Discretization

- Computers cannot naturally handle continuous properties
- But the granularity of a simulation is not apparent on sufficiently long length scales

"Physical Laws"

finite-element decomposition
lattice regularization

do coarse graining
multi-scale analysis

discrete models
Coarse graining

- Process of spatial averaging over local regions
- $3 \times 3$ averaging of binary cells gives distinct 10 levels
- Recover continuous scalar field in large region limit
Coarse graining

- E.g., 3x3 averaging of 4-state clocks
- Recover continuous vector field in large region limit
Hierarchy of length scales

key requirement: \( l \ll \xi \ll L \)
Number representations

- The obvious strategies ...
  - simple enumeration:
  - labelling: e.g., Roman numerals
- For computation, we need a systematic number representation in which basic arithmetic operations are mechanistic

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>II</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>III</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VII</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>VIII</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>IX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>XX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>XXX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>XL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>LX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>LXX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>LXXX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>XC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>MCMXCVIII</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Positional number systems

- positional notation

\[(\cdots a_3 a_2 a_1 a_0. a_{-1} a_{-2} \cdots)_b \quad 0 \leq a_k < b\]

radix point

radix (or base)

- conventional number system

\[b = 10\]

\[a_k \in \{0, 1, 2, \ldots, 9\}\]
Positional number systems

- base 2: 10010111₂
- base 8: 173₅₈
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: 23° 44′ 12″
Positional number systems

- base 2: $10010111_2$ ← binary
- base 8: $1735_8$ ← octal (octonal)
- base 10: 0, 234, 1983 ← decimal
- base 16: $3F7A$ ← hexadecimal (sexadecimal)
- base 60: $23^\circ 44' 12''$ ← sexagesimal
Positional number systems

- base 2: $10010111_2$
- base 8: $1735_8$
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: $23^\circ 44' 12''$
Positional number systems

- base 2: 10010111₂
- base 8: 1735₈
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: 23° 44′ 12″

= $2^7 + 2^4 + 2^2 + 2^1 + 2^0$
= $128 + 32 + 4 + 2 + 1$
= 167
Positional number systems

- base 2: $10010111_2$
- base 8: $1735_8$
- base 10: $0, 234, 1983$
- base 16: $3F7A$
- base 60: $23^o 44' 12''$

$$= 1 \cdot 8^3 + 7 \cdot 8^2 + 3 \cdot 8^1 + 5 \cdot 8^0$$
$$= 512 + 7 \cdot 64 + 3 \cdot 8 + 5$$
$$= 989$$
Positional number systems

- base 2: \(10010111_2\)
- base 8: \(1735_8\)
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: \(23^\circ 44' 12''\)

conventional hexadecimal digits

\[a_k \in \{0, \ldots, 9, A, B, C, D, E, F\}\]

\[
\begin{align*}
\text{base 16: } & 3F7A &= 3 \cdot 16^3 + 15 \cdot 16^2 + 7 \cdot 16^1 + 10 \cdot 16^0 \\
& &= 3 \cdot 4096 + 15 \cdot 256 + 7 \cdot 16 + 10 \\
& &= 16250
\end{align*}
\]
Binary systems

- Western system of musical notation
Binary systems

- English system of weights and measures

2 gills = 1 chopin
2 chopins = 1 pint
2 pints = 1 quart
2 quarts = 1 pottle
2 pottles = 1 gallon
2 gallons = 1 peck
2 pecks = 1 demibushel
2 demibushels = 1 firkin
2 firkins = 1 kilderkin
2 kilderkins = 1 barrel
2 barrels = 1 hogshead
2 hogsheads = 1 pipe
2 pipes = 1 tun
Binary systems

- A modern digital computer stores information in a memory cell called a “bit”.

- The four-transistor arrangement has two stable internal states, used to denote 0 and 1.

memory transistors $M_1,\ldots,M_4$
access transistors $M_5,M_6$
Fixed-width binary

- An unsigned, 8-bit binary number can represent the natural numbers 0 — 255
- There are $2^8$ unique patterns of 0 and 1

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 1
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
```
Fixed-width binary

-128 ≤ x ≤ 127

two's complement

0 ≤ x ≤ 255

conventional binary

sign information resides in the high bit
### C++ integer types

- Only relative sizes guaranteed; on the Intel architecture . . .

- **char**
  - 8 bits, 1 byte (on all platforms)

- **short int**
  - 2 bytes, 1 word

- **int, long int**
  - 4 bytes, 1 double word
C++ integer types

› Types of fixed width available in C

#include <stdint.h>

00000000 int8_t, uint8_t

8 bits, 1 byte

0000000000000000 int16_t, uint16

16 bits, 2 bytes

000000000000000000 int32_t, uint_32

32 bits, 4 bytes

32 bits, 4 bytes
Potential dangers

- Fixed-width binary numbers can represent only a limited range of integers

- The result of an operation (such as addition or multiplication) performed on pairs of representable integers may not be representable itself!

- This condition is called “overflow”

- Wait, does this really matter? Yes, there are many famous real-life examples (YouTube: Ariane 5 explosion)
Arithmetic operators

```c
unsigned char a = 3;
a -= 5; // or: a = a - 5;
assert(a == 254);

char b = 64;
b = ++b*2;
assert(b == -126);

int x = 2*(21+4);
int y = 5 + x++/17;
assert(x == 51);
assert(y == 5 + 2);

for (int i = 0; i < 1000; ++i)
    if (i%15 == 0) do_something();
```

- **integer 0–255**
- **integer -128–127**
- **prefix increment**
- **truncation rather than rounding**
- **modulus**


**Bitwise operators**

```cpp
enum directions { N = 1, E = 2, S = 4, W = 8 };  
const uint8_t opt1 = 020;  // 2*8 == 16  
const uint8_t opt2 = 0x20;  // 2*16 == 32

unsigned char flags = N | W;
assert( (flags & N) and (flags & W) );

flags |= S | E;
assert( flags == N | S | E | W );

flags &= S;
assert( flags == N | E | W );

flags ^= N | E | opt1;
assert( flags == W | opt1 );
flags ^= opt1 | opt2;
assert( !(flags & opt1) and (flags & opt2) );
```
Spatial data structures

- Associate properties with each site (or link) of a lattice
- Encode some sense of which sites are neighbours
- For **hypercubic** lattices, the setup is trivial with C arrays:

```c
// square lattice as 2D array
int lattice[100][100];
lattice[60][99] = 0;

// square lattice as 1D array
int lattice[100*100];
inline int index(int i, int j)
{ return i+j*L; }
lattice[index(60,99)] = 0;
```
Square lattice

[0][0]  [L-1][0]
[0][0]  [L-1][L-1]

matrix-style storage
Square lattice

row-major storage
Square lattice

A and B sublattices

\[(L \% 2 == 0)\]
```cpp
#include <vector>
using std::vector;

template <typename T>
class square_lattice
{
private:
  const int L; vector<T> data;
public:
  square_lattice(int L_) : L(L_), data(L*L) {}
  T& operator()(int i, int j) { return data[i+j*L]; }
  int length(void) { return L; }
};

struct cell { int speciesA; speciesB; }; square_lattice<cell> lattice(20);
for (int i = 0; i < lattice.length(); ++i)
  lattice(4,i).speciesA = 3;
```
• Topology preserved when distorted to a brick-wall lattice

inequivalent A and B sites

simple periodic boundary conditions
Triangular lattice

- Topology preserved when sheared to orthogonal axes

triptite: A, B, and C sublattices

\( L_x = L_y + 1 \)
Kagomé lattice

- Further deplete the triangular lattice

wasted storage