Computergestütztes wissenschaftliches Rechnen
SoSe 2004

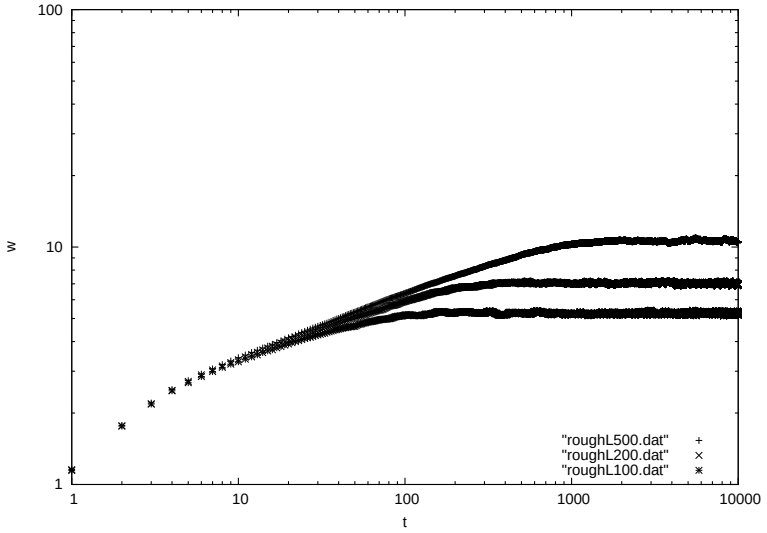
Alexander K. Hartmann, Universität Göttingen

9. Juli 2004

9.2 Finite-size Scaling

Physikalische Systeme: groß ($O(10^{23})$ Teilchen).
Simulationen: endliche (kleine) Systemgrößen.
Ausweg: Studiere *mehrere verschiedene* Systemgrößen.

Bsp.: Entwicklung der Rauhigkeit w bei ballistischer Diffusion als Funktion der Zeit:



Rauhigkeit konvergiert für $t \to \infty$ gegen Wert abhängig von Größe L. Grund: Korrelationen durch Wechselwirkungen zwischen Nachbarn. Annahme (durch Theorie gerechtfertigt):

$$w(t \to \infty, L) \sim L^{\alpha} \tag{1}$$

mit $\alpha =$ "Rauhigkeitsexponent". Diese Relation gilt nur für große (aber machbare) Systeme. Die weiter unten abgeleiteten Relationen gelten damit auch nur für nicht zu kleine Systeme. Um auch ganz kleine Systeme zu beschreiben, müsste man sog. *Korrekturen zum Skalenverhalten* mitbeschreiben.
Die Korrelation muß sich durch das System ausbreiten. Das ist die charakteristische Zeit t_X, wo w konvergiert. Das dauert um so länger, je größer das System ist, also wieder Annahme

$$t_x \sim L^z \tag{2}$$

z = "dynamischer Exponent". t_X beschreibt die typische Zeitskala des Systems. D.h. das Verhalten des Systems hängt nur von t/t_X ab.
Gleichungen (1) und (2) lassen sich durch eine Formel ausdrücken:

$$w(t, L) = L^{\alpha} F(t/L^{z}) \ , \tag{3}$$

wobei für die Funktion $F(\tilde{t})$ gelten muß: $\lim_{\tilde{t}\to\infty} F(\tilde{t}) = \text{const.}$
Von früher wissen wir, dass für kleine Zeiten gilt:

$$w(t) \sim t^{\beta} \tag{4}$$

Also $F(\tilde{t}) \sim \tilde{t}^{\beta}$ für kleine Zeiten $\tilde{t} \ll 1$, damit

$$w(t, L) \sim L^{\alpha}(t/L^{z})^{\beta} = t^{\beta} L^{\alpha - z\beta}$$

Für kleine Zeiten kann das Verhalten nicht von der Systemgröße abhängen (da sich die Korrelation noch nicht durch das System "fortgepflanzt" haben kann) $\rightarrow$ $\alpha - z\beta = 0$ oder

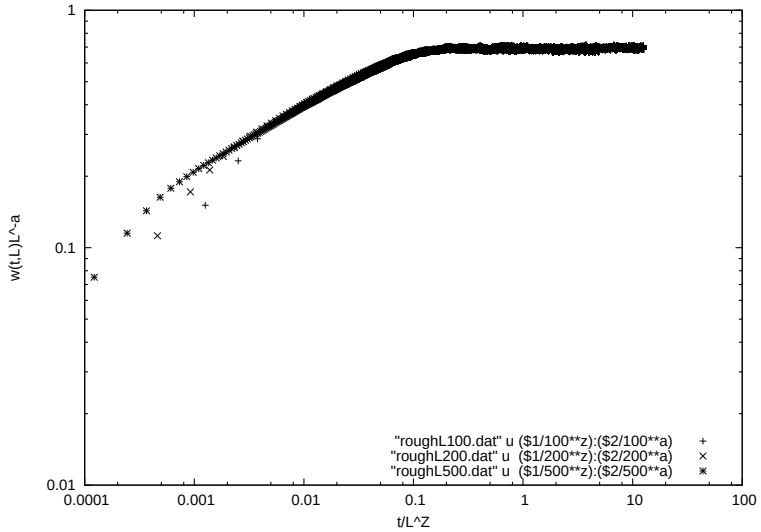
$$z = \alpha/\beta$$

Das ist eine sogenannte *Skalenrelation*, weil es eine Verbindung zwischen den verschiedenen Skalenexponenten α, β, z herstellt.

Gl. (3) kann man nutzen um α und z zu bestimmen: Wenn man $L^{-\alpha} w(t, L)$ für verschiedene Größen L als Funktion von t/L^{z} plottet, dann müssen alle Datenpunkte auf der gleichen Kurve (gegeben durch die Funktion $F(\tilde{t})$) liegen. Das kann man mit `gnuplot` machen:

```
gnuplot> z=1.45
gnuplot> a=0.44
gnuplot> plot "roughL100.dat" u ($1/100**z):($2/100**a),
         "roughL200.dat" u  ($1/200**z):($2/200**a),
         "roughL500.dat" u  ($1/500**z):($2/500**a)
```

(Hinweis: noch einfacher ist es, wenn man EIN File macht mit einer weiteren Spalte, die die Systemgröße enthält)



Im Allgemeinen sind die Skalenexponenten nicht bekannt, dann muss man – ausgehend von sinnvollen Startwerten – iterativ die Werte ändern bis die Kurven gut übereinanderliegen ("Datenkollaps").

10 Perkolation

Modell für poröses Gestein: Quader in kleine Würfel zerlegt, wo jeder Würfel mit einer Wahrscheinlichkeit $1 - p$ von Stein "belegt" ist. Der Anteil p ist "frei".

Frage: Wie groß muss p sein, damit Wasser von einer Seite durch den ganzen Stein zur gegenüberliegenden Seite hindurchfließen ("perkolieren") kann. → Phasenübergang bei kritischer Konzentration $p = p_c$ oberhalb der das Wasser hindurchfließt. Perkolation ist ein wichtiges (Spielzeug) Modell für Phasenübergänge generell (mehr in Statistische Physik II). Vielen Phasenübergängen liegen (versteckte) Perkolationsübergänge zugrunde.

Literatur: [1].

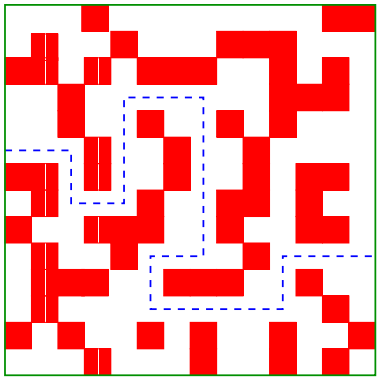


Abbildung 1: Perkolation in zwei Dimensionen: Es gibt einen Weg von freien Gitterplätzen, so dass man durch das ganzen System laufen kann (gestrichelte Linie). Das System perkoliert.

Der Wert von p_c hängt von der Dimension des Systems und der Kristallstruktur ab. In einer Dimension ist trivialerweise $p_c = 1$. Für größere Dimensionen kann man meistens keine analytischen Aussagen machen → Computersimulationen (Ergebnisse quadratisches Gitter: $p_c \approx 0.592746$, kubisch: $p_c \approx 0.3116$, ... [1]). Die dabei verwendeten Algorithmen (Clusteralgorithmen) kommen in VIELEN Bereichen der computerorientierten Physik vor.

10.1 Stacks

Zur folgenden Implementierung benötigen wir einen speziellen Datentyp: *Stacks*.

Stack = Stapel, man kann ein Element oben drauflegen und wieder wegnehmen. Nur das oberste Element ist zugänglich.
→ LIFO (last in first out).
Anmerkung: Warteschlange = FIFO Prinzip.

Realisierung: im Wesentlichen als Array.
Hier: objektorientierter Ansatz: Stacks = Objekte + Operationen für Stacks. Implementierung in Unterroutinen, Zugriff nur über diese (*Datenkapselung*).

C++ is eine Erweiterung von C, spezielle objektorientierte Unterstützung. Man kann aber in JEDER Programmiersparche objektorientiert programmieren, aber

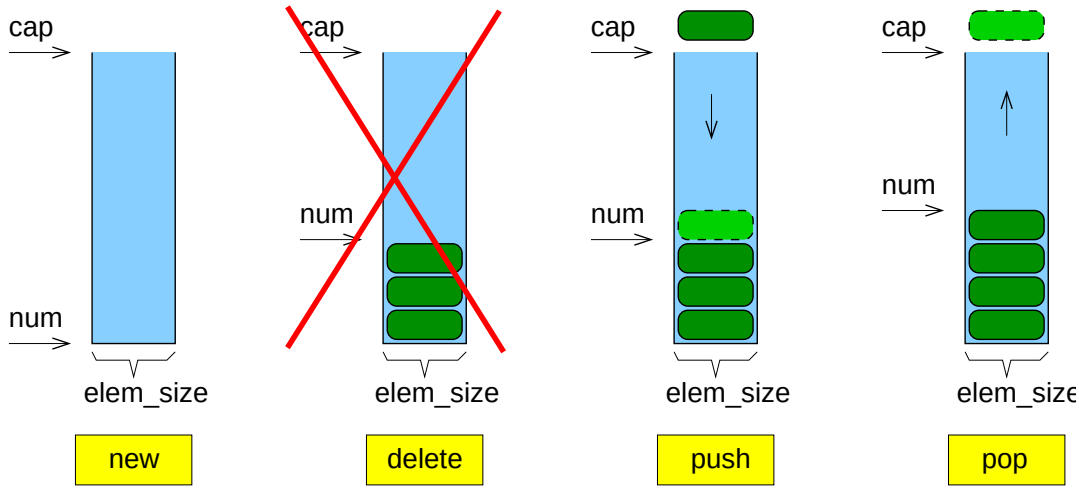
evtl. unpraktischer. Hier verwenden wir weiterhin C

Zugrundelegender Datentyp für Stacks von Elementen beliebigen aber gleichen Typs:

```
typedef struct
{
  int           num; /* current num. of elements in the stack */
  int           cap; /* size of a stack  (No. of elements)    */
  size_t  elem_size; /* size of an element                    */
  void        *data; /* pointer to stack-memory               */
} lstack_t;
```

Grundlegende Operationen:

- `new`: Anlegen eines Stacks für Elemente einer bestimten Größe, mit einer maximalen Zahl von Elementen.
- `delete`: Löschen eines Stacks.
- `push`: Ein Element wird oben auf den Stack gelegt.
- `pop`: Ein Element wir vom Stack runter genommen.



Implementierung:

```
/** Creates a stack of size 'cap', where each data-element **/
/** has the size 'elem_size' (in bytes = char).            **/
/** RETURN: pointer to the stack                           **/
lstack_t *lstack_new(int cap, size_t elem_size)
{
  lstack_t *stack;

  stack = (lstack_t *) malloc(sizeof(lstack_t));
  stack->num = 0;
  stack->cap = cap;
  stack->elem_size = elem_size;
  stack->data = malloc(cap*elem_size);
  return(stack);
}
```

```
/** Deletes a 'stack'       **/
void lstack_delete(lstack_t *stack)
{
  free(stack->data);
  free(stack);
}
```

Hinweis: Für `push` und `pop` wird immer ein Zeiger auf das Element übergeben.

```
/** Pushes an 'element' on top of the 'stack'.     **/
/** RETURN: =0 -> everything ok, or = LSTACK_FULL  **/
int lstack_push(lstack_t *stack, void *element)
{
  void *ptr;

  if(stack->num >= stack->cap) /* Tests if stack is full */
    return(LSTACK_FULL);
  ptr = stack->data + (stack->num++) * stack->elem_size;
  memcpy(ptr, element, stack->elem_size);
  return(0);
}

/** Pops the top 'element' from the 'stack' **/
/** RETURNS =0 -> ok, or = LSTACK_FULL      **/
int lstack_pop(lstack_t *stack, void *element)
{
  void *ptr;

  if(stack->num == 0)         /* Tests if stack is empty */
    return(LSTACK_EMPTY);
  /** return the element to be poped: **/
  ptr = stack->data + (--(stack->num)) * stack->elem_size;
  memcpy(element, ptr, stack->elem_size);
  return(0);
}
```

Literatur

[1] D. Stauffer und A, Aharony W.H. Press, S.A. Teukolsky, W.T. Vetterling, *Perkolationstheorie*, (Wiley-VCH, Weinheim 1995)