IN480 Larstruct Module

Desiree Adiutori Alessia Giulia Cossu

June 22, 2018

Contents

Introduction **4**

1 API **5**

2 Implementation **10**
2.1 checkStruct . . . 10
2.1.1 Conversion . . . 10
2.1.2 Parallelization . . . 10
2.1.3 Unit-Test . . . 11
2.2 larApply . . . 12
2.2.1 Conversion . . . 12
2.2.2 Parallelization . . . 14
2.2.3 Unit-Test . . . 14
2.2.4 Results . . . 16
2.3 box . . . 18
2.3.1 Conversion . . . 18
2.3.2 Parallelization . . . 19
2.3.3 Unit-Test . . . 20
2.3.4 Results . . . 21
2.4 traversal . . . 25
2.4.1 Conversion . . . 25
2.4.2 Parallelization . . . 25
2.4.3 Unit-Test . . . 26
2.4.4 Results . . . 26
2.5 Struct . . . 30
2.5.1 Conversion . . . 30
2.5.2 Parallelization . . . 32
2.5.3 Unit-Test . . . 34

2.5.4 Results . . . 34
2.6 embedTraversal . . . 38
2.6.1 Conversion . . . 38
2.6.2 Parallelization . . . 39
2.6.3 Unit-Test . . . 40
2.6.4 Results . . . 41
2.7 embedStruct . . . 43
2.7.1 Conversion . . . 43
2.7.2 Parallelization . . . 43
2.7.3 Unit-Test . . . 44
2.7.4 Results . . . 45
2.8 removeDups . . . 47
2.8.1 Conversion . . . 47
2.8.2 Parallelization . . . 47
2.8.3 Unit-Test . . . 48
2.8.4 Results . . . 48
2.9 struct2lar . . . 50
2.9.1 Conversion . . . 50
2.9.2 Parallelization . . . 52
2.9.3 Unit-Test . . . 54
2.9.4 Results . . . 55
2.10 larRemoveVertices . . . 58
2.10.1 Conversion . . . 58
2.10.2 Parallelization . . . 58
2.10.3 Unit-Test . . . 59
2.11 Result . . . 59

3 Examples **61**

4 Conclusion **63**

References **63**

List of Figures

1 API . . . 6
2 Execution time of function larApply . . . 17
3 Compare Parallel and Serial execution time of function larApply . . . 17
4 Execution time of function box . . . 22
5 Compared execution time of function box . . . 22

6 Execution time of function box on Tesla . . . 23
7 Compared Execution time of function box on Tesla . . . 24
8 Execution time of function traversal . . . 27
9 Compared execution time of function traversal . . . 28
10 Execution time of function traversal on Tesla . . . 29
11 Compared execution time of function traversal on Tesla . . . 29
12 Execution time of function Struct . . . 35
13 Compared execution time of function Struct . . . 35
14 Execution time of function Struct on Tesla . . . 36
15 Compared execution time of function Struct on Tesla . . . 37
16 Execution time of function embedTraversal on Tesla . . . 42
17 Compared execution time of function embedtraversal on Tesla . . . 42
18 Execution time of function embedStruct on Tesla . . . 45
19 Compared execution time of function embedtraversal on Tesla . . . 46
20 Execution time of function removeDups . . . 49
21 Compared execution time of function removeDups . . . 49
22 Execution time of function struct2lar . . . 55
23 Compared execution time of function struct2lar . . . 56
24 Execution Time of function struct2lar on Tesla . . . 57
25 Compared execution time of function struct2lar on Tesla . . . 57
26 Execution time of function larRemoveVertices . . . 60
27 Compared execution time of function larRemoveVertices . . . 60
28 Table with four chair . . . 61
29 Classroom . . . 62

Introduction

This module of LAR-CC library deals with the hierarchical structures with LAR. Hierarchical models of complex assemblies are generated by an aggregation of subassemblies, each one defined in a local coordinates's system, and relocated by affine transformations of coordinates.

In this module there are:

- The **Affine transformations** whom are based on elementary matrices for affine transformations of vectors in any dimensional vector space, including translation, scaling and rotation;
- The **Struct iterable class**, starting from an array representable geometric object, generates a new object representing the initial one in an alternative way, by means of specific fields attribution, e.g. body, box, etc..
- The **structure to LAR conversion** is based on functions for the embedding of two-dimensional LAR model into 3D space. It removes duplicate faces, vertices and cells of geometric objec

In the next sections the main functions of the module will be shown through the API. For each function will be proposed the codes' conversion from Python to Julia language, a parallelization frame, some unit tests and the study of the execution times.

1 API

- **larApply**(Matrix)(Tuple)→ Tuple

  Through the affine matrix given in input, it performs an affine transformation and returns in output a tuple containing two arrays: the list of the transformed coordinates of the vertices and a cells vector.

- **evalStruct**(Struct)→ Array

  It analyzes the elements contained in the Struct object and returns an array containing the main data structures. Each structure is described by an array containing two arrays: the list of the transformed coordinates of the vertices and a cells vector.

- **vcode**(Int)(Array)→String

  It approximates each element of the data structure to which it is applied. The approximation occurs with precision given by the integer given in input. The transformed data structure is returned in output as a string.

- **struct2lar**(Struct)→ Array

  It converts an object of type Struct in a pair (Vertices,Cells) containing the array of the coordinates of the vertices (of any the objects present) and a cells vector (without duplicates of vertices).

- **larRemoveVertices**(Array, Array)→ (Array,Array)

  It takes in input an array of vertices an the array of cells and removes any duplicates from them.

- **larEmbed**(Int)(Tuple)→ Tuple

  If allows to immerse a k-dimensional geometric object into an n-dimensional space with $n > k$, or to pull it back to its representation in a space of dimension n starting from its representation in a space of dimension $k > n$. Namely, given in input n and the tuple T representing the geometric object in a space of dimension k the function will return the representation of the object T in a space of dimension $k + n$. Note that the value of n can be negative. In order to understand the rational behind this function, it is enough to think about the application of 3D transformations to a two-dimensional LAR model, in this case it needs to be embedded in a 3D space adding one more coordinate to its vertices.

- **larBoundary**(Struct)→ (Array,Array,Array)

  It apply struct2lar to the input. If the output has dimension 3, the function returns a tuple containing 3 outputs, otherwise it returns a string containing an error message.

- **embedStruct**(Int)(Struct)→ Struct

  It returns a copy of the geometrical object of dimension k represented by the structure in input. The dimension of the copy will be equal to that of the original one plus the value of the integer n given in input.

Local Functions

- **checkStruct**(Array)→ Int
- **traversal**(Matrix,Array,Struct,Array)→ Array
- **fixedPrec**(Int)→ String
- **removeDups**(Array)→ Array

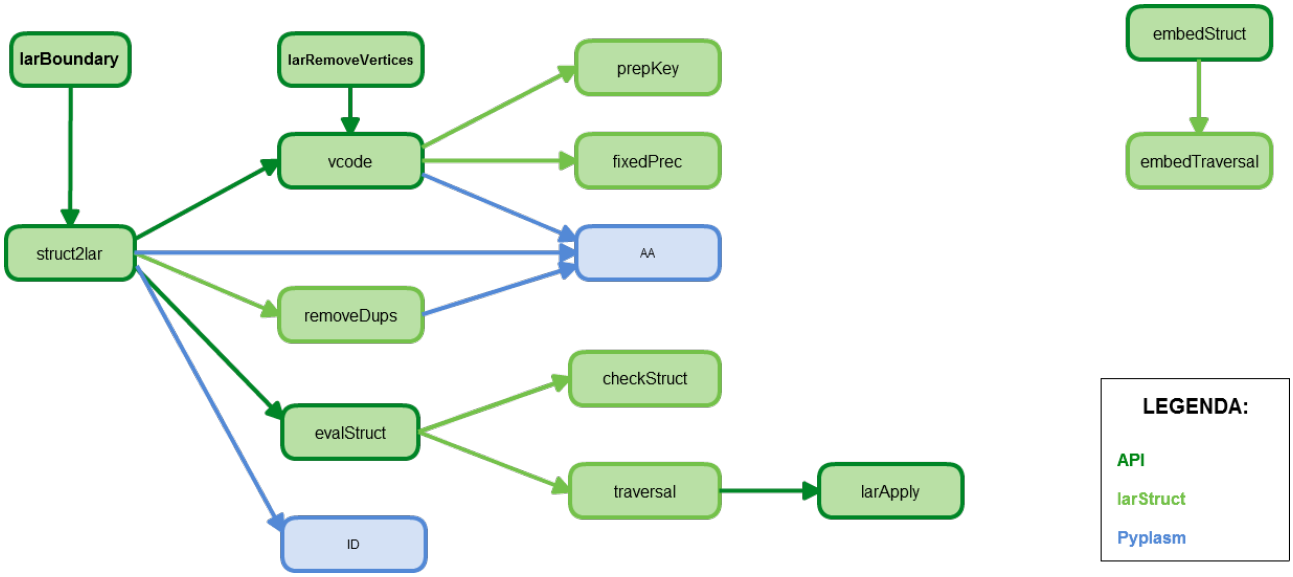


Figure 1: larStruct API

This module is divided into three sections: **"Implementation"**, **"Examples"** and **"Conclusion"**.

The section "Implementation" contains for each function four sub-sections. The sub-section **"Conversion"** shows the conversion of some of the most important functions of the module from Python to Julia language. The sub-section **"Parallelization"** is based on the use of the following Julia's macros to parallelize the code:

- **@everywhere**: can be used to directly define a function on all processes
- **@parallel**: can be used to run a for loop on any number of processes
- **@sync**: wait until all dynamically-enclosed uses of @async, @spawn, @spawnat and @parallel are complete
- **@async**: wraps an expression in a Task and adds it to the local machine's scheduler queue. Additionally it adds the task to the set of items that the nearest enclosing @sync waits for.

The Parallel computing is performed on 4 simultaneous processes in a personal computer running the command

```
addprocs(3)
```

within Julia console. Alternatively the parallelization can be performed running on the shell the command

```
julia -p 3
```

Parallel computing is performed also on 30 simultaneous processes running the command

```
addprocs(29)
```

within Julia console (in "Tesla") or running on the shell the command

```
julia -p 29
```

The main tasks in the sub-section **"Unit-Test"** are carried out via the use of a particular Julia module, named **Base.Test**. The latter provides simple unit testing functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete. To perform a simple unit testing, you need first to import the "Base.Test" package typing the following command within the Julia console:

```
using Base.Test
```

Then the test can be performed with the @test() macro.

The sub-section **"Result"** are divided into two parts: in the first part there are the plots representing the execution time of the underlying function performed on a personal computer with 4 processors. The magnitude of the input is increased by means of the function "Lar-Cuboids" (which is imported by the "largrid" module):

```
using PyCall
@pyimport larlib as lar
l=[]
for i in range(0,3)
    p=PyObject(lar.larCuboids([10^i,10^i]))
    push!(l,Tuple(append!([map(collect,PyObject(p[1]))],[PyObject(p[2])])))
    p=PyObject(lar.larCuboids([2*10^i,2*10^i]))
    push!(l,Tuple(append!([map(collect,PyObject(p[1]))],[PyObject(p[2])])))
    p=PyObject(lar.larCuboids([5*10^i,5*10^i]))
    push!(l,Tuple(append!([map(collect,PyObject(p[1]))],[PyObject(p[2])])))
end
```

The function used to compute the execution time is:

```
function Time(f,args)
    @elapsed f(args...)
    t=[]
    for i in range(1,10)
        push!(t,@elapsed f(args...))
    end
    m=mean(t)
    return m
end
```

The macro @elapsed is used to evaluates the execution time of the function given in input, discarding the resulting value, and returning just the amount of seconds elapsed to run the function. The time is expressed as a floating-point number.

The second part of the subsection "Result" contains the plot of the underlying function performance, run on "Tesla" with 30 processors, where the input magnitute is increased via the following function:

```
function addn2D(n,model)
  body=[]
  for i in range(1,n)
    el=[]
    matrix=rand(1:3)
    if matrix==1
      x=rand(1:10)/10
      y=rand(1:10)/10
      append!(el,larApply(t(x,y))(model))
      append!(body,[el])
    elseif matrix ==2
```

```
      x=rand(1:10)/10
      y=rand(1:10)/10
      append!(el,larApply(s(x,y))(model))
      append!(body,[el])
    elseif matrix ==3
      x=rand(1:10)
      append!(el,larApply(r(pi/x))(model))
      append!(body,[el])
    end
  end
  a=Struct(body)
  return a
end
```

To generate the plots you need to import "Plots" and "Distributions" packages typing the following commands within the Julia console:

- `using Plots`
- `using Distributions`

2 Implementation

2.1 checkStruct

2.1.1 Conversion

Python

```
def checkStruct(lst):
    obj = lst[0]
    if(isinstance(obj,tuple) or isinstance(obj,list)):
        dim = len(obj[0][0])
    elif isinstance(obj,Model):
        dim = obj.n
    elif isinstance(obj,Mat):
        dim = obj.shape[0]-1
    elif isinstance(obj,Struct):
        dim = len(obj.box[0])
return dim
```

Julia

```
function checkStruct(lst)
  obj = lst[1]
  if isa(obj,Matrix)
      dim=size(obj)[1]-1
  elseif(isa(obj,Tuple) || isa(obj,Array))
      dim=length(obj[1][1])
  elseif isa(obj,Struct)
      dim=length(obj.box[1])
  end
  return dim
end
```

2.1.2 Parallelization

```
function pcheckStruct(lst)
  obj = lst[1]
  if isa(obj,Matrix)
    dim=size(obj)[1]-1
  elseif(isa(obj,Tuple) || isa(obj,Array))
    dim=length(obj[1][1])
  elseif isa(obj,pStruct)
    dim=length(obj.box[1])
  end
  return dim
end
```

2.1.3 Unit-Test

```
@testset "checkStruct Tests" begin
  list=([[0.575,-0.175],[0.575,0.175],[0.925,-0.175],[0.925,0.175]],[[0,1,2,3]])
@test checkStruct(list)==length(list[1][1][1])
@test typeof(checkStruct(list))==Int
end
```

2.2 larApply

2.2.1 Conversion

The function larApply returns as output an affine trasformation of the input object and it is based on the following affine matrices:

Rotation

Python

```
def r(*args):
  args = list(args)
  n = len(args)
  if n == 1: # rotation in 2D
    angle = args[0]; cos = COS(angle); sin = SIN(angle)
    mat = scipy.identity(3)
    mat[0,0] = cos;    mat[0,1] = -sin;
    mat[1,0] = sin;    mat[1,1] = cos;
  if n == 3: # rotation in 3D
    mat = scipy.identity(4)
    angle = VECTNORM(args); axis = UNITVECT(args)
    cos = COS(angle); sin = SIN(angle)
    if axis[1]==axis[2]==0.0:    # rotation about x
      mat[1,1] = cos;    mat[1,2] = -sin;
      mat[2,1] = sin;    mat[2,2] = cos;
    elif axis[0]==axis[2]==0.0:    # rotation about y
      mat[0,0] = cos;    mat[0,2] = sin;
      mat[2,0] = -sin;    mat[2,2] = cos;
    elif axis[0]==axis[1]==0.0:    # rotation about z
      mat[0,0] = cos;    mat[0,1] = -sin;
      mat[1,0] = sin;    mat[1,1] = cos;
    else:  # general 3D rotation
      I = scipy.identity(3) ; u = axis
      Ux = scipy.array([
          [0,        -u[2],      u[1]]
          [u[2],        0,     -u[0]],
          [-u[1],     u[0],         0]])
      UU = scipy.array([
          [u[0]*u[0],    u[0]*u[1],    u[0]*u[2]],
          [u[1]*u[0],    u[1]*u[1],    u[1]*u[2]],
          [u[2]*u[0],    u[2]*u[1],    u[2]*u[2]]])
          mat[:3,:3] = cos*I + sin*Ux + (1.0-cos)*UU
return mat.view(Mat)
```

Julia

```
@everywhere function r(args...)
   args = collect(args)
   n = length(args)
   if n == 1 # rotation in 2D
     angle = args[1]; COS = cos(angle); SIN = sin(angle)
     mat = eye(3)
     mat[1,1] = COS;    mat[1,2] = -SIN;
     mat[2,1] = SIN;    mat[2,2] = COS;
   end
   if n == 3 # rotation in 3D
     mat = eye(4)
     angle = norm(args); axis = normalize(args)
     COS = cos(angle); SIN= sin(angle)
     if axis[2]==axis[3]==0.0    # rotation about x
       mat[2,2] = COS;    mat[2,3] = -SIN;
       mat[3,2] = SIN;    mat[3,3] = COS;
     elseif axis[1]==axis[3]==0.0   # rotation about y
       mat[1,1] = COS;    mat[1,3] = SIN;
       mat[3,1] = -SIN;    mat[3,3] = COS;
     elseif axis[1]==axis[2]==0.0    # rotation about z
       mat[1,1] = SIN;    mat[1,2] = -SIN;
       mat[2,1] = COS;    mat[2,2] = COS;
     else
       I=eye(3); u=axis
       Ux=[0 -u[3] u[2] ; u[3] 0 -u[1] ;  -u[2] u[1] 1]
       UU =[u[1]*u[1]    u[1]*u[2]   u[1]*u[3];
           u[2]*u[1]    u[2]*u[2]   u[2]*u[3];
           u[3]*u[1]    u[3]*u[2]   u[3]*u[3]]
       mat[1:3,1:3]=COS*I+SIN*Ux+(1.0-COS)*UU
     end
   end
 return mat
 end
```

Translation

Python

```
def t(*args):
  d = len(args)
  mat = scipy.identity(d+1)
  for k in range(d):
    mat[k,d] = args[k]
  return mat.view(Mat)
```

Julia

```
@everywhere function t(args...)
   d=length(args)
   mat=eye(d+1)
   for k in range(1,d)
     mat[k,d+1]=args[k]
   end
   return mat
 end
```

Scaling

Python

```
def s(*args):
  d = len(args)
  mat = scipy.identity(d+1)
  for k in range(d):
    mat[k,k] = args[k]
  return mat.view(Mat)
```

Julia

```
@everywhere function s(args...)
  d=length(args)
  mat=eye(d+1)
  for k in range(1,d)
    mat[k,k]=args[k]
  end
  return mat
end
```

Python

```
def larApply(affineMatrix);
  def larApply0(model):
    if isinstance(model,Model):
      V = scipy.dot(array([v+[1.0] for v in model.verts]),affineMatrix.T).tolist()
      V = [v[:-1] for v in V]
      CV = copy.copy(model.cells)
      return Model((V,CV))
    elif isinstance(model,tuple) or isinstance(model,list):
      if len(model)==2: V,CV = model
      elif len(model)==3: V,CV,FV = model
      V=scipy.dot([list(v)+[1.0]for v in V],affineMatrix.T).tolist()
      if len(model)==2: return [v[:-1] for v in V],CV
      elif len(model)==3: return [v[:-1] for v in V],CV,FV
  return larApply0
```

Julia

```
function larApply(affineMatrix)
  function larApply0(model)
    if length(model)==2
      V,CV=model
    elseif length(model)==3
      V,CV,FV = model
    end
    V1=Array{Float64}[]
    for (k,v) in enumerate(V)
      append!(v,[1.0])
      push!(V1,vec((v')*transpose(affineMatrix)))
      pop!(V[k])
      pop!(V1[k])
    end
    if length(model)==2
      return V1,CV
    elseif length(model)==3
      return V1,CV,FV
    end
```

```
   end
   return larApply0
end
```

2.2.2 Parallelization

The affine transformation matrices are the same as in the sequential case.

```
@everywhere function plarApply(affineMatrix)
  function plarApply0(model)
    if length(model)==2
      V,CV=deepcopy(model)
    elseif length(model)==3
      V,CV,FV = deepcopy(model)
    end
    V1=Array{Float64}[]
    V1=@sync @parallel (append!)for v in V
      append!(v,[1.0])
      [collect(vec((v')*transpose(affineMatrix)))]
    end
    for v in V1
      pop!(v)
    end
    if length(model)==2
      return fetch(V1),CV
    elseif length(model)==3
      return V1,CV,FV
    end
  end
  return plarApply0
end
```

2.2.3 Unit-Test

Serial Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
cubes=([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]],[[0,1,2,3,4,5,6,7]])

@testset "larApply Tests" begin
  @testset "2D" begin
    @testset "larApply Translation 2D" begin
      @test typeof(larApply(t(-0.5,-0.5))(square))==Tuple{Array{Array{Float64,N} where N,1},
      Array{Array{Int64,1},1}}
      @test larApply(t(-0.5,-0.5))(square)==([[-0.5,-0.5],[-0.5,0.5],[0.5,-0.5],[0.5,0.5]],
      [[0,1,2,3]])
    end
```

```
    @testset "larApply Scaling 2D" begin
      @test typeof(larApply(s(-0.5,-0.5))(square))==Tuple{Array{Array{Float64,N} where N,1},
      Array{Array{Int64,1},1}}
      @test larApply(s(-0.5,-0.5))(square)==([[0.0,0.0],[0.0,-0.5],[-0.5,0.0],[-0.5,-0.5]],
      [[0,1,2,3]])
    end

    @testset "larApply Rotation 2D" begin
      @test typeof(larApply(r(0))(square))==Tuple{Array{Array{Float64,N}  where N,1},
      Array{Array{Int64,1},1}}
      @test larApply(r(0))(square)==square
    end
  end

  @testset "3D" begin
    @testset "larApply Translation 3D" begin
      @test typeof(larApply(t(-0.5,-0.5,-0.5))(cubes))==Tuple{Array{Array{Float64,N}
      where N,1},Array{Array{Int64,1},1}}
      @test larApply(t(-0.5,-0.5,-0.5))(cubes)==([[-0.5,-0.5,-0.5],[-0.5,-0.5,0.5],
      [-0.5,0.5,-0.5],
      [-0.5,0.5,0.5],[0.5,-0.5,-0.5],[0.5,-0.5,0.5],[0.5,0.5,-0.5],[0.5,0.5,0.5]],
      [[0,1,2,3,4,5,6,7]])
    end

    @testset "larApply Scaling 3D" begin
      @test typeof(larApply(s(-0.5,-0.5,-0.5))(cubes))==Tuple{Array{Array{Float64,N}
      where N,1},Array{Array{Int64,1},1}}
      @test larApply(s(-0.5,-0.5,-0.5))(cubes)==([[0.0,0.0,0.0],[0.0,0.0,-0.5],[0.0,-0.5,0.0],
      [0.0,-0.5,-0.5],[-0.5,0.0,0.0],[-0.5,0.0,-0.5],[-0.5,-0.5,0.0],[-0.5,-0.5,-0.5]],
      [[0,1,2,3,4,5,6,7]])
    end

    @testset "larApply Rotation 3D" begin
      @test typeof(larApply(r(pi,0,0))(cubes))==Tuple{Array{Array{Float64,N} where N,1},
      Array{Array{Int64,1},1}}
      @test isapprox(larApply(r(pi,0,0))(cubes)[1],[[0.0,0.0,0.0],[0.0,-1.22465e-16,-1.0],
      [0.0,-1.0,1.22465e-16],[0.0,-1.0,-1.0],[1.0,0.0,0.0],[1.0,-1.22465e-16,-1.0],
      [1.0,-1.0,1.22465e-16],[1.0,-1.0,-1.0]])
    end
  end
end
```

Parallel Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
cubes=([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]],[[0,1,2,3,4,5,6,7]])
@testset "plarApply Tests" begin
  @testset "2D" begin
    @testset "plarApply Translation 2D" begin
```

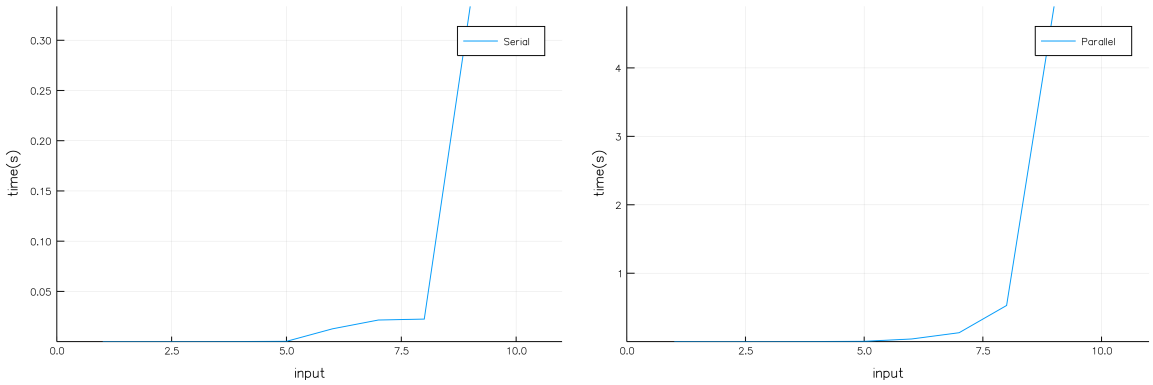
```
      @test typeof(plarApply(t(-0.5,-0.5))(square))==Tuple{Array{Array{Float64,1},1},
      Array{Array{Int64,1},1}}
      @test plarApply(t(-0.5,-0.5))(square)==([[-0.5,-0.5],[-0.5,0.5],[0.5,-0.5],[0.5,0.5]],
      [[0,1,2,3]])
    end
    @testset "plarApply Scaling 2D" begin
      @test typeof(plarApply(s(-0.5,-0.5))(square))==Tuple{Array{Array{Float64,1},1},
      Array{Array{Int64,1},1}}
      @test plarApply(s(-0.5,-0.5))(square)==([[0.0,0.0],[0.0,-0.5],[-0.5,0.0],
      [-0.5,-0.5]],[[0,1,2,3]])
    end
    @testset "plarApply Rotation 2D" begin
      @test typeof(plarApply(r(0))(square))==Tuple{Array{Array{Float64,1},1},
      Array{Array{Int64,1},1}}
      @test plarApply(r(0))(square)==square
    end
  end
  @testset "3D" begin
    @testset "plarApply Translation 3D" begin
    @test typeof(plarApply(t(-0.5,-0.5,-0.5))(cubes))==Tuple{Array{Array{Float64,1},1},
    Array{Array{Int64,1},1}}
    @test plarApply(t(-0.5,-0.5,-0.5))(cubes)==([[-0.5,-0.5,-0.5],[-0.5,-0.5,0.5],[-0.5,0.5,-0.5],
    [-0.5,0.5,0.5],[0.5,-0.5,-0.5],[0.5,-0.5,0.5],[0.5,0.5,-0.5],[0.5,0.5,0.5]],[[0,1,2,3,4,5,6,7]])
  end
    @testset "plarApply Scaling 3D" begin
      @test typeof(plarApply(s(-0.5,-0.5,-0.5))(cubes))==Tuple{Array{Array{Float64,1},1},
      Array{Array{Int64,1},1}}
      @test plarApply(s(-0.5,-0.5,-0.5))(cubes)==([[0.0,0.0,0.0],[0.0,0.0,-0.5],[0.0,-0.5,0.0],
      [0.0,-0.5,-0.5],[-0.5,0.0,0.0],[-0.5,0.0,-0.5],[-0.5,-0.5,0.0],[-0.5,-0.5,-0.5]],
      [[0,1,2,3,4,5,6,7]])
    end
    @testset "plarApply Rotation 3D" begin
      @test typeof(plarApply(r(pi,0.0,0.0))(cubes))==Tuple{Array{Array{Float64,1},1},
      Array{Array{Int64,1},1}}
      @test isapprox(plarApply(r(pi,0,0))(cubes)[1],[[0.0,0.0,0.0],[0.0,-1.22465e-16,-1.0],
      [0.0,-1.0,-1.22465e-16],[0.0,-1.0,-1.0],
      [1.0,0.0,0.0],[1.0,-1.22465e-16,-1.0],[1.0,-1.0,1.22465e-16],[1.0,-1.0,-1.0]])
    end
  end
end
```

2.2.4 Results

Execution time on PC

```
times=[]
ptimes=[]
append!(times,Time(larApply(t(-0.5,-0.5)),[l[i]]) for i in range(1,length(l)))
append!(ptimes,Time(plarApply(t(-0.5,-0.5)),[l[i]]) for i in range(1,length(l)))
```

```
plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 2: Execution time of function larApply

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

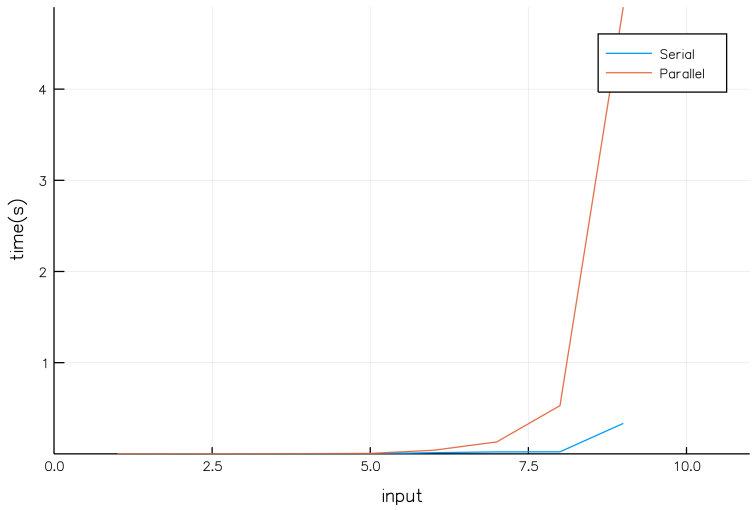


Figure 3: Compare Parallel and Serial execution time of function larApply

2.3 box

2.3.1 Conversion

Python

```
def box(model):
  if isinstance(model,Mat): return []
  elif isinstance(model,Struct):
    dummyModel = copy.deepcopy(model)
    dummyModel.body = [term if (not isinstance(term,Struct))
  else [term.box,[[0,1]]]  for term in model.body]
    listOfModels = evalStruct( dummyModel )
    theMin,theMax = box(listOfModels[0])
    for theModel in listOfModels[1:]:
      modelMin, modelMax = box(theModel)
      theMin = [val if val<theMin[k] else theMin[k] for k,val in enumerate(modelMin)]
      theMax = [val if val>theMax[k] else theMax[k] for k,val in enumerate(modelMax)]
    return [theMin,theMax]
  elif isinstance(model,Model):
    V = model.verts
  elif (isinstance(model,tuple) or isinstance(model,list)) and (len(model)==2 or len(model)==3):
    V = model[0]
  coords = TRANS(V)
  theMin = [min(coord) for coord in coords]
  theMax = [max(coord) for coord in coords]
  return [theMin,theMax]
```

Julia

```
function box(model)
  if isa(model,Matrix)
    return []
  elseif isa(model,Struct)
    dummyModel=deepcopy(model)
    dummyModel.body=Any[]
    for term in model.body
      if isa(term,Struct)
        push!(dummyModel.body,[term.box,[0,1]])
      else
        push!(dummyModel.body,term)
      end
    end
    listOfModels=evalStruct(dummyModel)
    theMin,theMax=box(listOfModels[1])
    for theModel in listOfModels[2:end]
      modelMin,modelMax= box(theModel)
      for (k,val) in enumerate(modelMin)
        if val < theMin[k]
          theMin[k]=val
```

```
        end
      end
      for (k,val) in enumerate(modelMax)
        if val > theMax[k]
          theMax[k]=val
        end
      end
    end
    return Array[theMin,theMax]
  elseif (isa(model,Tuple)||isa(model,Array)) &&(length(model)==2||length(model)==3)
    V=model[1]
    theMin=[]
    theMax=[]
    for j in range(1,length(V[1]))
      Min=V[1][j]
      Max=V[1][j]
      for i in range(1,length(V))
        Min=min(Min,V[i][j])
        Max=max(Max,V[i][j])
      end
      push!(theMin,Min)
      push!(theMax,Max)
    end
    return Array[theMin,theMax]
  end
end
```

2.3.2 Parallelization

```
function pbox(model)
  if isa(model,Matrix)
    return []
  elseif isa(model,pStruct)
    dummyModel=deepcopy(model)
    dummyModel.body=Any[]
    @sync for term in model.body
      if isa(term,pStruct)
        push!(dummyModel.body,[term.box,[0,1]])
      else
        push!(dummyModel.body,term)
      end
    end
    listOfModels=pevalStruct(dummyModel)
    theMin,theMax=pbox(listOfModels[1])
    @sync for theModel in listOfModels[2:end]
      modelMin,modelMax= pbox(theModel)
      @async begin
        for (k,val) in enumerate(modelMin)
          if (val < theMin[k])
```

```
            theMin[k]=val
          end
        end
        for (k,val) in enumerate(modelMax)
          if (val > theMax[k])
            theMax[k]=val
          end
        end
      end
      end
      return Array[theMin,theMax]
  elseif (isa(model,Tuple) ||isa(model,Array))&&(length(model)==2 || length(model)==3)
    V=model[1]
    theMin=[]
    theMax=[]
    @sync for j in range(1,length(V[1]))
      Min=V[1][j]
      Max=V[1][j]
      for i in range(1,length(V))
        Min=min(Min,V[i][j])
        Max=max(Max,V[i][j])
      end
      @async begin
        push!(theMin,Min)
        push!(theMax,Max)
      end
    end
    return Array[theMin,theMax]
  end
end
```

2.3.3 Unit-Test

These Unit-Test can be runned after the definition of the Struct type on the Section 2.5

Serial Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])

cubes=([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]],[[[0],[1],[2],[3],[4],[5],
[6],[7]],[[0,1],[2,3],[4,5],[6,7],[0,2],[1,3],[4,6],[5,7],[0,4],[1,5],[2,6],[3,7]],[[0,1,2,3],
[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7]],[[0,1,2,3,4,5,6,7]]])

@testset "box Tests" begin
  @testset "box Tests 2D" begin
    @test typeof(box(square))==Array{Array,1}
    @test length(box(square))==2
```

```
    @test length(box(square)[1])==2
  end

  @testset "box Tests 3D" begin
    @test typeof(box(cubes))==Array{Array,1}
    @test length(box(cubes))==2
    @test length(box(cubes)[1])==3
  end
end
```

Parallel Tests

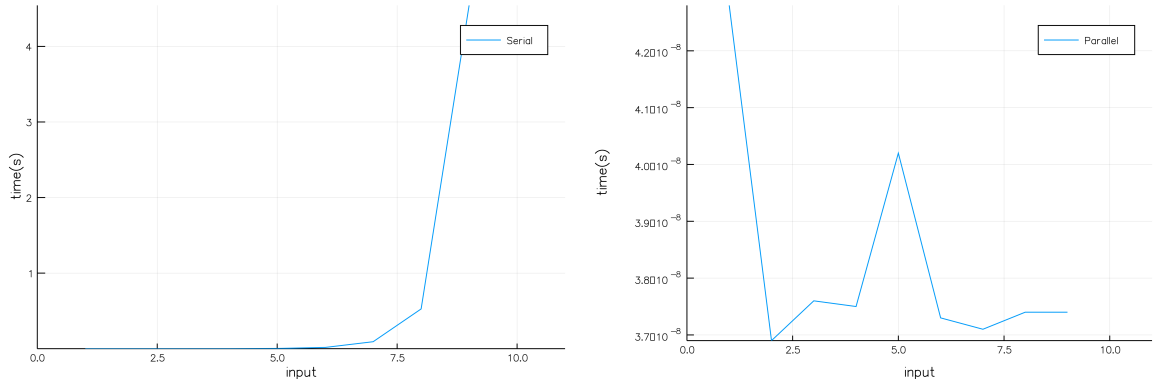
```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])

cubes=([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]],[[[0],[1],[2],[3],[4],[5],
[6],[7]],[[0,1],[2,3],[4,5],[6,7],[0,2],[1,3],[4,6],[5,7],[0,4],[1,5],[2,6],[3,7]],[[0,1,2,3],[4,5,6,7]
[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7]],[[0,1,2,3,4,5,6,7]]])

@testset "pbox Tests" begin
  @testset "pbox Tests 2D" begin
    @test typeof(pbox(square))==Array{Array,1}
    @test length(pbox(square))==2
    @test length(pbox(square)[1])==2
  end

  @testset "pbox Tests 3D" begin
    @test typeof(pbox(cubes))==Array{Array,1}
    @test length(pbox(cubes))==2
    @test length(pbox(cubes)[1])==3
  end
end
```

2.3.4 Results

Execution time on PC

```
times=[]
ptimes=[]
input=[]

for i in range(1,length(l))
  push!(input,Struct([repeat([l[i]],outer=i)...]))
end
append!(times,Time(box,[input[i]]) for i in range(1,length(input)))
append!(ptimes,Time(pbox,[input[i]])  for i in range(1,length(input)))
```

```
plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 4: Execution time of function box

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(ptimes)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

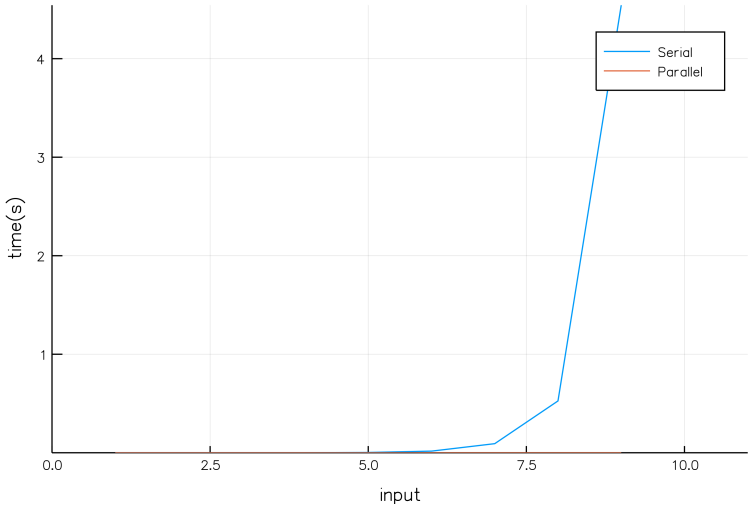
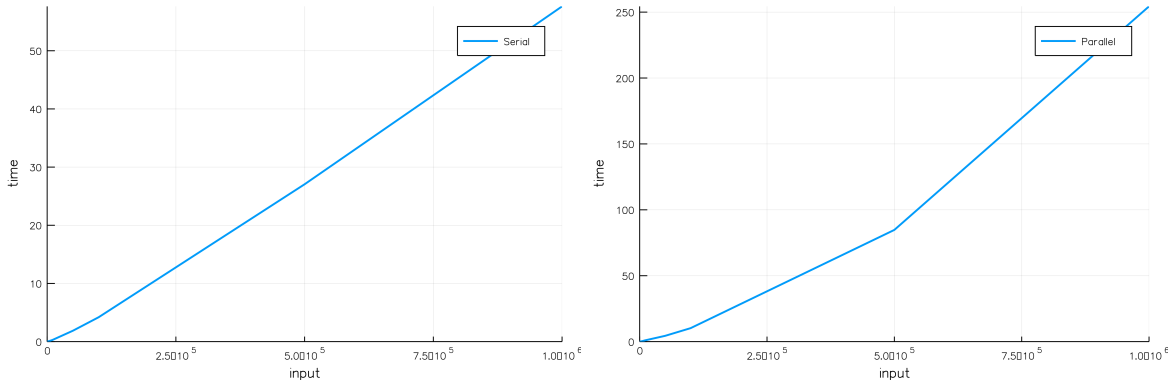


Figure 5: Compared execution time of function box

Execution time on Tesla

```
input =input =[1,10,20,50,10^2,5*10^2,2*10^3]
function timeFstruct(f::Function,pf::Function,model,input)
  t=Array{Float64}(length(input))
  pt=Array{Float64}(length(input))
  for i in range(1,length(input))
    structo=addn2D(input[i],model)
    pstructo=pStruct(structo.body)
    f(structo)
    pf(pstructo)
    t[i]=@elapsed f(structo)
    pt[i]=@elapsed pf(pstructo)
  end
  return t,pt
end

y,yp=timeFstruct(box,pbox,square,input)

p=plot(y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),ylims=(0,maximum(y)+0.5),
label=["Serial"],lw=2)
pp=plot(yp,xaxis="input",yaxis="time",xlims=(0,length(input)+1),ylims=(0,maximum(y)+0.5),
label=["Parallel"],lw=2)
```



(a) Serial (b) Parallel

Figure 6: Execution time of function box on Tesla

Compare

```
yc=[y,yp]
pc=plot(y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),ylims=(0,maximum(y)+0.5),
label=["Serial" "Parallel"],lw=2)
```

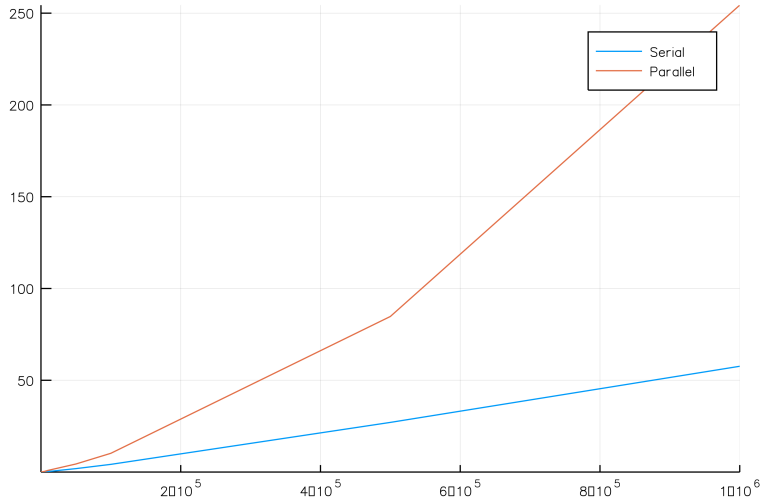


Figure 7: Compared Execution time of function box on Tesla

2.4 traversal

2.4.1 Conversion

Python

```
def traversal(CTM, stack, obj, scene=[]):
  for i in range(len(obj)):
    if isinstance(obj[i],Model):
      scene += [larApply(CTM)(obj[i])]
    elif(isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (len(obj[i]==2 orlen(obj[i])==3):
      scene += [larApply(CTM)(obj[i])]
    elif isinstance(obj[i],Mat):
      CTM = scipy.dot(CTM, obj[i])
    elif isinstance(obj[i],Struct):
        stack.append(CTM)
        traversal(CTM, stack, obj[i], scene)
        CTM = stack.pop()
  return scene
```

Julia

```
function traversal(CTM,stack,obj,scene=[])
  for i in range(1,len(obj))
    if isa(obj.body[i],Matrix)
      CTM = CTM * obj.body[i]
    elseif (isa(obj.body[i],Tuple)||isa(obj.body[i],Array))&&
    (length(obj.body[i])==2||length(obj.body[i])==3)
      l=larApply(CTM)(obj.body[i])
      push!(scene,l)
    elseif isa(obj.body[i],Struct)
      push!(stack,CTM)
      traversal(CTM,stack,obj.body[i],scene)
      CTM = pop!(stack)
    end
  end
  return scene
end
```

2.4.2 Parallelization

```
@everywhere function ptraversal(CTM,stack,obj,scene=[])
  @sync for i in range(1,len(obj))
    if isa(obj.body[i],Matrix)
      CTM = CTM*obj.body[i]
    elseif (isa(obj.body[i],Tuple)||isa(obj.body[i],Array))&&
```

```
    (length(obj.body[i])==2||length(obj.body[i])==3)
      l=plarApply(CTM)(obj.body[i])
      push!(scene,l)
    elseif isa(obj.body[i],pStruct)
      push!(stack,CTM)
      ptraversal(CTM,stack,obj.body[i],scene)
      CTM = pop!(stack)
    end
  end
  return scene
end
```

2.4.3 Unit-Test

These tests can be runned after the definition of the Struct type on the section 2.5

Serial Tests

```
square=([[0, 0], [0, 1], [1, 0], [1, 1]], [[0, 1, 2, 3]])
structure=Struct([square])
d=checkStruct(structure.body)

@testset "traversal Tests" begin
   @test length(traversal(eye(d+1),[],structure,[]))==length(structure.body)
   @test typeof(traversal(eye(d+1),[],structure,[]))==Array{Any,1}
end
```

Parallel Tests

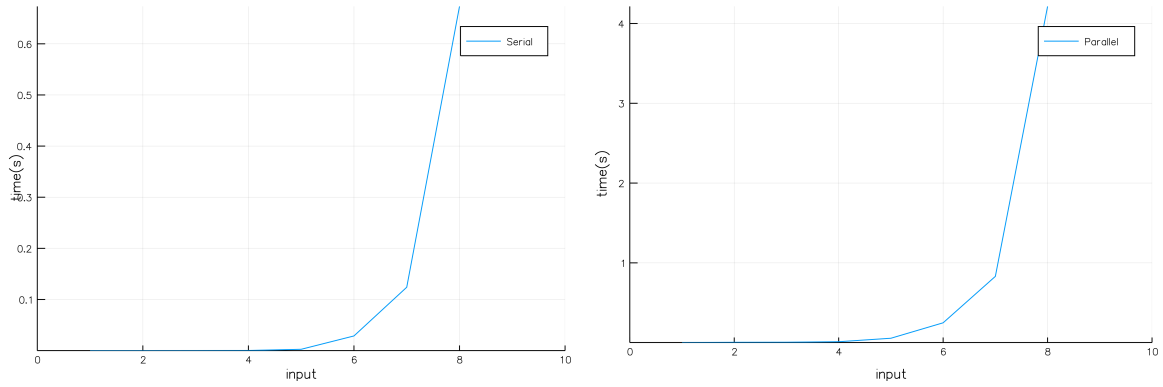
```
square=([[0, 0], [0, 1], [1, 0], [1, 1]], [[0, 1, 2, 3]])
structure=pStruct([square])
d=pcheckStruct(structure.body)

@testset "ptraversal Tests" begin
  @test length(ptraversal(eye(d+1),[],structure,[]))==length(structure.body)
  @test typeof(ptraversal(eye(d+1),[],structure,[]))==Array{Any,1}
end
```

2.4.4 Results

Execution time on PC

```
times=[]
ptimes=[]
input=[]
dim=[]
```

```
for i in range(1,length(l)-1)
  push!(input,Struct([repeat([l[i]],outer=i)...]))
end

append!(dim,checkStruct(input[i].body) for i in range(1,length(input)))

for i in range(1,length(input))
  args=(eye(dim[i]+1),[],input[i],[])
  append!(times,Time(traversal,(args...)))
end
for i in range(1,length(input))
  args=(eye(dim[i]+1),[],input[i],[])
  append!(ptimes,Time(ptraversal,(args...)))
end

plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 8: Execution time of function traversal

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(ptimes)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

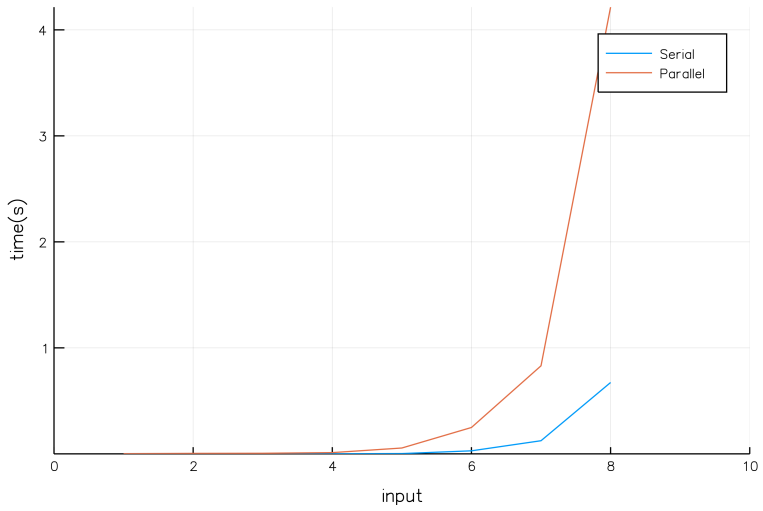
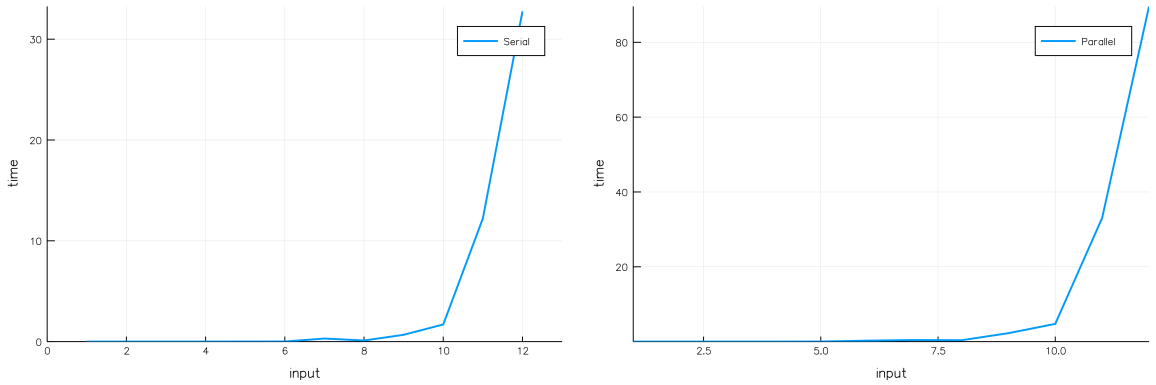


Figure 9: Compared execution time of function traversal

Execution time on Tesla

```
input=[1,10,20,50,10^2,2*10^2,5*10^2,10^3,2*10^3]

function timeTraversal(model,input)
   t=Array{Float64}(length(input))
   pt=Array{Float64}(length(input))
   for i in range(1,length(input))
      structo=addn2D(input[i],model)
      pstructo=pStruct(structo.body)
      dim=checkStruct(structo.body)
      traversal(eye(dim+1),[],structo,[])
      ptraversal(eye(dim+1),[],pstructo,[])
      pt[i]=@elapsed ptraversal(eye(dim+1),[],pstructo,[])
      t[i]=@elapsed traversal(eye(dim+1),[],structo,[])
   end
   return t,pt
end

y,yp=timeTraversal(square,input)
```

```
p=plot(y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),ylims=(0,maximum(y)+0.5),
  label=["Serial"],lw=2)
pp=plot(yp,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5),label=["Parallel"],lw=2)
```



(a) Serial (b) Parallel

Figure 10: Execution time of function traversal on Tesla

Compare

```
yc=[y,yp]
pc=plot(yc,label=["Serial" "Parallel"])
```

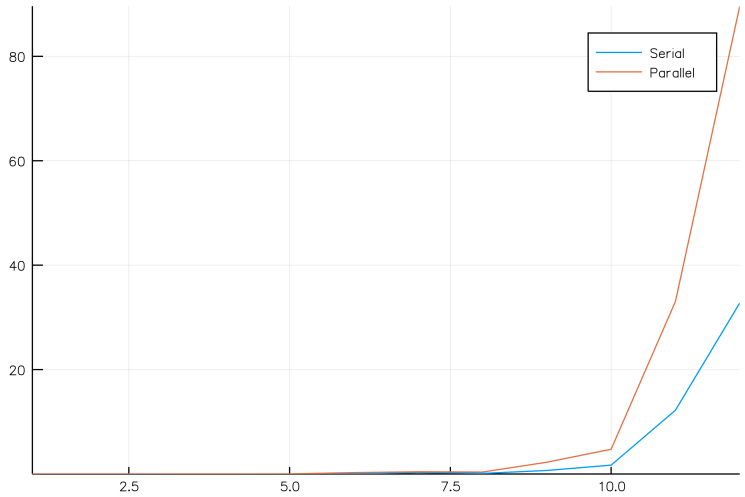


Figure 11: Compared execution time of function traversal on Tesla

2.5 Struct

2.5.1 Conversion

evalStruct

Python

```
def evalStruct(struct):
  dim = checkStruct(struct.body)
  CTM, stack = scipy.identity(dim+1), []
  scene = traversal(CTM, stack, struct, [])
  return scene
```

Julia

```
function evalStruct(self)
  dim = checkStruct(self.body)
  CTM, stack = eye(dim+1), []
  scene = traversal(CTM, stack, self, [])
  return scene
end
```

Python

```
class Struct:
  def __init__(self,data=None,name=None,category=None):
    if data==None or data==[ ]:
      self.body = [ ]
    else:
      #self.body = [item for item in data if item != None]
      self.body = [item for item in data]
      self.box = box(self)
      self.dim = len(self.box[0])
    if name != None:
      self.name = str(name)
    else:
      self.name = str(id(self))
    if category != None:
      self.category = str(category)
    else:
      self.category = "feature"
  def __name__(self):
    return self.name
  def __category__(self):
    return self.category
  def __iter__(self):
    return iter(self.body)
  def __len__(self):
    return len(list(self.body))
  def __getitem__(self,i):
    return list(self.body)[i]}
  def __setitem__(self,i,value):
    self.body[i] = value
  def __print__(self):
    return "<Struct name: %s>" % self.__name__()
  def __repr__(self):
    return "<Struct name: %s>" % self.__name__()
```

```
  def set_name(self,name):
    self.name = str(name)
  def clone(self,i=0):
    from copy import deepcopy
    newObj = deepcopy(self)
    if i != 0: newObj.name = self.name + "_" + str(i)
    return newObj
  def set_category(self,category):
    self.category = str(category)
```

Julia

```
type Struct
  body::Array
  box
  name::AbstractString
  dim
  category::AbstractString
  function Struct()
    self=new([],Nullable{Any},"new",Nullable{Any},"feature")
    self.name=string(object_id(self))
    return self
  end
  function Struct(data::Array)
    self=Struct()
    self.body=data
    self.box=box(self)
    self.dim=length(self.box[1])
    return self
  end
  function Struct(data::Array,name)
    self=Struct()
    self.body=[item for item in data]
    self.box=box(self)
    self.dim=length(self.box[1])
    self.name=string(name)
    return self
  end
  function Struct(data::Array,name,category)
    self=Struct()
    self.body=[item for item in data]
    self.box=box(self)
    self.dim=length(self.box[1])
    self.name=string(name)
    self.category=string(category)
    return self
  end
end
```

```
function name(self::Struct)
  return self.name
end
function category(self::Struct)
  return self.category
end
function len(self::Struct)
  return length(self.body)
end
function getitem(self::Struct,i::Int)
  return self.body[i]
end
function setitem(self::Struct,i,value)
  self.body[i]=value
end
function pprint(self::Struct)
  return "<Struct name: $(self.__name__())"
end
function set_name(self::Struct,name)
  self.name=string(name)
end
function clone(self::Struct,i=0)
    newObj=deepcopy(self)
    if i!=0
      newObj.name="$(self.__name__())_$(string(i))"
    end
    return newObj
end
function set_category(self::Struct,category)
  self.category=string(category)
end
```

2.5.2 Parallelization

```
function pevalStruct(self)
  dim = pcheckStruct(self.body)
  CTM, stack = eye(dim+1), []
  scene = ptraversal(CTM, stack, self, [])
return scene
end



@everywhere type pStruct
  body::Array
  box
  name::AbstractString
  dim
```

```
  category::AbstractString
  function pStruct()
    self=new([],Nullable{Any},"new",Nullable{Any},"feature")
    self.name=string(object_id(self))
    return self
  end
  function pStruct(data::Array)
    self=pStruct()
    self.body=data
    self.box=pbox(self)
    self.dim=length(self.box[1])
    return self
  end
  function pStruct(data::Array,name)
    self=pStruct()
    self.body=[item for item in data]
    self.box=pbox(self)
    self.dim=length(self.box[1])
    self.name=string(name)
    return self
  end
  function pStruct(data::Array,name,category)
    self=pStruct()
    self.body=[item for item in data]
    self.box=pbox(self)
    self.dim=length(self.box[1])
    self.name=string(name)
    self.category=string(category)
    return self
  end
end
function name(self::pStruct)
  return self.name
end
function category(self::pStruct)
  return self.category
end
function len(self::pStruct)
  return length(self.body)
end
function getitem(self::pStruct,i::Int)
  return self.body[i]
end
function setitem(self::pStruct,i,value)
  self.body[i]=value
end
function pprint(self::pStruct)
  return "<Struct name: $(self.__name__())"
end
```

```
function set_name(self::pStruct,name)
  self.name=string(name)
end
function clone(self::pStruct,i=0)
  newObj=deepcopy(self)
  if i!=0
    newObj.name="$(self.__name__())_$(string(i))"
  end
  return newObj
end
function set_category(self::pStruct,category)
  self.category=string(category)
end
```

2.5.3 Unit-Test

Serial Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])

@testset "Struct Tests" begin
        @test Struct([square]).body==[square]
        @test Struct([square]).dim==length(square[1][1])
        @test Struct([square]).box==[[0,0],[1,1]]
end
```

Parallel Tests

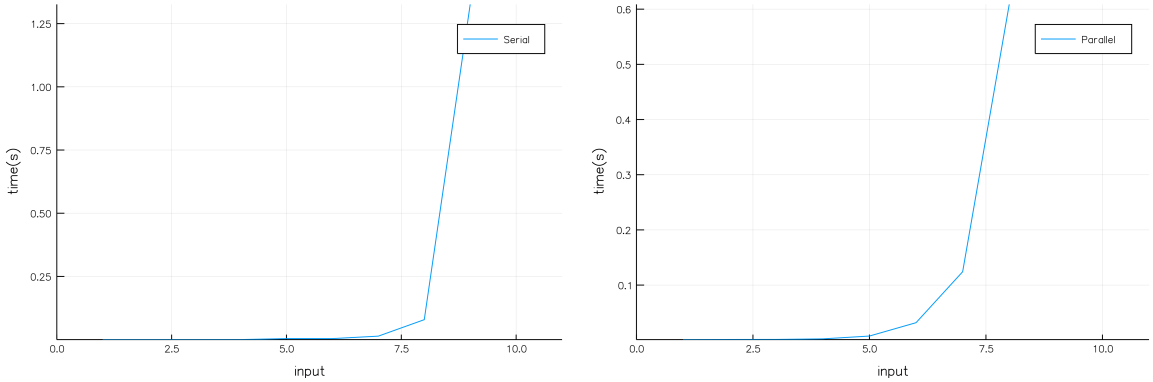
```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])

@testset "pStruct Tests" begin
    @test pStruct([square]).body==[square]
    @test pStruct([square]).dim==length(square[1][1])
    @test pStruct([square]).box==[[0,0],[1,1]]
    @test pStruct([square]).category=="feature"
    @test pStruct([square],"quadrato").name=="quadrato"
end
```

2.5.4 Results

Execution time on PC

```
times=[]
ptimes=[]

append!(times,Time(Struct,[[l[i]]]) for i in range(1,length(l)) )
append!(ptimes,Time(pStruct,[[l[i]]]) for i in range(1,length(l)-1))
```

```
plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 12: Execution time of function Struct

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

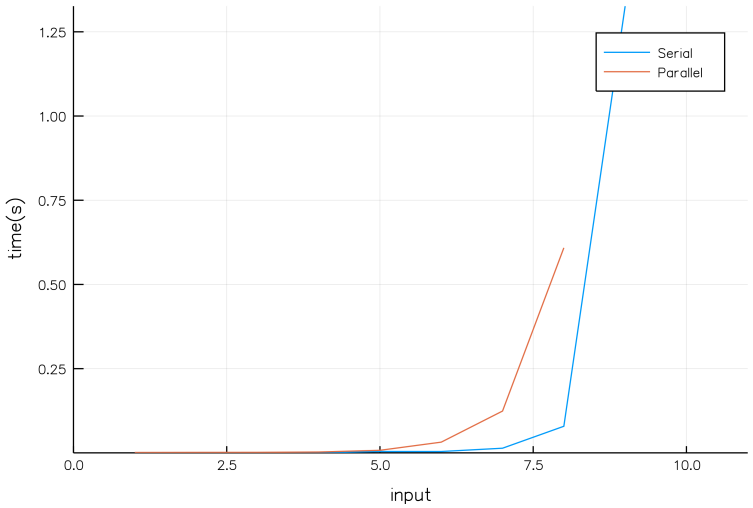
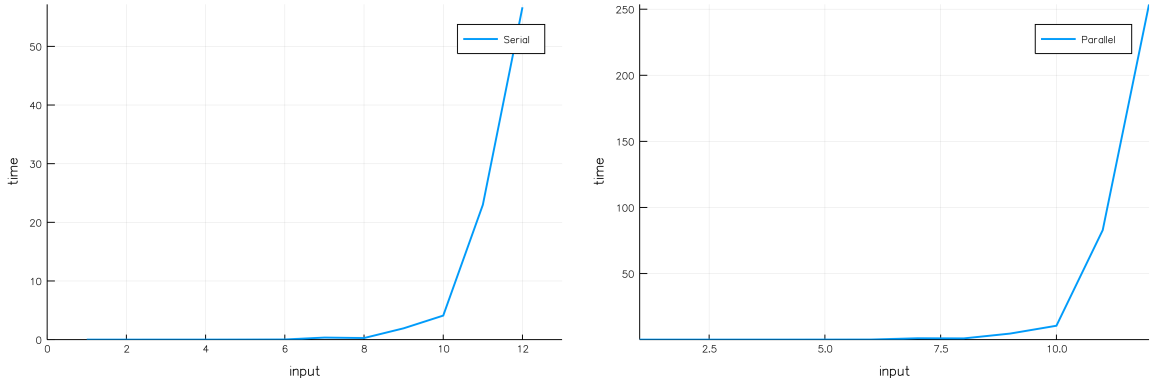


Figure 13: Compared execution time of function Struct

Execution time on Tesla

```
input=[1,10,20,50,10^2,2*10^2,5*10^2,10^3,2*10^3]
function timeStruct(model,input)
  t=Array{Float64}(length(input))
  pt=Array{Float64}(length(input))
  for i in range(1,length(input))
    structo=addn2D(input[i],model)
    Struct(structo.body)
    pStruct(structo.body)
    t[i]=@elapsed  Struct(structo.body)
    pt[i]=@elapsed pStruct(structo.body)
  end
  return t,pt
end

y,yp=timeStruct(square,input)
p=plot(y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5), label=["Serial"],lw=2)

pp=plot(yp,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5),label=["Parallel"],lw=2)
```



(a) Serial (b) Parallel

Figure 14: Execution time of function Struct on Tesla

Compare

```
yc=[y,yp]
pc=plot(yc,label=["Serial" "Parallel"])
```

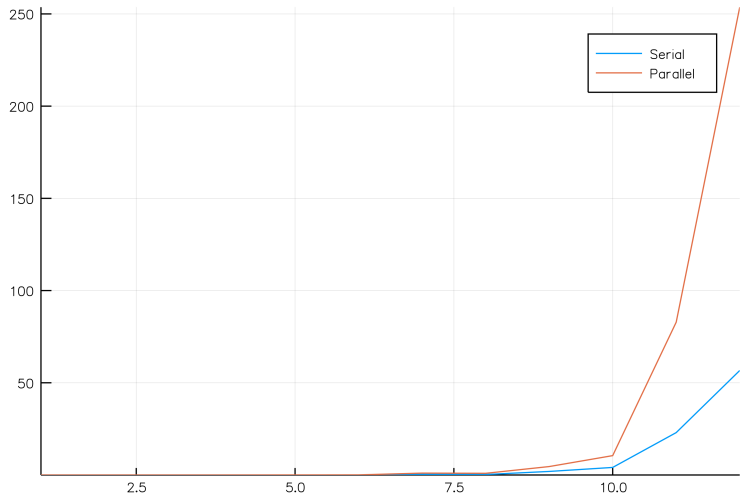


Figure 15: Compared execution time of function Struct on Tesla

2.6 embedTraversal

2.6.1 Conversion

Python

```
def embedTraversal(cloned, obj,n,suffix):
  for i in range(len(obj)):
    if isinstance(obj[i],Model):
      cloned.body += [obj[i]]
    elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (len(obj[i])==2):
      V,EV = obj[i]
      V = [v+n*[0.0] for v in V]
      cloned.body  += [(V,EV)]
    elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (len(obj[i])==3):
      V,FV,EV = obj[i]
      V = [v+n*[0.0] for v in V]
      cloned.body  += [(V,FV,EV)]
    elif isinstance(obj[i],Mat):
      mat = obj[i]
      d,d = mat.shape
      newMat = scipy.identity(d+n*1)
      for h in range(d-1):
        for k in range(d-1):
          newMat[h,k] = mat[h,k]
        newMat[h,d-1+n*1] = mat[h,d-1]
        cloned.body  +=  [newMat.view(Mat)]
    elif isinstance(obj[i],Struct):
      newObj = Struct()
      newObj.box = hstack((obj[i].box, [n*[0],n*[0]]))
      newObj.name = obj[i].name+suffix
      newObj.category = obj[i].category
      cloned.body +=[embedTraversal(newObj, obj[i], n, suffix)]
    return cloned
```

Julia

```
function embedTraversal(cloned,obj,n,suffix)
  for i in range(1,len(obj))
    if isa(obj.body[i],Matrix)
      mat=obj.body[i]
      d,d=size(mat)
      newMat=eye(d+n*1)
      for h in range(1,d-1)
        for k in range(1,d-1)
          newMat[h,k]=mat[h,k]
        end
        newMat[h,d-1+n*1]=mat[h,d-1]
      end
      append!(cloned.body,newMat)
```

```
    elseif (isa(obj.body[i],Tuple) ||isa(obj.body[i],Array))&&length(obj.body[i])==3
      V,FV,EV=obj.body[i]
      dimadd=fill([0.0],n)
      for k in dimadd
        for v in V
          append!(v,k)
        end
      end
      append!(cloned.body,[(V,FV,EV)])
    elseif (isa(obj.body[i],Tuple) ||isa(obj.body[i],Array))&&length(obj.body[i])==2
      V,EV=deepcopy(obj.body[i])
      dimadd=fill([0.0],n)
      for k in dimadd
        for v in V
          append!(v,k)
        end
      end
      append!(cloned.body,[(V,EV)])
    elseif isa(obj.body[i],Struct)
      newObj=Struct()
      newObj.box=hcat((obj.body[i].box,[fill([0],n),fill([0],n)]))
      newObj.category=obj.body[i].category
      append!(cloned.body,embedTraversal(newObj,obj.body[i],n,suffix))
    end
  end
  return cloned
end
```

2.6.2 Parallelization

```
@everywhere function pembedTraversal(cloned,obj,n,suffix)
  for i in range(1,len(obj))
    if (isa(obj.body[i],Matrix) || isa(obj.body[i],SharedArray))
      mat=obj.body[i]
      d,d=size(mat)
      newMat=eye(d+n*1)
      @sync begin
        for h in range(1,d-1)
          @async begin
            for k in range(1,d-1)
              newMat[h,k]=mat[h,k]
            end
          end
          newMat[h,d-1+n*1]=mat[h,d-1]
        end
      end
      append!(cloned.body,newMat)
    elseif (isa(obj.body[i],Tuple) ||isa(obj.body[i],Array))&& length(obj.body[i])==2
      V,EV=deepcopy(obj.body[i])
```

```
      dimadd=fill([0.0],n)
      @sync begin
        for k in dimadd
          @async begin
            for v in V
              append!(v,k)
            end
          end
        end
      end
      append!(cloned.body,[(V,EV)])
    elseif (isa(obj.body[i],Tuple) ||isa(obj.body[i],Array))&&length(obj.body[i])==3
      V,FV,EV=obj.body[i]
      dimadd=fill([0.0],n)
      @sync begin
        for k in dimadd
          @async begin
            for v in V
              append!(v,k)
            end
          end
        end
      end
      append!(cloned.body,[(V,FV,EV)])
    elseif isa(obj.body[i],pStruct)
      newObj=pStruct()
      @async begin
        newObj.box=hcat((obj.body[i].box,[fill([0],n),fill([0],n)]))
        newObj.category=obj.body[i].category
        append!(cloned.body,embedTraversal(newObj,obj.body[i],n,suffix))
      end
    end
  end
  return cloned
end
```

2.6.3 Unit-Test

Serial Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
x=Struct([square])

@testset "embedTraversal Tests" begin
  @test length(embedTraversal(deepcopy(x),deepcopy(x),1,"New").body[2][1][1])==
  length(x.body[1][1][1])+1
  #in this case n=1, but generally:
  # length(length(embedTraversal(x,x,1,"New")=length(x.body[1][1][1])+n
  @test length(embedTraversal(deepcopy(x),deepcopy(x),3,"New").body[2][1][1])==
```

```
  length(x.body[1][1][1])+3
  @test typeof(embedTraversal(deepcopy(x),deepcopy(x),1,"New"))==Struct
end
```

Parallel Tests

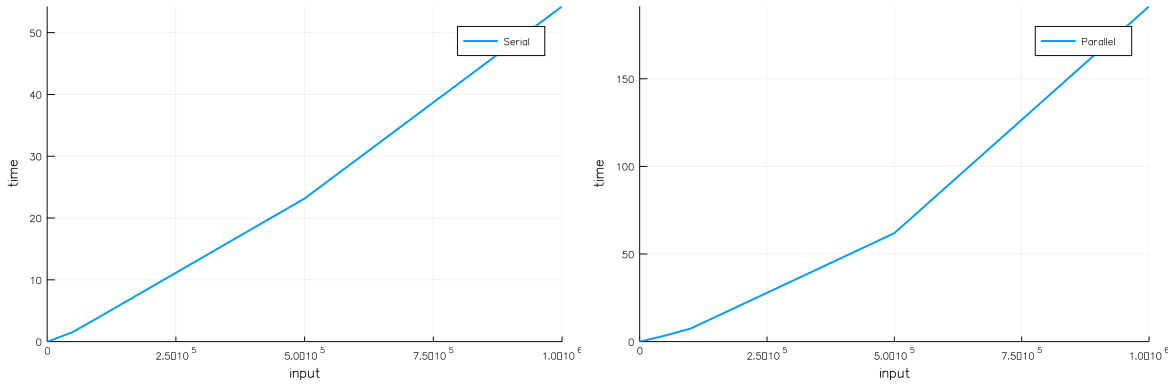
```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
x=pStruct([square])

@testset "pembedTraversal Tests" begin
  @test length(pembedTraversal(deepcopy(x),deepcopy(x),1,"New").body[2][1][1])==
  length(x.body[1][1][1])+1
  #in this case n=1, but generally:
  #length(length(embedTraversal(x,x,1,"New")=length(x.body[1][1][1])+n
  @test length(pembedTraversal(deepcopy(x),deepcopy(x),3,"New").body[2][1][1])==
  length(x.body[1][1][1])+3
  @test typeof(pembedTraversal(deepcopy(x),deepcopy(x),1,"New"))==pStruct
end
```

2.6.4 Results

Execution time on Tesla

```
input=[1,10,20,50,10^2,2*10^2,5*10^2,10^3,2*10^3]
function timeEmbedTraversal(model,input)
  t=Array{Float64}(length(input))
  pt=Array{Float64}(length(input))
  for i in range(1,length(input))
    structo=addn2D(input[i],model)
    pstructo=pStruct(structo.body)
    cloned=Struct()
    cloned.box=hcat((structo.box,[fill([0],10),fill([0],10)]))
    cloned.name=string(object_id(cloned))
    cloned.category=structo.category
    cloned.dim=structo.dim+10
    pcloned=pStruct()
    pcloned.box=hcat((pstructo.box,[fill([0],10),fill([0],10)]))
    pcloned.name=string(object_id(pcloned))
    pcloned.category=pstructo.category
    pcloned.dim=pstructo.dim+10
    embedTraversal(cloned,structo,10,"New")
    pembedTraversal(pcloned,pstructo,10,"New")
    t[i]=@elapsed embedTraversal(cloned,structo,10,"New")
    pt[i]=@elapsed pembedTraversal(pcloned,pstructo,10,"New")
  end
  return t,pt
end
y,yp=timeEmbedTraversal(square,input)
```

```
p=plot(input,y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5), label=["Serial"],lw=2)
pp=plot(input,yp,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5),label=["Parallel"],lw=2)
```



(a) Serial (b) Parallel

Figure 16: Execution time of function embedTraversal on Tesla

Compare

```
yc=[y,yp]
pc=plot(input,yc,label=["Serial" "Parallel"])
```

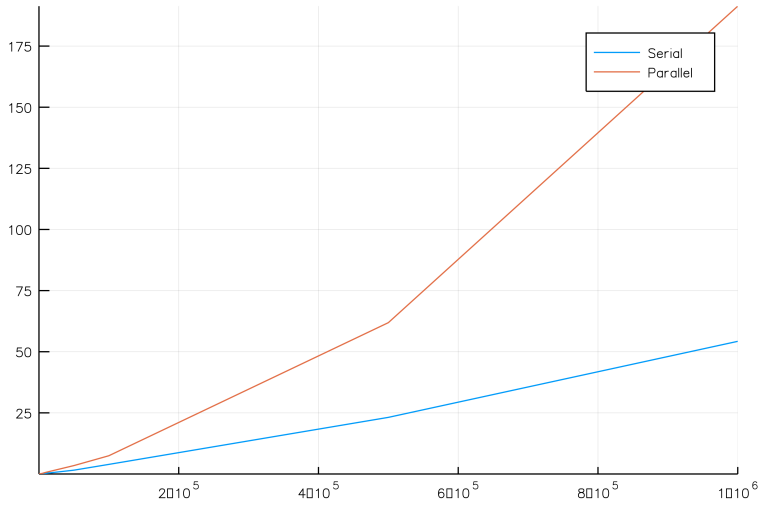


Figure 17: Compared execution time of function embedtraversal on Tesla

2.7 embedStruct

2.7.1 Conversion

Python

```
def embedStruct(n):
  def embedStruct0(struct,suffix="New"):
    if n==0:
      return struct, len(struct.box[0])
    cloned = Struct()
    cloned.box = hstack((struct.box, [n*[0],n*[0]])).tolist()
    cloned.name = str(id(cloned))  #struct.name+suffix
    cloned.category = struct.category
    cloned.dim = struct.dim + n
    cloned = embedTraversal(cloned,struct,n,suffix)
    return cloned
  return embedStruct0
```

Julia

```
function embedStruct(n)
  function embedStruct0(self,suffix="New")
    if n==0
      return self, length(self.box[1])
    end
    cloned=Struct()
    cloned.box=hcat((self.box,[fill([0],n),fill([0],n)]))
    cloned.name=string(object_id(cloned))
    cloned.category=self.category
    cloned.dim=self.dim+n
    cloned=embedTraversal(cloned,self,n,suffix)
    return cloned
 end
 return embedStruct0
end
```

2.7.2 Parallelization

```
@everywhere function pembedStruct(n)
  function pembedStruct0(self,suffix="New")
    if n==0
      return self, length(self.box[1])
    end
    cloned=pStruct()
```

```
    cloned.box=hcat((self.box,[fill([0],n),fill([0],n)]))
    cloned.name=string(object_id(cloned))
    cloned.category=self.category
    cloned.dim=self.dim+n
    cloned=pembedTraversal(cloned,self,n,suffix)
    return cloned
  end
  return pembedStruct0
end
```

2.7.3 Unit-Test

Serial Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
x=Struct([square])

@testset "embedStruct Tests" begin
  @test length(embedStruct(1)(x).body[1][1][1])==length(x.body[1][1][1])+1
  #in this case n = 1, but generally:
  #length(embedStruct(n)(x).body[1][1][1])=length(x.body[1][1][1])+n
  @test length(embedStruct(3)(x).body[1][1][1])==length(x.body[1][1][1])+3
  @test typeof(embedStruct(1)(x))==Struct
end
```

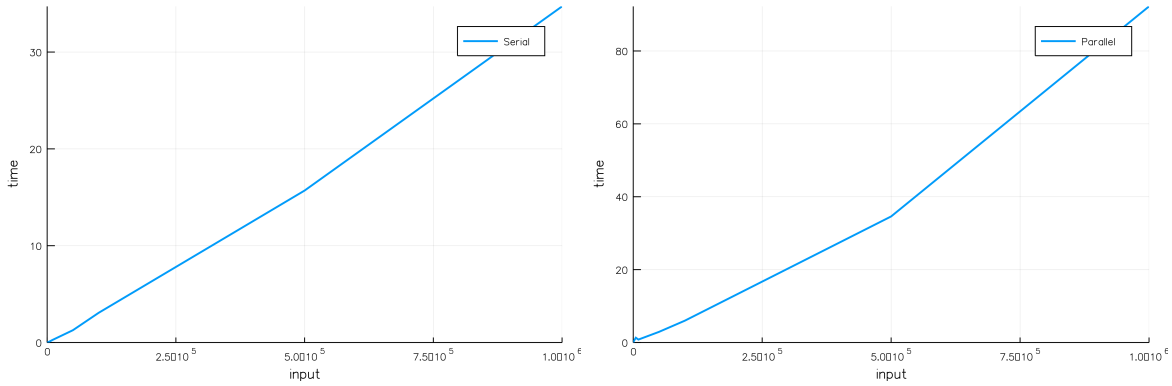
Parallel Tests

```
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
x=pStruct([square])

@testset "pembedStruct Tests" begin
  @test length(pembedStruct(1)(x).body[1][1][1])==length(x.body[1][1][1])+1
  #in this case n = 1, but generally:
  #length(embedStruct(n)(x).body[1][1][1])=length(x.body[1][1][1])+n
  @test length(pembedStruct(3)(x).body[1][1][1])==length(x.body[1][1][1])+3
  @test typeof(pembedStruct(1)(x))==pStruct
end
```

2.7.4 Results

Execution time on Tesla

```
input=[1,10,20,50,10^2,2*10^2,5*10^2,10^3,2*10^3]

function timeEmbedStruct(n,model,input)
  t=Array{Float64}(length(input))
  pt=Array{Float64}(length(input))
  for i in range(1,length(input))
    structo=addn2D(input[i],model)
    pstructo=pStruct(structo.body)
    embedStruct(n)(structo)
    pembedStruct(n)(pstructo)
    t[i]=@elapsed embedStruct(n)(structo)
    pt[i]=@elapsed pembedStruct(n)(pstructo)
  end
  return t,pt
end
y,yp=timeEmbedStruct(10,square,input)

p=plot(input,y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5), label=["Serial"],lw=2)
pp=plot(input,yp,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5),label=["Parallel"],lw=2)
```



(a) Serial (b) Parallel

Figure 18: Execution time of function embedStruct on Tesla

Compare

```
yc=[y,yp]
pc=plot(input,yc,label=["Serial" "Parallel"])
```

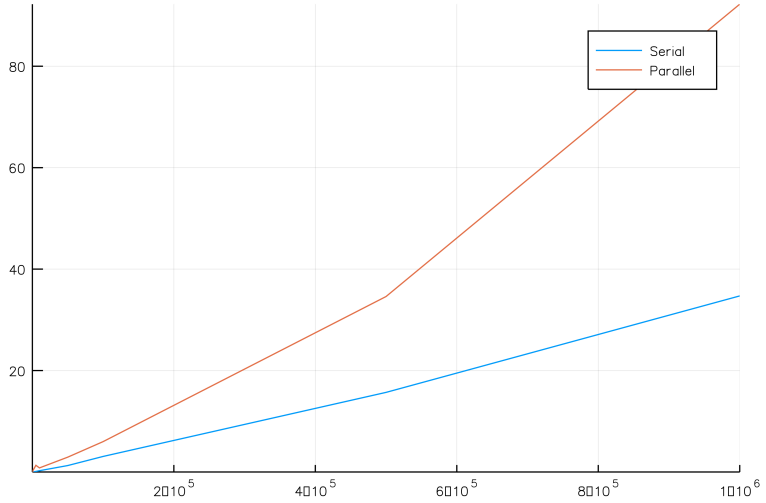


Figure 19: Compared execution time of function embedtraversal on Tesla

2.8 removeDups

2.8.1 Conversion

Python

```
def removeDups (CW):
  CW = list(set(AA(tuple)(CW)))
  CWs = list(set(AA(tuple)(AA(sorted)(CW))))
  no_duplicates = defaultdict(list)
  for f in CWs: no_duplicates[f] = []
    for f in CW:
      no_duplicates[tuple(sorted(f))] += [f]
    CW = [f[0] for f in no_duplicates.values()]
  return CW
```

Julia

```
function removeDups(CW)
  CW=collect(Set(CW))
  CWs=collect(map(sort,CW))
  no_duplicates=Dict()
  for f in CWs
    no_duplicates[f] = []
  end
  for f in CW
    no_duplicates[sort(f)]=[f]
  end
  CW=[f[1] for f in values(no_duplicates)]
  return CW
end
```

2.8.2 Parallelization

```
function premoveDups(CW)
  CW=collect(Set(CW))
  CWs=collect(@sync pmap(sort,CW))
  no_duplicates=Dict()
  @parallel for f in CWs
    no_duplicates[f] = []
  end
  @parallel for f in CW
    no_duplicates[sort(f)]=[f]
  end
  @parallel for f in values(no_duplicates)
    append!(CW,f[1])
  end
  return CW
end
```

2.8.3 Unit-Test

Serial Tests

```
CW1=[[0,0],[0,1],[1,0],[0,0],[1,1],[0,1],[1,1]]
CW2=[[0,0,0],[1,0,0],[0,1,0],[0,0,1],[0,0,0],[1,1,0],[0,1,1],[1,1,0],[1,0,1],[1,1,1]]

@testset "removeDups Tests" begin
  @testset "removeDups 3D" begin
    @test length(removeDups(CW1))<= length(CW1)
      @test typeof(removeDups(CW1))==Array{Array{Int64,1},1}
   end
   @testset "removeDups 2D" begin
      @test length(removeDups(CW2))<= length(CW2)
      @test typeof(removeDups(CW2))==Array{Array{Int64,1},1}
   end
end
```

Palallel Tests

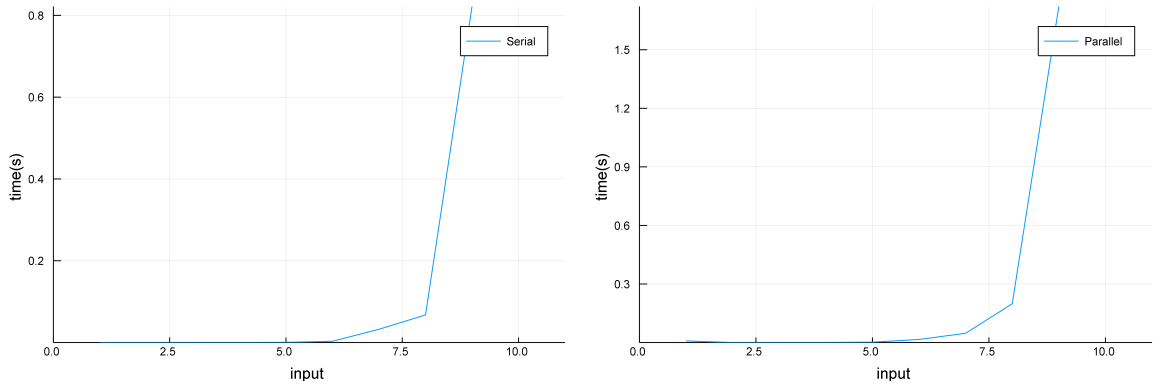
```
CW1=[[0,0],[0,1],[1,0],[0,0],[1,1],[0,1],[1,1]]
CW2=[[0,0,0],[1,0,0],[0,1,0],[0,0,1],[0,0,0],[1,1,0],[0,1,1],[1,1,0],[1,0,1],[1,1,1]]

@testset "premoveDups Tests" begin
   @testset "premoveDups 3D" begin
      @test length(premoveDups(CW1))<= length(CW1)
      @test typeof(premoveDups(CW1))==Array{Array{Int64,1},1}
   end
   @testset "premoveDups 2D" begin
      @test length(premoveDups(CW2))<= length(CW2)
      @test typeof(premoveDups(CW2))==Array{Array{Int64,1},1}
    end
end
```

2.8.4 Results

Execution time on PC

```
input=[]
times=[]
ptimes=[]

append!(input,l[i][1] for i in range(1,length(l)))
append!(times,Time(removeDups,[input[i]]) for i in range(1,length(input)))
append!(ptimes,Time(premoveDups,[input[i]]) for i in range(1,length(input)))
```

```
plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 20: Execution time of function removeDups

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

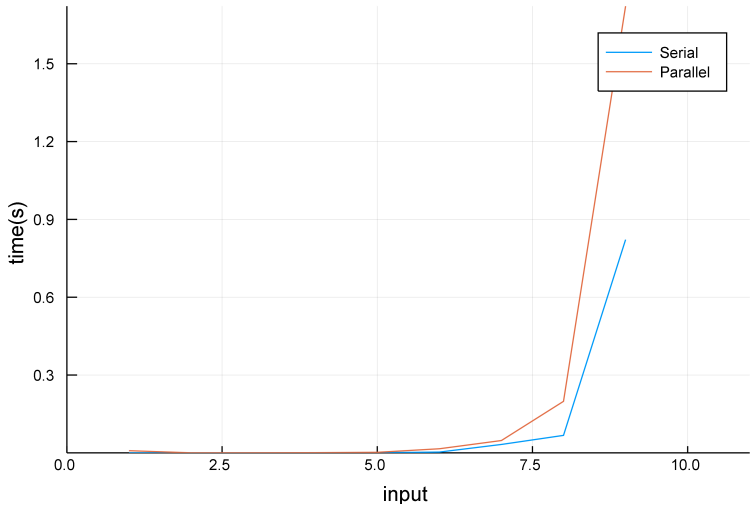


Figure 21: Compared execution time of function removeDups

2.9 struct2lar

2.9.1 Conversion

fixedPrec

Python

```
def fixedPrec(PRECISION):
  def fixedPrec0(value):
    out = round(value*10**(PRECISION))/10**(PRECISION)
    if out == -0.0: out = 0.0
      return str(out)
    return fixedPrec0
```

Julia

```
@everywhere function fixedPrec(PRECISION)
  function fixedPrec0(value)
    out=round.(value,PRECISION)
    if out==-0.0
      out=0.0
    end
    return string(out)
  end
  return fixedPrec0
end
```

vcode

Python

```
def vcode (PRECISION=4):
  def vcode0 (vect):
    return prepKey(AA(fixedPrec(PRECISION))(vect))
  return vcode0
```

Julia

```
@everywhere function vcode(PRECISION=4)
  function vcode0(vect)
    return fixedPrec(PRECISION)(vect)
  end
  return vcode0
end
```

Python

```
def struct2lar(structure,metric=ID):
  listOfModels = evalStruct(structure)
  vertDict = dict()
  index,defaultValue,CW,W,FW = -1,-1,[],[],[]
  for model in listOfModels:
    if isinstance(model,Model):
      V= model.verts
      FV=model.cells
    elif (isinstance(model,tuple) or isinstance(model,list)):
      if len(model)==2:
        V,FV=model
        dim=len(model)
      elif len(model)==3:
        V,FV,EV,dim=model,len(model)
        for k,incell in enumerate(EV):
          outcell = []
          for v in incell:
            key = vcode(4)(V[v])
            if vertDict.get(key,defaultValue) == defaultValue:
              index += 1
```

```
            vertDict[key] = index
            outcell += [index]
            W += [eval(key)]
          else:
            outcell += [vertDict[key]]
        FW += [outcell]
  for k,incell in enumerate(FV):
    outcell = []
    for v in incell:
      key = vcode(4)(V[v])
      if vertDict.get(key,defaultValue) == defaultValue:
        index += 1
        vertDict[key] = index
        outcell += [index]
        W += [eval(key)]
      else:
        outcell += [vertDict[key]]
    CW += [outcell]
  if ((isinstance(model,tuple) or isinstance(model,list))and len(model)==2) or
  ((isinstance(model,Model)and model.n==2)):
    if len(CW[0])==2:
      CW = list(set(AA(tuple)(AA(sorted)(CW))))
    else: CW = removeDups(CW)
    return metric(W),CW
  if ((isinstance(model,tuple) or isinstance(model,list))and len(model)==3):
    FW = list(set(AA(tuple)(AA(sorted)(FW))))
    CW = removeDups(CW)
    return metric(W),CW,FW
```

Julia

```
function struct2lar(structure)
  listOfModels=evalStruct(structure)
  vertDict= Dict()
  index,defaultValue,CW,W,FW = -1,-1,[],[],[]
  for model in listOfModels
    if  length(model)==2
      V,FV=model
    elseif lenght(model)==3
      V,FV,EV=model
    end
    for (k,incell) in enumerate(FV)
      outcell=[ ]
      for v in incell
        key=vcode(4)(V[v+1])
        if get(vertDict,key,defaultValue)==defaultValue
          index =index+1
          vertDict[key]=index
          append!(outcell,index)
```

```
          append!(W,[eval(parse(key))])
        else
          append!(outcell,vertDict[key])
        end
      end
      append!(CW,[outcell])
    end
    if length(model)==3
      for (k,incell) in enumerate(FV)
        outcell=[]
        for v in incell
          key=vcode(4)(V[v+1])
          if get(vertDict,key,defaultValue)==defaultValue
            index =index+1
            vertDict[key]=index
            append!(outcell,[index])
            append!(W,[eval(parse(key))])
          else
            append!(outcell,vertDict[key])
          end
        end
        append!(FW,[outcell])
      end
    end
  end
  if length(listOfModels[end])==2
    if length(CW[1])==2
      CW=map(Tuple,map(sort,CW))
    else
      CW=removeDups(CW)
    end
    return W,CW
  end
  if length(listOfModels[end])==3
    FW=map(Tuple,map(sort,FW))
    CW=removeDups(CW)
    return W,CW,FW
    end
end
```

2.9.2 Parallelization

```
@everywhere function pstruct2lar(structure)
  listOfModels=pevalStruct(structure)
  vertDict= Dict()
  index,defaultValue,CW,W,FW = -1,-1,[],[],[]
  for model in listOfModels
```

```
  if  length(model)==2
    V,FV=model
  elseif lenght(model)==3
    V,FV,EV=model
  end
  @sync begin
    for (k,incell) in enumerate(FV)
      outcell=[]
      @async begin
        for v in incell
          key=vcode(4)(V[v+1])
          if get(vertDict,key,defaultValue)==defaultValue
            index =index+1
            vertDict[key]=index
            append!(outcell,index)
            append!(W,[eval(parse(key))])
          else
            append!(outcell,vertDict[key])
          end
        end
      end
      append!(CW,[outcell])
    end
  end
  if length(model)==3
    @sync begin
      for (k,incell) in enumerate(FV)
        outcell=[]
        @async begin
          for v in incell
            key=vcode(4)(V[v+1])
            if get(vertDict,key,defaultValue)==defaultValue
              index =index+1
              vertDict[key]=index
              append!(outcell,[index])
              append!(W,[eval(parse(key))])
            else
              append!(outcell,vertDict[key])
            end
          end
        end
        append!(FW,[outcell])
      end
    end
  end
end
if length(listOfModels[end])==2
  if length(CW[1])==2
    CW=pmap(Tuple,pmap(sort,CW))
```

```
    else
      CW=premoveDups(CW)
    end
    return W,CW
  end
  if length(listOfModels[end])==3
    FW=pmap(Tuple,pmap(sort,FW))
    CW=premoveDups(CW)
    return W,CW,FW
  end
end
```

2.9.3 Unit-Test

Serial Tests

```
@testset "struct2lar Tests" begin
square=([[0, 0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
table=larApply(t(-0.5,-0.5))(square)
structure=Struct([repeat([table,r(pi/2)],outer=2)...])
  @testset "struct2lar 2D" begin
    @test typeof(struct2lar(structure))==Tuple{Array{Any,1},Array{Array{Any,1},1}}
    @test length(struct2lar(structure)[1][1])==2
  end

  BV=[[0,1,2,3],[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7]]
  V=[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]
  block=[V,BV]
  structure=Struct(repeat([block,t(1,0,0)],outer=2))
  @testset "struct2lar 3D" begin
    @test typeof(struct2lar(structure))==Tuple{Array{Any,1},Array{Array{Any,1},1}}
    @test length(struct2lar(structure)[1][1])==3
  end
end
```

Palallel Tests

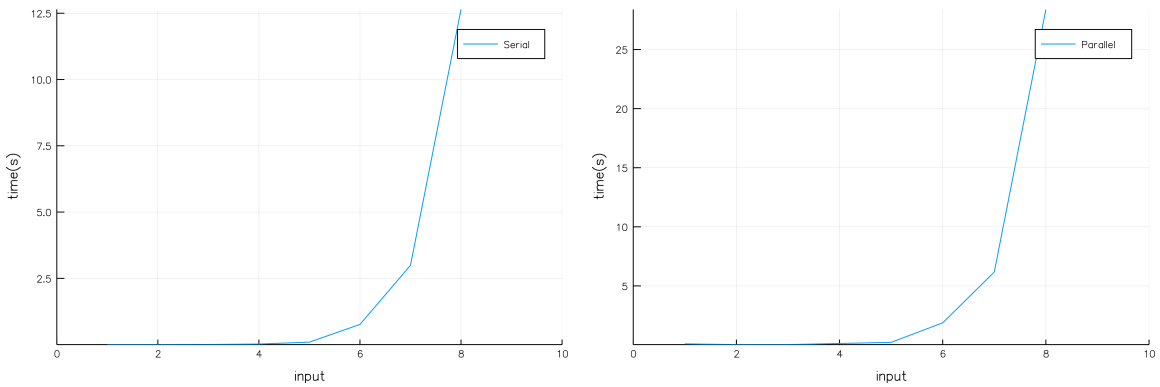
```
@testset "pstruct2lar Tests" begin
square=([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
table=plarApply(t(-0.5,-0.5))(square)
structure=pStruct([repeat([table,r(pi/2)],outer=2)...])

  @testset "pstruct2lar 2D" begin
    @test typeof(pstruct2lar(structure))==Tuple{Array{Any,1},Array{Any,1}}
    @test length(pstruct2lar(structure)[1][1])==2
  end
```

```
  BV=[[0,1,2,3],[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7]]
  V=[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]
  block=[V,BV]
  structure=pStruct(repeat([block,t(1,0,0)],outer=2));

  @testset "pstruct2lar 3D" begin
    @test typeof(pstruct2lar(structure))==Tuple{Array{Any,1},Array{Any,1}}
    @test length(pstruct2lar(structure)[1][1])==3
   end
end
```

2.9.4 Results

Execution time on PC

```
times=[]
ptimes=[]
input=[]

for i in range(1,length(l)-1)
  push!(input,Struct([repeat([l[i]],outer=i)...]))
end
append!(times,Time(struct2lar,[input[i]]) for i in range(1,length(input)))
append!(ptimes,Time(pstruct2lar,[input[i]]) for i in range(1,length(input)))


plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 22: Execution time of function struct2lar

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

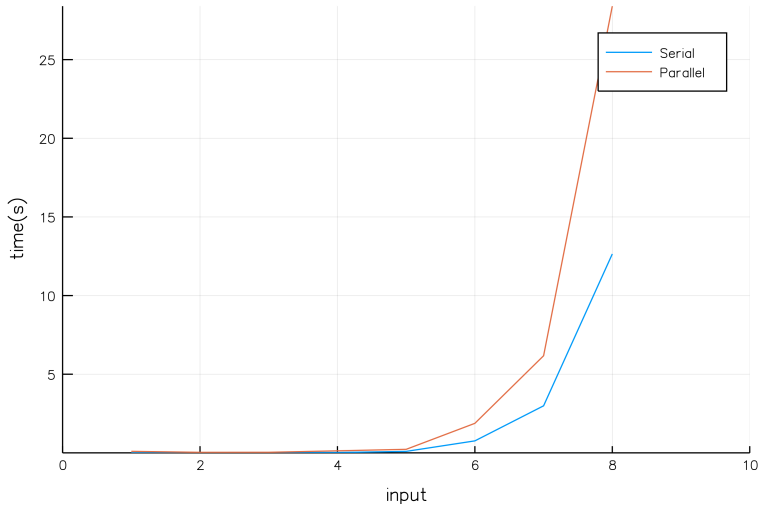
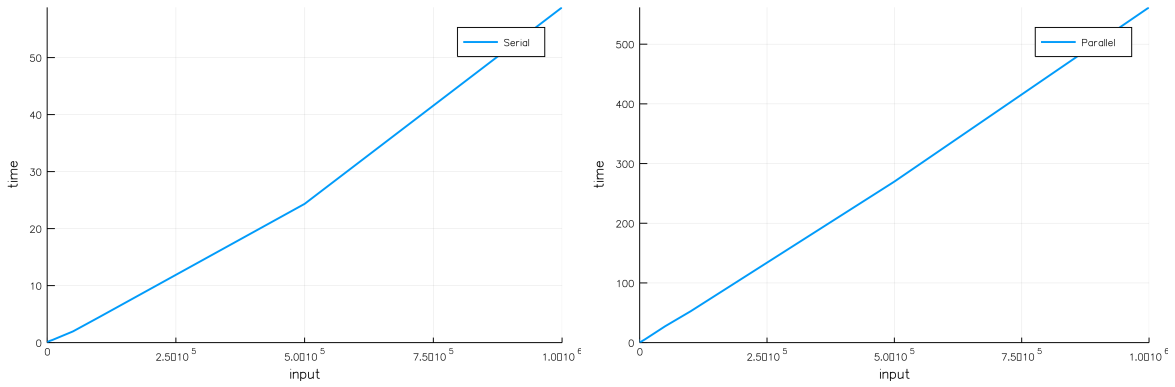


Figure 23: Compared execution time of function struct2lar

Execution time on Tesla

```
input=[1,10,20,50,10^2,2*10^2,5*10^2,10^3,2*10^3]
y,yp=timeFstruct(struct2lar,pstruct2lar,square,input)
```

```
p=plot(input,y,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
     ylims=(0,maximum(y)+0.5), label=["Serial"],lw=2)
pp=plot(input,yp,xaxis="input",yaxis="time",xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5),label=["Parallel"],lw=2)
```



(a) Serial (b) Parallel

Figure 24: Execution Time of function struct2lar on Tesla

Compare

```
yc=[y,yp]
pc=plot(input,yc,label=["Serial" "Parallel"])
```

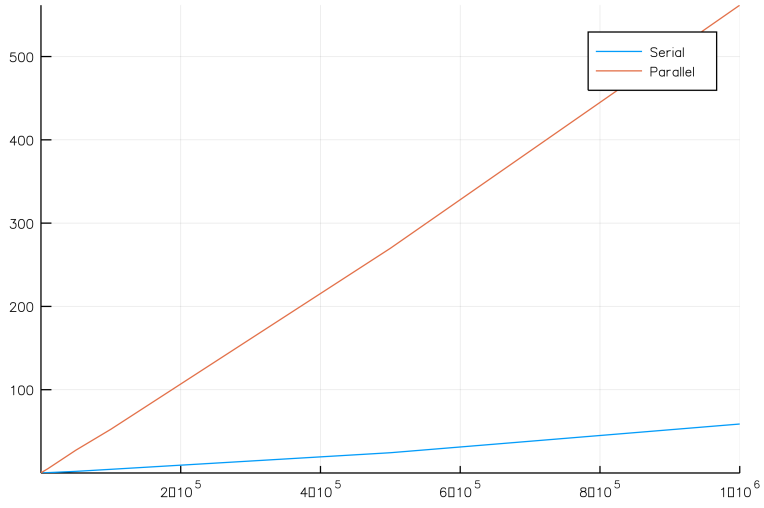


Figure 25: Compared execution time of function struct2lar on Tesla

2.10 larRemoveVertices

2.10.1 Conversion

Python

```
def larRemoveVertices(V,FV):
  vertDict = dict()
  index,defaultValue,FW,W = -1,-1,[],[]
  for k,incell in enumerate(FV):
    outcell = []
    for v in incell:
      key = vcode(4)(V[v])
      if vertDict.get(key,defaultValue) == defaultValue:
        index += 1
        vertDict[key] = index
        outcell += [index]
        W += [eval(key)]
      else:
        outcell += [vertDict[key]]
    FW += [outcell]
  return W,FW
```

Julia

```
function larRemoveVertices(V,FV)
  vertDict= Dict()
  index,defaultValue,CW,W,FW = -1,-1,[],[],[]
  for (k,incell) in enumerate(FV)
    outcell=[]
    for v in incell
      key=vcode(4)(V[v+1])
      if get(vertDict,key,defaultValue)==defaultValue
        index =index+1
        vertDict[key]=index
        append!(outcell,index)
        append!(W,[eval(parse(key))])
      else
        append!(outcell,vertDict[key])
      end
    end
    append!(FW,[outcell])
  end
  return W,FW
end
```

2.10.2 Parallelization

```
@everywhere function plarRemoveVertices(V,FV)
  vertDict= Dict()
  index,defaultValue,CW,W,FW = -1,-1,[],[],[]
  @async begin
    for (k,incell) in enumerate(FV)
      outcell=[]
      @sync begin
```

```
        for v in incell
          key=vcode(4)(V[v+1])
          if get(vertDict,key,defaultValue)==defaultValue
            index =index+1
            vertDict[key]=index
            append!(outcell,index)
            append!(W,[eval(parse(key))])
          else
            append!(outcell,vertDict[key])
          end
        end
      end
      append!(FW,[outcell])
    end
  end
  return W,FW
end
```

2.10.3 Unit-Test

Serial Tests

```
V=[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1, 1, 0], [1, 1, 1]]
FV=[[0,1,2,3],[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7]]

@testset "larRemoveVertices Tests" begin
  @test typeof(larRemoveVertices(V,FV))==Tuple{Array{Any,1},Array{Any,1}}
  @test length(larRemoveVertices(V,FV)[1])<= length(V)
end
```

Palallel Tests

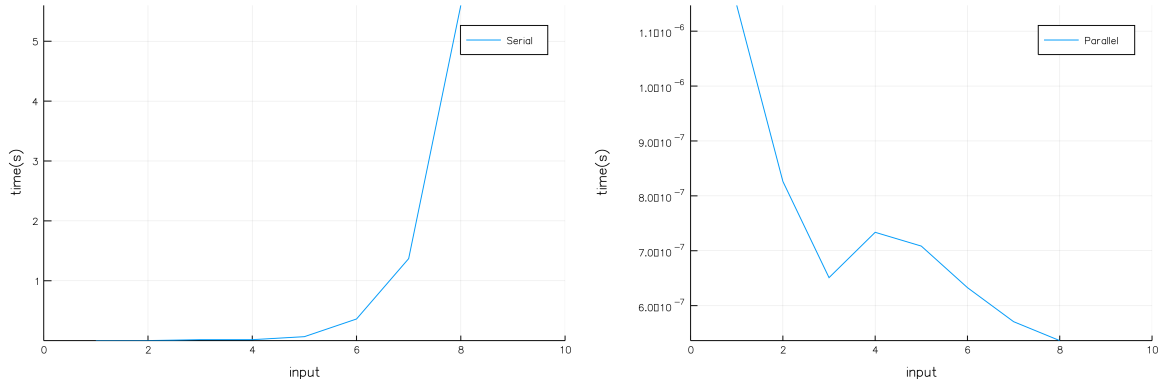
```
V=[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1, 1, 0], [1, 1, 1]]
FV=[[0,1,2,3],[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7]]

@testset "plarRemoveVertices Tests" begin
  @test typeof(plarRemoveVertices(V,FV))==Tuple{Array{Any,1},Array{Any,1}}
  @test length(plarRemoveVertices(V,FV)[1])<= length(V)
end
```

2.11 Result

Execution time on PC

```
times=[]
ptimes=[]
append!(times,Time(larRemoveVertices,[l[i][1],l[i][2]]) for i in range(1,length(l)-1))
append!(ptimes,Time(plarRemoveVertices,[l[i][1],l[i][2]]) for i in range(1,length(l)-1))
```

```
plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])
plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])
```



(a) Serial (b) Parallel

Figure 26: Execution time of function larRemoveVertices

Compare

```
plot([times,ptimes],xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",
label=["Serial","Parallel"])
```

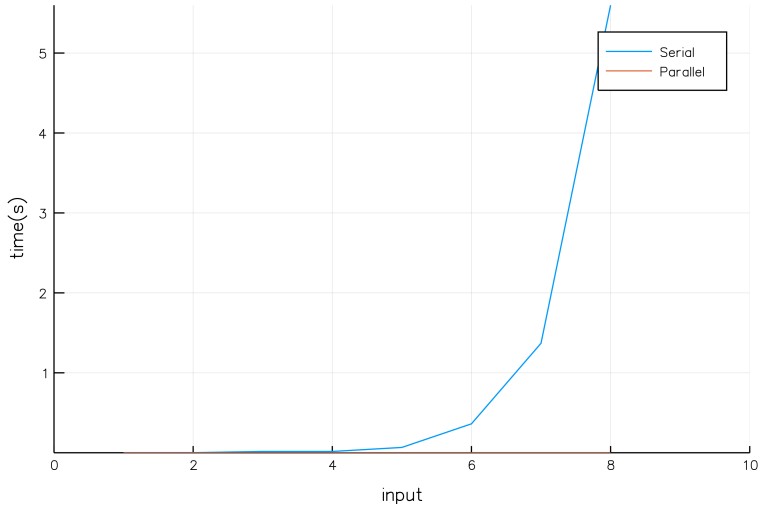


Figure 27: Compared execution time of function larRemoveVertices

3 Examples

In the following section some examples of how to use the module are presented.

Examples 1

```
square= ([[0,0],[0,1],[1,0],[1,1]],[[0,1,2,3]])
table= larApply(t(-.5,-.5))(square)
chair= larApply(s(.35,.35))(table)
chair1= larApply(t(.75,0))(chair)
chair2= larApply(r(pi/2))(chair1)
chair3= larApply(r(pi/2))(chair2)
chair4= larApply(r(pi/2))(chair3)
```

This execution return the following immage:

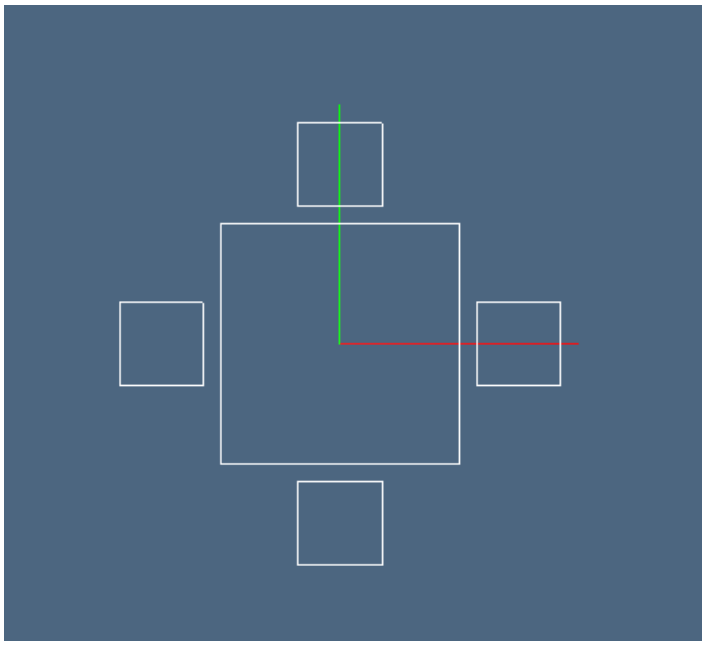


Figure 28: Table with four chair

Examples 2

```
desk=([[0, 0],[0, 1],[0, 2],[1, 0],[1, 1],[1, 2],[2, 0],[2, 1],[2, 2],[3, 0],[3, 1],[3, 2],
[4, 0],[4, 1],[4, 2],[5, 0],[5, 1],[5, 2],[6, 0],[6, 1],[6, 2],[7, 0],[7, 1],[7, 2],[8, 0],
[8, 1],[8, 2],[9, 0],[9, 1],[9, 2],[10, 0],[10, 1],[10, 2],[11, 0],[11, 1],[11, 2],[12, 0],
[12, 1],[12, 2],[13, 0],[13, 1],[13, 2],[14, 0],[14, 1],[14, 2],[15, 0],[15, 1],[15, 2],
[16, 0],[16, 1],[16, 2],[17, 0],[17, 1],[17, 2],[18, 0],[18, 1],[18, 2],[19, 0],[19, 1],
[19, 2],[20, 0],[20, 1],[20, 2]], [[0, 1, 3, 4],[1, 2, 4, 5],[3, 4, 6, 7],[4, 5, 7, 8],
[6, 7, 9, 10],[7, 8, 10, 11],[9, 10, 12, 13],[10, 11, 13, 14],[12, 13, 15, 16],
[13, 14, 16, 17],[15, 16, 18, 19],[16, 17, 19, 20],[18, 19, 21, 22],[19, 20, 22, 23],
```

```
[21, 22, 24, 25],[22, 23, 25, 26],[24, 25, 27, 28],[25, 26, 28, 29],[27, 28, 30, 31],
[28, 29, 31, 32],[30, 31, 33, 34],[31, 32, 34, 35],[33, 34, 36, 37],[34, 35, 37, 38],
[36, 37, 39, 40],[37, 38, 40, 41],[39, 40, 42, 43],[40, 41, 43, 44],[42, 43, 45, 46],
[43, 44, 46, 47],[45, 46, 48, 49],[46, 47, 49, 50],[48, 49, 51, 52],[49, 50, 52, 53],
[51, 52, 54, 55],[52, 53, 55, 56],[54, 55, 57, 58],[55, 56, 58, 59],[57, 58, 60, 61],
[58, 59, 61, 62]])
seat=larApply(s(0.02,0.3))(desk)
seat=larApply(t(0.55,-0.85))(seat)
line=Struct([desk,repeat([seat,t(1.65,0)],outer=12)...])
line=Struct([line,t(23,0),line])
lines=Struct([repeat([line,t(0,-3)],outer=6)...])
teacherdesk=larApply(s(0.5,0.8))(desk)
teacherdesk=larApply(t(16.5,5))(teacherdesk)
chair=larApply(s(0.15,0.5))(teacherdesk)
chair=larApply(t(18,5))(chair)
classroom=Struct([teacherdesk,chair,lines])
class=evalStruct(classroom)
```

This execution return the following immage:

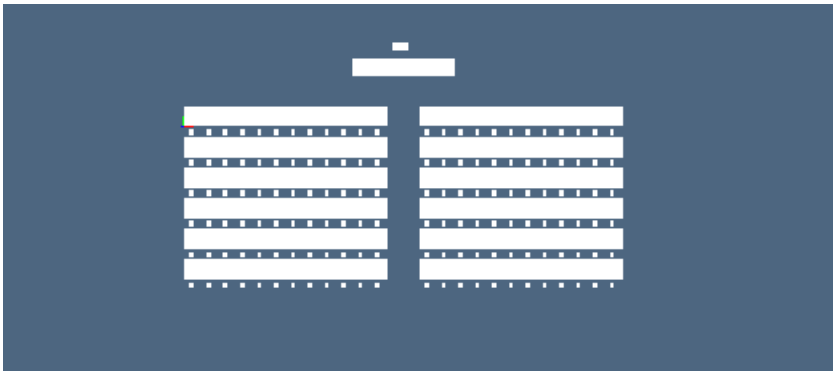


Figure 29: Classroom

4 Conclusion

As shown in the plots, generally, parallelization slows execution times, expecially when the functions are generated by large Struct objects. In this case, execution times stay large even when the tests are performed in the sequential framework on Tesla. Coversely, functions generated by the increase of array objects show lower times of execution.This suggests that functions generated by user defined Struct objects are systematically slower than those generated using native language objects (as Tuples or Arrays).

References

[1] A.Paoluzzi Hierarchical structures with LAR,March 29,2016.

[2] Julia Documentation.