# The simulation engine

This document describes how the simulation engine can be constructed in Free Pascal. The simulation engine consists of several libraries that will be linked together into a single simulation engine. The engine is capable to be used without a graphical frontend but with command line arguments and to be compiled as a dynamically linked library so it can be used from other software as well e.g. Scilab, R, MS Excel, OpenOffice Calc, etc. One could even call the OpSim simulation engine from any programming language to be embedded in other projects.

**Variables**
A single variable is a fairly complex data structure internally. Not only does it has to hold the actual (calculated) value, but it needs some contextual information as described below.

| Variable | Type | Description |
|---|---|---|
| Derivative | Double | The derivative calculated for the variable |
| Description | String | A short description of the function of the variable |
| Default value | Double | The variable can be reset to this default variable. This can be easy when a variable's specification is switched from Free to Fixed. |
| Limit lower | Double | Used by the solver |
| Limit upper | Double | Used by the solver |
| Name | String | The local unique name of the variable |
| Scale | Double | Scaling factor for the variable, used by the solver |
| Specification | Integer | This variable determines whether the variable is Fixed (effectively constant) or Free (calculated by the solver). |
| TagName | String | The tag name describing the variable. This name is the concatenated value of the model name and variable name |
| Unit | | |
| Unit base | Pointer | The base unit the internal value is in |
| Unit selected | Pointer | The unit that is selected in the UI |
| Value | | |
| Internal | Double | The internal (raw) value |
| User defined | Double | The value according to the requested unit |

This record contains links to the unit conversion list. The data is stored in an internal unit and shown to the user in the desired unit.

**Equation parser**
The simulation engine is designed from the beginning to be easily extended with user made scripts. These scripts freely allow model blocks to be created and inherited. The equations in the models will be parsed and put into an equation stack which can be executed at any point in time. Evaluating such an equation is slower than evaluating a compiled function, but surprisingly enough this process is relatively efficient. A first test with an equation parser (Pars7) found in the public domain was carried out. The results which are shown below.

| Test machine | Compiled | Parsed | Performance | Optimized | Performance |
|---|---|---|---|---|---|
| i386-win32 | 5,0s | 13,0s | +159% | 7,8s | +53% |
| x86_64-linux | 2,6s | 4,4s | +69% | 2,2s | -14% |
| x86_64-windows | 3,9s | 10,6s | +170% | 5,3s | +36% |
| i386-darwin | 3,8s | 7,6s | +98% | 4,5s | +17% |

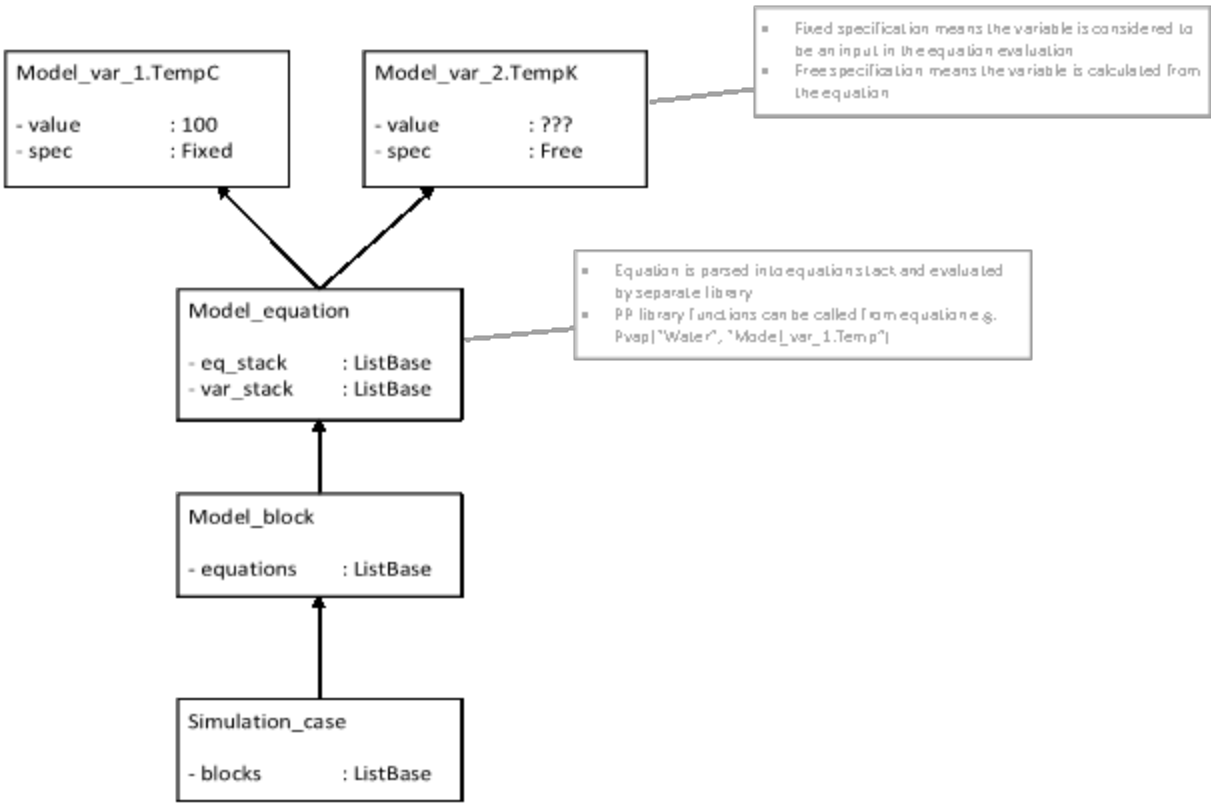
Each test comprised of 100 Mio evaluations for the simple function $\frac{-2}{3}x+5(x-4)-8$ .

There are differences in performance between the different platforms and architectures. A performance loss of between 69% and 170% compared to compiled function evaluation seems to be still acceptable. Especially as this is the performance out-of-the-box. There are numerous ways to optimize performance, for instance by caching results of previous runs to name a relatively easy enhancement.

Another benefit that parsed equations have over compiled, is the ability to do symbolic expression simplification. While the model equations are set up in a logical and human readable format, the computer should be able to evaluate (parts of) the equations and remove unneeded calculations. This will lead to a speed increase in the evaluation process. For example, simplifying $\frac{-2}{3}x+5(x-4)-8$ to $x-6.461538462$ results in a decrease from 12 operations to only 2 operations. This will reduce the performance loss between the parsed and compiled versions of the same equation as can be seen in the above table. The result is that the maximum decrease in performance seen is 53% but that on linux the speed is even better than the compiled alternative!

**Model structure overview**

Below the simplest representation of a complete model structure is shown. This representation will be used as a starting point to get to a working proof of concept. It is based on the assumption only mathematical equations are part of the model and no other programming statements are used.




The simulation case holds a list of model blocks. The model block is actually what is described by a script. A model block itself holds a list of model equations that are the parsed form of a human written equation. A single model equation holds the contents of the equation stack, which is a list of operations that the computer needs to perform in the order in which they are positioned on the stack. These operations can either be a mathematical operation, such as an addition, division but also taking the sine for example. It can also be a function from the PP library or any other library for that matter. (NOTE: this needs investigation how to do it in a uniform way. Possibly the PP library will generate "model" scripts that can be loaded by the simulation case). At some stage it should be possible to even create, and use, custom (dynamically linked) libraries in the equations. The equation stack will also hold a variable stack. This stack is a list of model variables that represent a value and specification. The specification of variables together with the number of equations determines the degrees of freedom for the overall model. The DOF should be 0 in any case or the model is indeterminate. When changing a specification from Fixed to Free another specification should be changed from Free to Fixed or another equation should be added to the model. The Fixed variables are considered as being inputs during equation evaluation.

**Example**
Consider the following equation;

$$TempC = TempK - 273{,}15$$

This will be parsed as one single operation variable;

| Variable | Value | Description |
|---|---|---|
| ARG1 | TempK | Pointer to first argument |
| ARG2 | 273,15 | Pointer to second argument |
| DEST | <calculated result> | Result from operation |
| NEXT | nil | Next operation |
| OP | Pointer to subtract function | Pointer to operation function |
| OPNUM | 5 | Value identifying the operation |

Calculating this equation is fairly straightforward. A value for TempK will immediately yield a value for TempC ± eps. However, if one changes the specs for TempK and TempC a solver will be needed to find the correct value for TempK that yields the desired TempC value. To reduce the load on the solver it would be needed to reorder the function in such a way so that the function result is TempK instead of TempC. This can be a difficult and even impossible task if the function is implicit. This functionality can be added at a later stage to optimize performance of the solver. A more general approach is to write the function in the following form;

$$residual = TempK - 273{,}15 - TempC$$

The residual is the error between the desired and estimated value. The solver will try to minimize the squared error by iterating several times and adjusting the value of TempK and calculating the residual error again. Once the residual is below a certain threshold the final value is found to the desired accuracy.

To ensure performance the residuals and Free variables will be linked to a separate linked list that will be traversed to update the model variables and calculate the residual error.

**Model complexity**

The downside of only using parsed equations for models is that other programming statements are harder to implement. This means that conditional statements like if…then and while, or loops like for…next need to be implemented separately. Also more complex features like for instance inheritance is more difficult.

To get more complex features in OpSim one might look at embedding a scripting language such as Python or Lua. OpSim however requires a modified scripting language that allows parts of the script to be used for setting up the model structure (like inheritance) and other parts of the scripts to be used by the model equations. In Python there is something like the Abstract Sytax Tree (ast package: https://docs.python.org/3.5/library/ast.html) which should make it possible for the application to do this. Possibly Lua will also allow for this. The advantage of using Python or Lua is that the interpreter is well developed and mature and has an active developer base. Also the scripting language are widely known and extensively documented and will pose a low threshold for new users. Also a lot of software packages are available (like NumPy) that can be used as well.

However OpSim will not be able to fully implement a scripting language but will need to implement only a subset. For instance, using Python classes would mean too much and

unneeded complexity, unless there would be some base Python class that each new model would inherit from. We should strive for a fit for purpose script to make it easy for users to understand how to use it and limit the amount of (possible) syntax errors.

Next to embedding a well-known scripting language interpreter there is also the possibility to implement an interpreter in native pascal. The benefit of this is threefold;

1. There is no need any more to provide and distribute an external interpreter. In other words there will be no extra dependency on an external interpreter and no hassle when versions or dialects change to update. Also there will be no need to maintain multiple binaries for all different platforms and architectures OpSim is released for.
2. As the interpreter is written in Free Pascal, it will support all platforms and architectures that OpSim is distributed on. This means that only the FPC compiler is the limiting factor to distribute to a new platform and/or architecture.
3. New features can easily be added (or removed) as seemed suitable. The script language can be made fit for purpose.

Of course implementing a scripting language adds a lot to the complexity of the project, but so does embedding a scripting language. Also the burden on documenting the scripting language will be completely on the OpSim team's side but that would be the case for a large part anyway.

In the public domain several projects exist that provide an implementation of a scripting language.

| | |
|---|---|
| Pascal-lisp: | https://github.com/bobappleyard/pascal-lisp |
| WANT script: | http://www.want-tool.org |
| PascalScript: | https://github.com/remobjects/pascalscript |

For OpSim it will be important to come with a sustainable solution that is built up right from the beginnings. The proposal is to start simple and later add complexity, but make sure the foundation on which everything is built is correct.

**Plex and Pyacc**

Free Pascal comes with substitutes for the GNU projects Lex and YACC. They are called Plex and Pyacc and they can be used to generate compilers and regular expression analysers in Pascal instead of C. (http://wiki.freepascal.org/Plex_and_Pyacc)

Using Lex and YACC might be a sound basis for a native scripting language. We might even use the same lexer and parser scripts that Free Pascal is using and adjust them! However, Lex and YACC scripts are an art on their own, so for going this route (which seems to be the best if embedding a native scripting language) needs a dedicated and experienced developer.

Some public domain lexer/parser scripts:

- https://github.com/scaperoth/simple-mini-pascal-compiler

**Conclusion**

There is currently no good idea of how programming statements in model scripts should be implemented in OpSim internally. It is recognized however that programming statements are a necessity for more complex models. How to implement this needs more thought and planning. Overseeing all options for adding script support it seems that adding a native scripting support is the preferred way to go. The proposal is to construct the most basic scripting language for now that only parses model equations. We would do that by creating a Lex and YAC script which potentially can be found and adapted from the public domain.