Towards A Parallel Distributed Equation-Based Simulation Environment

Robert Braun[1] and Petter Krus[1]

Linköping University, Division of Fluid and Mechatronic Systems, SE-58183 Linkping, Sweden

Abstract. Distributed solvers provide several benefits, such as linear scalability and good numerical robustness. By separating components with transmission line elements, simulations can be run in parallel on multi-core processors. At the same time, equation-based modelling offers an intuitive way of writing models. This paper presents an algorithm for generating distributed models from Modelica code using bilinear transform. This also enables hard limitations on variables and their derivatives. The generated Jacobian is linearised and solved using LU-decomposition. The algorithm is implemented in the Hopsan simulation tool. Equations are transformed and differentiated by using the SymPy package for symbolic mathematics. An example model is created and verified against a reference model. Simulation results are similar, but the equation-based model is four to five times slower. Further optimisation of the algorithm is thus required. The future aim is to develop a distributed simulation environment with integrated support for equation-based modelling.

Keywords: model generation, equation-based modelling, distributed solvers, symbolic expressions, numerical solvers

1 Introduction

As products are becoming more complex, the importance of large-scale and multi-disciplinary system simulation is constantly increasing. The use of new powerful product development methods, such as numerical optimization, real-time simulations and hardware-in-the-loop has greatly increased the need for high-performance simulations.

State-of-the-art environments in modern system simulations use equation-based object-oriented (EOO) modelling together with centralized solver algorithms. The equation-based approach offers a very intuitive way for users to write models. Centralized solvers, however, suffer from poor scalability, which means that execution time grows more than linearly with the model size, and are naturally difficult to parallelise for multi-core processors. There are also difficulties in splitting up models for co-simulation or hardware-in-the-loop simulations. Furthermore, fault tracing in an erroneous model is often difficult.

This paper proposes the use of distributed solvers in combination with equation based modelling. Having one small Jacobian matrix for each sub-component

offers nearly-linear scalability. Figure 1 shows an overview of the basic idea. With the use of transmission line element modelling, components can be numerically isolated from one another. This guarantees numerical stability and makes models inherently parallel, and thereby suitable for taking advantage of multi-core processors [1]. Distributed solver simulations can also be seen as a sort of natural co-simulation, making them very suitable for interaction between different simulation tools. A possible drawback with the transmission line element method is that wave propagation phenomena are affected by the size of the time step. If wave propagation is of importance, fixed-size time steps are therefore to be preferred. Experiments using variable time steps with distributed solvers was performed by [2], but the gains was found to be very small. On the other hand, the use of distributed solvers makes it possible to use smaller time steps in certain parts of the model, where a higher resolution is required. Time steps can thus be varied in model space rather than in time space.

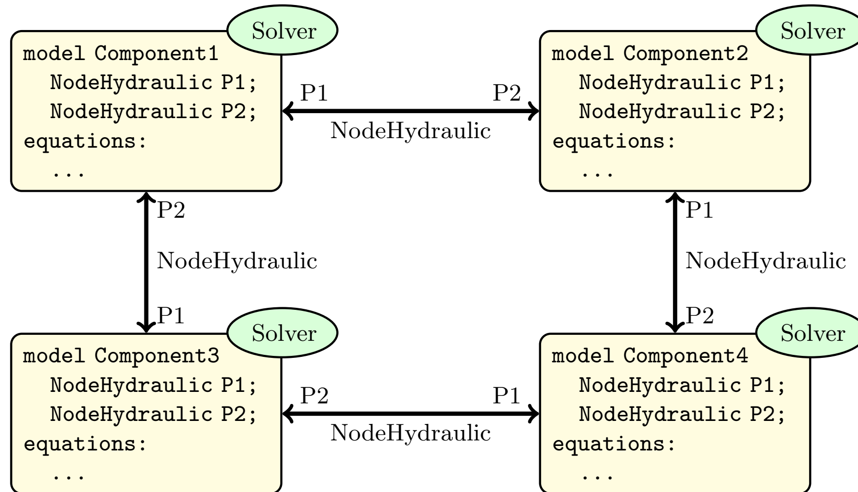


Fig. 1. This paper proposes the use of equation-based modelling in a distributed solver environment. Distributed solvers can provide linear scalability, good numerical properties and natural parallelism.

Distributed simulation environments exist, but they lack built-in support for equation-based modelling. The Hopsan simulation package described in this paper uses pre-compiled libraries written in plain C++ code. Previous attempts to introduce equation-based modelling manually using Mathematica have been successful [3]. This has in turn been used to generate models from Modelica by transforming them to Mathematica syntax [4]. It does, however, require the use of external proprietary tools which cannot be embedded in the simulation tool. Experiments have also been made by going the other way around and intro-

ducing transmission line element models in existing Modelica tools [5][6]. The advantage of implementing equation-based modelling in an existing distributed environment is that already existing features, such as connectors, delays, utility functions and parallel algorithms, can be used directly.

An alternative solution could be to import equation-based models from the Functional Mock-up Interface (FMI), an open standard for interfacing between simulation environments. This is currently being implemented in the Hopsan simulation tool [7]. It is, however, a more cumbersome and likely slower solution, more suitable for importing large models already created in another simulation tool.

2 Transmission Line Element Method

Distributed solvers provide great advantages for performance and numerical properties [8]. Splitting up a model does, however, introduce time delays between sub-models, which affects the mathematical correctness. It is therefore desirable to use a modelling approach where these delays can be physically motivated. This is possible because of the fact that information propagation never happens infinitely fast in a physical system. One such approach is the transmission line element method (TLM). Most physical system models can be rearranged to consist of resistive and capacitive subcomponents. With TLM modelling, each capacitive component in the model is replaced by a transmission line element with a characteristic impedance. The method is related to the method of characteristics [9] and to transmission line modelling as described in [10].

In fluid power systems the capacitive components are represented by pipes (or volumes). These would equal springs in mechanical systems or capacitors in electrical systems. Consider a pipe with time delay ΔT, see figure 2. Each side has two state variables, pressure (p) and flow (q).

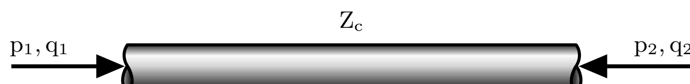


Fig. 2. Transmission line elements are used to numerically isolate different parts of a model from each other. Each part can then be solved independently from the rest of the model.

At a given time t the pressure at one end of the pipe is a function of the characteristic impedance Z_c, the flow at this end at time t, and the flow and pressure at the other end at time $t - \Delta T$, see equations 1 and 2. This can be derived from the four-pole equation [11].

$$p_1(t) = Z_c[q_1(t) + q_2(t - \Delta T)] + p_2(t - \Delta T) \tag{1}$$

$$p_2(t) = Z_c[q_2(t) + q_1(t - \Delta T)] + p_1(t - \Delta T) \tag{2}$$

For simplicity, these equations can be decoupled by introducing a wave variable (c), representing the information travelling from one end to the other at time ΔT:

$$c_1(t) = Z_c q_2(t - \Delta T) + p_2(t - \Delta T) \tag{3}$$

$$c_2(t) = Z_c q_1(t - \Delta T) + p_1(t - \Delta T) \tag{4}$$

This yields the decoupled transmission line equations. These are used as boundary equations in the resistive components:

$$p_1(t) = Z_c q_1(t) + c_1(t) \tag{5}$$

$$p_2(t) = Z_c q_2(t) + c_2(t) \tag{6}$$

In a hydraulic system model, this is implemented by letting the capacitive components calculate c and Z_c from pressure and flow. The resistive components then read c and Zc, apply the boundary equations to calculate pressures, and then in turn use the pressures to calculate flows.

3 Model Generation Algorithm

To solve an equation system, it is necessary to generate a symbolic Jacobian matrix, together with vectors of state variables and system equations. The first step is to parse a Modelica file, containing a model object with connectors, algorithms, equations, variables and parameters. Connectors are hard-typed and the actual connection must be handled by the target simulation environment. Variable limitations specified by the user must also be taken into account. There are two important criteria for the equation system to be solvable. First, the number of equations must equal the number of variables. Second, the resulting Jacobian matrix must not be singular. Verifying the first condition is obviously trivial. Solving the dynamic parts of the system can be done by using the trapezoidal rule, see equation 7. A more effective way of using this is to use bilinear transform, see equation 8. This transforms the equations from continuous to discrete-time by replacing all Laplace operators with a function of the delay operator (z), see figure 4.

$$x(t + h) = x(t) + \frac{1}{2} h(x(t) + x(h + t)) \tag{7}$$

$$F_d(z) = F_a(s)\Big|_{s=\frac{2}{T}\frac{z-1}{z+1}} = F_a\left(\frac{2}{T}\frac{z-1}{z+1}\right) \tag{8}$$

An important aspect in models of non-linear systems is variable limitations. Certain variables are for physical reasons not allowed to be smaller than or larger than specified limits. It is often also necessary to explicitly set the derivatives of the variable to zero when exceeding the limits. In a fixed-step environment with distributed solvers it is not practical to rely on event handling for restarting the solvers when a variable exceeds its limitation. Instead, all limitations must be inserted directly into the transformed equations. Two special functions are used to specify limitations, `VarLimit()` and `VarDerLimit()`, which limit only the variable, or the variable together with the derivative, respectively. In the current implementation, the call to the limitation function must be written directly after the equation(s) defining the variable (and derivative) to be limited. It would be desirable that these equations can be identified automatically by the parser, but for time reasons this has not been included in this paper.

The use of bilinear transform makes it possible to explicitly factor out a given variable from an equation. This allows limit functions to be inserted around the remaining part of the equation. First consider equation 9. This equation gives a relationship between a position x and an external force f. First, the position variable is factored out symbolically as in equation 10. Finally, a `limit()` function is inserted around the remaining part of the equation, effectively limiting the variable x, see equation 11.

$$\mathrm{F}_1(x, f) = 0 \tag{9}$$

$$x - \mathrm{F}_1(f) = 0 \tag{10}$$

$$x - \mathrm{limit}(\mathrm{F}_1(f), x_{min}, x_{max}) = 0 \tag{11}$$

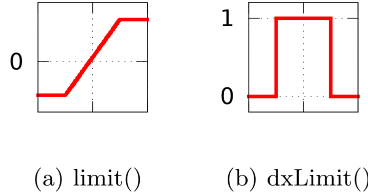
The limitation of derivatives are performed in a similar way. Equation 12 defines the velocity v as a function of the position and the force. The velocity variable is factored out (equation 13), and a function called `dxLimit()` is inserted before the remaining part of the equation (14). This function returns one if the position variable is within limits, otherwise zero. The limitation functions `limit()` and `dxLimit()` are shown in figure 3. The derivative of `limit()` is `dxLimit()`, and the derivative of `dxLimit()` is zero.

$$\mathrm{F}_2(v, x, f) = 0 \tag{12}$$

$$v - \mathrm{F}_2(x, f) = 0 \tag{13}$$

$$v - \mathrm{dxLimit}(x, x_{min}, x_{max})\mathrm{F}_2(x, f) = 0 \tag{14}$$

It is important that the generated equations are simplified as much as possible. Having longer equations than necessary means that the generated code will take longer to evaluate each time step, resulting in a slower model. Furthermore, it is important that equations are simplified in the correct way. It is for example



(a) limit() (b) dxLimit()

Fig. 3. Two limitation functions can be inserted into the transformed equations, one that limits a variable, and one that limits the derivative of a variable.

desirable that all delay operators are factored together. Having too many delayed variables will otherwise become a bottleneck. Too many power operators, including square roots, will likely also reduce performance.

The next step is to generate a Jacobian matrix. This is done by analytically differentiating each equation with respect to each state variable. As a consequence of the fact that users are allowed to write equations in arbitrary order, it is possible that the Jacobian matrix will become singular. When solving an equation system in matrix form, however, inverting the Jacobian matrix will be inevitable. A singular matrix thus makes the system unsolvable and must be avoided. One way to guarantee this is to ensure that no element on the diagonal are zero. This can be achieved by ensuring all diagonal elements are constants. With this method, the matrix will always be invertible regardless of the working point. This is possible for most physical equation systems. A weaker requirement is that all diagonal elements must at least be analytically different from zero. In this case, the Jacobian may become singular, but only for certain values of the state variables.

Once a non-singular Jacobian has been generated, any fixed-step numerical solver can be used. In this paper, an iterative Newton-Rhapson solver is used, see equation 15. In most cases one iteration is sufficient, but more iterations may be required in models with strong non-linearities.

$$x_{k+1} = x_k(t) - J_k(t)^{-1} G(x_k(t)) \tag{15}$$

Inverting the Jacobian matrix each time step is a time-consuming solution. A more efficient method is to use LU-decomposition, a matrix form of Gaussian elimination [12]. The Jacobian is first decomposed to a product of an upper matrix, which only has elements above the diagonal, and one lower that only has elements on or below the diagonal: $Jx = b \Leftrightarrow LUx = b$. Then the system $Ly = b$ is solved for y, which in turn is used to solve $Ux = y$ for x. This algorithm can also easily be re-written for parallel execution [13]. In this paper, parallelism was, however, implemented in model space rather than using parallel algorithms.

The full procedure for generating and simulating equation-based models are shown in figure 4.

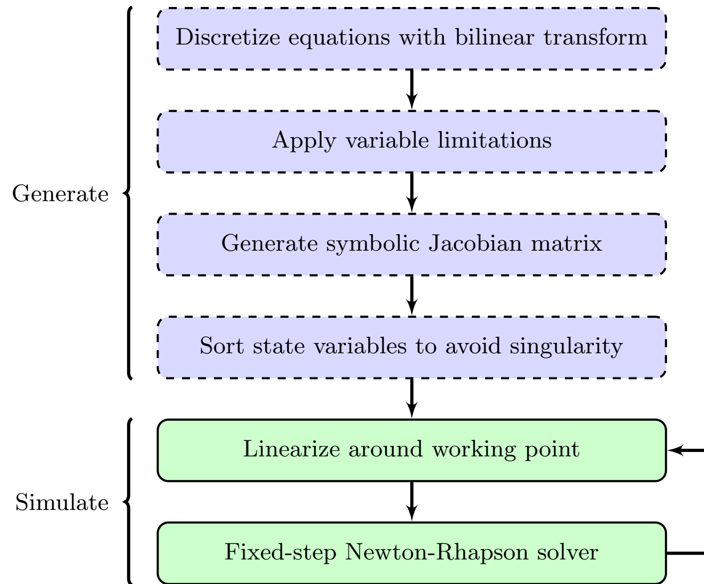


Fig. 4. Bilinear transform is used to convert equations to discrete form. This makes it possible to apply variable limitations before generating the symbolic Jacobian. Equations are solved using distributed fixed-step solvers.

4 Implementation

The algorithm described in this paper was implemented in Hopsan, a cross-platform distributed simulation environment developed at Linköping University [14][7]. The application is fully object-oriented and uses pre-compiled component libraries. No compilation is thus required during runtime. The simulation core is separated from the graphical interface, making it suitable for both desktop and embedded applications [15]. It has built-in support for multi-threaded simulations, which uses the time independences introduced by the transmission line element method [1].

Converting equations to plain code requires symbolic computations. SymPy is a free Python library for symbolic computations, providing objects for symbols, functions and expressions. It is capable of all necessary operations such as replacing symbols, simplifications, factorization and differentiation [16][17]. The choice fell on SymPy mainly due to the fact that Hopsan has a built-in Python console, making a Python library ideal for early experimenting.

Due to the use of distributed solvers, only a subset of the Modelica language can be used. Most importantly, all connectors are hard-coded to match the Hopsan node types. An example of a hydraulic node connector is shown in listing 1.1. This is necessary because all connections are handled by the simulation core. Custom connectors are, therefore, not allowed. Other Modelica

features that contradict the distributed modelling approach, such as the `inner` and `outer` keywords, are also not allowed. Another reduction is imposed by the use of fixed-step solvers. This practically eliminates the need for event handling, which is thus not supported. Sub-classing and functions are also not supported, although this could easily be implemented in the future. Algorithm sections are allowed, but only once before and once after the equation section in each model. Nested algorithm sections are not allowed due to the limitation of only one Jacobian matrix in each component. There are, however, no technical difficulties in introducing this in the future. The standard Modelica library contains many built-in intrinsic mathematical functions [18], most of which are supported. A list of supported functions are shown in table 4.

sin	atan2	exp	div
cos	sinh	log	rem
tan	cosh	log10	mod
asin	tanh	sign	floor
acos	abs	integer	ceil
atan	sqrt	der	

Table 1. These Modelica functions are supported by the implementation of the model generation algorithm.

```
connector NodeHydraulic "Hydraulic Node"
   Real      p "Pressure";
   Real      q "Flow";
   Real      c "Wave Variable";
   Real      Zc "Characteristic Impedance";
end NodeHydraulic;
```

Listing 1.1. Connectors must be hard-typed to match the Hopsan node types. This code shows a hydraulic node connector in Modelica syntax.

Equations can either be written directly in the graphical interface in Hopsan or loaded from an external Modelica file. The generator utility function parses the equation, verifies the syntax and the number of unknowns and replaces any reserved words with temporary strings.

Lists with all equations, variables, state variables and parameters are created. These are in turn used to define symbols, functions and expressions in SymPy. The equations are transformed to discrete form. After this, the variable limits are applied using the `factor()` and `subs()` SymPy functions for factorization and variable substitution. The Jacobian matrix is created by differentiating the equations using the `diff()` SymPy function. The equations are then returned to Hopsan and translated from Python to C++ syntax. Unit delays in the equations are replaced by Hopsan delay methods ($z^{-n}x = \text{mDelay}(x, n)$). All integer

variables are also converted to decimal numbers to ensure precision and avoid phenomena such as "$1/2 = 0$".

The next step is to ensure that the resulting Jacobian matrix is not singular. This is performed by a bubble-sort algorithm that attempts to arrange system equations so that they include their corresponding state variable. If this fails, the generation is aborted because the system is not solvable. The performance of this sorting could be improved by a more sophisticated algorithm, but this was not considered necessary as the required time was very small compared to the rest of the process. If everything was successful, the Jacobian is converted to C++ source code together with a fixed step Newton-Rhapson using LU-decomposition, which is in turn compiled to a Hopsan component. The symbolic Jacobian is also displayed in a dialogue to the user.

5 Example: Hydraulic Pressure Relief Valve

As a demonstration of the method, a model of a hydraulic pressure relief valve is presented. This problem is interesting because it contains several difficult modelling phenomena, such as second order dynamics, variable limitations and a non-linear flow function. A relief valve consists of a cone attached to a spring. When pressure on the high-pressure side overcomes the pre-tension of the spring, the cone will move, allowing oil to flow to the low-pressure side, see figure 5.

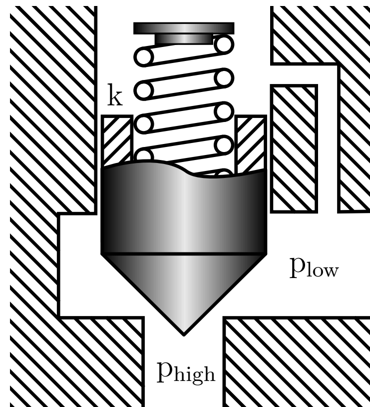


Fig. 5. A pressure relief valve consists of a spring-loaded cone. It will open when the force from the high-pressure side exceeds the pre-load of the spring.

The cone is modelled as an inertia with damping and end of stroke limitations. It is subjected to a spring force, a spring pre-load and the forces from the low and high oil pressure, see equation 16. The flow is a function of the square root

of the pressure difference. Square roots, however, are undefined for numbers that are smaller than or equal to zero. For this reason, a sign function is used. An overlapping is also introduced to avoid the non-linearity at zero, see equation 17. Normally, a model of a hydraulic valve should take cavitation (zero pressure) into account. For time and space reasons, this has been left out in this example.

$$M_v \ddot{x}_v + B_v \dot{x}_v + k_e x_v = (p_1 - p_2 - p_{ref}) A_v \tag{16}$$

$$q = C_q A \sqrt{\frac{2}{\rho}(p_1 - p_2)} \tag{17}$$

The Modelica code is shown in listing 1.2. An annotation is added that tells Hopsan that this is a Q-type component (see section 2). Two hydraulic connectors and two signal connectors are then added; the latter are used to show cone position and cone velocity for debugging purposes. Parameters are specified in standard Modelica syntax with type, name, unit, default value and description. Four local variables are also used. These are assigned in the algorithm section, before the equations. Moving explicit expressions from equations to algorithms like this can greatly improve simulation performance. In the equation section, the `Variable2Limits` function is written among the other equations. The two equations above must define the variable and the derivative to be limited. The limit equation will not be included in the system equations later on; it will be removed once the limitation is applied. The resulting Jacobian matrix and system equations are shown in equation 18.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \mathrm{f}(p_1,p_2) & \mathrm{f}(p_1,p_2) \\ \mathrm{f}(p_1,p_2,x_v) & 1 & 0 & 0 & \mathrm{f}(x_v) & \mathrm{f}(x_v) \\ \mathrm{f}(p_1,p_2) & 0 & 1 & 0 & \mathrm{f}(p_1,p_2,x_v) & \mathrm{f}(p_1,p_2,x_v) \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -Z_{c1} & 1 & 0 \\ 0 & 0 & -Z_{c1} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ d_{xv} \\ q_2 \\ q_1 \\ p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} \mathrm{f}(p_1,p_2,q_2,x_v) \\ \mathrm{f}(d_{xv},p_1,p_2,x_v) \\ \mathrm{f}(d_{xv},x_v) \\ \mathrm{f}(q_1,q_2) \\ \mathrm{f}(p_1,q_1) \\ \mathrm{f}(p_2,q_2) \end{bmatrix} \tag{18}$$

The example model was verified against an existing model of a relief valve written in plain code. For this, an example model consisting of a volume provided with constant flow, connected to a pressure relief valve and an orifice was used, see figure 6. The size of the orifice is reduced by a step function after 0.3 seconds to test the dynamics of the relief valve. All components in the test system model were created from Modelica equations and compared to a reference system model where all components were written in plain code. The resulting pressure in the volume in the two models are shown in figures 7 and 8.

Simulation performance was investigated by running 10^6 iterations and measuring simulation time. The model generated from Modelica had an average of 2379 ms while the reference model was notably faster, with an average of 454 ms. Enabling the multi-core support in Hopsan on a dual-core computer reduced simulation time for the generated model to 1696 ms. The reference model

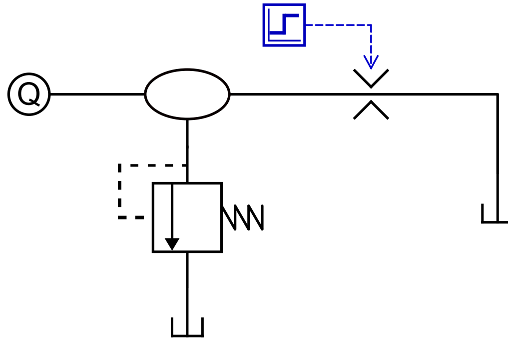


Fig. 6. A system model consisting of a pressure relief valve connected to a volume with a flow source and an outlet orifice was used for validation.

performed less well with parallel simulations, due to overhead time costs. Comparing the smallest average time for both models, the Modelica model was 4.535 times slower than the reference model. A fair comparison with another simulation tool is not possible because Hopsan is the only tool with support for TLM with correct time delays. In the test run, only one iteration was used in the solvers. This was sufficient to give accurate results when using the same time step as the reference model.

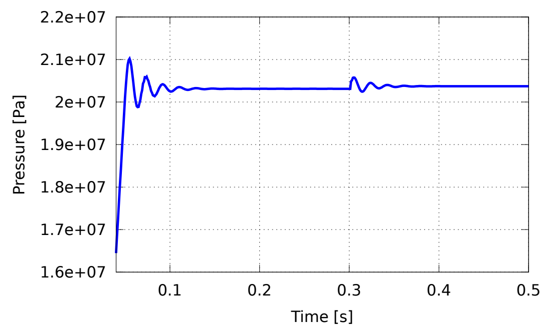


Fig. 7. When the pressure relief valve reaches its reference value it will have some oscillations due to the mass-spring dynamics and a disturbance after .3 seconds.

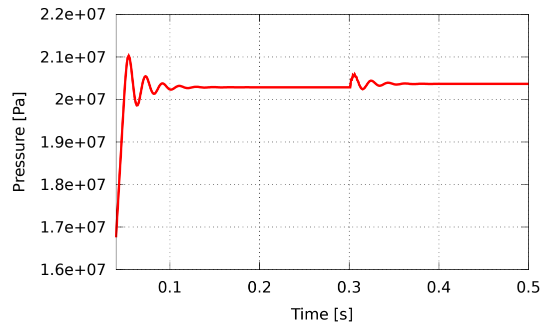


Fig. 8. The generated model show no differences in simulation results compared to the reference model.

```
model MyPressureReliefValve "My Hydraulic Pressure Relief Valve"
  annotation(hopsanCqsType = "Q");

  NodeHydraulic P1, P2;
  NodeSignalOut xv, dxv;

  parameter Real rho(unit="kg/m^3")=870 "Oil Density";
  parameter Real visc(unit=Ns/m^2"")=0.03 "Dynamic Viscosity";
  parameter Real Dv(unit="m")=0.03 "Spool Diameter";
  parameter Real Bv(unit="N/(m*s)")=0.03 "Damping Coefficient";
  parameter Real Mv(unit="kg")=0.03 "Spool Mass";
  parameter Real Xvmax(unit="m")=0.03 "Maximum Spool Displacement";
  parameter Real Cq(unit="-")=0.67 "Pressure-Flow Coefficient";
  parameter Real phi(unit="rad")=0.01 "Stream Angle";
  parameter Real ks(unit="N/m")=100 "Spring Constant";
  parameter Real p0(unit="Pa")=1e5 "Pressure For Turbulent Flow";
  parameter Real pref(unit="Pa")=2e7 "Reference Opening Pressure";

  Real Av "Valve Cross Section Area";
  Real w "Area Gradient";
  Real kf "Flow Force Spring Constant";
  Real ke "Total Effective Spring Constant";

algorithm
  Av := 3.1415*Dv**2/4;
  w  := 3.1415*Dv*sin(phi);
  kf := 2*Cq*w*cos(phi)*(p1-p2);
  ke := ks+kf;

equation
  Mv*der(der(xv.out))+Bv*der(xv.out)+ke*xv.out = (P1.p-P2.p-pref)*Av;
  Mv*der(dxv.out)+Bv*dxv.out+ke*xv.out = (P1.p-P2.p-pref)*Av;
  VarDerLimit(xv.out, dxv.out, 0, Xvmax);
  P2.q = xv.out*Cq*w*sqrt(2/rho)*(sqrt(p0+abs(P1.p-P2.p))-sqrt(p0))*sign(P1.p-P2.p);
  P1.q = -P2.q;
  P1.p = P1.c+P1.Zc*P1.q;
  P2.p = P2.c+P2.Zc*P2.q;

end MyPressureReliefValve;
```

Listing 1.2. A hydraulic pressure relief valve was modelled in Modelica. A special variable limitation function was introduced.

6 Conclusions

A distributed solver approach provides good numerical properties and is suitable for running parallel simulations. This paper presents a method for generating components for a distributed solver simulation tool using the Modelica language. A solution for efficiently implementing variable limitations, including their derivatives, is also described. Finally, the method is demonstrated by generating a dynamic model and comparing it to a reference model written in plain C++ code.

Results show that the method is applicable. Experimental results show no fundamental differences in simulation results compared to a reference model. The generated model is, however, substantially slower than the reference model. This was expected since equation-based models require the solvers to do more work than in a manually coded model, where equations to a large extent can be solved beforehand. Optimizing simulation performance further is, however, still desirable, especially if models are to be used in real-time applications. Possible speed-ups could be achieved from performing the LU decomposition analytically before generating the components, instead of numerically each time step. Further simplification of the equations may also be possible, as well as optimizing the generated code and the solver.

The model generation in itself was quite slow, due to the use of a Python package. Some symbolic operations in SymPy are also not implemented for special cases, and therefore not fully reliable. A great improvement would be to use a C++ library for symbolic computations instead. Using the Modelica parser and rewriting it for distributed solvers could also be an option.

One of the most important advantages of using equation-based modeling with distributed solvers is scalability. The time required for solving an equation system numerically increases super-linearly to the number of equations, making centralized solvers slow for large models. With distributed modeling, the equation system is naturally decomposed into one small system for each component, which can greatly reduce simulation time.

7 Acknowledgements

This work was supported by ProViking research school and the Swedish Foundation for Strategic Research (SSF).

References

[1] R. Braun, P. Nordin, B. Eriksson, and P. Krus. High Performance System Simulation Using Multiple Processor Cores. In *The Twelfth Scandinavian International Conference On Fluid Power*, Tampere, Finland, May 2011.

[2] A. Jansson, P. Krus, and J-O Palmberg. Variable time step size applied to simulation of fluid power systems using transmission line elements. In *Fifth Bath International Fluid Power Workshop*, Bath, England, 1992.

[3] Petter Krus. An automated approach for creating component and subsystem models for simulation of distributed systems. In *Proceeedings of the Nineth Bath International Fluid Power Workshop*, Bath, England, 1996.

[4] B. Johansson and P. Krus. Modelica in a Distributed Environment Using Transmission Line Modelling. In *Modelica 2000 Workshop*, Lund, Sweden, October 2000.

[5] Kaj Nystrm and Peter Fritzson. Parallel Simulation with Transmission Lines in Modelica. In *5th International Modelica Conference*, Vienna, Austria, September 2006.

[6] M. Sjölund, R. Braun, P. Fritzson, and P. Krus. Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling. In *3rd International Workshop on Equation-Based Object-Oriented Languages and Tools*, Oslo, Norway, October 2010.

[7] http://www.iei.liu.se/flumes/system-simulation/hopsanng/, February 2012.

[8] P. Krus. Robust System Modelling Using Bi-lateral Delay Lines. In *Proceedings of the 2nd Conference on Modeling and Simulation for Safety and Security*, Linköping, Sweden, 2005.

[9] Air Force Aero Propulsion Laboratory. Aircraft hydraulic system dynamic analysis. Technical report, Air Force Aero Propulsion Laboratory, 1977.

[10] P.B. Johns and M.A. O'Brian. Use of the transmission line modelling (T.L.M) method to solve nonlinear lumped networks. *The Radio And Electronic Engineer*, 50(1/2):59–70, 1980.

[11] D.M. Auslander. Distributed system simulation with bilateral delay-line models. *Journal of Basic Engineering*, pages 195–200, June 1968.

[12] Petter Krus. Robust Modelling Using Bi-Lateral Delay Lines for High Speed Simulation of Complex Systems. In *DINAME 2011 : 14th International Symposium on Dynamic Problems in Mechanics*, 2011. Invited conference contribution.

[13] M. Vlach. Lu decomposition and forward-backward substitution of recursive bordered block diagonal matrices. *Electronic Circuits and Systems, IEE Proceedings G*, 132(1):24 –31, february 1985.

[14] M. Axin, R. Braun, A. Dell'Amico, B. Eriksson, P. Nordin, K. Pettersson, I. Staack, and P. Krus. Next Generation Simulation Software Using Transmission Line Elements. In *Fluid Power and Motion Control*, Bath, England, October 2010.

[15] B. Eriksson, P. Nordin, and P. Krus. Hopsan NG, A C++ Implementation Using The TLM Simulation Technique. In *The 51st Conference On Simulation And Modelling*, Oulu, Finland, 2010.

[16] David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E. Granger. Open source computer algebra systems: Sympy. *ACM Commun. Comput. Algebra*, 45(3/4):225–234, January 2012.

[17] SymPy Development Team. Sympy. http://sympy.org/, February 2012.

[18] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Computer Society Pr, 2006.