team members should present models to each other at regular intervals or, if possible, rotate responsibilities: all team members should be able to work on all models (e**X**treme **P**rogramming works for modeling, too)
- Naming and coding style conventions should be supported by whole team – simplifies communication

Model Testing

- It is very difficult to overdo component robustness testing (you'll be bored before that happens!)
- Every model should have an associated test case:
  - Test model that runs over full range of legal operating conditions
- Testing should be as automatic as possible: use scripts, maybe even in other platform (e.g. matlab scripts to test Dymola-models from matlab in the same way as other matlab code)
- For every model you make, I want to see a test-script that demonstrates the model capabilities
  - Script that demonstrates failures and limits are also fine.
- Testing must be repeatable: keep original model-generated output data.
- Models mature over time through extended use.
  - Assign maturity levels e.g. (exploratory, experimentally verified, production qualified) to models
  - Verification data that proves the model to achieve a certain level of predictive capability shall be part of the model library or version control repository

Model Robustness Metrics

Homework reading

- Use relevant and difficult tests to check model robustness (e.g. initialize in all corners of operating envelope)
- Robustness criteria depend on model use:
  - Simulation models should run almost always (e.g. initialization works in 98% of cases is fine
  - Optimization models have to be 100% robust, because the model is called many times during a single run
- Robustness problems are almost always due to one of the numerical pitfalls described earlier on these slides!
- High Component-level robustness a necessary precondition for system level robustness
- Option: use many randomized parameters and initial conditions or Taguchi-test (spans input space in an optimal way)
  - If all tests are passed, great!
  - Model robustness is unacceptable if a client-defined percentage of failures occurs

Model Robustness

Homework reading

- Taguchi tests
  - Idea: Reduce # of tests by assuming linear dependence and use orthogonal vectors.
  - Our result: Testing robustness for 9 input variables with three levels reduced to 27 experiments.
  - Experiments implemented as a sequence of steps.
  - Scenario can be extended and reduced dependent on # of variables and # of levels.

Model Robustness

Homework reading

- Taguchi tests
  - Example from fuel cell testing: Special reservoir with test sequences for pressure, temperature and compositions.