slangTNG

Introduction

Christian Bucher and Sebastian Wolff
Center of Mechanics and Structural Dynamics

September 10, 2012

Contents

1 Introductory examples

1.1 General concept

slangTNG is a scripting language for a variety of numerical problems related to mathematics and structural mechanics based on Lua[1]. Actually, slangTNG provides additional functionality to Lua by wrapping C++ functions (involving additional C and FORTRAN libraries) in such a way that the C++ objects and methods are accessible from the Lua interpreter. This is done by an automatic wrapping process using SWIG[2]. In addition to the mathematical algorithms, there is a binding to a GUI system (provided by Nokia's QT toolkit (on MacOS, Windows, Linux) or Apple's UIkit (iPhone, iPod, iPad).

This document describes the basic features of slangTNG by solving a selected set of simple problems related to mathematics, statistics, optimization, time series analysis, finite elements, and structural dynamics.

It is assumed that an executable program (slangTNG-application) with the name `slangTNG` is available. From a terminal, you can run a script, say `intro.tng` with the command

```
slangTNG intro.tng
```

Depending on your system configuration you may need to provide the full path to the slangTNG-application.

Alternatively, you can start slangTNG by double-clicking or tapping its icon, and choose the script to be run from a file chooser inside slangTNG.

1.2 flow.tng

This section describes the basic flow control elements available in slangTNG. These are actually `Lua` constructs such as loops, functions. Note that the `Lua`-interpreter first preprocesses the entire input file. Here elementary syntax checks are performed. In a second pass, the file is actually interpreted. At this stage, errors related to the actual functions to be performed may occur.

The following listing shows the computation of $n!$ by a loop construct involving a function call.

```lua
--[[
slangTNG
Simple test example demonstrating flow control in lua
It computes n factorial
(c) 2009 - 2012 Christian Bucher, Vienna University of Technology
--]]

-- This is a function returning two variables, the first is a number, the second
    a bool
function dummy(k)
  return k, k>1
  end

N=10
n = N

-- check if any computation is needed
i, go_on = dummy(n)
result = n

-- enter loop depending on bool go_on
while(go_on) do
```

[1]see `www.lua.org`
[2]see `www.swig.org`

```
  n = n-1
-- Call function to determine further steps
  i, go_on = dummy(n)
-- Accumulate product
  result = result*i
  end
-- Output result
print("N:", N, "result:", result)
```

The resulting output to the console is:

```
N: 10 result: 3628800
```

1.3 SimpleMath.tng

Let us assume that we would like to compute the functions $f_k(x)$ in the interval $x \in [0, 2\pi]$. The functions are $f_1(x) = \sin x$, $f_2(x) = \sin 2x$, $f_3(x) = \sin 4x$ and $f_4(x) = \exp(0.001 \cdot x)$. We compute these functions for 100 discrete values of x in the given interval, and then plot the following:

- f_2 vs. f_1
- f_3 vs. f_1
- f_4 vs. $\frac{x}{6}$

```
--[[
slangTNG - demonstration of simple mathematical functions and plotting
    capabilities
(c) 2012 Christian Bucher, Vienna University of Technology
--]]

-- Fill a vector with values from zero to 2*pi
n = 100
x = tmath.Matrix(n)
x:SetLinearRows(0,2*math.pi)

-- Apply functions to this vector
f1 = tmath.Sin(x)
f2 = tmath.Sin(x*2)
f3 = tmath.Sin(x*4)
f4 = tmath.Exp(x)*.001

-- Plot functions ad output to PDF file
gr = graph.Graph("My First Plot")
gr:AxisLabels("X-axis", "Several Functions")
gr:Plot(f1, f2, 1, "sin(x) vs. sin(2x)")
gr:Plot(f1, f3, 2, "similar stuff")
gr:Plot(x/6., f4, -3, "exponential")
gr:PDF("My_First_Plot.pdf")
```

The plot generated by slangTNG is shown in Fig. 1.

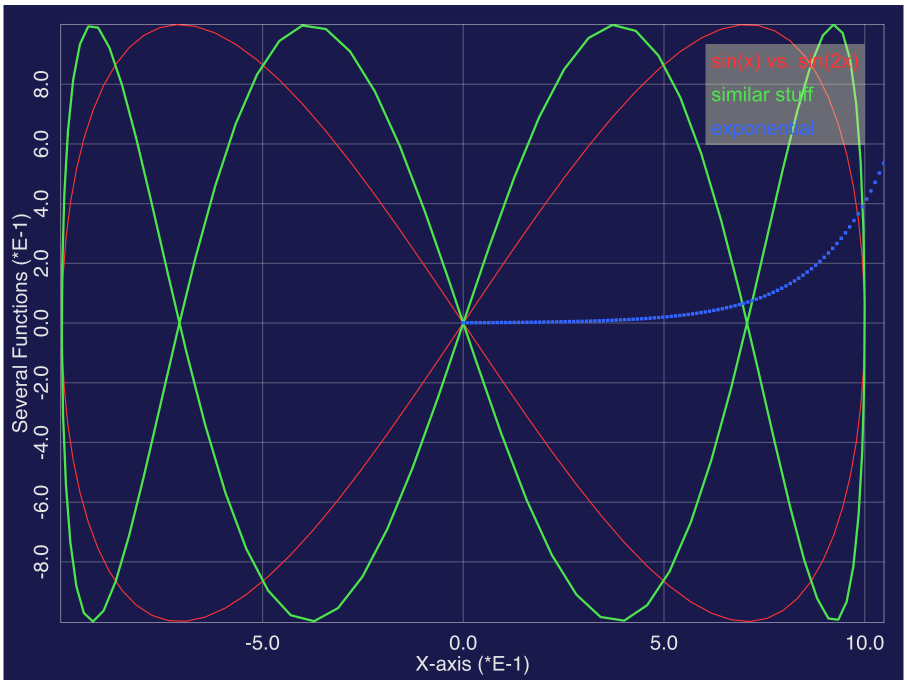

Figure 1: Graphics generated by slangTNG

1.4 MC.tng

This example shows the application of a Monte Carlo simulation to a problem involving two correlated random variables. Variable `rv1` is log-normally distributed with a mean value of 1 and a standard deviation of 0.5, variable `rv2` is Gaussian with the same mean and standard deviation. Both variables are correlated with a coefficient o correlation $\rho_{12} = 0.5$.

The script generates 300 samples of these variables, shows the samples in a scatter plot and checks the statistics of the generated samples. A randomized Sobol sequence is used to generate the samples.

```
-- Create lognormal random variable
rv1=stoch.RanvarLognormal()
-- set mean value to 1, standard deviation to 0.5
rv1:SetStats(tmath.Matrix({{1, 0.5}}))
-- Create normal random variable
rv2 = stoch.RanvarNormal()
-- set mean value to 1, standard deviation to 0.5
rv2:SetStats(tmath.Matrix({{1, 0.5}}))
-- Produce samples for both random variables
NSIM = 300
sample1 = rv1:Simulate(NSIM)
-- Estimate mean value and standard deviation
m1 = stoch.Mean(sample1)
s1 = stoch.Sigma(sample1)
-- print statistics and target
print("mean value is", m1[0], "should be", 1)
print("standard deviation is", s1[0], "should be", 0.5)

-- Assemble both random variables into a random vector
vec=stoch.Ranvec()
vec:AddRanvar(rv1)
vec:AddRanvar(rv2)
```

```
-- Define correlation matrix
rho = 0.5
corr = tmath.Matrix({
  {1, rho},
  {rho, 1}
  })
-- Assign correlation to random vector
vec:SetCorrelation(corr)
-- Simulate random vector and estimate statistics
sample = vec:Simulate(NSIM, stoch.Sobol)
mean = stoch.Mean(sample)
print("mean vector", mean)
sigma = stoch.Sigma(sample)
print("standard deviation", sigma)
scorr = stoch.Correlation(sample)
print("correlation matrix", scorr)

-- Output samples to text file
tmath.Output(sample:Transpose(), "MC_samples.txt")
-- Draw scatterplot
vis=graph.Graph("Scatter Plot", "Bright")
vis:AxisLabels("Variable 1", "Variable 2")
vis:Plot(sample:GetRows(0), sample:GetRows(1), -4, "Monte Carlo Sample")
vis:PDF("scatter.pdf")
```

The scatterplot generated by the script is shown in Fig. 2.

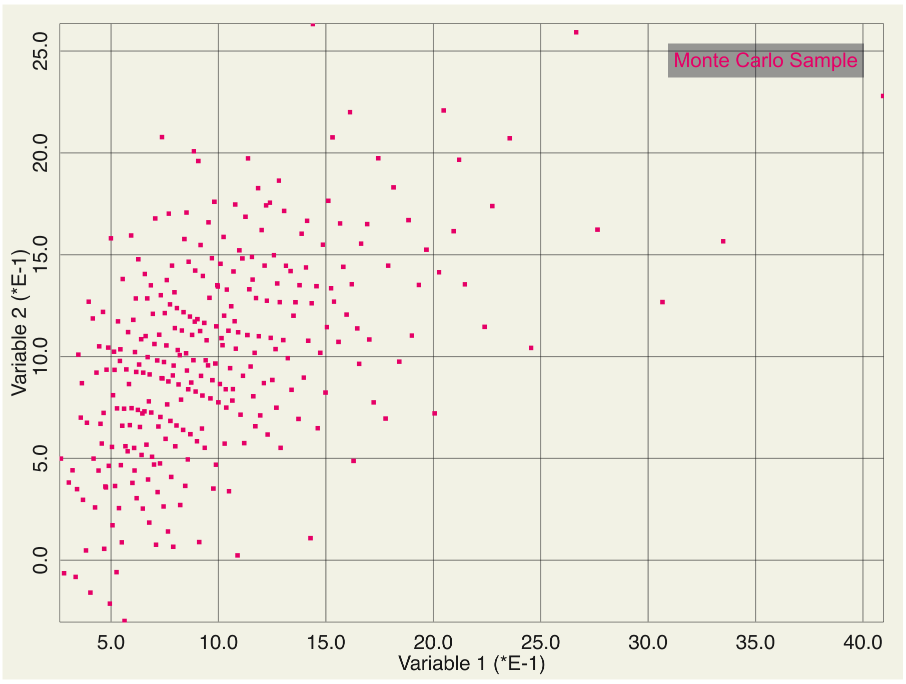

Figure 2: Random samples generated by slangTNG

The text output to the console is:

```
mean value is 1.0243909912693 should be 1
standard deviation is 0.49030993019579 should be 0.5
mean vector Matrix 2x1
```

```
1.00889
1.01068

standard deviation Matrix 2x1
0.520209
0.504662

correlation matrix Matrix 2x2
1 0.515067
0.515067 1
```

Of course, these values will be slightly different each time the script is run.

1.5 frame.tng

This example demonstrates user interaction in slangTNG together with static structural analysis using finite beam elements. The structure under consideration is a portal frame subjected to a horizontal load H and a vertical load V as shown in Fig. 3. The finite element mesh is extremely

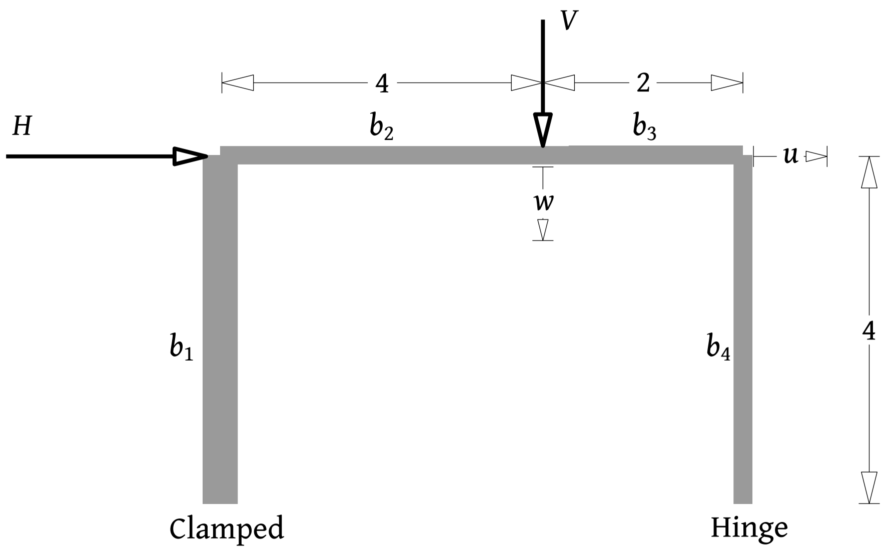

Figure 3: Simple portal frame

simple and consists of only 4 elements as shown in Fig. 4. Note that the node with ID 16 (which is actually not part of the structure) is required to give the beam elements a unique positioning in 3D space. Without this reference node, the beams could be arbitrarily rotated about their longitudinal axis. The reference node together with the two actual nodes defines the local $x-y$-plane for the beam element, and therefore must not lie on the line connecting the two actual nodes. The material is assumed to be linear elastic with a modulus of elasticity E = 210 GPa, Poisson's ration ν = 0.3 and a mass density ρ = 7850 kg/m^3. Cross sections are squares with $b_1 = b_2$ = 0.05 m and $b_3 = b_4$ = 0.08 m.

The applied loads can be changed interactively by entering values into the NumberInputBoxes labelled H and V

```
--[[
SLangTNG
Simple test example for Finite Element analysis
```

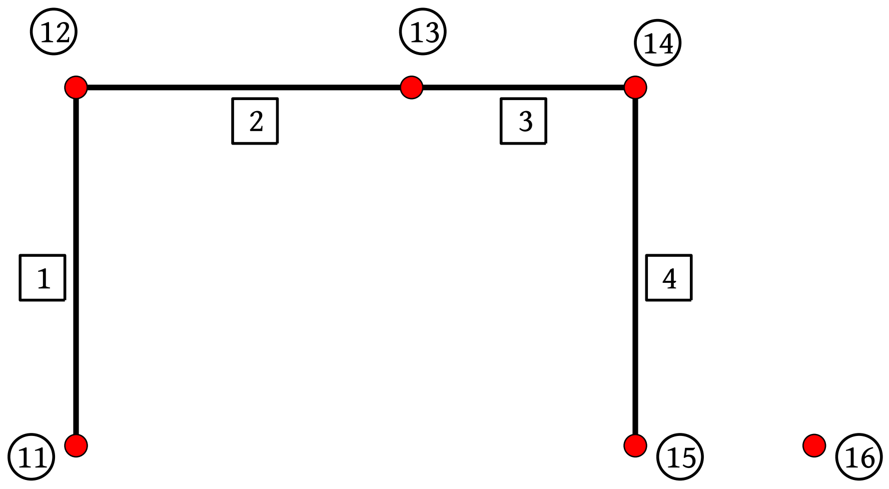

Figure 4: Finite element model of portal frame

```
(c) 2009-2012 Christian Bucher, CMSD-VUT
--]]

-- Callback function for NumberInputBoxes
function newLoads()
    local fh = H:GetValue()
    local fv = V:GetValue()
    solve(fh*100, fv*100)
end

-- Function to solve for a specific load vector
-- and show results
function solve (Fh, Fv)
    -- Construct a load vector
  local F1=structure:GetAllDisplacements()
  F = 100
-- DOF 0 of second node
  node = 1; dof = 0
  F1[{node,dof}] = Fh
-- DOF 1 of third node
  node = 2; dof = 1
  F1[{node,dof}] = -Fv
-- Convert to a vector containing only active DOF's
  FA=structure:ToDofDisplacements(F1)

-- Solve for displacements and assign to structure
  U=K:Solve(FA)
  U1=structure:ToAllDisplacements(U)
  print("U1", U1)
  structure:SetAllDisplacements(U1)

-- Draw the structure in first screen
    g:Clear()
    tris = structure:Draw()
    g:Triangles(tris)
    g:Autoscale()
    g:Zoom(0.7)
-- Draw the bending moments
```

```
  trim = structure:DrawSectionForces(5, 1, .01)
  g:Triangles(trim)
    g:Render()

-- Draw the structure in second screen
    g2:Clear()
    g2:Triangles(tris)
    g2:Autoscale()
    g2:Zoom(0.7)
  triq = structure:DrawSectionForces(1, 1, .01)
  g2:Triangles(triq)
    g2:Render()
end

-- Main program starts here
-- Define GUI
    H = gui.NumberInputBox("H: ", -3, 3, 1, "newLoads")
    V = gui.NumberInputBox("V: ", -3, 3, 2, "newLoads")
    g = graph.Graph3D("Bending Moment")
    g2 = graph.Graph3D("Shear Force")


-- Create and new FE structure
  structure=fem.Structure("frame")

-- Define node IDs and coordinates
  nodes = tmath.Matrix({
    {11, 0, 0, 0},
    {12, 0, 4, 0},
    {13, 4, 4, 0},
    {14, 8, 4, 0},
    {15, 8, 0, 0},
    {16, 4, 0, 0}
  })
  structure:AddNodes(nodes)

-- Define support conditions and fix reference node 16
  structure:GetNode(11):SetAvailDof(tmath.Matrix({{0, 0, 0, 0, 0, 0}}))
  structure:GetNode(15):SetAvailDof(tmath.Matrix({{0, 0, 0, 0, 0, 1}}))
  structure:GetNode(16):SetAvailDof(tmath.Matrix({{0, 0, 0, 0, 0, 0}}))
-- Define cross sections
  b1 = 0.05
  b3 = 0.08
  grey = tmath.Matrix({{150, 150, 150, 255}})
  s1 = structure:AddSection(1, "RECT"); s1:SetData(tmath.Matrix({{b1, b1}}))
  s1:SetColor(grey)
  s2 = structure:AddSection(2, "RECT"); s2:SetData(tmath.Matrix({{b3, b3}}))
  s2:SetColor(grey)

-- Define material
  m=structure:AddMaterial(8, "LINEAR_ELASTIC")
  m:SetData(tmath.Matrix({{2.1e11, .3, 7850}}))

-- Define elements
  structure:AddElements("RECT", 8, 1,
      tmath.Matrix({  {1, 11, 12, 16},
                      {2, 12, 13, 16}}))

  structure:AddElements("RECT", 8, 2,
      tmath.Matrix({  {3, 13, 14, 16},
                      {4, 14, 15, 16}}))

-- Find global DOFs and assemble stiffness
  nd=structure:GlobalDof()
  structure:Print()
  K=structure:SparseStiffness()

-- Show initial load case
    newLoads()
```

The bending moment and shear force can be seen in Fig. 5.

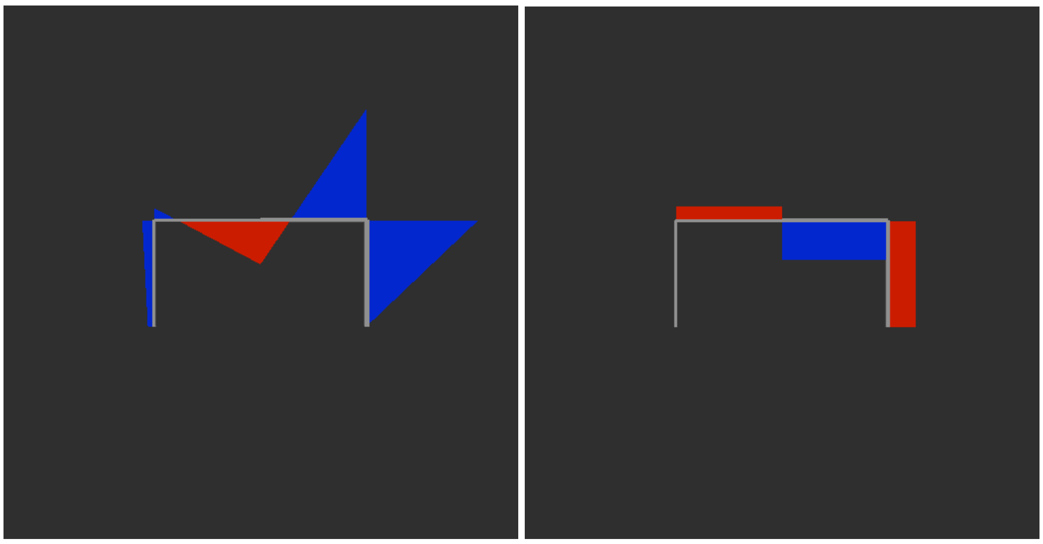
Figure 5: Bending moment and shear force of portal frame

1.6 frame_conmin.tng

This example performs a structural design optimization. The structure under consideration is the portal frame as shown in Fig. 3 which is subjected to a horizontal load H and a vertical load V. The goal of the optimization is to minimize the total mass which is equivalent to reducing the total volume of the structural elements

$$V = 4(b_1^2 + b_2^2 + b_4^2) + 2b_3^2 \to \text{Min.!} \tag{1}$$

This minimization is subject to two constraints on the displacements, i.e. $|u| \leq 0.05$ m and $|v| \leq 0.05$ m. The optimization algorithm being used is CONMIN which is based on the method of feasible directions. The slangTNG-script carrying out the optimization is

```
--[[
slangTNG
Test example demonstrating structural optimization with constraints
(c) 2009 - 2012 Christian Bucher, Vienna University of Technology
--]]

-- This function establishes the FE model depending on
-- the parameter set x as chosen by the optimization algorithm
```

```
function frame(x)
  struct=fem.Structure("frame")
-- Create nodes
  nodes = tmath.Matrix({
    {11, 0, 0, 0},
    {12, 0, 4, 0},
    {13, 4, 4, 0},
    {14, 6, 4, 0},
    {15, 6, 0, 0},
    {16, 10, 0, 0}
  })
  struct:AddNodes(nodes)

-- Define support conditions and fix reference node 16
  struct:GetNode(11):SetAvailDof(tmath.Matrix({{0, 0, 0, 0, 0, 0}}))
  struct:GetNode(15):SetAvailDof(tmath.Matrix({{0, 0, 0, 0, 0, 1}}))
  struct:GetNode(16):SetAvailDof(tmath.Matrix({{0, 0, 0, 0, 0, 0}}))

  colors = tmath.Matrix({
      {255, 0, 0, 255},
      {255, 255, 0, 255},
      {0, 255, 255, 255},
      {0, 0, 255, 255}
      })
--define 4 cross sections
  for i=0,3 do
      ss = struct:AddSection(i+1, "RECT", 0)
      ss:SetColor(colors:GetRows(i))
      ss:SetData(tmath.Matrix({{x[i], x[i]}}))
  end
-- define material
  local mm = struct:AddMaterial(8, "LINEAR_ELASTIC")
  mm:SetData(tmath.Matrix({{2.1e11, .3, 7850}}))
--define elements
  struct:AddElements("RECT", 8, 1, tmath.Matrix({{1, 11, 12, 16}}))
  struct:AddElements("RECT", 8, 2, tmath.Matrix({{2, 12, 13, 16}}))
  struct:AddElements("RECT", 8, 3, tmath.Matrix({{3, 13, 14, 16}}))
  struct:AddElements("RECT", 8, 4, tmath.Matrix({{4, 14, 15, 16}}))

-- assemble stiffness matrix and load vector
  local nd=struct:GlobalDof()
  local K=struct:SparseStiffness()
  local F1=struct:GetAllDisplacements()
  F1[{1,0}] = 1e5
  F1[{2,1}] = -1e5
  local F=struct:ToDofDisplacements(F1)
-- solve for displacements
  U=K:Solve(F)
  U1=struct:ToAllDisplacements(U)
  u=U1[{1,0}]
  w=U1[{2,1}]
    struct:SetDofDisplacements(U)
-- plot deformed structure
    tri = struct:Draw(5)
    ww:Clear()
    ww:Triangles(tri)
    if(first) then ww:Autoscale() end
    ww:Render()
    first = false
```

```lua
-- return displacements for constraints
  return u,w
end

-- objective function
function v(x)
  local cc=4*x[0]^2+4*x[1]^2+2*x[2]^2+4*x[3]^2
  return cc
end

-- constraints function (scaling the values affects convergence!)
function c(x)
  local u, w = frame(x)
  local cc=tmath.Matrix({{math.abs(u)-ULIM},{math.abs(w)-ULIM}})
  return cc*SC_CONSTR
  end

-- Main program starts here
ww = graph.Graph3D("Structure")
first = true
n=4
SC_CONSTR=100
ULIM=.05

-- construct optimizer and assign bounds and starting values
ops=optimize.Conmin(n,2)
bounds = tmath.Matrix(n,2)
bounds:SetLinearCols(1e-2, 10)
ops:SetBounds(bounds)
start = tmath.Matrix(n)
start:SetConstant(.3)
ops:SetDesign(start)

-- run optimizer until converging
done = false
iter = 0
while (not done) do
    iter = iter+1
  done=(ops:Compute()==0)
  x = ops:GetDesign()
  obj = v(x)
  ops:SetObjective(obj)
  constraints = c(x)
  ops:SetConstraints(constraints)
  if (oo == nil) then oo = tmath.Matrix({{obj}}) else oo=oo:AppendRows(
      tmath.Matrix({{obj}})) end
  if (cc == nil) then cc = tmath.Matrix(constraints:Transpose()) else cc=cc:
      AppendRows(constraints:Transpose()) end
  end

-- get optimal design
x = ops:GetDesign()
print("x",x)
u,w=frame(x)
print("obj", v(x)*7850,"u",u,"w",w)
print("with", iter, "iterations")

-- plot optimization history
oplot=graph.Graph("Convergence of objective")
```

```
oplot:AxisLabels("Iteration number", "Objective function")
counter = tmath.Matrix(oo)
counter:SetLinearRows(1,counter:Rows())
oplot:Plot(counter, oo, 1, "Objective")
oplot:PDF("Objective.pdf")

cplot=graph.Graph("Convergence of constraints", "Artsy")
cplot:AxisLabels("Iteration number", "Constraints")
cplot:Plot(counter, cc:GetCols(0), 1, "Constraint u")
cplot:Plot(counter, cc:GetCols(1), 1, "Constraint w")
cplot:PDF("Constraints.pdf")
```

The optimization history is shown in Figs. 6

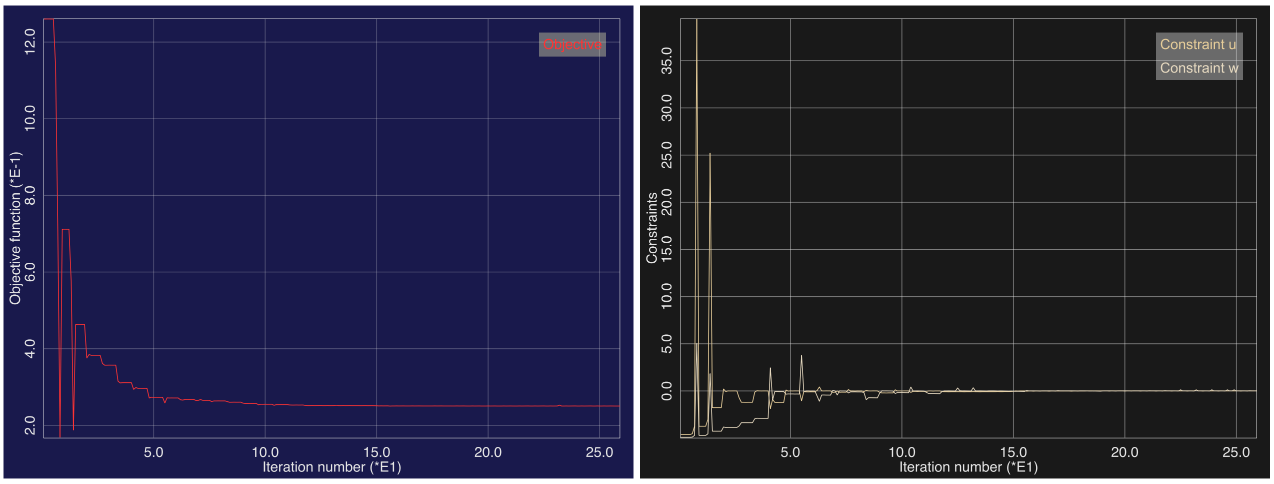

Figure 6: Optimization history, left: Objective, right: Constraints

1.7 statistics.tng

This example demonstrates the use of random variables for simulation and the statistics functions for estimation as available in slangTNG.

```
--[[
slangTNG
Simple test example demonstrating random variables, simulation and statistics
(c) 2009 - 2012 Christian Bucher, Vienna University of Technology
--]]

-- create several random variables with different types
rv1=stoch.RanvarNormal("First")
s = tmath.Matrix({{10},{4}})
rv1:SetStats(s)
rv1:Print()
print("rv1", rv1)

rv2=stoch.RanvarLognormal("Second")
s = tmath.Matrix({{6, 3}})
rv2:SetStats(s)
rv2:Print()

```

```
rv3=stoch.RanvarTriangular("Carl Friedrich")
rv3:SetParams(tmath.Matrix({{5, 7, 10}}))
rv3:Print()

rv4=stoch.RanvarUniform("Number 4")
rv4:SetParams(tmath.Matrix({{-1, 1}}))
rv4:Print()

rv5=stoch.RanvarGumbel("105")
rv5:SetStats(tmath.Matrix({{17, 4}}))
rv5:Print()

rv6=stoch.RanvarExponential("Expos□e")
rv6:SetStats(tmath.Matrix({{17, 4}}))
rv6:Print()

rv7=stoch.RanvarChisquare("testing")
rv7:SetStats(tmath.Matrix({{4}}))
rv7:Print()

rv8=stoch.RanvarSkewnormal("Skewer")
rv8:SetStats(tmath.Matrix({{1, 1, -.4}}))
rv8:Print()

-- Compute the CDF of random variable rv3 for different values of x
NPT=20
x=tmath.Matrix(NPT)
x:SetLinearRows(4, 11)

x3=rv3:CDF(x)
x[NPT]=x3
print("x", x)

-- Create a vector of correlated random variables
-- This uses a Gaussian copula ("Nataf model")
rho = 0.8
corr=tmath.Matrix({
    {1, rho, rho},
    {rho, 1, rho},
    {rho, rho, 1}
    })

rvec=stoch.Ranvec("My Collection")
rvec:AddRanvar(rv1)
rvec:AddRanvar(rv2)
rvec:AddRanvar(rv3)
rvec:SetCorrelation(corr)

rvec:Print()
print("rvec", rvec)


-- Create a vector of uncorrelated random variables
rvec2=stoch.Ranvec()
rvec2:AddRanvar(rv4)
rvec2:AddRanvar(rv5)

rvec2:Print()

```

```

-- Simulate samples of vector rvec using a randomized Sobol sequence
m=rvec:Simulate(100, stoch.Sobol)
-- m=rvec:Simulate(100) -- This would be crude Monte Carlo

-- Check statistics of simulated sample
mean=stoch.Mean(m)
print("mean", mean)
sigma=stoch.Sigma(m)
print("sigma", sigma)

skewness = stoch.Skewness(m)
print("skewness", skewness)

kurtosis=stoch.Kurtosis(m)
print("kurtosis", kurtosis)

covariance=stoch.Covariance(m)
print("covariance", covariance)

correlation=stoch.Correlation(m)
print("correlation", correlation)

kendall=stoch.Kendall(m)
print("kendall", kendall)

spearman=stoch.Spearman(m)
print("spearman", spearman)

-- Just generate 100 samples of 3 standardized uncorrelated Gaussian variables w/
    o creating random variables
u=stoch.Simulate(3,100, stoch.Sobol)
corr = stoch.Correlation(u)
print("corr", corr)
```

1.8 loscordelis.tng

This example demonstrates static structural analysis in slangTNG using finite shell elements (triangular elements with 6 DOFs per node). The structure under consideration is the well known Lo-Scordelis roof which is part of a cylindrical shell. The finite element mesh for this shell model has been generated using the public domain software Gmsh. It is shown in Fig. 7. The mesh is imported

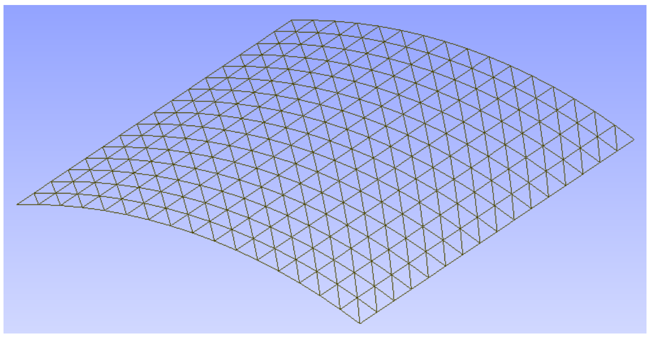
Figure 7: FE mesh for Lo-Scordelis roof

into slangTNG. Since Gmsh has been used for the definition of the nodes, elements, and physical groups only, it is still necessary to assign material properties and physical data (e.g. shell thickness) to the elements. The load case considered is gravity (dead load) with an acceleration of gravity of g = -9.81 m/s^2 pointing in the negative z-direction. The entire flow is shown in the following listing

```lua
--[[
SLangTNG
Test for Finite Element analysis
FE model imported from Gmsh
(c) 2009-2012 Christian Bucher, CMSD-VUT
--]]

-- import the model (Tetrahedra for volumes, triangles for surfaces) and set all
    DOF's to available

  struc=fem.StructureImportGmsh("loscordelis.msh")
  struc:SetAvailDof(tmath.Matrix({{1, 1, 1, 1, 1, 1}}))

-- Get the element group containing the support surface and convert to node group
  support=struc:GetGroup(1)
  nsup=support:ToNodeGroup(101)

-- remove all availabled DOF's for support
  struc:SetAvailDof(tmath.Matrix({{0, 1, 0, 1, 1, 1}}), nsup:GetMemberList())

-- Get the element group defining the roof (triangles)
  roof=struc:GetGroup(4)
  roofList = roof:GetMemberList()

-- Get the element group defining the symmetry line along the meridian
  longitude=struc:GetGroup(2)
  nlong=longitude:ToNodeGroup(102)
  longList = nlong:GetMemberList()
  struc:SetAvailDof(tmath.Matrix({{0, 1, 1, 1, 0, 0}}), longList)

-- Get the element group defining the symmetry line along the circle
  latitude=struc:GetGroup(3)
  nlat=latitude:ToNodeGroup(103)
  latList = nlat:GetMemberList()
  struc:SetAvailDof(tmath.Matrix({{1, 0, 1, 0, 1, 0}}), latList)

-- Get the corner point
  ncor=nlong:Intersection(nlat, 105)
  corList = ncor:GetMemberList()
  struc:SetAvailDof(tmath.Matrix({{0, 0, 1, 0, 0, 0}}), corList)

-- Define section and material properties (Gmsh provides only the mesh)

  t = 0.25
  s=struc:AddSection(300, "SHELL", 0)
  s:SetData(tmath.Matrix({{t}}))
  s:SetColor(tmath.Matrix({{255,0,0,255}}))
  m=struc:AddMaterial(800, "LINEAR_ELASTIC")
  m:SetData(tmath.Matrix({{4.32E8, .0, 360}}))
  struc:SetMaterial(800, roofList)
  struc:SetSection(300, roofList)

-- Assign global DOF numbers
  nd=struc:GlobalDof()
```

```

-- Assemble global stiffness matrix
  K=struc:SparseStiffness(roofList)

-- Assemble the global mass matrix
  M = struc:SparseMass(roofList)

-- Dead load
  gravity = struc:GetAllDisplacements()
  gravity:SetZero()
  g = gravity:GetCols(2)
  g:SetConstant(-9.81)     -- acceleration of gravity
  gravity:SetCols(g, 2)    -- z-axis
  G = struc:ToDofDisplacements(gravity)    -- convert this to a vector having a
  F = M:Dot(G)                             -- size equal to the number of DOFs

-- Solver for displacements
  U=K:Solve(F)

-- Show deformed structure with von Mises stress
  struc:SetDofDisplacements(U)
  U2 = struc:GetAllDisplacements()
  print("U2", U2)
  vec = struc:NodeVector(U2, 1)
  vis=graph.Graph3D("Lo Scordelis Roof")
  vis:Rotate(0, -100)
  stress = struc:ElementStress(0) -- 0 is von Mises stress
  tri = struc:ElementResults(stress, 1)
  vis:Triangles(tri)
  vis:Lines(vec)
  vis:Autoscale()
  vis:Zoom(1.4)
  vis:Render()
  vis:PNG("loscordelis.png")
  tmath.Output(vec, "vec.txt")
```

The deformed structure with the von Mises stress is shown in Fig. 8.

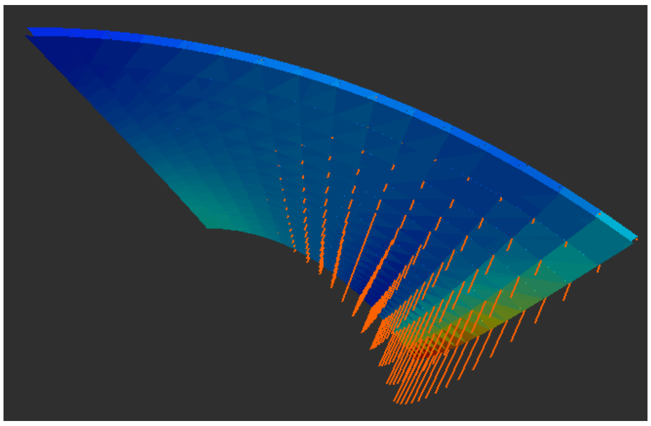
Figure 8: Deformations and von Mises stress for Lo-Scordelis roof

1.9 block.tng

This example demonstrates static structural analysis in slangTNG using finite volume elements (tetrahedral elements with 6 DOFs per node). The structure under consideration is a cuboid block with a circular hole. The block is fixed at one end. The finite element mesh for this shell model has been generated using the public domain software Gmsh. The mesh is imported into slangTNG. Since Gmsh has been used for the definition of the nodes, elements, and physical groups only, it is still necessary to assign material properties to the elements. The load case considered is a uniformly distributed shear loading at the free end. The entire flow is shown in the following listing

```
--[[
SLangTNG
Test for Finite Element analysis
FE structure imported from Gmsh
(c) 2009 Christian Bucher, CMSD-VUT
--]]


-- import the structure (Tetrahedra vor volumes, triangles for surfaces) and set
    all DOF's to available
  structure=fem.StructureImportGmsh("block.msh")
  structure:SetAvailDof(tmath.Matrix({{1, 1, 1, 1, 1, 1}}))

-- Get the element group containing the support surface and convert to node group
  support=structure:GetGroup(1)
  nsup=support:ToNodeGroup(101)

-- remove all availabled DOF's for support
  structure:SetAvailDof(tmath.ZeroMatrix(6,1), nsup:GetMemberList())

-- Get the element group carrying the distributed load (triangles)
  load=structure:GetGroup(2)
  loadList = load:GetMemberList()

-- Get the element group defining the body (tetrahedra)
  evol=structure:GetGroup(3)
  evolList = evol:GetMemberList()

-- Define section and material properties (Gmsh provides only the mesh)
  ss=structure:AddSection(301, "SHELL", 0)
  ss:SetData(tmath.Matrix({{0.01}}))
  ss:SetColor(tmath.Matrix({{200,200,200,255}}))
  structure:SetSection(301, loadList)
  structure:SetSection(301, support:GetMemberList())

  s=structure:AddSection(300, "VOLUME", 0)
  s:SetColor(tmath.Matrix({{255,0,0,255}}))
  m=structure:AddMaterial(800, "LINEAR_ELASTIC")
  m:SetData(tmath.Matrix({{1, .3, 1}}))
  structure:SetMaterial(800, evolList)
  structure:SetSection(300, evolList)

-- Assign global DOF numbers
  nd=structure:GlobalDof()
  print("nd", nd)

-- define distributed load in global y-direction
  force=tmath.Matrix({{0},{1},{0}})
```

```

-- Assemble global load vector
  F=structure:GlobalForce(force, loadList)

-- Assemble global stiffness matrix
  K=structure:SparseStiffness(evolList)

-- Solver for displacements
  U=K:Solve(F)

-- Show deformed structure (only volume elements are set visible)
  structure:SetDofDisplacements(U)

  vis=graph.Graph3D("Structure")
  vis:Rotate(20, -20)
  tri = structure:Draw(0.05)
  vis:Triangles(tri)

-- Add a vector plot showing the displacements
  U2 = structure:GetAllDisplacements()
  vec = structure:NodeVector(U2*0.02, .05)
  vis:Lines(vec)
  vis:Autoscale()
  vis:PNG("block_def.png")

--[[ Compute and visualize stresses
    0 v.Mises stress
    1 s_xx
    2 s_yy
    3 s_zz
    4 t_xy
    5 t_xz
    6 t_yz
--]]
  vis2 = graph.Graph3D("Stress")
  vis2:Rotate(20, -20)
  stress = structure:ElementStress(1) -- s_xx
  tri = structure:ElementResults(evolList, stress, 0.05)
  vis2:Triangles(tri)
  vis2:Autoscale()
  vis2:Render()
  vis:PNG("block_stress.png")
```

The deformed structure with the displacement vectors of all nodes is shown in Fig. 9. The deformed structure stress component σ_{xx} is also shown in Fig. 9.

1.10 two_dof_response.tng

This shows how to compute the solution of a system of ordinary differential equations (initial value problem) using the Runge-Kutta method. The system under consideration is a mechanical oscillator with two degrees of freedom. The equations of motion are given by

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} + \begin{bmatrix} 2c & -c \\ -c & c \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} + \begin{bmatrix} 2k & -k \\ -k & k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f(t) \\ f(t) \end{bmatrix} \tag{2}$$

Here m = 10 kg, c = 5 Ns/m, k = 500 N/m and $f(t)$ is a white noise with intensity D_0 = 1 N^{2}s (its auto covariance function is $D_0\delta(t)$ in which δ denotes the Dirac function). For the application of the

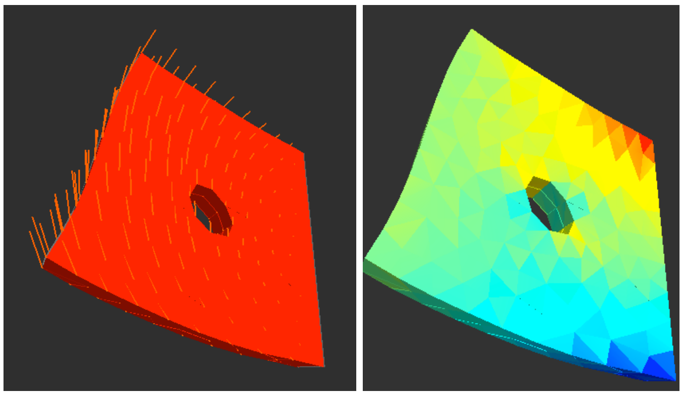

Figure 9: left: Deformations for block model, right: Stress σ_{xx} for block model

Runge-Kutta method, the equations need to be rewritten in first-order form by introducing the displacements and velocities as state variables. So we get a state vector $\mathbf{y}$ defined by

$$\mathbf{y} = \begin{bmatrix} x_1 \\ \dot{x}_1 \\ x_2 \\ \dot{x}_2 \end{bmatrix} \tag{3}$$

whose derivatives can be obtained from Eq. 2. The entire flow is shown in the following listing

```lua
--[[
Forced response of two-DOF system to white noise
Uses Runge-Kutta 4th order explicit time integration with substepping
(c) 2010 - 2012 Christian Bucher, CMSD, Vienna University of Technology
--]]

-- define derivative of state vector
-- This function uses the global variables MI, C, K
function derivative(t, y)
  local z=tmath.Matrix(4)
  z:SetZero()
  local index = math.floor(math.max(t/dt-.00001,0))
  local load = force[index]

  local x = tmath.Matrix({{y[0]},{y[2]}})
  local v = tmath.Matrix({{y[1]},{y[3]}})
  local f = tmath.Matrix({{1},{1}})

  local Fr = -K*x - C*v + f*load

  local a = MI*Fr

  z[0] = y[1]
  z[1] = a[0]
  z[2] = y[3]
  z[3] = a[1]
  return z
end

--[[
Main program:
```

```
Compute dynamic response of two-DOF system
--]]

-- Set system data
m = 10
k = 500
c = 5

-- Mass matrix
M = tmath.Matrix({
  {m, 0},
  {0, m}
  })
MI = tmath.Inverse(M)

-- Stiffness matrix
K = tmath.Matrix({
  {2*k, -k},
  {-k, k}
  })

-- Damping matrix
C = tmath.Matrix({
  {2*c, -c},
  {-c, c}
  })

N = 800
dt = 0.05
T = N*dt
-- Generate white noise with intensity D0
D0 = 1
force = stoch.Simulate(N, 1):Transpose()/math.sqrt(dt/D0)

-- Create differential equation object
DE = ode.RK4(4, "derivative")

-- Set initial conditions
start = tmath.Matrix(4)
start:SetZero()
DE:SetState(start)

-- Compute response with 2 substeps
resp = DE:Compute(0, T, N, 2)

-- Plot results
t = resp:GetRows(0)
t:SetLinearCols(0,T)

vis = graph.Graph("Responses", "Bright")
vis:AxisLabels("Time [s]", "Response [m]")

vis:Plot(t, resp:GetRows(0), 1, "Displacement 1")
vis:Plot(t, resp:GetRows(2), 1, "Displacement 2")
vis:PDF("two_dof_response.pdf")

inputOutput = t:AppendRows(force):AppendRows(resp:GetRows(0)):AppendRows(resp:
    GetRows(2))
tmath.Output(inputOutput:Transpose(), "two_dof_inout.txt")
```

The response of the system to one realization of the white noise in shown in Fig. 10. Note that this response may look different for each realization of the input.

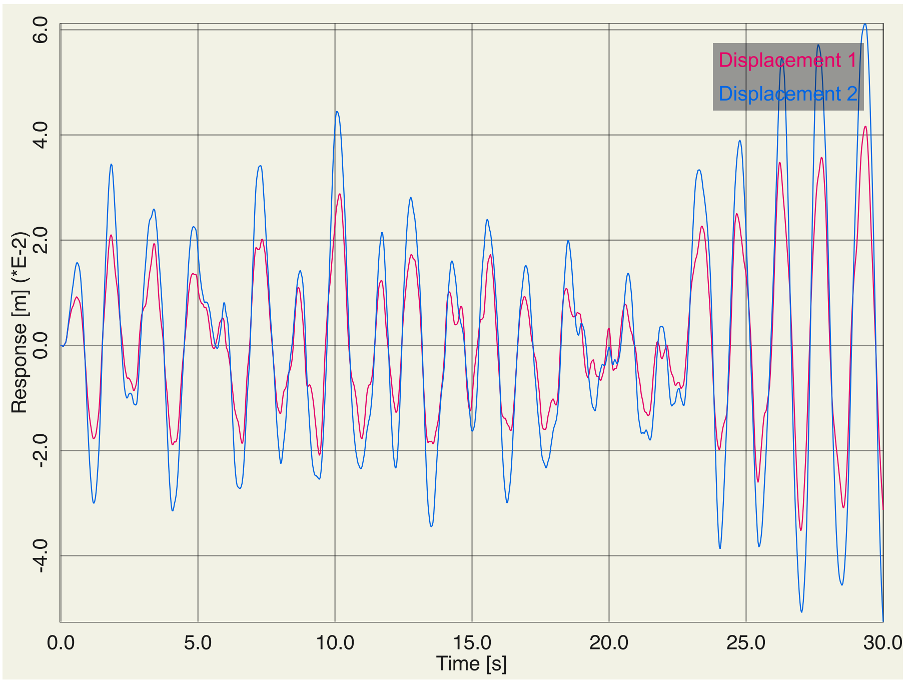

Figure 10: Displacement response for two-degree-of-freedom system

1.11 transfer.tng

This example shows how the frequency response function of a linear system (transfer function) can be computed from time series containing the excitation and the responses. The results as obtained in the previous section are used for the purpose of demonstration. Of course, measurements of a real system could be used just as well.

The flow is shown in the listing below.

```
--[[
Estimate complex transfer function from input/output measurements
(c) 2010 - 2012 Christian Bucher, CMSD, Vienna University of Technology
--]]

-- read input/output data
inputOutput = tmath.MatrixInput("two_dof_inout.txt")
t = inputOutput:GetCols(0)
dt = t[1] - t[0]
force = inputOutput:GetCols(1)
response = inputOutput:GetCols(2)

vis = graph.Graph("Input", "Mystic")
vis:AxisLabels("Time [s]", "Load [N]")
vis:Plot(t, force, 1, "Random Loading")
vis2 = graph.Graph("Output", "Mystic")
vis2:AxisLabels("Time [s]", "Response [m]")
```

```
vis2:Plot(t, response, 1, "Random Response")

FTload, domega = spectral.FFT(force, dt)
print("FTload", FTload)
print("domega", domega)
FTresponse, domega = spectral.FFT(response, dt)

transfer_r, transfer_i = tmath.ComplexDivide(FTresponse:GetCols(0), FTresponse:
    GetCols(1), FTload:GetCols(0), FTload:GetCols(1))

M = FTresponse:Rows()
omega = tmath.Matrix(M)
omega:SetLinearRows(0, (M-1)*domega)

vis3 = graph.Graph("Transfer", "Mystic")
vis3:AxisLabels("Circular frequency [rad/s]", "Transfer [m/N]")
vis3:Plot(omega, transfer_r, 1, "Real part")
vis3:Plot(omega, transfer_i, 1, "Imaginary part")
vis3:PDF("transfer.pdf")

magnitude, phase = tmath.ComplexToPolar(transfer_r, transfer_i)
vis4 = graph.Graph("Transfer2", "Mystic")
vis4:AxisLabels("Circular frequency [rad/s]", "Transfer [mm/N]")
vis4:Plot(omega, magnitude*1000, 1, "Magnitude*1000")
vis4:Plot(omega, phase, 1, "Phase")
vis4:PDF("transfer2.pdf")

transout = omega:AppendCols(transfer_r):AppendCols(transfer_i)
tmath.Output(transout, "two_dof_transfer_function.txt")
```

The real and imaginary parts of the frequency response function are shown in Fig. 11. The fun-

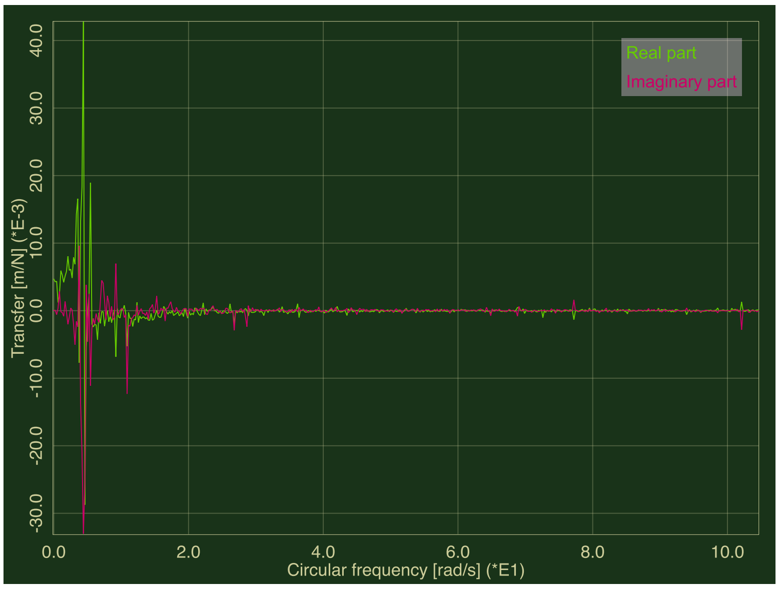

Figure 11: Real and imaginary parts of frequency response function obtained from time series of excitation and response

damental frequency is clearly visible in both real and imaginary parts. A different representation in terms of magnitude and phase is shown in Fig. 12.

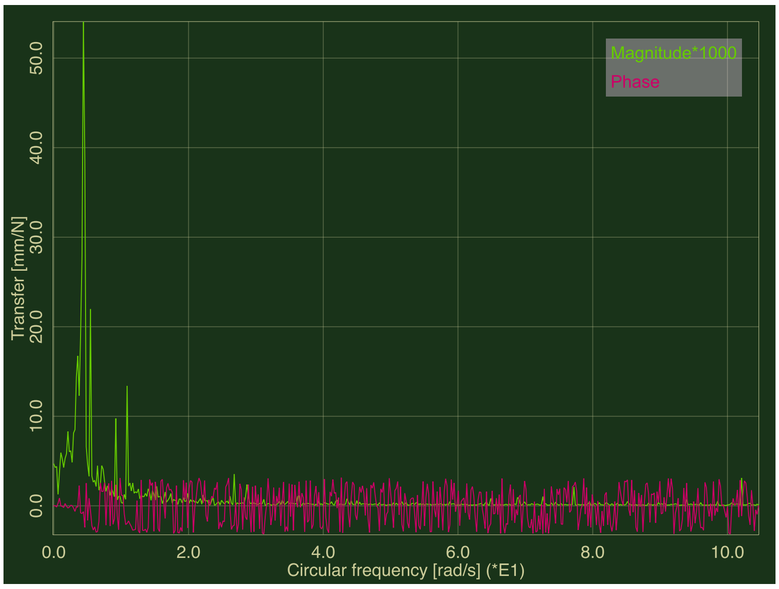

Figure 12: Magnitude and phase of frequency response function

1.12 modelfit.tng

This example shows how to use an optimization algorithm to fit a mechanical model to an experimentally obtained frequency response function. Here the results from the two previous examples are utilized. The model used describes a single-degree-of-freedom oscillator with the equation of motion

$$m\ddot{x} + c\dot{x} + kx = f(t) \tag{4}$$

The complex frequency response function $H(\omega)$ is given by

$$H(\omega) = \frac{1}{k - m\omega^2 + ic\omega} = \frac{k - m\omega^2 - ic\omega}{(k - m\omega^2)^2 + (c\omega)^2} \tag{5}$$

```lua
--[[
Fit a SDOF model to a numerically determined complex transfer function
from input/output measurements (which actually is a 2 DOF system)
(c) 2010 - 2012 Christian Bucher, CMSD, Vienna University of Technology
--]]

function model(m,c,k,om)
  local om2 = om:CW()*om
  local rr = om2*(-m)
  rr = rr:CW() + k
  local ii = om*(-c)
  local deno = rr:CW()*rr + ii:CW()*ii
  local r = rr:CW()/deno
  local i = ii:CW()/deno
  return r:AppendCols(i)
end

function objective(x)
  local m = x[0]
  local c = x[1]
  local k = x[2]
  local diff = model(m,c,k,omega)-realimag
  local diff2 = diff:GetRows(0,200)
  return tmath.Norm(diff2)^2
end

-- read numerical transfer function
transfer = tmath.MatrixInput("two_dof_transfer_function.txt")
omega = transfer:GetCols(0)
realimag = transfer:GetCols(1,2)

-- set up optimization problem (three variables, no constraints)
ops = optimize.Conmin(3, 0)

-- define bounds (parameters must be positive)
bounds = tmath.Matrix(3,2)
bounds:SetLinearCols(0,10000)
ops:SetBounds(bounds)

start = tmath.Matrix({{1},{0.1},{1}})
ops:SetDesign(start)

ans = 1
while(ans>0) do
  params = ops:GetDesign()
  obj = objective(params)
  ops:SetObjective(obj)
  ans = ops:Compute()
end

params = ops:GetDesign()
print("m = ", params[0], "c = ", params[1], "k = ", params[2])
print("omega_0 = ", math.sqrt(params[2]/params[0]))

approx = model(params[0], params[1], params[2], omega)

vis3 = graph.Graph("Real Part")
vis3:AxisLabels("Circular frequency [rad/s]", "Transfer [m/N]")
vis3:Plot(omega, realimag:GetCols(0), 1, "Measured")
vis3:Plot(omega, approx:GetCols(0), 1, "Fitted")
vis3:PDF("fitted_real.pdf")

vis4 = graph.Graph("Imaginary Part")
vis4:AxisLabels("Circular frequency [rad/s]", "Transfer [m/N]")
vis4:Plot(omega, realimag:GetCols(1), 1, "Measured")
vis4:Plot(omega, approx:GetCols(1), 1, "Fitted")
vis4:PDF("fitted_imag.pdf")
```

The real and imaginary parts of the frequency response function of the fitted model are compared to the experimental data in Figs. 13 and 14. The system parameters of the fitted model are m = 10.11 kg, c = 4.71 Ns/m and k = 203 N/m. The natural circular frequency of the this model is ω_0 = 4.48 rad/s.

1.13 duffing_radau5.tng

This example demonstrates the solution of a nonlinear differential equation using the RADAU5 solver. The problem under consideration is Duffing oscillator with harmonic excitation governed

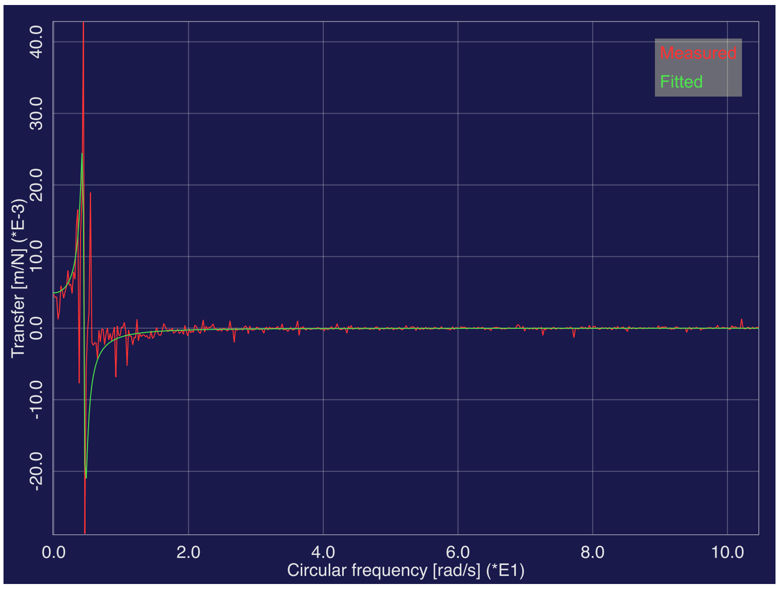

Figure 13: Real part of fitted and measured frequency response function

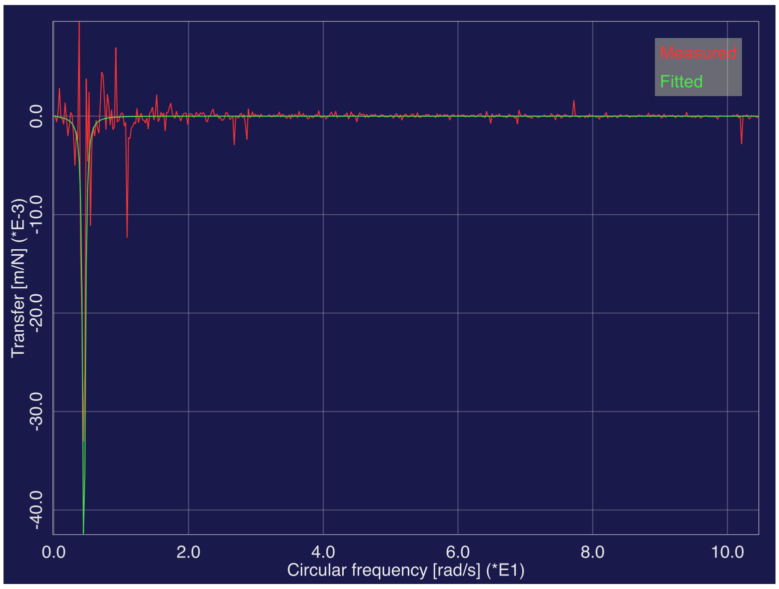

Figure 14: Imaginary part of fitted and measured frequency response function

by the equation of motion

$$m\ddot{x} + c\dot{x} + k\left(x + \epsilon x^3\right) = \sin t \tag{6}$$

Introducing state space notation this can be written in first-order form as

$$\begin{bmatrix}\dot{y}_1\\ \dot{y}_2\end{bmatrix} = \begin{bmatrix} y_1 \\ \frac{1}{m}\left(-ky_1 - k\epsilon y_1^3 - cy_2 + \sin t\right)\end{bmatrix} \tag{7}$$

The Jacobian matrix of this system is given by

$$\mathbf{J} = \frac{\partial(\dot{y}_1, \dot{y}_2)}{\partial(y_1, y_2)} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m}\left(1 + 3\epsilon y_1^2\right) & -\frac{c}{m}\end{bmatrix} \tag{8}$$

The computational flow is shown in the following listing:

```lua
--[[
slangTNG
Simple test example for the solution of initial value problems
This example describes a Duffing oscillator with cubic nonlinearity
subjected to harmonic load (resonance with the linearized system)
The example uses the analytical Jacobian matrix
(c) 2010 - 2012 Christian Bucher, Vienna University of Technology
--]]

-- This function defines the derivatives of the state variables
-- It is called by the ODE solver Radau5
  function derivative(t, y)
    local yd=tmath.Matrix(2)
    yd[0] = y[1]
    yd[1] = 1/m*(-k*y[0]*(1+epsilon*y[0]^2) - c*y[1] + math.sin(t))
    return yd
    end

-- This function defines the jacobian of the derivative wrt the state variables
-- It is called by the ODE solver Radau5
  function jacobian(t, y)
    local j = tmath.Matrix({
      {0, 1},
      {-k/m*(1+3*epsilon*y[0]^2), -c/m}
      })
    return j
    end

-- Main program
  T = 20*math.pi
  dt = 0.005
  N = T/dt
  k = 1
  m = 1
  c = 0.1
  epsilon = 0.1
-- Initialize a data object for the ODE solver
-- (implicit Runge Kutta code RADAU5 by E. Hairer und G. Wanner)
-- comment the following line and uncomment the next one to solve the problem
    without Jacobian
  system=ode.Radau5(2, "derivative", "jacobian")
--  system=ode.Radau5(2, "derivative")

-- Define the initial conditions
  start=tmath.Matrix(2)
  start[0] = 1
  start[1] = 0
  system:SetState(start)

-- Compute the solution
  t=tmath.Matrix(1,N)
  t:SetLinearCols(0,dt*N)
  result = system:Compute(0,dt*N, N)

-- Plot the result
  vis=graph.Graph("Duffing solution", "Bright")
  vis:AxisLabels("Time [sec]", "State variables [-]")
  vis:Plot(t, result:GetRows(0), 2, "Displacement")
  vis:Plot(t, result:GetRows(1), 2, "Velocity")
  vis:PDF("duffing_radau5.pdf")
```

The displacement and velocity response of the Duffing oscillator is shown in Fig. 15.

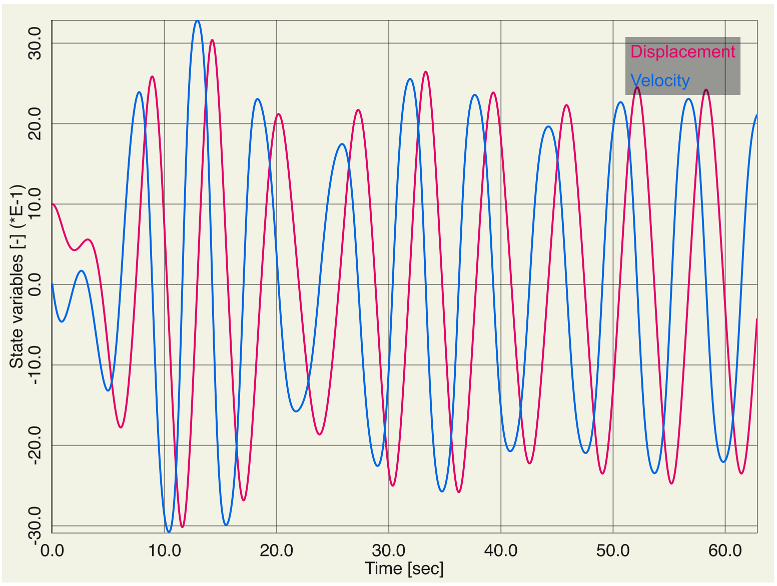

Figure 15: Response of Duffing oscillator to harmonic excitation

1.14 FPS_newmark.tng

This example shows the application of the Newmark time integration method to a problem of non-linear structural dynamics. The nonlinear effect in this example comes from a specific base isolation system, the so-called friction pendulum system (FPS). This systems involves a sliding device with an effective radius of curvature (this provides re-centering capabilities) and Coulomb friction (this provides energy dissipation). The model is subjected to a horizontal earthquake load defined in terms of a real accelerogram.

The flow of computation is shown in the following listing:

```
--[[
Test example to demonstrate implicit dynamics with friction pendulum system (FPS)
```

```
The pendulum is realized using the element type "CONTACT"
(c) 2011 - 2012 Christian Bucher, Vienna University of Technology
--]]

g = 9.81
Radius = 3
mu = .04
k0 = 1/Radius   -- this is multiplied by the current value of the contact force
    inside the
                -- contact element
k1 = 1e6

dt = 0.02
T = 20
N = T/dt

structure=fem.Structure("frame")

-- Create Nodes and set DOFs

structure:AddNodes(tmath.Matrix({
    {11, 0, 0, 0},
    {12, 0, 4, 0},
    {13, 4, 4, 0},
    {14, 6, 4, 0},
    {15, 6, 0, 0},
    {16, 10, 0, 0},
    {9, 0, 1, 0},
    {10, 0, -1, 0},
    {8, 7, 0, 0}
    }))

fixed = tmath.ZeroMatrix(6,1)
horizontal_slider = tmath.Matrix({{1,0,0,0,0,1}})
structure:GetNode(8):SetAvailDof(fixed)
structure:GetNode(9):SetAvailDof(fixed)
structure:GetNode(10):SetAvailDof(fixed)
structure:GetNode(11):SetAvailDof(tmath.Matrix({{1,1,0,0,0,1}}))
structure:GetNode(15):SetAvailDof(horizontal_slider)
structure:GetNode(16):SetAvailDof(fixed)

-- Section and Material
s=structure:AddSection(1, "RECT", 0)
s:SetData(tmath.Matrix({{.1, .1}})) -- this is for real beams
s:SetColor(tmath.Matrix({{255,255,0,255}}))

s=structure:AddSection(2, "RECT", 0)
s:SetData(tmath.Matrix({{.03, .3}})) -- just to visualize the FPS
s:SetColor(tmath.Matrix({{255,0,0,255}}))

mat8 = structure:AddMaterial(8, "LINEAR_ELASTIC")
mat8:SetData(tmath.Matrix({{2.1e11, .3, 7850}}))
mat9 = structure:AddMaterial(9, "CONTACT")
mat9:SetData(tmath.Matrix({{k0, k1, mu}}))

-- Elements
structure:AddElements("RECT", 8, 1, tmath.Matrix({
    {31, 11, 12, 16},
    {32, 12, 13, 16},
    {33, 13, 14, 16},
    {34, 14, 15, 16},
    {35, 11, 15, 9}
    }))

structure:AddElements("CONTACT", 9, 2, tmath.Matrix({{100, 10, 11, 8}}))


-- Determine global DOF numbering
nd=structure:GlobalDof()

-- Assemble global matrices
MM=structure:SparseMass();
KK=structure:SparseStiffness();

-- Compute eigenvalues to Rayleigh damping
eval, evec = KK:Eigen(MM, 5)
freq = eval:CW():Sqrt()/2/math.pi
print("freq", freq)

-- Set Rayleigh damping
zeta1 = 0.02
zeta2 = 0.02
omega1 = freq[0]*2*math.pi
omega2 = freq[1]*2*math.pi

alpha = 2*(zeta1*omega1 - zeta2*omega2)/(omega1^2-omega2^2)
beta = 2*omega1*zeta1 - alpha*omega1/2
CC = KK:Add(MM, alpha, beta)

a0 = 4/dt^2
a1 = 2/dt
a2 = 4/dt
a3 = 1
a4 = 1
a5 = 0
a6 = dt/2
a7 = dt/2

-- Effective "stiffness" for Newmark method
Keff = KK:Add(MM, a0):Add(CC,a1)

-- Define a load case (unit acceleration in y-direction)
F1=structure:GetAllDisplacements()
col = F1:GetCols(0)
F1:SetZero()
col:SetConstant(1)

F1:SetCols(col, 1)

-- Bring load case into vector form and multiply unit acceleraton with mass
   matrix
Dead1=structure:ToDofDisplacements(F1)
Dead = MM:Dot(Dead1)*(-g)

-- Define a load case (unit acceleration in x-direction)
F1=structure:GetAllDisplacements()
col = F1:GetCols(0)
F1:SetZero()
```

```
col:SetConstant(1)

F1:SetCols(col, 0)

-- Bring load case into vector form and multiply unit acceleraton with mass
    matrix
F2=structure:ToDofDisplacements(F1)
F = MM:Dot(F2)


-- solve for static displacement under dead load for initial values
U = KK:Solve(Dead)
V = tmath.Matrix(U)
V:SetZero()
A = tmath.Matrix(V)


-- Read ground acceleration data
Bam = tmath.MatrixInput("Bam.txt");
dt_a = Bam[1] - Bam[0]
quake=Bam:GetCols(1)

-- Set time step for output
NT = T/dt_a
t = tmath.Matrix(NT)
t:SetZero()
d1 = tmath.Matrix(NT,2)
d1:SetZero()
old_index = -1


v=graph.Graph3D("Frame")
tri = structure:Draw()
v:Clear()
v:Triangles(tri)
v:Autoscale()
v:Render()


-- Newmark loop NOTE: this implements a simple Newton-Raphson iteration using the
     initial effective stiffness within one time step
t0 = control.Time()
for i=0,N-1 do
  ti = i*dt
    index = math.floor(math.max(ti/dt_a-.00001,0))
  accel = quake[index]

  R1 = Dead + F*(-accel) + MM:Dot(A + V*a2 + U*a0) + CC:Dot(V + U*a1)
  U1 = Keff:Solve(R1)
  for k=0,5 do
    structure:SetDofDisplacements(U1)
    R = R1 - structure:GlobalResForce() - MM:Dot(U1*a0) - CC:Dot(U1*a1)
    Rnorm = tmath.Norm(R)
    if (Rnorm < 1) then break end
    DU = Keff:Solve(R)
    U1 = U1+DU
  end
  V1 = U1*a1 - U*a1 - V
  A1 = V1*a1 - V*a1 - A
  U = tmath.Matrix(U1)
  V = tmath.Matrix(V1)
  A = tmath.Matrix(A1)
  structure:SetDofDisplacements(U)
  structure:GlobalUpdate()

    tri = structure:Draw(3)
    v:Clear()
    v:Triangles(tri)
    v:Render()

  if (not(old_index == index)) then
    old_index = index
    t[index] = index*dt
    D = structure:GetAllDisplacements()
    d1[index] = D[0]
    d1[index+NT] = D[1]-D[0]
  end -- if
end --for

v2=graph.Graph("Response", "Bright")
v2:AxisLabels("Time [s]", "Displacement [m]")
v2:Plot(t,d1:GetCols(0), 2, "Support displacement")
v2:Plot(t,d1:GetCols(1), 2, "Relative (top-support)")
v2:PDF("FPS_newmark.pdf")
print("Done")
```

The displacement response of the friction pendulum and the relative displacement between support and top of the structure is shown in Fig. 16.

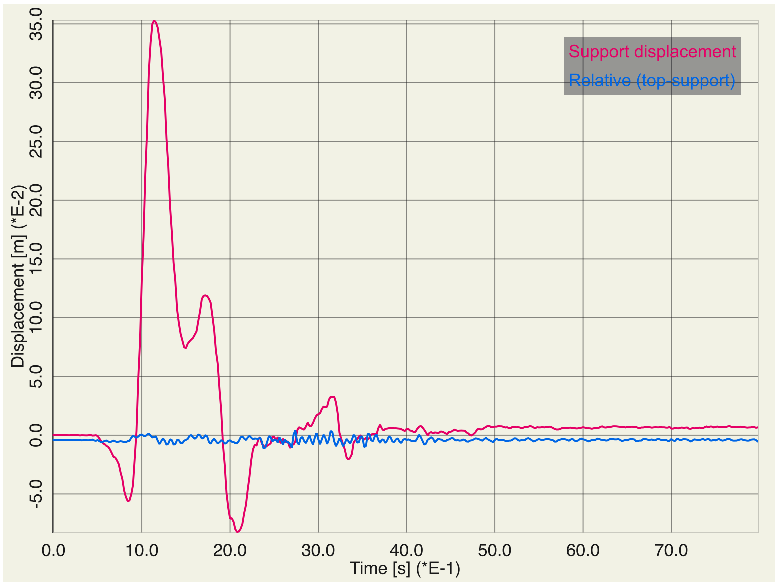

Figure 16: Response of structure with base isolation (FPS) to earthquake excitation

1.15 double_pendulum.tng

This example demonstrates the use of a simple symplectic integrator to solve the equations of motion for a Hamiltonian system such that total energy remains bounded for arbitrarily long time. The kinematic relations for the double pendulum can readily be seen in Fig. 17.

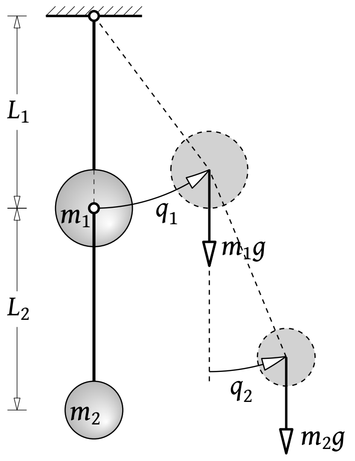

Figure 17: Double pendulum

$$\begin{aligned} x_1 &= L \sin q_1;\; z_1 = L_1(1-\cos q_1) \\ x_2 &= x_1 + L_2 \sin q_2;\; z_2 = z_1 + L_2(1-\cos q_2) \end{aligned} \tag{9}$$

From this we immediately get the potential energy due to gravity

$$V = m_1 z_1 + m_2 z_2 = (m_1+m_2)L_1(1-\cos q_1) + m_2 L_2(1-\cos q_2) \tag{10}$$

The velocities are obtained by differentiation

$$\begin{aligned} \dot{x}_1 &= L_1\dot{q}_1 \sin q_1 \\ \dot{z}_1 &= L_1\dot{q}_1 \cos q_1 \\ \dot{x}_2 &= \dot{x}_1 + L_2\dot{q}_2 \sin q_2 \\ \dot{z}_2 &= \dot{z}_1 + L_2\dot{q}_2 \cos q_2 \end{aligned} \tag{11}$$

The kinetic energy is then given by

$$\begin{aligned} T &= \frac{m_1}{2}(\dot{x}_1^2+\dot{z}_1^2) + \frac{m_2}{2}(\dot{x}_2^2+\dot{z}_2^2) \\ &= \frac{m_1}{2}(L_1^2\dot{q}_1^2\sin^2 q_1 + L_1^2\dot{q}_1^2\cos^2 q_1)+ \\ &+ \frac{m_2}{2}(L_1^2\dot{q}_1^2\sin^2 q_1 + 2L_1L_2\dot{q}_1\dot{q}_2 \sin q_1 \sin q_1 + L_2^2\dot{q}_2^2\sin^2 q_2 + \\ &+ L_1^2\dot{q}_1^2\cos^2 q_1 + 2L_1L_2\dot{q}_1\dot{q}_2\cos q_1 \cos q_1 + L_2^2\dot{q}_2^2\cos^2 q_2) = \\ &= \frac{m_1}{2}L_1^2\dot{q}_1^2 + \frac{m_2}{2}(L_1^2\dot{q}_1^2 + 2L_1L_2\dot{q}_1\dot{q}_2\cos(q_1-q_2) + L_2^2\dot{q}_2^2) \end{aligned} \tag{12}$$

We further use the canonical momenta

$$\begin{aligned} p_1 &= \frac{\partial T}{\partial \dot{q}_1} = m_1L_1^2\dot{q}_1 + m_2(L_1^2\dot{q}_1 + L_1L_2\dot{q}_2\cos(q_1-q_2)) \\ p_2 &= \frac{\partial T}{\partial \dot{q}_2} = m_2(L_1L_2\dot{q}_1\cos(q_1-q_2) + L_2^2\dot{q}_2) \end{aligned} \tag{13}$$

From these relations, we obtain the velocities in terms of the momenta as

$$\begin{aligned} \dot{q}_1 &= -\frac{l_1\,p_2\,\cos\left(q_2-q_1\right)-l_2\,p_1}{{l_1}^2\,l_2\,\left(m_2\,\sin^2\left(q_2-q_1\right)+m_1\right)} \\ \dot{q}_2 &= -\frac{l_2\,m_2\,p_1\,\cos\left(q_2-q_1\right)-l_1\,m_2\,p_2-l_1\,m_1\,p_2}{l_1\,{l_2}^2\,m_2\,\left(m_2\,\sin^2\left(q_2-q_1\right)+m_1\right)} \end{aligned} \tag{14}$$

and upon inserting into the kinetic energy we have

$$T = -\frac{2\,l_1\,l_2\,m_2\,p_1\,p_2\,\cos\left(q_2-q_1\right)-{l_1}^2\,m_2\,{p_2}^2-{l_1}^2\,m_1\,{p_2}^2-{l_2}^2\,m_2\,{p_1}^2}{2\,{l_1}^2\,{l_2}^2\,m_2\,\left(m_2\,\sin^2\left(q_2-q_1\right)+m_1\right)} \tag{15}$$

Finally we obtain Hamilton's equations as

$$\begin{aligned} \dot{p}_1 &= C-(m_1L_1+m_2L_2)\sin q_1 \\ \dot{p}_2 &= -C-m_2L_2\sin q_2 \\ C &= -\frac{\sin\left(q_2-q_1\right)\,\left(p_1\,L_2-p_2\,\cos\left(q_2-q_1\right)\,L_1\right)\,\left(m_2\,p_1\,\cos\left(q_2-q_1\right)\,L_2-m_2\,p_2\,L_1-m_1\,p_2\,L_1\right)}{\left(m_2\,\sin^2\left(q_2-q_1\right)+m_1\right)^2\,{L_1}^2\,{L_2}^2} \end{aligned} \tag{16}$$

together with Eq. 14.

This set of equations is implemented in the auxiliary script file derivatives_pendulum.tng:

```
-- Derivatives for double pendulum in Hamiltonian system
-- This assumes two unit masses, unit lengths, and unit gravity

function derivatives_pendulum (q, p)
  local q1 = q[0]
  local q2 = q[1]
  local p1 = p[0]
  local p2 = p[1]
    local si = math.sin(q1-q2);
    local co = math.cos(q1-q2);
    local C1 = p1*p2*si/(1.+si*si);
    local C2 = (p1*p1+2*p2*p2-p1*p2*co)/2/(1+si*si)/(1+si*si)*math.sin(2*(q1-q2))
        ;
    local q_dot = tmath.Matrix(2)
    local p_dot = tmath.Matrix(2)

    q_dot[0] = (p1 - p2*co)/(1+si*si);
    q_dot[1] = (2*p2 - p1*co)/(1+si*si);
    p_dot[0] = -2.*math.sin(q1) - C1 + C2;
    p_dot[1] = -math.sin(q2) + C1 - C2;
  return q_dot, p_dot;
end
```

The actual solution procedure (symplectic Euler scheme) is implemented in the auxiliary script file euler_symplectic.tng:

```
-- Symplectic Euler integrator.

function euler_symplectic(q, p, h, function_derivative)
  local q_dot, p_dot = function_derivative(q, p)
    local old_p_dot = p_dot*1

    -- Increment of p's, this is implicit!
    local bool done = false;
    local tolerance = math.max(1e-9, 1e-7*tmath.Norm(p_dot))
    while(not done) do
    trial_p = p + p_dot*h;
    q_dot, p_dot = function_derivative(q, trial_p)
        if (tmath.Norm(old_p_dot-p_dot)<tolerance) then
            done = true; end
        old_p_dot = p_dot*1
  end
  p = trial_p;

    -- Increment of q's, this is explicit
    q = q + q_dot*h
  return q, p;
end
```

The main program including the set-up of interactive elements to change the initial conditions, the flow of computation and the visualization is contained in the main script file `double_pendulum.tng`

```
--[[
Demo example showing integration and visualization of double pendulum
(c) 2012, Vienna University of Technology, Christian Bucher
--]]

dofile("euler_symplectic.tng")
dofile("derivatives_pendulum.tng")

function define_gui()
    local world = graph.Graph3D("Double Pendulum")
    local L1 = 1.4*L
    world:SetRange(-L1, L1, -L1, L1, -L1, L1)
    phi1 = gui.NumberInputBox("phi_1", -math.pi, math.pi, 1, "set_phi")
    phi2 = gui.NumberInputBox("phi_2", -math.pi, math.pi, 2, "set_phi")
    return world
end

function set_phi()
    q[0] = phi1:GetValue()
    q[1] = phi2:GetValue()
    p[0] = 0
    p[1] = 0
    draw_pendulum(q, w)
    control.Pause("pause")
end

function draw_pendulum(phi, scene)
    scene:Clear()
    local x1 = L*math.sin(phi[0])
    local y1 = -L*math.cos(phi[0])
    scene:Sphere(0, 0, 0, .3, 100, 100, 100, 255, 1363) -- support
    scene:Sphere(x1, y1, 0, .6, 255, 0, 0, 255, 1363) -- mass 1
    local x2 = x1 + L*math.sin(phi[1])
    local y2 = y1 - L*math.cos(phi[1])
    scene:Sphere(x2, y2, 0, .6, 255, 255, 0, 255, 1364) -- mass 2
    scene:Cylinder(x1, y1, 0, x2, y2, 0, .15, 0, 0, 255, 255, 1390)
    scene:Cylinder(x1, y1, 0, 0, 0, 0, .15, 0, 0, 255, 255, 1390)
    scene:Render()
end

L = 3
w = define_gui()

h = 0.005
q = tmath.Matrix(2)
q[0] = phi1:GetValue()
q[1] = phi2:GetValue()

p = tmath.ZeroMatrix(2,1)

i = 0
n = 10

draw_pendulum(q, w)
control.Pause("pause")

while(true) do
    q, p = euler_symplectic(q, p, h, derivatives_pendulum);
    if (i<n) then
    i = i+1
  else
    draw_pendulum(q, w)
    i = 0
  end
end
```

2 Module descriptions

2.1 Module tmath

The module `math` is constructed around the object type `Matrix`. Various mathematical operations can be carried out with the help of this class. It also serves as a communication object between methods in the other modules such as `fem` od `stoch`.

A=tmath.Matrix(3,3)	Create an uninitialized matrix of size 3 x 3
a=tmath.Matrix(3)	Create an uninitialized matrix of size 3 x 1 (vector)
A=tmath.ZeroMatrix(3,3)	Create a zero-initialized matrix of size 3 x 3
B=tmath.Matrix(A)	Create matrix B as a copy of matrix A
B=tmath.Matrix({{a,b},{c,d}})	Create matrix B of size 2 x 2 from a list of lua tables
A:SetZero()	Set matrix A to all zeroes
A:SetConstant(c)	Set all elements of matrix A equal to c
A:SetLinearRows(c, d)	Set all columns of A such that the values increase from c in the first row to d in the last row
A:SetLinearCols(c,d)	Set all rows of A such that the values increase from c in the first column to d in the last column
tmath.Output(A, ”file.txt”)	Write contents of matrix A to a text file named ”file.txt”
tmath.Output(A, ”file.bin”)	Write contents of matrix A to a binary file named ”file.txt”
A = tmath.MatrixInput(”file.txt”)	Create matrix A from the contents of text file ”file.txt”. Number of rows and columns are determined by the structure of the text file.
A = tmath.MatrixInput(”file.bin”)	Create matrix A from the contents of a binary file previously created by tmath.Output()
B = A:GetCols(0)	Create matrix B containing the first column of matrix A
B = A:GetCols(3,4)	Create matrix B containing columns 4 through 7 of matrix A
C = A:GetRows(1)	Create matrix C containing the second column of matrix A
C = A:AppendCols(B)	Create matrix C by appending the columns of matrix B to the columns of matrix A. Matrices A and B must have the same number of rows.
C = A:AppendRows(B)	Create matrix C by appending the rows of matrix B to the rows of matrix A. Matrices A and B must have the same number of columns.
B = tmath.Sin(A)	Create a matrix B whose elements are $b_{jk} = \sin a_{jk}$
B = tmath.Sin(A, c)	Create a matrix B whose elements are $b_{jk} = c \sin a_{jk}$

B = tmath.Cos(A)	Create a matrix B whose elements are $b_{jk} = \cos a_{jk}$
B = tmath.Exp(A)	Create a matrix B whose elements are $b_{jk} = \exp a_{jk}$
B=A:CW()	Create a component wise object B from matrix A. Used for various component wise operations.
B = A:CW():Pow(4)	Create a matrix B whose elements are given by $b_{jk} = a_{jk}^4$
B = A:CW():Sqrt()	Create a matrix B whose elements are given by $b_{jk} = \sqrt{a_{jk}}$
C = A:CW()*B	Create a matrix B whose elements are given by $c_{jk} = a_{jk}b_{jk}$. Matrices A and B must have equal sizes
B = A:Transpose()	Creates matrix B as transpose of matrix A
B = tmath.Inverse(A)	Creates B as inverse of matrix A by carrying out an LU factorization. Matrix A must be square and non-singular
C=A*B	Creates matrix C as the usual matrix product of A and B. The number of columns of A must be equal to the number of rows of B
Br, Bi = tmath.ComplexInverse(Ar, Ai)	Computes the real part Br and imaginary part Bi of the inverse B to a complex matrix A with real part Ar and imaginary part Ai. Matrices Ar and Ai must be square and have the same size
Cr, Ci = tmath.ComplexProduct(Ar, Ai, Br, Bi)	Computes the real part Cr and imaginary part Ci of the complex matrix C which arises from the multiplication of two complex matrices A and B with real parts Ar and Br as well as imaginary parts Br and Bi, respectively. Matrices Ar and Br must have the same sizes, matrices Br and Ri mast have same sizes, and the number of columns of Ar must be equal to the number of rows of Br
M, Phi = tmath.ComplexToPolar(A, B)	Creates matrices M and Phi containing the elements $m_{jk} = \sqrt{a_{jk}^2 + b_{jk}^2}$ and $\phi_{jk} = \arctan \frac{b_{jk}}{a_{jk}}$
A, B = tmath.ComplexFromPolar(M, Phi)	Creates matrices A and B containing the elements $a_{jk} = m_{jk} \cos \phi_{jk}$ and $b_{jk} = m_{jk} \sin \phi_{jk}$
E, F = tmath.ComplexMultiply(A, B, C, D)	Create matrices A and B representing complex numbers $e_{jk} + if_{jk} = (a_{jk} + ib_{jk})(c_{jk} + id_{jk})$ (i is the imaginary unit)
E, F = tmath.ComplexDivide(A, B, C, D)	Create matrices A and B representing complex numbers $e_{jk} + if_{jk} = \frac{a_{jk}+ib_{jk}}{c_{jk}+id_{jk}}$

2.2 Module graph

This module provides 2D and 3D plotting capabilities.

g = graph.Graph(”Curves”)	Create a 2D graphics window with the title ”Curves” using the default color scheme
g = graph.Graph(”Plots”, ”Mystic”)	Creates a graphics window with the title ”Plots” using the predefined color scheme ”Mystic”. Other available color schemes are ”Bright” and ”Artsy”
g:AxisLabels(”x”, ”y”)	Sets the text for the label on the x-Axis to ”x” and on the y-Axis to ”y”
g:Plot(x, y, s, ”Label”)	Plots a curve using x for the values on the x-axis and y for the values on the y-axis. The line thickness is s if it is > 0. If s < 0, then the curve is draw only by dots of size s. ”Label” is a text put into the right upper corner of the graphics window identifying the curve. Multiple plots can be put into the same graphics window
g:PDF(”file.pdf”)	Write the contents of the graphics window into a vector PDF file. The size of the graphics in the file is fixed to 800x600 pixels regardless of the current size of the graphics window.
g:PNG(”file.png”)	Write the contents of the graphics window into a pixel PNG file. The size of the graphics in the file is fixed to 800x600 pixels.

g3 = graph.Graph3D(”Surfaces”)	Creates a 3D graphics window with the title ”Surfaces”
g3:Clear()	Removes any 3D content from the graphics window
g3:Triangles(A)	Add triangles defined in matrix A to the contents of the window g3
g3:Autoscale()	Scale the contents of the window g3 such that all triangles and lines fit
g3:Sphere(x, y, z, r, red, blue, green, alpha, tag)	Draw a sphere with radius r at position x, y, z. The sphere has a color defined by red, blue, green and alpha. It has an identifier number tag
g3:Cylinder(x1, y1, z1, x2, y2, z2, r, red, green, blue, alpha, tag)	Draw a cylinder with radius r from the point (x1, y1, z1) to the point (x2, y2, z2)
g3:Render()	Draws the current content of the graphics window
g3:SetRange(xmin, xmax, ymin, ymax, zmin, zmax)	Scales the contents of the graphics window according to the cuboid defined by the parameters xmin...zmax.

2.3 Module gui

This module is intended to provide user interaction with pre-define slangTNG-scripts.

button=gui.PushButton(”Label”, ”Target”)	Creates a PushButton with the text label ”Label”. When it is clicked or tapped, the slangTNG-function ”Target” is called. This function must be defined in the current script.
number=gui.NumberInputBox(”x”, -1, 10, 2, ”Target”)	Creates a NumberInputBox with the text label ”x”. The current value of the box is 2, the acceptable range for input by the user is between -1 and 10. Larger or smaller values are clamped to these bounds. Upon change of the current value by user interaction, the slangTNG-function ”Target” is called.
val = number:GetValue()	Returns the numerical value of the NumberInputBox number in the variable val
number:SetValue(val)	Sets the current value of the NumberInputBox number to val.
text=guiTextBox(”Label”)	Creates a TextBox with the label text ”Label”. Its content text is empty.
text:SetText(”Hello”)	Sets the contents of the TextBox text to ”Hello”
text:SetNumber(val)	Sets the contents of the TextBox text to a string representation of the number val.

2.4 Module stoch

This module provides functionality for digital simulation of random variables and for statistics.

rv = stoch.ranvarNormal(”Carl”)	Creates a normally distributed random variable
rv = stoch.ranvarLognormal(”Friedrich”)	Creates a log-normally distributed random variable
rv = stoch.ranvarUniform(”Uni”)	Creates a uniformly distributed random variable
rv = stoch.ranvarTriangular(”Trigger”)	Creates a triangularly distributed random variable
rv = stoch.ranvarGumbel(”E.J.”)	Creates a Gumbel (Extreme Type I largest) distributed random variable
rv = stoch.ranvarExponential(”Shifted”)	Creates a (shifted) exponentially distributed random variable
rv = stoch.ranvarChisquare(”Test me!”)	Creates a χ^2-distributed random variable
rv = stoch.ranvarSkewnormal(”Azzalini”)	Creates a skew-normal random variable

rv:SetStats(A)	Assigns the contents of the matrix A as statistical parameters to the random variable rv. The number of elements must match the number of statistics required by the random variable rv. For random variables of type RanvarNormal, RanvarLognormal, RanvarEponential, RanvarGumbel, and RanvarUniform this number is 2. For type RanvarChisquare this number is 1. For types RanvarTriangular and RanvarSkewnormal this number is 3. The meaning of these statistics is always the same, i.e. 1. mean, 2. standard deviation, 3. coefficient of skewness (4. coefficient of kurtosis)
rv:setParams(B)	Assigns the contents of the Matrix B as parameters to the random variable on rv. The meaning of these parameters differs between different types of random variables
A=rv:Simulate(N)	create a vector A containing N Monte Carlo samples of the random variable rv
A=rv:Simulate(N, stoch.Sobol)	create a vector A containing N samples of the random variable rv generated from randomized Sobol sequence

rvec=stoch.Ranvec(”Collection”)	Create a random vector
rvec:AddRanvar(rv)	Add an existing random variable to the random vector rvec
rvec:SetCorrelation(corr)	Assign the matrix corr as correlation matrix to the random vector rvec, This matrix must be square and its size must match the number of random variables contained in rvec. Also, the matrix must contain unit entries on the main diagonal and be positive definite
A=rvec:Simulate(100, stoch.Sobol)	Create a matrix A containing 100 Sobol samples of the random vector rvec. Samples are arranged column-wise in A

m = stoch.Mean(A)	Computes the mean value vector of the samples contained in the matrix A. Averaging is performed over all columns
s = stoch.Sigma(A)	Computes a vector s containing the standard deviations of the samples contained in A. Averaging is performed over all columns
s = stoch.Skewness(A)	Computes a vector s containing the coefficients of skewness of the samples contained in A. Averaging is performed over all columns
s = stoch.Kurtosis(A)	Computes a vector s containing the coefficients of kurtosis of the samples contained in A. Averaging is performed over all columns

cov = stoch.Covariance(A)	Computes the covariance matrix of the samples contained in A. Averaging is performed over all columns
corr= stoch.Correlation(A)	Computes the linear (Pearson) correlation matrix of the samples contained in A. Averaging is performed over all columns
spear= stoch.Spearman(A)	Computes the Spearman rank order correlation matrix of the samples contained in A. Averaging is performed over all columns
tau= stoch.Kendall(A)	Computes the Kendall tau correlation matrix of the samples contained in A. Averaging is performed over all columns

2.5 Module optimize

This module provides functionality for the gradient-bases optimized CONMIN as well as a zero-order particle swarm optimizer (PSO)

ops = optimize.Conmin(3, 2)	Creates an optimizer object of type CONMIN for 3 design variables and 2 inequality constraints
ops:SetDesign(A)	Assign a current design contained in the vector A to the optimizer object ops. The size of the vector A must match the number of design variables of the optimizer ops
x = ops:GetDesign()	Creates a vector x containing the current design as computed by the optimizer
ops:SetBounds(B)	Assign the matrix B as bounds to the optimizer ops. The matrix must have two columns and as many rows as there are design variables.
answer = ops:Compute()	Compute one optimization step. The algorithm is finished if answer = 0. Otherwise, you will need to compute objective function and constraint and assign these to the optimizer
ops:SetObjective(obj)	Assigns the number obj as objective to the optimizer ops.
ops.SetConstraints(constr)	Assigns the matrix constr as constraints values to the optimizer ops. constr must be a Matrix even if there is only one constraints. For an unconstrained problem, this function is not needed.

pso = optimize.PSO(3, omega)	Create a PSO object for 3 design variable. PSO cannot handle constraints directly, you will need to formulate penalties for constraints violation. omega is the so-called ”inertia term”. Choose omega = 0.8 in the absence of further knowledge

pso:SetBounds(B)	Assign the matrix B as bounds to the optimizer pso. The matrix must have two columns and as many rows as there are design variables.
x = pso:Start(30)	Create a matrix x containing an initial swarm of size 30. Designs are ordered column wise in the matrix x
pso:SetFitness(f)	Assign the contents of the vector f as fitness to the optimizer pso. The values contained in f must be computed from the last swarm received from the optimizer pso (either through pso:Start() or pro:Compute())
x = pso:Compute(clamp)	Create a matrix x containing the next swarm for the PSO algorithm. clamp defines limiting factor on velocities to avoid leaving the bounds. Use clamp = 0.2 in the absence of further knowledge
b = pso:GetBestDesign()	Retrieve the best design computed so far from the optimizer pso.

2.6 Module spectral

This module provides functionality for spectral analysis.

f, domega = spectral.FFT(x, dt)	Computes the discrete Fourier transform of the real-valued sequence contained in the vector x with time interval dt. The Fourier transform is contained in the matrix f in which the first column contains the real parts, and the second column contains the imaginary parts. The frequency interval based on the Nykvist theorem is returned in the number domega
x, dt = spectral.IFT(f, domega)	Computes the inverse discrete Fourier transform of the complex sequence contained in the matrix f with frequency interval domega. The inverse Fourier transform is contained in the vector x. The time interval based on the Nykvist theorem is returned in the number dt
spec, omega = spectral.AutoSpectrum(f, dt)	Computes an estimate for the auto power spectral density of the time series contained in the vector f. The time interval is dt. The matrix spec contains the frequency range in the first column, and the spectral estimates in the second column.

spec, omega = spectral.CrossSpectrum(f, g, dt)	Computes an estimate for the cross power spectral density of the time series contained in the vectors f an. The time interval is dt. The matrix spec contains the frequency range in the first column, the real part of the cross spectrum in the second column, and the imaginary part in the third column.
B = spectral.Butterworth(A, dt, omega, p)	Applies a low-pass Butterworth filter with cut-off frequency omega and order p to the time series contained in the vector A with time step dt. The filtered time series is returned in the vector B.

2.7 Module ode

This module provides functionality for the solution of systems of first order differential equation. The constructors for the types RK4 and Radau5 take a string argument e.g. ”Derivatives” which defines the derivatives of the state variables (see the table below). This string argument is interpreted as the name of a Lua-function. This function must take two arguments t and y, in which t is a number giving the current time (i.e. the independent variable) and y is a Lua-table containing the current values of the state vector. Individual values from y can be retrieved using the []-operator. The function returns a matrix z of the same size as y containing the derivatives. As an example, for a SDOF oscillator with mass *m* and stiffness *k* in free vibration, this function would be

```
function Derivative(t, y)
  local z = tmath.Matrix(2)
  z[0] = y[1]
  z[1] = -k/m*y[0] -- Using global variables k and m
  return z
end
```

The constructor for Radau5 takes an optional string argument ”Jacobian” which defines the name of a Lua-function computing the Jacobian matrix of the system (i.e. all partial derivatives of the state derivative with respect to the state variables). For the SDOF oscillator, this function would simply be

```
function Jacobian(t, y)
  local z = tmath.Matrix(2,2)
  z[{0,0}] = 0
  z[{0,1}] = 1
  z[{1,0}] = -k/m  -- Using global variables k and m
  z[{1,1}] = 0
  return z
end
```

DE = ode.RK4(3, ”Derivative”)	Creates an differential equation object of type RK4 (explicit Runge-Kutta 4th order). The differential equation has 3 state variables, and the derivative is defined in the Lua-function ”Derivative”.

DE = ode.Radau5(3, ”Derivative”)	Creates an differential equation object of type Radau5 (implicit Runge-Kutta, Hairer & Wanner). The differential equation has 3 state variables, and the derivative is defined in the Lua-function ”Derivative”.
DE = ode.Radau5(3, ”Derivative”, ”Jacobian”)	Creates an differential equation object of type Radau5. The differential equation has 3 state variables, and the derivative is defined in the Lua-function ”Derivative”, the Jacobian matrix is defined in the Lua-function ”Jacobian”.
DE:SetState(start)	Assigns the values of the vector start as current state vector to the differential equation object DE. This is typically used to define initial conditions. Works for both RK4 and Radau5.
resp = DE:Compute(0, T, N)	Compute the solution of the system of differential equations by integrating over the independent variable from 0 to T. The result is returned in the matrix ”reps” containing the values at N points between 0 and T. Works for both RK4 and Radau5.
resp = DE:Compute(0, T, N, 4)	Compute the solution of the system of differential equations by integrating over the independent variable from 0 to T. The result is returned in the matrix ”reps” containing the values at N points between 0 and T. For increased precision, each time interval is subdivided into 4 sub-steps. Works for RK4 only.

2.8 Module fem

This is a simple finite element module aiming at linear static and dynamic analysis. The element library is fairly limited and only linear-elastic material is available for general element types. Only geometrically linear analysis can be performed.

structure=fem.Structure(”Frame”)	Creates a Structure object (finite element model) with the name ”Frame”
structure:AddNodes(nodes)	Adds nodes to the finite element model. The matrix nodes contains as many rows as there are nodes to be added and 4 columns. The first column contains a unique identifier for each node, and the remaining 3 columns contain the x, y, and z coordinates of the nodes.
s = structure:AddSection(101, ”RECT”)	Creates a new section with unique identifier 101 of type ”RECT” and adds it to the structure. Other admissible section types are ”SHELL” for shell elements and ”VOLUME” for volume elements.

s:SetColor(col)	Sets the color of the section s. The argument col is a vector of size 4 containing the color in RGBA specification. Color values range between 0 and 255.
s:SetData(values)	Sets the section data to those contained in values. The size of the vector values must match the number of data required by the section type of s. For type ”RECT”, this number is 2. For type ”SHELL” it is 1, and for ”VOLUME” it is 0.
m=structure:AddMaterial(201, ”LINEAR_ELASTIC”)	Creates a new material with unique identifier 201 of type ”LINEAR_ELASTIC” and adds it to the structure.
m:SetData(values)	Sets the material data to those contained in values. The size of the vector values must match the number of data required by the material type of m. Other material types are ”DAMPER” and ”CONTACT”. For type ”LINEAR_ELASTIC”, this number is 3 (modulus of elasticity, Poisson’s ration, mass density). For ”DAMPER” it is 2 (damping constant, velocity exponent), for ”CONTACT” it is 3 (normal stiffness=lateral stiffness before slip, lateral stiffness after slip, maximum friction force)
structure:AddElements(”RECT”, 201, 101, elms)	Adds elements pf type ”RECT” with material 201 and section 101 to the structure. The matrix elms contains as many rows as elements should be added, the first column contains a unique ID for each element, and the remaining columns contain a list of node IDs (as many as required by the element type). Available element types and necessary number of nodes are: ”RECT” (3), ”TRIANGLE3N” (3), ”TET4N” (4), ”TRUSS” (3), ”DAMPER” (3), ”CONTACT” (3), ”LINE” (2).
structure:SetAvailDof(dofs)	Sets the available DOFs for all nodes of the structure. dofs is a vector of size 6 containing 1 in a specific position if the respective DOF should be available (free) and 0 if the respective DOF should not be available (restricted).
nd=structure:GlobalDof()	Assigns the global DOF numbers to all nodes and returns the total number of DOFs. This method must be called after the definition of nodes, elements and constraints, and before carrying out any structural analysis!
M = structure:SparseMass()	Assembles the global mass matrix for the whole structure in sparsearray form.

C = structure:SparseDamping()	Assembles the global damping matrix for the whole structure in sparsearray form.
K = structure:SparseStiffness()	Assembles the global stiffness matrix for the whole structure in sparsearray form.
U = K:Solve(F)	Solves the system of equations $\mathbf{KU} = \mathbf{F}$. The factorization of K is stored in K itself and will be re-used for the solution of another right hand side.
structure:SetDofDisplacements(U)	Assign the values contained in U to the nodes as displacements according to the respective DOFs.
U=structure:GetDofDisplacements()	Retrieve the displacements from all DOFs of all nodes and store them in the vector U
structure:SetAllDisplacements(U1)	Assign the values contained in the matrix U1 as displacements to all nodes regardless of available DOFs. U1 has as many rows as there are nodes, and 6 columns (3 translations, 3 rotations).
U1 = structure:GetAllDisplacements()	Retrieve the displacements of all nodes regardless of available DOFs and stores them in the matrix U1. U1 has as many rows as there are nodes, and 6 columns (3 translations, 3 rotations).
F2=structure:ToDofDisplacements(F1)	Converts a matrix containing displacements in matrix form (number of nodes x 6) into a vector containing the displacement of all available DOFs.
F1=structure:ToAllDisplacements(F2)	Converts a vector containing displacements of all available DOFs into matrix form (number of nodes x 6) containing all displacement .
R = structure:GlobalResForce()	Computes the vector of displacement-dependent nodal restoring forces from all elements. Typically needed for nonlinear analysis.
structure:GlobalUpdate()	Accept all currently computed displacements and material stresses as final and update the material history variables accordingly. Typically used in nonlinear analysis.
str = ElementStress(comp)	Computes the stresses in all elements and returns one component of the stress tensor for all elements. The number of values returned is equal to the total sum of stress points in the elements, which depends on the element types. if comp=0, the von Mises stress is returned. Other possible values for comp are 1(σ_{xx}), 2(σ_{yy}), 3(σ_{zz}, 4(τxy, 5(τ_{xz}, 6(τ_{yz}).

tri=structure:Draw(fac)	Triangulate the entire structure for later use in a 3D Graph. The deformations are multiplied by a factor fac and are then added to the nodal coordinates. The triangles are stored in the matrix tri.
tri = structure:ElementResults(str, fac)	Triangulates the entire structure coloring the elements according to the stress values contained in the vector str. The structure is drawn in deformed state with a factor fac. The triangles are stored in the matrix tri.
lin = structure:Vector(U1, fac)	Prepares 3D lines representing the nodal vector U2. It is drawn such that the vectors start from the deformed structure (factor fac) . The lines are stored in the matrix lin.

n = structure:GetNode(key)	Get a reference to the node with ID key.
n:SetAvailDof(dofs)	Sets the available DOFs of node n. dofs is a vector of size 6 containing 1 for free DOFs and 0 for restrained DOFs.