LOCIS

An intuitive language for modeling and simulation

Arjun Ramesh
Atharv Bhosekar

Introduction

LOCIS was developed with an aim to integrate object oriented equation based modeling seamlessly with the finite element method (FEM) for distributed systems. The idea was to develop an intuitive language framework that solves systems of linear, non-linear, differential, differential algebraic and multiphysics partial differential equations (PDEs) in a unified environment. The language supports intelligible object-oriented modeling constructs, thereby allowing the user to easily develop components and connect them together. PDEs in LOCIS are coherently connected to the above-mentioned feature. For example, PDE variables are created as objects of a geometry model that can be combined with other systems of equations. This combination of features enables the modeling and simulation of a host of innovative, highly complex large-scale systems.

Keeping in mind the importance of efficient computations in simulating large scale systems, the framework has been developed using C++ as the primary programming language. In addition to efficient computations, ease of writing assembly code with C++ made it the best choice. Currently, LOCIS relies on Python for on-the-fly code compilation. However, these dependencies will eventually be removed and replaced with C++ modules for greater control and fine tunability.

Easy accessibility to cloud services like Azure (Microsoft) and Amazon Web Services (AWS), and the availability of many libraries for front end web development would make LOCIS an attractive platform for web based mathematical modeling and simulation applications. Moreover, cloud-based implementations benefit from ease of implementing more advanced features like High-performance computing (HPC) and machine learning. Although the current implementation is meant to run on a local computer, plans of an eventual deployment on a cloud computing service are underway

Features

LOCIS is still actively under development and several essential features will be added routinely. Here is a list of currently supported features:

- Solution of steady-state and dynamic models; this applies for PDEs as well. The same model can be run in dynamic or steady state mode.

- The lexer, parser and interpreter were developed from the ground-up for enabling the reporting of fine-tuned syntax and sematic errors. The parser rules are described in a concise macro-based definition language, thus improving maintainability and allowing addition of new grammar rules easily.

- Residuals and jacobians are compiled on-the-fly in memory via the Python C application programming interface (API). The Jacobians are generated using automatic differentiation with the use 'autograd' library in Python.

- Sundials KINSOL and Sundials IDA solvers are used for the solution of non-linear algebraic equations and differential algebraic equations (DAEs). These are industrial-strength solvers developed by Lawrence Livermore National Laboratory (LLNL) with over 40 years of development history. They are periodically updated and has a well-documented API.

- LOCIS is designed to be an object-oriented language, which allows users to develop complex nested models. The object-oriented structure is flattened out before the solution process.

- Separate sections within a model can be used to fix/guess variables and set parameters to specific values. For DAEs, initial conditions in the form of additional equations need to be specified in an initialization section.

- Currently, one dimensional non-linear PDEs with mixed boundary conditions can be solved. The standard Galerkin formulation is used to generate the residuals for the PDEs. The geometry is specified intuitively as a line with its corresponding segments. The PDE variables can be applied to all the segments or can be selectively chosen to apply on the user-specified segments. The object-oriented paradigm extends to the line geometry too, allowing the creation of array of lines. The system can be specified as a multivariable, multidomain coupled set of PDEs using the weak form as input. Linear, quadratic and cubic basis functions with numerical integration schemes of trapezoidal, quadratic and cubic are supported. The discretized PDEs along with the other model equations are solved simultaneously.

LOCIS under the hood

The framework comprises of the lexer, parser, abstract syntax tree (AST) data structure, interpreter, finite element engine, equation generator, function factory and the solver layer as its core components. Currently, the only external dependency is Python, which is used via the Python C API to generate functions (Residuals and Jacobians) at runtime. The parser is designed to be a LL(1) recursive descent parser with an option to be LL(k). The grammar for the language is specified in a modified extended backus naur form (EBNF) notation. The lexer is used to tokenize the input model stream to its smallest components, e.g. keywords, identifiers, operators etc. The parser, uses the lexer and processes the model file based on the grammar rules to generate the AST. The AST is a compact tree representation of the model and forms basis of further computations. In the interpreter, the AST is executed to perform functions such as fixing/guessing variables, setting parameter values and initializing objects/arrays. In the equation generator, the same AST is used to generate the equations by executing the tree under the equation section of the model. FEM equations are generated by summing up the contribution from each element individually, which allows for selecting the basis function and integration scheme at the element level. Time-dependent variables in the FEM formulation are treated implicitly and are not part of the discretization process. The equation generator also uses the interpreter to resolve any array indices in the equations, as these must be known prior to generating the residual or Jacobian. Once all the equations are generated, the residual and Jacobian functions are constructed and compiled into a module via the Python C API. The functions can then be called by the solvers when required. A graphical description of the LOCIS infrastructure layout is given below:

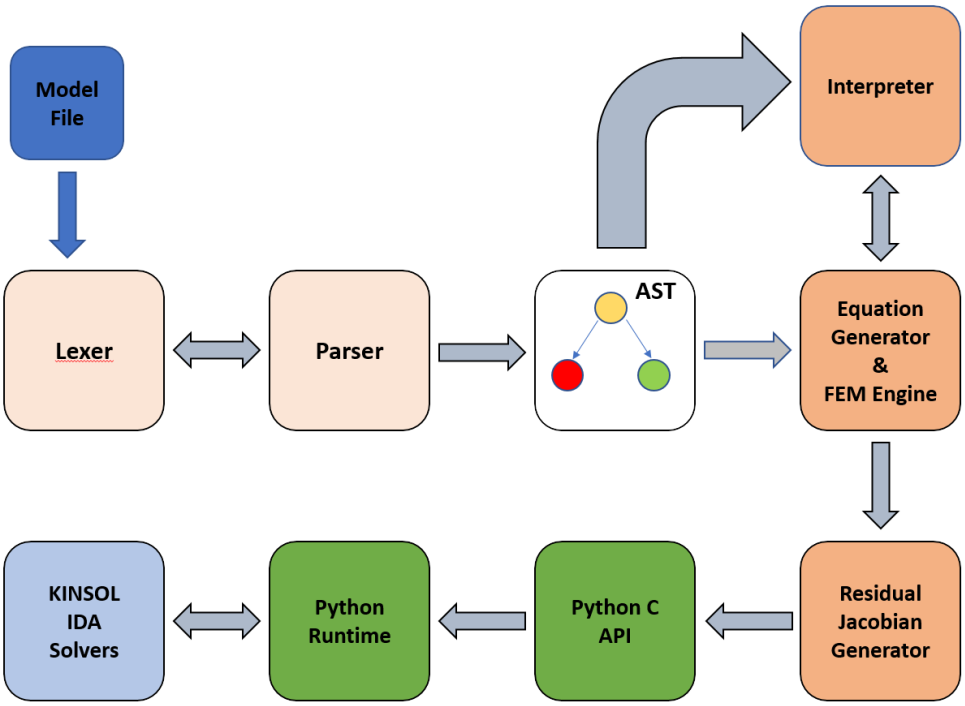


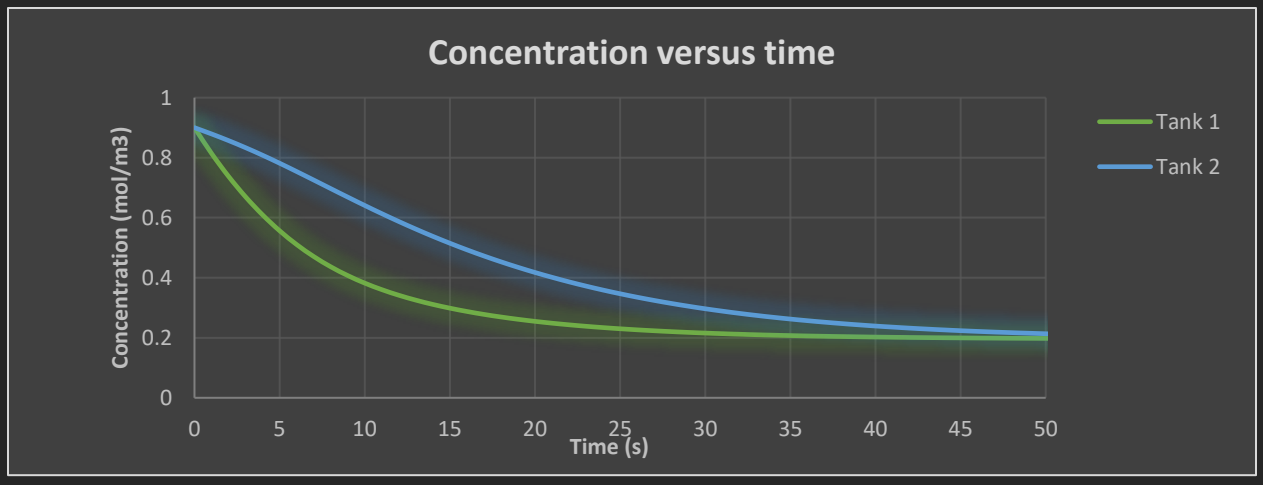
Fig 1.0 LOCIS Framework layout

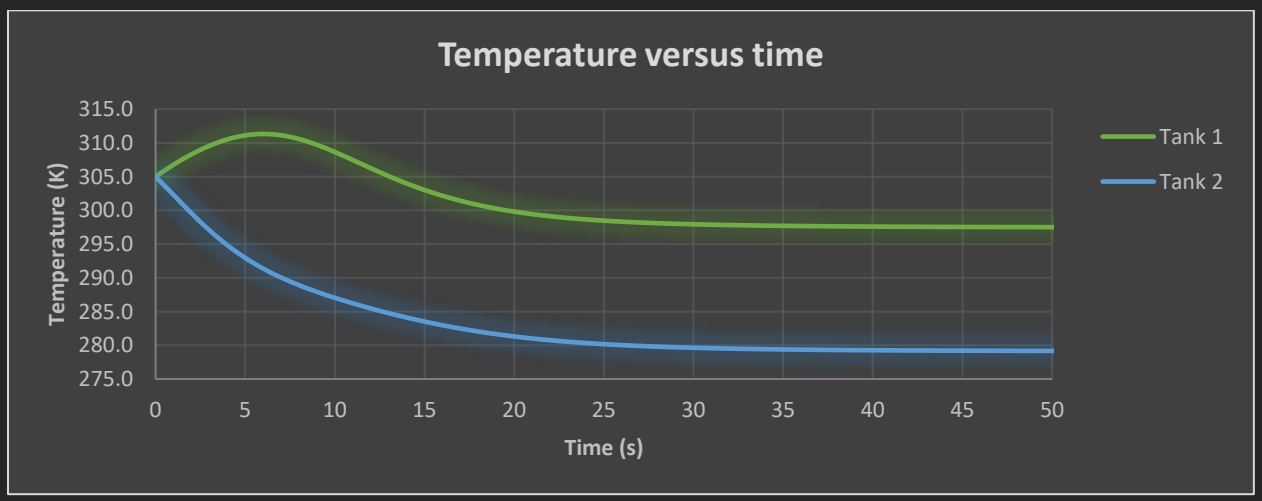
Example 1: Non-Isothermal CSTRs in series

```
# CSTR with energy balance
# Reaction A -> B
# Adapted from http://apmonitor.com/che436/index.php/Main/CaseStudyCSTR

model cstr
{
      parameter T_c = 270;
      parameter q = 10;
      parameter V = 100;
      parameter rho = 1000;
      parameter C_p = 0.239;
      parameter del_H = 5.0E4;
      parameter E_R = 8750;
      parameter k_o = 7.2E10;
      parameter U_a = 0.5E4;

      variable Ca_in;
      variable T_in;
      variable Ca;
      variable T;

      guess
      {
            Ca = 0.9;
            T = 305;
      }

      init
      {
            Ca = 0.9;
            T = 305;
      }

      equation
      {
            # Mole Balance
            V * $Ca = q * (Ca_in - Ca) - k_o * V * EXP(-E_R/T) * Ca^2;

            # Energy Balance
            rho * C_p * V * $T = q * rho * C_p * (T_in - T) + V * del_H * k_o
            * EXP(-E_R/T) * Ca + U_a * (T_c - T);
      }
}
```

CONTD...

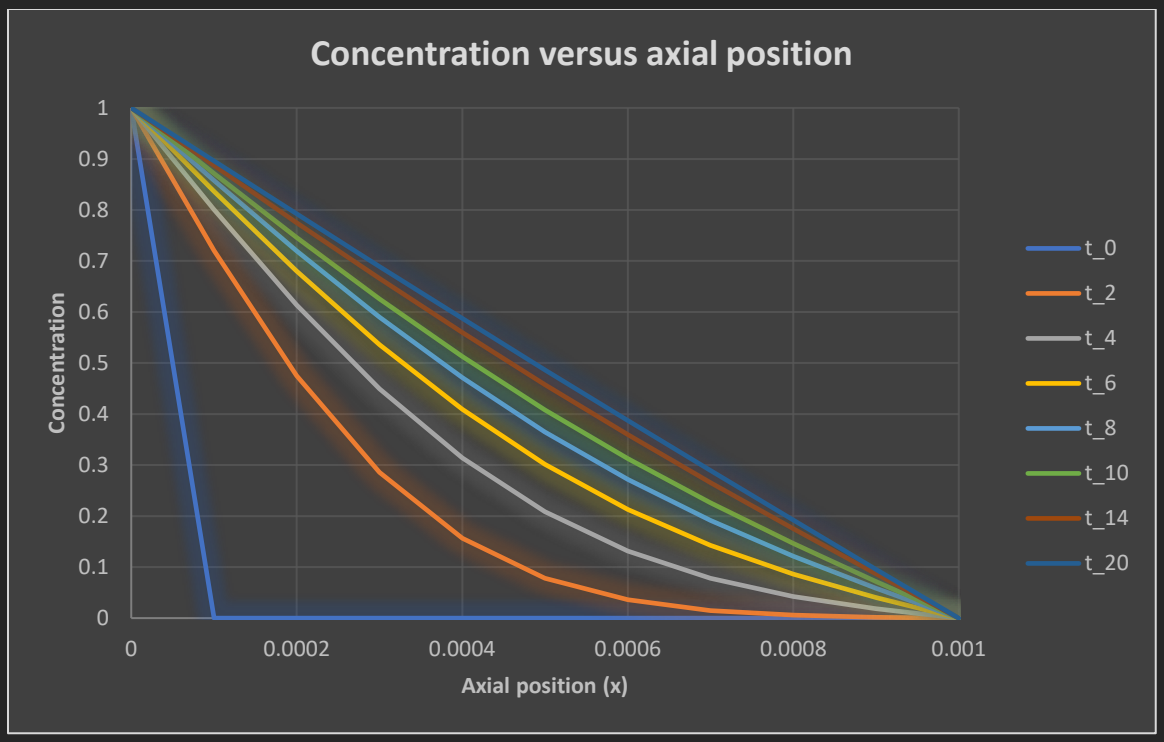
```
model tank_farm <DYNAMIC 0.0 100.0 100 1.0E-5 1.0E-5>
{
	parameter NUM_TANKS = 2;
	cstr Tanks[NUM_TANKS];
	iter i;

	fix
	{
		Tanks[1].Ca_in = 0.2;
		Tanks[1].T_in = 350;
	}

	equation
	{
		for(i = 1, NUM_TANKS - 1)
		{
			Tanks[i].Ca = Tanks[i+1].Ca_in;
			Tanks[i].T = Tanks[i+1].T_in;
		}
	}
}
```





Example 2: Fick's 2nd law

```
# Fick's 2nd law on a one-dimensional domain
# dC/dt = D*d^2C/dx^2; c(x = 0) = 1.0, c(x = L) = 0.0
#                     c(t = 0) = 0 for all 0 < x < L

model FEMoneDtest <DYNAMIC 0.0 2.0 20 1.0e-8 1.0e-8>
{
      line ficks_domain::(basis = "linear",
                  segment = {id = "water", length = 1.0e-3, numel = 10},
                  pdevar = {id = "Conc"});

      parameter DO2Water = 1.97E-7;

      iter i;

      guess
      {
            for(i = ficks_domain.water["left"],
            ficks_domain.water["right"])
            {
                  ficks_domain.water.$Conc[i] = 1.0;
                  ficks_domain.Conc[i] = 1.0;
            }
      }

      init
      {
            for(i = ficks_domain.water["left"]+1,
            ficks_domain.water["right"]-1)
            {
                  ficks_domain.water.Conc[i] = 0;
            }
            ficks_domain.water.$Conc["left"] = 0;
            ficks_domain.water.$Conc["right"] = 0;
      }

      equation
      {
            #dirichlet boundary conditions
            ficks_domain.water.Conc["left"] = 1;
            ficks_domain.water.Conc["right"] = 0;

            PDE, ficks_domain.water.Conc, VALUE, VALUE:
            DO2Water*ficks_domain.Conc'*_v'   +   DO2Water*ficks_domain.Conc*_v
            + ficks_domain.$Conc*_v = 0, 0, 0;
      }
}
```


Concentration versus axial position
Concentration
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.0002
0.0004
0.0006
0.0008
0.001
Axial position (x)
t_0
t_2
t_4
t_6
t_8
t_10
t_14
t_20

Features to be added

The features mentioned below are mentioned in order of highest to lowest priority.

- Implementation of a code generator /virtual machine infrastructure capable of creating callable residual and Jacobian functions in memory, thereby eliminating the Python dependency.

- Support for two-dimensional finite element method with intuitive geometry input. Users will be able to specify the geometry through a simple set of drawing functions. The implementation will include all the features currently supported in the one-dimensional case.

- Support for the solution of sparse systems through the “suiteSparse” set of linear sparse solvers.

- Development of an in-house built automatic differentiation engine capable of both forward and reverse mode differentiation. This will replace the Python “autograd” package.

- Cloud-based deployment of the LOCIS engine by using the Python-based Django back end framework and HTML/Javascript/CSS for the front end.

- Implementation of an optimization module that supports features akin to frameworks like GAMS, AMPL etc., along with additional support for robust dynamic optimization, for e.g., automatic discretization of the time domain.

- Implementation of the Pantelides and dummy derivative algorithms for DAE index reduction.

- Implementation of the Block Lower Triangular (BLT) algorithm for solving the system of equations in block decomposition mode. Specifically, this breaks the original system into many smaller sub-systems, which can be solved sequentially.

- Addition of an event-tracking framework for handling discontinuities.

- Implementation of a parameter estimation module with support for a variety of data input formats.

- Development of equation pruning algorithms for reducing the final number of equations to be solved. This includes removing redundant variables such as those arising from connection equations.

- Addition of ports and streams for easy connectivity between model objects.

- Addition of an HPC Framework for the solution of large-scale systems and exploring graphics processing unit (GPU) computing.

- Implementation of a machine learning infrastructure through the “Tensorflow” library. This will enable the use of deep learning for artificial neural networks (ANNs) in the context of model order reduction.