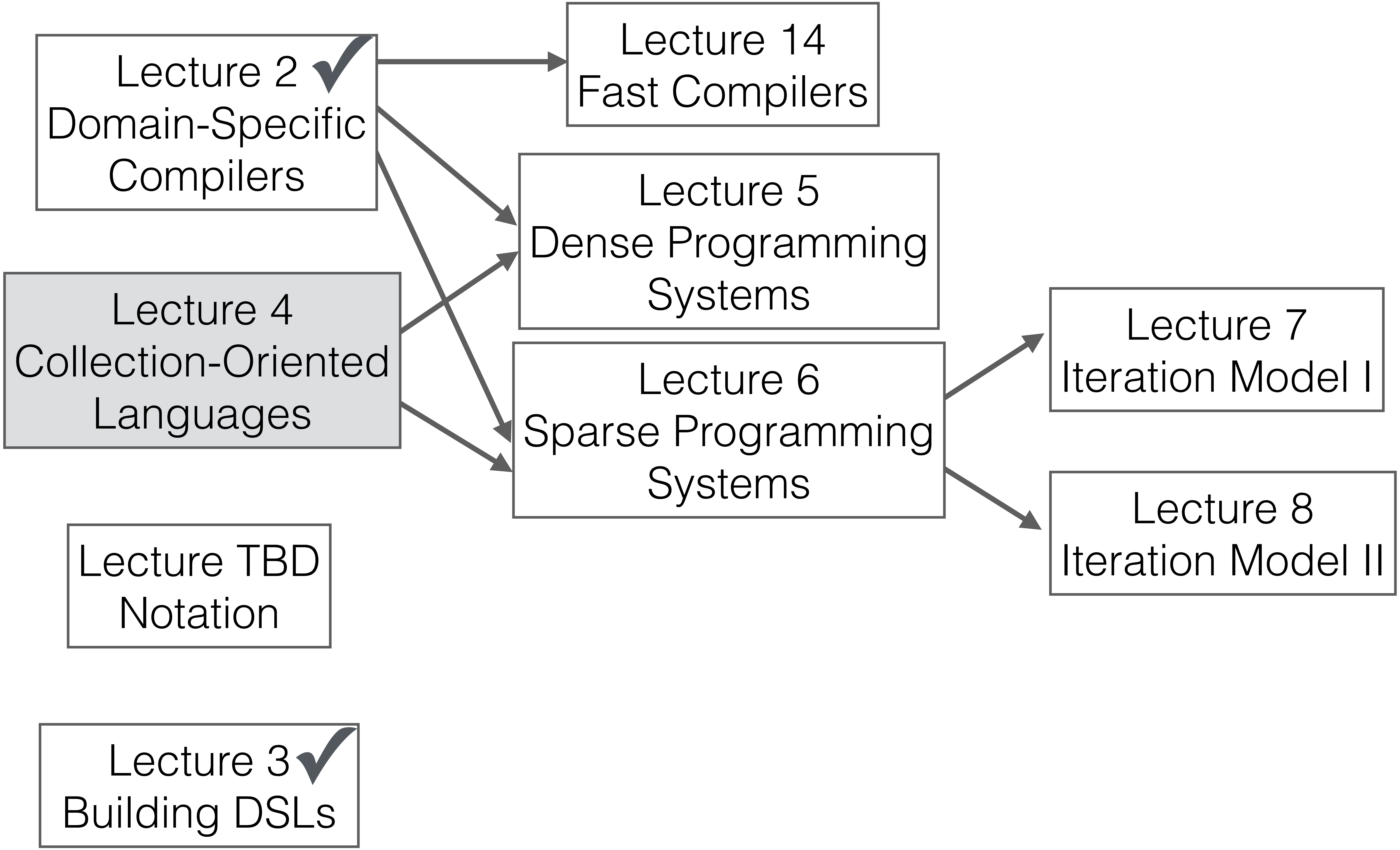
Lecture 4 — Collection-Oriented Languages

Stanford CS343D (Winter 2023)
Fred Kjolstad

Lecture Overview

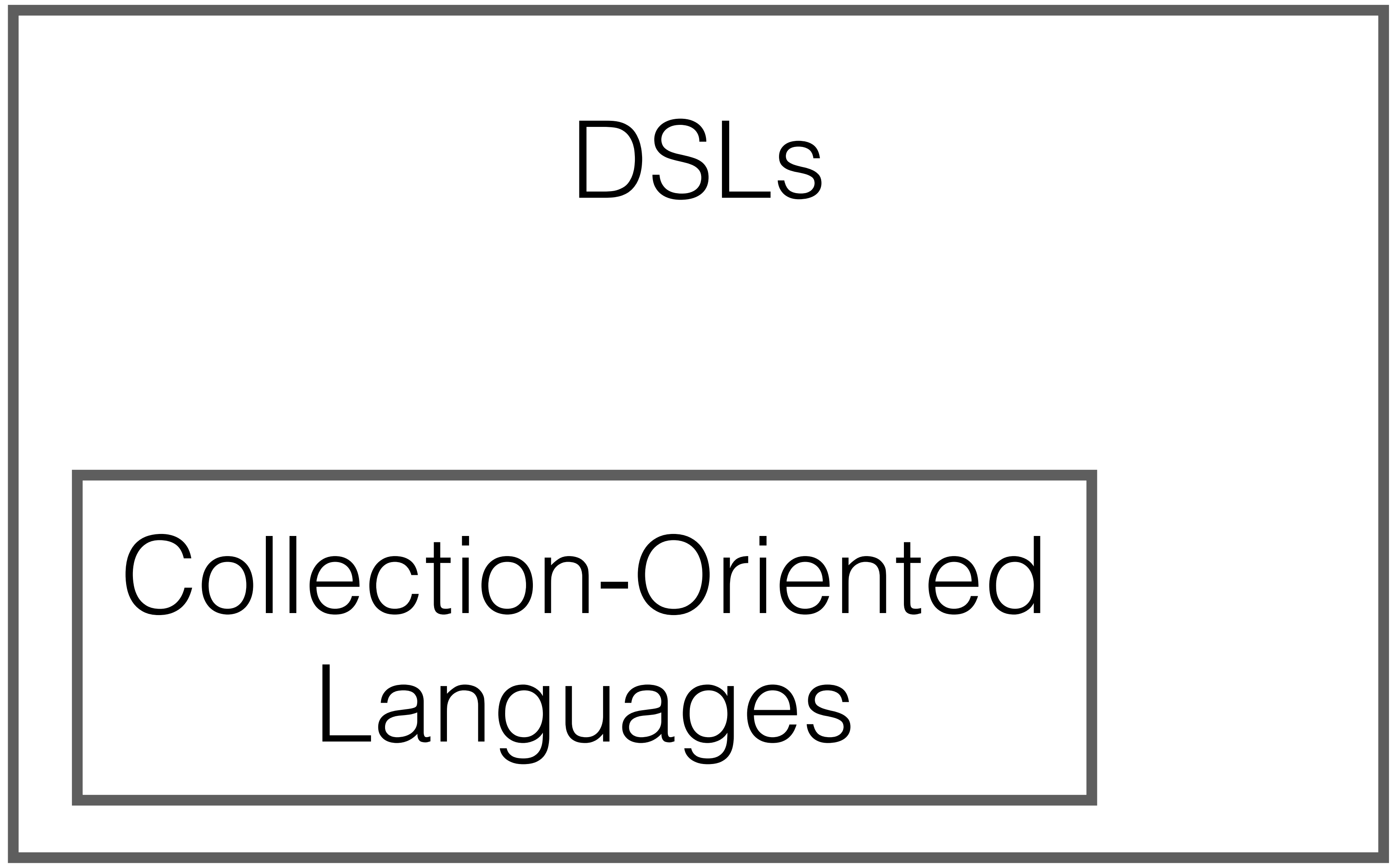


Languages are tools for thought

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on the more advanced problems, and in effect increases the mental power of the race."

— Alfred N. Whitehead

Collection-Oriented languages are an important subclass of DSLs as discussed in this course



Economy of scale

in notation and execution

$$C = A \bowtie B$$

$$c = Ab$$

```
[x * 2 for x in my_list]
```

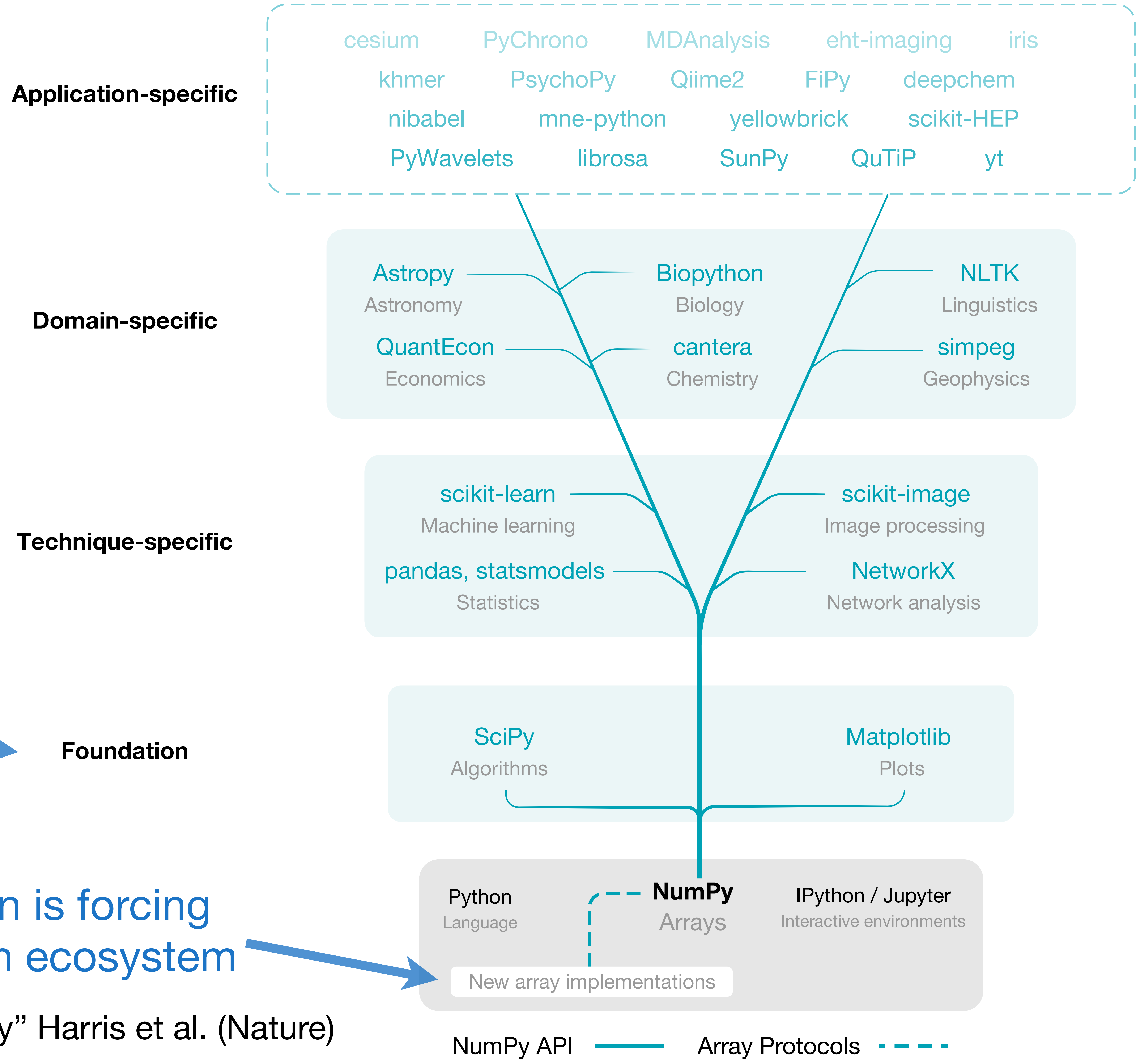
How many operations?

Collection-oriented languages are relatively general

DSLs

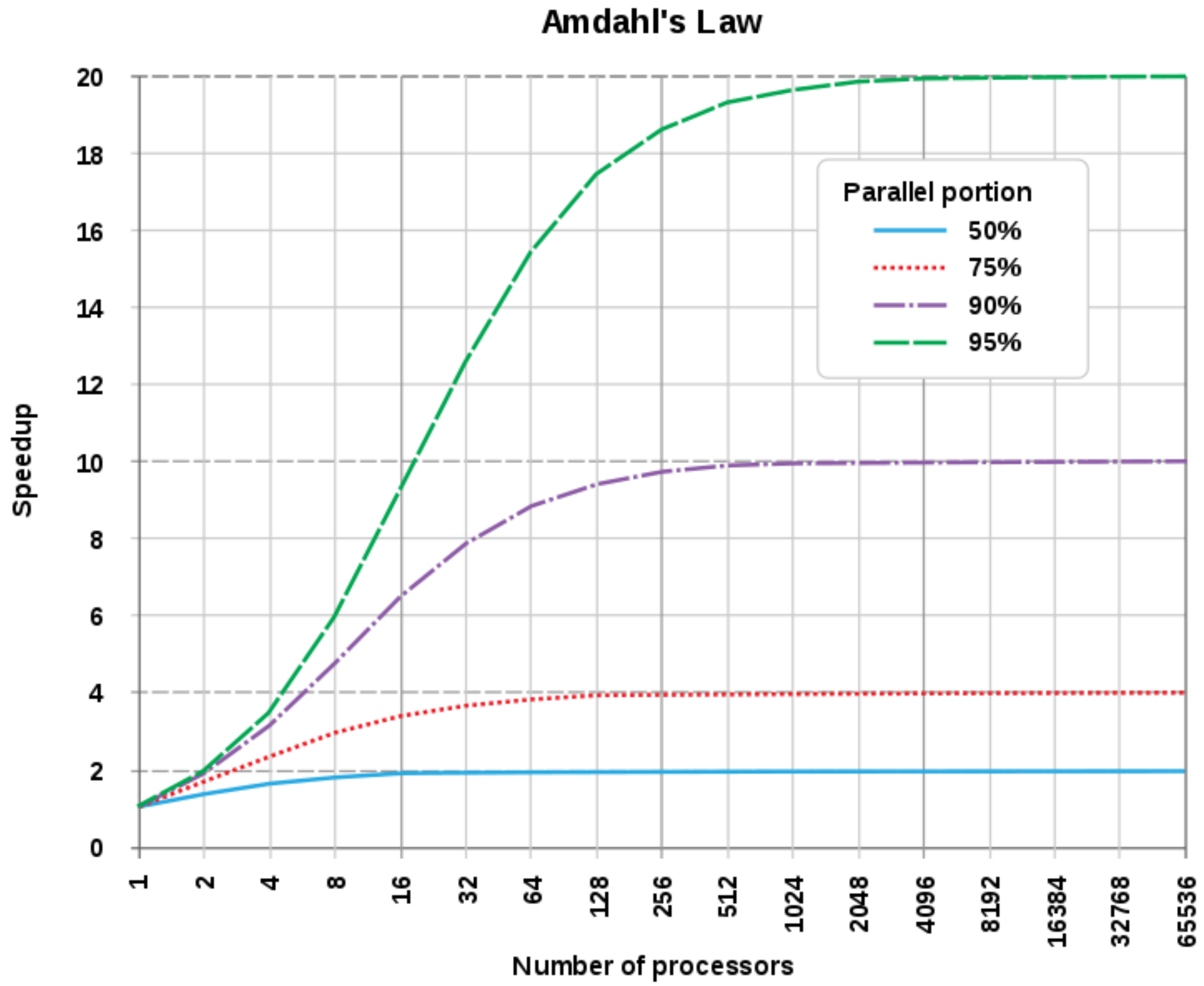
Collection-Oriented
(also DSLs?)

Fragmentation is forcing
implementation ecosystem

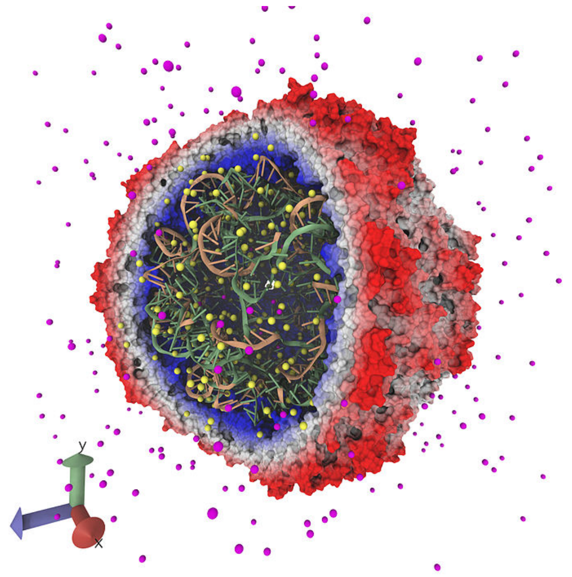


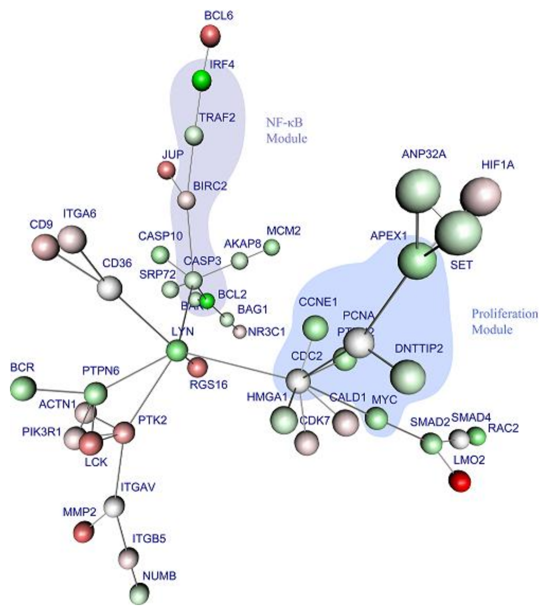
“Array Programming with NumPy” Harris et al. (Nature)

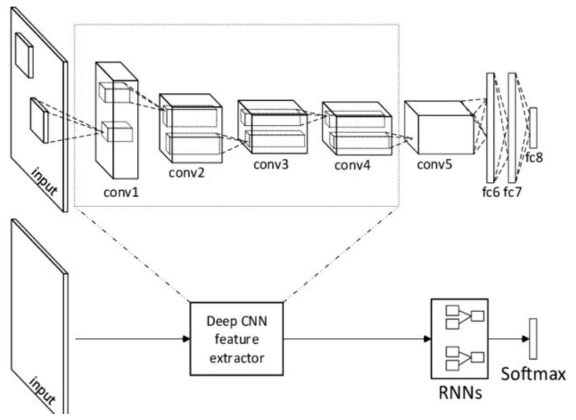
We need collections for performance due to Amdahl's law

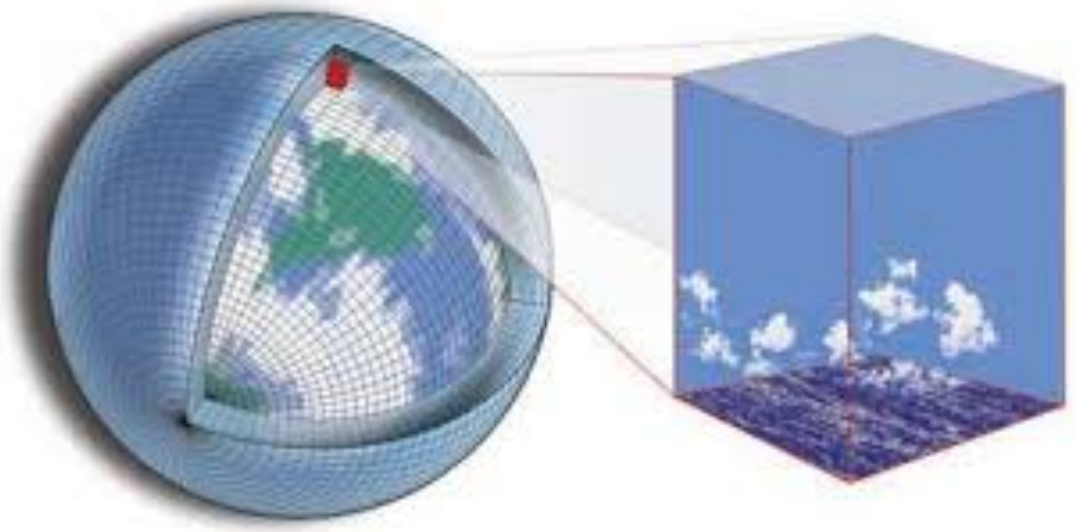


But many applications are data-rich









Plot from Wikipedia

Avoiding the von Neumann model of languages

Imperative Form

```
c := 0
for i := 1 step 1 until n do
  c := c + a[i] x b[i]
```

poor world of statements

transfers one scalar value to memory:
von Neumann bottleneck in software
the assignment transfers one value to memory

rich world of expressions

Functional Form

```
c = sum(a[0:n]*b[0:n])
```

produces a vector

Collection-oriented operations let us operate on collections as a whole

- A record-at-a-time user interface forces the programmer to do manual query optimization, and this is often hard.
- Set-a-time languages are good, regardless of the data model, since they offer much improved physical data independence.
- The programming language community has long recognized that aggregate data structures and general operations on them give great flexibility to programmers and language implementors.

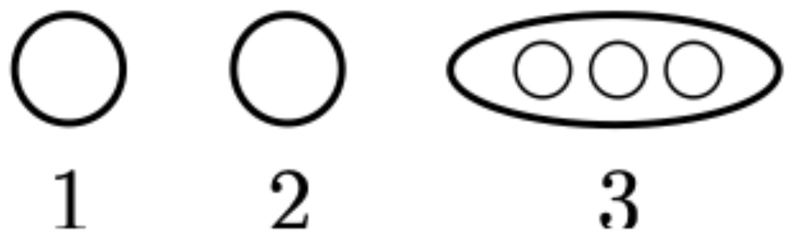
“What Goes Around Comes Around” Stonebraker and Hellerstein
“Collection-Oriented Languages” Sipelstein and Blelloch

Collection-Oriented Languages

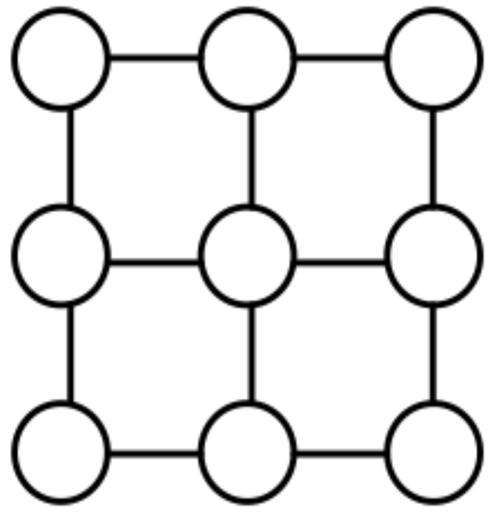
Lists
Lisp [1958]

Sets
SETL
[1970]

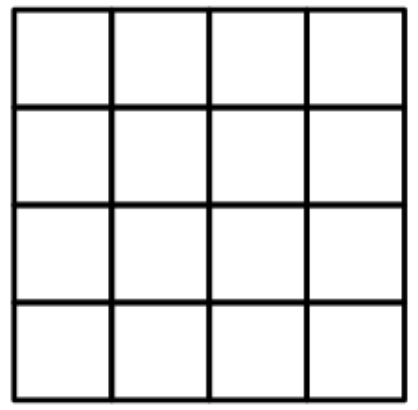
Nested Sequences
NESL [1994]



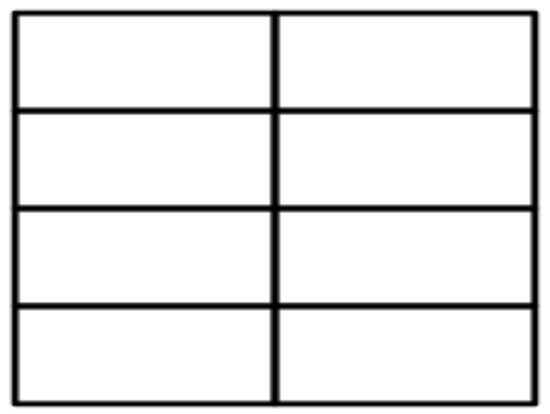
Grids
Halide [2012]



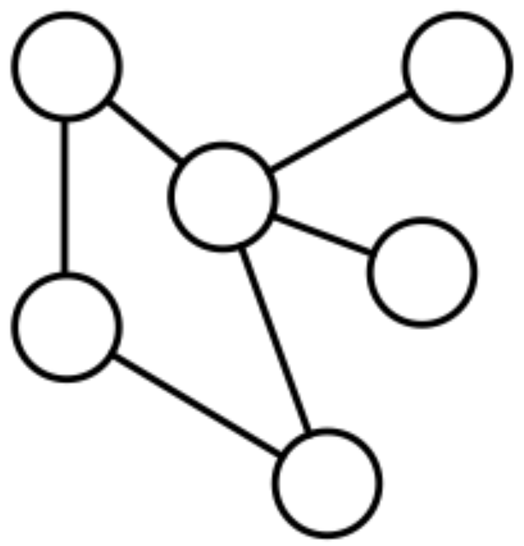
Arrays
APL [1962]
NumPy [2020]



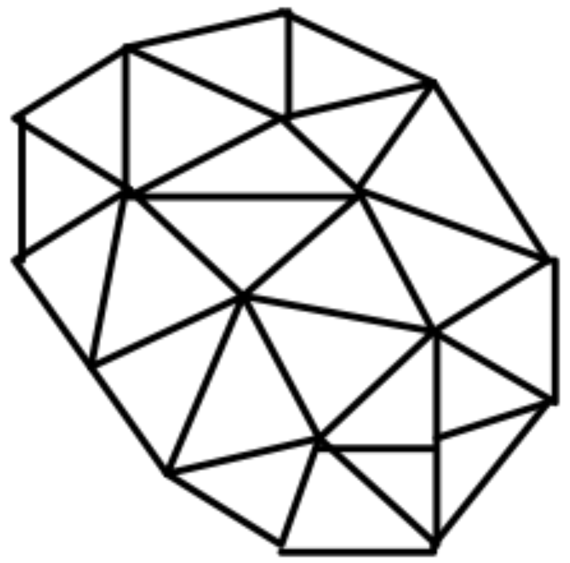
Relations
Relational Algebra [1970]



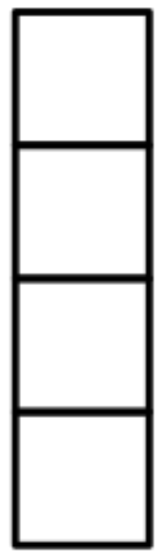
Graphs
GraphLab [2010]



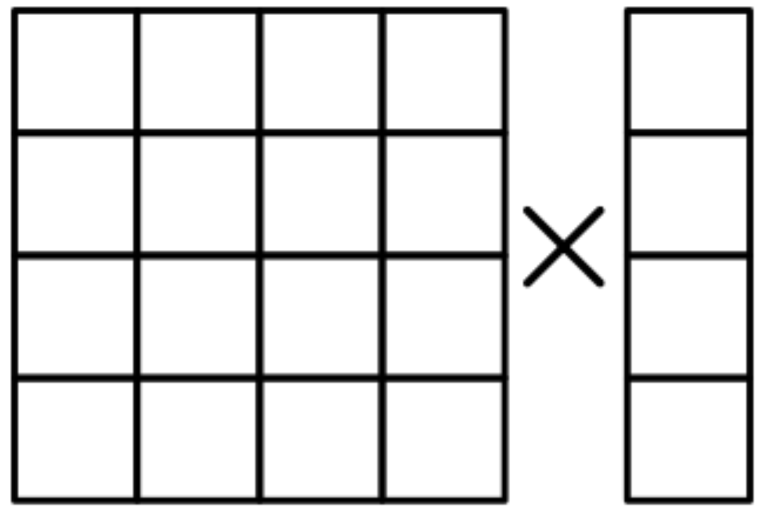
Meshes
2011



Vectors
Vector Model 1990



Matrices and Tensors
Matlab 1979], Taco 2017



A collection-oriented programming model provides collective operations on some collection/abstract data structure

Objects Orientation vs.
Collection Orientation

Features of collections

- Ordering: unordered, sequence, or grid-ordered?
- Regularity: Can the collection represent irregularity/sparsity?
- Nesting: nested or flat collections?
- Random-access: can individual elements be accessed?

The APL Programming Language

```
n ← 4 5 6 7
```

i.e., mkArray

```
n+4
8 9 10 11
```

4 is broadcast across each n

```
n+ι4
5 7 9 11
```

element-wise addition
(ι4 makes the array [1,2,3,4])

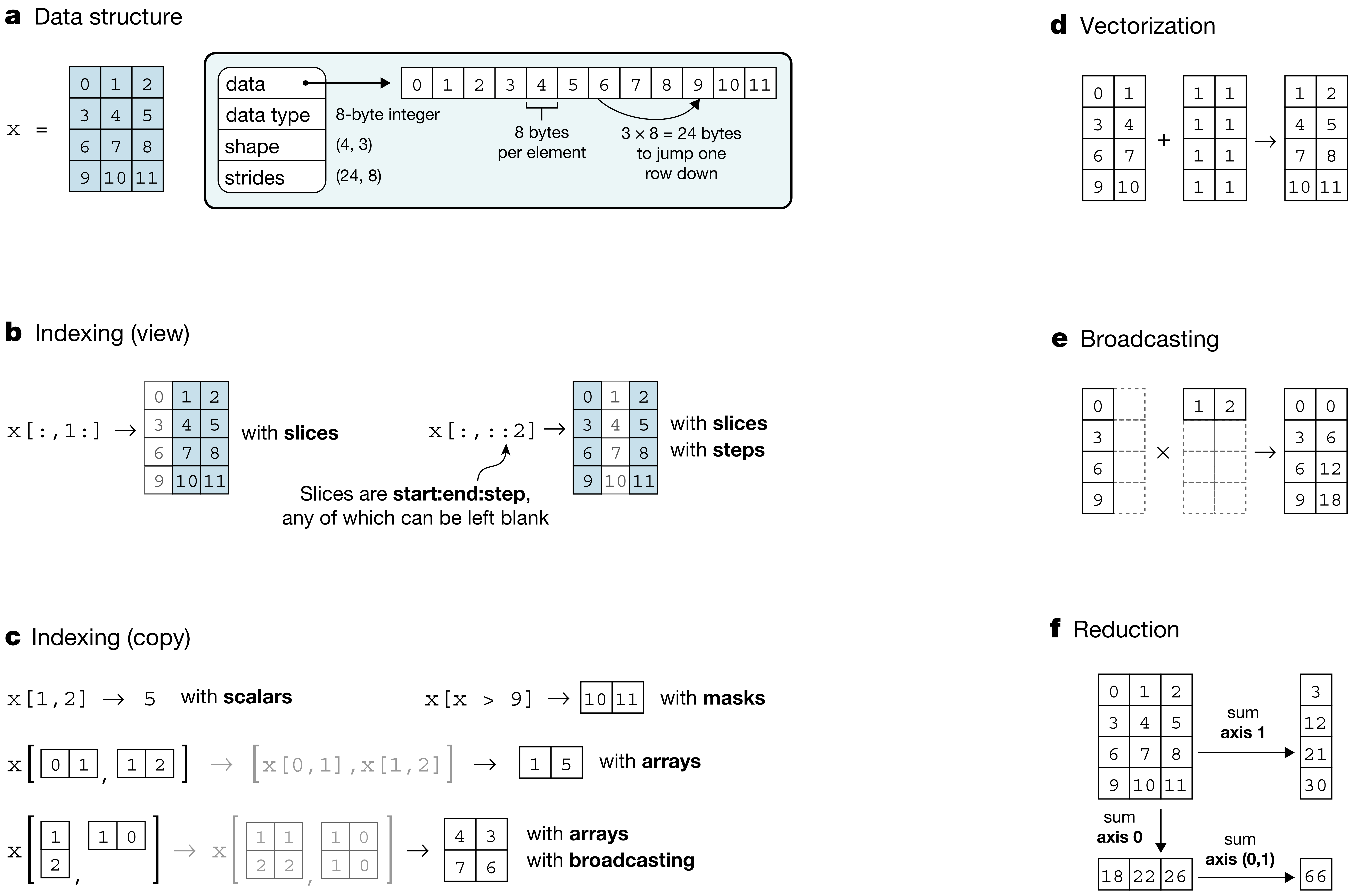
```
+/n
22
```

$$\sum_{i=0}^{n} n_i$$

```
+/(3+ι4)
22
```

$$\sum_{i=1}^{4} (i+3)$$

Array Programming with NumPy



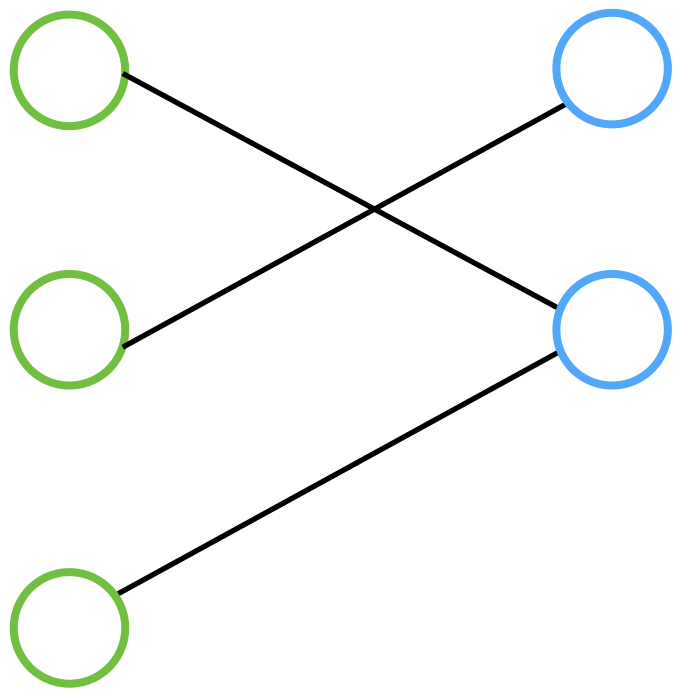
“Array Programming with NumPy” Harris et al. (Nature)

The SETL Language

Sets

Tuples

Functions



SETL Set Former Notation

Notation	Example
$\{x \in s \mid C(x)\}$	$\{x \in \{1,5,10,32\} \mid x \textbf{ lt } 10\} \rightarrow \{1,5\}$
$\{e(x), x \in s \mid C(x)\}$	$\{i * i, i \in \{1,3,5\}\} \rightarrow \{1,9,25\}$
$\{e(x), \text{min} \leq i \leq \text{max} \mid C(i)\}$	$\{i * 2 - 1, 1 \leq i \leq 5\} \rightarrow \{1,3,5,7,9\}$
$[\text{op} : x \in s \mid C(x)]e(x)$	$[+ : x \in \{1,2,3\}](x * x) \rightarrow 14$
$\forall x \in s \mid C(x)$	$\forall x \in 1,2,4 \mid (x//2) \textbf{ eq } 1 \rightarrow \textbf{f}$
$[+ : x \in s_1, y \in s_2]\{<x, y>\}$	$[+ : x \in \{1,2\}, y \in \{a, b\}]\{<x, y>\} \rightarrow$ $\{<1,a>, <1,b>, <2,a>, <2,b>\}$

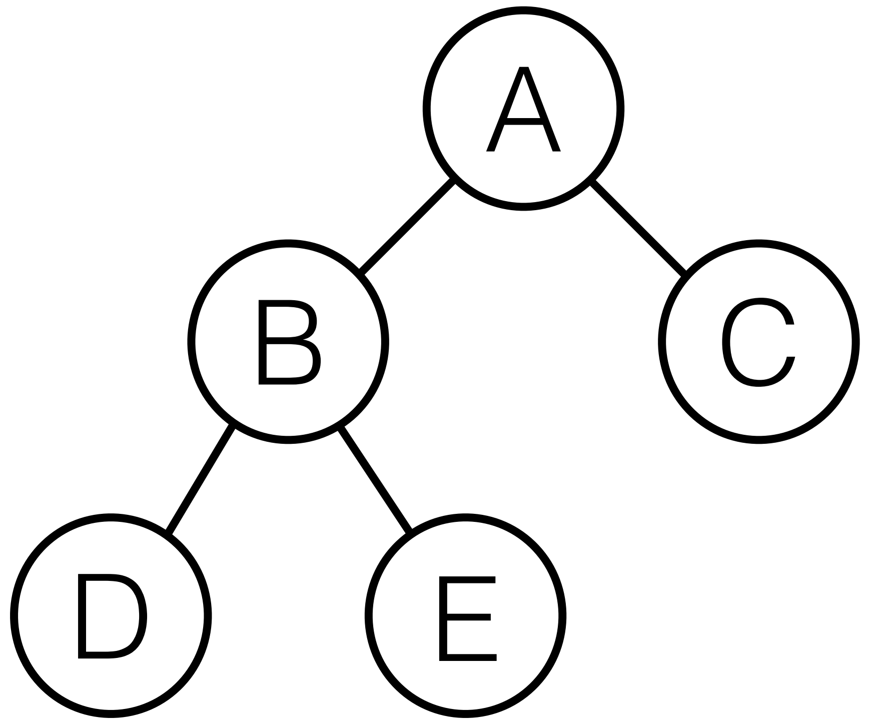
Standard set operations like union, intersection, and set difference are also supported

SETL Table Functions

$$f = \{ < 1,1 > , < 2,4 > , < 3,9 > \}$$

$$f(2) \rightarrow 4$$

$$f + \{ < 2,5 > \} \rightarrow \{ < 1,1 > , < 2,5 > , < 3,9 > \}$$



left = $\{ < A, B > , < B, D > \}$

right = $\{ < A, C > , < B, E > \}$

Relational Algebra

employees

name	id	department
Harry	3245	CS
Sally	7264	EE
George	1379	CS
Mary	1733	ME
Rita	2357	CS

departments

department	manager
CS	George
EE	Mary

Projection (Π)

$\Pi_{name,department}$ employees

Name	Department
Harry	CS
Sally	EE
George	CS
Mary	ME
Rita	CS

Selection (σ)

$\sigma_{department=CS}$ employees

Name	ID	Department
Harry	3245	CS
George	1379	CS
Rita	2357	CS

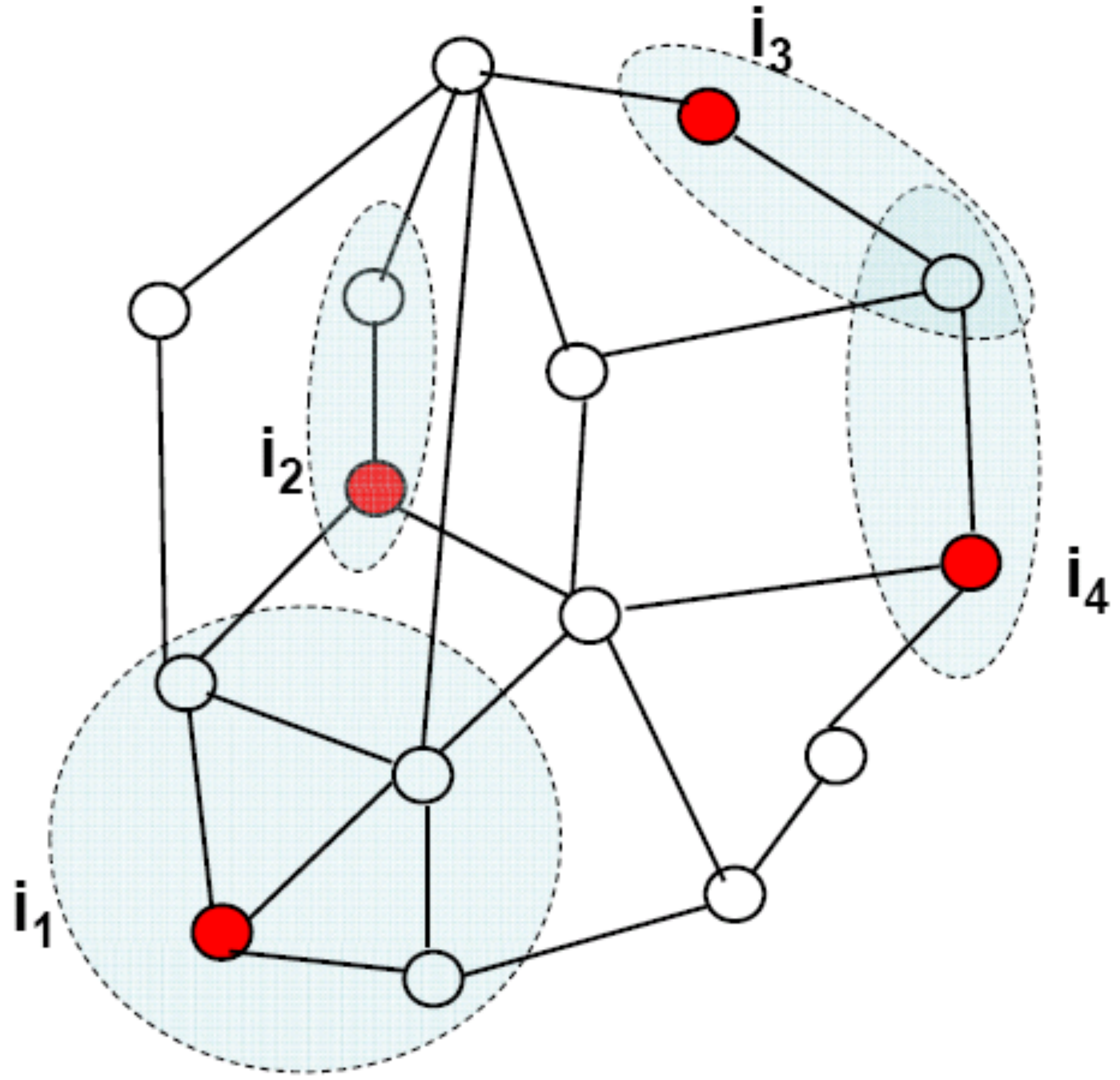
Natural join ($\bowtie$)

employees $\bowtie$ departments

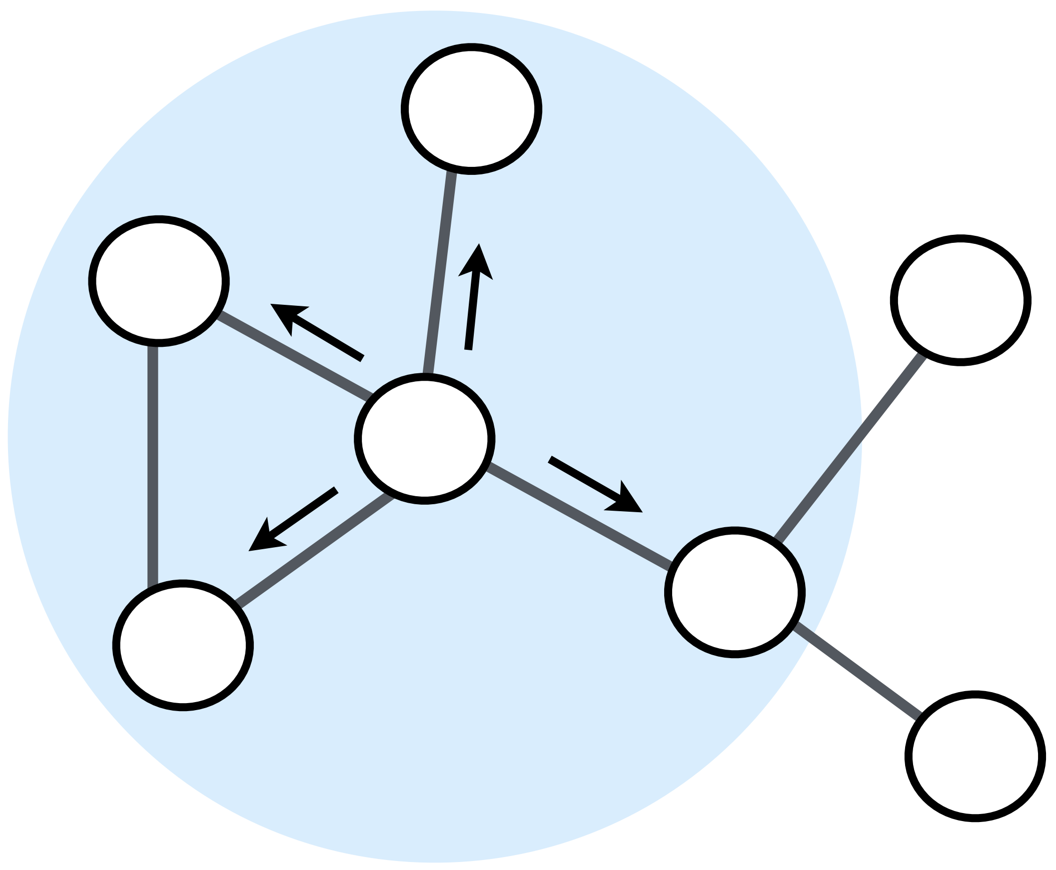
name	id	department	manager
Harry	3245	CS	George
Sally	7264	EE	Mary
George	1379	CS	George
Rita	2357	CS	George

Graph operations

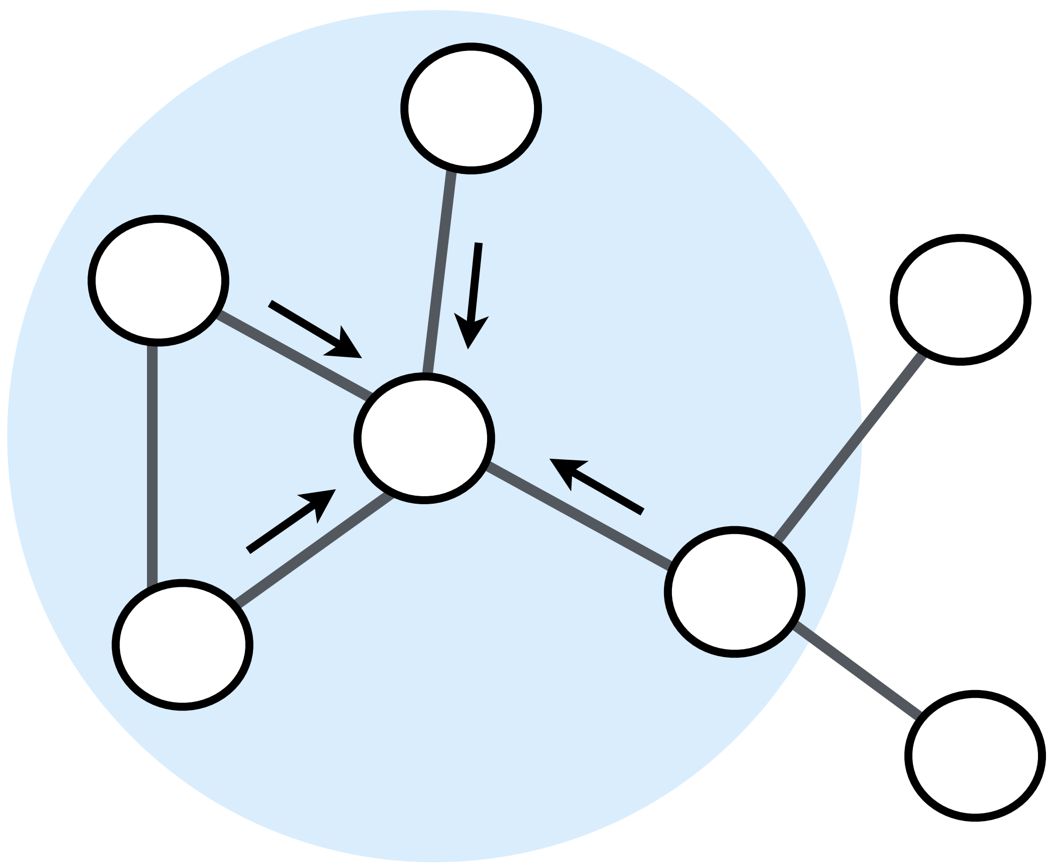
Simultaneous operations on different parts of the graph



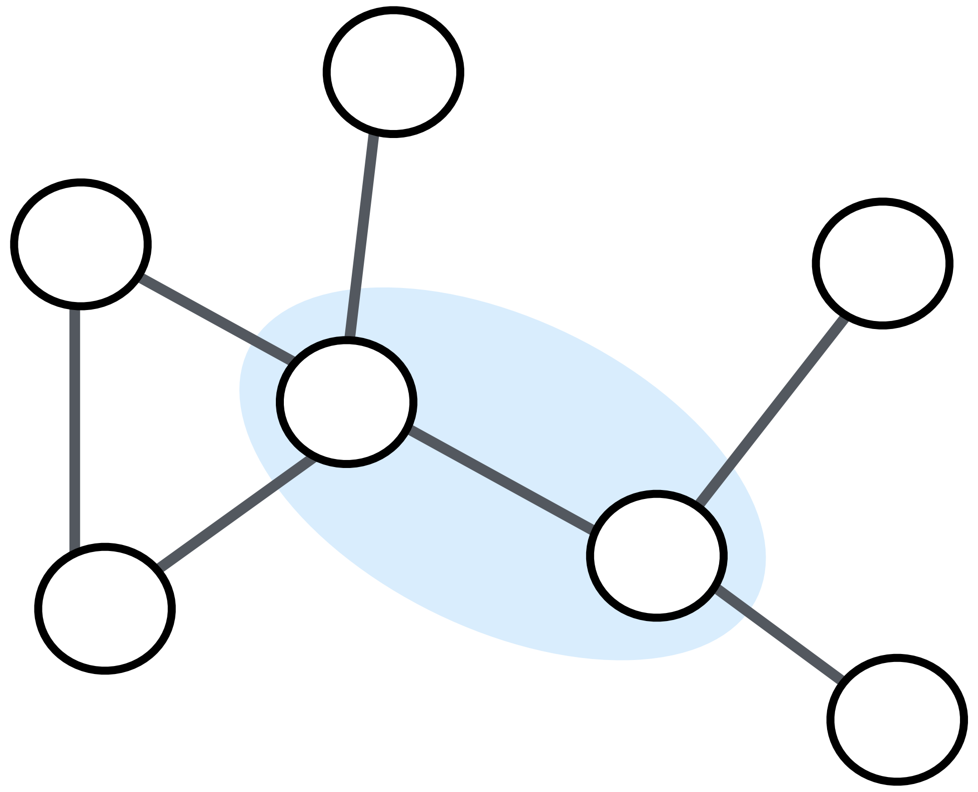
push



pull



edge functions



Left figure from Tao of Parallelism in Algorithms (Pingali et al.)

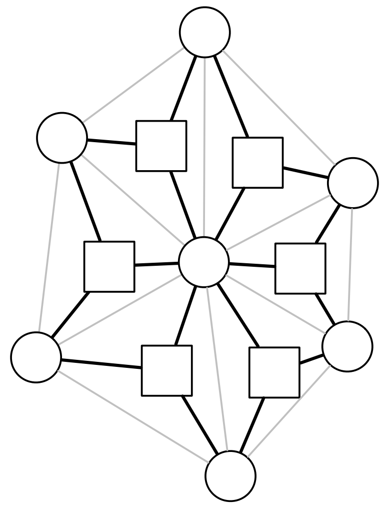
Relations, Graphs, and Algebra: No glove fits all

Relations

Names	City	Age
Peter	Boston	54
Mary	San Fransisco	35
Paul	New York	23
Adam	Seattle	84
Hilde	Boston	19
Bob	Chicago	76
Sam	Portland	32
Angela	Los Angeles	62

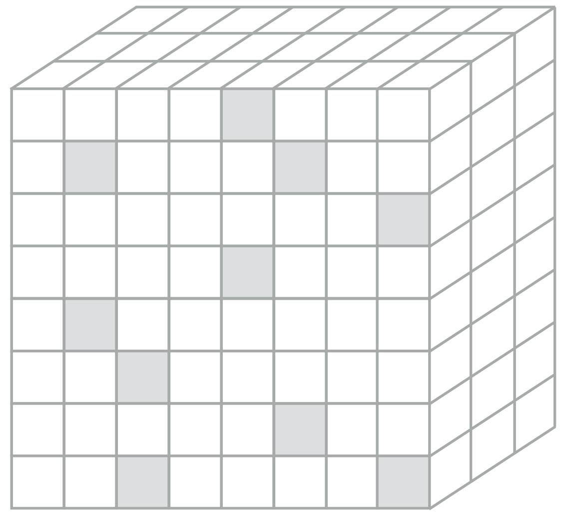
Ideal for combining data to form systems

Graphs



Ideal for local operations

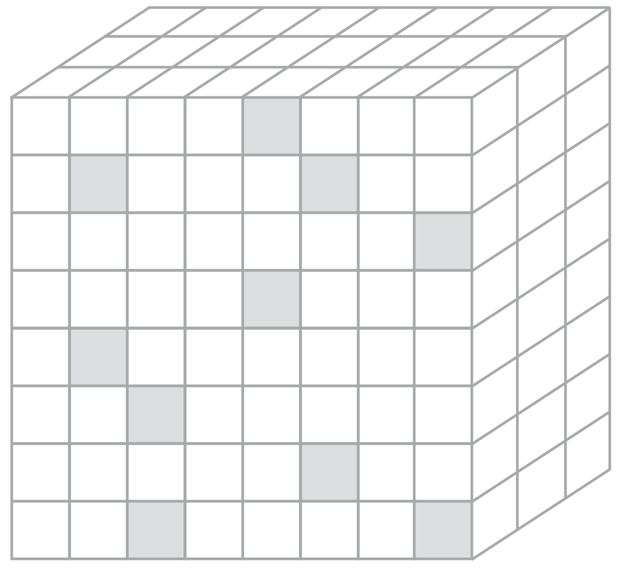
Tensors



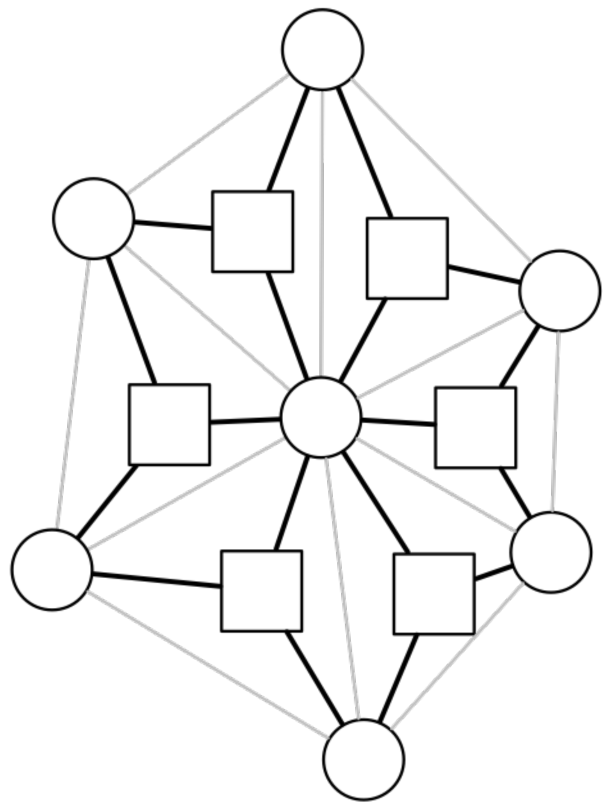
Ideal for global operations

It is critical to be able to compose languages and abstractions

Tensors



Graphs



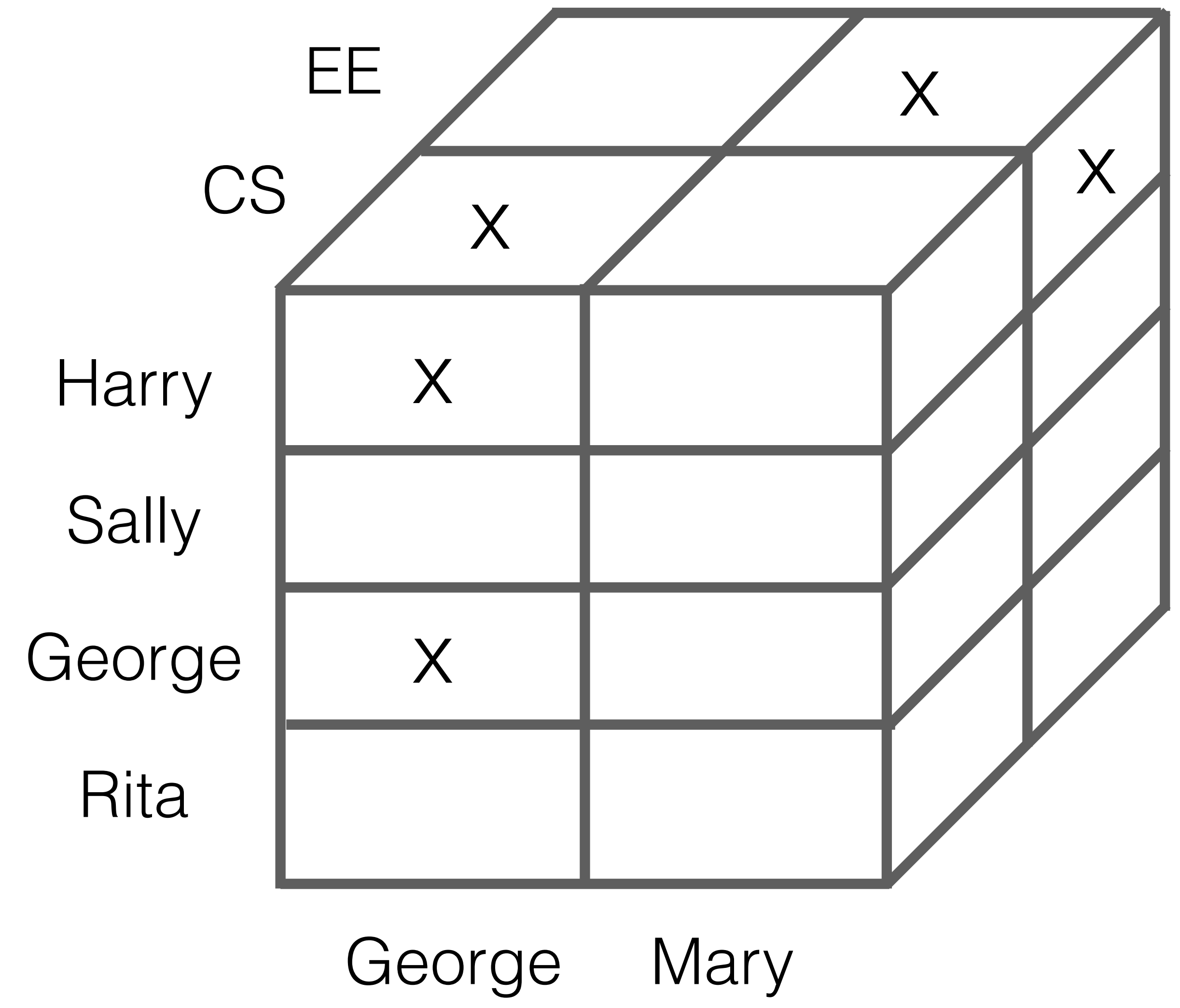
Relations

Names	City	Age
Peter	Boston	54
Mary	San Fransisco	35
Paul	New York	23
Adam	Seattle	84
Hilde	Boston	19
Bob	Chicago	76
Sam	Portland	32
Angela	Los Angeles	62

Example: Relations and Tensors

name	department	manager
Harry	CS	George
Sally	EE	Mary
George	CS	George
Rita	CS	George

Tensor Assembly

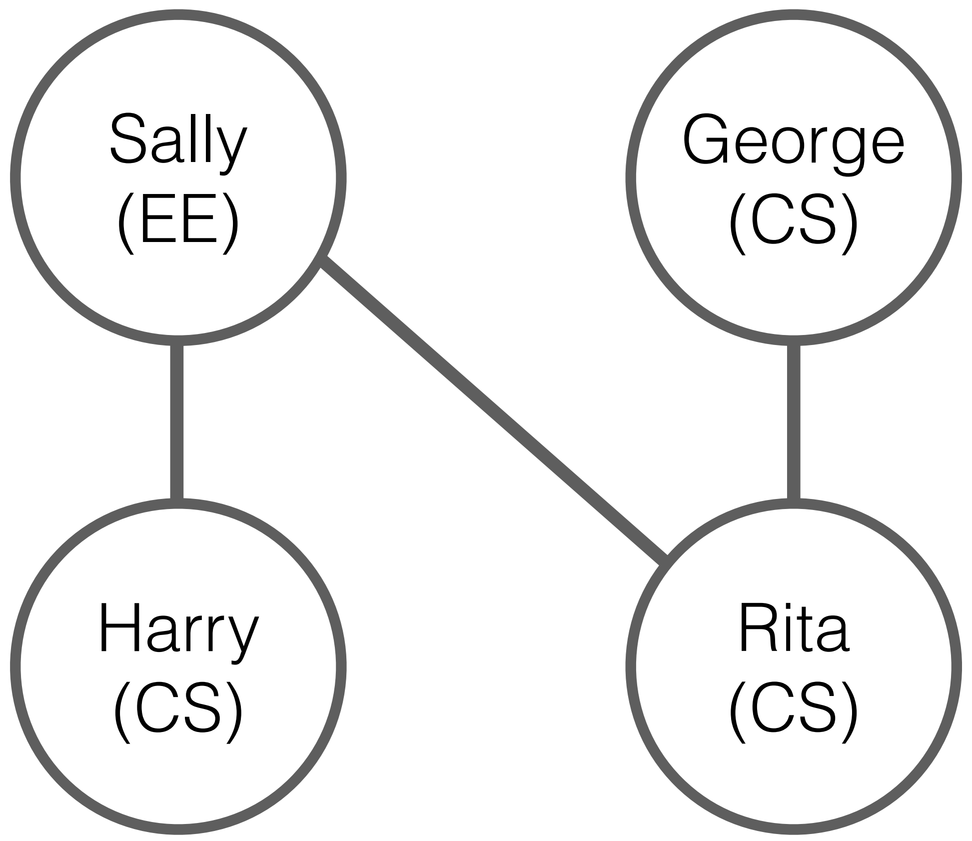


Example: Relations and Graphs

name	department
Harry	CS
Sally	EE
George	CS
Rita	CS

$\bowtie$

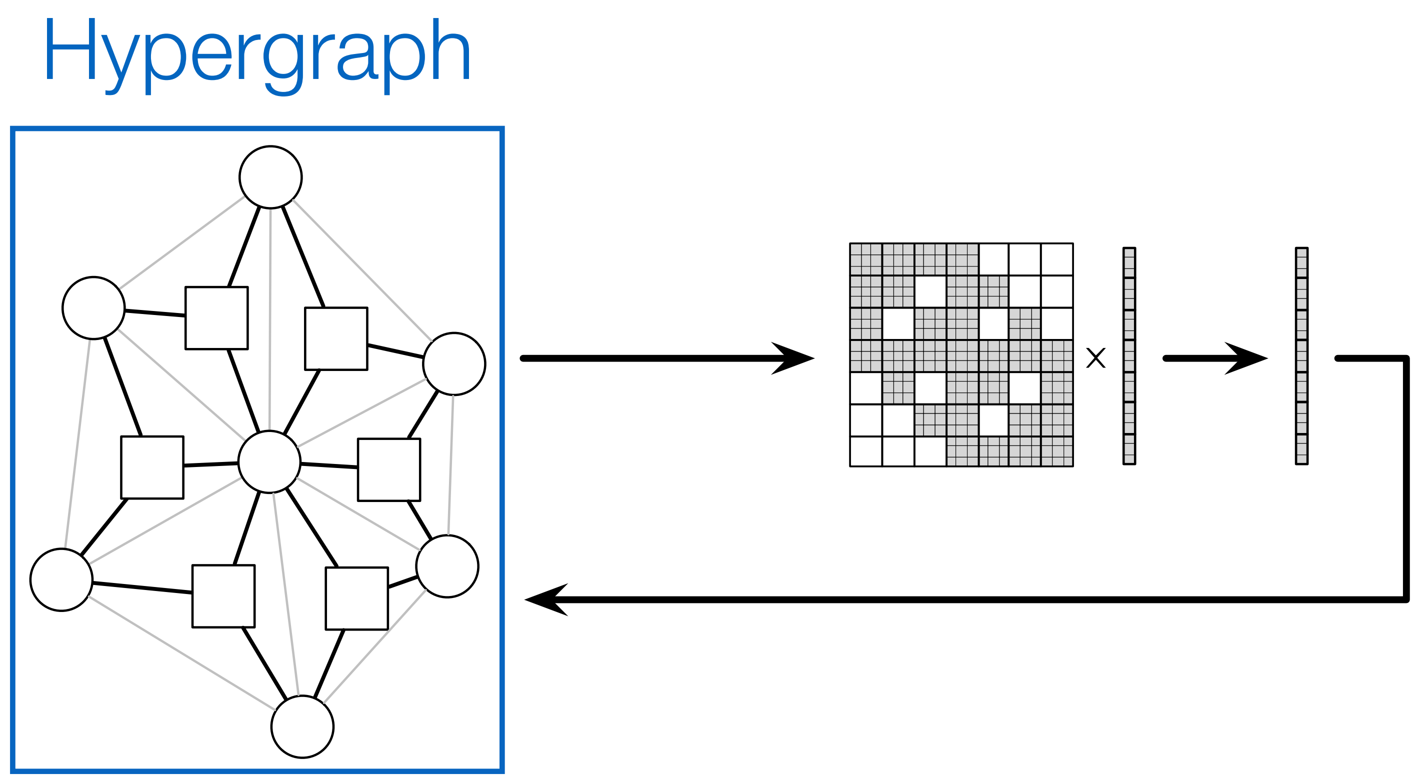
name1	name2
Harry	Sally
Sally	Harry
George	Rita
Rita	George
Sally	Rita
Rita	Sally



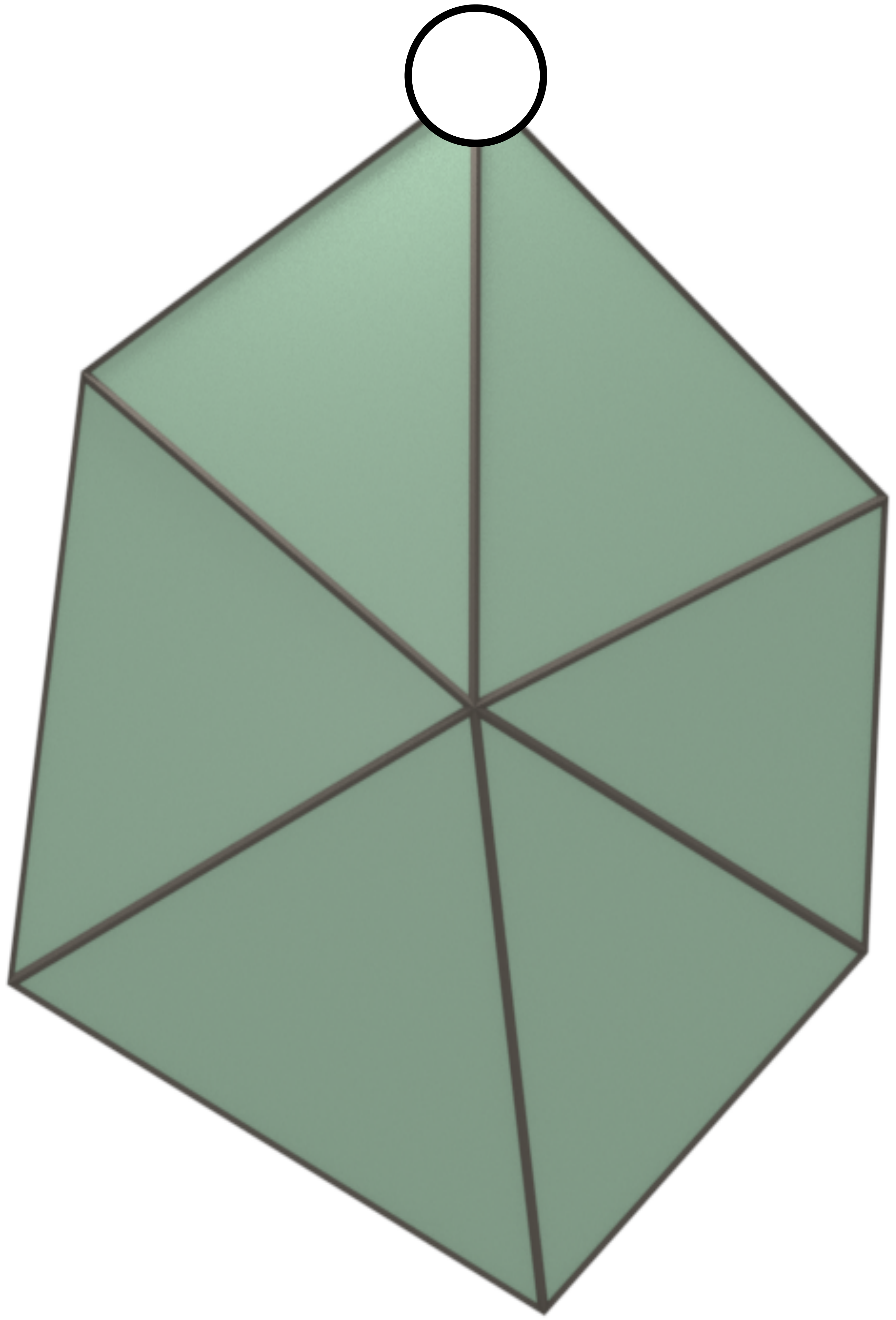
Example: Graphs and Tensors (Simit)

Statics Tetrahedral Neo-Hookean FEM Simulation

Statics Triangular Neo-Hookean FEM Simulation

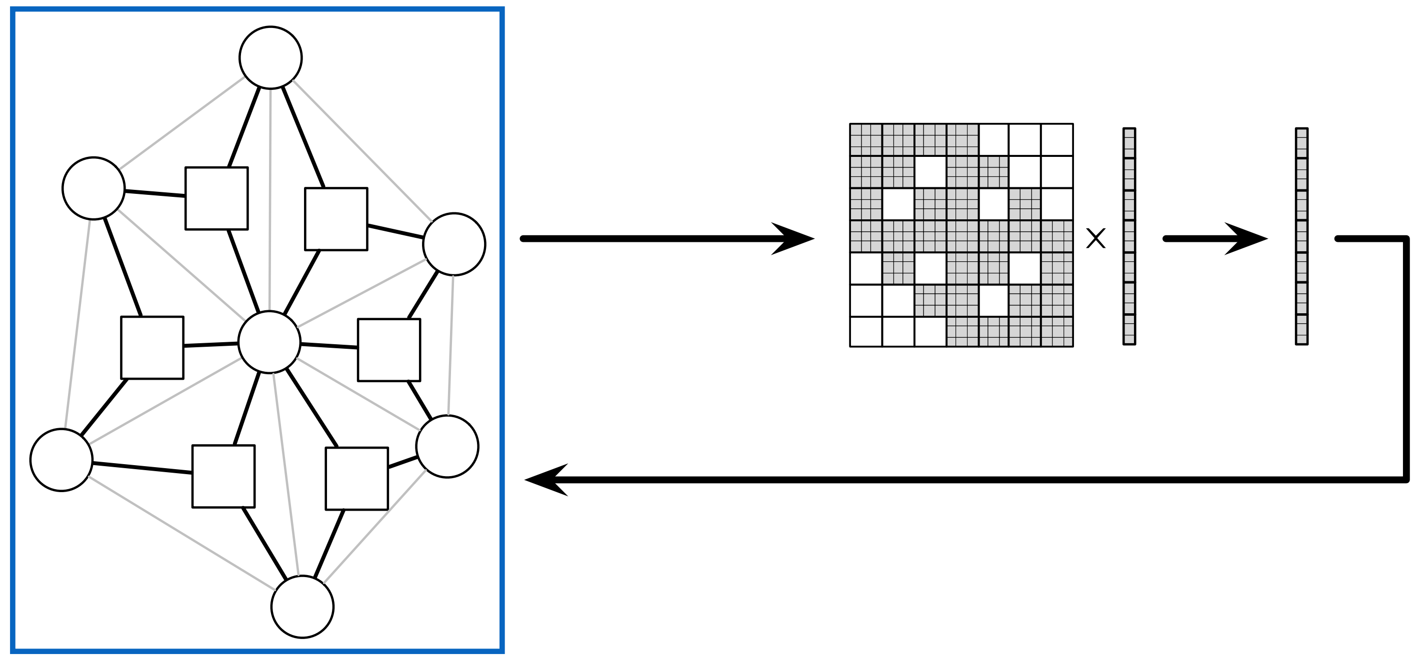



```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end
```

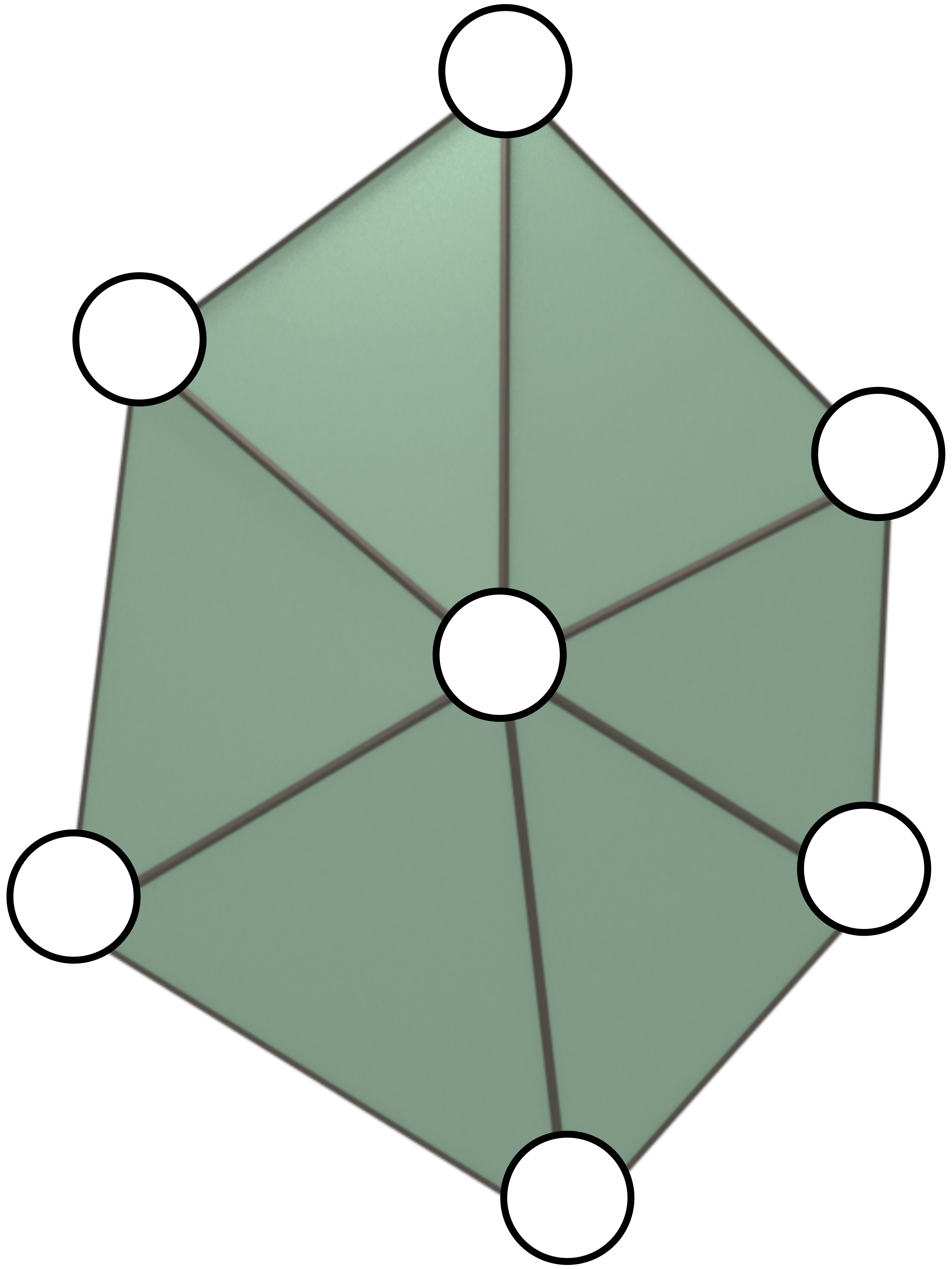


Statics Triangular Neo-Hookean FEM Simulation

Hypergraph

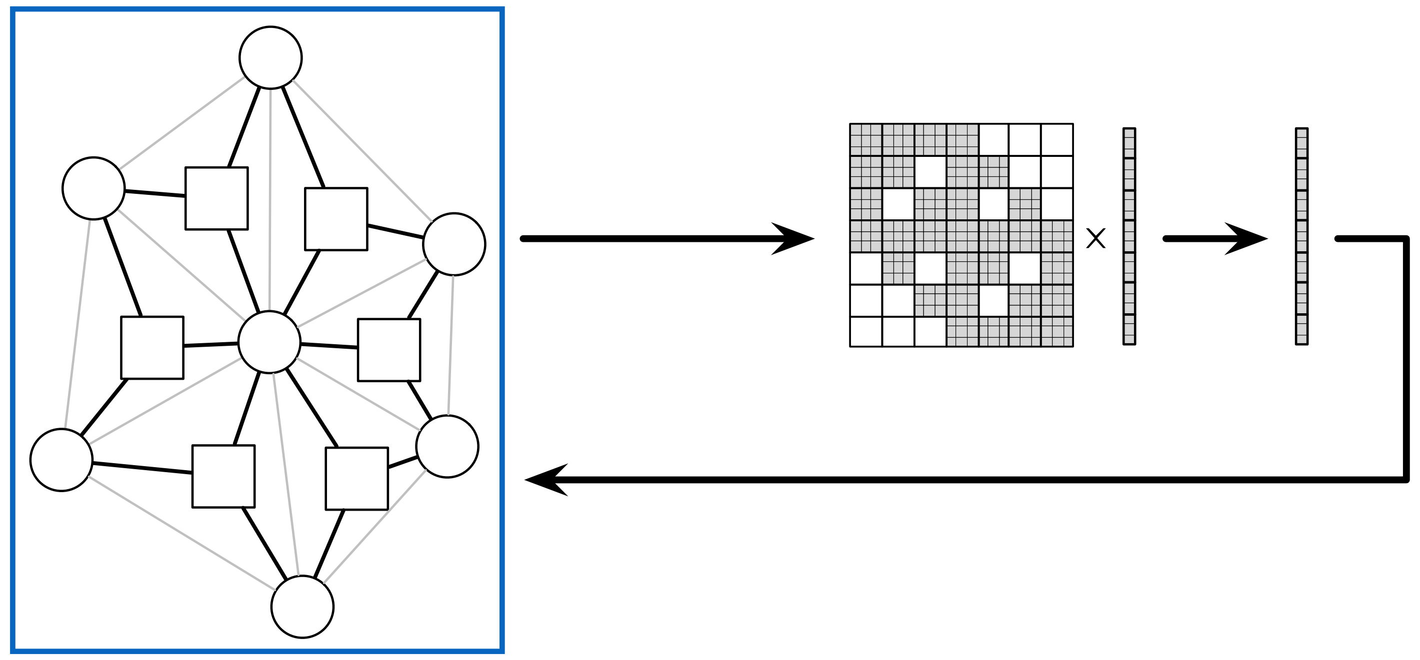


```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
extern verts      : set{Vertex};
```



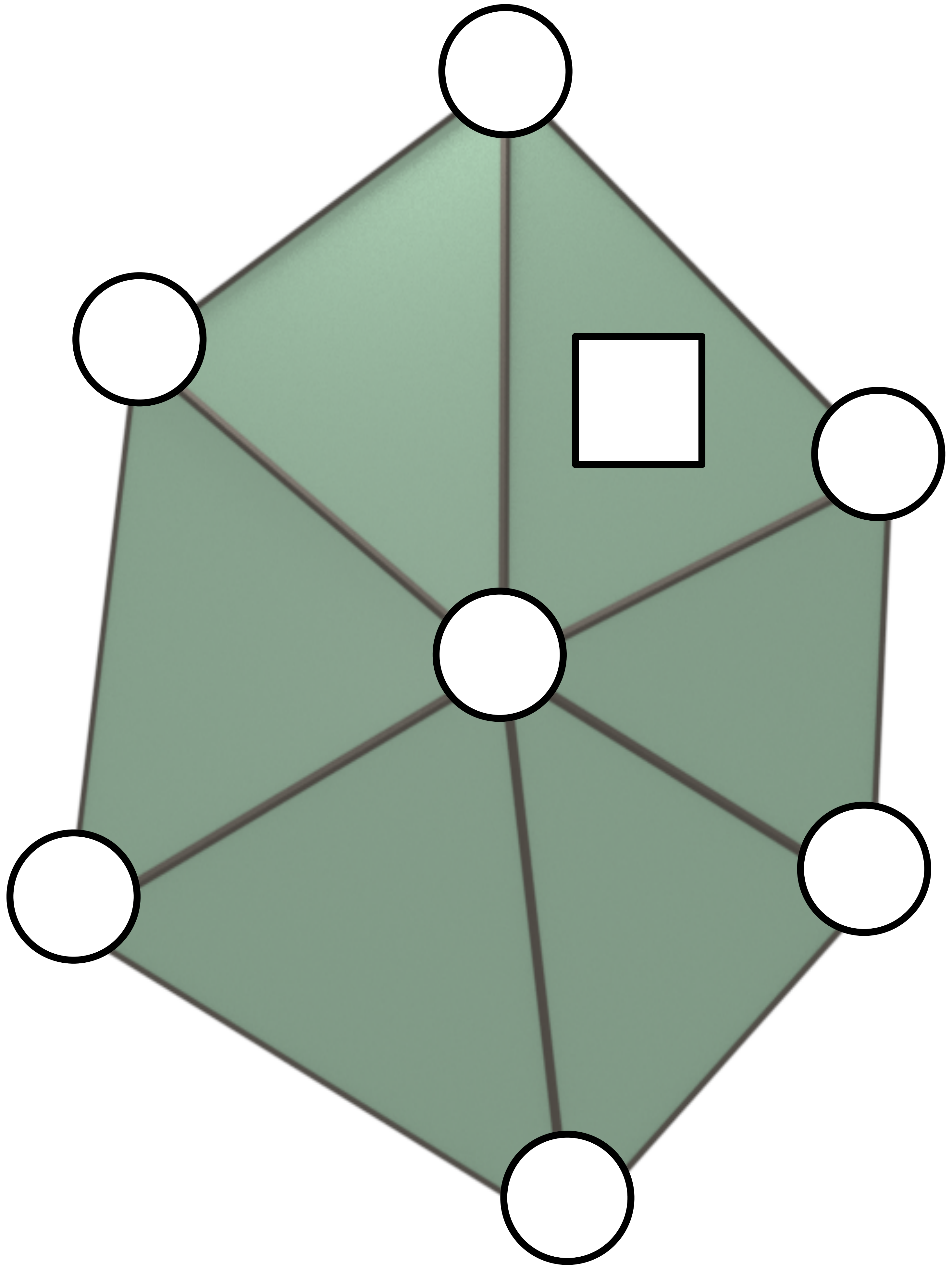
Statics Triangular Neo-Hookean FEM Simulation

Hypergraph



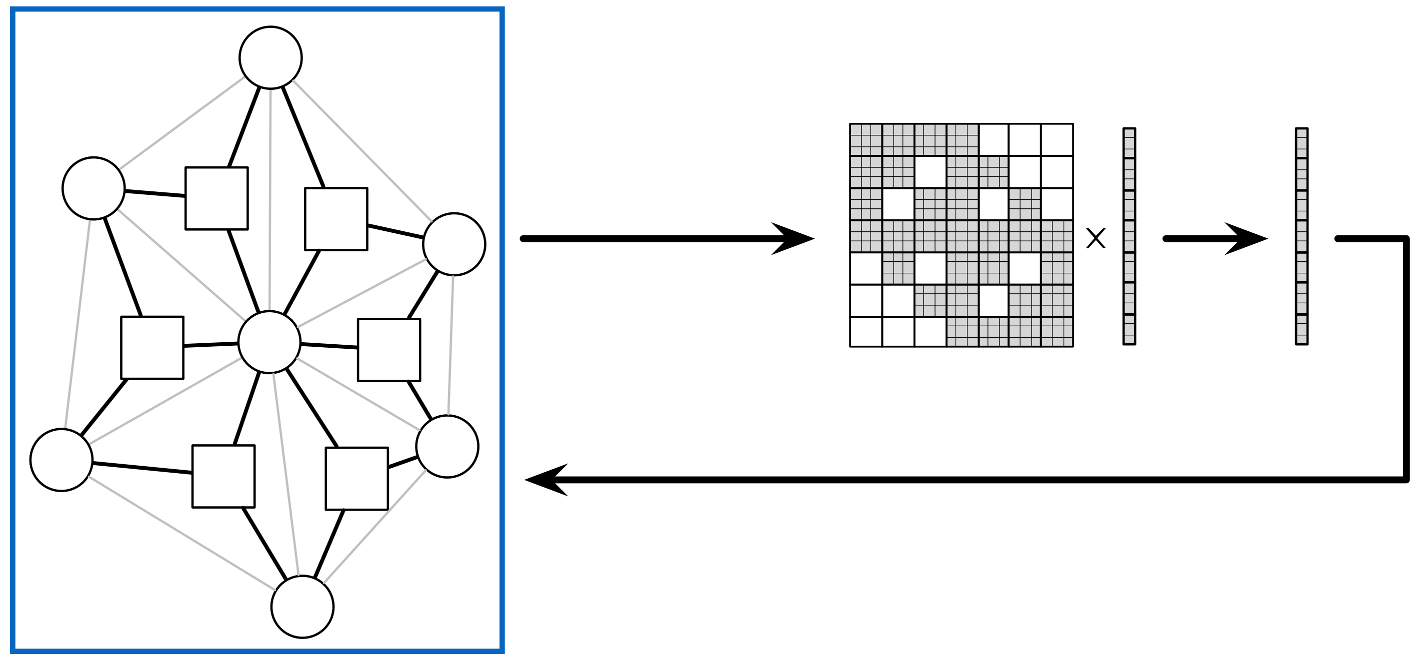
```
element Triangle
  u  : float;                  % shear modulus
  l  : float;                  % lame's first parameter
  W  : float;                  % volume
  B  : matrix[3,3](float);     % strain-displacement
end
```

```
% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
```



Statics Triangular Neo-Hookean FEM Simulation

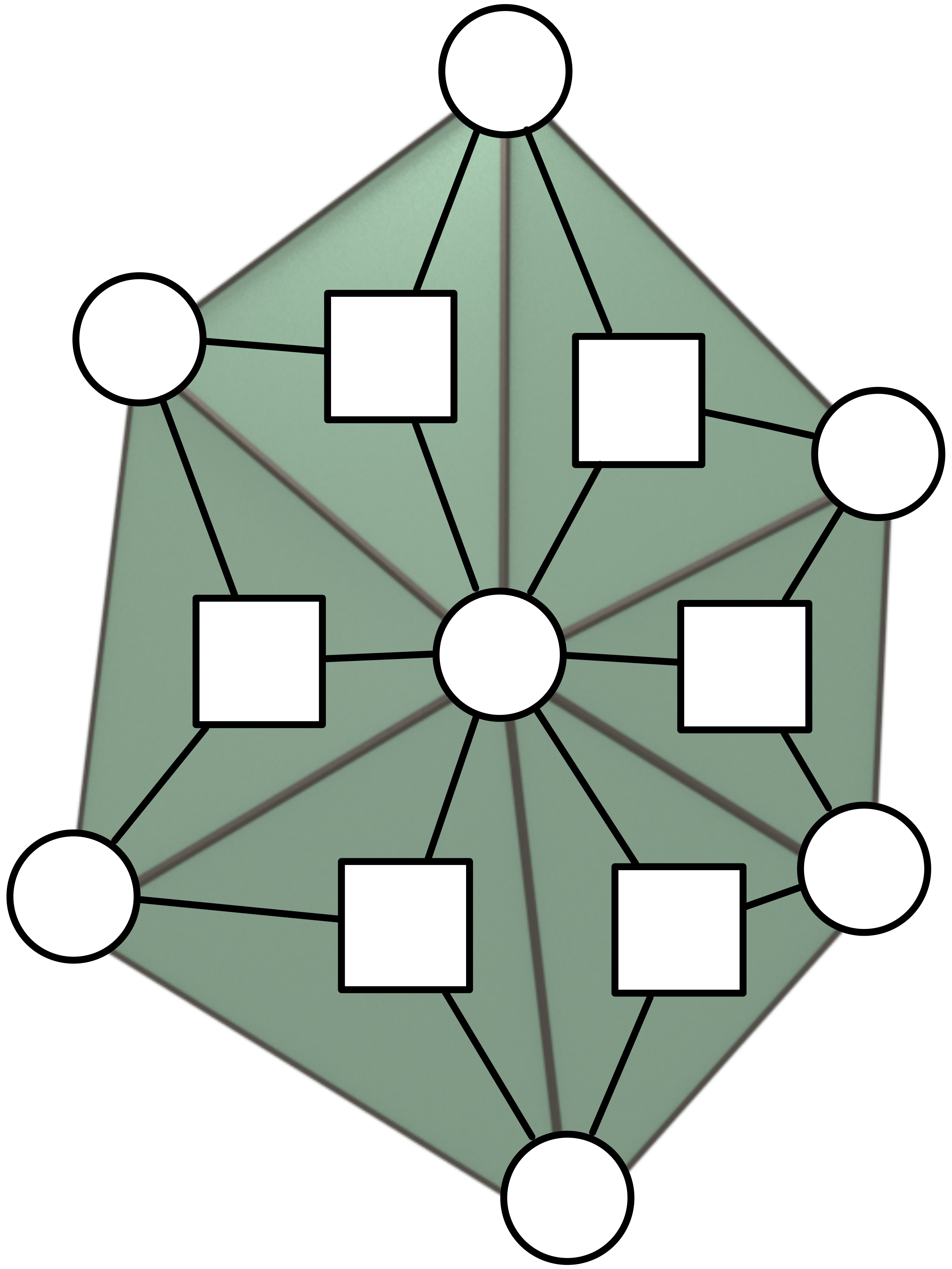
Hypergraph



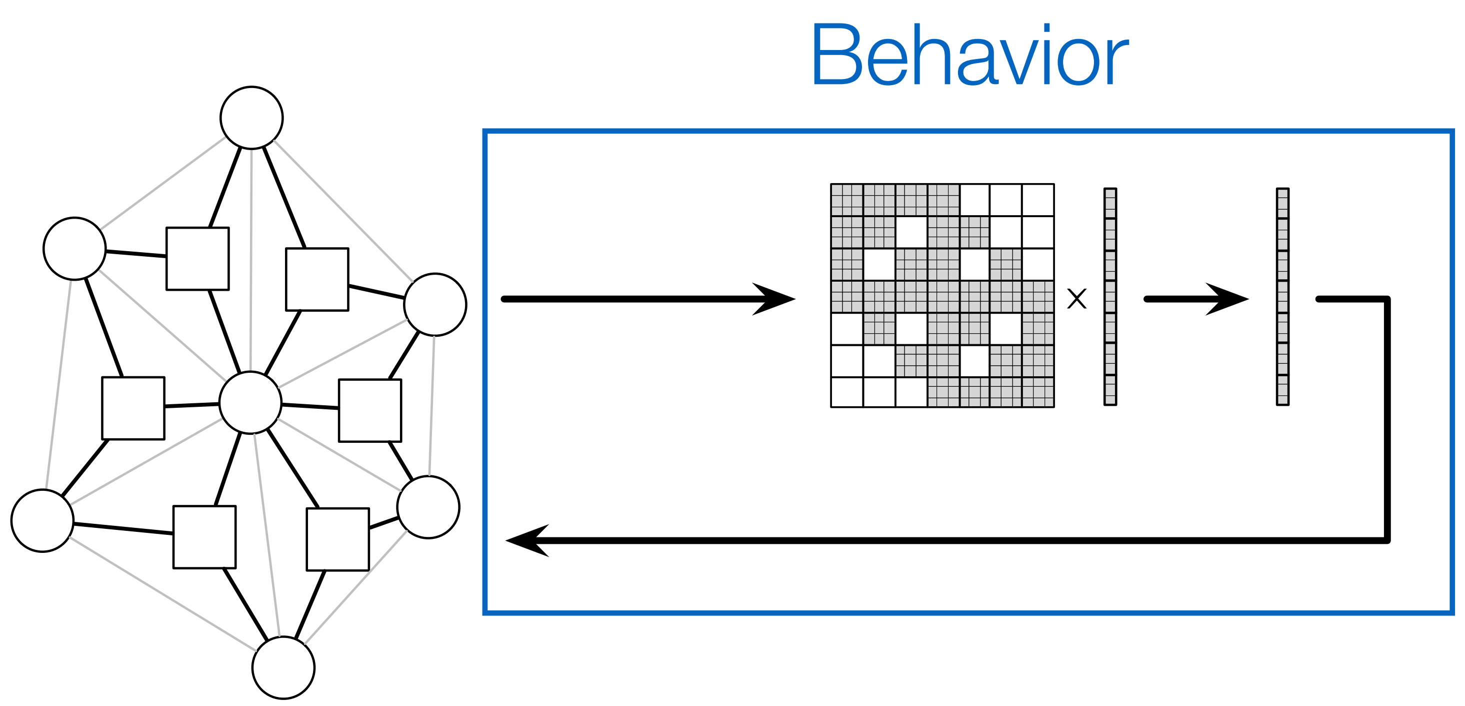
```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
```

```
extern triangles : set{Triangle}(verts, verts, verts);
```

```
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
```



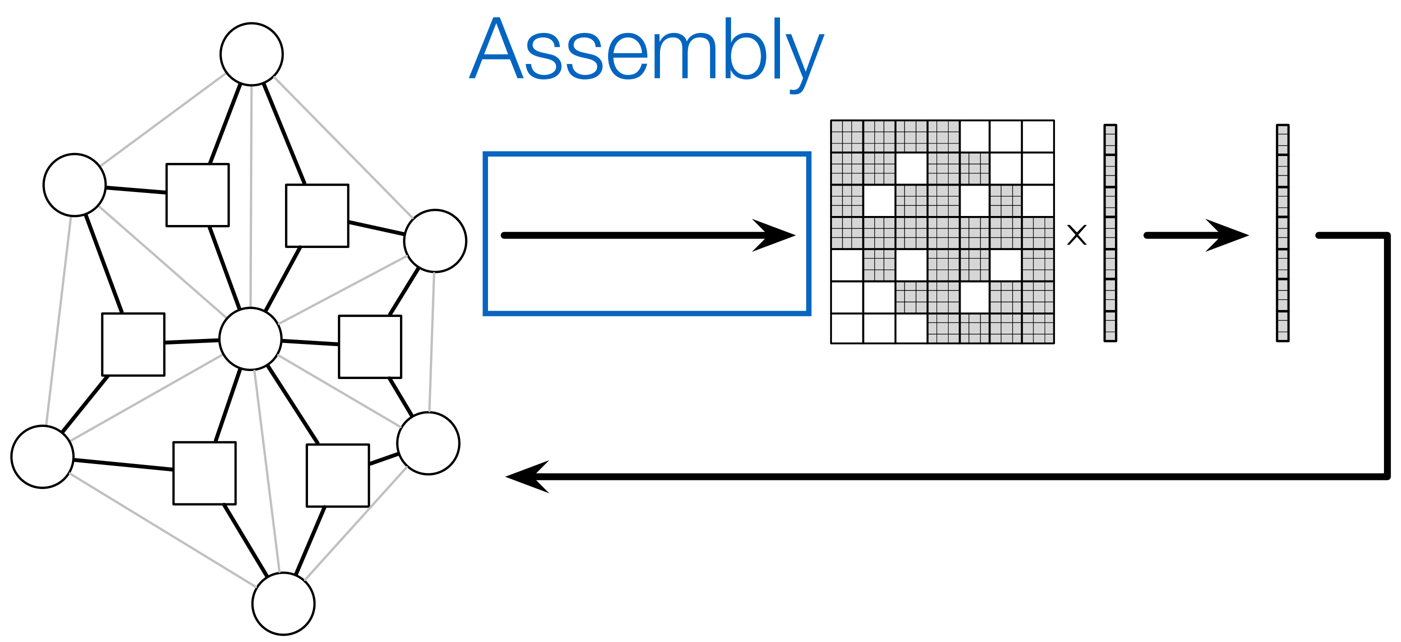
Statics Triangular Neo-Hookean FEM Simulation

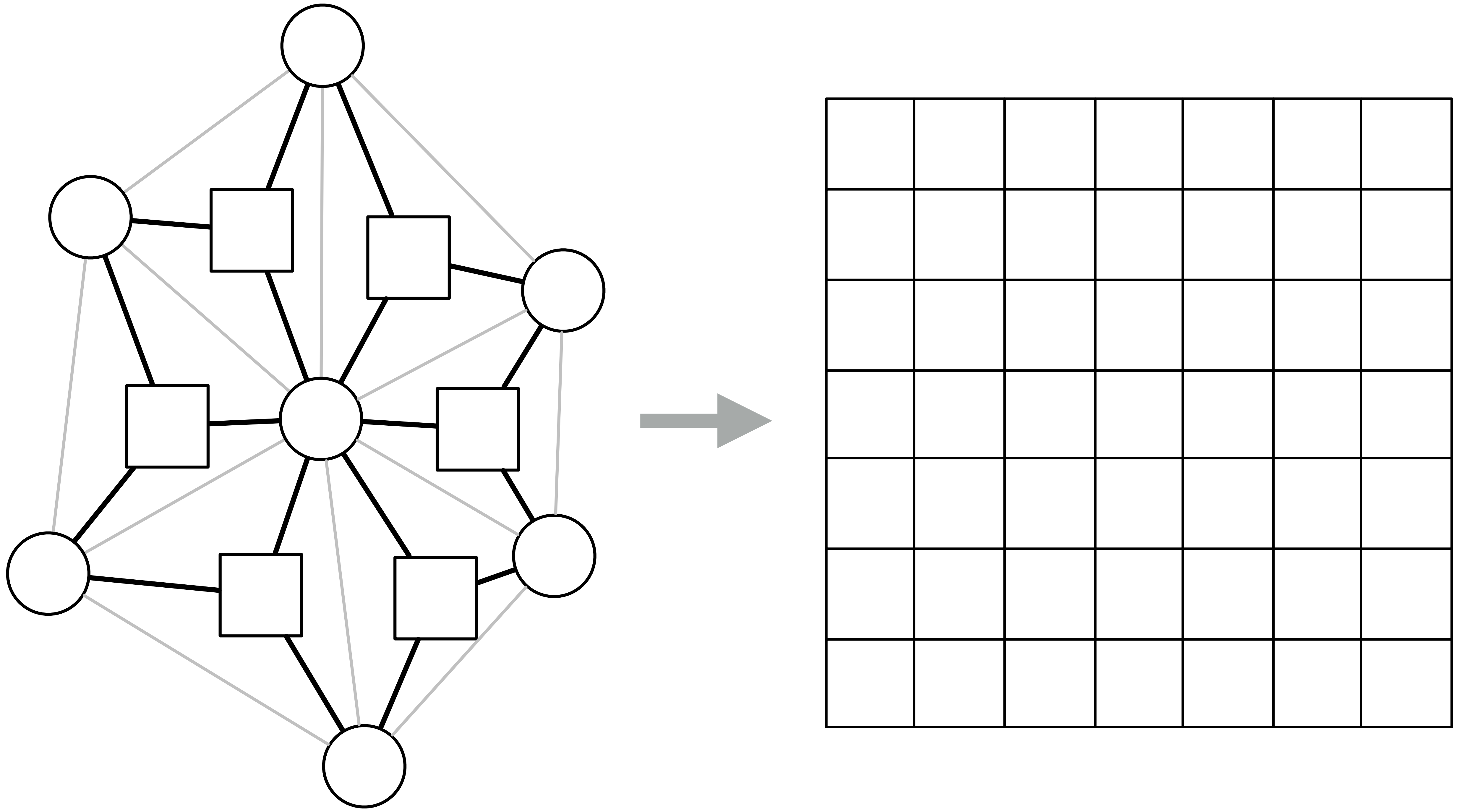



```
element Vertex
  x  : vector[3](float
  v  : vector[3](float
  fe : vector[3](float
end

element Triangle
  u  : float;
  l  : float;
  W  : float;
  B  : matrix[3,3](flo
end

% graph vertices and t
extern verts     : set
extern triangles : set

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    // assemble stiffness matrix
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



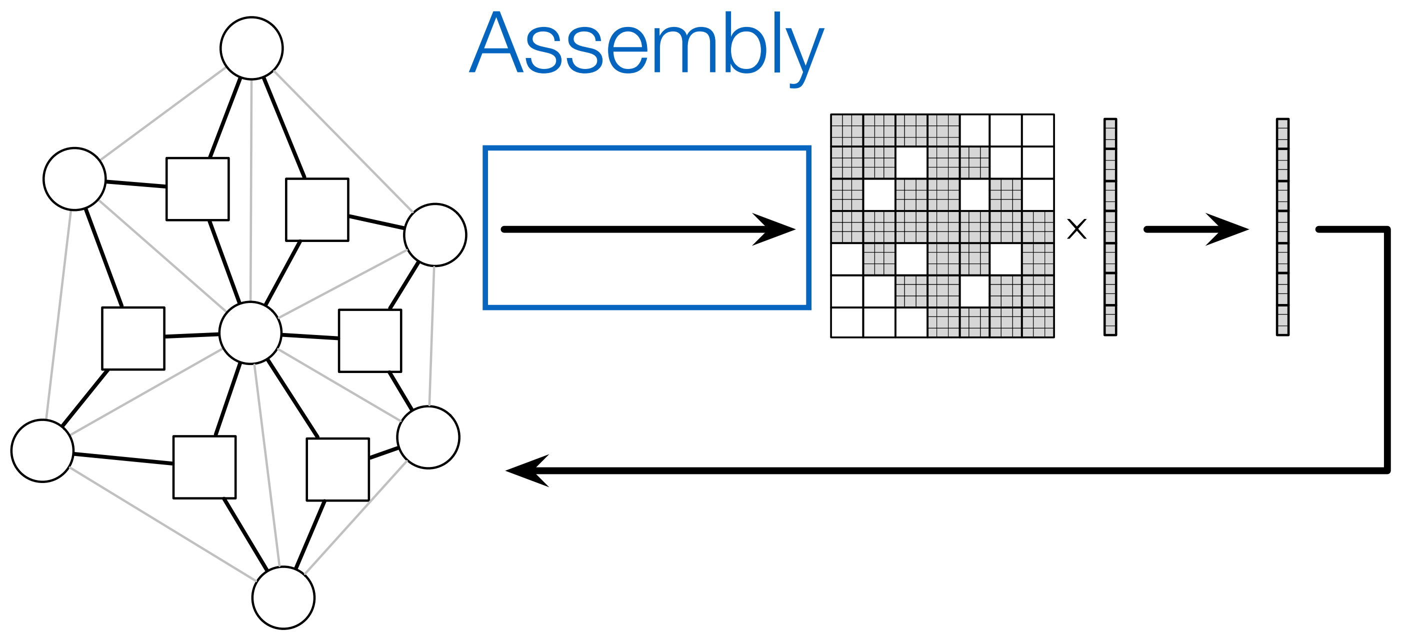


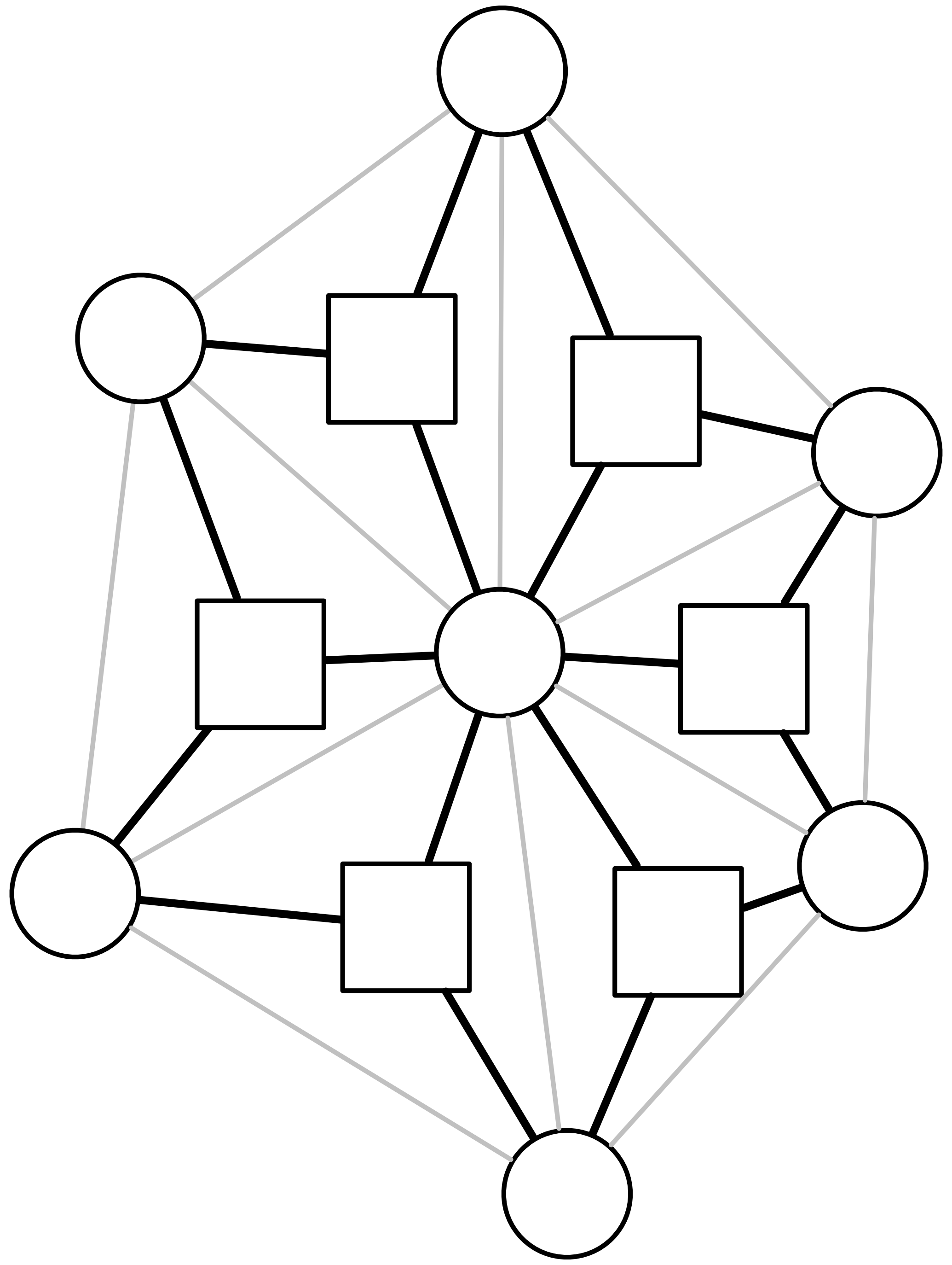
```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l
  W
  B
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

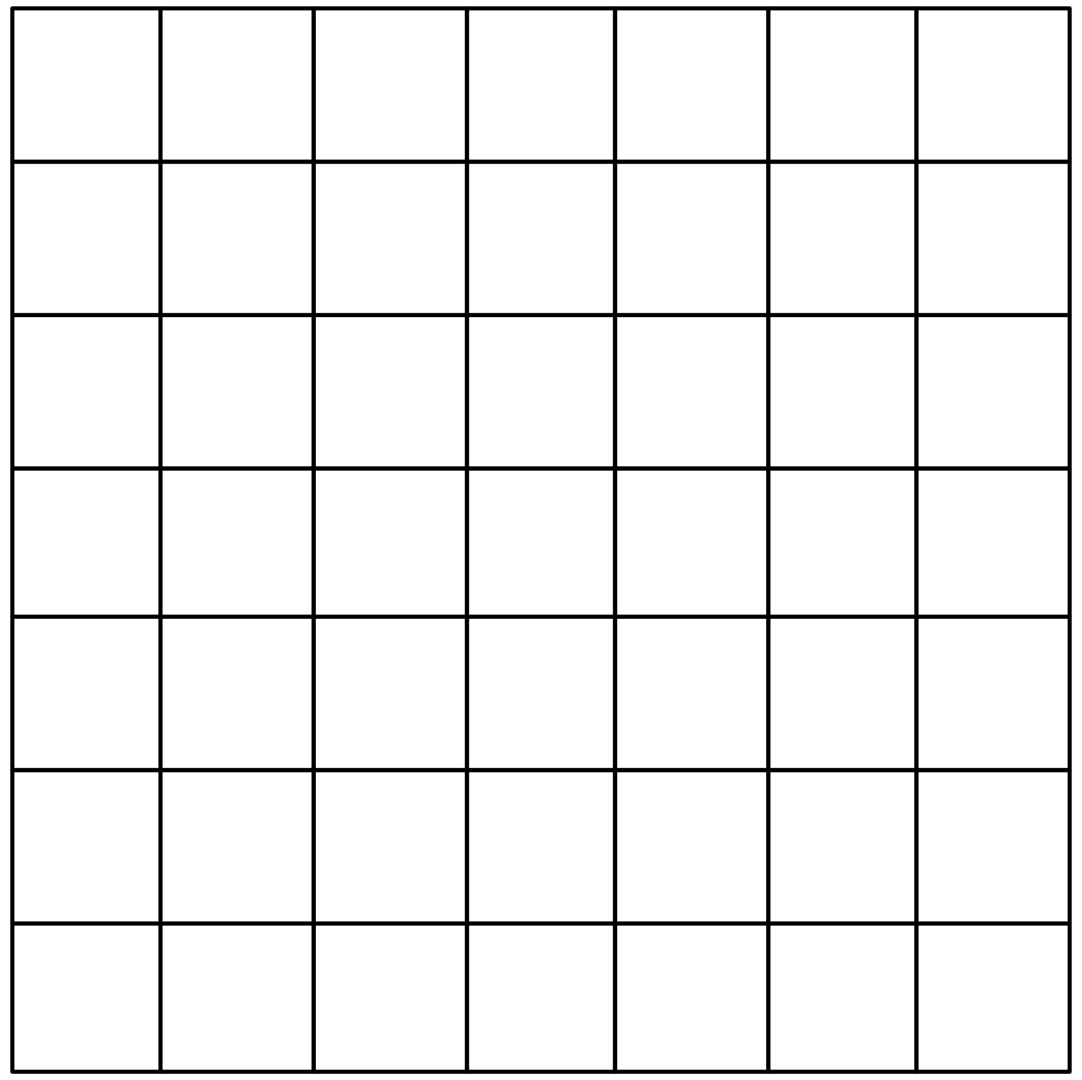
```
K = map triangle_stiffness to triangles reduce +;
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    // assemble stiffness matrix
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



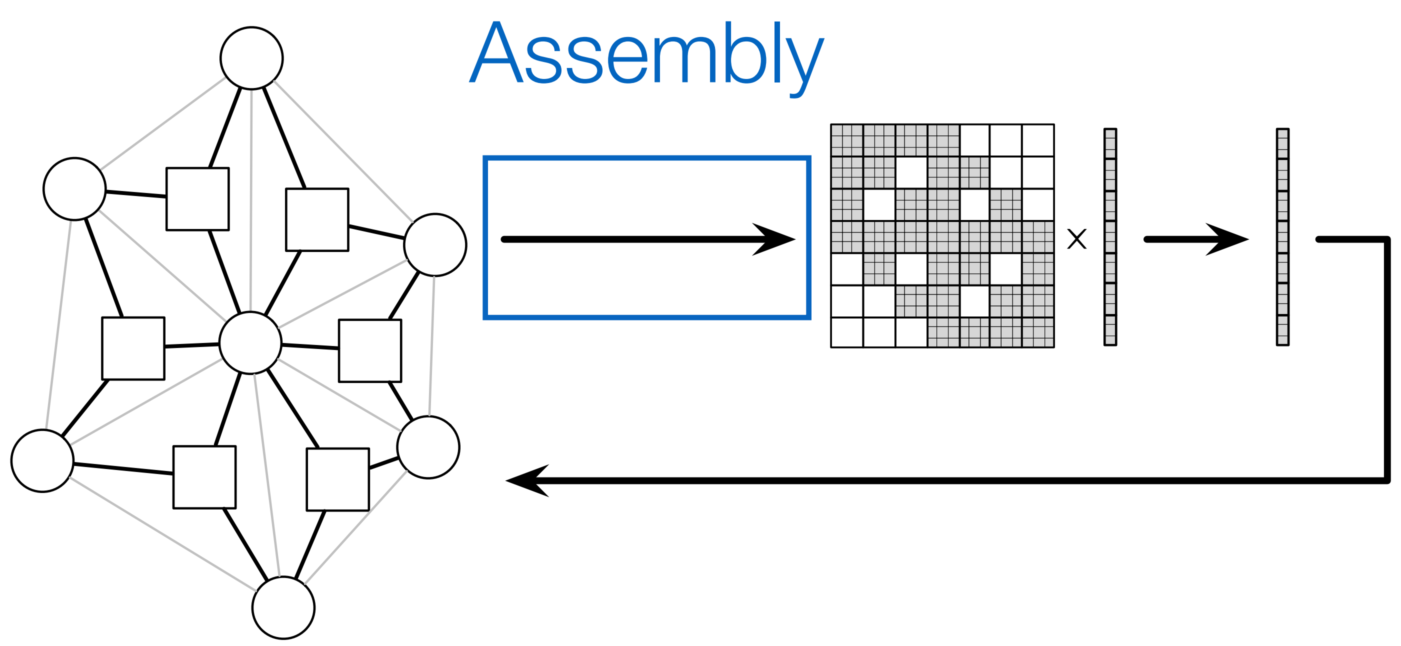


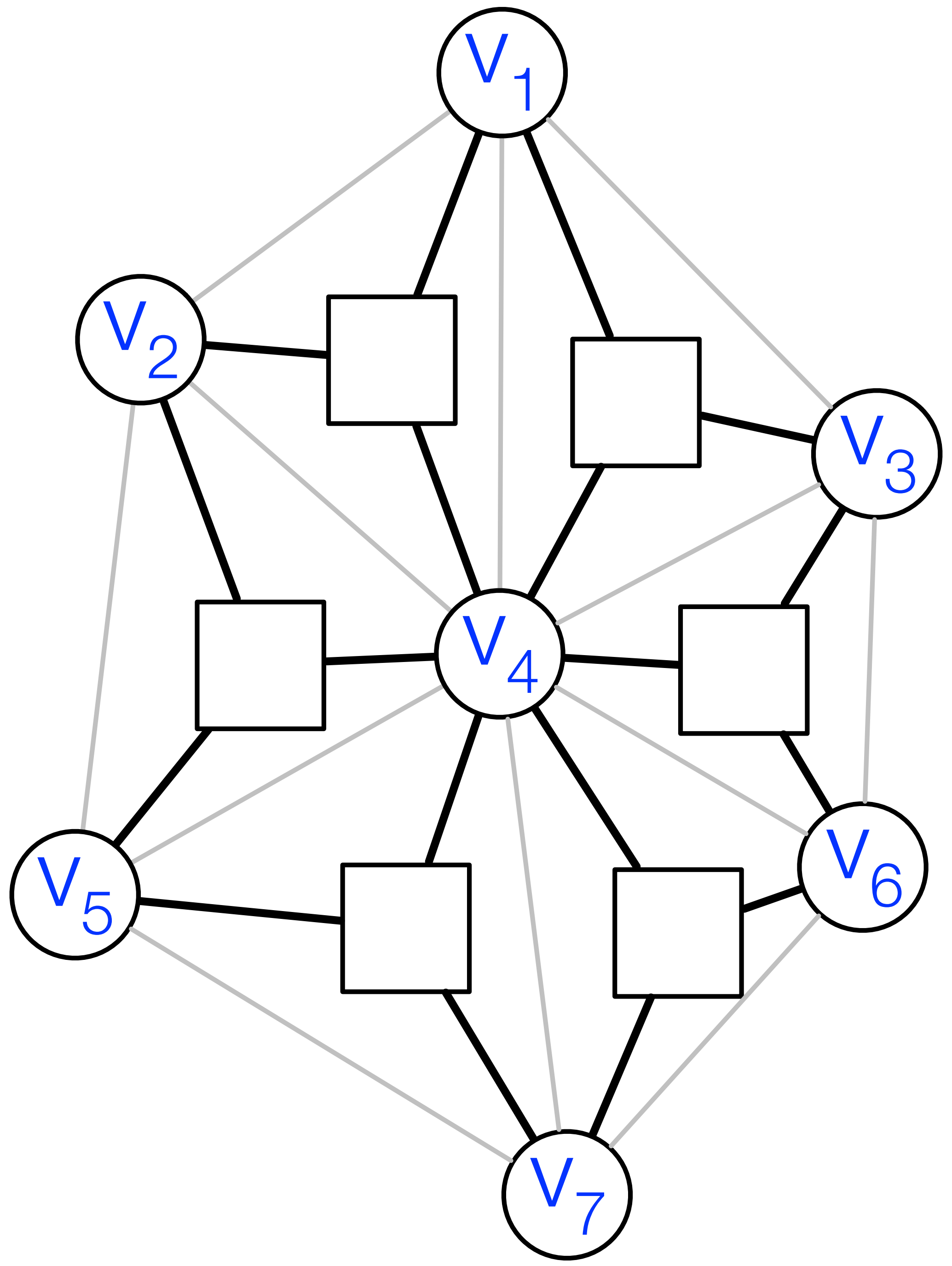


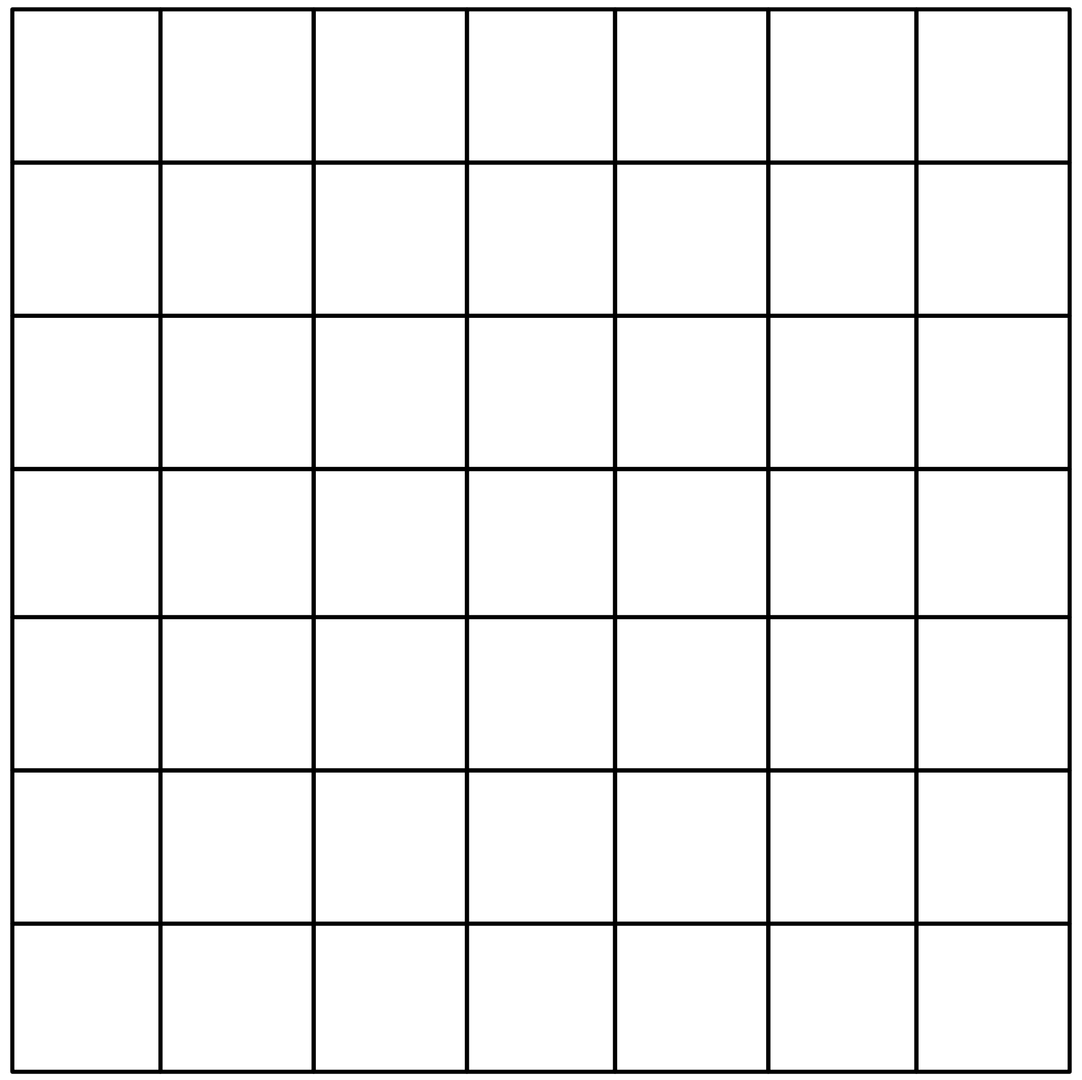
```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



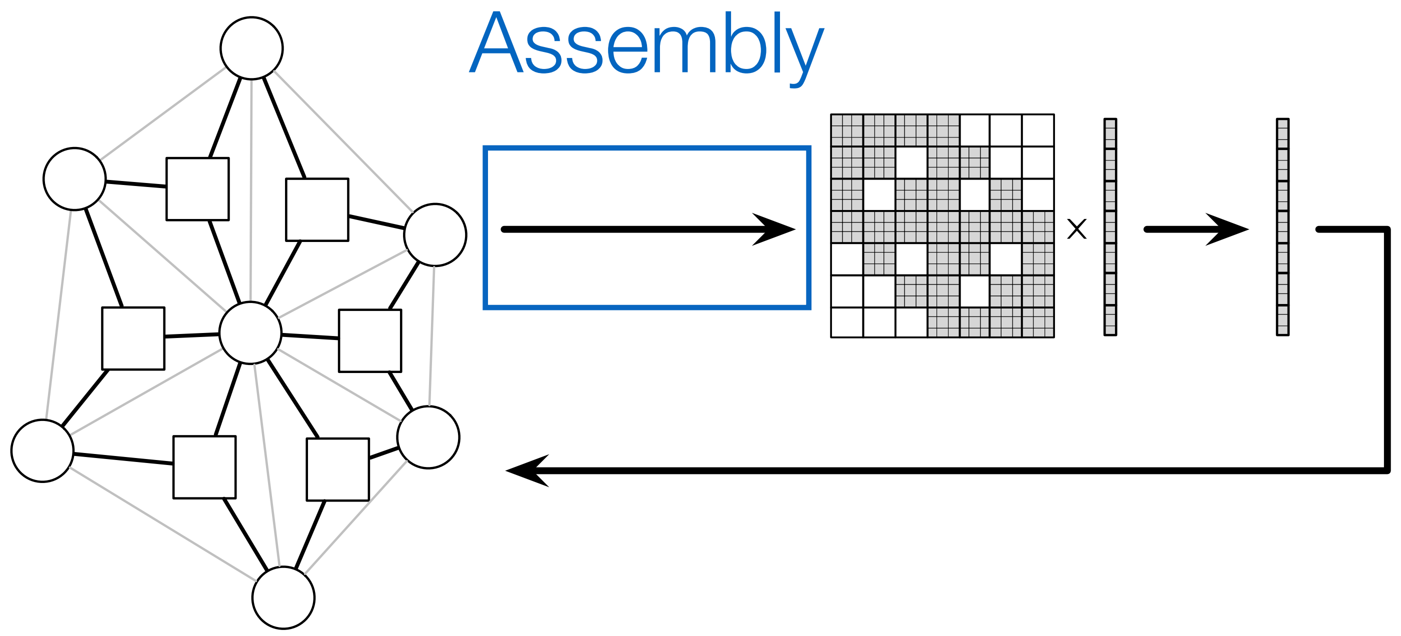


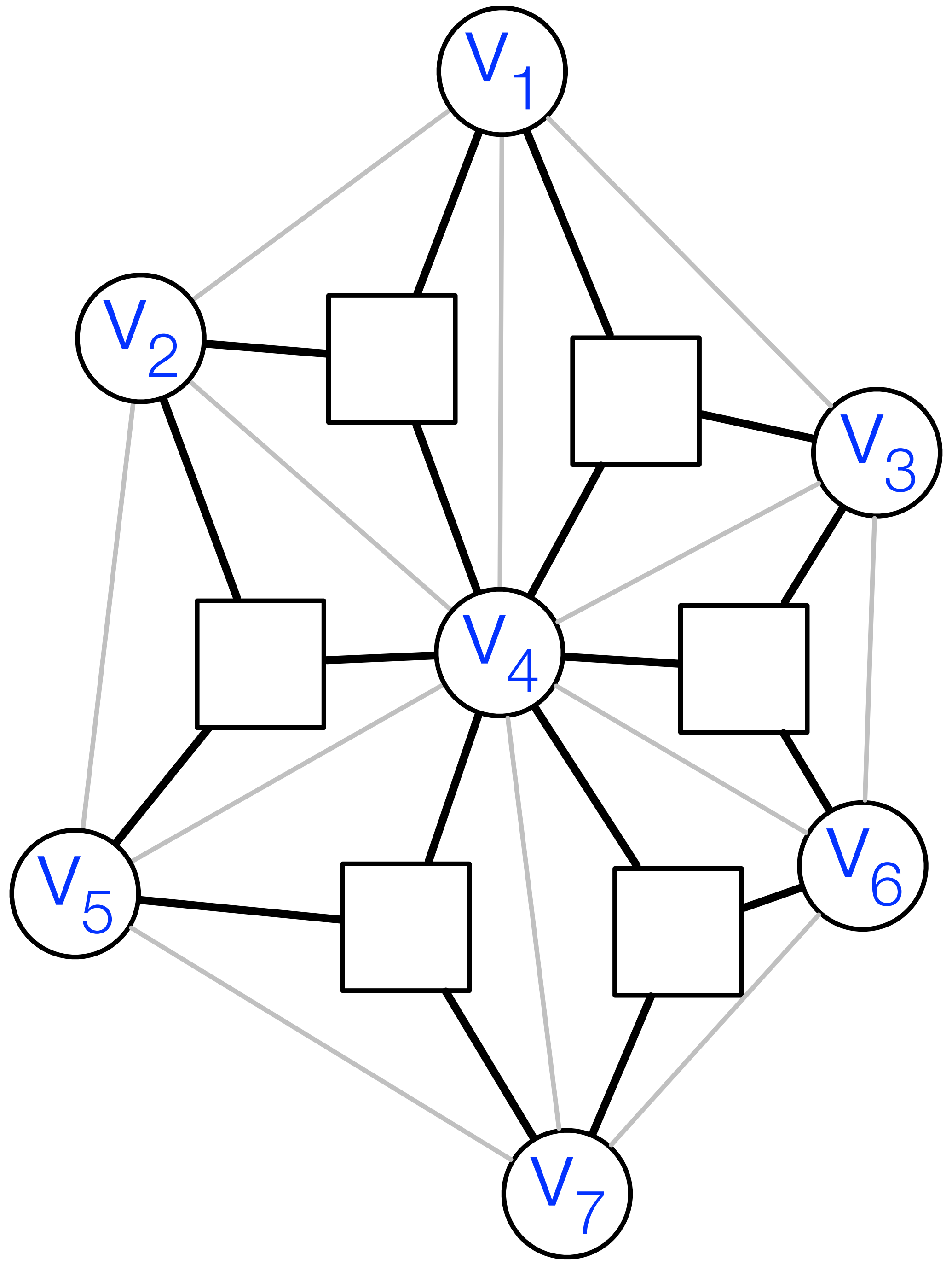


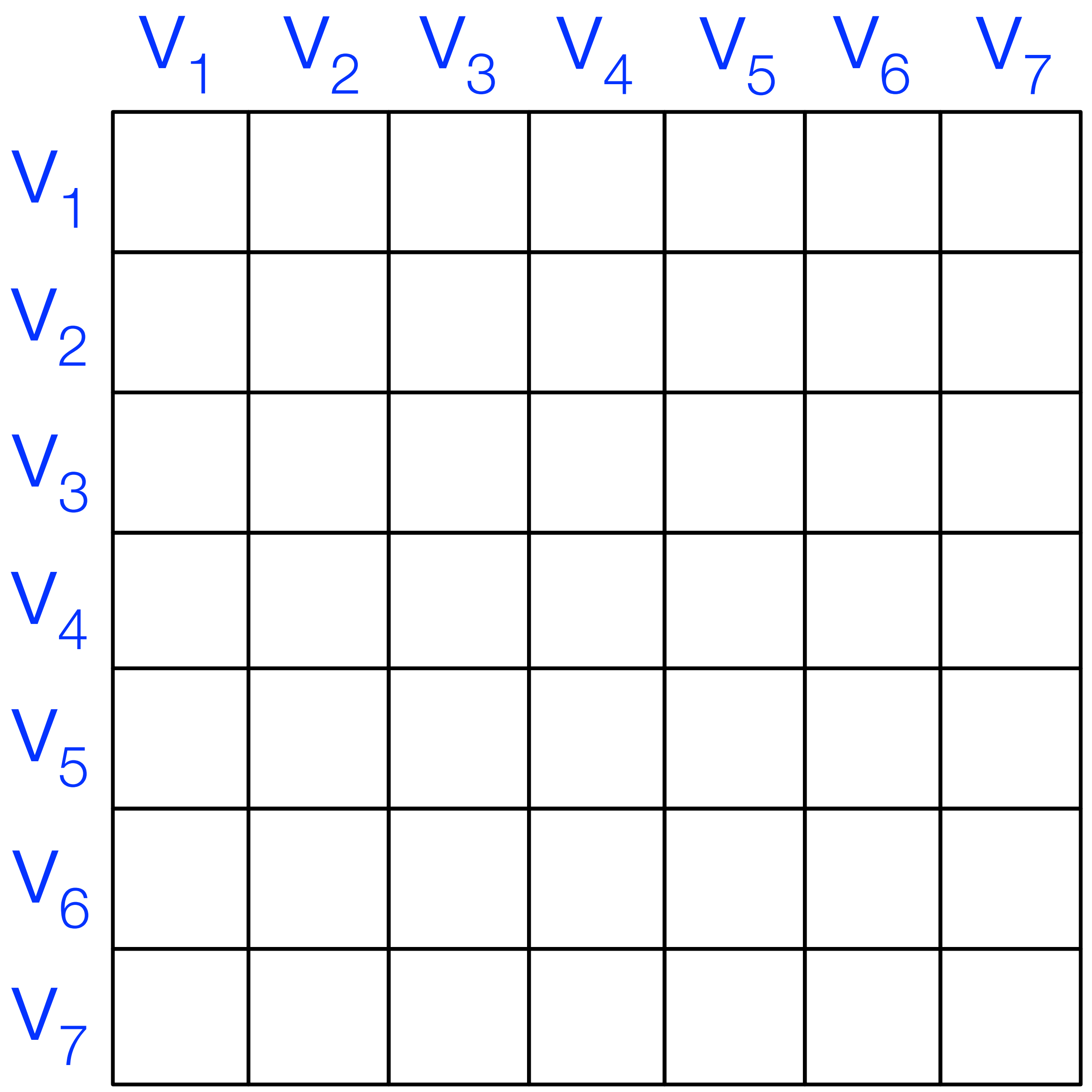
```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



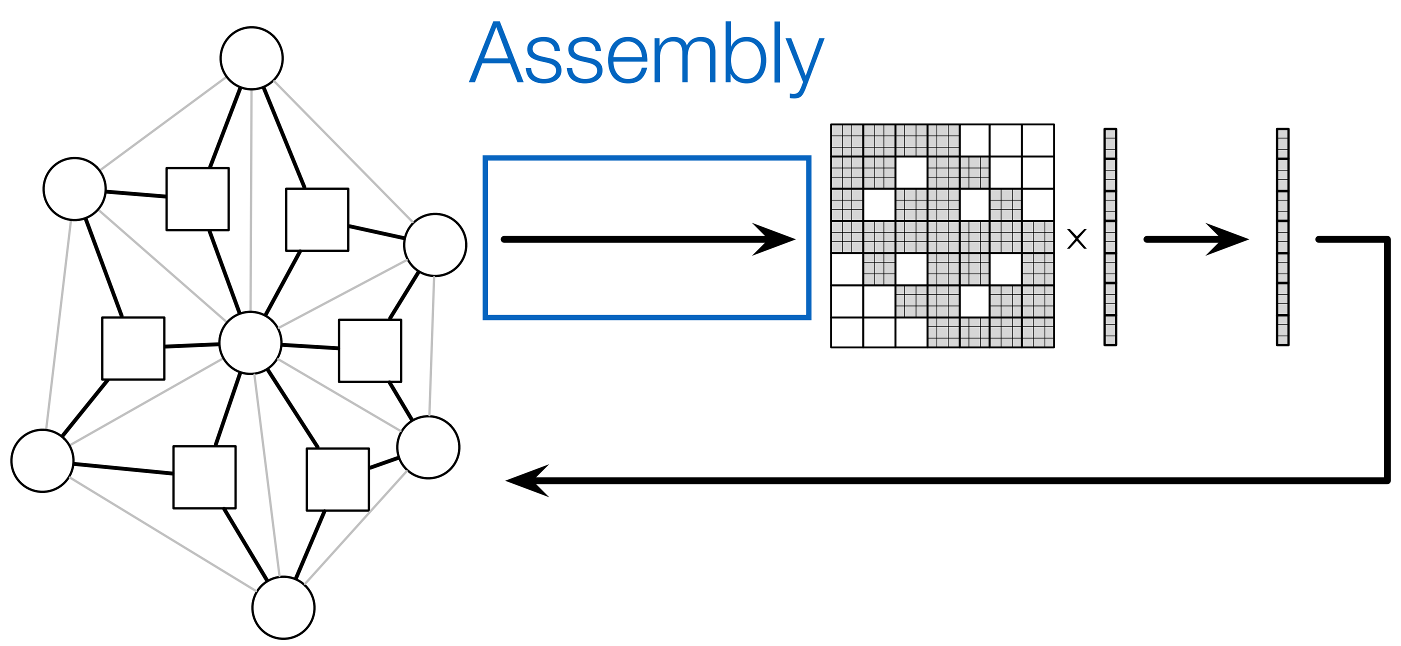


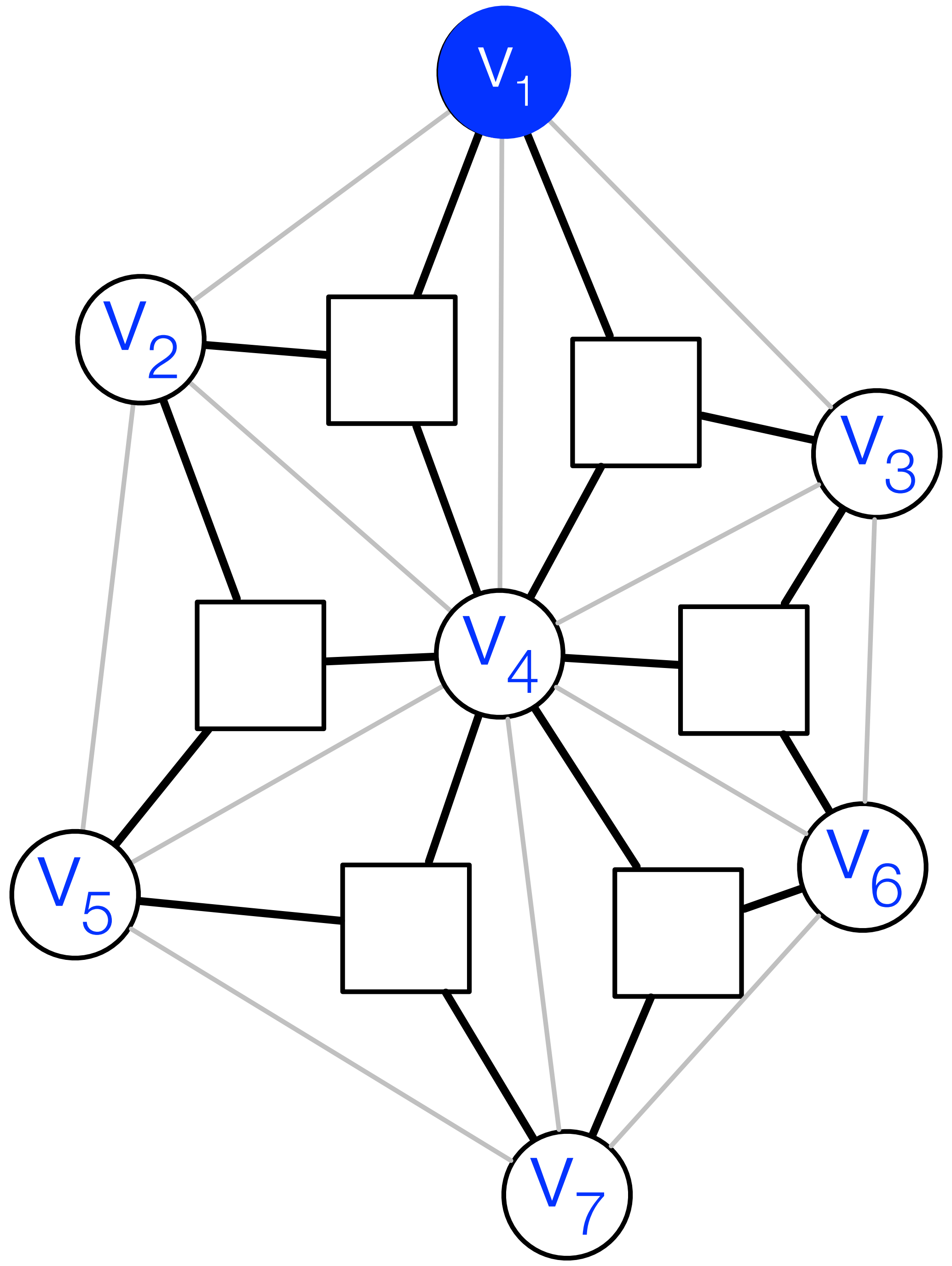


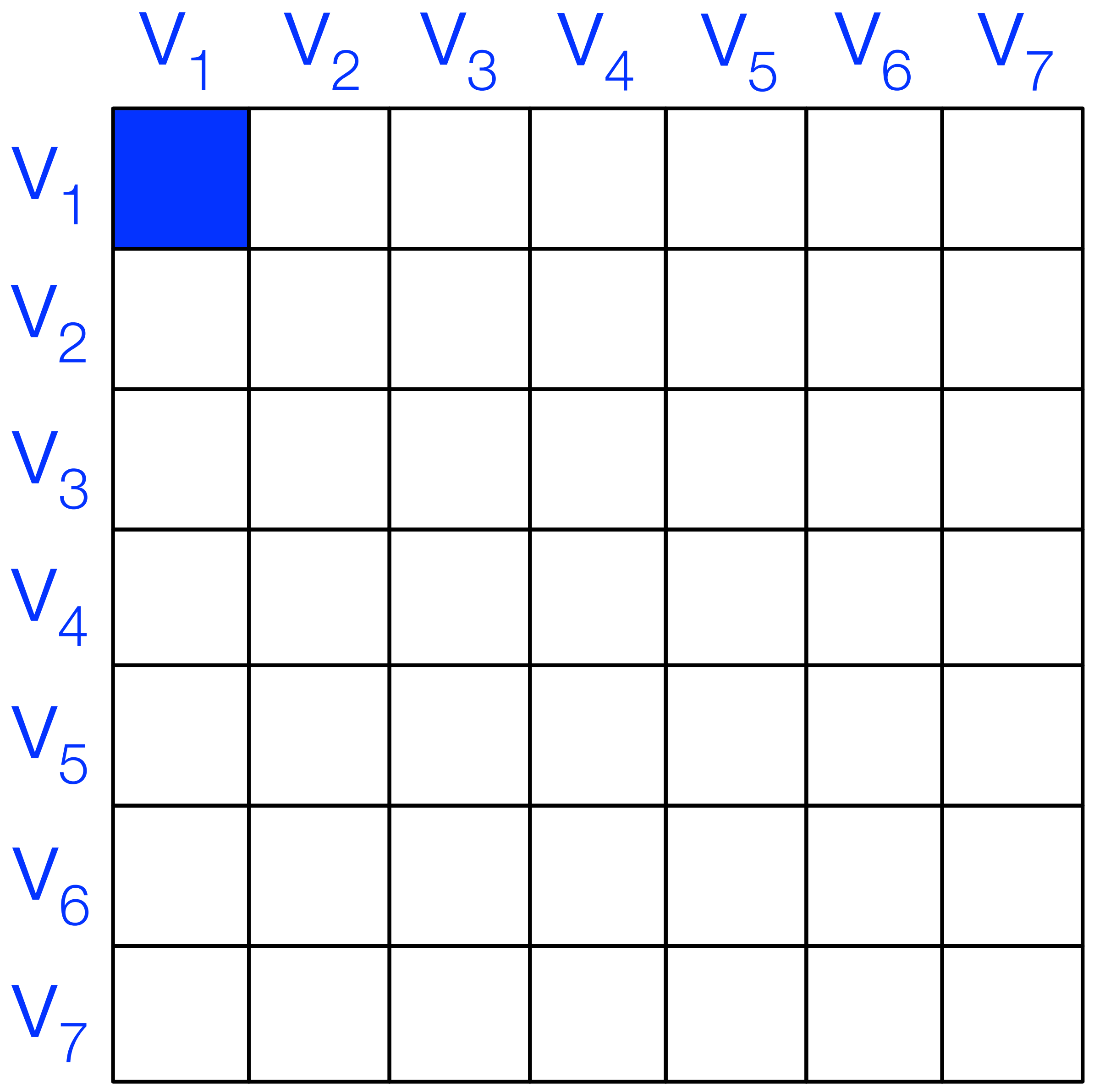

```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



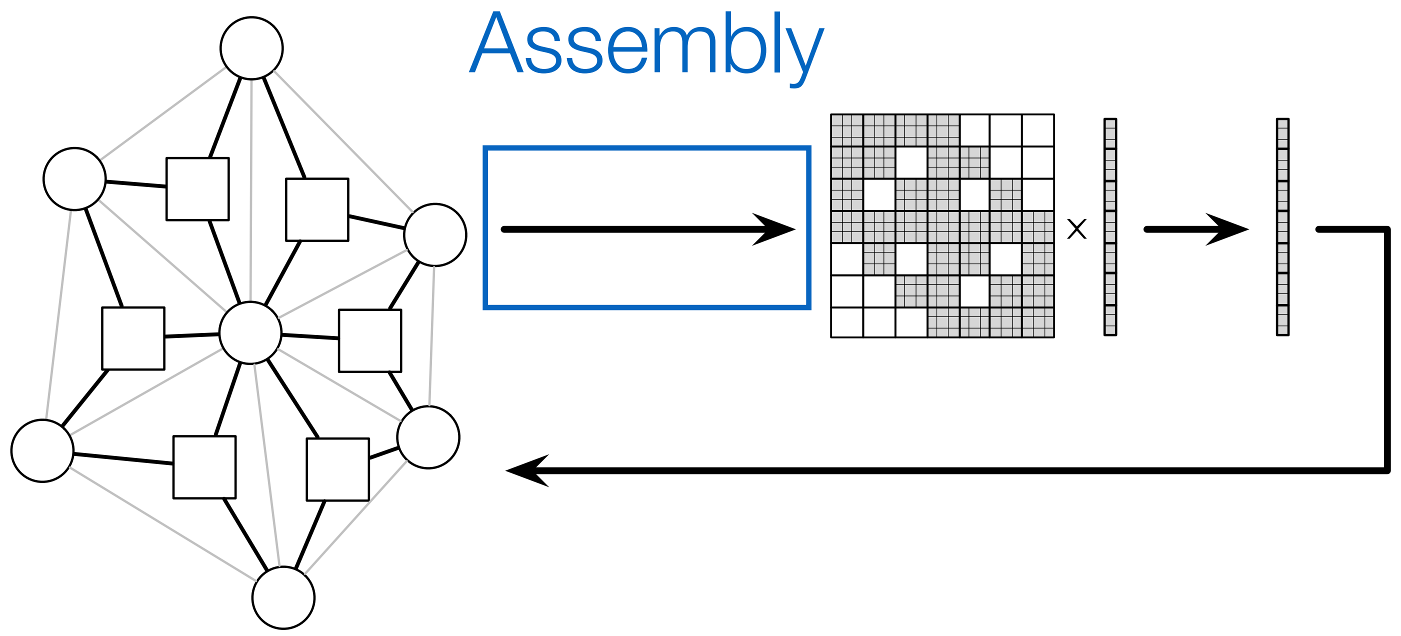


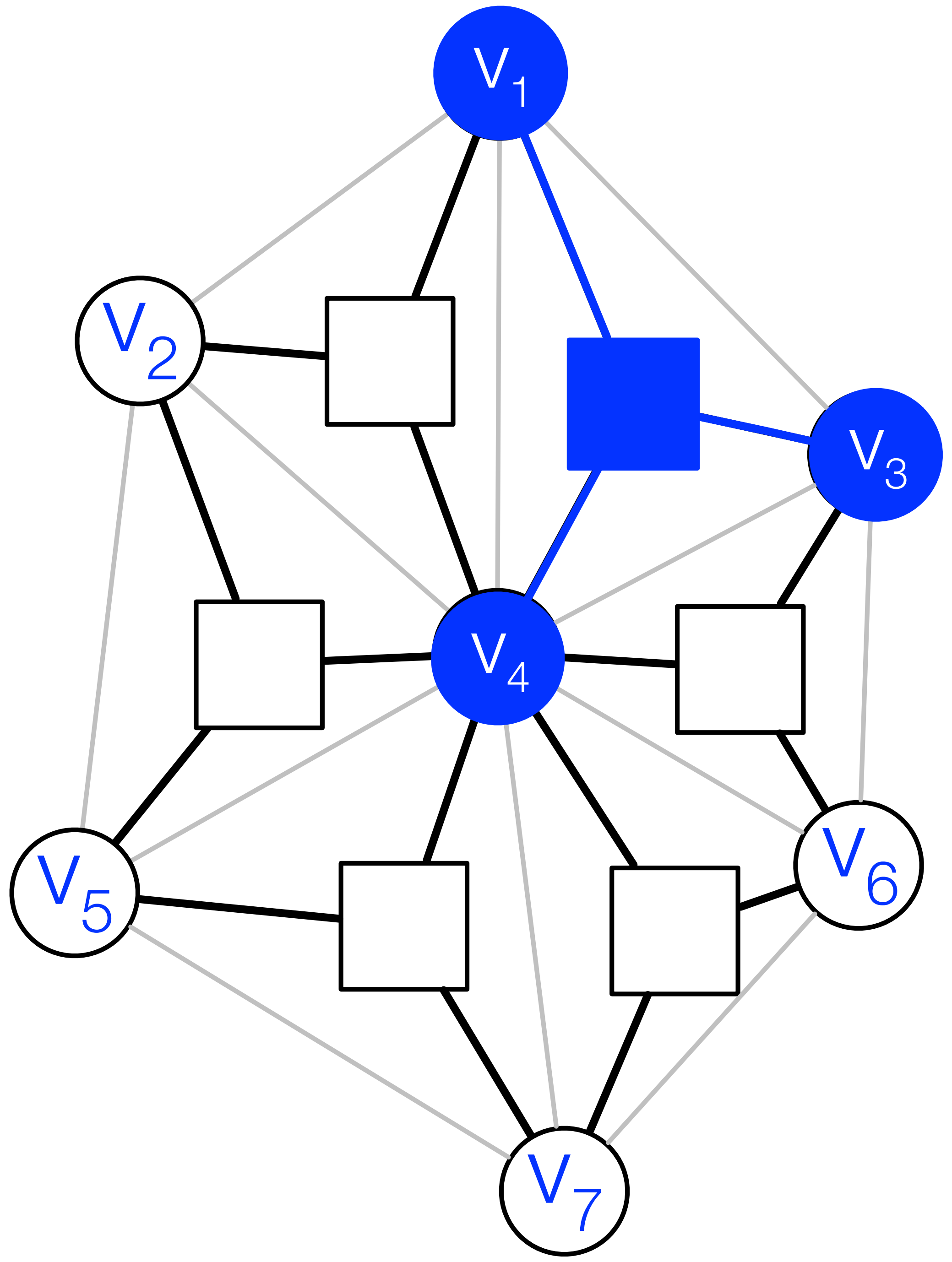



```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;               % shear modulus
  l  : float;               % lame's first parameter
  W  : float;               % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



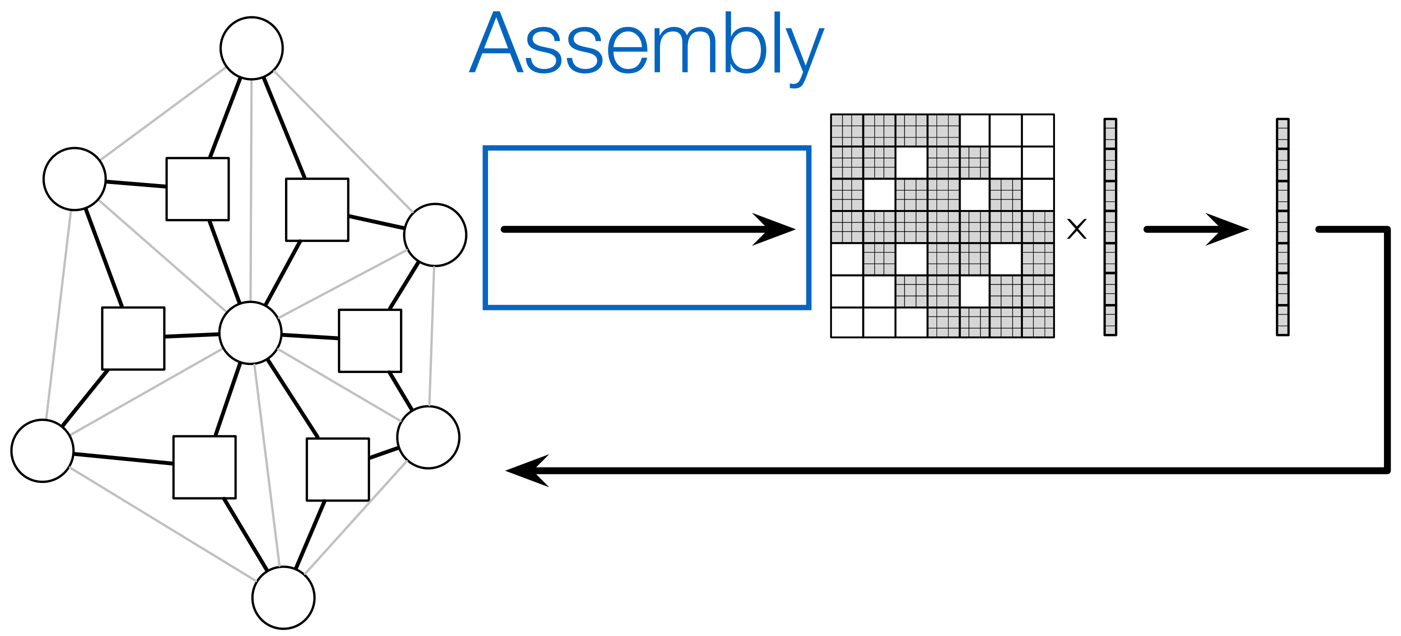


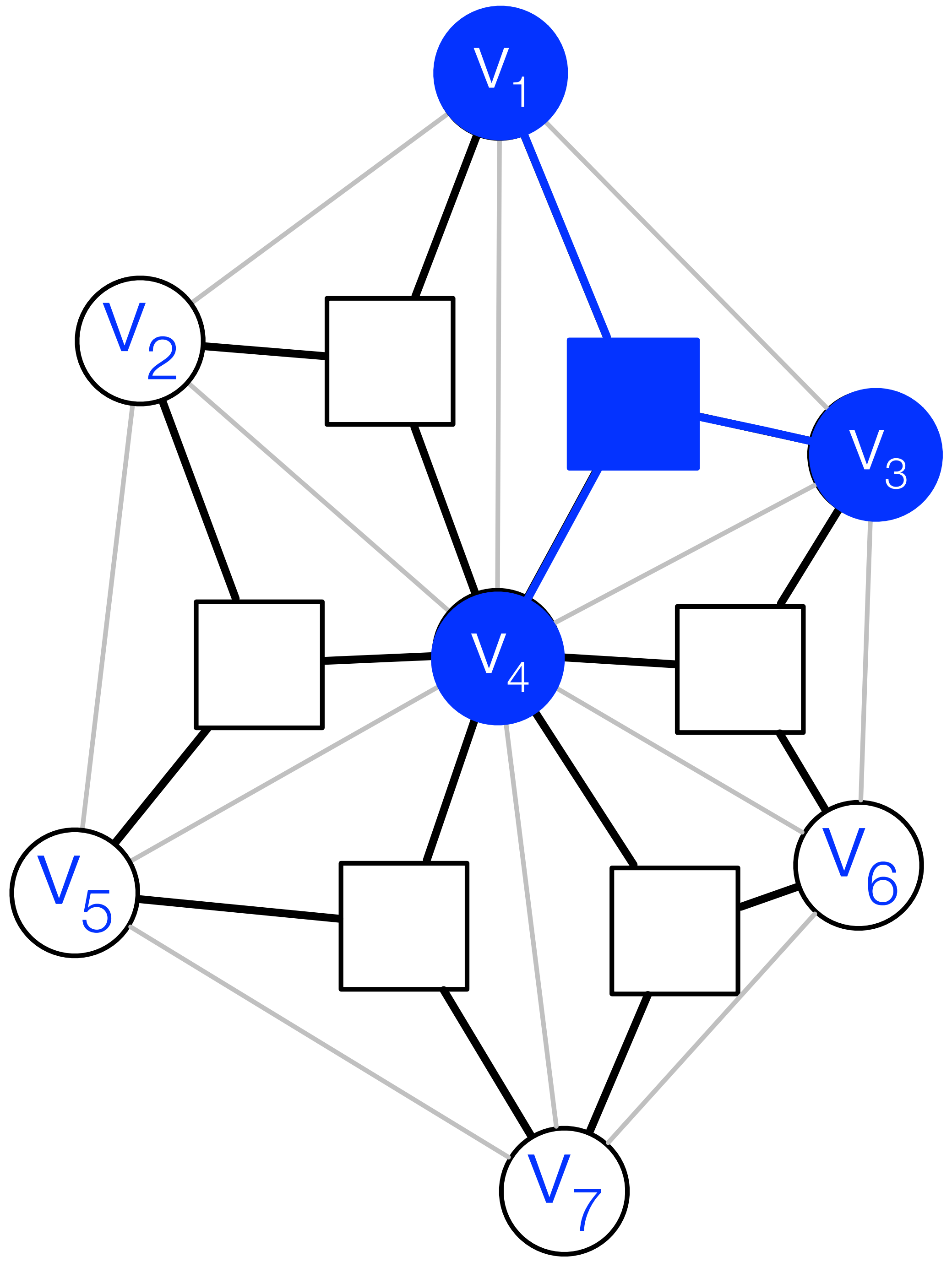
	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1							
V_2							
V_3							
V_4							
V_5							
V_6							
V_7							

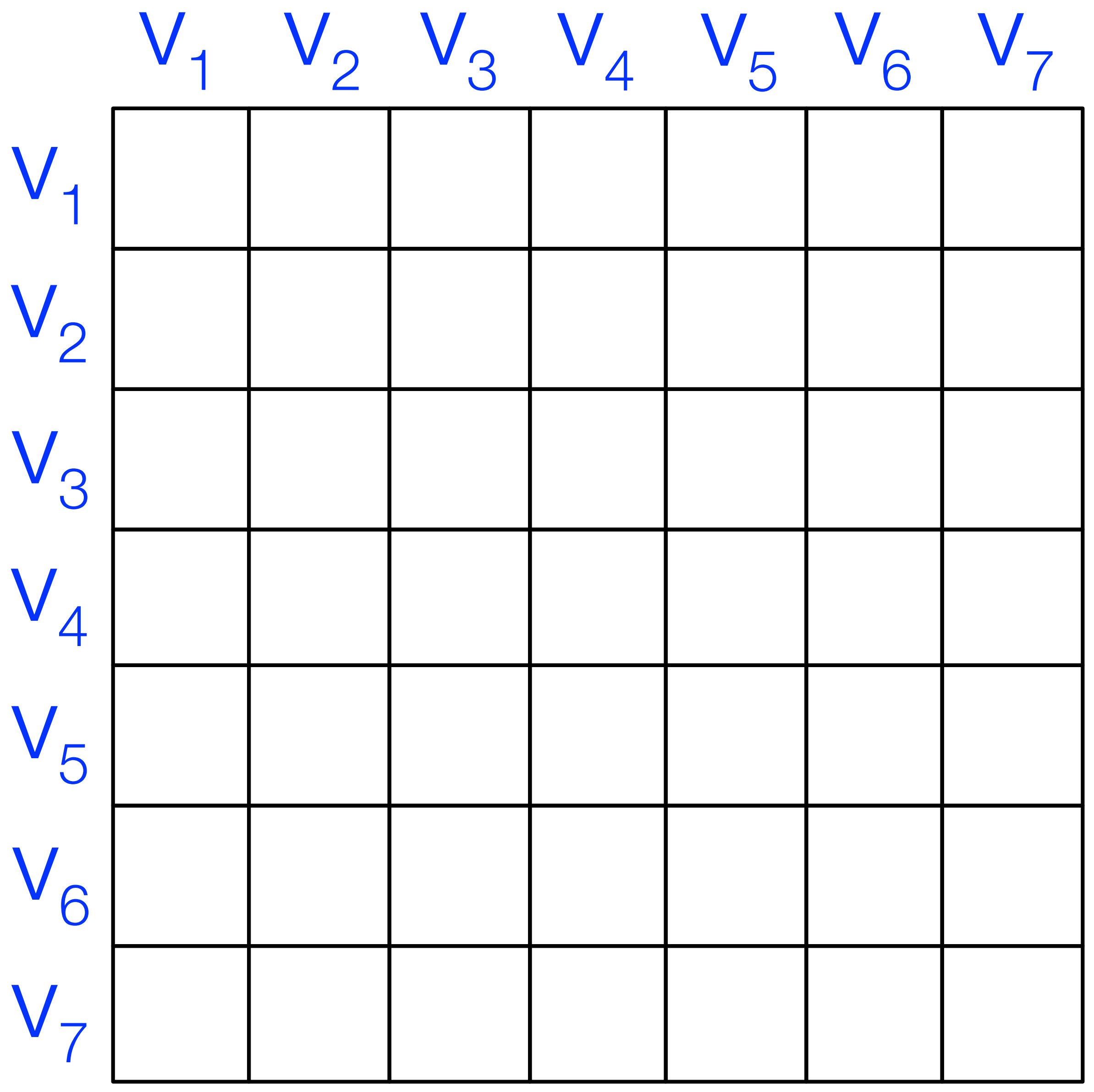
```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;               % shear modulus
  l  : float;               % lame's first parameter
  W  : float;               % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% newton's method
export func newton_method()
  while abs(f − verts.fe) > 1e−6
    K = map triangle_stiffness to triangles reduce +;
    // assemble force vector
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation





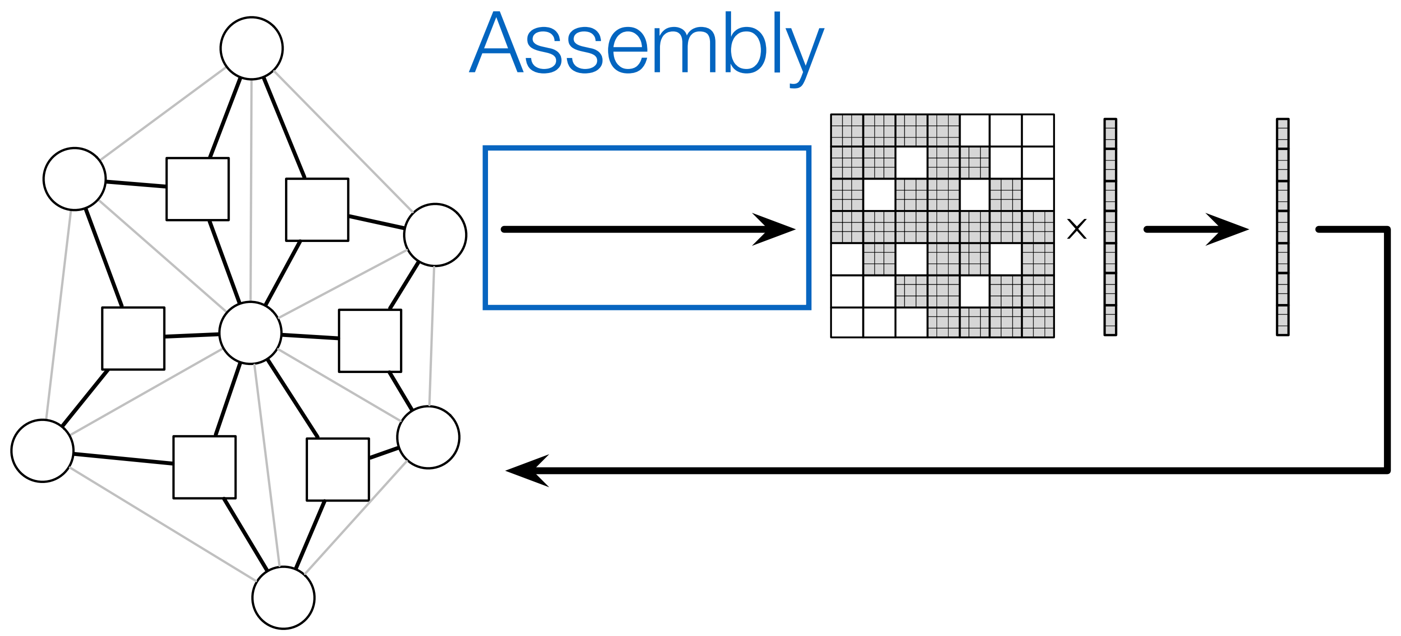


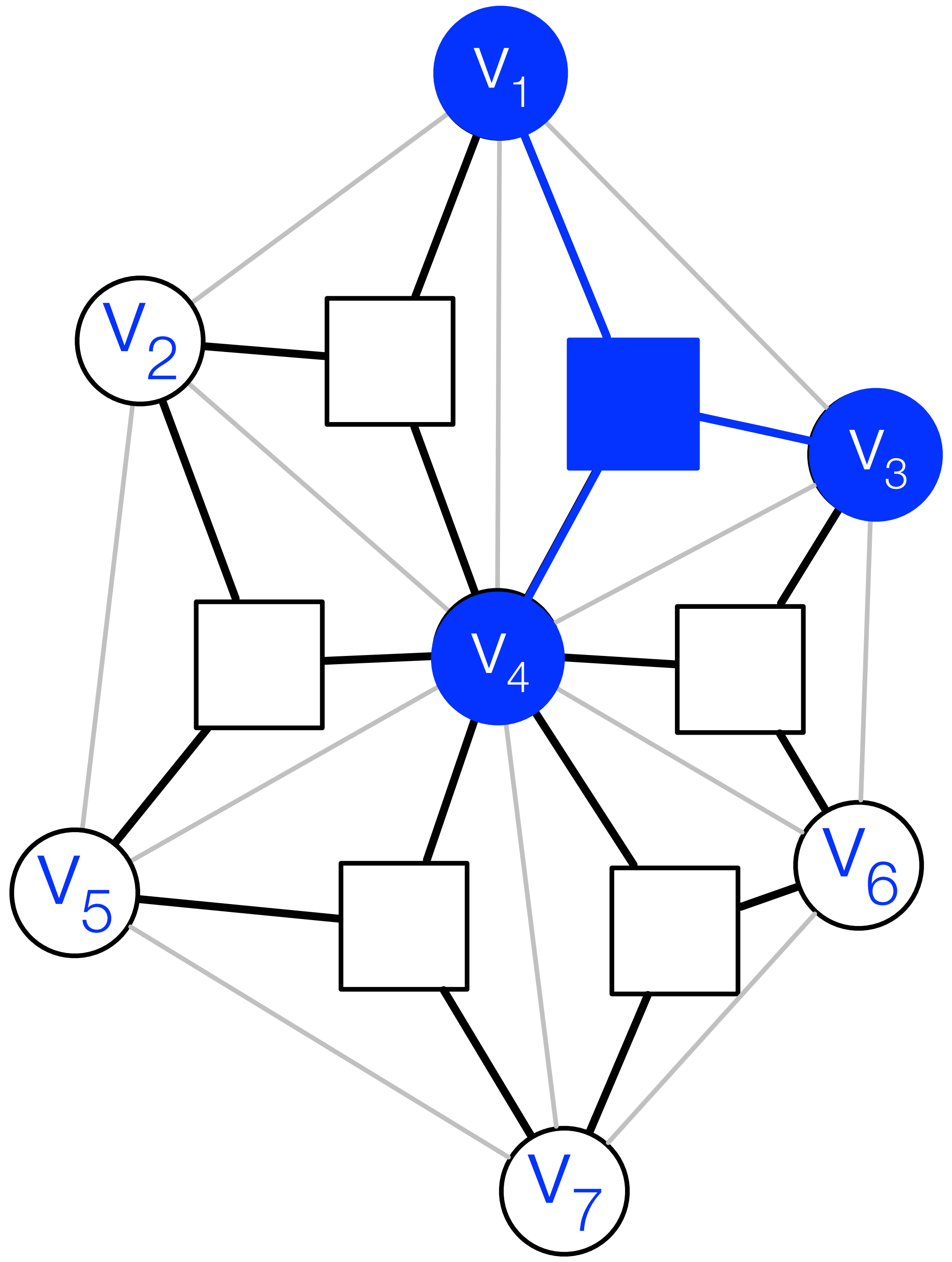

```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l
  W
  B
end

% gr
exte
exte

% co
func
  t.
  t.
end

expo
  ap
end
```

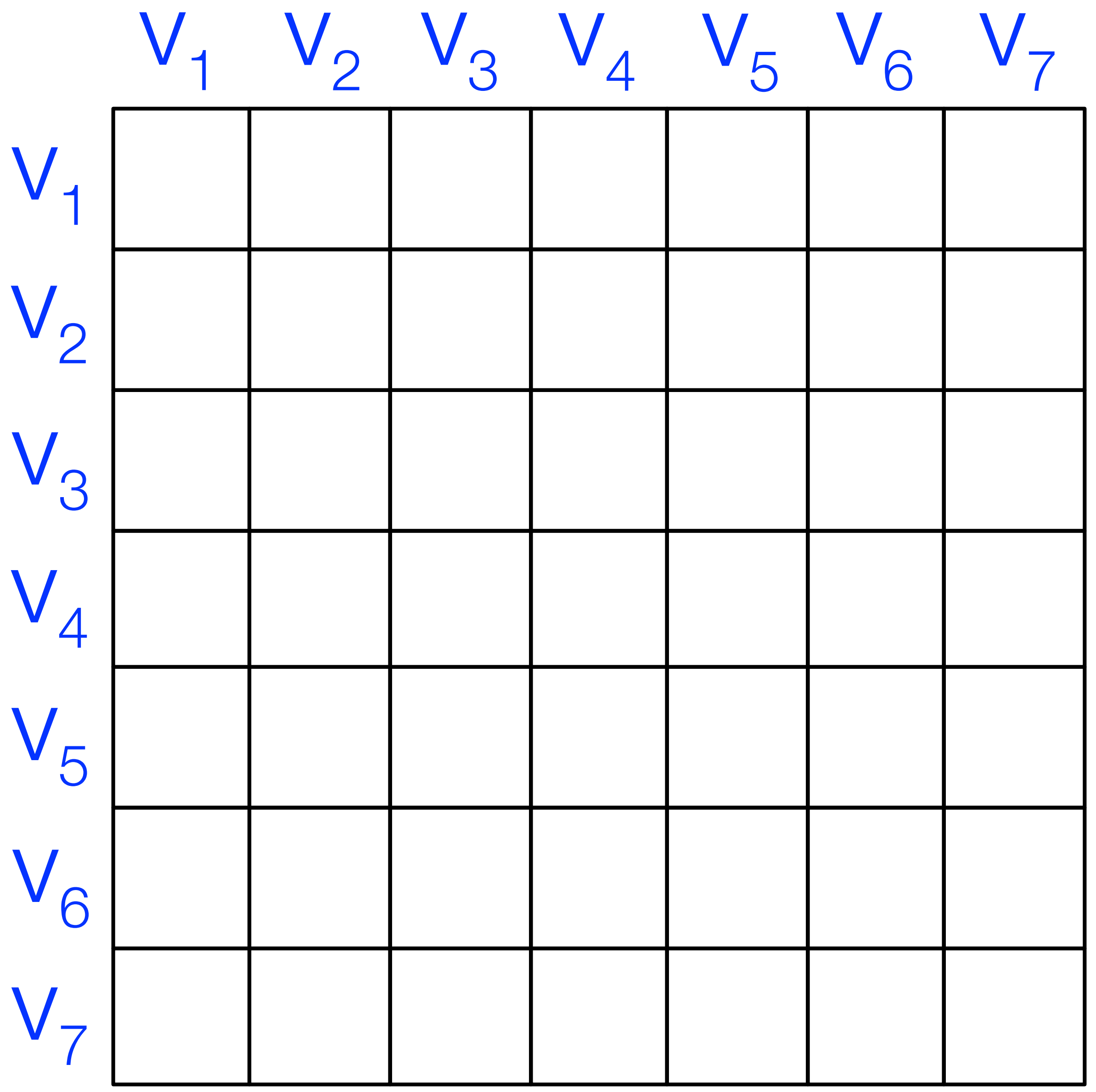
```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

e +;

Statics Triangular Neo-Hookean FEM Simulation





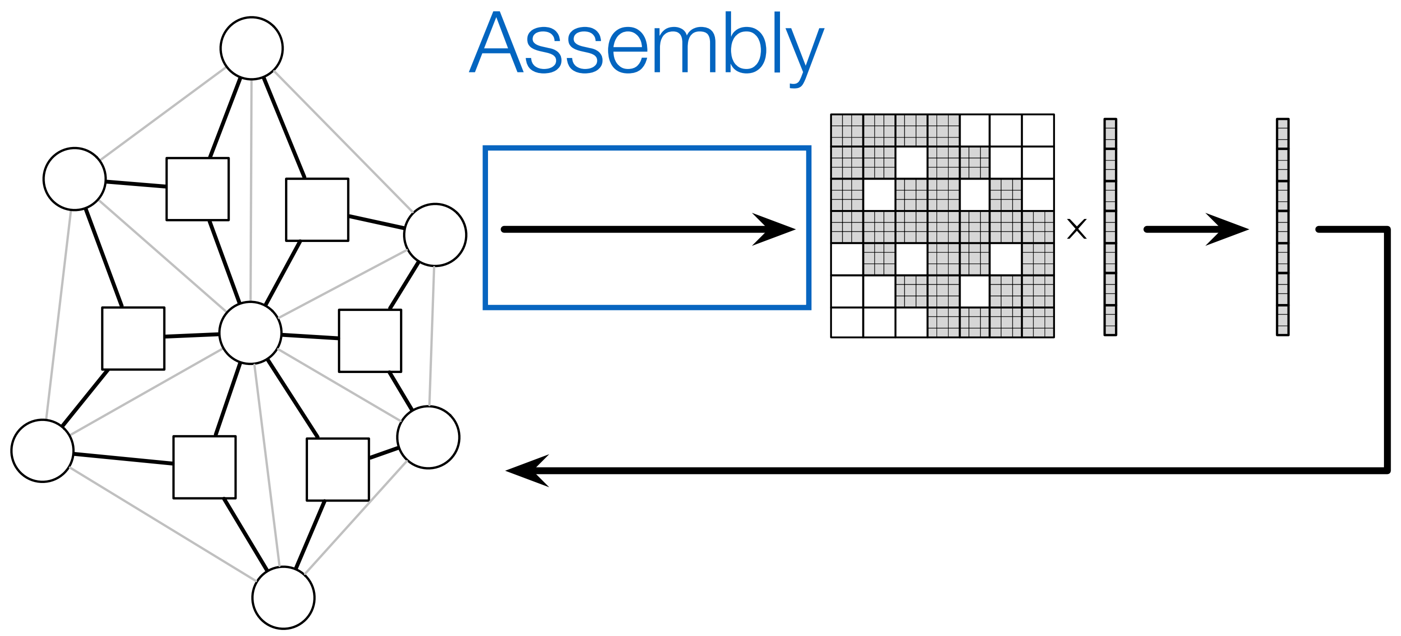


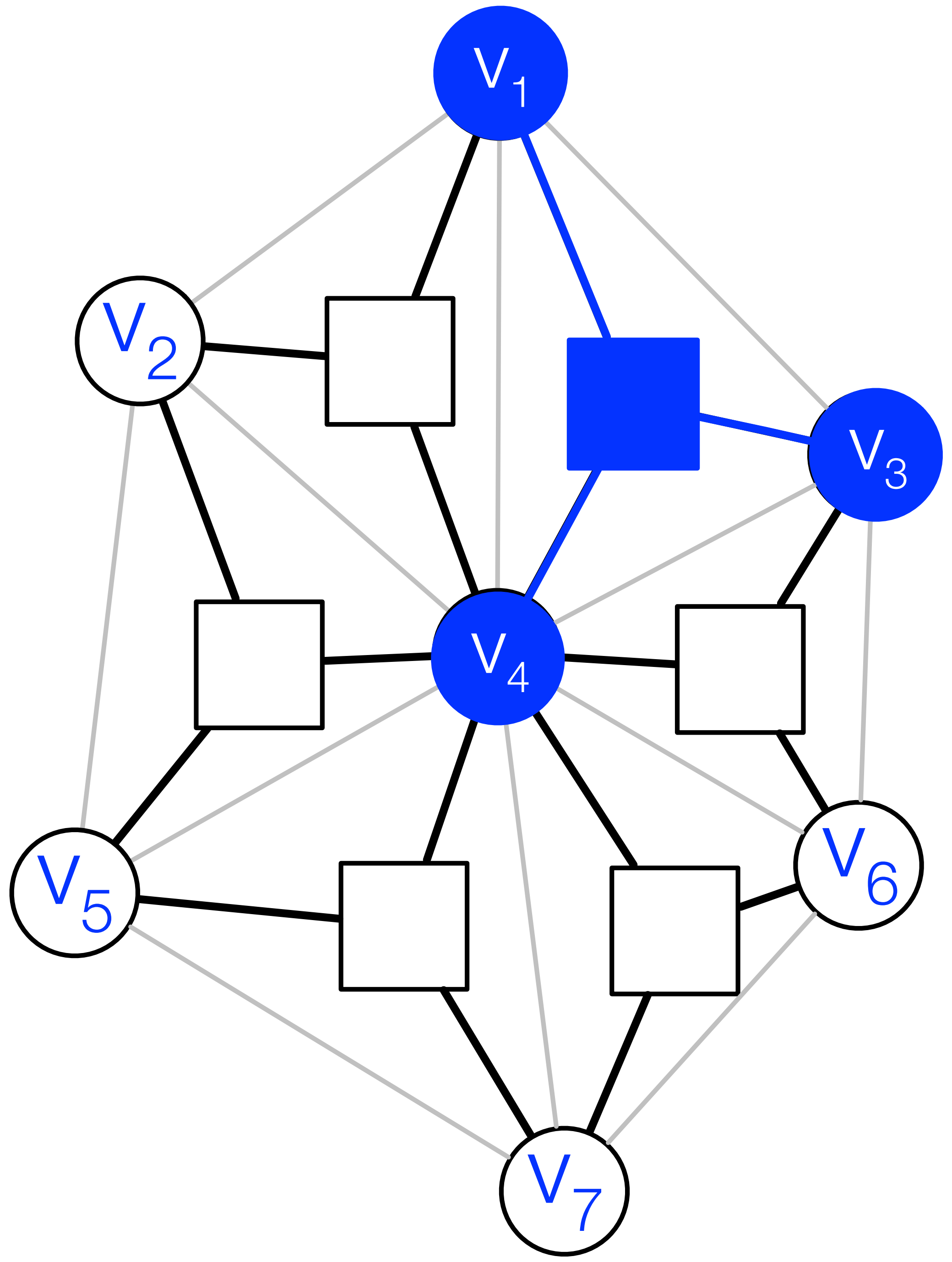

```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l
  W
  B
end

% gr
exte
exte

% co
func
  t.
  t.
end

expo
  ap
end
```

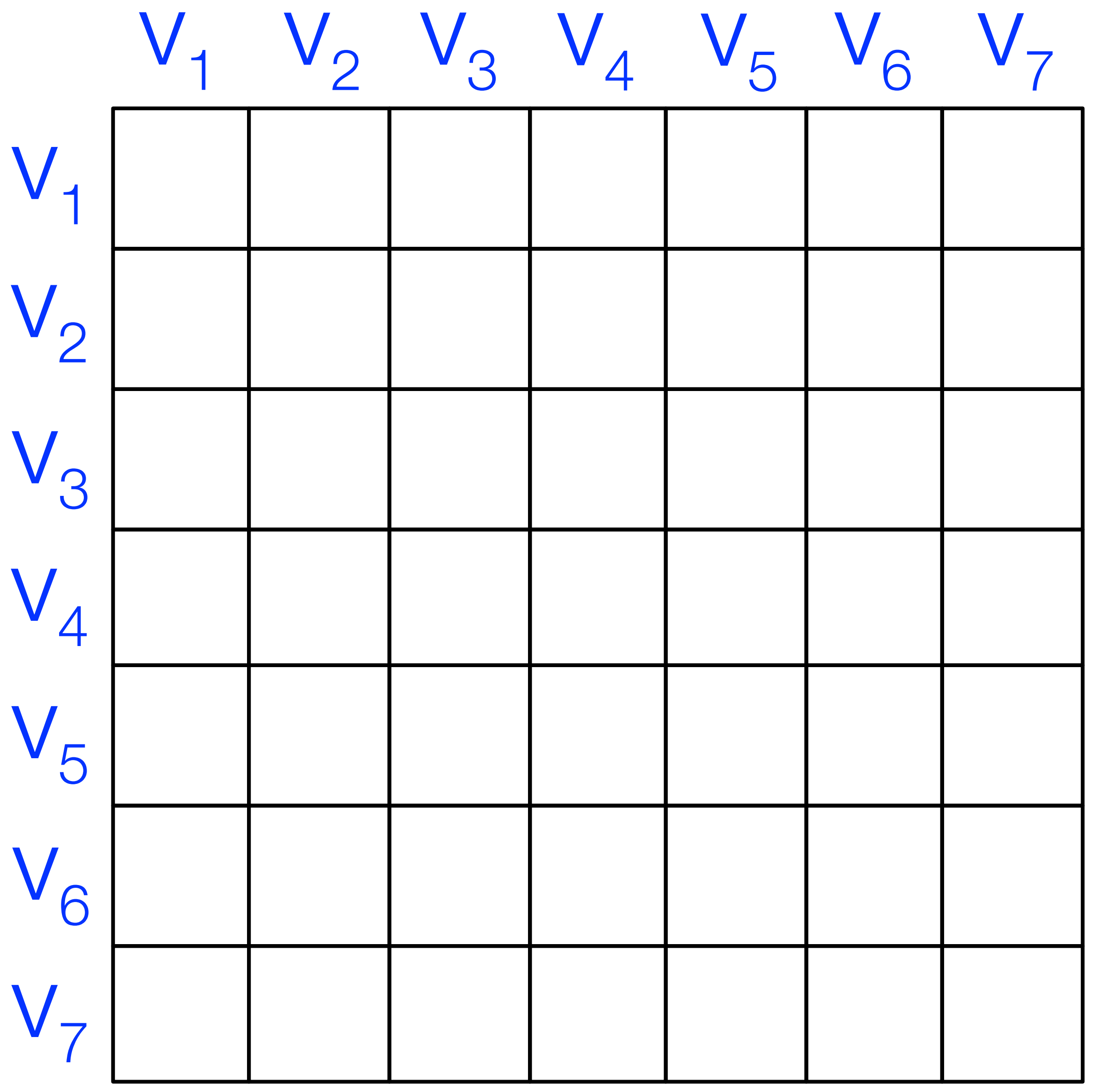
```
e +;
```

```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation





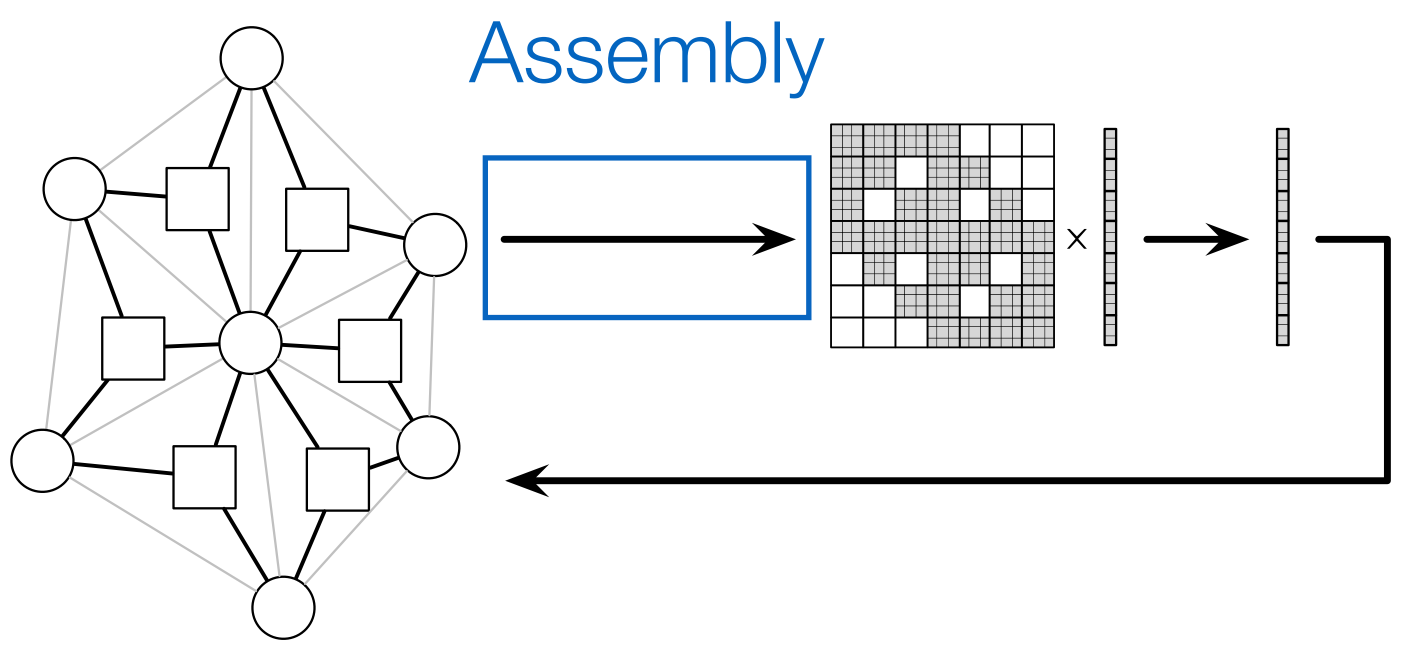


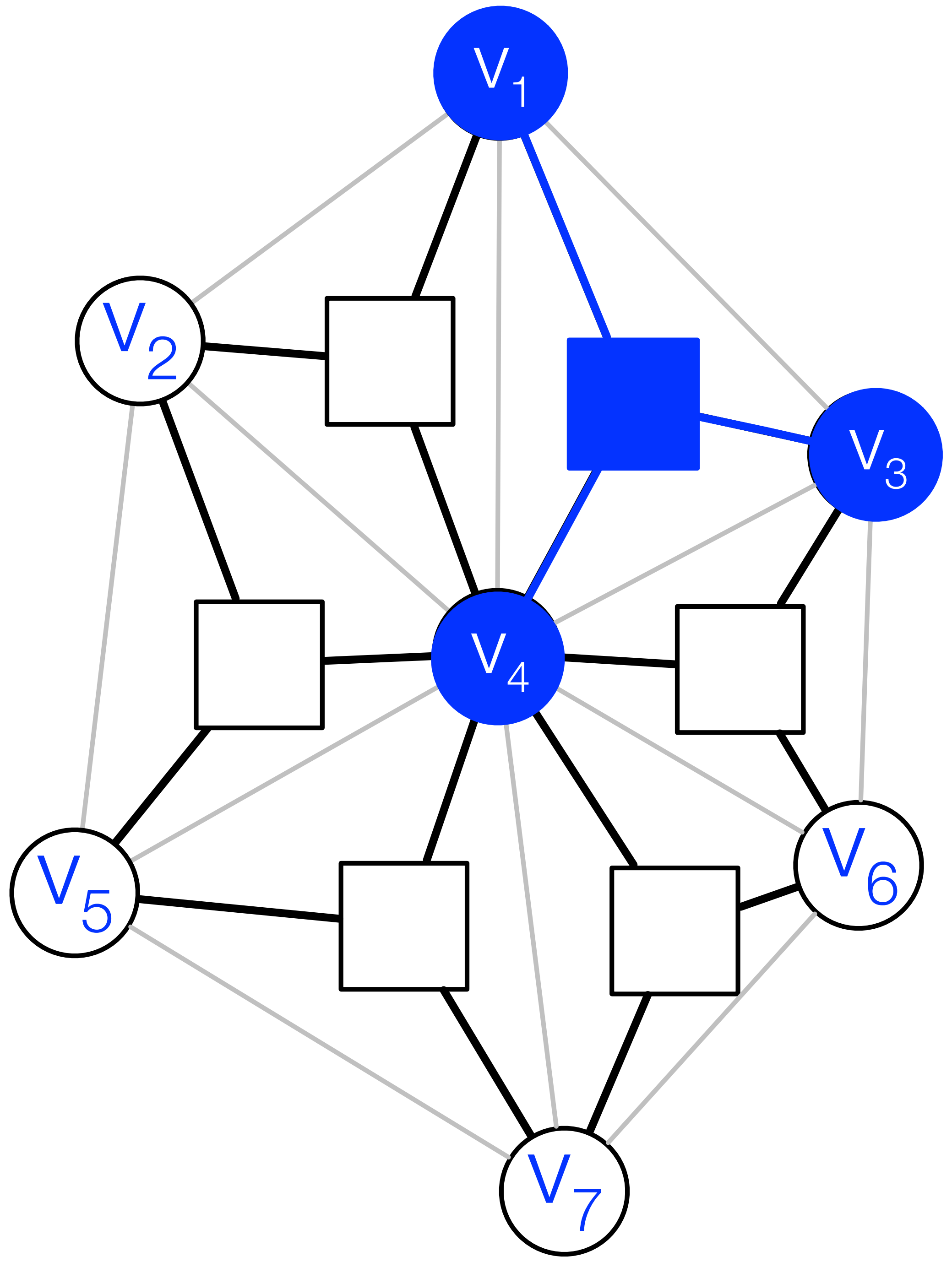

```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l
  W
  B
end

% gr
exte
exte

% co
func
  t.
  t.
end

expo
  ap
end
```

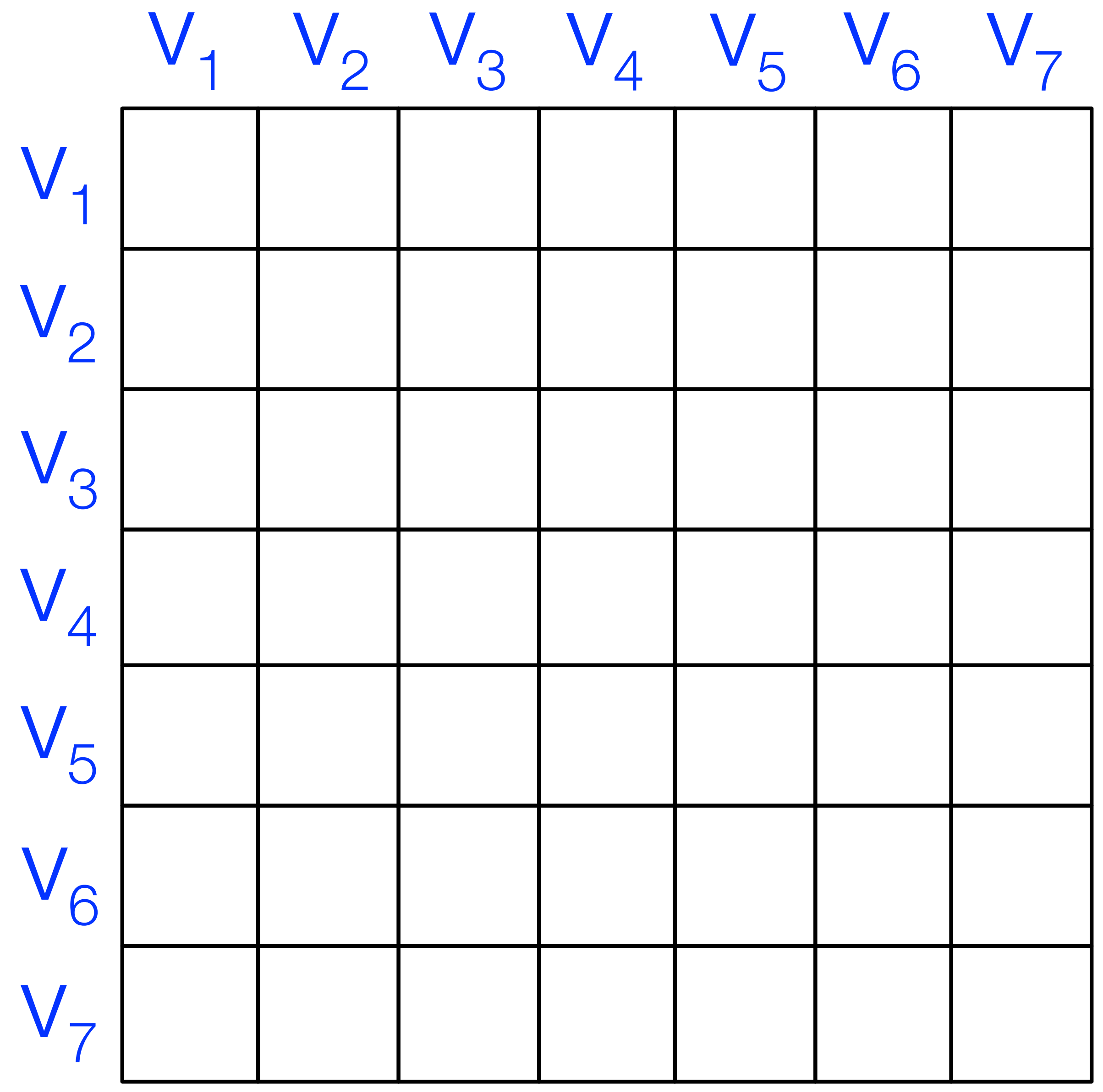
```
e +;
```

```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation





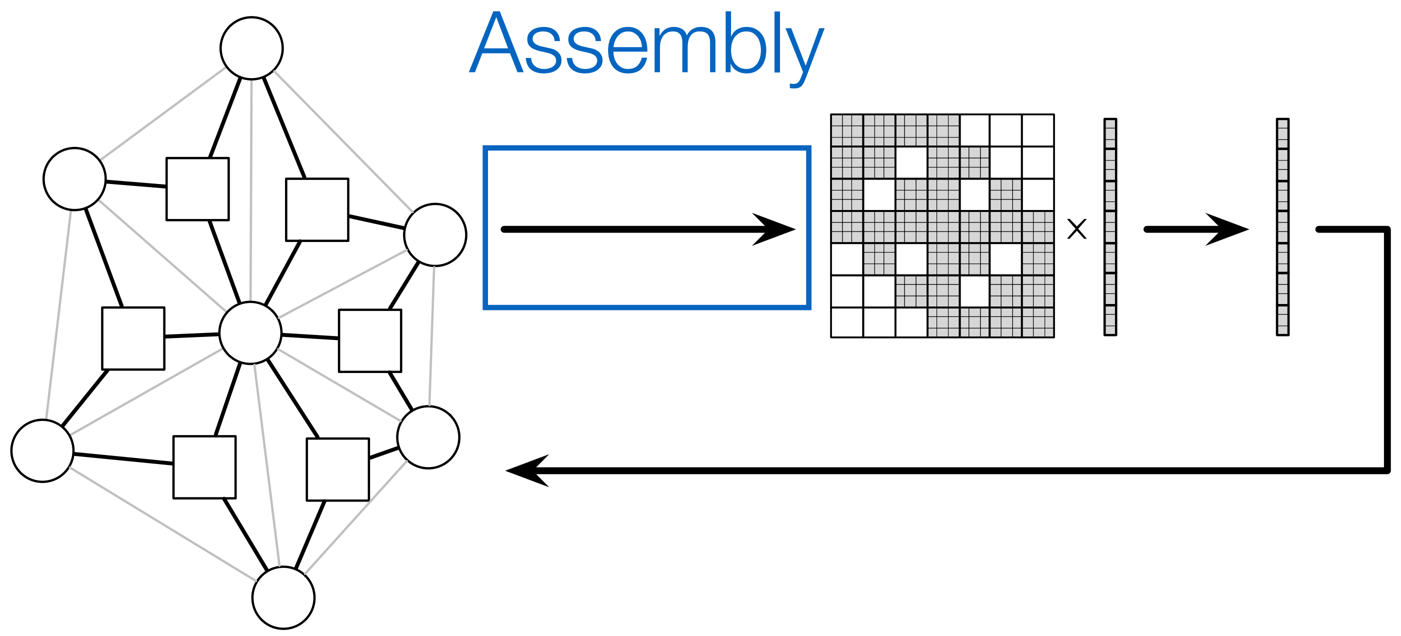


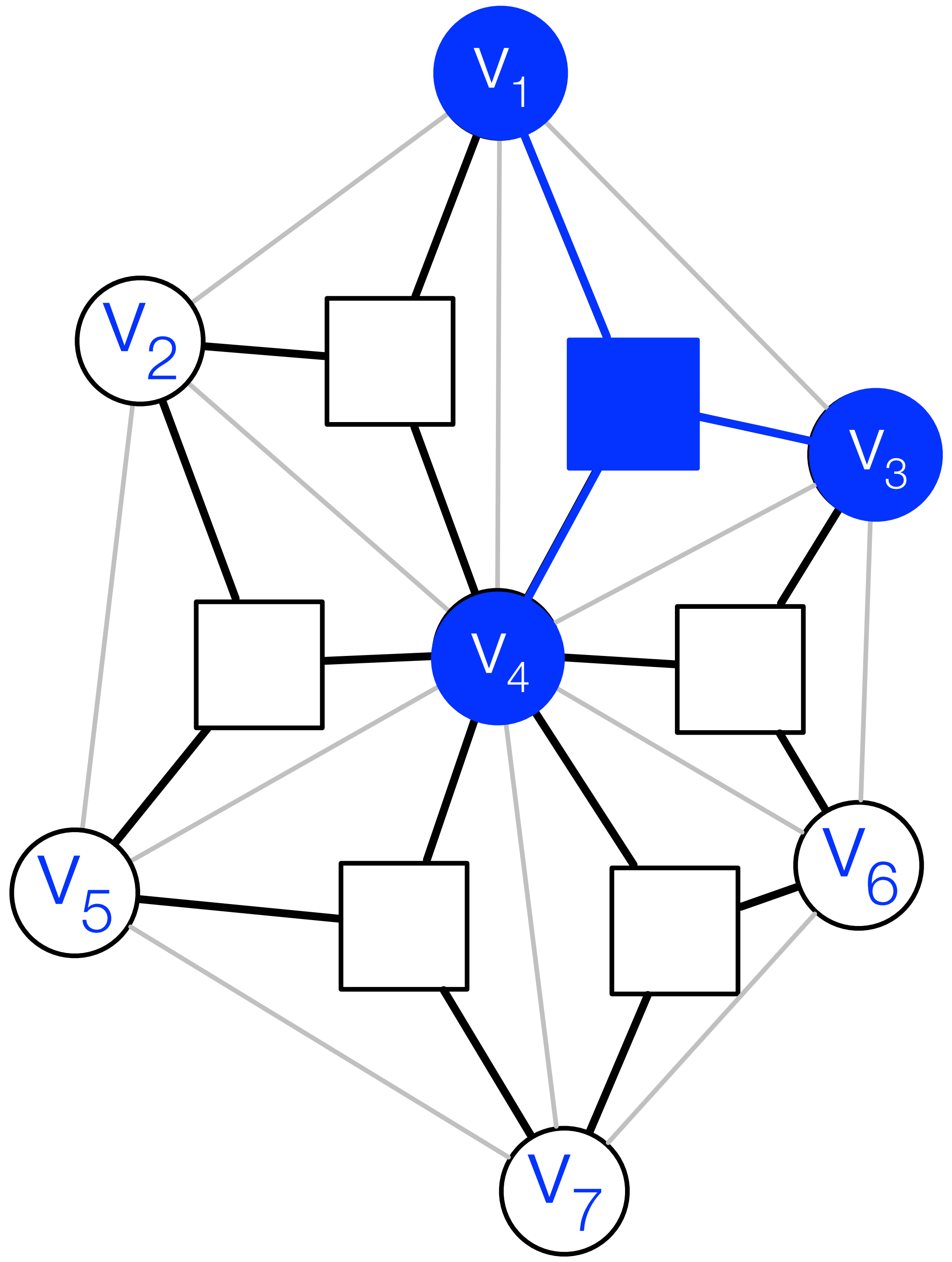

```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l
  W
  B
end

% gr
exte
exte

% co
func
  t.
  t.
end

expo
  ap
end
```

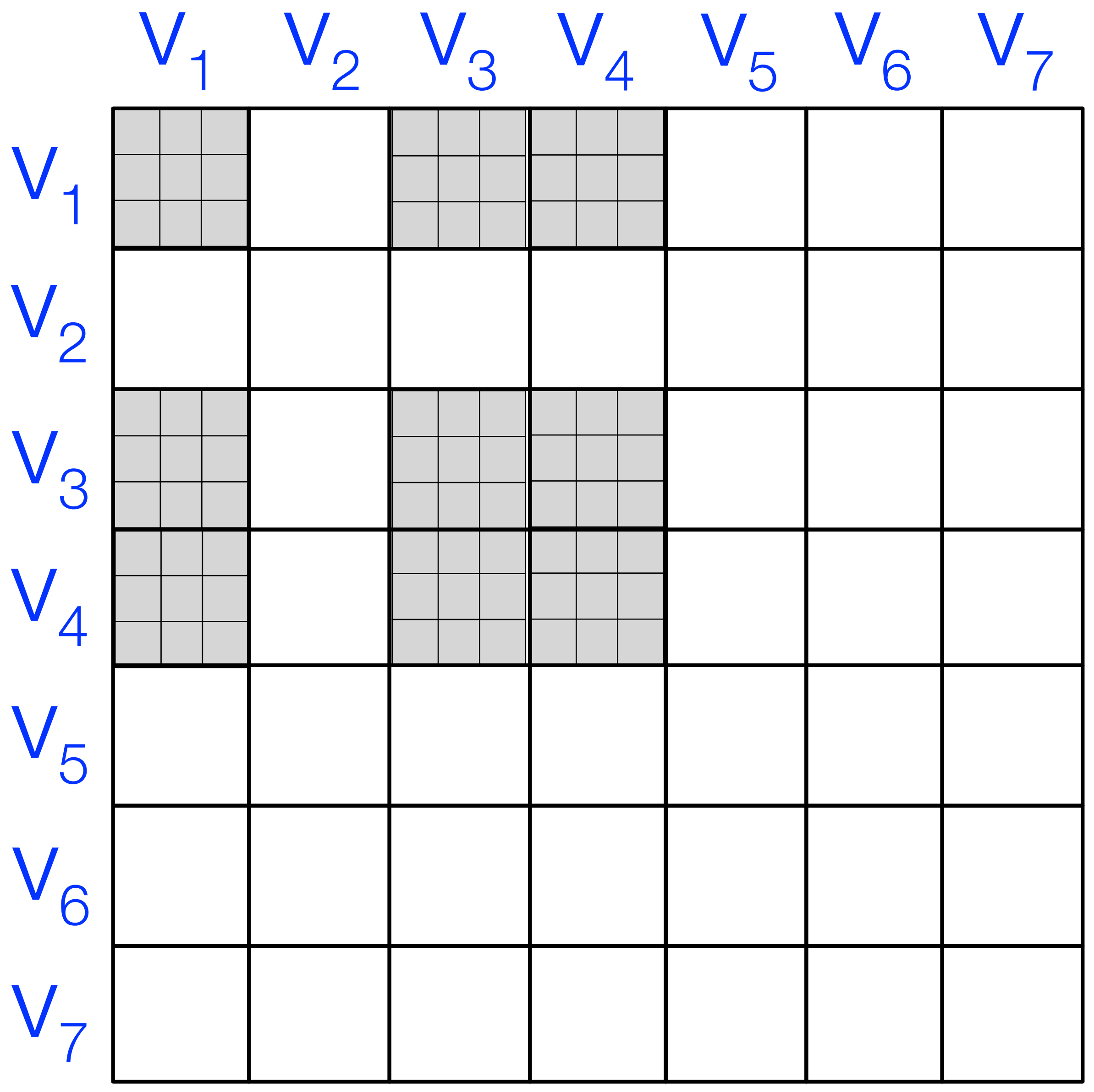
```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
     -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

e +;

Statics Triangular Neo-Hookean FEM Simulation





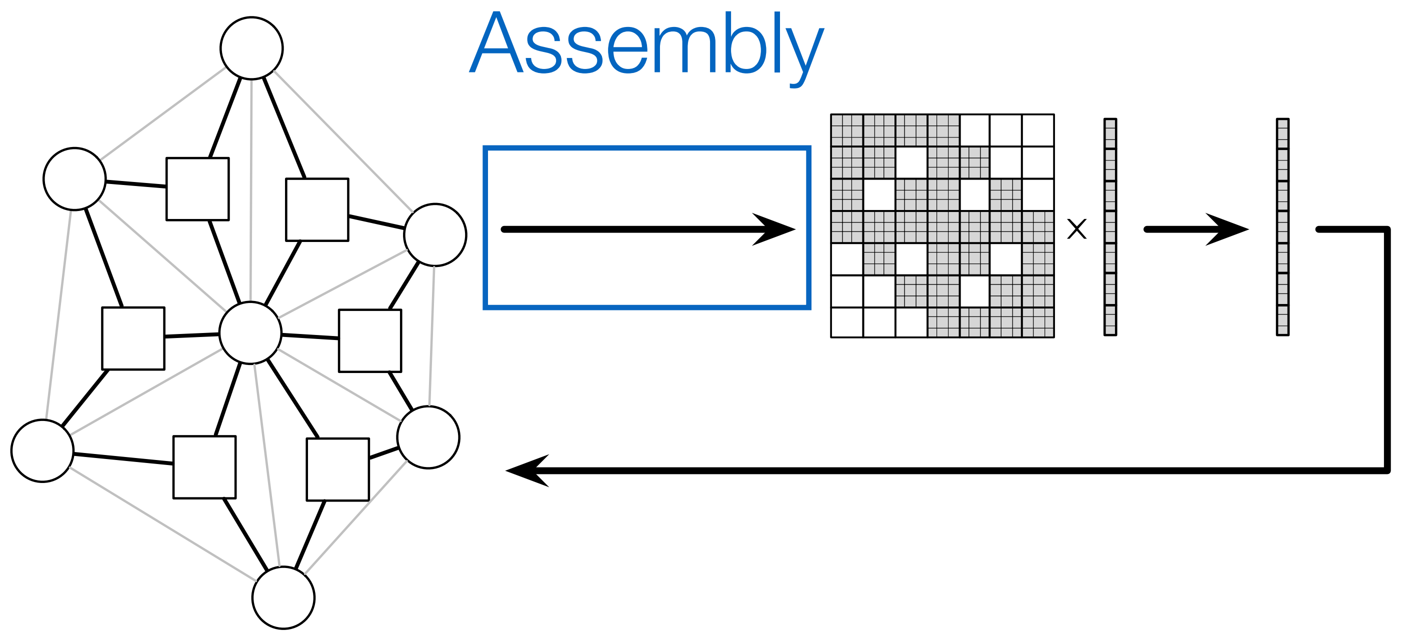


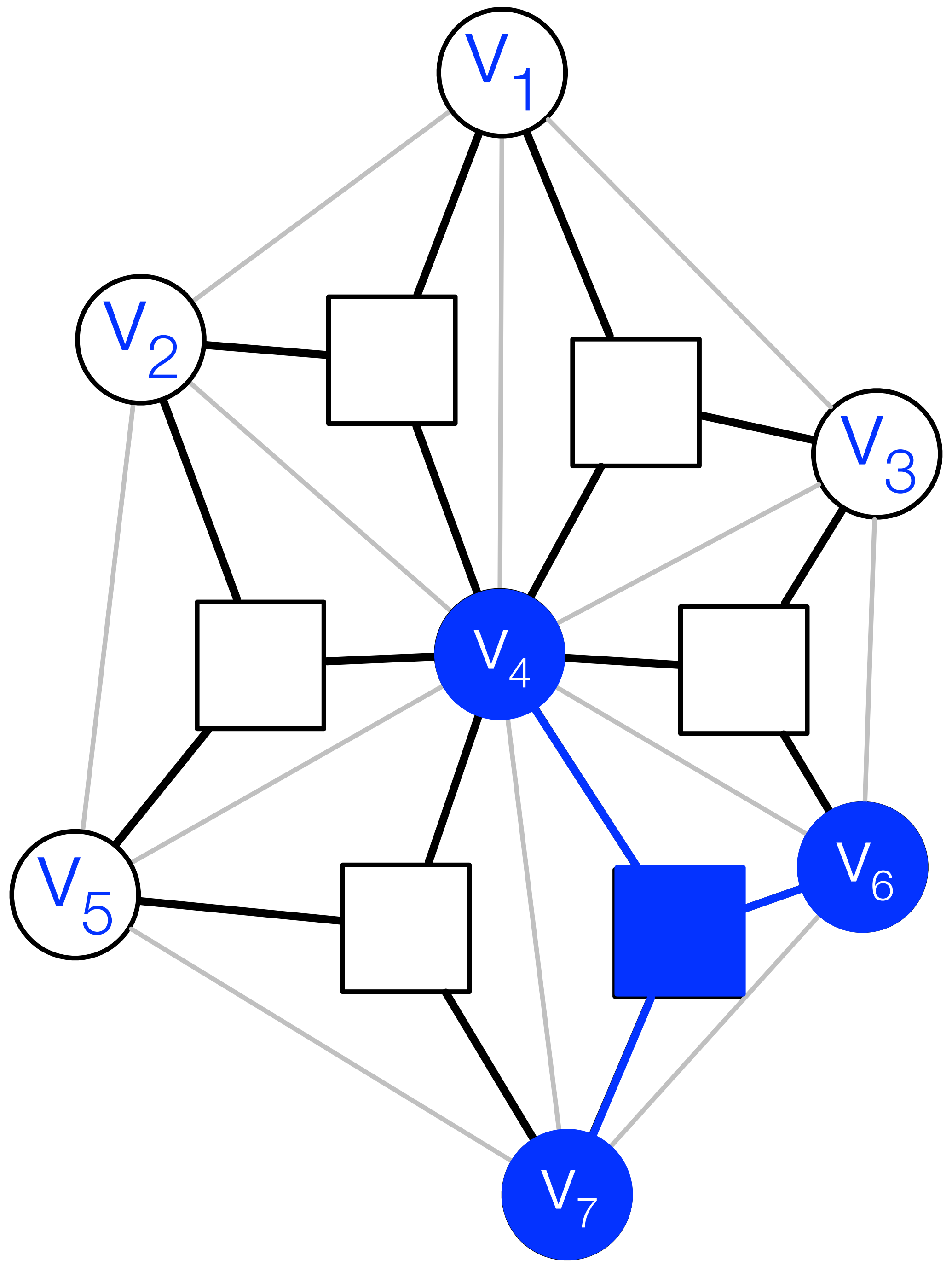

```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l
  W
  B
end

% gr
exte
exte

% co
func
  t.
  t.
end

expo
  ap
end
```

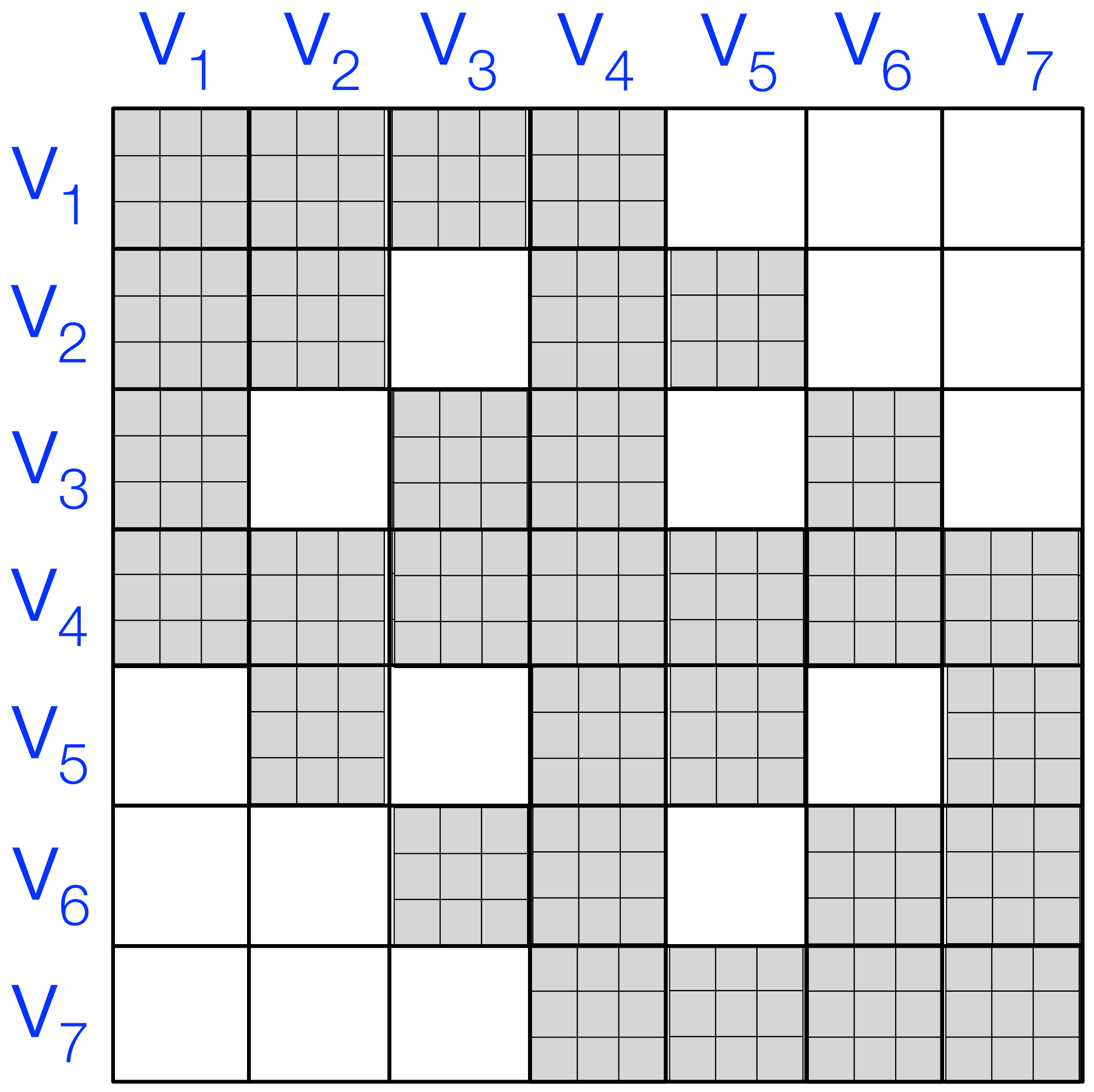
```
e +;
```

```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation





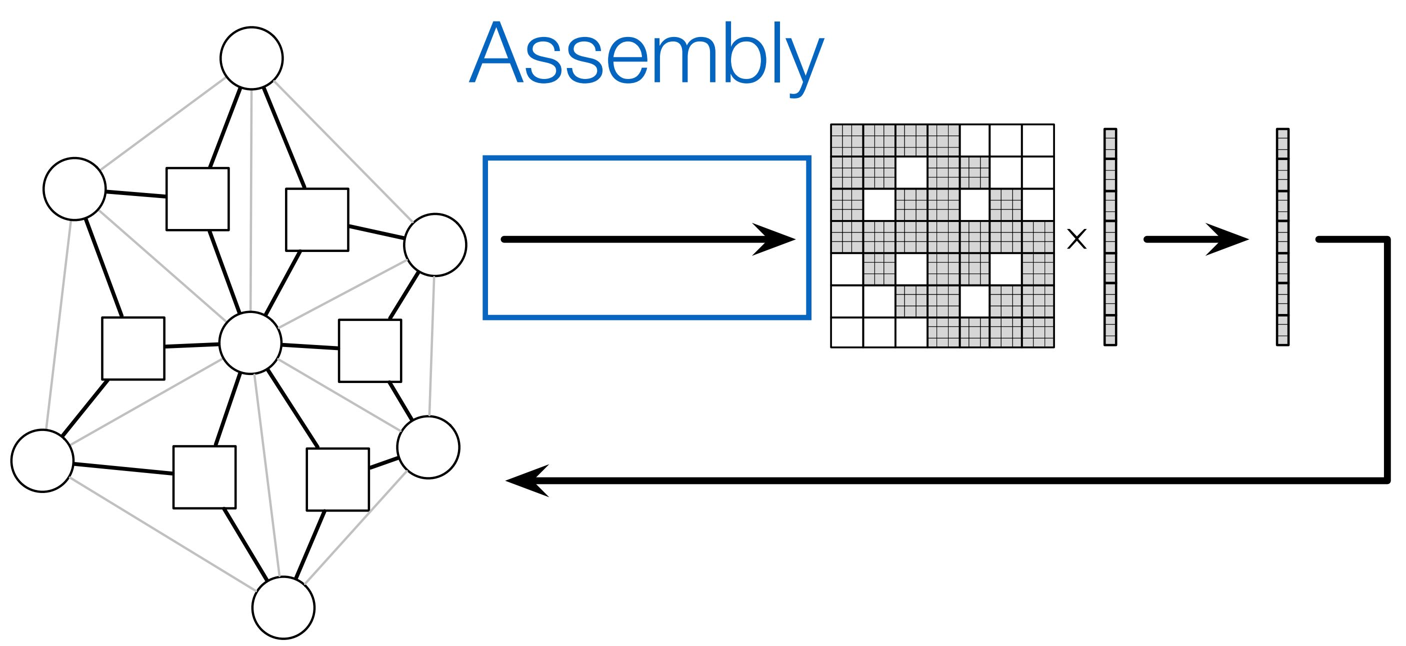



```
element Vertex
  x  : vector[3](float);    % position
  v  : vector[3](float);    % velocity
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l
  W
  B
end

% gr
exte
exte

% co
func
  t.
  t.
end

expo
  ap
end
```

```
e +;
```

```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation

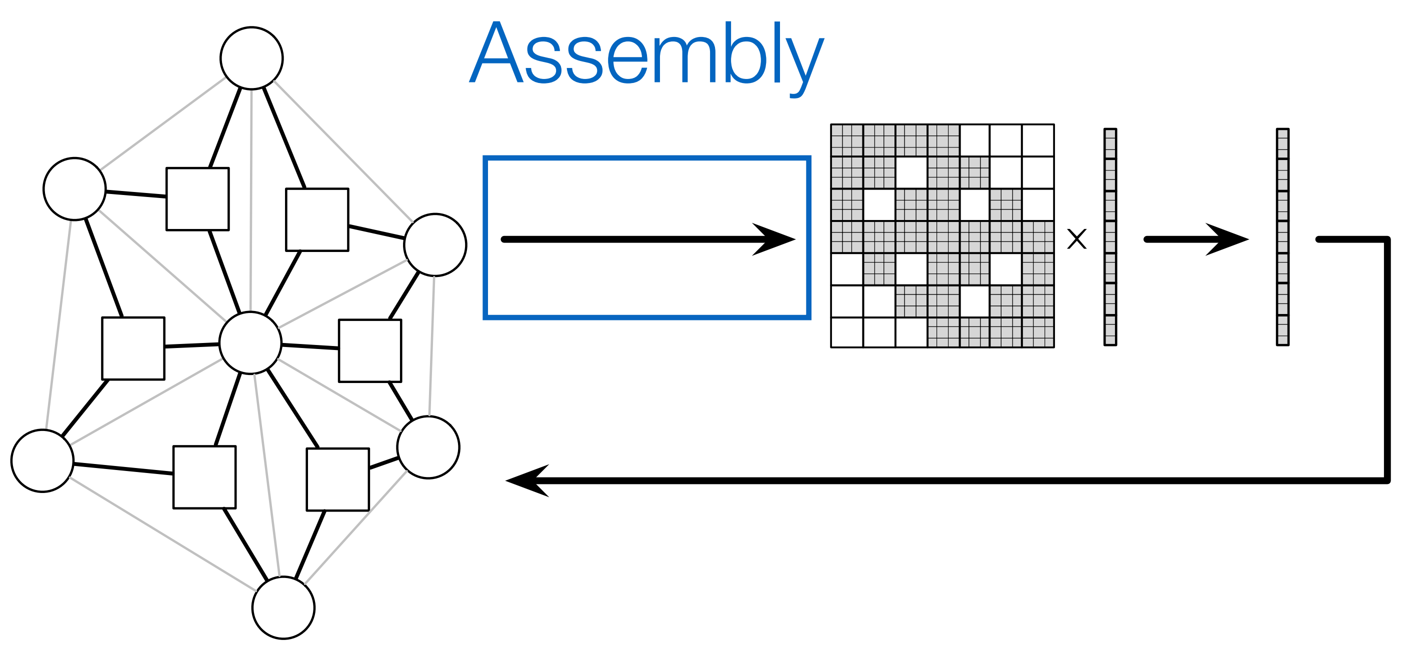



```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end
```

```
% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    // assemble force vector
    // compute new position
  end
end
```

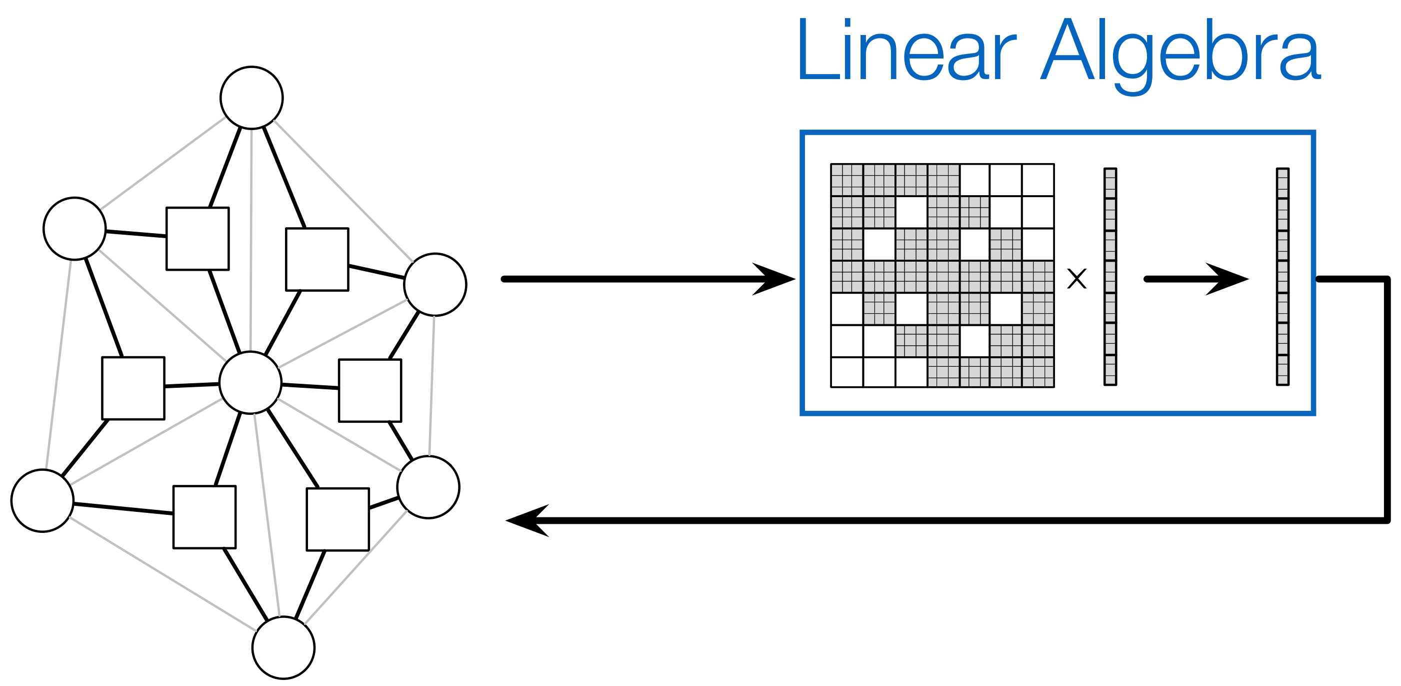
Statics Triangular Neo-Hookean FEM Simulation




```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end

% computes the force of a triangle on its vertices
func triangle_force(t : Triangle, v : (Vertex*3))
    -> f : vector[verts](vector[3](float))
  for i in 0:3
    f(v(i)) += compute_force(t,v,i);
  end
end

% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    f = map triangle_force     to triangles reduce +;
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



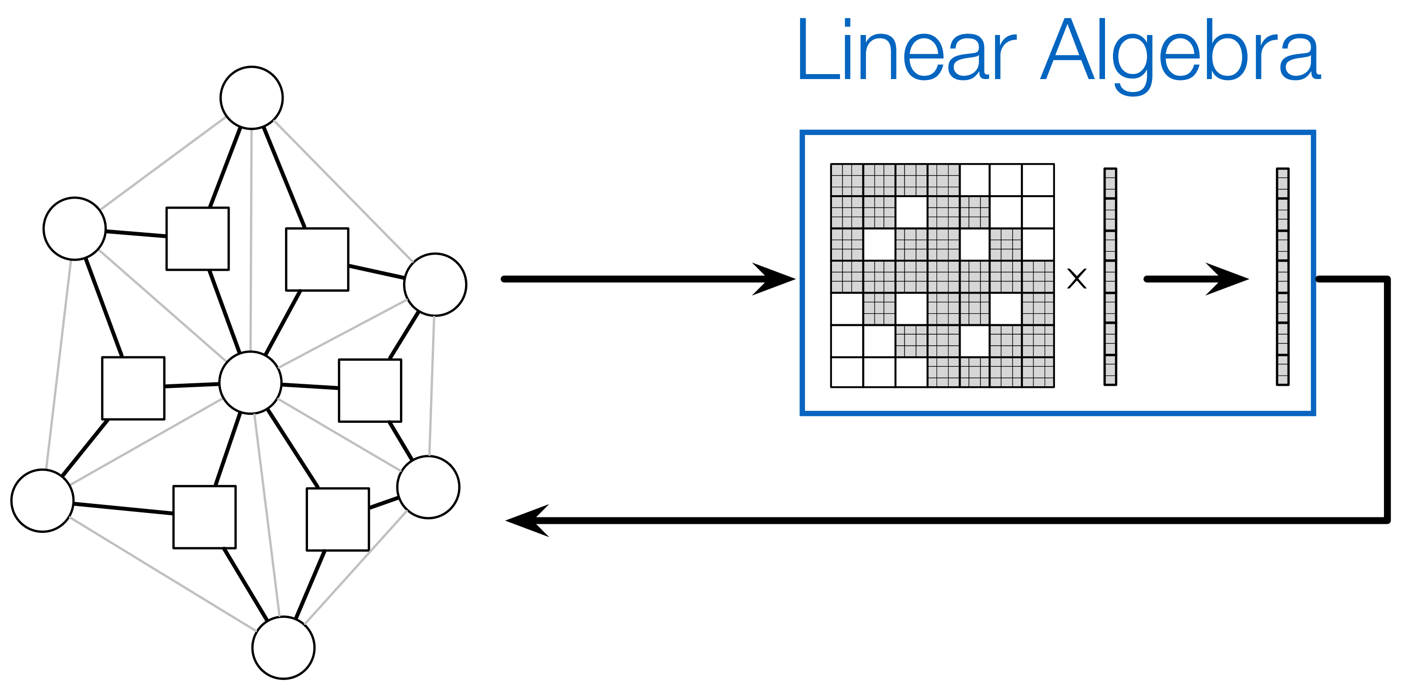
$$x_{t+1} = x_t + K^{-1}(f_{external} - f)$$

```
verts.x = verts.x + K \ (verts.fe - f);
```

```
element Vert
  x  : vecto
  v  : vecto
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l  : float;               % lame's first parameter
  W  : float;               % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
e
 v : (Vertex*3))
[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end

% computes the force of a triangle on its vertices
func triangle_force(t : Triangle, v : (Vertex*3))
    -> f : vector[verts](vector[3](float))
  for i in 0:3
    f(v(i)) += compute_force(t,v,i);
  end
end

% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    f = map triangle_force     to triangles reduce +;
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



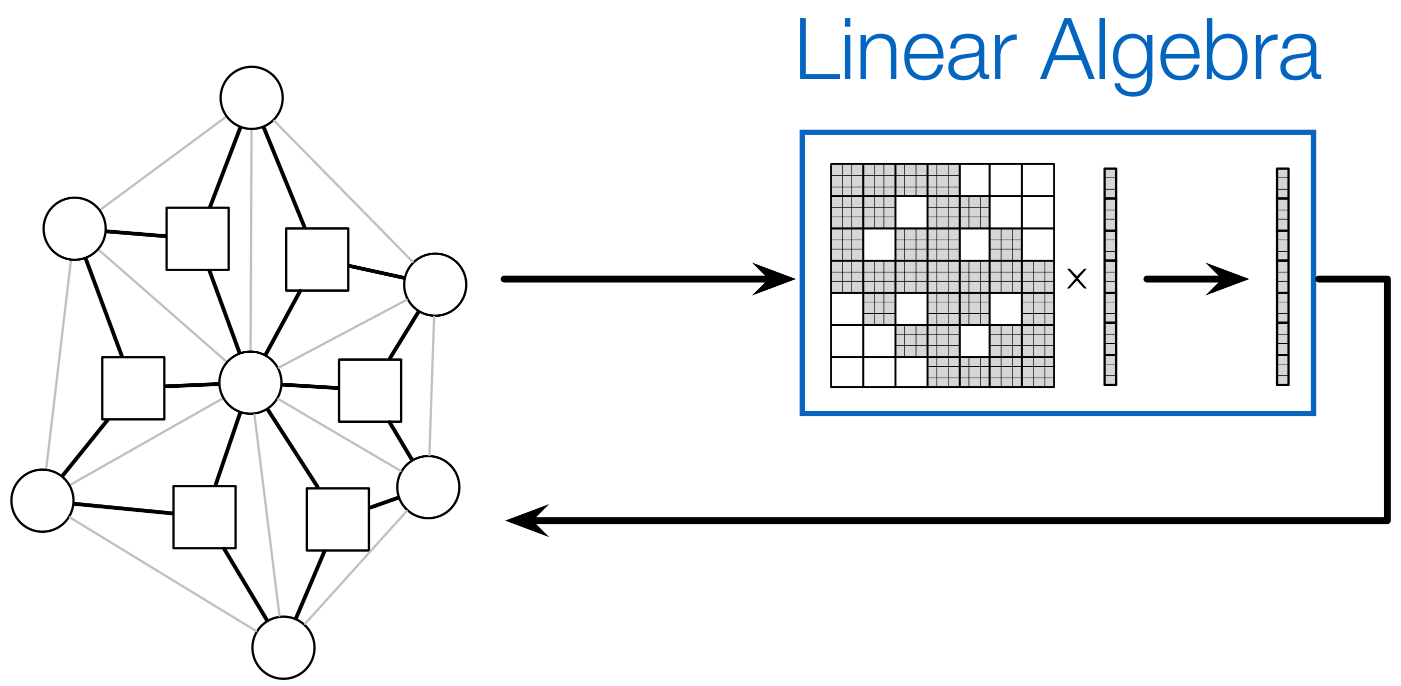
$$x_{t+1} = x_t + K^{-1}(f_{external} - f)$$

```
verts.x = verts.x + K \ (verts.fe - f);
```

```
element Vert
  x  : vecto
  v  : vecto
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l  : float;               % lame's first parameter
  W  : float;               % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
                                              e
                                               v : (Vertex*3))
                                              [3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end

% computes the force of a triangle on its vertices
func triangle_force(t : Triangle, v : (Vertex*3))
    -> f : vector[verts](vector[3](float))
  for i in 0:3
    f(v(i)) += compute_force(t,v,i);
  end
end

% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    f = map triangle_force     to triangles reduce +;
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



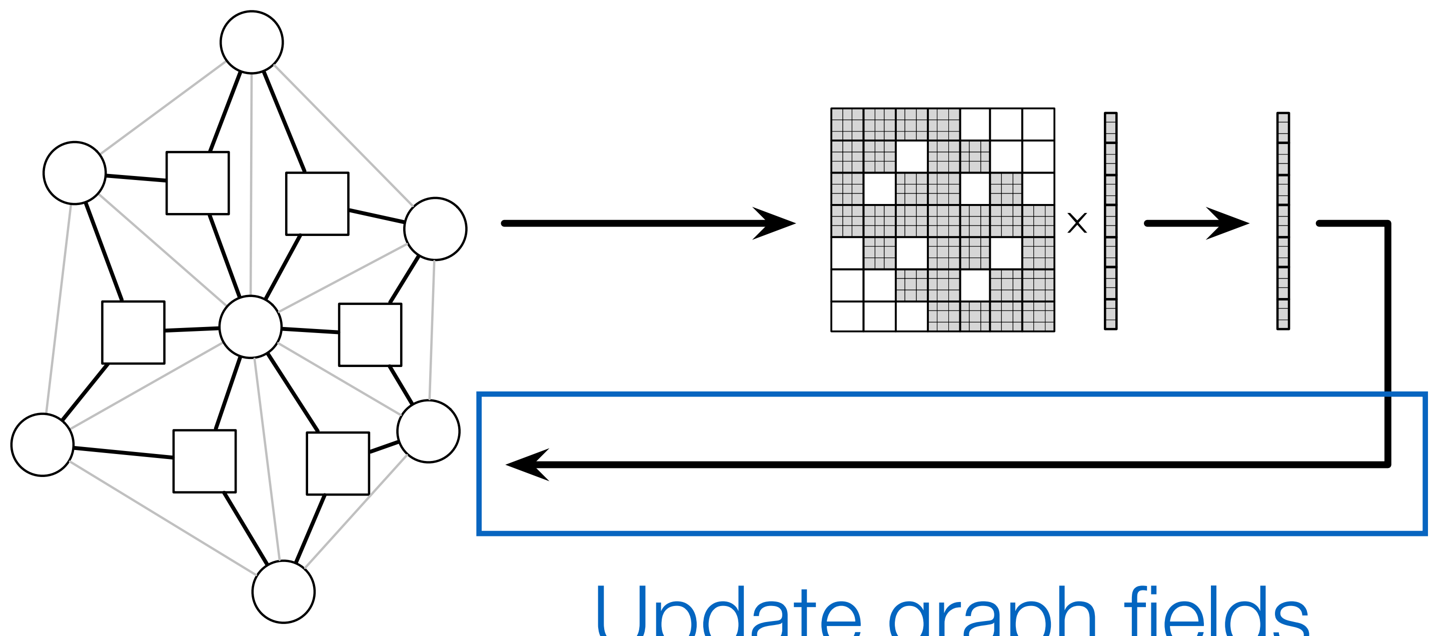
$$x_{t+1} = x_t + K^{-1}(f_{external} - f)$$

```
verts.x = verts.x + K \ (verts.fe - f);
```

```
element Vert
  x  : vecto
  v  : vecto
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
                                             e
                                             v : (Vertex*3))
                                             [3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end

% computes the force of a triangle on its vertices
func triangle_force(t : Triangle, v : (Vertex*3))
    -> f : vector[verts](vector[3](float))
  for i in 0:3
    f(v(i)) += compute_force(t,v,i);
  end
end

% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    f = map triangle_force     to triangles reduce +;
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



Update graph fields

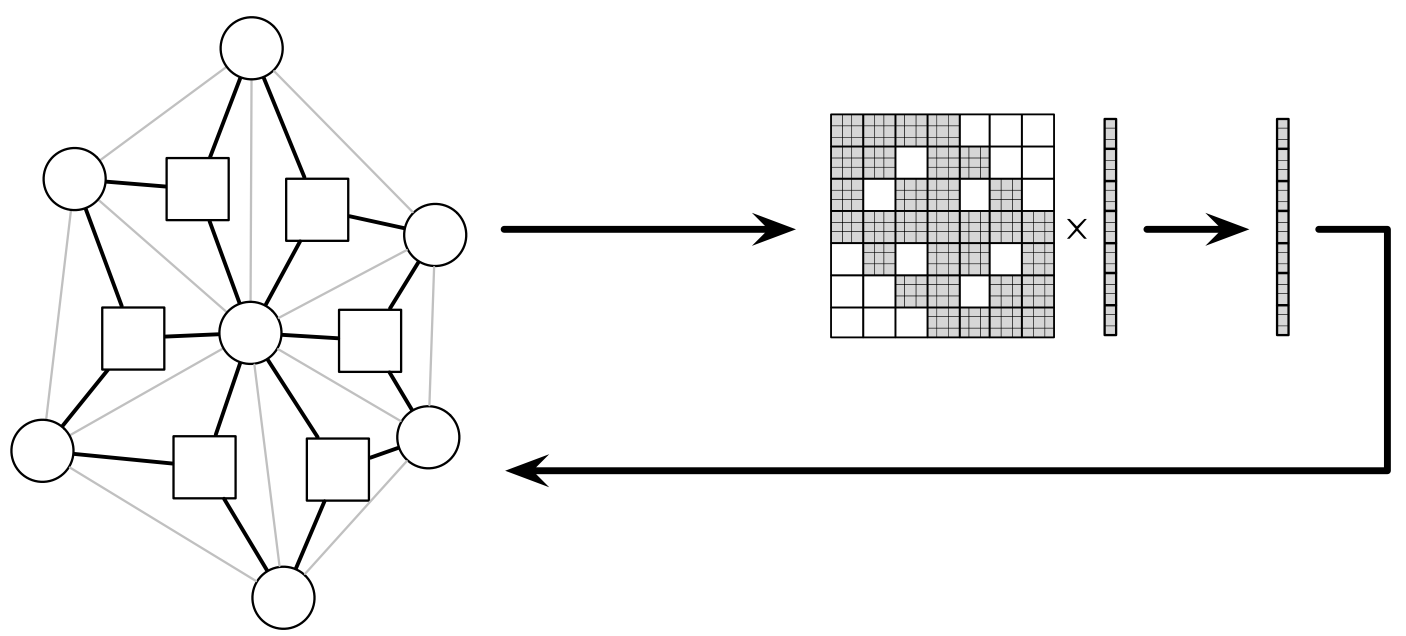
$$x_{t+1} = x_t + K^{-1}(f_{external} - f)$$

```
verts.x = verts.x + K \ (verts.fe - f);
```

```
element Vert
  x  : vecto
  v  : vecto
  fe : vector[3](float);    % external force
end

element Triangle
  u  : float;               % shear modulus
  l  : float;               % lame's first parameter
  W  : float;               % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

```
                                    e
                                     v : (Vertex*3))
                                    [3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end

% computes the force of a triangle on its vertices
func triangle_force(t : Triangle, v : (Vertex*3))
    -> f : vector[verts](vector[3](float))
  for i in 0:3
    f(v(i)) += compute_force(t,v,i);
  end
end

% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    f = map triangle_force     to triangles reduce +;
    // compute new position
  end
end
```

Statics Triangular Neo-Hookean FEM Simulation



```
element Vertex
  x  : vector[3](float);   % position
  v  : vector[3](float);   % velocity
  fe : vector[3](float);   % external force
end

element Triangle
  u  : float;              % shear modulus
  l  : float;              % lame's first parameter
  W  : float;              % volume
  B  : matrix[3,3](float); % strain-displacement
end

% graph vertices and triangle hyperedges
extern verts     : set{Vertex};
extern triangles : set{Triangle}(verts, verts, verts);

% compute triangle area
func compute_area(inout t : Triangle, v : (Vertex*3))
  t.B = compute_B(v);
  t.W = det(B) / 2.0;
end

export func init()
  apply compute_area to triangles;
end
```

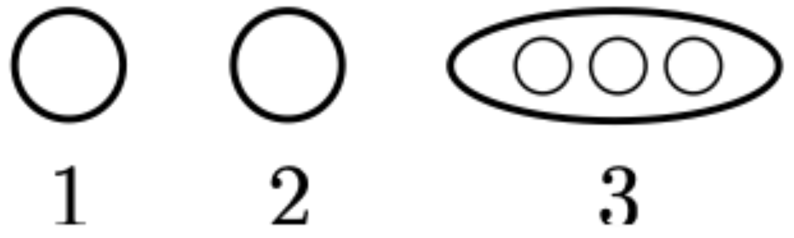
```
% computes the stiffness of a triangle
func triangle_stiffness(t : Triangle, v : (Vertex*3))
    -> K : matrix[verts,verts](matrix[3,3](float))
  for i in 0:3
    for j in 0:3
      K(v(i),v(j)) += compute_stiffness(t,v,i,j);
    end
  end
end

% computes the force of a triangle on its vertices
func triangle_force(t : Triangle, v : (Vertex*3))
    -> f : vector[verts](vector[3](float))
  for i in 0:3
    f(v(i)) += compute_force(t,v,i);
  end
end

% newton's method
export func newton_method()
  while abs(f - verts.fe) > 1e-6
    K = map triangle_stiffness to triangles reduce +;
    f = map triangle_force     to triangles reduce +;
    verts.x = verts.x + K \ (verts.fe - f);
  end
end
```

Collection-Oriented Languages

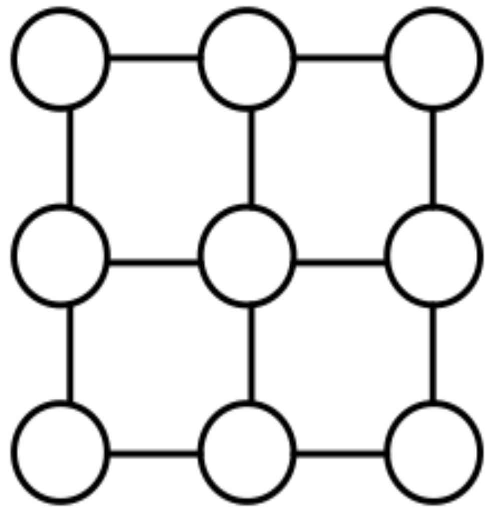
Lists
Lisp M58

Sets
SETL S70

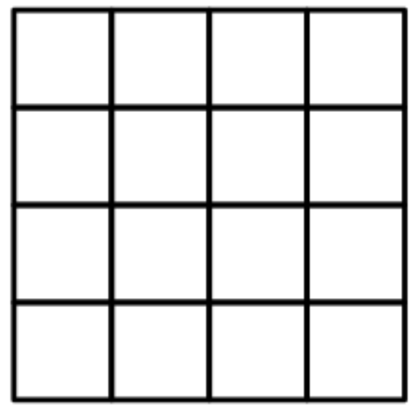
Nested Sequences
NESL B94



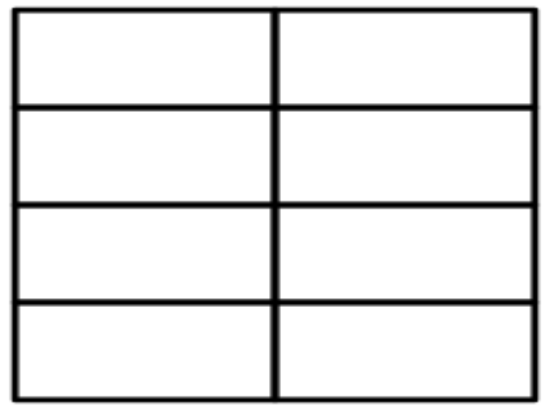
Grids
Sejits S09, Halide



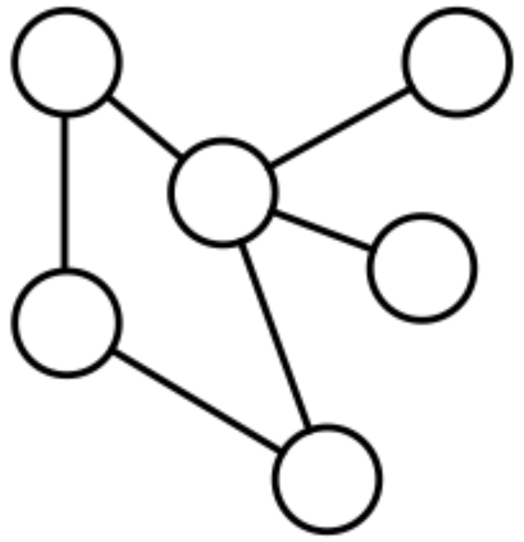
Arrays
APL I62
NumPy



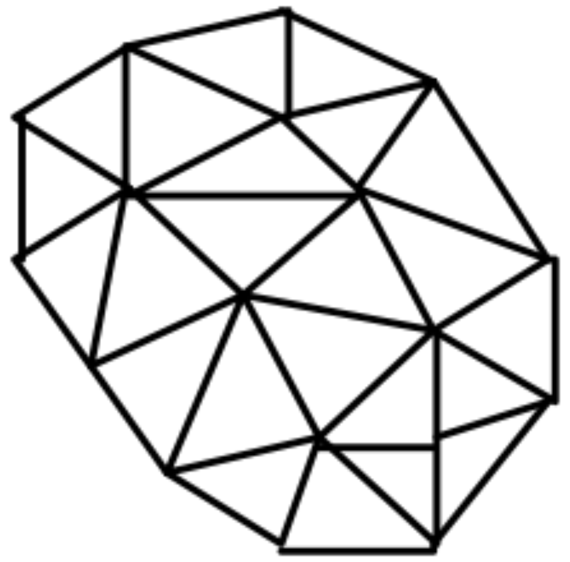
Relations
Relational Algebra C70,



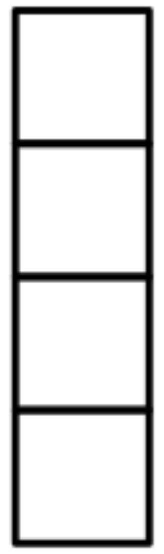
Graphs
GraphLab L10



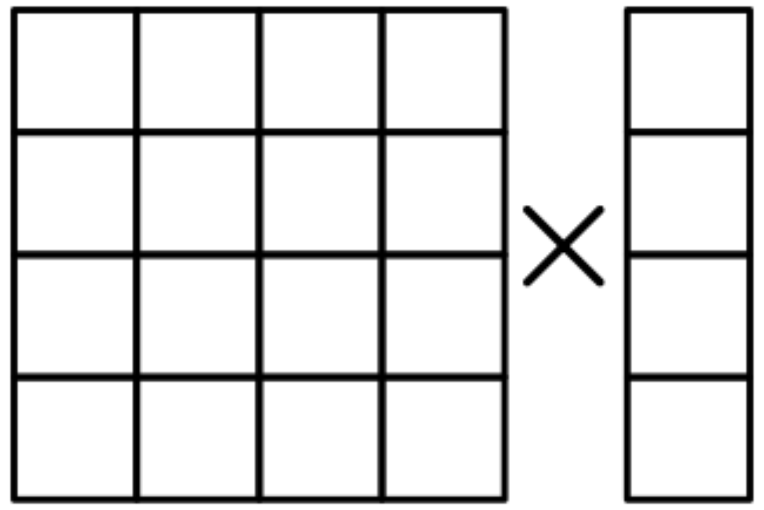
Meshes
Liszt D11



Vectors
Vector Model B90



Matrices and Tensors
Matlab M79, taco K17



A collection-oriented programming model provides collective operations on some collection/abstract data structure

Lecture Overview