

Typen und Konstruktoren (auch Kompatibilität zu existierenden Packages (DynamicalSystems.jl etc.) testen und sicherstellen, Redundanz vermeiden wo möglich)

```
BoxCollection(center::Point,radius::Number,initialSubdivisions::Number=1)
BoxCollection(x::Point,y::Point,initialSubdivisions::Number=1)
#Sollten wir das alles als Bäume implementieren, wäre natürlich z.B.
#InitialLevels besser, also log(initialSubdivisions)

Box(center::Point,radius::Number)
Box(x::Point,y::Point)

Point(...)
```

Wichtige Funktionen:

```
subdivide(bc::BoxCovering)::BoxCovering
subdivide!(bc::BoxCovering)
subdivide(bc,testFunction)

keep!(bc,TestFunction)
discard!(bc,TestFunction)

subCovering(bc::BoxCovering, ...)::BoxCovering
subCovering(bc::BoxCovering, TestFunction)::BoxCovering

inside(b::Box,p::Point)::Boolean

intersection(bc1::BoxCovering,bc2::BoxCovering)::BoxCovering
union(bc1::BoxCovering,bc2::BoxCovering)::BoxCovering
difference(bc1::BoxCovering,bc2::BoxCovering)::BoxCovering
...

testPoints(bc::BoxCovering, ...)::Array{Point}
testPoints(box::Box)::Array{Point}

plot(bc)
```

Beispielimplementierung Subdivisions-Algorithmus:

```
F = ... #Dynamik

bc = BoxCovering(Point(0,0,0),1.0)

function hits!(box::Box,points::Array{Points})
    #Wahrscheinlich nicht in korrekter Julia Syntax
    pointsThatHit = getAll(points,inside(box,point))
    remove(points,pointsThatHit)
    return !isEmpty(pointsThatHit)
end

for 1:iterations
    subdivide!(bc)
    points = testPoints(bc)
    mappedPoints = F(points)
    keep(bc,box->hits!(box,mappedPoints))
end

plot(bc)
```

Beispielimplementierung Fortsetzungs-Algorithmus:

```
F = ...
bigBoxCovering = BoxCovering(Point(0,0,0),1.0,initialSubdivisions=1000)

manifold = SubCovering(bc,Point(0.2,0.1,0.3),0.01) #Subcovering um einen Fixpunkt
... #Subdivisionsalgorithmus mit manifold

activeBoxes = manifold
for 1:iterations
    mappedPoints = F(testPoints(activeBoxes))
    activeBoxes = SubCovering(bc,box->hits(box,mappedPoints))
    manifold = union(manifold,activeBoxes)
end

plot(manifold)
```

Beispielimplementierung invariantes Maß (vmtl. sehr naiv/ineffizient erstmal):

```
F = ...
bc = BoxCovering(...,initialSubdivisions=10)
#evtl. vorverarbeitung (boxen, die sicher ~0 sind wegwerfen)
A = zeros(length(bc),length(bc))

i = 1
for boxFrom in bc:
    mappedPoints = F(testPoints(box))
    j = 1
    for boxTo in bc:
        A[i,j]=count(inside(boxTo,mappedPoints)) / size(mappedPoints)
        j++
    end
    i++
end

v,invMeasure = eigen(A)

plot(bc,invMeasure) #hier bräuchte man noch eine nette plot methode,
                    #die das farblich darstellt o.Ä.

#Alternativ:
threshold=1/length(bc)
keep!(bc,box -> invMeasure[box.nr]>threshold)
plot(bc)
```

Weitere Überlegungen nach Meeting 30.5.:

Das, was unsere Datenstruktur schnell kann und worauf sie ausgelegt ist, ist es schnell die korrekte Box für einen Punkt (oder Intervall¹ etc.) zu finden. Testfunktionen, die auf Boxen arbeiten, verleiten den Nutzer also leicht zu ineffizientem Code (der dann letztlich über alle Boxen iterieren muss). Entweder also komplett ersetzen durch Testpunkte (`keep!(bc,Points)` o.Ä.), was aber etwas unschön ist (ich behalte ja nicht die Punkte, sondern Boxen) , oder mit einem Zwischenschritt, in etwa so:

```
find(bc::BoxCovering,points::List{Point})::SubCovering
keep!(bc::BoxCovering,sc::SubCovering)

...
    subdivide!(bc)
    mappedPoints = F(testPoints(bc))
    boxesHit = find(bc,mappedPoints)
    keep!(bc,boxesHit)
```

Ob die Testfunktionen dann überhaupt noch eine Berechtigung haben (gibt es Anwendungsfälle, in denen sie natürlich sind UND man sowieso über alle Boxen iterieren muss?), bleibt zu diskutieren. (Ich tendiere zu nein, aber hänge natürlich emotional ein bisschen an meiner Idee :))

Unabhängig davon, ein Gedanke zur vorgeschlagenen Datenstruktur: Es lassen sich (mit zusätzlichem Overhead) sogar noch ein par level mehr erzeugen, wenn wir die Länge des aktiven Teils nicht explizit speichern, sondern einfach z.B. sagen “Die letzte 1 hat keine Bedeutung”. Kurzes Beispiel (mit 8-Bit, um mir Tipparbeit zu sparen):

01000000 hätte Tiefe 1, **0101**1000 Tiefe 4 und **01010001** Tiefe 7.

Das würde uns Tiefe 63 erlauben und macht auch nicht die Vergleiche kaputt (denn z.B. auf **00010100** folgt ja frühestens die **00011001**). Vermutlich übersehe ich gerade noch Komplikationen, aber zumindest wäre es ein interessanter Tradeoff zwischen Tiefe und zusätzlichem Overhead (fixed Bytes für die Tiefe sind natürlich klar schneller).

¹das müsste mit der sortierten Liste ja auch super einfach sein