INTRODUCTION TO NUMERICAL ANALYSIS

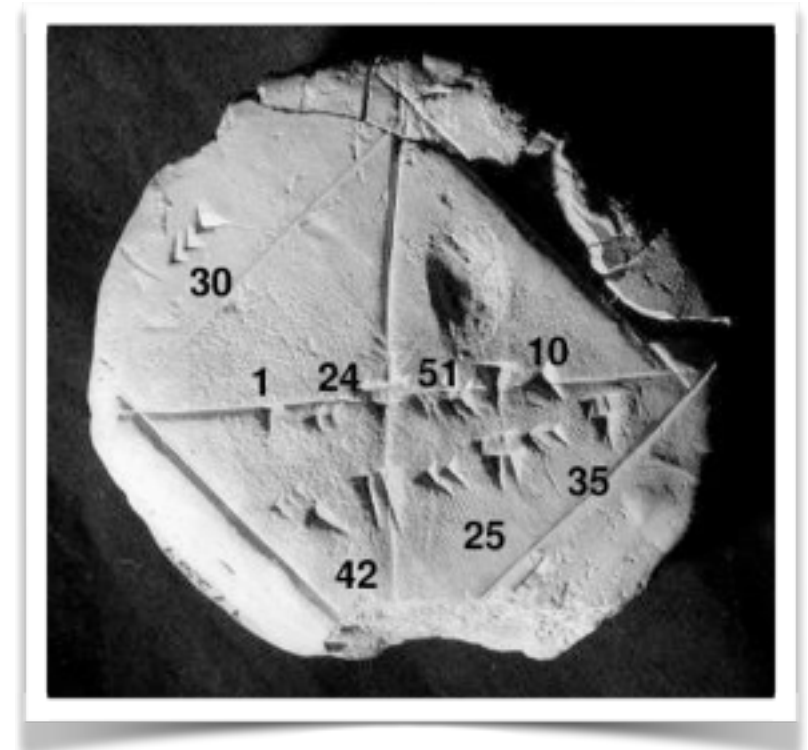
Lecture 2-1:
The Art of Numerical Analysis

Kai-Feng Chen
National Taiwan University

THE ART OF NUMERICAL ANALYSIS

- Wikipedia: numerical analysis is the study of algorithms that use **numerical approximation** (as opposed to general symbolic manipulations) for the problems of mathematical analysis.
- This does not require a very precise calculation. Sometimes the key point is to solve the problems with a (relatively) **quick-and-dirty(?)** way, comparing to a full analytical solution.

Babylonian clay tablet (1800–1600 BC) with annotations. The approximation of the square root of 2 is four sexagesimal figures, which is about six decimal figures. 1 + 24/60 + 51/602 + 10/603 = 1.41421296...



THE ART OF NUMERICAL ANALYSIS (II)

- Thanks to the rapid development of computers, now we don't need to do the calculations on a piece of clay, nor with papers and pens.
- The real speciality of computers is **repetition**. Your computer can do whatever any extremely boring calculations for million times. Sometimes it can be a powerful tool to solve the problems which cannot be calculated with the old fashioned way.
- Caveat: it does not mean one can always do brainless calculations (*sometimes we do!*). Smarter way can provide quick and precision results; stupid way will never produce what you want.

THE FUN

- This course would like to introduce you the merit of numerical analysis (or some not-so-stupid methods to solve the problems).
- As mentioned in the first lecture, the most important goal of this course is to have fun!
- The fun: sometimes, you may find you are able to do some extremely difficult/fancy things with only little efforts!

I'm going to show you a demo why the numerical analysis can be entertaining!

Have you ever think of converting a nice piece of tune to the sheet music?

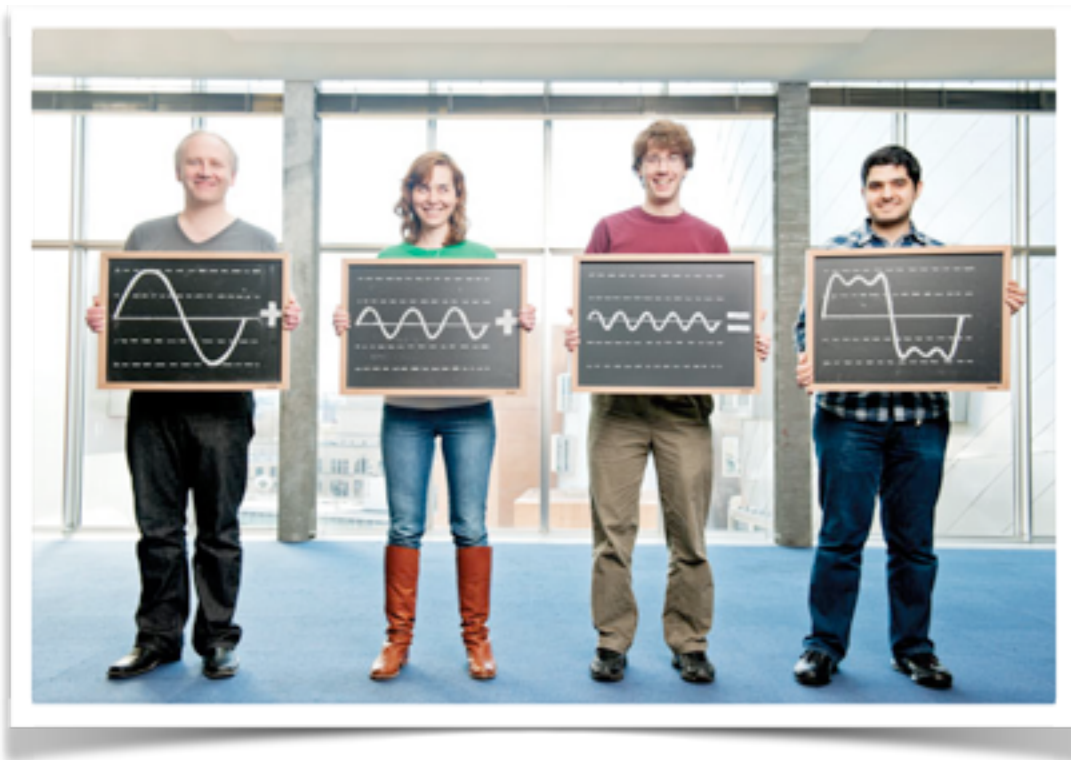
Surely if you are able to find a good musician,
it will work in principle...
(Still a tough job!)

TECHNICALLY SPEAKING...

- —In principle— if you can find some softwares to analyze your CD. Finally you may be able to produce a music sheet, ie.:
  - Rip the wave from your favorite music CD **[EASY]**.
  - Analyze the wave and extract the notes, store it to a midi file **[TOUGH]**.
  - Use some software to read the midi file and produce the sheet music **[FINE]**.

CATCH THE PITCH

- If you do some googling, indeed there are some software can do wave–midi conversion. Unfortunately most of them are paid software...
- But if you think about this more carefully, find the key of a tune is nothing magical but a **Fourier transformation**, right?



If one can do a scan over all of the possible pitches, find the one/few keys with highest amplitudes, you already know how to do a wave-midi conversion!

PYTHON + SCIPY CAN DO THE WORK!

- It is not difficult at all to use **Python + SciPy + NumPy** to do such a job. All you need to do are:
  - Prepare the source wave file.
  - Read the wave file as a NumPy array (SciPy already has such function!).
  - For each time interval, perform the Fourier transformation (with SciPy) for each target frequency. **Record the amplitudes as a function of pitch and time**.
  - Clean up (removing the noise and unwanted harmonics).
  - Phrase the data and write to a MIDI file accordingly (with MidiUtil python package, googled).
  - Done!

A CONCEPTUAL EXAMPLE

- We need a reference wave file; let's get the "pitch standard" from the Wikipedia: http://en.wikipedia.org/wiki/A440_(pitch_standard)

A440 (pitch standard)

From Wikipedia, the free encyclopedia

For other uses, see A440.

A440, which has a frequency of 440 Hz, is the musical note A above middle C and serves as a general tuning standard for musical pitch.

A440 Play (help·info).

Prior to the standardization on 440 Hz, many countries and organizations followed the Austrian government's 1885 recommendation of 435 Hz. The American music industry reached an informal standard of 440 Hz in 1926, and some began using it in instrument manufacturing. In 1936 the American Standards Association recommended that the A above middle C be tuned to 440 Hz.[1] This standard was taken up by the International Organization for Standardization in 1955 (reaffirmed by them in 1975) as **ISO 16**.[2] Although not universally accepted, since then it has served as the audio frequency reference for the calibration of acoustic equipment and the tuning of pianos, violins, and other musical instruments.

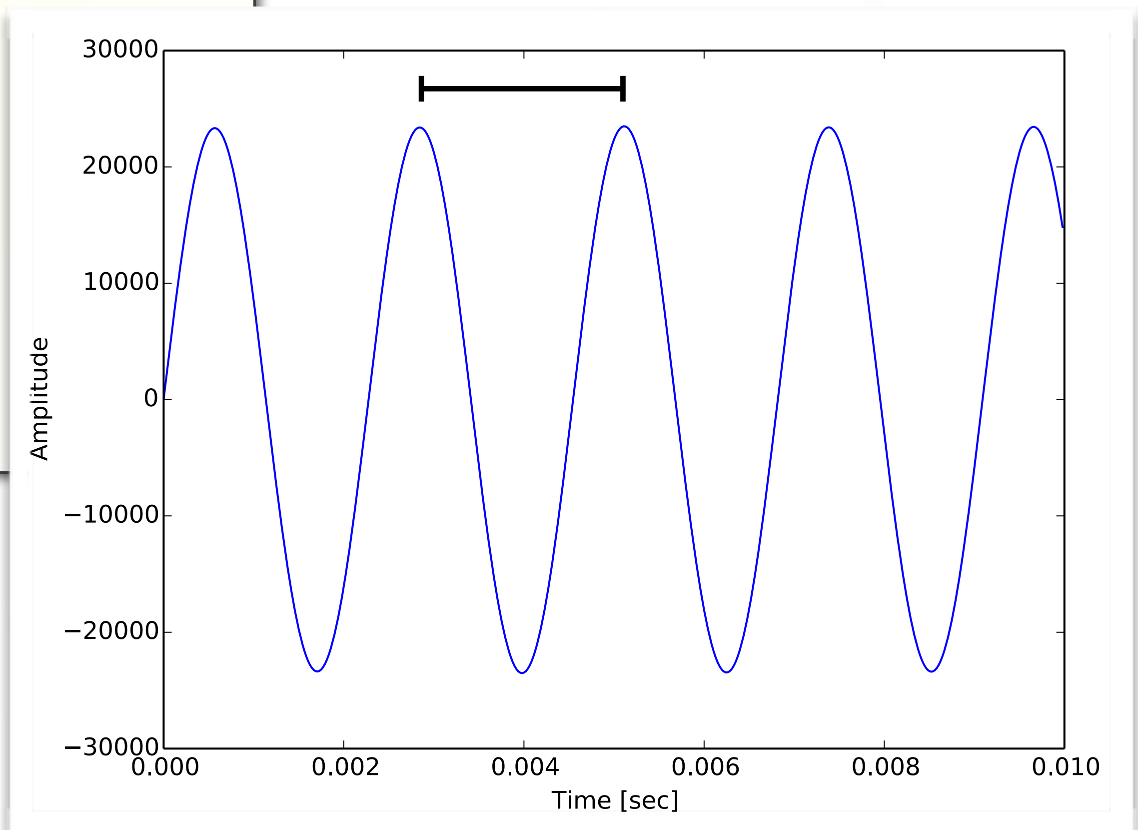
- You'll find a file with sine wave at 440 Hz. Convert it to a standard wave file by **ffmpeg** or any other tool you may find.

A CONCEPTUAL EXAMPLE (II)

- Let's load the wave with SciPy and draw it with Matplotlib:

```
import scipy
import scipy.io.wavfile as wavfile
import matplotlib.pyplot as plt

rate, data = wavfile.read('Sine_wave_440.wav')
t = scipy.linspace(0.,1.,rate,endpoint=False)

plt.figure(figsize=(8,6), dpi=80)
plt.plot(t[:rate//100],data[:rate//100])
plt.xlabel('Time [sec]')
plt.ylabel('Amplitude')
plt.show()
```

period ~2.3 ms,
roughly right!

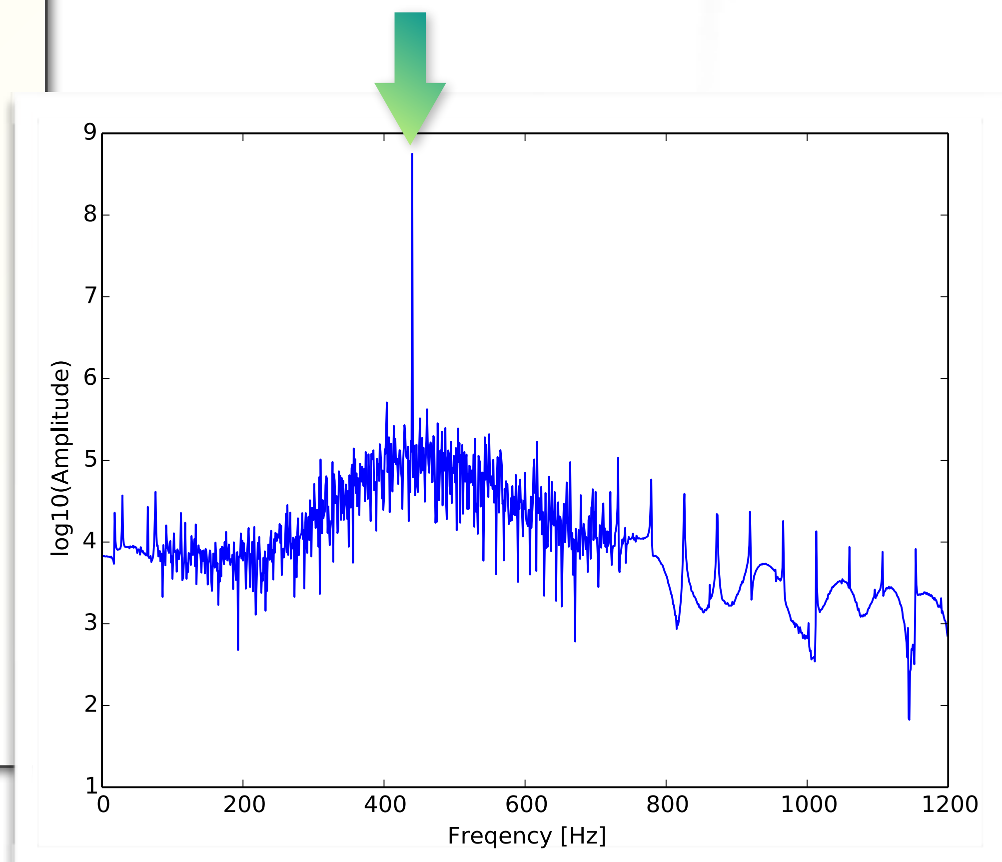


A CONCEPTUAL EXAMPLE (II)

- Call the discrete Fourier transform package:

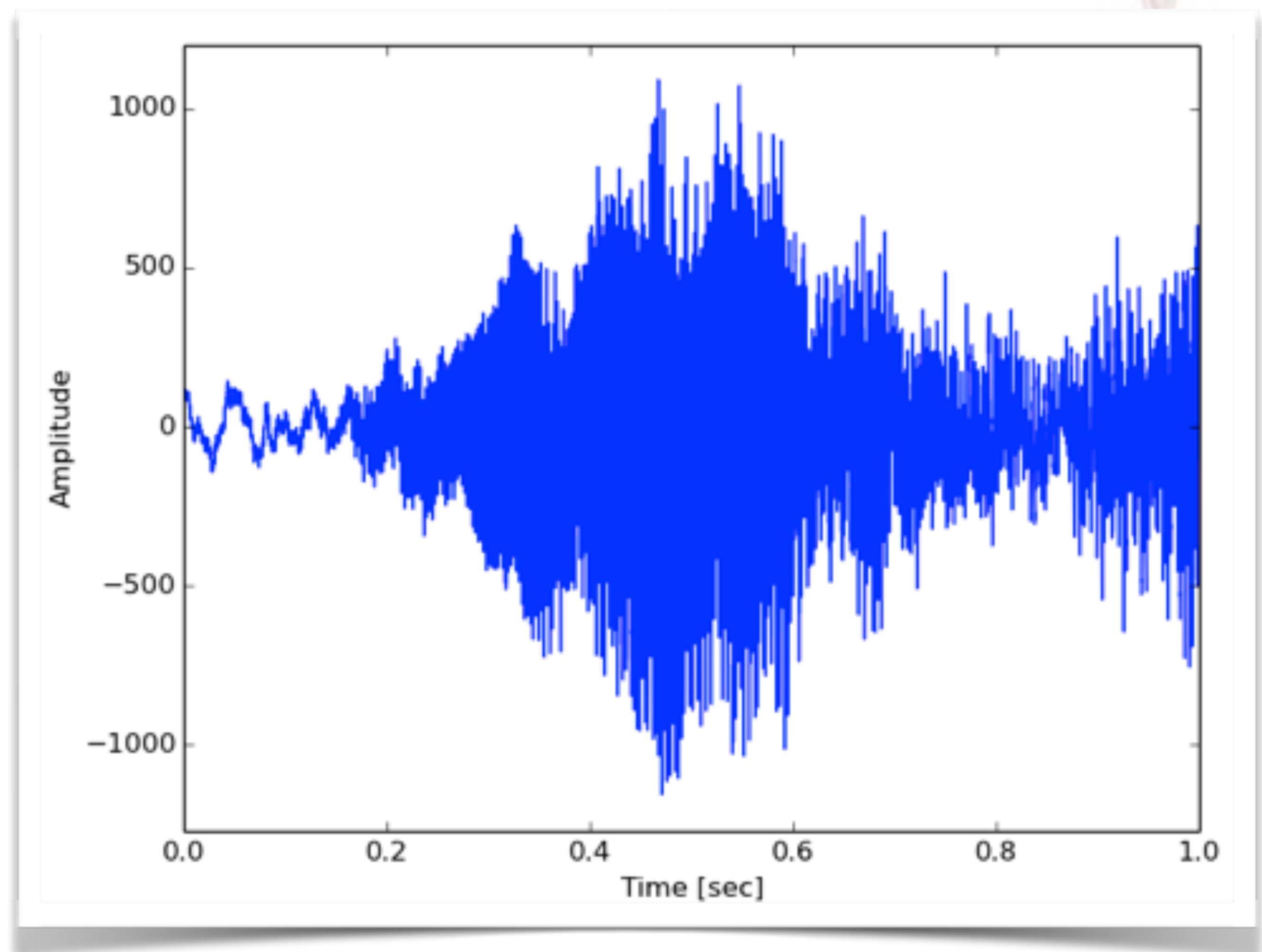
```
import scipy
import scipy.fftpack as fftpack
import scipy.io.wavfile as wavfile
import matplotlib.pyplot as plt

rate, data = wavfile.read('Sine_wave_440.wav')

fft = abs(scipy.fft(data[:rate]))
freqs = fftpack.fftfreq(rate,1./rate)

plt.figure(figsize=(8,6), dpi=80)
plt.plot(freqs[:1200],scipy.log10(fft[:1200]))
plt.xlabel('Freqency [Hz]')
plt.ylabel('log10(Amplitude)')
plt.show()
```

Sharp peak at 440 Hz!



STEP 1: LOADING THE WAVE

- **Real work** — start with loading a wave file into a NumPy array.
- For stereo waves, I simply average the left and right channels to make a mono wave.
- Remove the beginning period with zero amplitude.



Although it sounds a lot of things to carried out, but in reality only ~15 lines of code needed up to this figure.

STEP 2: A LITTLE BIT OF KNOWLEDGE OF MUSIC

- Before performing the Fourier transformation, one need to know what are the frequencies we need to analyze.
- You may find a frequency table like this. But it is easier to use this definition:

$$f_m = 2^{\frac{m-69}{12}} \times 440\text{Hz}$$

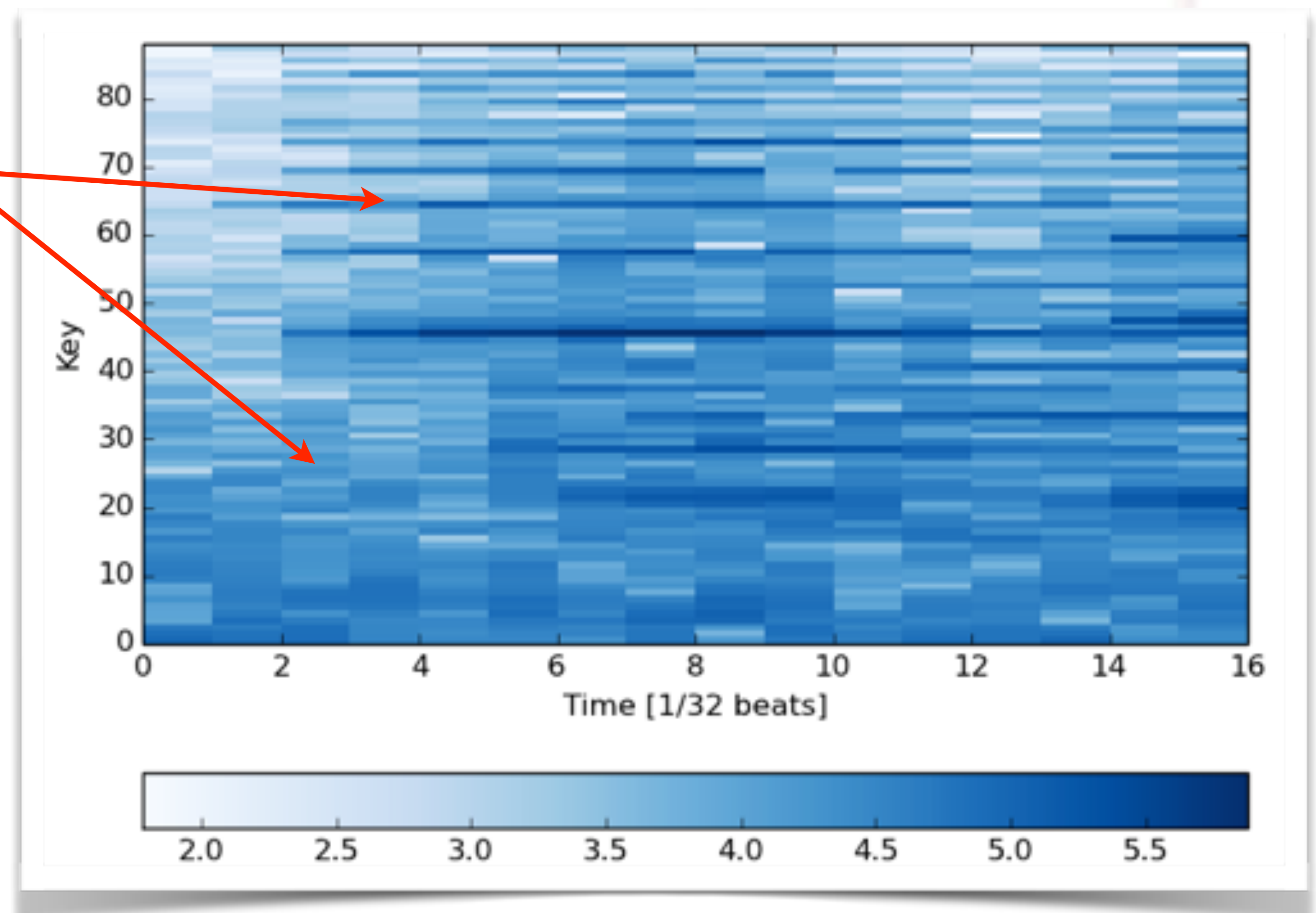
One only need to consider
$21 \le m \le 108$ (88 keys in total)

Frequency		Note name	MIDI number	
	4186.0	C8		108
	3951.1	B7		107
3729.3	3520.0	A7	106	105
3322.4	3136.0	G7	104	103
2960.0	2793.8	F7	102	101
	2637.0	E7		100
2489.0	2349.3	D7	99	98
2217.5	2093.0	C7	97	96
	1975.5	B6		95
1864.7	1760.0	A6	94	93
1661.2	1568.0	G6	92	91
1480.0	1396.9	F6	90	89
	1318.5	E6		88
1244.5	1174.7	D6	87	86
1108.7	1046.5	C6	85	84
	987.77	B5		83
932.33	880.00	A5	82	81
830.61	783.99	G5	80	79
739.99	698.46	F5	78	77
	659.26	E5		76
622.25	587.33	D5	75	74
554.37	523.25	C5	73	72
	493.88	B4		71
466.16	**440.0**	**A4**	70	**69**
415.30	392.00	G4	68	67
369.99	349.23	F4	66	65
	329.63	E4		64
311.13	293.67	D4	63	62
277.18	**261.6**	**C4**	61	**60**
	246.94	B3		59
233.08	220.00	A3	58	57
207.65	196.00	G3	56	55
185.00	174.61	F3	54	53
	164.81	E3		52
155.56	146.83	D3	51	50
138.59	130.81	C3	49	48
	123.47	B2		47
116.54	110.00	A2	46	45
103.83	97.999	G2	44	43
92.499	87.307	F2	42	41
	82.407	E2		40
77.782	73.416	D2	39	38
69.296	65.406	C2	37	36
	61.735	B1		35
58.270	55.000	A1	34	33
51.913	48.999	G1	32	31
46.249	43.654	F1	30	29
	41.203	E1		28
38.891	36.708	D1	27	26
34.648	32.703	C1	25	24
	30.868	B0		23
29.135	27.500	A0	22	21

STEP 3: FOURIER TRANSFORMATION

- Perform the Fourier transformation for each targeting frequency for every time interval (here I use 1/16 second).
- Make a 2D array of amplitudes per key per time step.

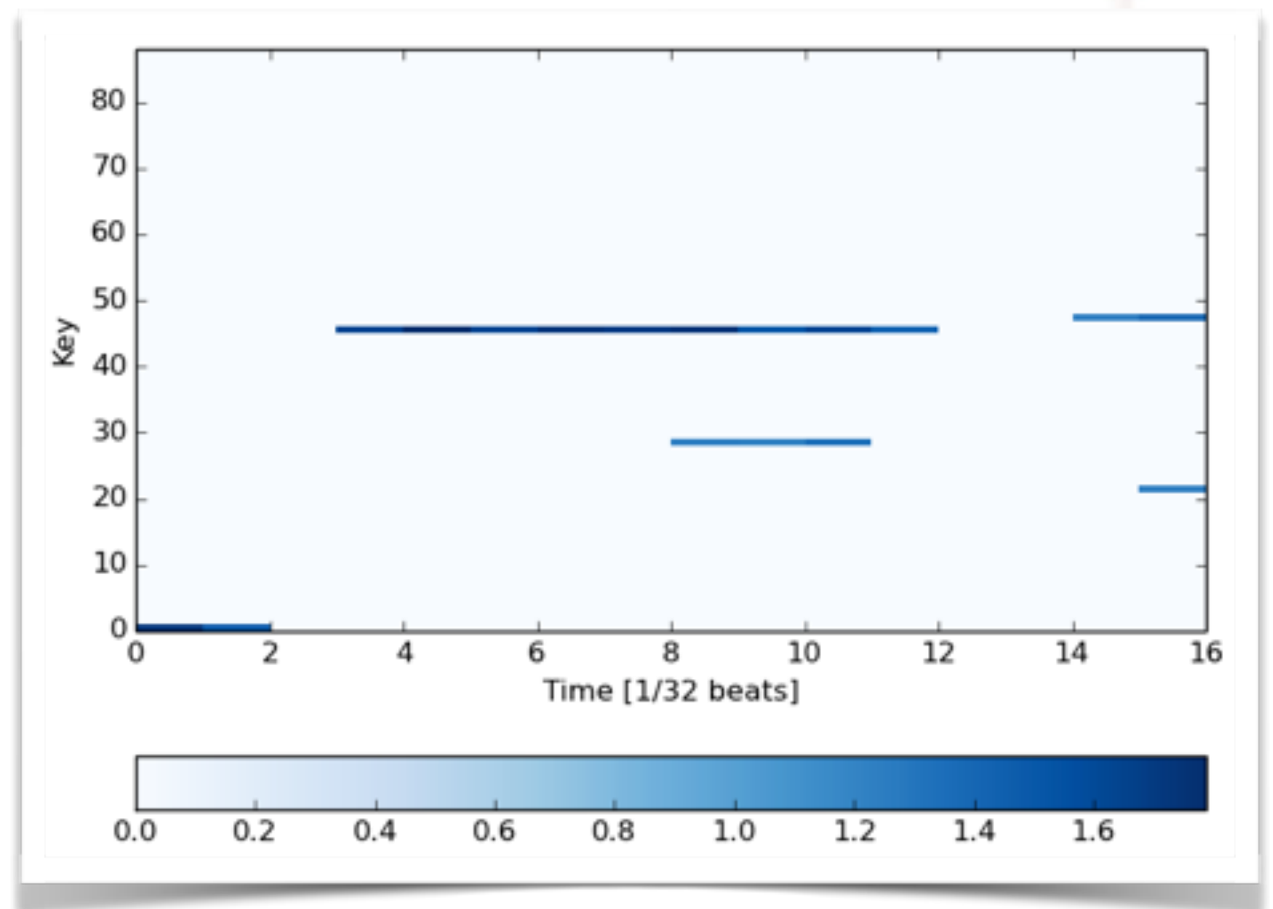
We do see lots of noise and harmonics!



This can be done with NumPy/SciPy easily. Although it will take a while to read the manual.

STEP 4: FILTERING

- Here I use a very simple hand-made "cleaning-up" code.
- For each time interval: sort the keys according to the amplitudes; pedestal is obtained by averaging the lower-amplitude keys (I took weaker 72 keys).
- Subtract the pedestal, and clean up the nodes with a threshold (2.5σ cut here).
- Remove isolated notes.
- Also wipe out any near by notes and harmonics.



STEP 5: CONVERT TO MIDI/SHEET MUSIC

- Joint the block and write to a MIDI file (with the help of MidiUtil package).
- Use some free software (I tried MidiSheetMusic, although it is not very precise...) to read the midi and generate the sheet music.

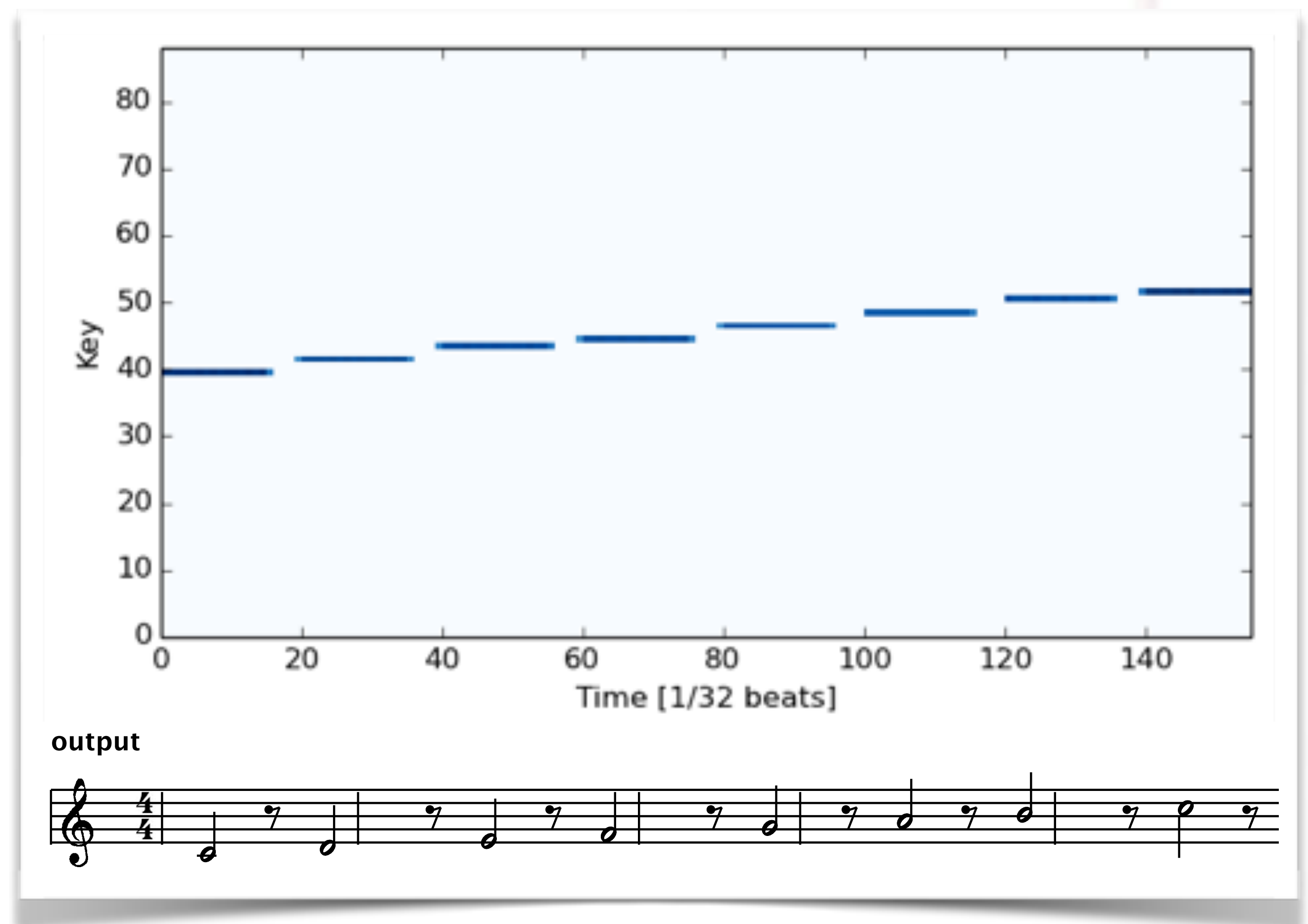
output

PERFORMANCE TEST

- Surely we shall start with something much simpler in order to ensure the code is working.
- Test sample #1: 8 single notes, from C4 to C5, each note takes 1 sec (with 1/4 sec mute time, sine wave only).

Original WAVE

Converted MIDI

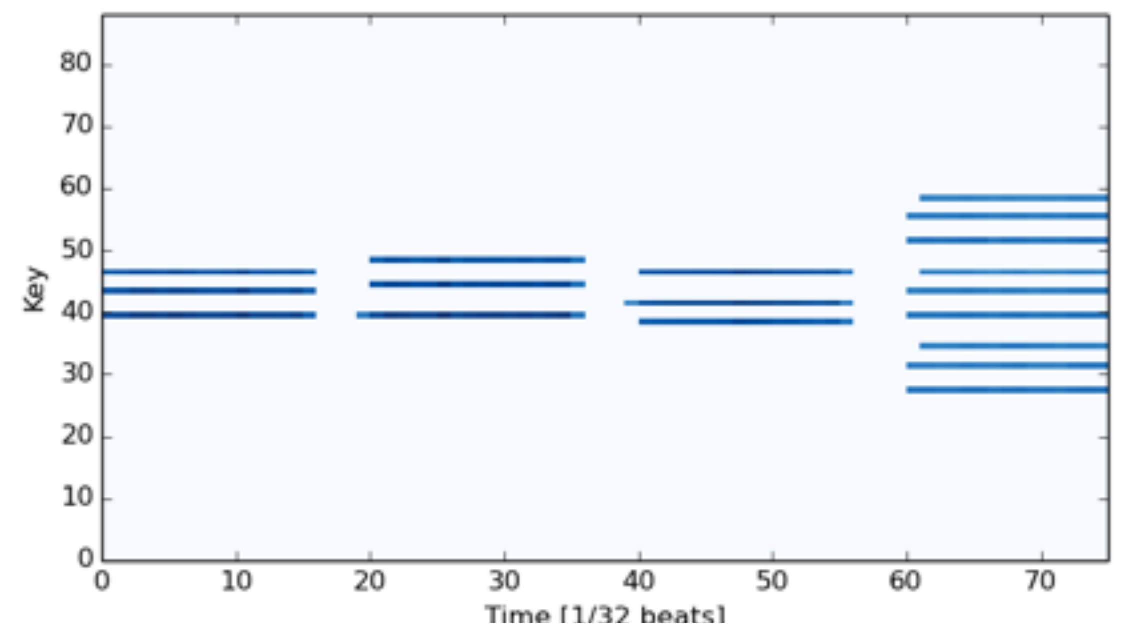


PERFORMANCE TEST (II)

- But the real songs have chords, in most of the cases...
- Test sample #2: 4 chords, each takes 1 sec (also sine waves)
  C4+E4+G4,
  C4+F4+A4,
  B3+D4+G4,
  C3+E3+G3+C4+E4+G4+C5+E5+G5

Original WAVE

Converted MIDI

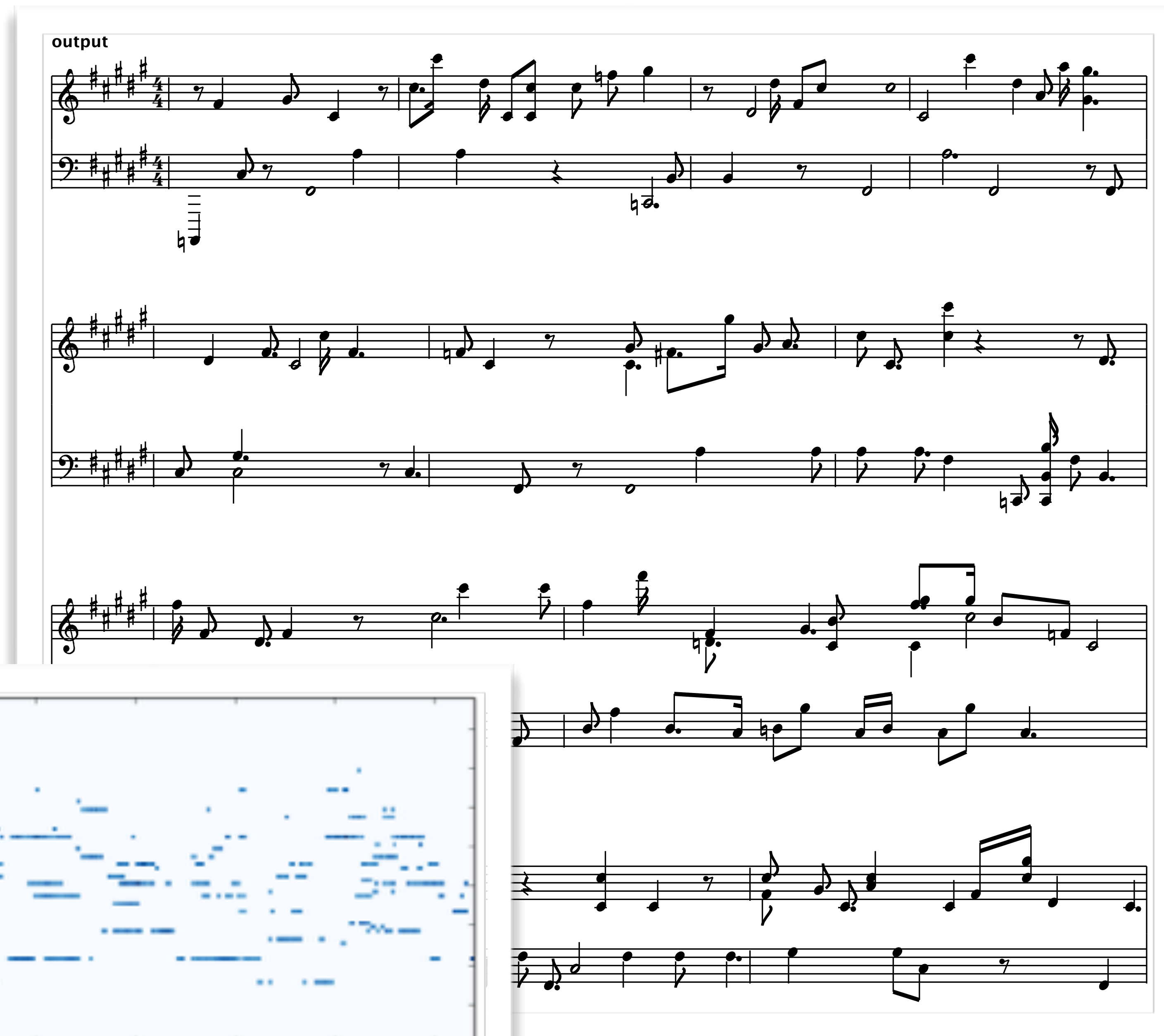


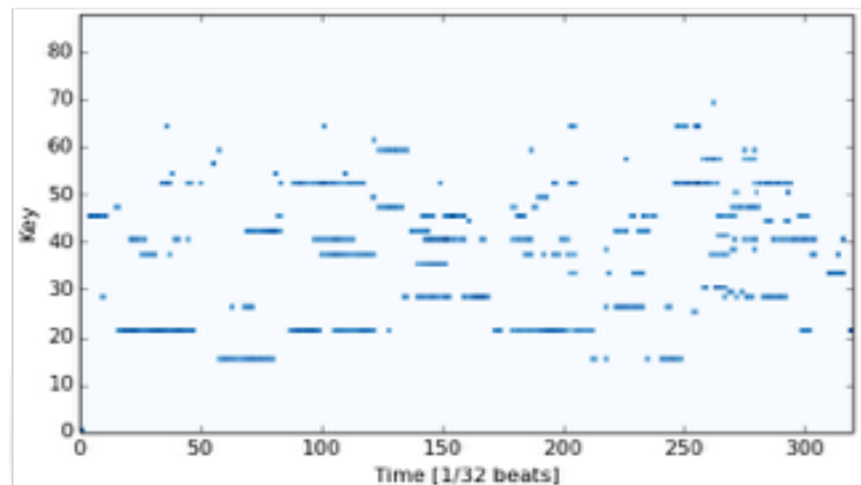
A REAL SONG?

- How about a real song? Well, this is much harder...
- Test song: Dvořák: Humoresque In G Flat, Op. 101/7 (Yo-Yo Ma + Itzhak Perlman)

A REAL SONG? (II)

- The sheet music would look like this:





Converted MIDI

Converted MID
(lower threshold)

COMMENTS

- You probably noticed this is not trivial to analyze a real song with such a super naive code (*but it is already fancy enough, right?*)
- With some more googling, it seems that doing such thing (analyzing a wave and catching the right pitch) is a research level (still in development) topic.
- With the support of NumPy and SciPy, doing all the work above only requires ~200 lines of coding. It is not difficult at all — surely it will not be easy if you want to improve the performance further.

All the work done here is already a nice demo of **numerical analysis** (and it is **FUN**!).

INTERMISSION

- Anybody wants to sing a song and let’s convert it to a MIDI file?

*Then we will come back with the first step toward numerical analysis — **errors in computation**.*

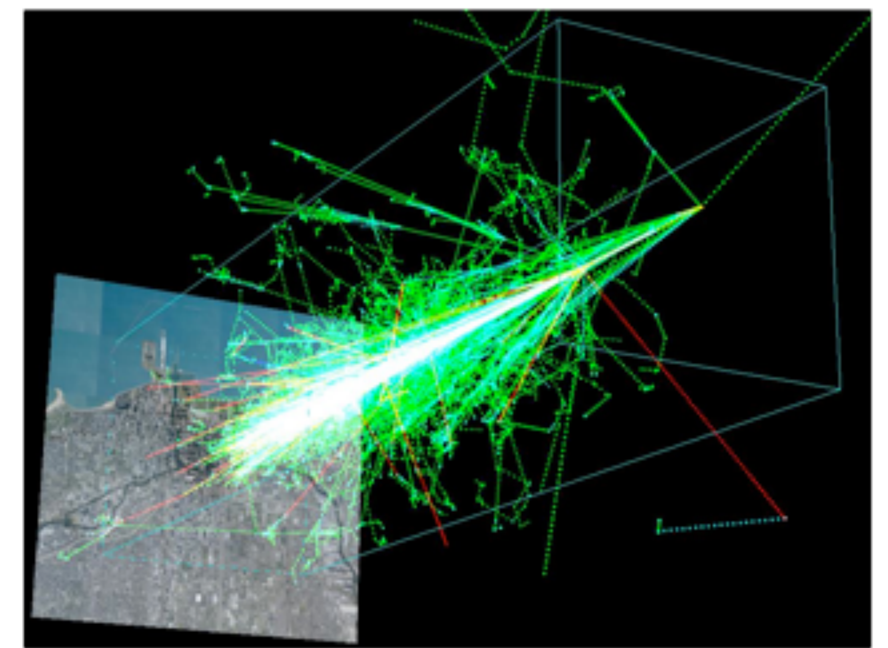
FIRST STEP TOWARD NUMERICAL ANALYSIS: ERRORS IN COMPUTATION

- As you may already know: if you pick up a calculator and insert any number, and start to press [√] for many many times, in the end you'll finally reach exactly the number "1".
- This is nothing but a simplest show of computation uncertainty.

FIRST STEP TOWARD NUMERICAL ANALYSIS:

ERRORS IN COMPUTATION (II)

- Types of errors:
  - **Blunders / human error / bugs:**
    Well, actually this is the #1 case. Typographical errors many enter your program or data, running a wrong program, inserting a wrong data file.
  - **Random errors:**
    Although it may be rare, but the chance is not absolute zero to have some random error simply due to unstable power, unstable system, noise, or even cosmic ray (this is a serious problem for the PC in the space).



FIRST STEP TOWARD NUMERICAL ANALYSIS:

ERRORS IN COMPUTATION (III)

- Types of errors (cont.):
  - **Approximation errors:**
    Basically, this is the error due to the selected algorithm. In principle if we throw away some higher order term, naturally we have this error in the calculations.
  - **Roundoff errors:**
    Surely, the numbers (especially the float point numbers) in the computers are not infinitely precise. Then it's very easy to have this round-off error at the end of the digits.

Today we will discuss these two types of errors. Assuming I made no mistake and the cosmic muons do not hit my PC.

BITS, BYTES, INTEGERS...

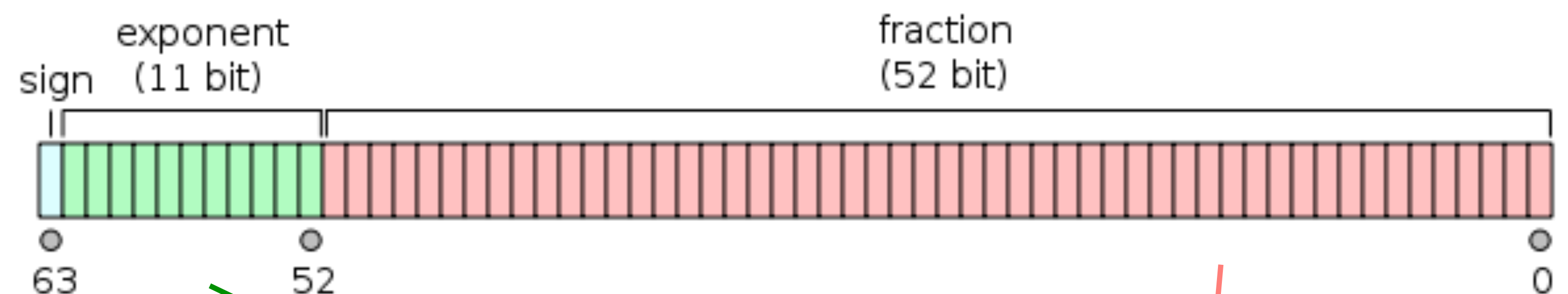
- A bit is the basic unit in computing and physically implemented with a two-state device (**0 or 1**).
- Historically, the byte was the number of bits used to encode a single character. Now it has been fixed to 8 bits, permitting the values between **0 and 255 (=2^8-1)**.
- In python, the integer are implemented using long in C, which gives them 8 bytes (64 bits) of precision. The long integers in python have unlimited precision.

Range of integer: **$-(2^{63}) \sim (2^{63})-1$**
The long type in python can be very ***long***, e.g.
123456789123456789123456789123456789123456789

In python 2, the long integers and integers are different (need to add"L" in the end for python 2).

FLOATING POINT ARITHMETIC

- Floating-point numbers are represented in computer hardware as base 2 (binary) fractions.
- In python, floating point numbers are implemented using double in C (64 bits in total). The internal representation follows the IEEE 754 binary64 standard with 3 components:
  - Fraction precision: **53 bits (52 bits explicitly stored)**
  - Exponent width: 11 bits
  - Sign bit: 1 bit



The real value is expressed as

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-1023} \times \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right)$$

FLOATING POINT ARITHMETIC (II)

- Given the “fraction” part of float point number has a limited precision (up to 52+1 bits), the float point number cannot be 100% precise for most of decimal fractions.
- In general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers.
- The precision of a 64-bit float point number is approximately **16 decimal digits**.
- Some factors:

Largest number: $1.7976931348623157 \times 10^{+308}$
Minimal positive number: $2.2250738585072014 \times 10^{-308}$
Machine epsilon: $\sim 2.22 \times 10^{-16}$

FLOATING POINT ARITHMETIC (III)

- It is good to keep in mind that there is no real continuous “real number” in computation.
- For example, 0.1 is not 1/10, but 0.1000000000000000055511151231257827021181583404541015625.
- It is not surprising at all to see something like this:

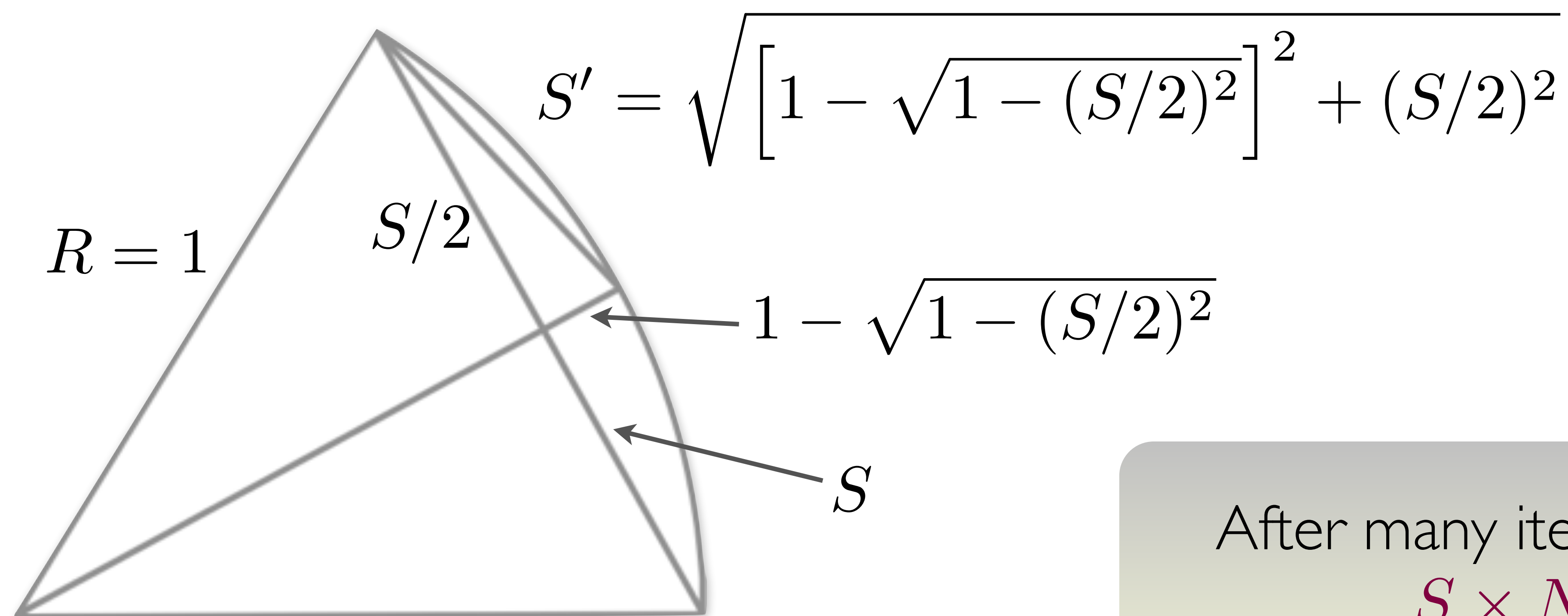
```
>>> 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
0.9999999999999999
>>> 0.1+0.2-0.3
5.551115123125783e-17
```

Spoon is not real,
real number is also not real...

Let's take the standard example of π calculation!

GIVE ME A π

- Let's practice a very classical calculation of π: approximating a circle by polygons (**Liu Hui method**):

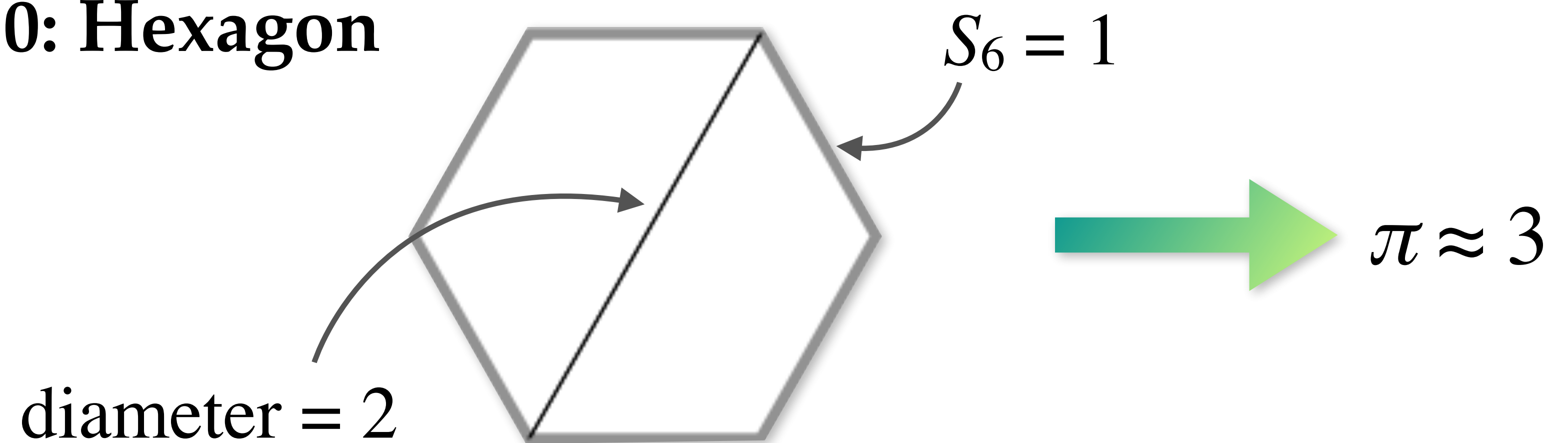


After many iterations:

$$\pi \approx \frac{S \times N_{\mathrm{sides}}}{2}$$

A NAIVE IMPLEMENTATION

- **Step 0: Hexagon**



- **Step 1: Dodecagon**

$$S_{12} = \sqrt{\left[1 - \sqrt{1 - (S_6/2)^2}\right]^2 + (S_6/2)^2} = \sqrt{2 - \sqrt{4 - S_6^2}}$$

$$S_{12} = \sqrt{2 - \sqrt{3}} \approx 0.5176$$

$$\pi \approx \frac{S_{12} \times 12}{2} \approx 3.1058$$

A NAIVE IMPLEMENTATION (II)

- Coding time! Just a plain python code can do this:

```
import math

nsides = 6
length = 1.

for i in range(20):
    length = (2. - (4. - length**2)**0.5)**0.5
    nsides *= 2
    pi = length*nsides/2.

    print('-'*30)
    print('Polygon of',nsides,'slides:')
    print('pi(calc) = %.15f' % pi)
    print('diff = %.15f' % abs(math.pi-pi))
```

l201-example-01.py

RESULTS

- Output:

```
Polygon of 12 slides:
pi(calc) = 3.105828541230250, diff = 0.035764112359543

Polygon of 24 slides:
pi(calc) = 3.132628613281237, diff = 0.008964040308556

Polygon of 48 slides:
pi(calc) = 3.139350203046872, diff = 0.002242450542921

Polygon of 96 slides:
pi(calc) = 3.141031950890530, diff = 0.000560702699263

Polygon of 192 slides:
pi(calc) = 3.141452472285344, diff = 0.000140181304449
```

*Smaller **approximation error** with more sides (closer to a real circle).*

RESULTS (II)

- How about more steps?

```
384 slides:      pi(calc) = 3.141557607911622, diff = 0.000035045678171
768 slides:      pi(calc) = 3.141583892148936, diff = 0.000008761440857
1536 slides:     pi(calc) = 3.141590463236762, diff = 0.000002190353031
3072 slides:     pi(calc) = 3.141592106043048, diff = 0.000000547546745
6144 slides:     pi(calc) = 3.141592516588155, diff = 0.000000137001638
12288 slides:    pi(calc) = 3.141592618640789, diff = 0.000000034949004
24576 slides:    pi(calc) = 3.141592645321216, diff = 0.000000008268577
49152 slides:    pi(calc) = 3.141592645321216, diff = 0.000000008268577
98304 slides:    pi(calc) = 3.141592645321216, diff = 0.000000008268577
196608 slides:   pi(calc) = 3.141592645321216, diff = 0.000000008268577
393216 slides:   pi(calc) = 3.141593669849427, diff = 0.000001016259634
786432 slides:   pi(calc) = 3.141592303811738, diff = 0.000000349778055
1572864 slides:  pi(calc) = 3.141608696224804, diff = 0.000016042635011
3145728 slides:  pi(calc) = 3.141586839655041, diff = 0.000005813934752
6291456 slides:  pi(calc) = 3.141674265021758, diff = 0.000081611431964
```

Oh-oh...

WHAT'S WRONG HERE?

- Go back to the recursive formula:

$$S' = \sqrt{2 - \sqrt{4 - S^2}}$$

S is actually a very small number!
When we do the calculation, we have to worry about the finite accuracy of the float point number.

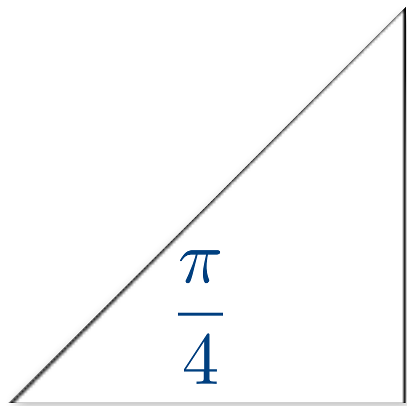
For a **6291456** slides polygon, S ~ 1/1,000,000 (1 over 1M):

$$4 - S^2 \approx 4 - 10^{-12}$$

It's already not too far from the limit of a double-precision float point number!
This is a typical **round-off error**!

ANOTHER CLASSICAL METHOD: LEIBNIZ FORMULA

- Let's start from the trigonometric function:



$$\tan\left(\frac{\pi}{4}\right) = 1 \ \rightarrow \ \arctan(1) = \frac{\pi}{4}$$

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots$$

$$\rightarrow \ 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

Without adding "small" number to "big" number anymore, always adding smaller and smaller numbers.

ANOTHER CLASSICAL METHOD: LEIBNIZ FORMULA (II)

- Coding time. Such a calculation can be done easily with python!

```
import math

pi = 0.
numerator = 1.

for n in range(1001):  ⇐ sum up to n=1000 first!
    pi += numerator/(2.*n+1.)*4.
    numerator = -numerator

    if n%100 == 0:
        print('-'*30)
        print('Sum up to',n,'step:')
        print('pi(calc) = %.15f' % pi)
        print('diff = %.15f' % abs(math.pi-pi))
```

1201-example-02.py

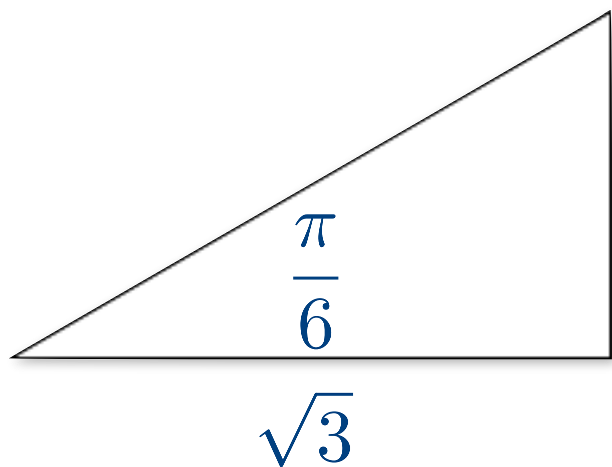
RESULTS: LIEBNIZ FORMULA

- It's working, but very ***inefficient***! Improving one more digit takes 10x steps in the summation:

```
Sum up to 100 step:
pi(calc) = 3.151493401070991, diff = 0.009900747481198

Sum up to 1000 step:
pi(calc) = 3.142591654339544, diff = 0.000999000749751

Sum up to 10000 step:
pi(calc) = 3.141692643590535, diff = 0.000099990000741

Sum up to 100000 step:
pi(calc) = 3.141602653489720, diff = 0.000009999899927

Sum up to 1000000 step:
pi(calc) = 3.141593653588775, diff = 0.000000999998981
```

A LITTLE BIT OF IMPROVEMENT (A TRICK!)

- It's not really hard: we can just pick up a smaller x!



$$\tan\left(\frac{\pi}{6}\right) = \frac{1}{\sqrt{3}} \rightarrow \arctan\left(\frac{1}{\sqrt{3}}\right) = \frac{\pi}{6}$$

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - ...$$

$$\rightarrow \frac{1}{\sqrt{3}}[1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 9} - \frac{1}{7 \cdot 27} + \frac{1}{9 \cdot 81} - ...] = \frac{\pi}{6}$$

It converges much quicker then the previous version. This is due to that the Taylor expansions are calculated around x = 0.

A LITTLE BIT OF IMPROVEMENT (II)

- Coding time again!

```python
import math

pi = 0.
numerator = 1./3.**0.5

for n in range(31):
    pi += numerator/(2.*n+1.)*6.
    numerator *= -1./3.

    print('-'*30)
    print('Sum up to',n,'step:')
    print('pi(calc) = %.15f' % pi)
    print('diff = %.15f' % abs(math.pi-pi))
```

l201-example-02a.py

A LITTLE BIT OF IMPROVEMENT (III)

- With only <30 terms, the limit of precision already reached (Bravo!)

```
Sum up to 10 step:
pi(calc) = 3.141593304503083, diff = 0.000000650913289

Sum up to 20 step:
pi(calc) = 3.141592653595636, diff = 0.000000000005843

Sum up to 30 step:
pi(calc) = 3.141592653589794, diff = 0.000000000000001
```

REMARK: Surely, this is not the real modern way to calculate π up to many digits.
(And surely this is not the purpose of this course...so no worry.)

Calculation of π is in fact rather interesting. Left for your own study!

REMARK: HOW THE ERRORS BEHAVE

- After N iterations of computing –
  - **Approximation errors:**
    In principle, the approximation errors will be reduced if we go for more iterations (e.g. more terms in Taylor expansions).

$$\epsilon_{\text{approx}} \approx \frac{\alpha}{N^{\beta}}$$

(α,β are algorithm dependent parameters)

  - **Roundoff errors:**
    The roundoff error goes to the opposite direction, the more iterations, the error actually accumulates.

$$\epsilon_{\text{roundoff}} \approx \sqrt{N}\epsilon_m$$

(ε_m is machine dependent precision)

REMARK:
HOW THE ERRORS BEHAVE

- Limitation of the total error:

$$\epsilon_{\text{total}} \approx \epsilon_{\text{approx}} + \epsilon_{\text{roundoff}} \approx \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_m$$

Example: π with Liebniz formula (ver. π/4): α ≈ 1, β ≈ 1
(since every 10x steps, we improve the calculation by 1 digit)

If we do the calculation in double precision, when will we catch the smallest (critical) total error?

$$\frac{\partial \epsilon_{\text{total}}}{\partial N} = 0 \ \rightarrow \ \approx -\frac{1}{N^2} + \frac{10^{-16}}{2\sqrt{N}}$$

$$\epsilon_{\text{minimum}} \approx 4 \times 10^{-11}$$
(when N~70,000,000,000)

It's actually just a rough guesstimation, but it's always good to keep this idea in mind when you write your code!

HANDS-ON SESSION

- **Practice 1:**
  The python **decimal** module provides support for decimal floating point arithmetic. By default it the module can already provide a 28 digits or more number and can interact with other python operations normally. For example:

```
import decimal

a, b, c = 0.1, 0.2, 0.3

dec_a = decimal.Decimal('0.1')
dec_b = decimal.Decimal('0.2')
dec_c = decimal.Decimal('0.3')

print('float a+b-c =',a+b-c)
print('decimal a+b-c =',dec_a+dec_b-dec_c)
```

⇐ 'Decimal' object instead of normal float type

```
float a+b-c = 5.55111512313e-17
decimal a+b-c = 0.0
```

HANDS-ON SESSION

- Modify `1201-example-01.py` to calculate π by replacing the numbers all with the **Decimal()** object and see what you can get!
- Hint:

```
import math
                ⇐ add import decimal
nsides = 6
length = 1.     ⇐ replaced by decimal.Decimal('1.')

for i in range(20):
    length = (2. - (4. - length**2)**0.5)**0.5
    nsides *= 2                          ⇑ to be re-written carefully!!
    pi = length*nsides/2.  ⇐ to be re-written carefully!!

    print('-'*30)
    print('Polygon of',nsides,'slides:')
    print('pi(calc) = %.15f' % pi)  ⇐ you can replace it by float(pi) for now
    print('diff = %.15f' % abs(math.pi-pi))
```

l201-example-01.py

HANDS-ON SESSION

- **Practice 2:**
  The **Riemann zeta function** is given by

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

  Modify `1201-example-02.py` to calculate π using this formula, and see how accurate you can reach with 1000, 10000, 100000 terms.

How to find Pi ...

$$\pi = 3.14159\,26535\,89793\,23846\,26433\,83279\,50288\,41971\,69399\ldots$$

$$\pi = 3 + \frac{1}{7+}\,\frac{1}{15+}\,\frac{1}{1+}\,\frac{1}{292+}\,\frac{1}{1+}\,\frac{1}{1+}\,\frac{1}{1+}\,\frac{1}{2+}\,\frac{1}{1+}\,\frac{1}{3+}\,\frac{1}{1+}\,\frac{1}{14+}\,\frac{1}{2+}\,\frac{1}{1+\cdots}$$

$$\pi = (-1)\sqrt{-1}\log(-1) \qquad \pi = \mathrm{RootOf}[\sin\theta] \quad (3 < \theta < 4)$$

$$\pi = 4\arctan 1 \qquad \pi = 4\left(\arctan\frac{1}{2} + \arctan\frac{1}{3}\right)$$

$$\pi = \left(\int_{-\infty}^{\infty} e^{-x^2}\,dx\right)^2 \qquad \pi = 16\arctan\frac{1}{5} - 4\arctan\frac{1}{239}$$

$$\pi = 4\int_0^1 \sqrt{1-x^2}\,dx \qquad \pi = \frac{22}{7} - \int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,dx$$

$$\pi = \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} \qquad \pi = \frac{4}{1+}\,\frac{1^2}{2+}\,\frac{3^2}{2+}\,\frac{5^2}{2+}\,\frac{7^2}{2+}\,\frac{9^2}{2+\cdots}$$

$$\pi = 2\prod_{k=1}^{\infty} \frac{(2k)^2}{(2k-1)(2k+1)} \qquad \pi = 3 + \frac{1^2}{6+}\,\frac{3^2}{6+}\,\frac{5^2}{6+}\,\frac{7^2}{6+}\,\frac{9^2}{6+\cdots}$$

$$\pi = \sqrt{\frac{6}{\prod_{\text{primes } p} 1 - \frac{1}{p^2}}} \qquad \pi = \frac{2}{\prod_{k=1}^{\infty} \frac{\left[a_0 = 0,\, a_k = \sqrt{2 + a_{k-1}}\right]}{2}}$$

$$\pi = \sqrt{\sum_{k=1}^{\infty} \frac{6}{k^2}} \qquad \pi = \frac{99^2}{\sqrt{8}\sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}}$$

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$$

$$\pi = \lim_{k\to\infty} \frac{\left(\left[a_0 = 1,\, a_{k+1} = \frac{a_k + b_k}{2}\right] + \left[b_0 = \frac{\sqrt{2}}{2},\, b_{k+1} = \sqrt{a_k b_k}\right]\right)^2}{4\left[t_0 = \frac{1}{4},\, t_{k+1} = t_k - 2^k(a_k - a_{k+1})^2\right]}$$

Some of these can be tried as well!