UNIVERSITÀ DEGLI STUDI DI PALERMO

Meccanica dei Materiali e delle Strutture
Vol. 3 (2012), no.4, pp. 10-17
ISSN: 2035-679X
Dipartimento di Ingegneria Civile, Ambientale, Aerospaziale, Dei Materiali
DICAM

SLANGTNG - SOFTWARE FOR STOCHASTIC STRUCTURAL ANALYSIS MADE EASY

Christian Bucher[*], Sebastian Wolff[†]

[*] Center of Mechanics and Structural Dynamics
Vienna University of Technology
Karlsplatz 13, A-1040 Vienna, Austria
e-mail: christian.bucher@tuwien.ac.at

[†] DYNARDO Austria GmbH
Wagenseilgasse 14, A-1120 Vienna, Austria
e-mail: sebastian.wolff@dynardo.at

(Ricevuto 10 Giugno 2012, Accettato 10 Ottobre 2012)

Key words: computational methods, structural mechanics, stochastic analysis, probability theory, software development.

Abstract. *Most engineering problems are so complex that the solution requires the application of computer-based numerical algorithms. For research purposes (particularly for algorithmic developments) interpreted scripting languages are chosen as the primary tools. While this enables rapid prototyping of the algorithms, it typically leads to substantial loss of computational performance as compared to solutions based on compiled languages. Hence, the final versions of the algorithms are frequently re-coded in a compilable language. This process, however, may involve quite substantial re-organization of the flow of execution, and possible introduces unwanted errors. This paper presents an innovative approach to bringing interpreted and compiled languages close together. Applications to simple random vibration analysis demonstrate the applicability and potential of this new approach.*

1 INTRODUCTION

In many engineering application there is an increasing demand on the availability of tools to incorporate unavoidable random variability of loads and system properties into the workflow of structural analysis. This requires a close relation between the data structures as required for traditional Finite Element analyses and the stochastics tool required to obtain a suitable statistical description of the relevant responses. This is readily achievable by using established software development environments such as e.g. C++. Due to the required compilation process and the possibly code optimization associated with it, the computational performance can be quite impressive. On the other hand, the compile-link-cycles do not allow for quick checks how minor algorithmic modifications or extensions affect the quality of the desired results. This is particularly annoying when developing larger software projects in a distributed

work environment, since each compile-link must check for potential changes in dependent modules which may lead to substantial delays.
It turns out that such algorithmic modifications or checking steps can be much faster performed using an interpreted scripting language, albeit at some loss of algorithmic performance. Typically this is not a real problem because test examples are usually chosen small enough not run into performance problems. A fairly thorough discussion on the use of scripting languages in computational science is given e.g. in (Langtangen 2008).
This paper focuses on the development of a C++ module library for structural, mathematical, and statistical analysis including graphics named slangTNG which can be driven through a scripting language as well. For performance reasons, the scripting language lua (Ierusalimschy 2006) was chosen. Since the flow control of lua is not too far from the flow of C++, it is fairly straightforward to convert pieces of lua-Code to C++-code carrying out the same task. This is quite useful once the algorithm is fixed and computational performance must be enhanced. The major advantages of lua can be summarized as follows:

- Very fast scripting language
- Popular for scripting of 3D games, recent developments of TEX and friends
- Lightweight basic interpreter code, compiles to a few kB
- Easily embeddable into final product, thus providing stand-alone solutions
- Flow structure close to C conventions (for, if, while, …)

However, there are of course also some things missing:

- lua contains only simple math functions (sin, cos, exp,...)
- Extensions for stochastic structural analysis needed

A previous software project in which the authors were involved (SLang - the Structural Language) has been presented in (Bucher, Schorling, and Wall 1995). For a current commercial software project (optiSLang 2012), the reliability analysis module was developed exactly in this way, i.e. by implementing and testing the algorithms in slangTNG. During this test phase, time-consuming compile-link-cycles could be eliminated thus sppeding up the development process significantly. After finalizing the algorithms in script form, they were subsequently transferred into fast-running C++.

2 CONNECTING COMPLIED AND SCRIPTED VERSIONS

Since compiled and scripted versions of an algorithm rely on substantially different realizations in computer code, any scripting language requires some “glue”-code with connects the data structures of the script interpreter to the data structures of the compiled object library. Establishing and maintaining this glue code can be substantial effort, particularly is parts of the class interfaces are changing during the development process. It is therefore helpful to utilize an automatic binding process. For the software package slangTNG, this binding of the C++ code to the scripting language **lua** is performed automatically using **swig** (SWIG Documentation 2012). Several tests showed that the wrapper code generated is fast and efficient for virtually all practical cases. A further advantage of **swig** is the fact that bindings to other scripting languages such as **python** can be generated without additional effort.

The general procedure can be summarized as follows:

- Define class methods in header file “class.hpp” as usual
- Prepare a SWIG input file “class.i” to define the headers to be scanned (independent of scripting language)
- SWIG reads these header file and generates C++ glue code “wrap_class.cxx” for all public methods (for specified scripting language)
- Wrapper code plus C++ implementation of class “class.cpp” are assembled in library
- Library is loaded by the language interpreter
- In **lua**, classes can be accessed through **lua** tables

This is shown schematically in in Fig.1.

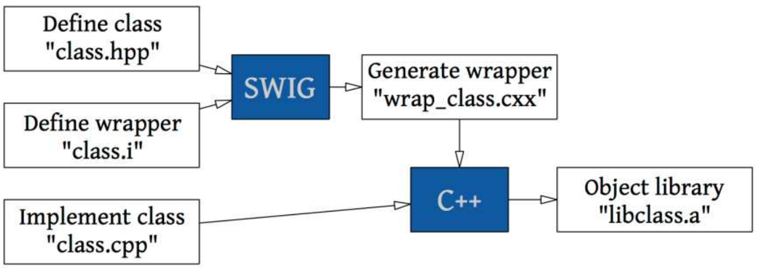

Define class "class.hpp"
Define wrapper "class.i"
Implement class "class.cpp"
SWIG
Generate wrapper "wrap_class.cxx"
C++
Object library "libclass.a"

Figure 1: Generating wrapper code for the lua interpreter.

In order to access and use the C++ classes contained in this wrapped module library, the following steps are required:

- From C++ code, allocate new **lua** interpreter in “main.cpp”
- Make class methods accessible to **lua** by calling “lua_openclass()”
- Use class in scripts “use_class.tng” passed to the newly created **lua** interpreter

This procedure is shown schematically in Fig.2.

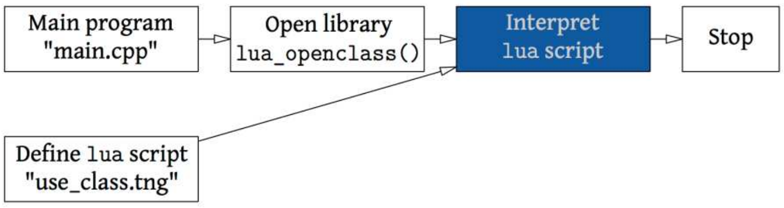

Main program "main.cpp"
Open library lua_openclass()
Interpret lua script
Stop
Define lua script "use_class.tng"

Figure 2: Opening lua module for the interpreter.

In order to demonstrate the close relation between the C++ implementation and the lua script version, consider the definition and simulation of a Gaussian random variable with a mean value of 1 and a standard deviation of 0.5. The code snipped as shown in Fig.3 shows an implementation of this process in C++. It can be seen that typical C++ features are utilized such as the use of class constructors and class methods. Note that in the background, the specific class **RanvarNormal** inherits from a more general class **Ranvar**. The C++ code needs to be compiled and linked against all required libraries before it can be tested and applied.

```
    // Create and simulate random variable
    stoch::RanvarNormal rv();   //Constructor of class RanvarNormal
    tmath::Matrix s(2);   // Constructor of class Matrix
    s[0] = 1; s[1] = 0.5;   // Accessor of class Matrix
    rv.SetStats(s);     // Method of class RanvarNormal
    tmath::Matrix r = rv.Simulate(100); // Method of class RanvarNormal
    // Access values in a loop
    for (int i=0; i<100; ++i) { // Loop construct
      printf("i: %d, r %g\n", i, r[i]);
      }
```

Figure 3: C++ code for simulation of a Gaussian random variable.

By binding this C++ code to the **lua** interpreter as outlined above, this process can be scripted and run from the **lua** intepreter. The SWIG input file required for the wrapping process is shown in Fig.4.

```
%module stoch
%{ /* The includes required for the wrapper code: */
#include "stoch/simulate/ranvar.hpp"
%}

/* Our classes and methods to be wrapped: */
%include "stoch/simulate/ranvar.hpp"
```

F

Figure 4: SWIG input file required to bind Ranvar class.

The **lua** code which is then usable from slangTNG is shown in Fig.5.

```
    -- Create and simulate random variables
    rv = stoch.RanvarNormal()   -- Constructor of class RanvarNormal
    s = tmath.Matrix(2) -- Constructor of class Matrix
    s[0] = 1 s[1] = 0.5   -- Accessor of class Matrix
    rv:SetStats(s)  -- Method of class RanvarNormal
    r = rv:Simulate(100)  -- Method of class RanvarNormal
    -- Access values in a loop
    for i=0,99 do
      print("i", i, "r", r[i])  -- Loop construct
    end
```

Figure 5: **lua** code (slangTNG) for simulation of a Gaussian random variable.

3 SIMULATION OF THE TRANSIENT RESPONSE OF A DUFFING OSCILLATOR

The equation of motion of a Duffing oscillator is given by

$$m\ddot{x} + c\dot{x} + kx + \varepsilon k x^3 = f(t) \tag{1}$$

Here x is the displacement, m denotes the mass, k the linear stiffness, ε is the nonlinearity parameter and $f(t)$ denotes the excitation process. In the following it is assumed, that the excitation is a stationary random process with a given power spectral density:

$$S_{ff}(\omega) = \frac{S_0}{1+\frac{\omega^4}{a^4}} \tag{2}$$

The initial conditions for the oscillator are assumed to be quiescent, therefore the response will exhibit a transient phase while asymptotically reaching stationarity. The excitation process is simulated using the classical Rice formulation (see e.g. Bucher, 2009), and the responses are computed using a fourth-order Runge-Kutta explicit integrator.

The slangTNG code carrying out the simulation is shown in Fig.6 and the code for the reponse analysis is shown in Fig.7.

```
--[[
SLangTNG
Simple test example for simulation of random processes
(c) 2009 - 2012 Christian Bucher, CMSD-VUT
--]]

-- This function defines the two-sided PSD of the process
function PSD (S, a, b)
  local p = S/(1+(b/a)^4)
  return p
  end

-- This function simulates one sample of the random process
function process(S0, om0, om_max, nOmega)
  dOmega = om_max/nOmega

-- Fill an array with PSD values
  spec = tmath.Matrix(nOmega)
  var = 0
  for i=0,nOmega-1 do
    spec[i] = PSD(S0, om0, (i+.5)*dOmega)
    var = var + 2*spec[i]*dOmega
    end

-- Generate random Fourier coefficients
  a = stoch.Simulate(nOmega,1)
  b = stoch.Simulate(nOmega,1)
  c = tmath.Pow(spec*2*dOmega, 0.5):CW()*a
  s = tmath.Pow(spec*2*dOmega, 0.5):CW()*b

-- Assemble real and imaginary parts, apply inverse Fourier transform
  help = c:AppendCols(s)*math.sqrt(nOmega/2)
  f, dt = spectral.IFT(help,dOmega)
    return f, dt
end
```

Figure 6: Sample code for Monte-Carlo simulation of a stationary random process with given power spectral density function.

```
--[[
Compute transient statistics of the response of a Duffing oscillator
to a random process with given PSD

(c) 2012 Christian Bucher, Vienna University of Technology
--]]

-- Load the code for generating the excitation
dofile("process.tng")

-- define the Duffing systems in terms of derivatives of state variables
function duffing(t, y)
    local m = 1
    local k = 1
    local c = 0.01
    local eps = 0.1
    local index = t/dt
    force = f[index]
    local z = tmath.Matrix(2)
    z[0] = y[1]
    z[1] = -c/m*y[1] - k/m*(y[0] + eps*y[0]^3) + force/m
    return z
end

-- define the excitation
S0 = 1
om0 = 3
om_max = 10
nOmega = 501
NT = 2*(nOmega-1)

-- Simulate the excitation and compute the response by Runge-Kutta
NSIM = 300
disps = tmath.Matrix(NT, NSIM)
for i=0, NSIM-1 do
    f, dt = process(S0, om0, om_max, nOmega)
    T = NT*dt
    diff_eq = ode.RK4(2, "duffing")
    response = diff_eq:Compute(0, T-dt, NT, 2) -- use 2 substeps
    disps:SetCols(response:GetRows(0):Transpose(), i)
    print("i", i)
end
mean = stoch.Mean(disps)
sigma = stoch.Sigma(disps)
t = tmath.Matrix(NT)
t:SetLinearRows(0, T-dt)

vis = graph.Graph("Transient Statistics", "Bright")
vis:AxisLabels("Time [s]", "Displacements [m]")
vis:Plot(t, mean, 1, "Mean")
vis:Plot(t, sigma, 1, "Sigma")
vis:PDF("Stats.pdf")
```

Figure 7: Sample code for carrying out Monte-Carlo simulation of a Duffing oscillator.

The results of the simulation based on 300 samples are shown in Fig.8. It can be seen that the standard deviation approaches the stationary solution well within the time frame as analyzed.

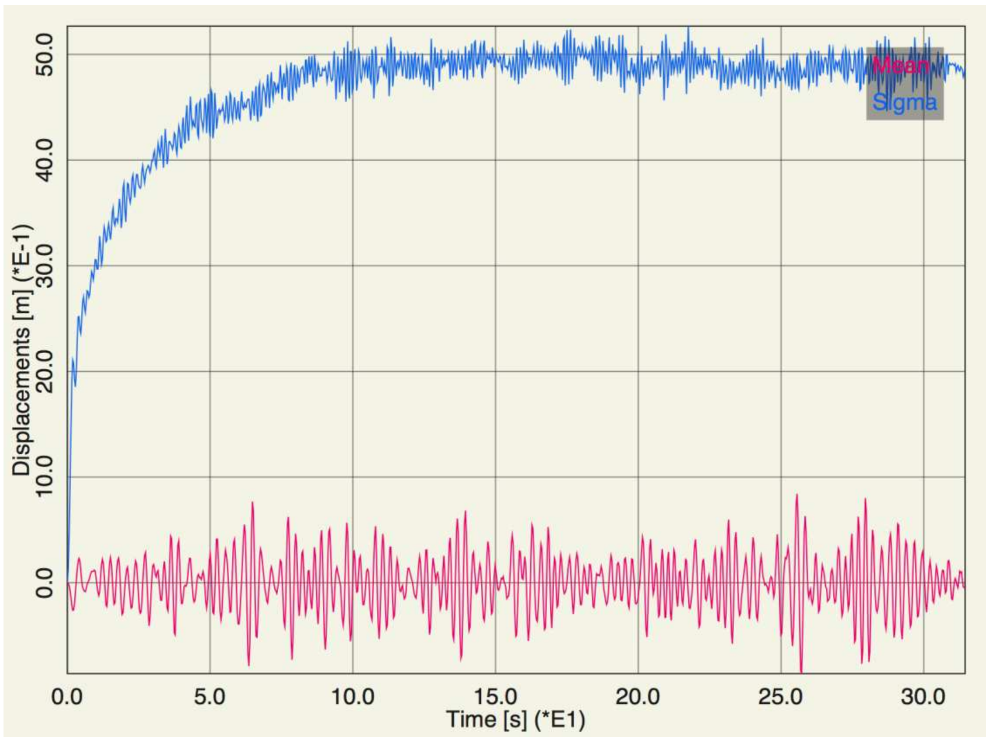


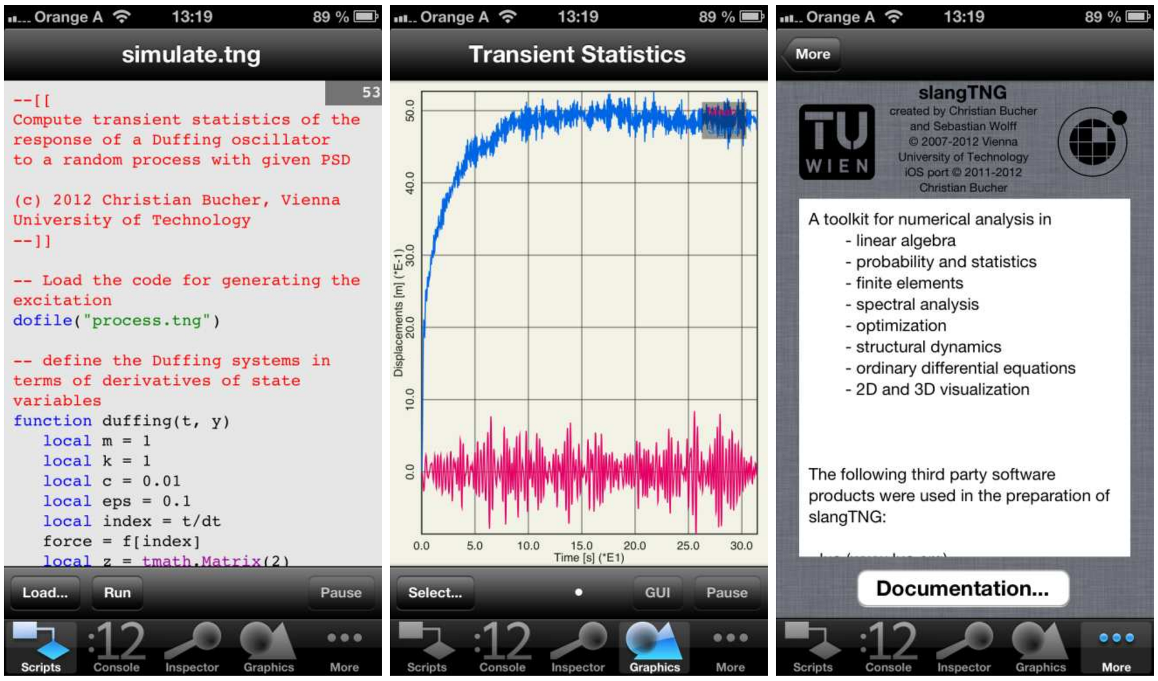
Figure 8: Mean value and standard deviation of the transient stochastic response of a Duffing oscillator.

4 CONCLUDING REMARKS

The software project slangTNG demonstrates that it is fairly easy to establish and maintain a stable connection between code written in a compiled language (C++) and an interpreted language (lua). This enables fast development cycles regarding the implementation of new or modified algorithms for stochastic structural analysis using scripting and yet allows for a smooth transition to compiled versions of these algorithms.

The software is in the public domain (BSD-style license) and can be downloaded from http://tng.tuxfamily.org. Ready-made binaries for Mac OSX and Windows are available from the first author's homepage at Vienna University of Technology http://info.tuwien.ac.at/bucher/Private/slangTNG.html. An iOS version is available on the Apple App Store.

Screenshots of the iOS version are given in Fig.9.



```
--[[
Compute transient statistics of the
response of a Duffing oscillator
to a random process with given PSD

(c) 2012 Christian Bucher, Vienna
University of Technology
--]]

-- Load the code for generating the
excitation
dofile("process.tng")

-- define the Duffing systems in
terms of derivatives of state
variables
function duffing(t, y)
   local m = 1
   local k = 1
   local c = 0.01
   local eps = 0.1
   local index = t/dt
   force = f[index]
   local z = tmath.Matrix(2)
```

slangTNG
created by Christian Bucher
and Sebastian Wolff
© 2007-2012 Vienna
University of Technology
iOS port © 2011-2012
Christian Bucher

A toolkit for numerical analysis in
- linear algebra
- probability and statistics
- finite elements
- spectral analysis
- optimization
- structural dynamics
- ordinary differential equations
- 2D and 3D visualization

The following third party software products were used in the preparation of slangTNG:

Figure 9: Screen shots of slangTNG on iOS.

5 ACKNOWLEDGMENT

The authors would like acknowledge financial support from the Austrian Science Funds (FWF) as part of the Vienna Doctoral Programme on Water Resource Systems (DK-plus W1219-N22).

REFERENCES

[1] Langtangen, H. P. (2008). *Python Scripting for Computational Science*. Springer.
[2] Bucher, C., Y. Schorling, and W. A. Wall (1995). "SLang–the Structural Language, a tool for computational stochastic structural analysis". In: *Engineering Mechanics, Proceedings of the 10th Conference*. Ed. By S. Sture. ASCE, pp. 1123–1126.
[3] *optiSLang*. URL: http://www.dynardo.de/en/software/optislang.html (visited on 10/10/2012).
[4] Ierusalimschy, R. (2006). *Programming in Lua*. 2nd. Rio de Janeiro: lua.org.
[5] *SWIG Documentation*. URL: http://www.swig.org/Doc2.0/SWIGDocumentation.html (visited on 06/21/2012).
[6] Bucher, C. (2009). Computational analysis of randomness in structural mechanics. Ed. By D. M. Frangopol. Structures and Infrastructures Book Series, Vol. 3. London: Taylor & Francis.