

# IN480 Larstruct Module

Desiree Adiutori Alessia Giulia Cossu

June 14, 2018

## Contents

<b>Introduction</b>	<b>4</b>
<b>1 API</b>	<b>5</b>
<b>2 Implementation</b>	<b>7</b>
2.1 checkStruct . . . . .	9
2.1.1 Conversion . . . . .	9
2.1.2 Parallelization . . . . .	9
2.1.3 Unit-Test . . . . .	10
2.2 larApply . . . . .	11
2.2.1 Conversion . . . . .	11
2.2.2 Parallelization . . . . .	13
2.2.3 Unit-Test . . . . .	13
2.2.4 Results . . . . .	15
2.3 box . . . . .	17
2.3.1 Conversion . . . . .	17
2.3.2 Parallelization . . . . .	18
2.3.3 Unit-Test . . . . .	19
2.3.4 Results . . . . .	20
2.4 traversal . . . . .	22
2.4.1 Conversion . . . . .	22
2.4.2 Parallelization . . . . .	24
2.4.3 Unit-Test . . . . .	24
2.4.4 Results . . . . .	25
2.5 Struct . . . . .	28
2.5.1 Conversion . . . . .	28
2.5.2 Parallelization . . . . .	30
2.5.3 Unit-Test . . . . .	32

2.5.4	Results . . . . .	32
2.6	embedTraversal . . . . .	34
2.6.1	Conversion . . . . .	34
2.6.2	Parallelization . . . . .	36
2.6.3	Unit-Test . . . . .	37
2.6.4	Results . . . . .	37
2.7	embedStruct . . . . .	40
2.7.1	Conversion . . . . .	40
2.7.2	Parallelization . . . . .	40
2.7.3	Unit-Test . . . . .	41
2.7.4	Results . . . . .	41
2.8	removeDups . . . . .	42
2.8.1	Conversion . . . . .	42
2.8.2	Parallelization . . . . .	43
2.8.3	Unit-Test . . . . .	44
2.9	struct2lar . . . . .	45
2.9.1	Conversion . . . . .	45
2.9.2	Parallelization . . . . .	47
2.9.3	Unit-Test . . . . .	49
2.9.4	Results . . . . .	50
2.10	larRemoveVertices . . . . .	51
2.10.1	Conversion . . . . .	51
2.10.2	Parallelization . . . . .	53
2.10.3	Unit-Test . . . . .	53
2.11	Result . . . . .	53
<b>3</b>	<b>Examples</b>	<b>55</b>
<b>4</b>	<b>Conclusion</b>	<b>55</b>
<b>References</b>		<b>55</b>

## List of Figures

1	API . . . . .	6
2	Execution time of function larApply . . . . .	16
3	Compare Parallel and Serial execution time of function larApply . . . . .	16
4	Execution time of function box on Tesla . . . . .	21
5	Compared Execution time of function box on Tesla . . . . .	22
6	Execution time of function box . . . . .	22

7	Compared execution time of function box . . . . .	23
8	Execution time of function traversal on Tesla . . . . .	25
9	Compared execution time of function traversal on Tesla . . . . .	26
10	Execution time of function traversal . . . . .	27
11	Compared execution time of function traversal . . . . .	27
12	Execution time of function Struct on Tesla . . . . .	33
13	Compared execution time of function Struct on Tesla . . . . .	34
14	Execution time of function embedTraversal on Tesla . . . . .	38
15	Compared execution time of function embedtraversal on Tesla . . . . .	39
16	Execution time of function embedStruct on Tesla . . . . .	42
17	Compared execution time of function embedtraversal on Tesla . . . . .	43
18	Execution Time of function struct2lar on Tesla . . . . .	50
19	Compared execution time of function struct2lar on Tesla . . . . .	51
20	Execution time of function struct2lar . . . . .	51
21	Compared execution time of function struct2lar . . . . .	52
22	Execution time of function larRemoveVertices . . . . .	54
23	Compared execution time of function larRemoveVertices . . . . .	55
24	Table with four chair . . . . .	56

# Introduction

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

In this module there are:

- The **Affine transformations** are based on elementary matrices for affine transformation 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 embedding two-dimensional LAR model in 3D space and remove duplicate faces, vertices and cells of geometric object.

In the next sections the main functions of the module will be shown through the API. Furthermore, each function will be proposed to convert code from the Python language to Julia, parallelization, some unit tests and the study of 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 cell vector.

- **evalStruct**(Struct)→ Array,

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 cell vector.

- **vcode**(Int)(Array)→String,

generates the function that approximates each element of the data structure to which it is applied. the approximation occurs with precision given by the entire input data. It transforms the data structure obtained in string to be returned to output.

- **struct2lar**(Struct)→ Array,

it extracts the elements composing the input and combines them into a single element represented by the output array, where: the first element contains the array of the coordinates of the vertices of all the objects present, the second contains vectors of larger cells without duplicates of vertices. If we are in 3D: the third element contains the 3D cells, this array can add elements depending on how many were added to the first one.

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

takes in input arrays of vertices and cells and returns them private of their duplicates

- **larEmbed**(Int)(Tuple)→ Tuple,

if the input n is such that  $n > 0$  the function generated increases n the size of the object to which it is applied, otherwise it reduces the size of n.

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

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

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

creates a copy of the object to which it is applied, increases its size, as much as the value entered in input and returns the new object created.

## Local Functions

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

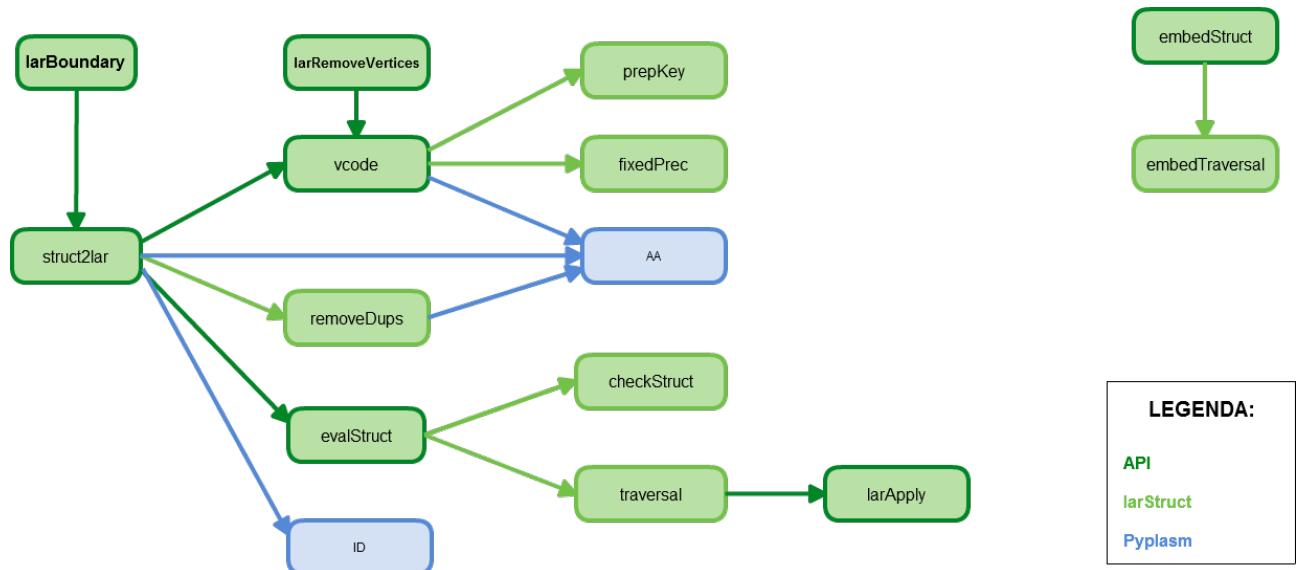


Figure 1: larStruct API

## 2 Implementation

This section shows the conversion from Python language to Julia, of some of the most important module functions. The Results section is divided into two parts: the first shows the performance plot of the functions, run on the "Tesla" with 30 processors, where the input is increased by increasing the body of the objects of Struct type, through the 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
```

In the second part plot the execution time on a personal computer with 4 processors, where the input's increase is the output of "LarCuboids" function, imported by "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 to calculate the execution time is:

```

function Time(f::Function,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` evaluates an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

The requirements for running scripts are:

- using `Base.Test`
- using `Plots`
- using `Distributions`

from the Julia terminal on Tesla:

```
addprocs(29)
```

or from the shell run Julia with the command:

```
julia -p 30
```

from the Julia terminal on personal computer:

```
addprocs(3)
```

or from the shell run Julia with the command:

```
julia -p 4
```

The subsection “Parallelization” of each function 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.

## 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" 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**

```
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**

```
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

```
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
```

```

end
return larApply0
end

```

### 2.2.2 Parallelization

The functions for 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]]]
cubess=[[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
@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!(ptimes,Time(plarApply(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"])

```

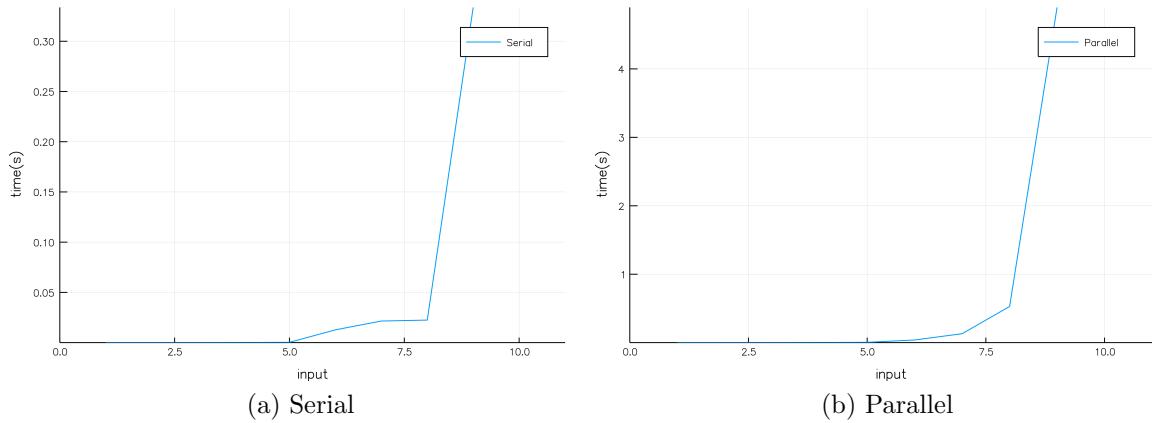


Figure 2: Execution time of function larApply

### Compare

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

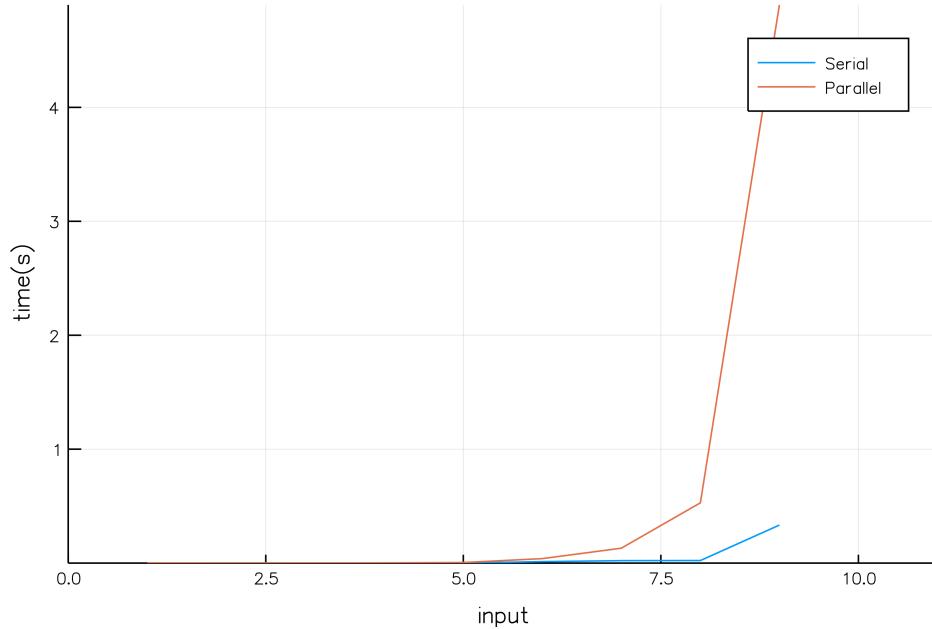


Figure 3: Compare Parallel and Serial execution time of function larApply

## 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
    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
    end
    return [theMin,theMax]
end
```

```

        end
    end
    for (k,val) in enumerate(modelMax)
        if val > theMax[k]
            theMax[k]=val
        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]])
cubess(([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 Tesla

```

input =[1,10,50,10^2,5*10^2,10^3,5*10^3,10^4,5*10^4,10^5,5*10^5,10^6]
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)
```

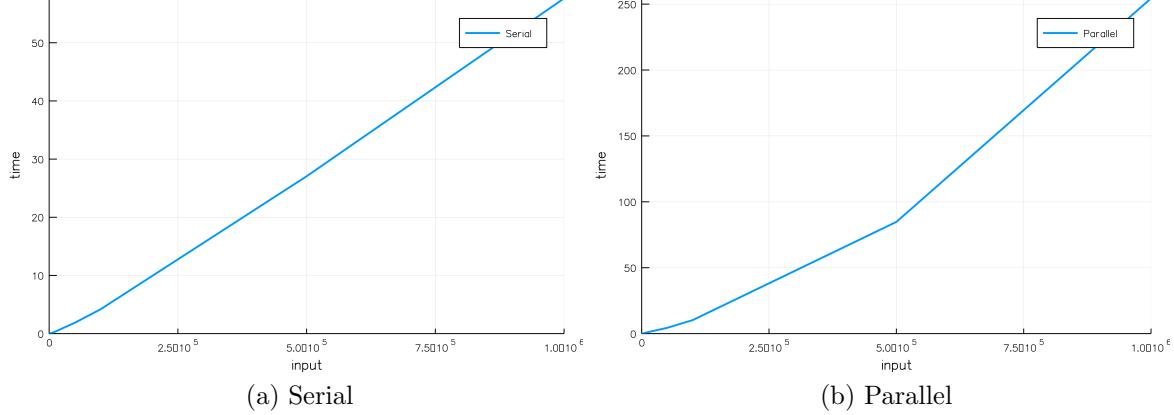


Figure 4: Execution time of function box on Tesla

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)
```

## 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(ptimes)+2),ylabel="time(s)",label=["Parallel"])
```

Compare

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

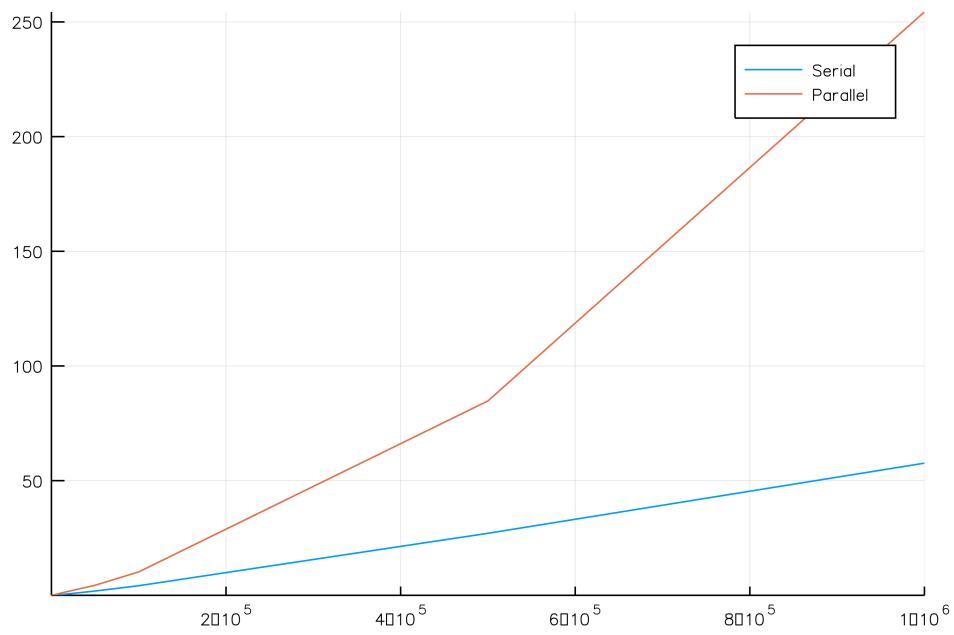


Figure 5: Compared Execution time of function box on Tesla

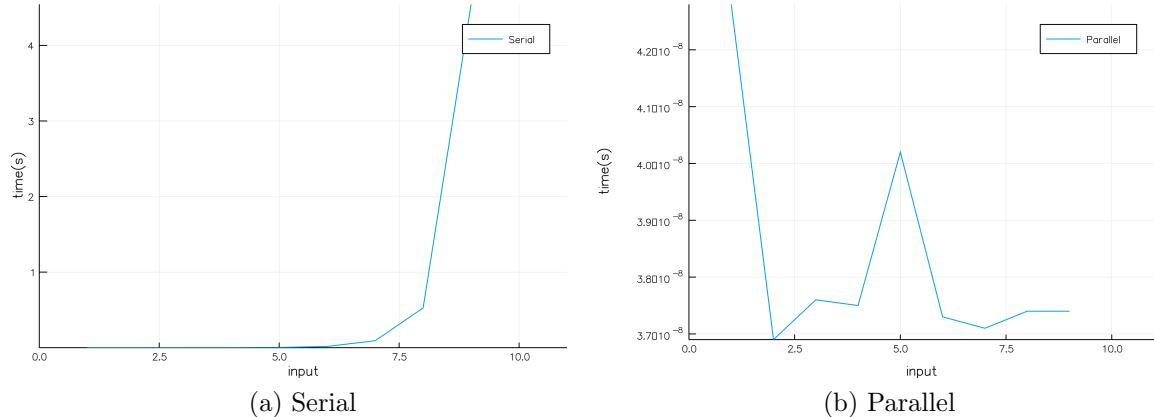


Figure 6: Execution time of function box

## 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):
```

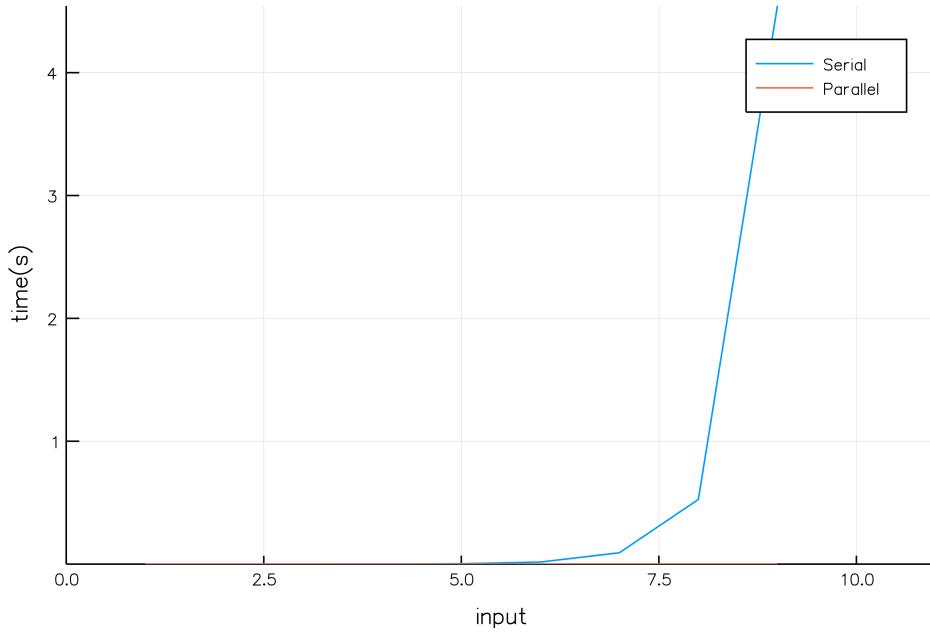


Figure 7: Compared execution time of function box

```

scene += [larApply(CTM)(obj[i])]
elif(isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (len(obj[i]==2 or len(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

```

```

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

```

@testset "traversal" begin
    square=[[0, 0], [0, 1], [1, 0], [1, 1]], [[0, 1, 2, 3]]
    @everywhere structure=Struct([square])
    @everywhere dim=checkStruct(structure.body)
    @test length(traversal(eye(dim+1),[],structure,[]))==length(structure.body)
    @test typeof(traversal(eye(dim+1),[],structure,[]))==Array{Any,1}
end

```

### Parallel Tests

```

square=[[0, 0], [0, 1], [1, 0], [1, 1]], [[0, 1, 2, 3]]
structure=pStruct([square])
dim=pcheckStruct(structure.body)
@testset "ptraversal" begin
    @test length(ptraversal(eye(dim+1),[],structure,[]))==length(structure.body)
    @test typeof(ptraversal(eye(dim+1),[],structure,[]))==Array{Any,1}
end

```

## 2.4.4 Results

### Execution time on Tesla

```

input=[1,10,50,10^2,5*10^2,10^3,5*10^3,10^4,5*10^4,10^5,5*10^5,10^6]
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)

```

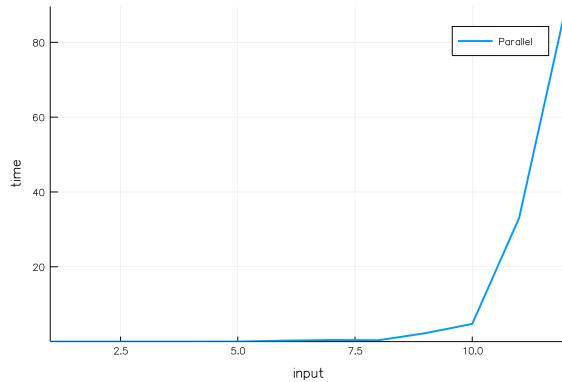
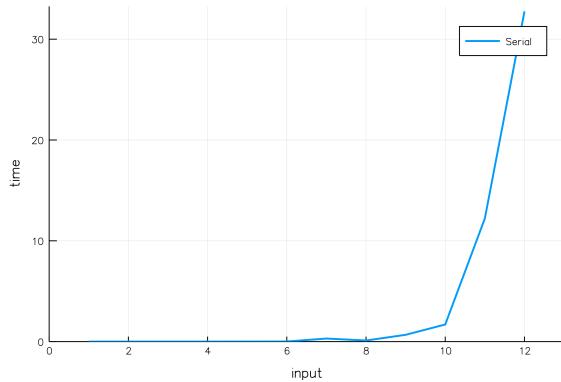


Figure 8: Execution time of function traversal on Tesla

Compare

```

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

```

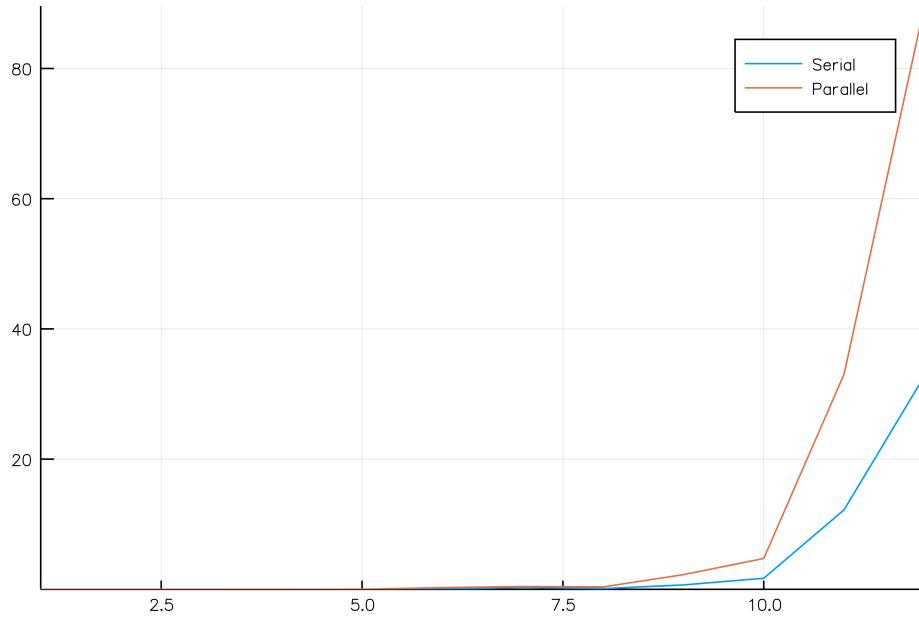


Figure 9: Compared execution time of function traversal on Tesla

### Execution time on PC

```

times=[]
ptimes=[]
input=[]
dim=[]
for i in range(1,length(1)-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"])

```

Compare

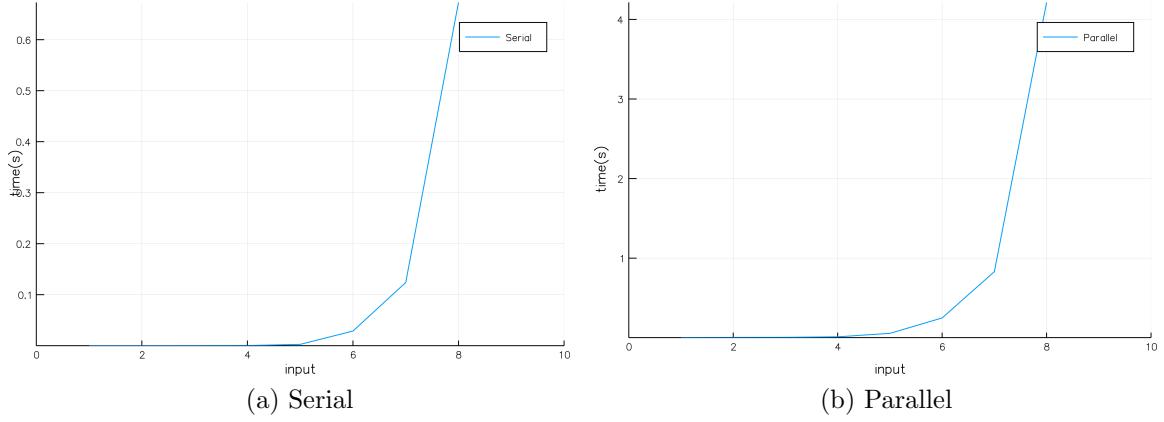


Figure 10: Execution time of function traversal

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

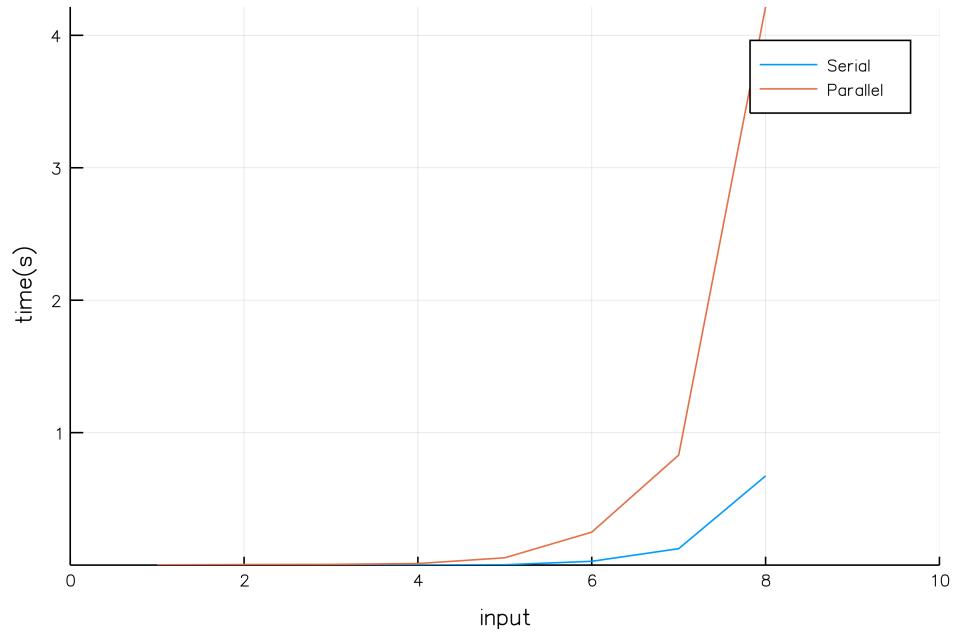


Figure 11: Compared execution time of function traversal

## 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))
    end

```

```

        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 Tesla

```

input=[1,10,50,10^2,5*10^2,10^3,5*10^3,10^4,5*10^4,10^5,5*10^5,10^6]
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

```

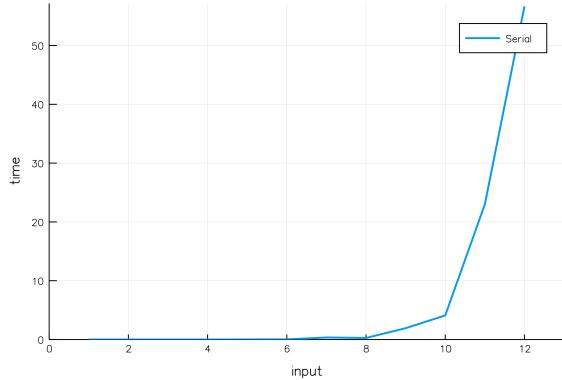
```

end

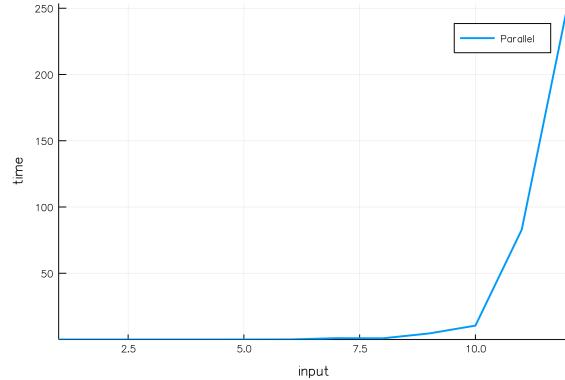
y,yp=timeStruct(square,input)
p=plot(y,xaxis='input',yaxis='time',xlims=(0,length(input)+1),
       ylims=(0,maximum(y)+0.5), label=['Serial'],lw=2)

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 12: Execution time of function Struct on Tesla

Compare

```

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

```

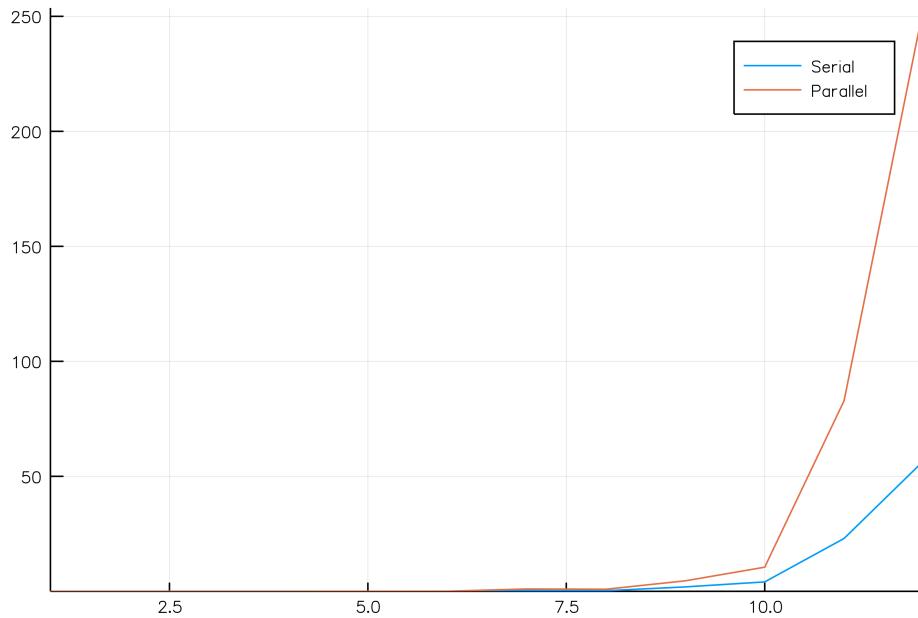


Figure 13: Compared execution time of function Struct on Tesla

## 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
                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
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=[1,10,50,10^2,5*10^2,10^3,5*10^3,10^4,5*10^4,10^5,5*10^5,10^6]
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))
       ylims=(0,maximum(y)+0.5), label=["Serial"],lw=2)

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

```

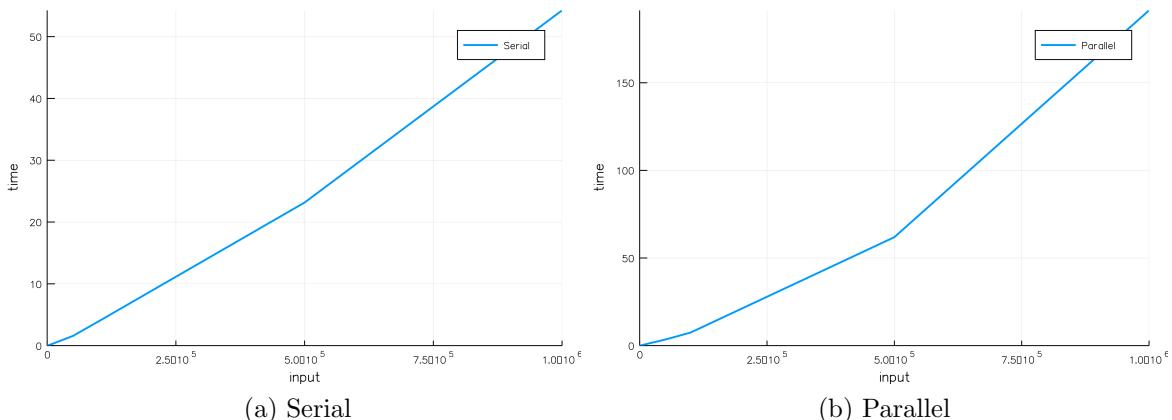


Figure 14: Execution time of function embedTraversal on Tesla

## Compare

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

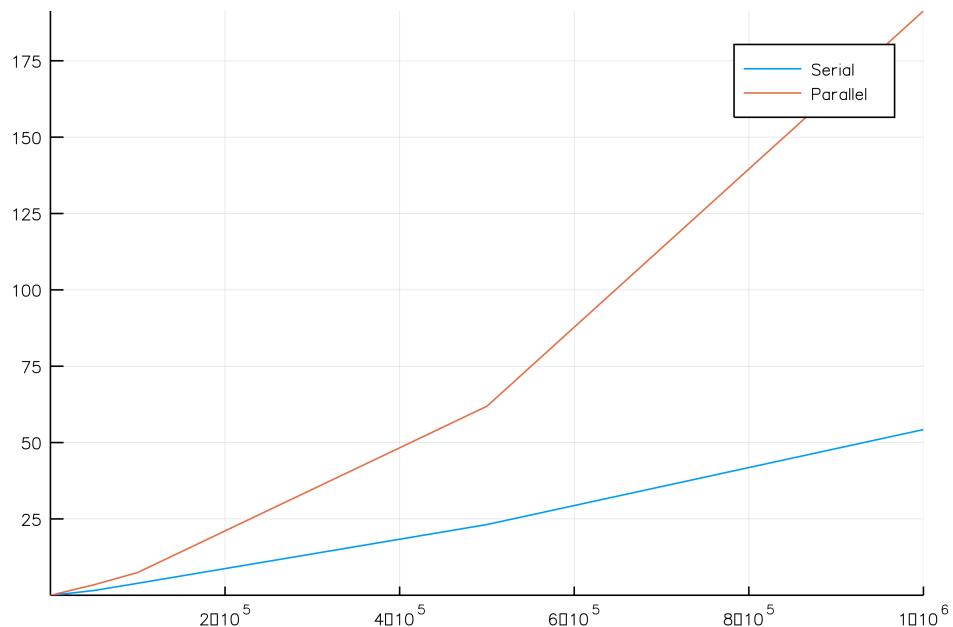


Figure 15: Compared execution time of function embedtraversal on Tesla

## 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

```

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)

```

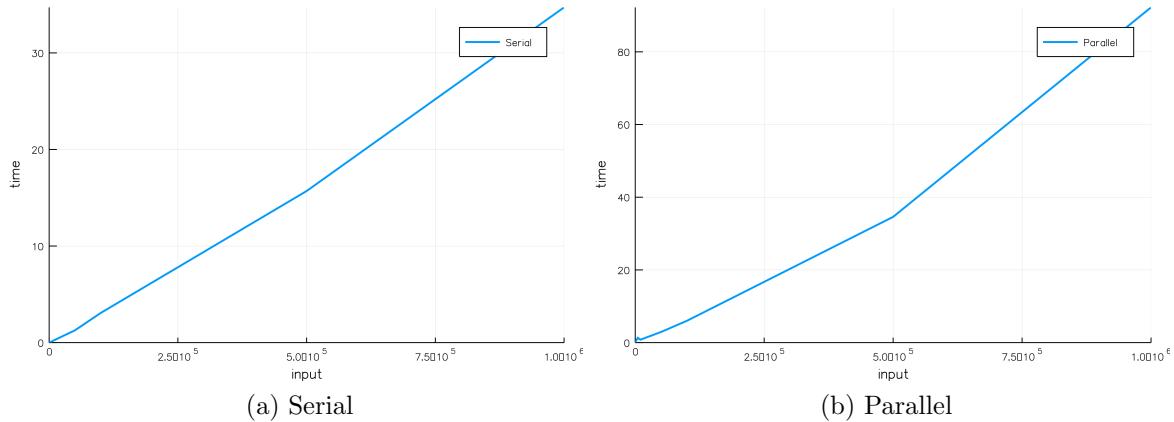


Figure 16: Execution time of function embedStruct on Tesla

## Compare

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

## 2.8 removeDups

### 2.8.1 Conversion

```
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

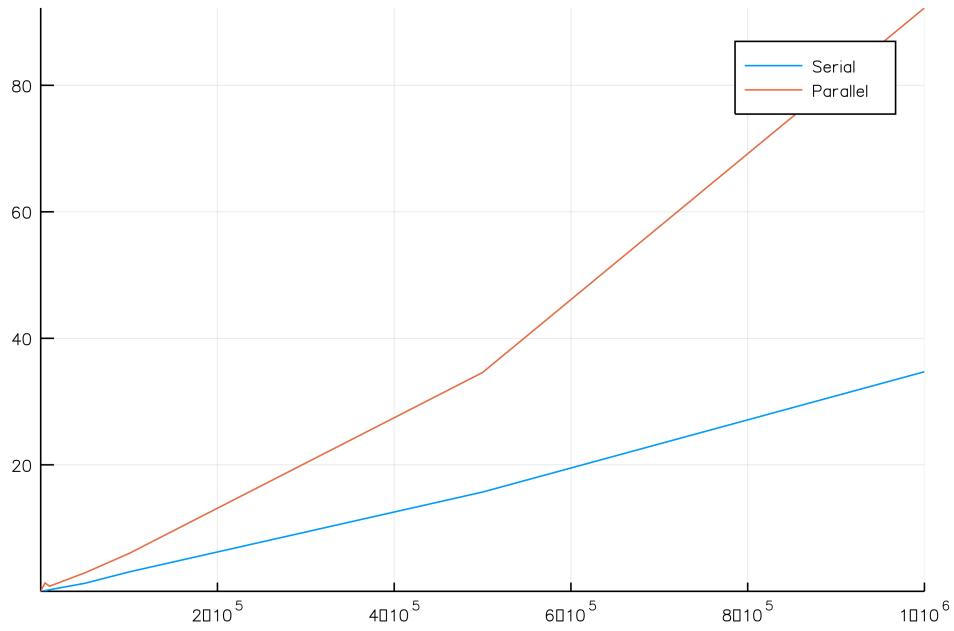


Figure 17: Compared execution time of function embed traversal on Tesla

```

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

```

```

end
@parallel for f in values(no_duplicates)
    append!(CW,f[1])
end
return CW
end

```

### 2.8.3 Unit-Test

#### Serial Tests

```

CW1=[[0,1,2,3],[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7],[4,5,6,7],[8,9,10,11],
[4,5,8,9],[6,7,10,11],[4,6,8,10],[5,7,9,11]]
CW2=[[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15],[16,17,18,19]]

```

```

@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

```

```

CW1=[[0,1,2,3],[4,5,6,7],[0,1,4,5],[2,3,6,7],[0,2,4,6],[1,3,5,7],[4,5,6,7],[8,9,10,11],
[4,5,8,9],[6,7,10,11],[4,6,8,10],[5,7,9,11]]
CW2=[[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15],[16,17,18,19]]

```

#### Parallel Tests

```

@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

```

## 2.9 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

```
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

```
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
end

```

## 2.9.2 Parallelization

```

@everywhere function pfixedPrec(PRECISION)
    function pfixedPrec0(value)
        out=round.(value,PRECISION)
        if out===-0.0
            out=0.0
        end
        return string(out)
    end
    return pfixedPrec0
end

function pvcode(PRECISION=4)
    function pvcode0(vect)
        return pfixedPrec(PRECISION)(vect)
    end
    return pvcode0
end

@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 length(model)==3
            V,FV,EV=model
        end
    @sync begin

```

```

for (k,incell) in enumerate(FV)
    outcell=[]
    @async begin
        for v in incell
            key=pvcode(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
if length(model)==3
    @sync begin
        for (k,incell) in enumerate(FV)
            outcell=[]
            @async begin
                for v in incell
                    key=pvcode(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
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" begin
@testset "struct2lar 2D" 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)...])
    @test typeof(struct2lar(structure))==Tuple{Array{Any,1},Array{Array{Any,1},1}}
    @test length(struct2lar(structure)[1][1])==2
end
@testset "struct2lar 3D" begin
    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));
    @test typeof(struct2lar(structure))==Tuple{Array{Any,1},Array{Array{Any,1},1}}
    @test length(struct2lar(structure)[1][1])==3
end
end

```

#### Parallel Tests

```

@testset "pstruct2lar" begin
@testset "pstruct2lar 2D" 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)...])
    @test typeof(pstruct2lar(structure))==Tuple{Array{Any,1},Array{Any,1}}
    @test length(pstruct2lar(structure)[1][1])==2
end
@testset "pstruct2lar 3D" begin
    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));
    @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 Tesla

```
input=[1,10,50,10^2,5*10^2,10^3,5*10^3,10^4,5*10^4,10^5,5*10^5,10^6]
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)
```

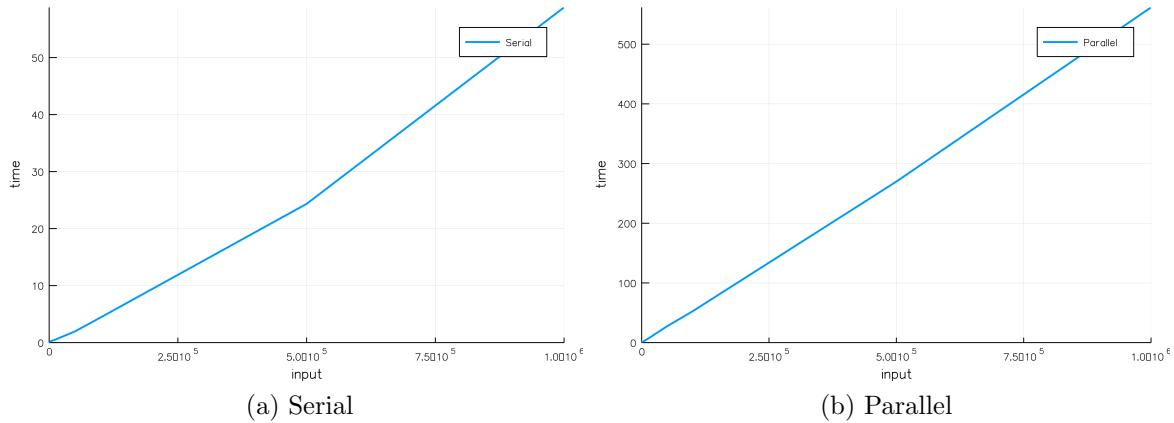


Figure 18: Execution Time of function `struct2lar` on Tesla

Compare	
<code>yc=[y,yp]</code>	
<code>pc=plot(input,yc,label=["Serial" "Parallel"])</code>	
<b>Execution time on PC</b>	
<code>times=[]</code>	
<code>ptimes=[]</code>	
<code>input=[]</code>	
<code>for i in range(1,length(l)-1)</code>	
<code>    push!(input,Struct([repeat([l[i]],outer=i)...]))</code>	
<code>end</code>	
<code>append!(times,Time(struct2lar,[input[i]])) for i in range(1,length(input)))</code>	
<code>append!(ptimes,Time(pstruct2lar,[input[i]])) for i in range(1,length(input)))</code>	
<code>plot(times,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial"])</code>	
<code>plot(ptimes,xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Parallel"])</code>	
Compare	
<code>plot([times,ptimes],xlabel="input",xlims=(0,length(times)+2),ylabel="time(s)",label=["Serial","Parallel"])</code>	

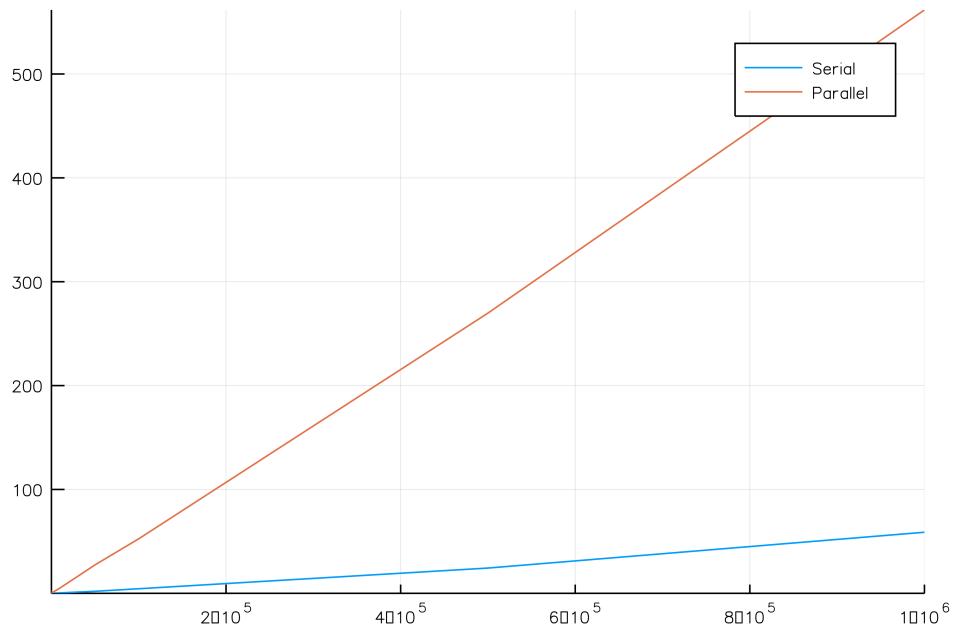


Figure 19: Compared execution time of function struct2lar on Tesla

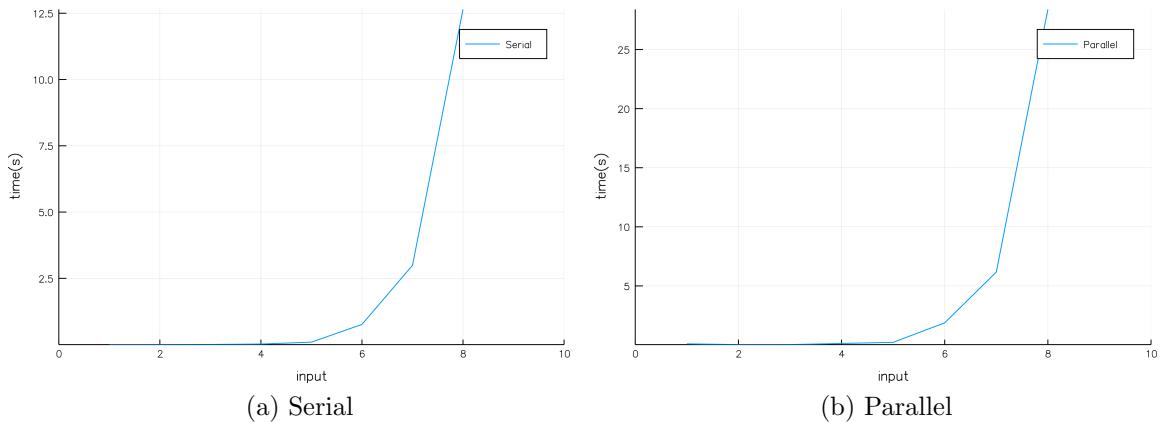


Figure 20: Execution time of function struct2lar

## 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):
```

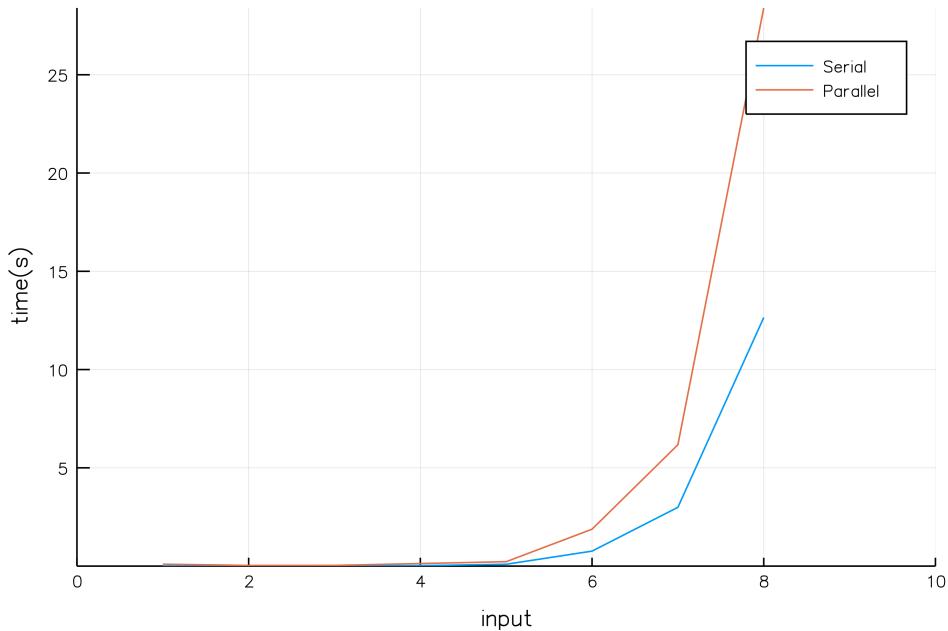


Figure 21: Compared execution time of function struct2lar

```

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

```

```

    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=pvcodes(4)(V[v+1])
                    if get(vertDict,key)==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
    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

```

#### Parallel 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

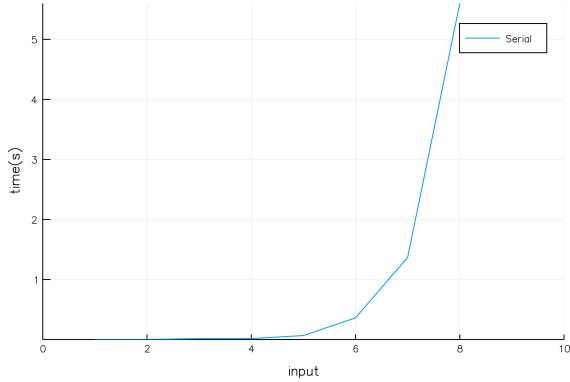
### Execution time on PC

```

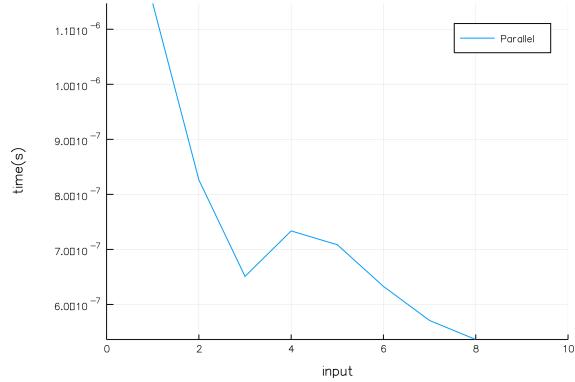
times=[]
ptimes=[]
append!(times,Time(larRemoveVertices,[l[i][1],l[i][2]])) for i in range(1,length(l)-1)
append!(ptimes,Time(plarRemoveVertices,[l[i][1],l[i][2]])) for i in range(1,length(l)-1)

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 22: Execution time of function larRemoveVertices

Compare

```

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

```

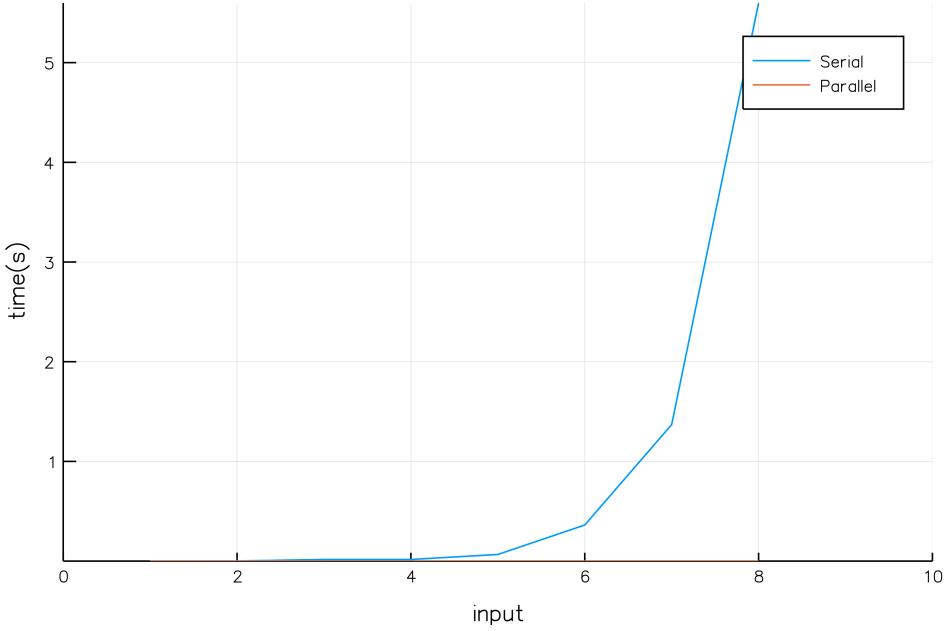


Figure 23: Compared execution time of function larRemoveVertices

### 3 Examples

```

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:

### 4 Conclusion

It can be seen from the results of the graphs that parallelization generally slows down execution times. The execution times also depend on how the input is increased, in fact the execution times calculated on Tesla also increase for the sequential. Performing a parallelization on native language objects such as Tuple or Array is faster compared to one carried out on a Struct type object.

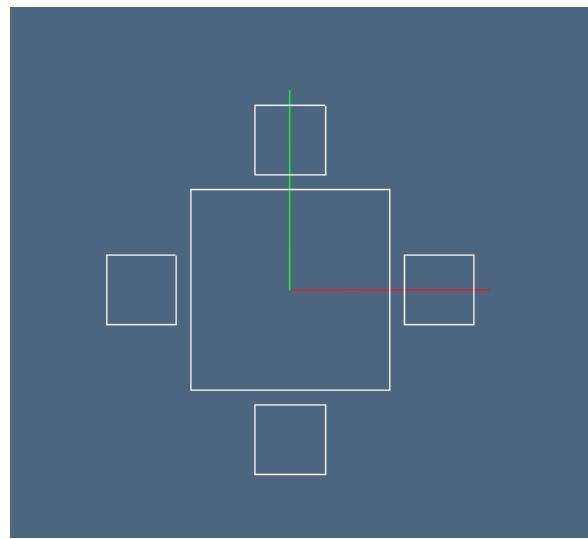


Figure 24: Table with four chair

## References

- [1] A.Paoluzzi Hierarchical structures with LAR,March 29,2016.
- [2] Julia Documentation.