

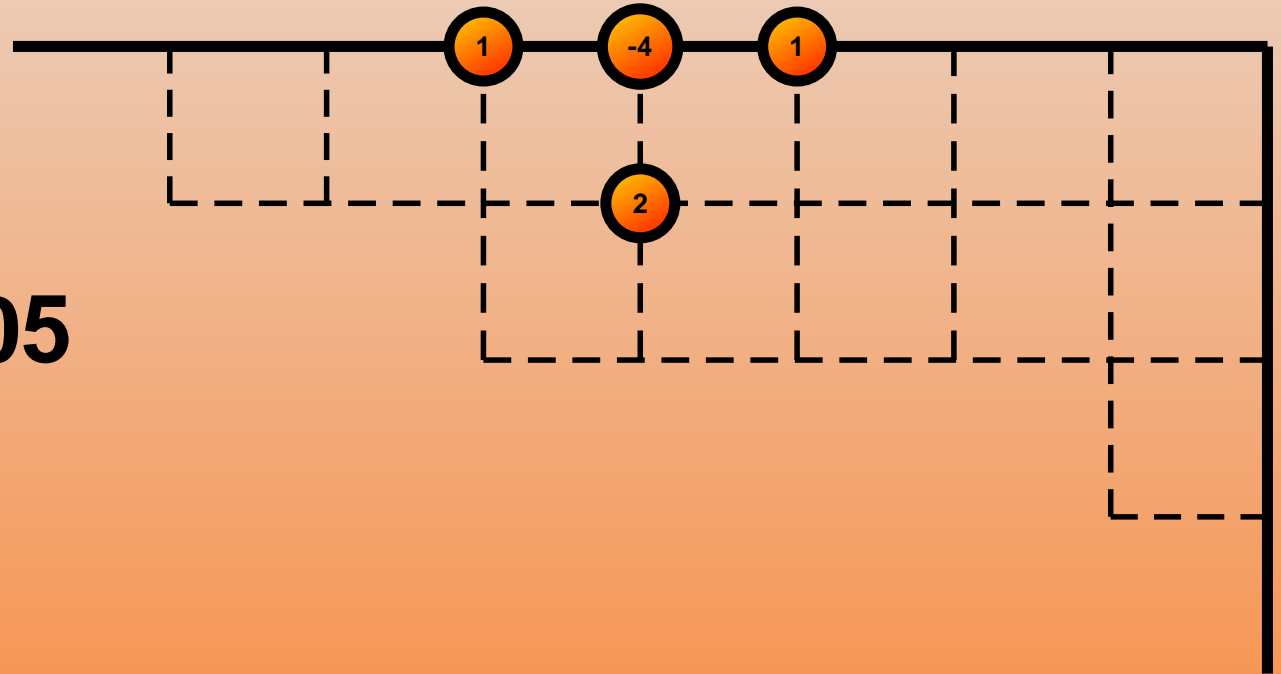
# FDM solvers in Python – a framework

**CSE05,  
Feb. 12 - 15, 2005**

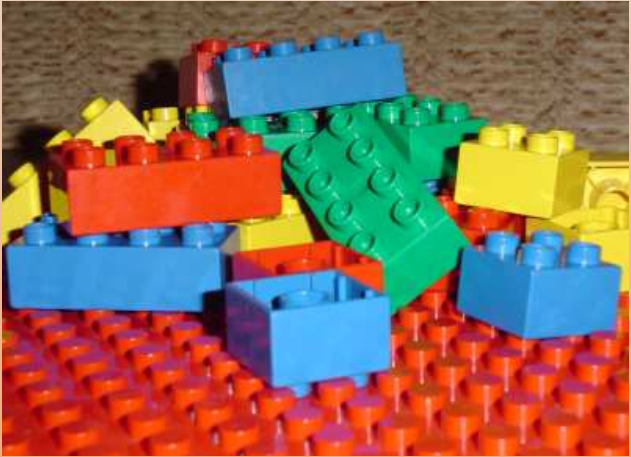
**Åsmund Ødegård**

`aasmund@simula.no`

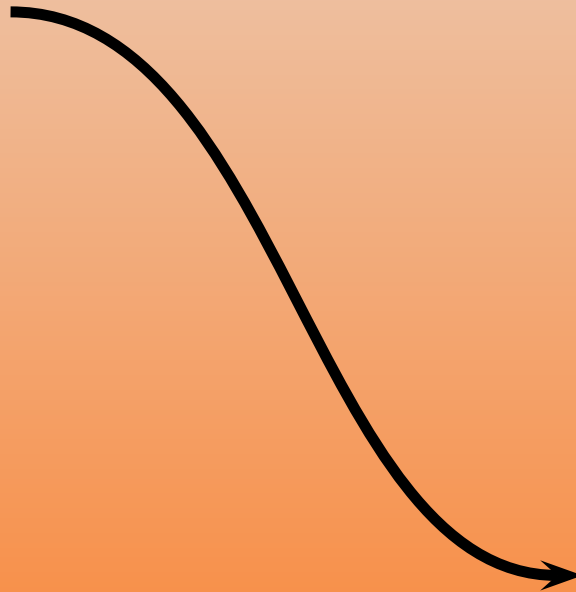
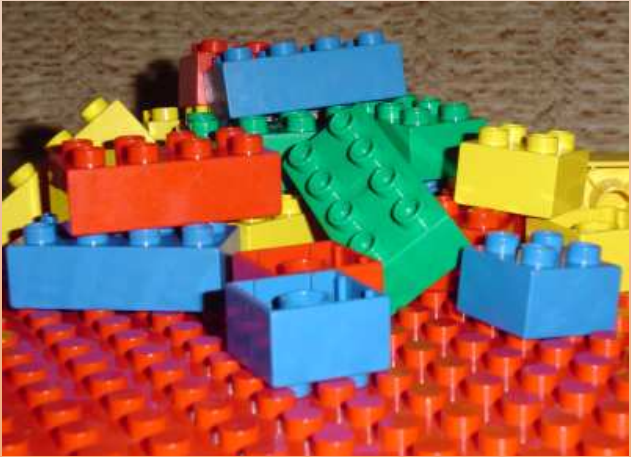
**Simula Research Laboratory**



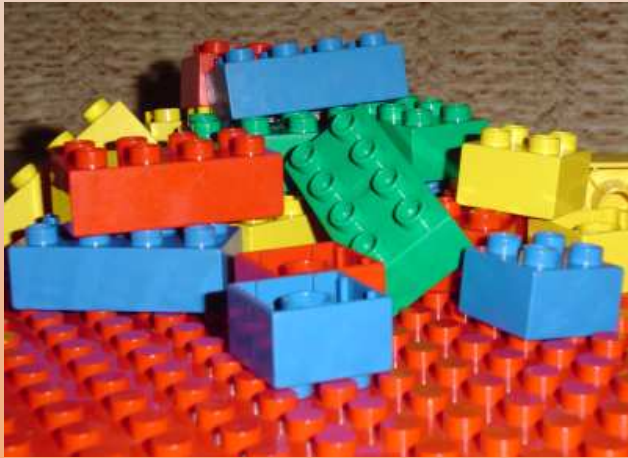
# paraStencils – from building blocks to solvers



# paraStencils – from building blocks to solvers



# paraStencils – from building blocks to solvers



Some related works:

- *hypre*, Falgout et al.
- A++/P++, Dan. Quinlan et al.
- Chombo, Colella et al.

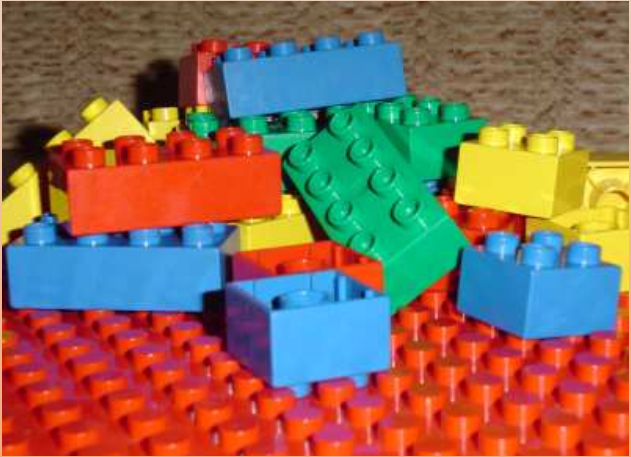
## Priorities

1. Abstractions
2. Parallelization
3. Efficiency

(Consider MS36!)

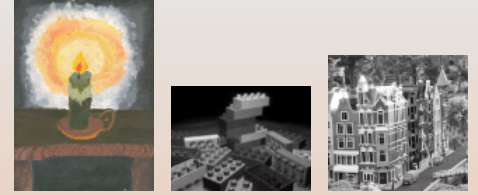


# paraStencils – from building blocks to solvers

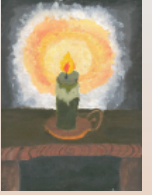


**First, some motivation**

# Motivation: why do we want another tool?

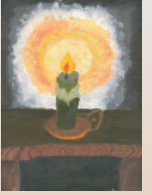


- For linear algebra, we have matlab
- Matlab: interactive, rapid prototyping, easy syntax
- What about Partial Differential Equations (PDEs)?



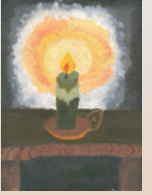
## Motivation: why do we want another tool?

- For linear algebra, we have matlab
- Matlab: interactive, rapid prototyping, easy syntax
- What about Partial Differential Equations (PDEs)?
  
- We like to build something just as easy as Matlab, for solving PDEs with the Finite Difference Method (FDM)
- The user-code should be high-level, clean and close to the math!

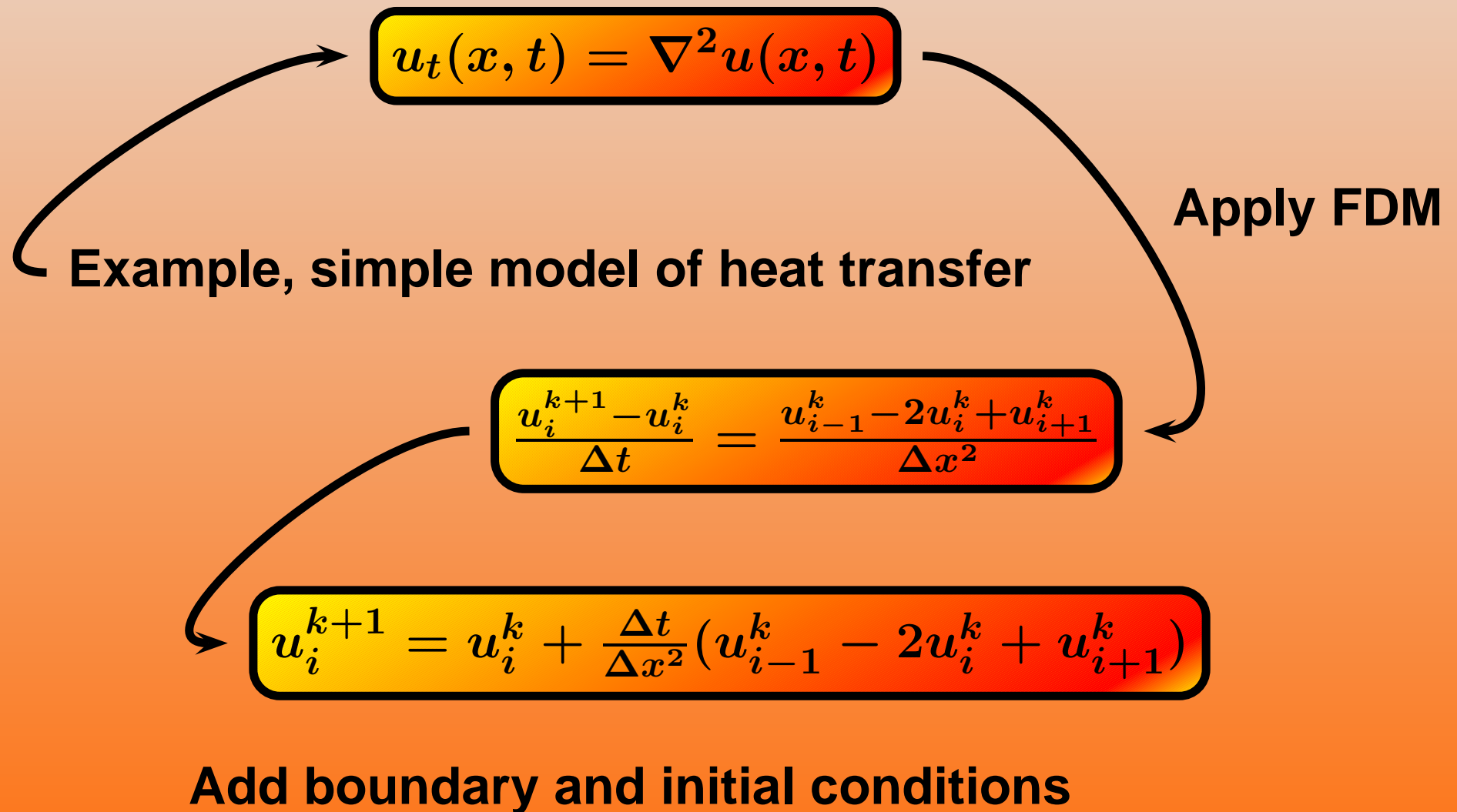


## Motivation: a simple problem

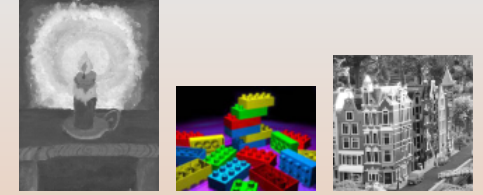
$$u_t(x, t) = \nabla^2 u(x, t)$$



## Motivation: a simple problem



# Building blocks used to specify a problem



## Grid

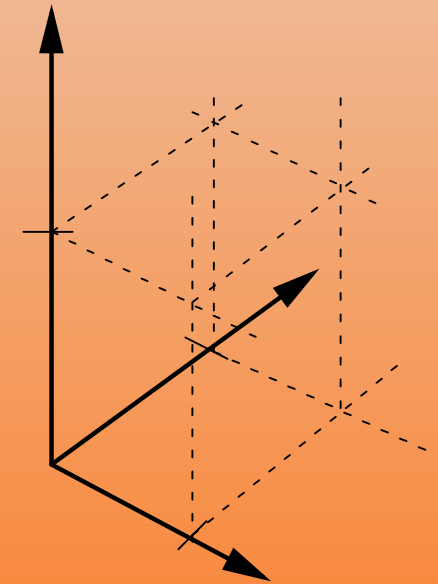
An abstraction of the domain

Number of dimensions

Domain and partition

Methods:

- setSize
- innerPoints
- boundary
- corners
- partition for parallel computer





## Building blocks used to specify a problem

**Grid**



**Stencil**

One of our main abstractions:

Define action of the PDE in one point.

Same stencil in many points, e.g all inner-points and all boundary points.

Stencils can be added:



$$u_i^{k+1} = u_i^k + \frac{\Delta t}{\Delta x^2} (u_{i-1}^k - 2u_i^k + u_{i+1}^k)$$



## Building blocks used to specify a problem

**Grid**



**Stencil**

```
from Stencil import Stencil  
s = Stencil(1, True) # 1D, variable coefficients
```

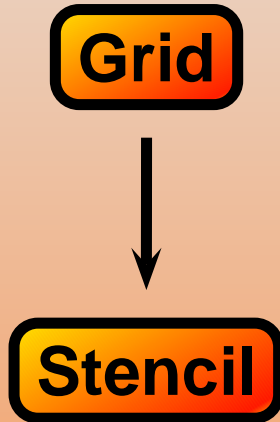
```
s.addNode(-1, [lambda x: x*x])  
s.addNode( 1, [lambda x: x*x])  
s.addNode( 0, [lambda x: -2])
```



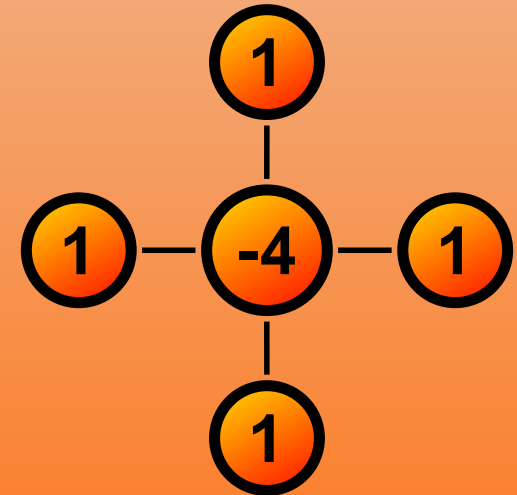
- This is the simple, direct use
- Next, we show a dimension-independent Laplace stencil.
- This is implemented in the framework as a subclass of Stencil.



# Building blocks used to specify a problem

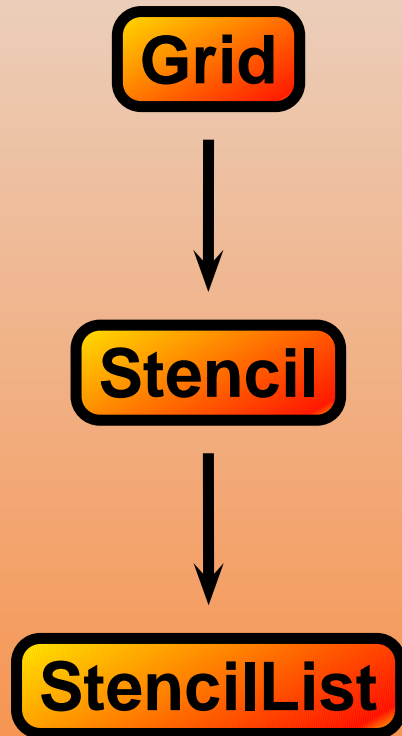


```
bi = map(lambda x: 0,range(self.nsd))
dx = grid.division
for i in xrange(self.nsd):
    # copy the basic index into changeable object
    cbi = copy(bi)
    # add "left" node
    cbi[i]=-1
    ti=tuple(cbi)
    value = 1.0/(dx[i]*dx[i])
    self.addNode(ti,value)
    # add "right" node
    cbi[i]=1
    ti=tuple(cbi)
    self.addNode(ti,value)
    # add "center"
    cbi[i]=0
    ti=tuple(cbi)
    value=-2.0/(dx[i]*dx[i])
    self.addNode(ti,value)
```





## Building blocks used to specify a problem



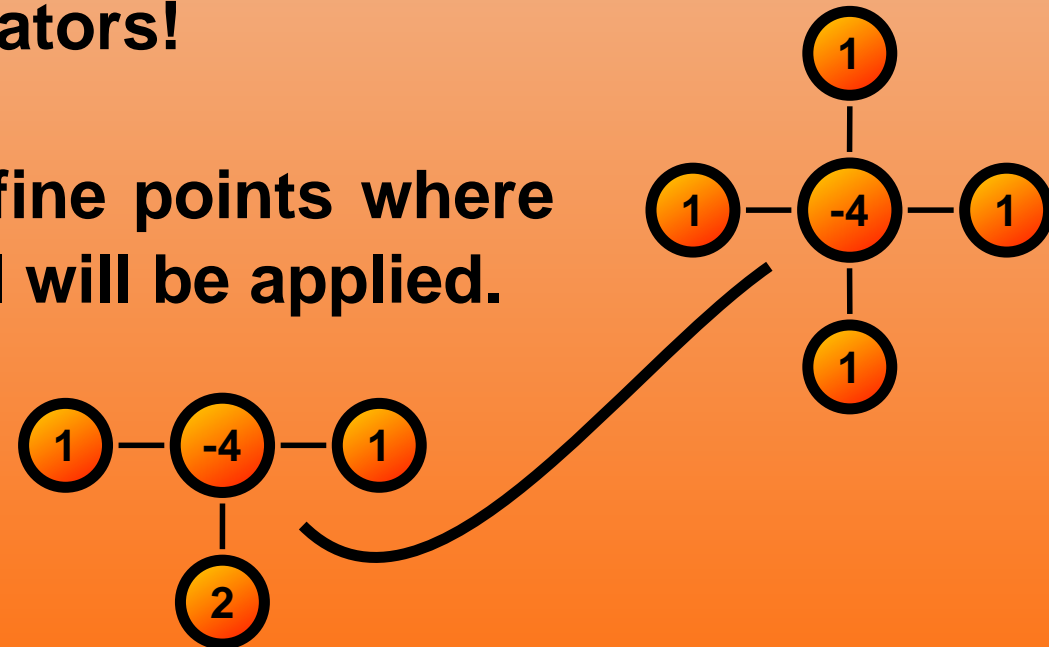
**StencilList – our next main abstraction**

**Build the action of the PDE as a list of stencils.**

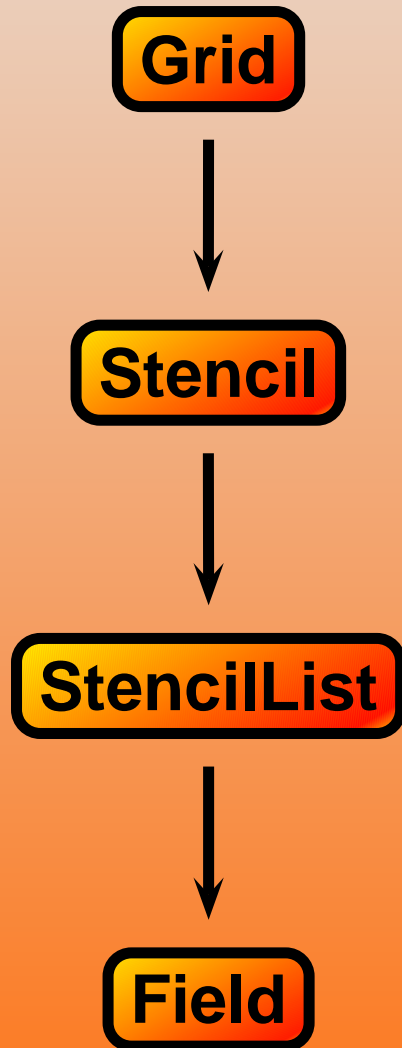
**Hold two lists:**

- List of Stencils
- List of iterators!

**Iterators define points where each Stencil will be applied.**



# Building blocks used to specify a problem



**Field** – an abstraction for data over a **Grid**

**Data**, a NumPy array

**Pointer** to a **Grid** object

**Methods:**

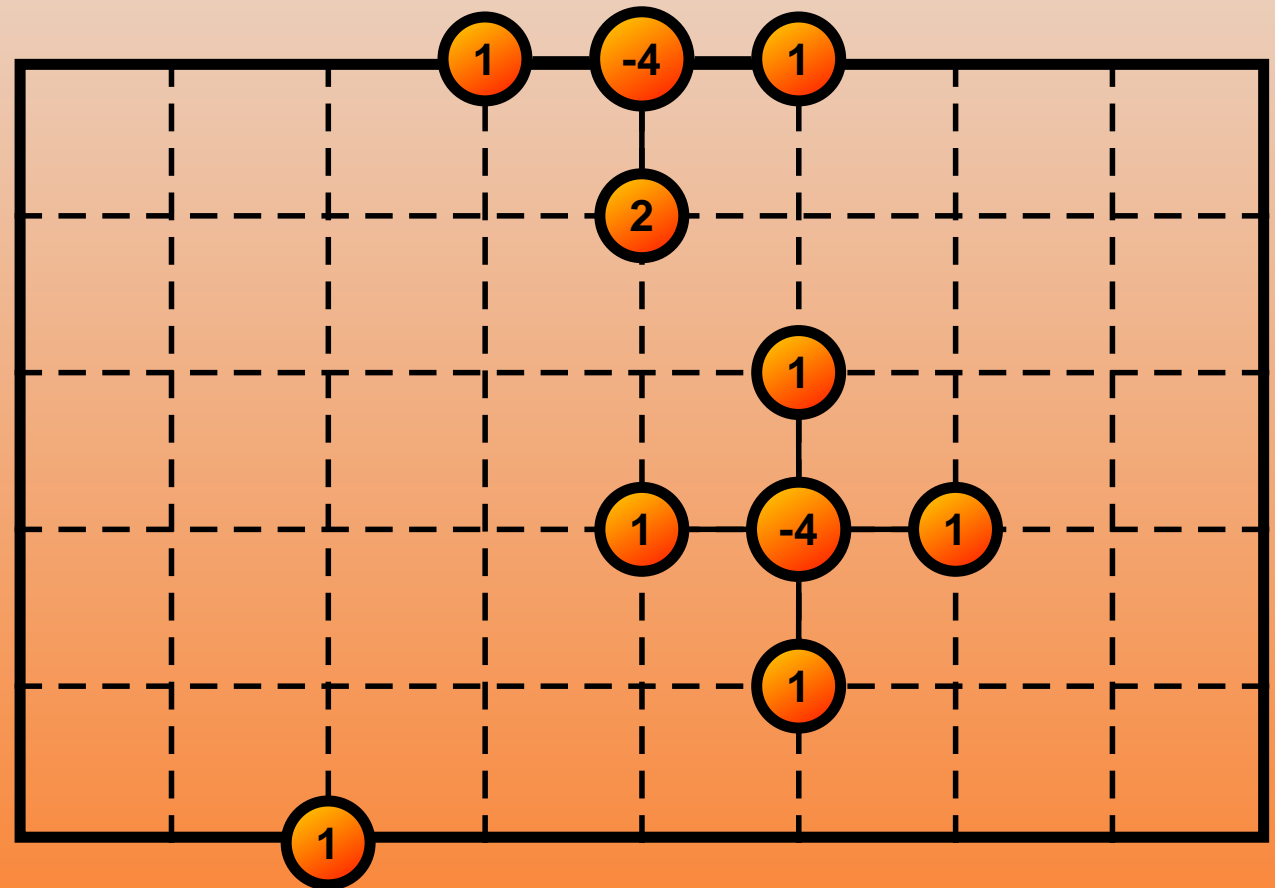
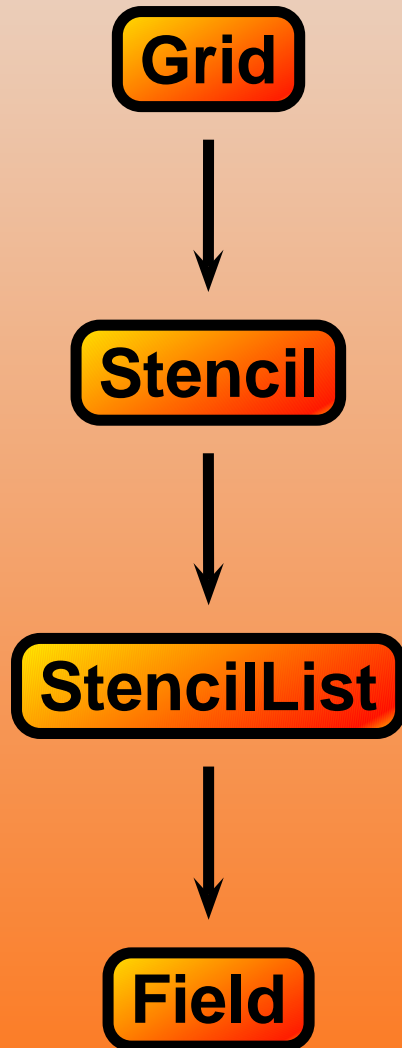
- **get and set item**

- **fill**, taking some function as argument

- **plot**

- **initParallel**, setup communication patterns

# Building blocks used to specify a problem



One explicit step:  $u_{n+1} = \text{stencillist}(u_n)$

# Parallelization built into the components



– Work in progress...

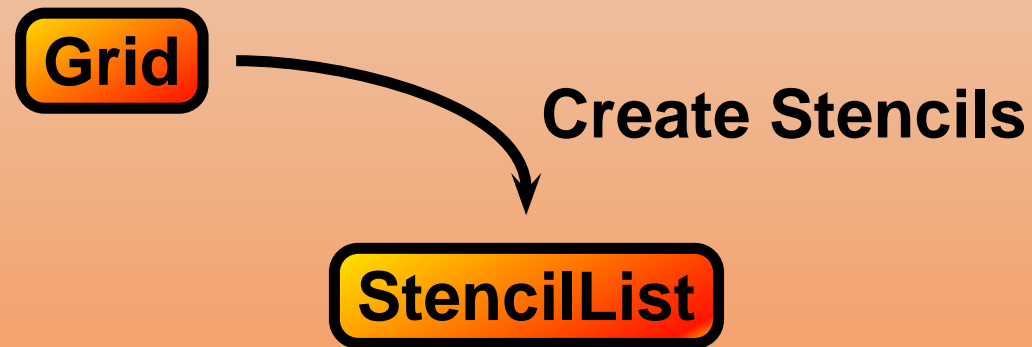
# Parallelization built into the components



**Grid**

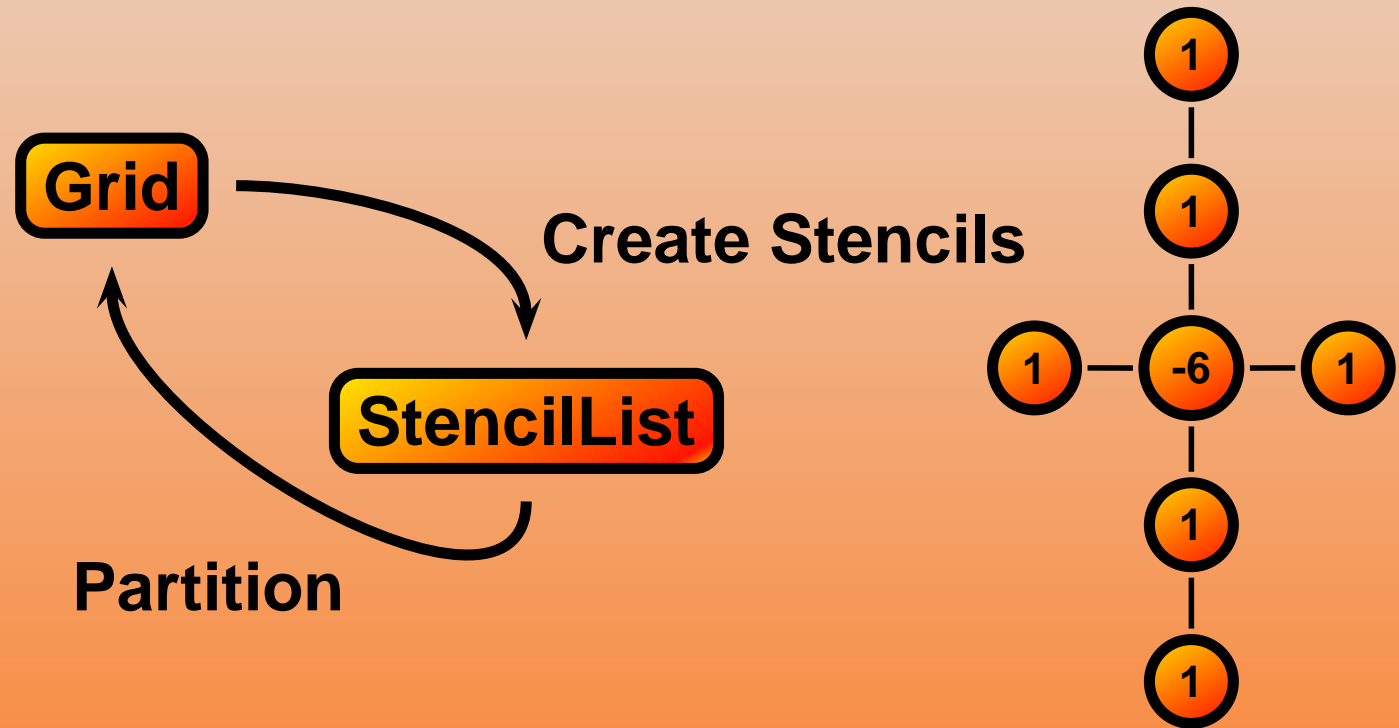
**Start with a scalar Grid**

# Parallelization built into the components



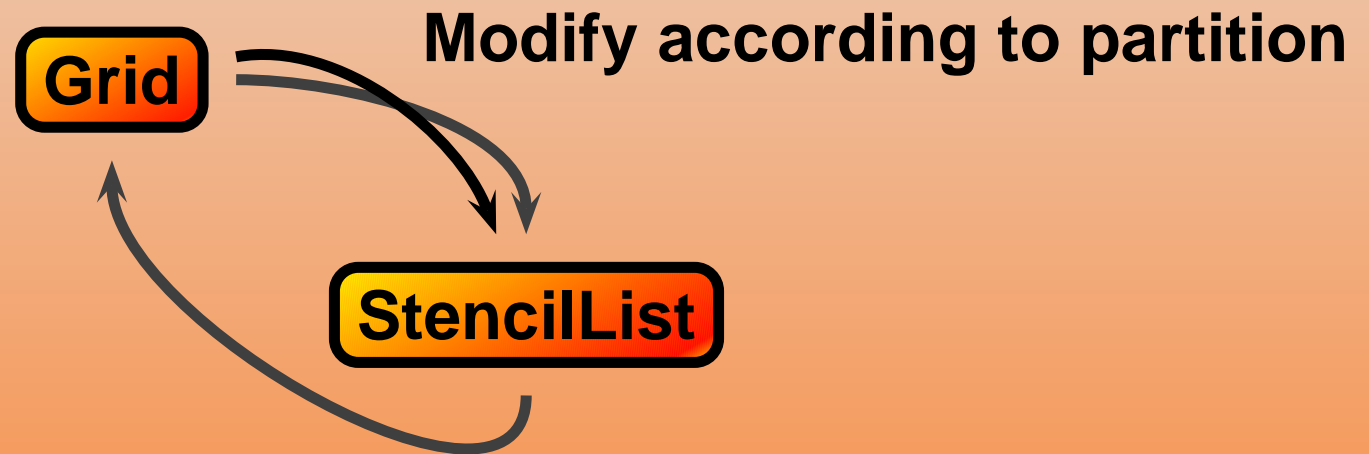
**Define a stencillist for the problem on the grid**

# Parallelization built into the components



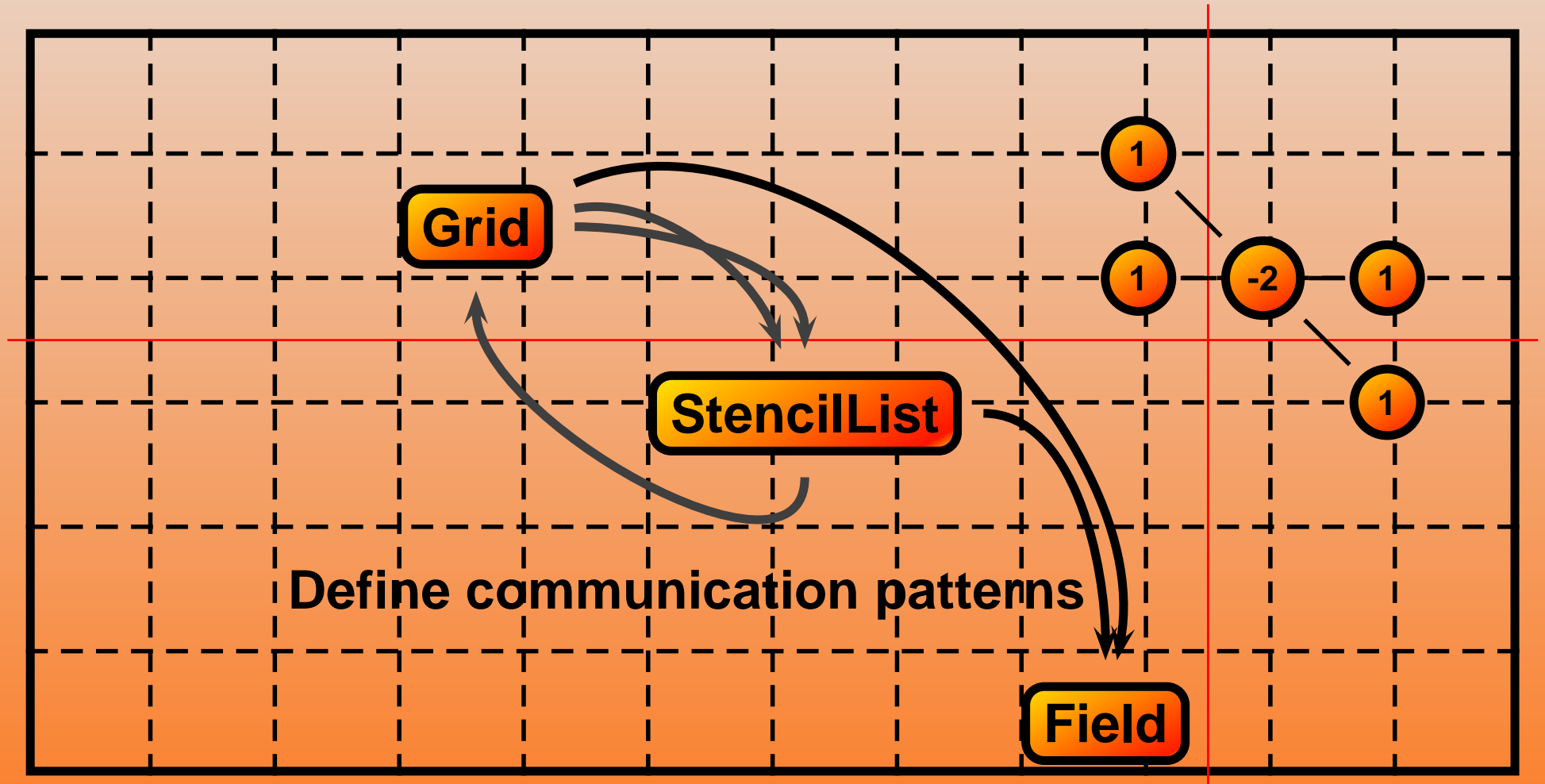
**Use StencilList to partition Grid, minimizing communication**

# Parallelization built into the components



**Modify where a stencil apply, according to partition**

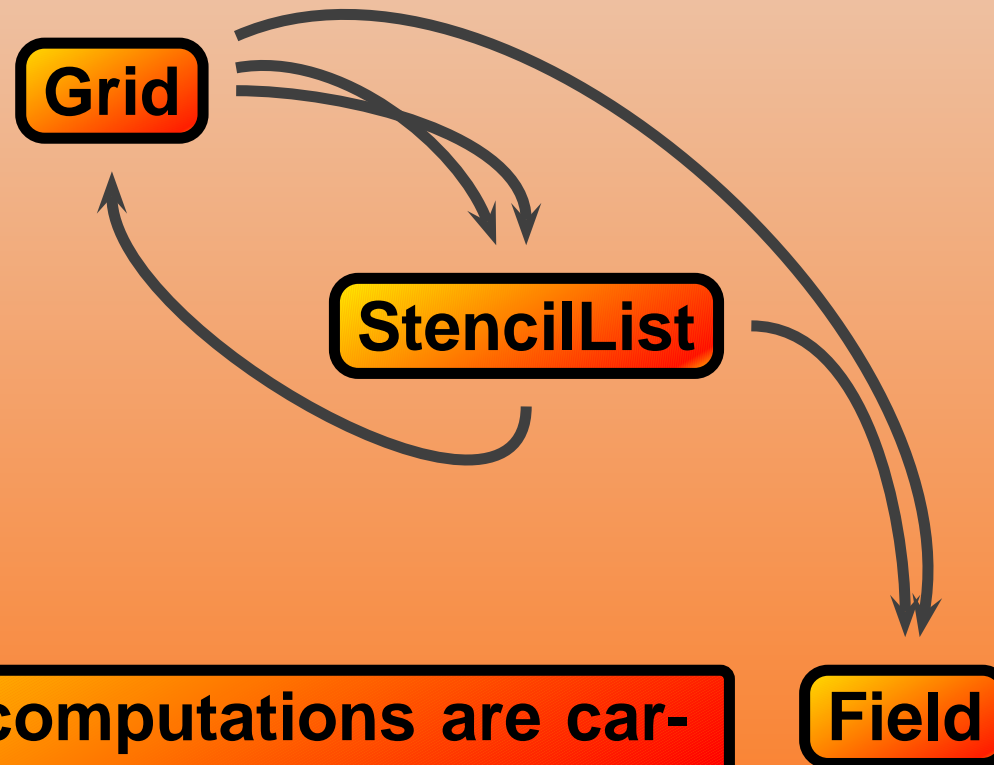
# Parallelization built into the components



Define communication patterns

Define communication-slices using both Grid and StencilList

# Parallelization built into the components



Further on, computations are carried out as in the scalar case.



## Use StencilList in implicit solvers

Let  $A$  be the stencil list  
– Above, we have seen:

$$u_{n+1} = A(u_n)$$

for an explicit step.



## Use StencilList in implicit solvers

Let  $A$  be the stencillist

– Above, we have seen:

$$u_{n+1} = A(u_n)$$

for an explicit step.

– An implicit method may require:

- $b - A * x$

- $(x, x)$

– We extend StencilList and Field with the necessary operations.



## Use StencilList in implicit solvers

- Let  $A$  be the stencillist
- Above, we have seen:

$$u_{n+1} = A(u_n)$$

for an explicit step.

- An implicit method may require:

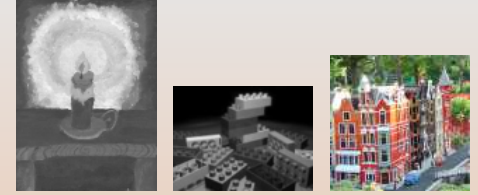
- $b - A * x$
- $(x, x)$

- We extend StencilList and Field with the necessary operations.

## Conjugate Gradients:

```
r = b - A*x
p = r.copy()
r0 = inner(r,r)
ri = r0
while ri/r0 > tolerance:
    w = A*p
    a = ri/inner(p,w)
    x = x + a*p
    r = r - a*w
    rt = inner(r,r)
    p = r + (rt/ri)*p
    ri = rt
```

## An example



Consider a simple Heat equation:

$$\begin{aligned}u_t &= \nabla^2 u & x \in \Omega \\u(x, 0) &= f(x), & x \in \Omega \\u_x(x, t) &= g(x, t), & x \in \partial\Omega\end{aligned}$$

Assume further that we want to solve this on the unit square with  $f$  and  $g$  given as initialfunc and neuman-ncond, respectively.



## An example

```
g = Grid(([0.0,1.0], [0.0,1.0]), (100,100))  
u = Field(g)  
t = 0; dt = T/n;  
sl = StencilList(g)  
innerstencil = Identity(g) + dt*Laplace(g)  
innerind = sl.addStencil(innerstencil, g.innerPoints())  
sl += NeumannBoundary.create(sl[innerind], g, neumanncond)  
u.fill(initialfunc)  
for t < T:  
    u = sl(u)  
    t += dt
```



## An example

```
g = Grid(([0.0,1.0], [0.0,1.0]), (100,100))
u = Field(g)
t = 0; dt = T/n;
sl = StencilList(g)
innerstencil = Identity(g) + dt*Laplace(g)
innerind = sl.addStencil(innerstencil, g.innerPoints())
sl += NeumannBoundary.create(sl[innerind], g, neumanncond)
u.fill(initialfunc)
for t < T:
    u = sl(u)
    t += dt
```

**Questions!**

<http://www.simula.no/~aasmundo/drscient/CSE05.pdf>

# Conclusions

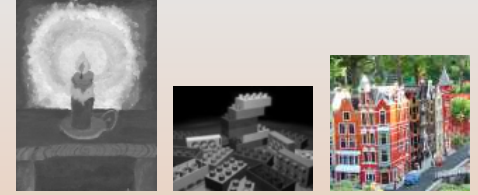


**We have created a framework suitable for experimenting with FDM.**

**The syntax is compact.**

**It can be used both interactively in a “matlab” kind of style, or you can create solver classes utilizing the presented abstractions.**

## Some Future steps



**More parallelization!**

**More geometry!**

**More focus on performance, e.g. vectorization of operators, implementing c or fortran modules for certain operations.**

An important issue: How to traverse the grid?





## An important issue: How to traverse the grid?

**Our solution: Use iterators!**

- **Benefit:** We don't need to create list for all indices in memory.

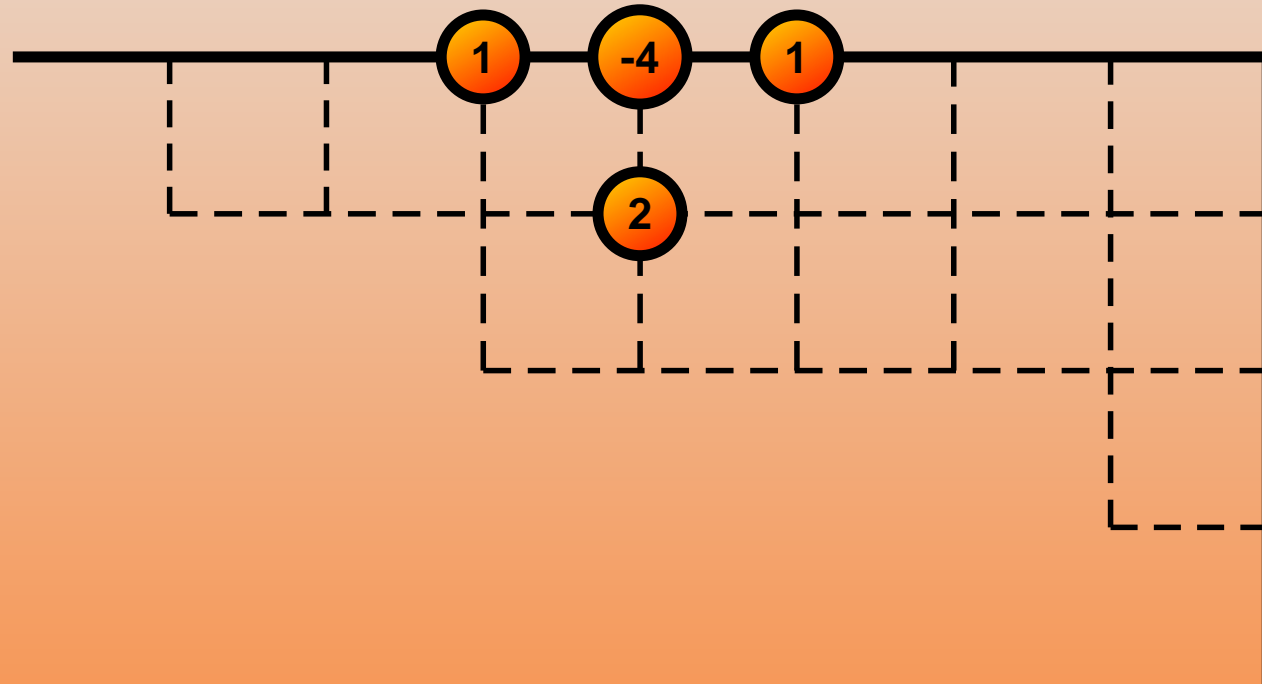
- **Drawback:** speed

- **Simplest case:**  $(1,1,\dots)$  -  $(n-1,m-1,\dots)$ , which represent all inner nodes in the grid.

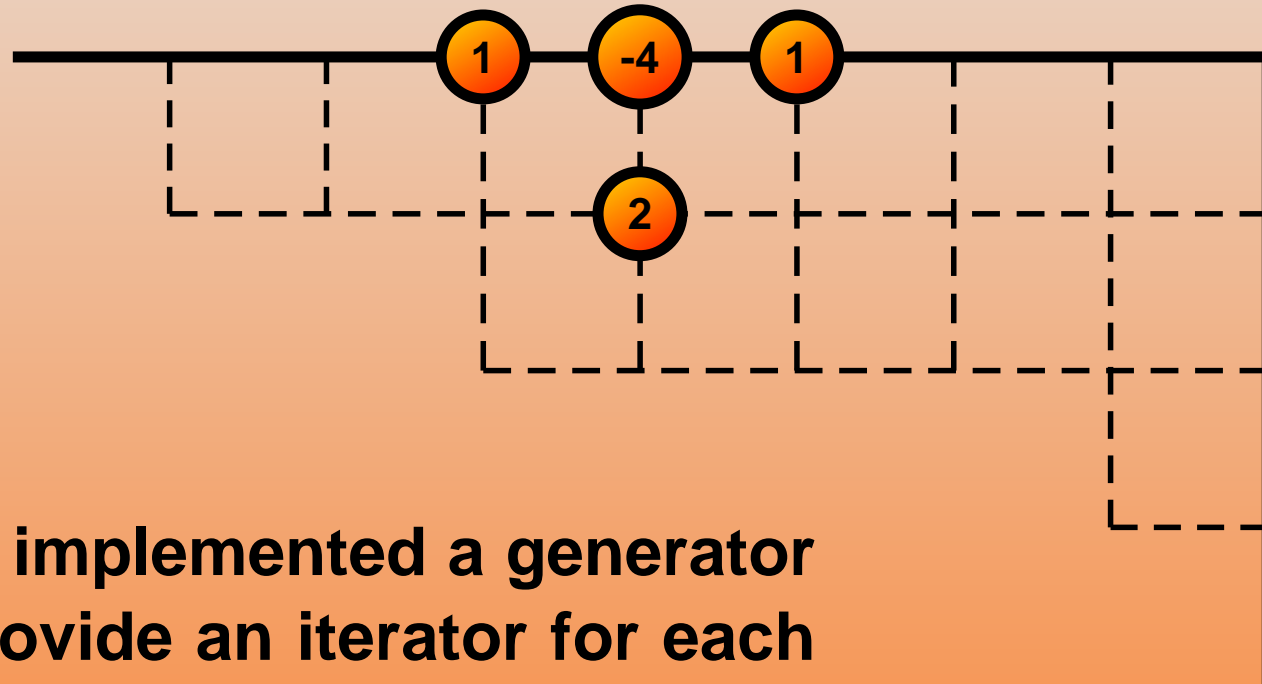
Can also be done with list-comprehension (but that will produce the list in memory)

– *<code>: simpletuple.py*

# Part of the boundary with a Neumann-condition



## Part of the boundary with a Neumann-condition



We have implemented a generator which provide an iterator for each part of the boundary, with necessary auxiliary information.

– `<code>`: *advancedtuple.py*