Query by Humming

T. H. Merrett*

McGill University

April 9, 2008

[1]

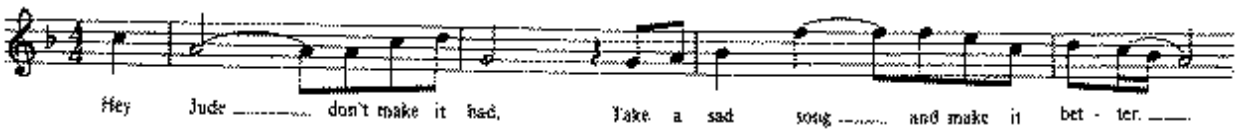
This is taken from [SZ04, Chap. 6]. It is the second of three applications they develop based on their review of timeseries techniques in the first four chapters.

1. "Query by humming" is a challenging unsolved problem in timeseries matching. Because matches cannot be exact, dynamic time warping (DTW) is needed but this is slow, even when we use dynamic programming (see timeseries.pdf, Note 7). Shasha and Zhu find an approximation which excludes false negatives (failed alarm) and speeds up the matching process substantially.

The problem is to match, from a database of tunes, the tune that somebody hums, maybe quite inaccurately. The melodic contour approach (a melody is represented as a sequence of `u, -` and `d` as the pitch goes up, stays the same or goes down—or maybe, more refined, as `U, u, -, d` and `D`) discriminates only very poorly among the many possible entries in the database. It also requires the identification of individual notes from the hum-query, which is a difficult problem. (The database entries can be derived from the written score, which provides the individual notes, so there is no problem there.)

So Shasha and Zhu represent the hum-query as a timeseries of pitches, sampled regularly, say every 10 ms, without concern about individual notes. They also store the scores in the database as timeseries of pitches.

2. Let's start with a database entry, "Hey Jude" (John Lennon, Paul McCartney, 1965) From the score



we can extract the following sequence of (pitch,duration) pairs

```
(60,4),(57,10),(57,2),(60,2),(62,2),(55,8),(-,4),(55,2),(57,2),
(58,4),(65,6), (65,2),(64,2),(60,2),(62,2),(60,1),(58,1),(57,8)
```


The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

where the pitch is given in units related to (but not the same as) the note frequency—on the treble cleff, G is 55, A is 57, B♭ is 58, C is 60, D is 62, E is 64 and F is 65 (the frequency for A, for example, is 440 Hertz, and A an octave higher is 880Hz, so the intermediate notes have frequencies that are multiples of powers of $\sqrt[12]{2}$ times each other). I've used - to mean a rest.

The durations are in units of semiquavers (16th notes), which are the quickest notes in the score. The four bars need $4\times16 = 64$ units, and this is the sum of all the durations.

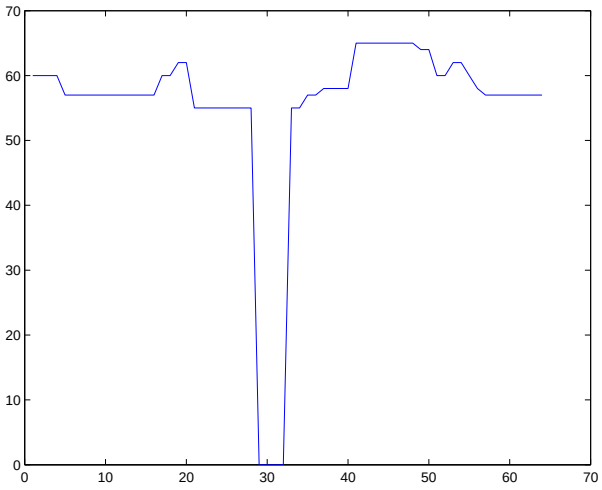
To convert this sequence of tuples (duples) to the timeseries that can be compared with the hum-query, we assign one time unit to each semiquaver, and so

```
(60,4),(57,10),..
```

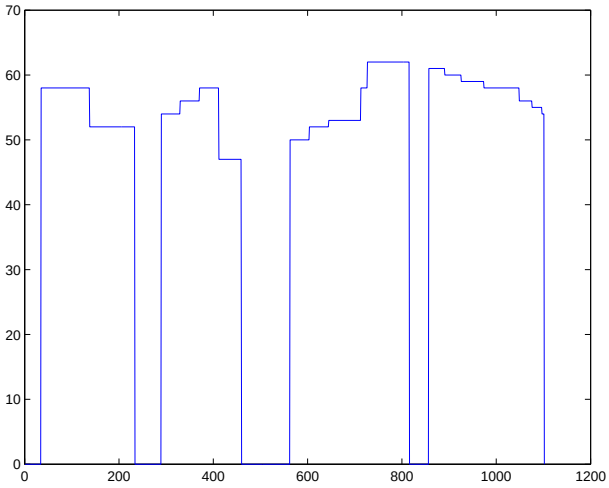
becomes

```
60,60,60,60,57,57,57,57,57,57,57,57,57,57,..
```

Here is the plot (the rest is assigned a pitch of zero)



3. The hum-query is taken from a microphone and converted by a pitch-tracking algorithm [T Tolonen and M Karjalainen IEEE Trans Speech & Audio Processing, 2000] directly to the same sort of timeseries of pitches. The units in the plot are 10ms, so the eleven minutes need 1102 units.



I took the following data from Shasha and Zhu's version of this plot on p.131 using a ruler and then scaling to 11.02 seconds.

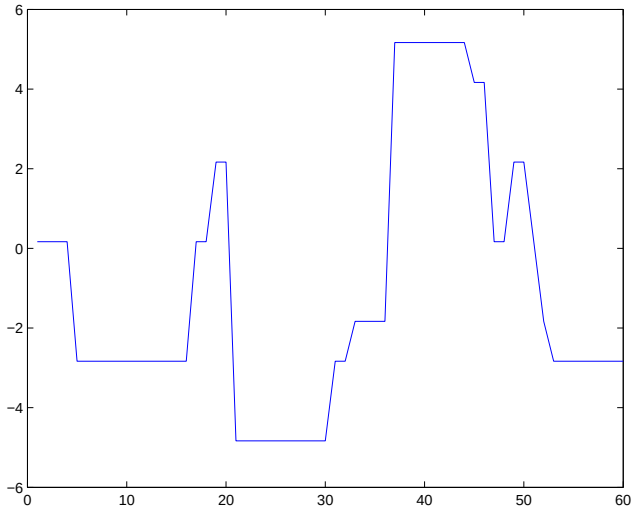
```
(-,.34),(58,1.03),(52,.96),(-,.56),(54,.4),(56,.41),(58,.41),(47,.48),
(-,1.03),(50,.41),(52,.41),(53,.68),(58,.14),(62,.89),(-,.41),(61,.34),
(60,.35),(59,.48),(58,.75),(56,.27),(55,.21),(54,.05),(-,.01)
```

(So it is not very accurate, but then neither was the hum they recorded.)

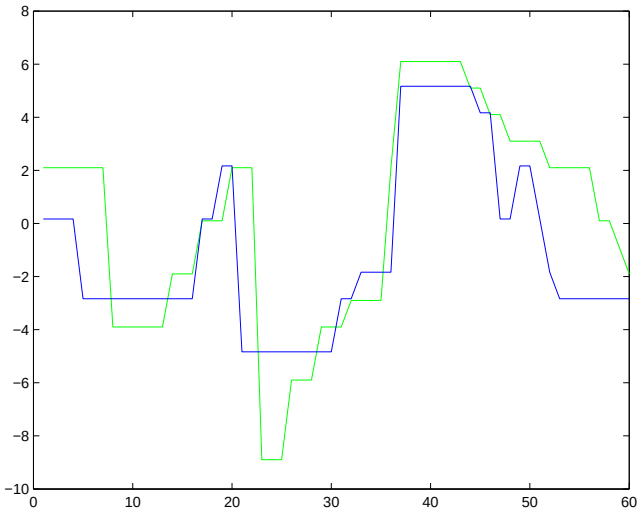
In turn, I converted the sequence of (pitch,duration) tuples to the timeseries that is in the plot above.

4 We will need to do three things with these two timeseries before trying to match them. First we remove all the rests. This eliminates the hesitations we see in the hum-query and rests from the score, which hummers might ignore. It brings the database entry down to 60 semiquavers and the hum-query down to 867 10-ms entries.

Second we *shift* both series so that their average is zero. This allows the hum-query to be off true pitch, since few people have perfect pitch. We do not, however, complete the normalization process by scaling the timeseries: most people have reasonable relative pitch and can hum intervals fairly well. Here is what the database entry looks like after these two operations.



Third, we time-scale the timeseries so they both have the same length. This is normally done by finding the least common multiple of the number of units, 60 and 867, of the two series, but to keep things small for this discussion I just reduced the 867 points of the hum-query to 60 by sampling the first, the last and every `round(867/60)` in between. Here are the two series ready to be compared. The hum-query is in green.



5. Now we can match the two series. In general we need dynamic time warp (DTW) because the hum tempo is likely neither to be the same as the specified tempo (although timescaling largely takes care of this) nor to be as regular as the timeseries representation of the written music.

Since dynamic time warp is slow, Shasha and Zhu find an approximation to it which is *lower-bounding*, i.e., the distance between two series given by the approximation is never greater than the DTW distance. This means that, if some threshold distance is specified, that the DTW must be less than for two series to be considered similar, the approximation will never miss two series at this distance or less, even though it may cause some false alarms. The alarm will not fail: we have no false negatives even though there may be false positives.

The approximation must be *dimension-reducing* in order to do the comparisons faster than full DTW. Shasha and Zhu pick the piecewise aggregation approximation (PAA) [SZ04, p.46] because it is a linear transformation all of whose coefficients are positive: this plays an important role in the proof that the approximation is lower-bounding.

PAA simply divides the points of the timeseries equally into groups and replaces the series by the average for each group. This is in the spirit of the Haar wavelet but simpler. (PAA has a very pretentious name for such a simple reduction.) Here it is applied to a slight modification of the eight-point timeseries we used to illustrate wavelets (see elasticBurst.pdf): the last two terms have been replaced by 1 and 2, respectively, so as to have no zeros, which are rests in our melody timeseries.

$$\frac{1}{2}\begin{pmatrix} 1 & 1 & & & & & & \\ & & 1 & 1 & & & & \\ & & & & 1 & 1 & & \\ & & & & & & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 3 \\ 5 \\ 11 \\ 12 \\ 13 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 8 \\ 12.5 \\ 1.5 \end{pmatrix}$$

The eight-dimensional timeseries has been reduced to four dimensions by averaging in pairs.

Similarly we could reduce our 60-dimensional vector to five by averaging in dozens.

6.To get the approximation to DTW distance between two series we need, Shasha and Zhu introduce *envelopes*. A k-envelope is simply a running min and a running max over the range $[-k, k]$ of the timeseries. The second and third vectors in the following 8-by-3 matrix are the min and max,

respectively, of a 1-envelope for the series. (Note that we take averages over shorter intervals at the two ends.)

$$\frac{1}{2}\begin{pmatrix} 1 & 1 & & & & & & \\ & & 1 & 1 & & & & \\ & & & & 1 & 1 & & \\ & & & & & & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 5 \\ 3 & 1 & 5 \\ 5 & 3 & 11 \\ 11 & 5 & 12 \\ 12 & 11 & 13 \\ 13 & 1 & 13 \\ 1 & 1 & 13 \\ 2 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 5 \\ 8 & 4 & 11.5 \\ 12.5 & 6 & 13 \\ 1.5 & 1 & 7.5 \end{pmatrix}$$

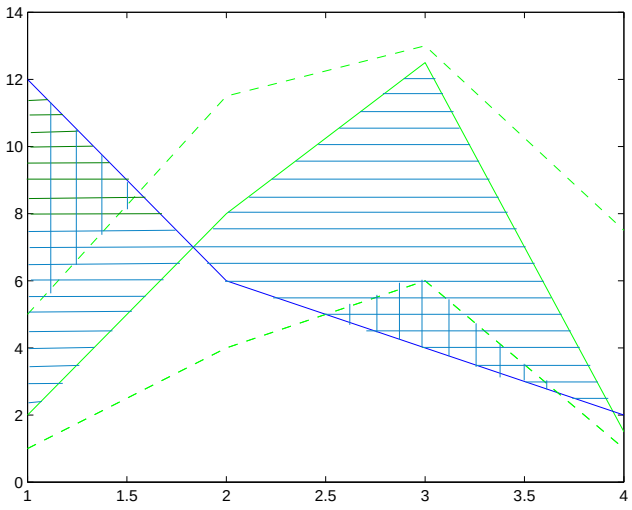
In this expression, I've repeated the PAA reducing transformation on the original timeseries as well as on the lower and upper bounding timeseries of the envelope.

Notice that the transformed envelope is an envelope of the transformed series.

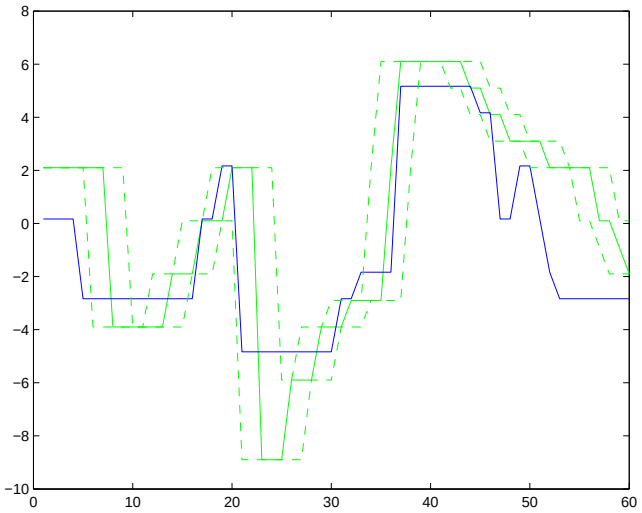
This is a property that Shasha and Zhu prove for the PAA transform (and they derive a more general envelope-transforming procedure for any linear transform).

The importance of envelopes is that the distance from a timeseries to the envelope of another timeseries lower-bounds the DTW distance between the two timeseries.

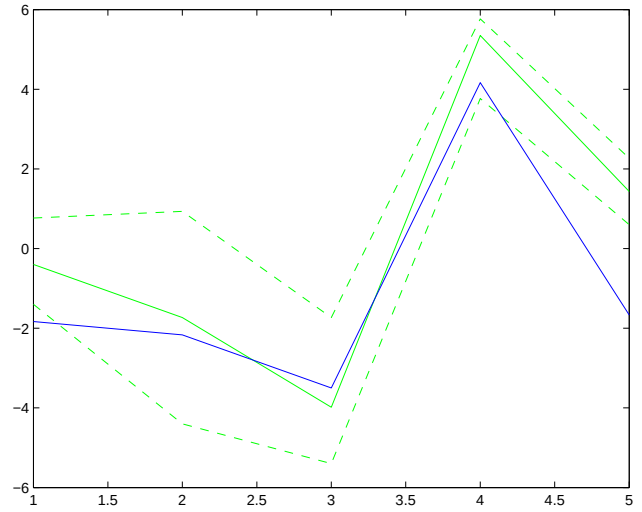
For the 4-term reduced timeseries and its envelope that we just calculated above (green), here is a picture of the distances from the series `12, 6, 4, 2` (blue). The distances are shown as areas: the horizontally-shaded area is the distance to the original series, and the vertically-shaded area is the distance to the envelope (and implicitly defines what is meant by this distance). We can see that the horizontal area contains the vertical area, and hence that the distance to the envelope lower-bounds the distance to the timeseries.



With this, we can return to *Hey Jude* and show the hum-query and its envelope (in green) and the database timeseries (in blue).



Here they are again, both PAA-reduced to five-point series from the sixty points we last had. The distance is evidently close, and the distance from the envelope to the database entry is even closer, so we expect that this entry (among the thousands of others in the database) will be identified and retrieved.



The five-point timeseries form points in a five-dimensional space, and we can index all the timeseries in the database in this space after suitable PAA reduction, and use the index to retrieve only points within a certain distance of the hum-query.

If the distances are first calculated from the hum-query envelope, this will be fast and will not miss any candidates once we have identified the potential candidates and done full DTW between each one and the hum-query.

7. Aldat We write the Aldat implementation in five parts. First we eliminate rests, then shift and timescale. Second we convert from duration representation to timeseries representation. (Note that it is faster to do the three operations of the first part in duration representation. I believe we will be able to do the whole job in duration representation but I haven't tried it yet.) Third we calculate the envelope of the hum-query. Fourth we find the PAA reduction. And finally we

calculate distances from envelope to selected database entries.

Instead of working with the *Hey Jude* series we take a shorter example. The database entry, D, consists of five points (it's a database of only one entry: extension to many is straightforward) and the hum-query, Q, has ten.

$D(s$	v	$d)$
1	12	1
2	6	1
3	0	1
4	4	1
5	2	1

$Q(s$	v	$d)$
1	1	1
2	3	1
3	5	1
4	0	1
5	11	1
6	12	1
7	13	1
8	0	1
9	1	1
10	2	1

The attributes are s (sequence number), v (value) and d (duration). The durations are all 1, to keep the next steps simple. Since this is simplistic, we will occasionally show examples with a variety of durations.

The code to remove rests ($v = 0$) and to shift to series averages of zero is

let *vav* **be** (**red** + **of** $v \times d$)/(**red** + **of** d);
let *v1* **be** $v-$ *vav*;
let v **be** *v1*;
let *s1* **be fun** + **of** 1 **order** s;
let s **be** *s1*;
Dshift <− $[s, v, d]$ **in** [*s1, v1, d*] **where** $v > 0$ **in** D;
Qshift <− $[s, v, d]$ **in** [*s1, v1, d*] **where** $v > 0$ **in** Q;

Dshift$(s$	v	$d)$
1	6	1
2	0	1
3	−2	1
4	−4	1

Qshift$(s$	v	$d)$
1	−5	1
2	−3	1
3	−1	1
4	5	1
5	6	1
6	7	1
7	−5	1
8	−4	1

We do timescaling the proper way: we find the least common multiple of the total durations of the two series and stretch both to that length. This is easy with durations.

m <− [**red** + of d] **in** *Dshift*;
n <− [**red** + of d] **in** *Qshift*;
ℓ <−$m \times n/\gcd[m, n]$;
let *d1* **be** $d \times \ell/$(**red** + **of** d);
let d **be** *d1*;
Dscale <− $[s, v, d]$ **in** [*s, v, d1*] **in** *Dshift*;
Qscale <− $[s, v, d]$ **in** [*s, v, d1*] **in** *Qshift*;

In our example this just doubles the durations in *Dshift*.

Dscale(*s*	*v*	*d*)
1	6	2
2	0	2
3	−2	2
4	−4	2

Qscale(*s*	*v*	*d*)
1	−5	1
2	−3	1
3	−1	1
4	5	1
5	6	1
6	7	1
7	−5	1
8	−4	1

A In another example, such as (v, d) given, s implied)

Dshift = [(0,3),(1,2),(−1,2),(0,3)]

Qshift = [(0,2),(1,2),(2,3),(0,2),(−2,2),(−1,4)]

we get $m = 10, n = 15$ with a gcd of 5, so $\ell = 30$ and

Dscale = [(0,9),(1,6),(−1,6),(0,9)]

Qscale = [(0,4),(1,4),(2,6),(0,4),(−2,4),(−1,8)]

8. The second operation is to convert from durations to the kind of timeseries we have been plotting above. Let's use an example with a variety of durations, say, *Dshift*, above.

In relational form, we need to go from *Dshift* to *Dts*:

Dshift(*s*	*v*	*d*)
1	0	3
2	1	2
3	−1	2
4	0	3

Dts(*s*	*v*)
1	0
2	0
3	0
4	1
5	1
6	−1
7	−1
8	0
9	0
10	0

We can do it with an update event handler.

comp post:add:*Dshift*() **is**
{ **let** *d1* **be** (**equiv min of** *d* **by** *s*) − 1;
let *d* **be** *d1*;
temp <−[*s*, *v*, *d*] **in** [*s, v, d1*] **where** *d1* > 0 **in** *Dshift*;
update *Dshift* **add** *temp*;
};

If we start this off with

temp <− **pick** *Dshift*;
update *Dshift* **delete** *temp*;
update *Dshift* **add** *temp*;

we get the following iterations on *temp*:

Dshift		
(*s*	*v*	*d*)
1	0	3
2	1	2
3	−1	2
4	0	3

temp		
(*s*	*v*	*d*)
1	0	2
2	1	1
3	−1	1
4	0	2

temp		
(*s*	*v*	*d*)
1	0	1
2	1	0
3	−1	0
4	0	1

and the fourth iteration of *temp* has $d = 0$ or -1 everywhere and the update stops.

To get *Dts*

let *s1* **be fun** + **of** 1 **order** *s*, *d*;

let s **be** *s1*;
Dts <− $[s, v, d]$ **in** [*s1,v,d*] **in** *Dshift*;

Of course, all this operation would be performed on *Dscale* and *Qscale*, but for the example these would require as many as nine iterations, so I worked with *Dshift* instead.

9. The third implementation step is to find the envelope of the hum-query. We revert to the main example of our Aldat implementation discussion, in which *Qscale* and *Qts* are

Qscale(s	v	d)	*Qts*(s	v)
1	−5	1	1	−5
2	−3	1	2	−3
3	−1	1	3	−1
4	5	1	4	5
5	6	1	5	6
6	7	1	6	7
7	−5	1	7	−5
8	−4	1	8	−4

We must adjust **redwin**() (week7p1) a little to calculate the lower, ℓ, and upper, u, pieces of the k-envelope. The adjustment, which I am not completely happy with, is to allow negative widths for w in **redwin**(w) and to interpret these to mean "take a full window width of $-w$ starting at the current tuple, but ignore values until the end of the relation".

let $s1$ **be** (**red max of** s) − s;
let w **be if** $s1 \geq 2 \times k$ **then** $2 \times k + 1$ **else**
if $s1 \leq k$ **then** $s1 + 1$ **else** $-(2 \times k + 1)$;
let $\ell 1$ **be redwin**(w) **min of** v **order** s;
let *u1* **be redwin**(w) **max of** v **order** s;
let ℓ **be fun pred**(k) **of** $\ell 1$ **order** s;
let u **be fun pred**(k) **of** *u1* **order** s;

For $k = 1$

Qts(s	v)	$s1$	w	$\ell 1$	*u1*	ℓ	u
1	−5	7	3	−5	−1	−5	−3
2	−3	6	3	−3	5	−5	−1
3	−1	5	3	−1	6	−3	5
4	5	4	3	5	7	−1	6
5	6	3	3	−5	7	5	7
6	7	2	3	−5	7	−5	7
7	−5	1	2	−5	−4	−5	7
8	−4	0	−3	−5	−3	−5	−4

Qenv <−$[s, v, \ell, u]$ **in** *Qts*;

gives the hum-query and its envelope.

10. The PAA dimension reduction can be done by matrix multiplication and we don't attempt to refine it here although there are better ways for this very special matrix.

PAA(s'	s	c)	*Qenv*(s	v	ℓ	u)
1	1	0.5	1	−5	−5	−3
1	2	0.5	2	−3	−5	−1
2	3	0.5	3	−1	−3	5
2	4	0.5	4	5	−1	6
3	5	0.5	5	6	5	7
3	6	0.5	6	7	−5	7
4	7	0.5	7	−5	−5	7
4	8	0.5	8	−4	−5	−4

And the result of the matrix multiplication is, after renaming attributes,

$Qred(s'$	v	ℓ	$u)$	$Dred(s$	$v)$
1	−4	−5	−4	1	6
2	2	−2	5.5	2	0
3	5.5	0	7	3	−2
4	−4.5	−5	1.5	4	−4

There is a similar result for *Dred*, except that it does not need the envelope. This is also shown above.

11. To compute the distance between *Dred* and the envelope of *Qred*, the last implementation step, we join the two relations after renaming, say v of *Dred* to, say vD, and use

let *d2e* **be red** + **of if** $vD > u$ **then** $vD - u$ **else**
if $vD < \ell$ **then** $\ell - vD$ **else** 0;

The spatial indexing method to retrieve only those database points that are within a certain distance of the hum-query would work like the grid index of elasticBurst.pdf. We can suppose that this is built into the implementation of Aldat.

References

[SZ04] Dennis Shasha and Yunyue Zhu. *High Performance Discovery in Time Series: Techniques and Case Studies*. Springer Verlag, New York, 2004.