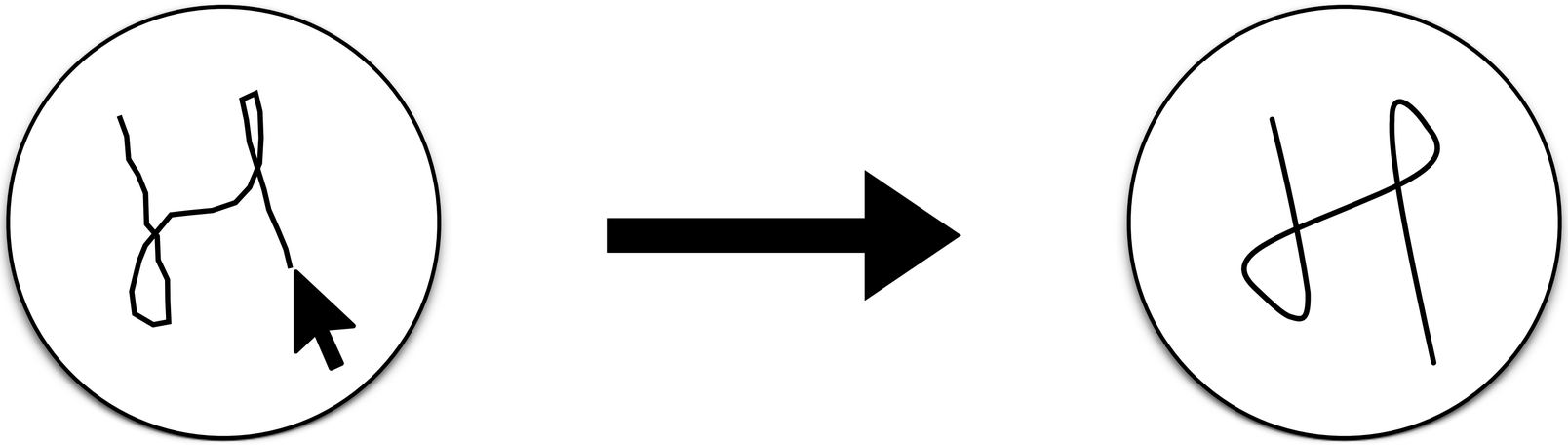
Infinitely Scalable Signatures using PostScript

Gustavo Sutter, July 25th 2024

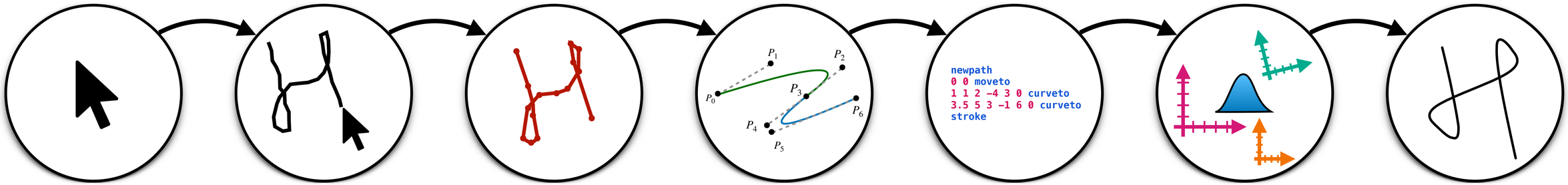
Project Goal



Input Strokes

Scalable Signature
in PostScript

Project Steps



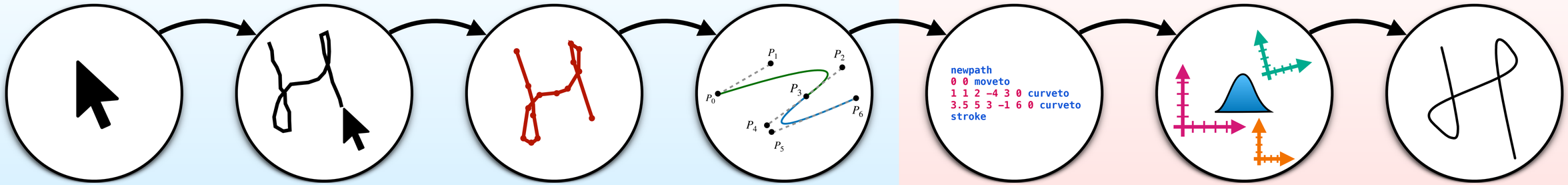
Input Strokes | Drawing Tool (GUI) | Stroke Simplification | Bezier Spline Fitting | Path Creation | Signature Randomization | Scalable Signature in PostScript

Important Design Choices

- **Goal: Use as much PostScript as possible**
    - PostScript is very powerful to display graphics, so let’s make use of that
    - By using PostScript we unsure that our signature is infinitely scalable
    - I program in Python everyday, so let’s use this chance to learn a new language :)
- Thus, we are only using Python to run algorithms that turn the strokes into bezier splines

Drawing the boundary

Python and PostScript



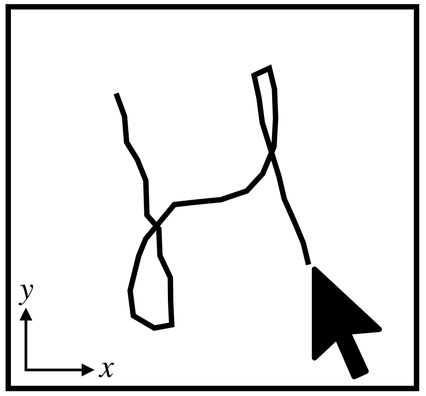
Input Strokes | Drawing Tool (GUI) | Stroke Simplification | Bezier Spline Fitting | Path Creation | Signature Randomization | Scalable Signature in PostScript

Python

PostScript

Drawing Tool
Collecting User Input

- Simple tool using Tkinter (Python library for Tk GUI tool)
- Record strokes (x, y) positions
- Allows basic control (undo, clear, save)
- Shows preview of next steps (simplification and splines)

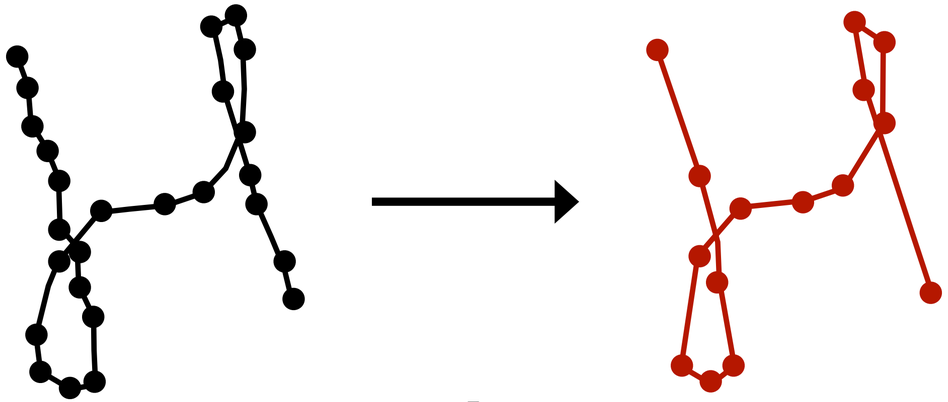


$$\{(200,400), (203,398), \ldots, (422,230)\}$$

Stroke Simplification

Removing Noise from Input

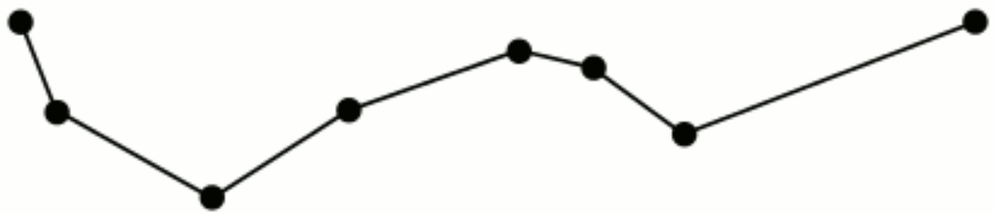
- Input strokes can be very noise, specially when inputing with a mouse
- To make the signature smoother we apply a downsampling algorithm to each stroke
  - Remove points while keeping general shape
- Strokes coordinates are normalized to the [0,1] interval



Stroke Simplification

Ramer–Douglas–Peucker algorithm

- The Ramer–Douglas–Peucker algorithm is used to simplify the stroke
- It is an iterative algorithm that eliminates nodes if they fall within the range between other two nodes

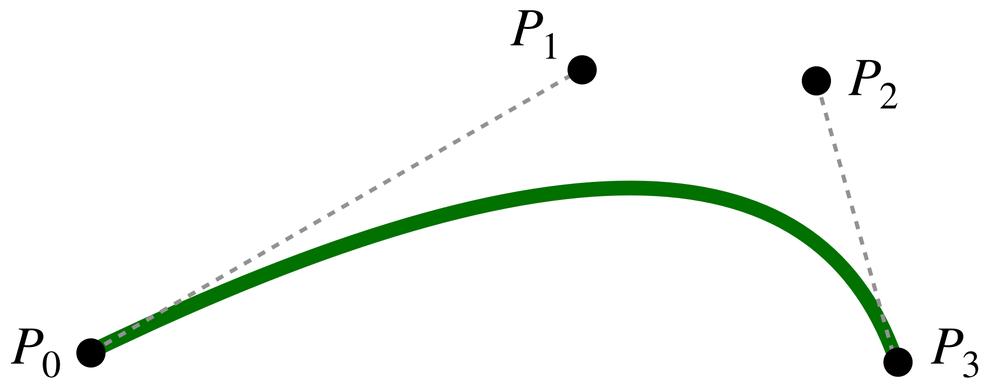


Animation from RDP Wikipedia page

Bézier Spline Fitting

Quick Bézier curves recap

- Bézier curves allows us to create smooth curves by defining control points
- PostScript supports Cubic Bézier curves, which have 4 control points



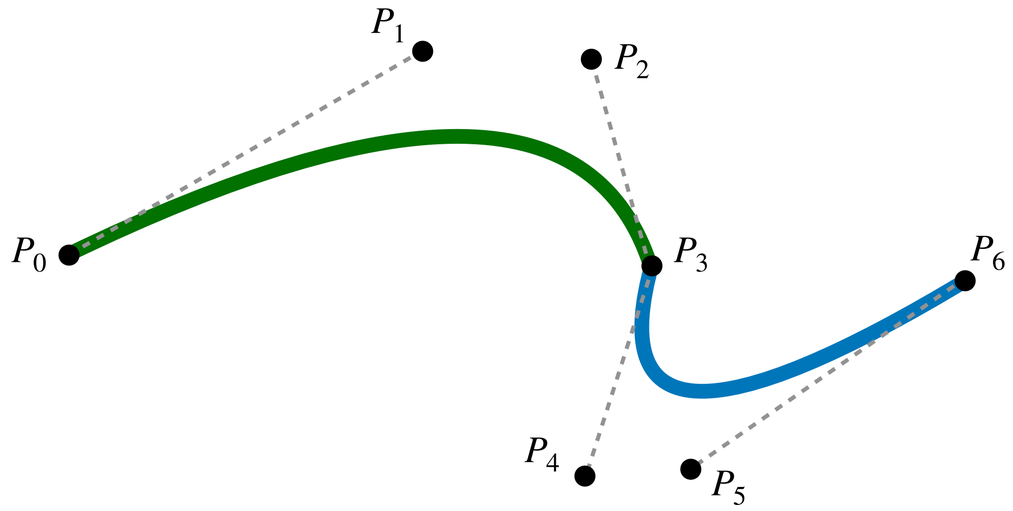
- Mathematically the curve is described by

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3$$

Bézier Spline Fitting

Chaining curves to get spline

- We can connect the end of one curve to the start of next one to create a Bézier spline
- A cubic Bézier splines with N curves has 3N+1 control points
  - The control points that connect two curves are also called knots

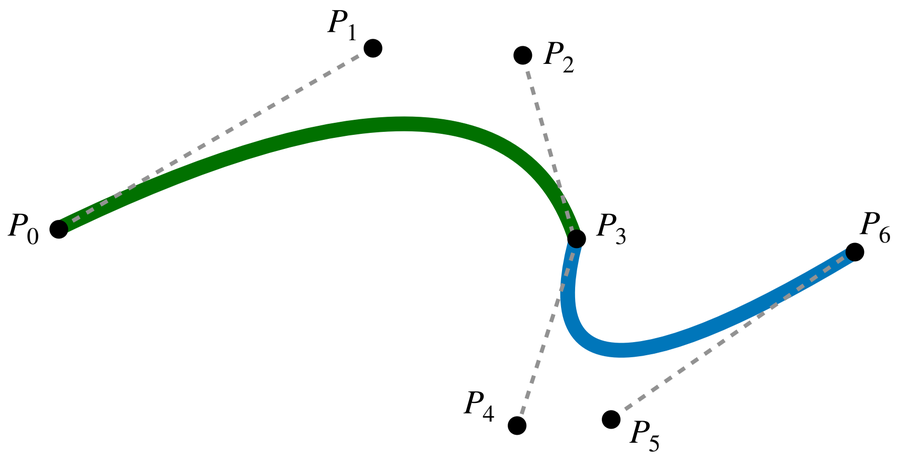


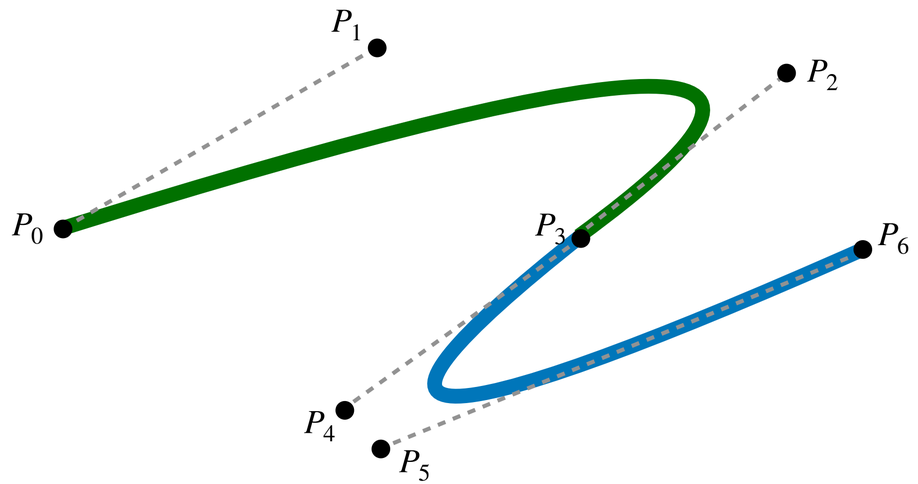
$$B_i(t) = (1-t)^3 P_{3i} + 3(1-t)^2 t P_{3i+1} + 3(1-t)t^2 P_{3i+2} + t^3 P_{3i+3}$$

Bézier Spline Fitting

Defining the control points

- Our goal then is to connect our smoothed stroke points using a cubic Bézier points
- The knots are defined by the stroke, but how about the other control points?





Same knots but different control points

Bézier Spline Fitting

Control Points

- We have 2 unknown control points per segments, that is, **2N unknowns**
- We want to find points that create smooth transition between segments:

$$\begin{cases} B_i'(1) = B_{i+1}'(0) \\ B_i''(1) = B_{i+1}''(0) \end{cases} \qquad \begin{cases} B_0''(0) = 0 \\ B_{N-1}''(1) = 0 \end{cases}$$

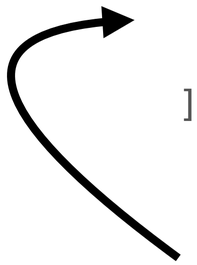
- Now we can just work out the math and arrive at a linear system of equations with **2N equations and 2N unknowns**.

Bézier Spline Fitting

Summing it all up

- For each (simplified) stroke we fit a cubic Bézier spline by solving a linear system of equations
- The output of this step is a **list of lists of control points for each stroke** that is written saved when the user is done
- Concretely, lines like the following are inserted into a PostScript file:

```
/signature [
    [[0.12 0.78] [0.15 0.73] [0.18 0.67] [0.19 0.62] [0.21 0.58] [0.2 0.55] [0.17 0.54]]
    [[0.36 0.6] [0.38 0.59] [0.4 0.59] [0.43 0.6]]
    [[0.41 0.84] [0.41 0.82] [0.41 0.8] [0.42 0.78] [0.43 0.76] [0.44 0.75]]
] def
```



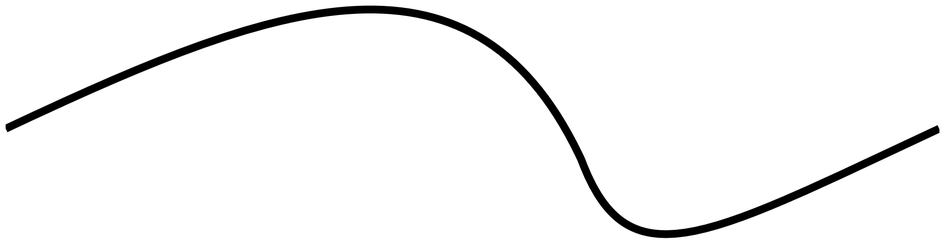
This code is creating an array in PostScript and saving it in a variable called signature :)

Path Creation

Drawing Bézier Splines in PostScript

- Before thinking about your control points let’s see how to create Bézier splines in PostScript

```
5 setlinewidth % Making line thicker

newpath % Initializing empty path
0 150 moveto % Moving to P0
170 230 300 280 370 130 curveto % Defining P1, P2, P3
400 50 450 80 600 150 curveto % Defining P5, P6, P7
stroke % Drawing the path

showpage % Creating page
```



Path Creation

Drawing Strokes

- It was implemented a PostScript procedure that creates a bezier spline given all its control points as a list
- With a generic procedure we have higher control on how each stroke is generated

```
/bezierspline {
    /cpoints exch def % Getting argument from the stack

    /nsegs cpoints length 1 sub 3 div def % Number of segments

    % Getting first point in the spline
    /pstart cpoints 0 get def
    /pstartx pstart 0 get def
    /pstarty pstart 1 get def

    % Creating a new path
    newpath

    % Starting from the first point
    pstartx pstarty moveto

    % For each segment we draw it
    0 1 nsegs 1 sub {
        /i exch def % Loop variable
        /idx i 3 mul def % Index of current segment

        /p1 cpoints idx 1 add get def % p1 = cpoints[idx][1]
        /p1x p1 0 get def % p1x = p1[0]
        /p1y p1 1 get def % p1y = p1[1]

        /p2 cpoints idx 2 add get def % p2 = cpoints[idx][2]ß
        /p2x p2 0 get def % p2x = p2[0]
        /p2y p2 1 get def % p2y = p2[1]

        /p3 cpoints idx 3 add get def % p3 = cpoints[idx][3]
        /p3x p3 0 get def % p3x = p3[0]
        /p3y p3 1 get def % p3y = p3[1]

        p1x p1y p2x p2y p3x p3y curveto % Creates segment
    } for
} def
```

Signature Randomization

Creating different versions of the signature

- Ideally we would like to change the location of the stroke points, which allows lots of flexibility
  - Changing the knots would require us to solve the linear system again
  - Which is not as simple now that we are in PostScript :(
- But we can still generate random variations using two components of PostScript:
  - Random numbers
  - Changes in the coordinate system

Signature Randomization

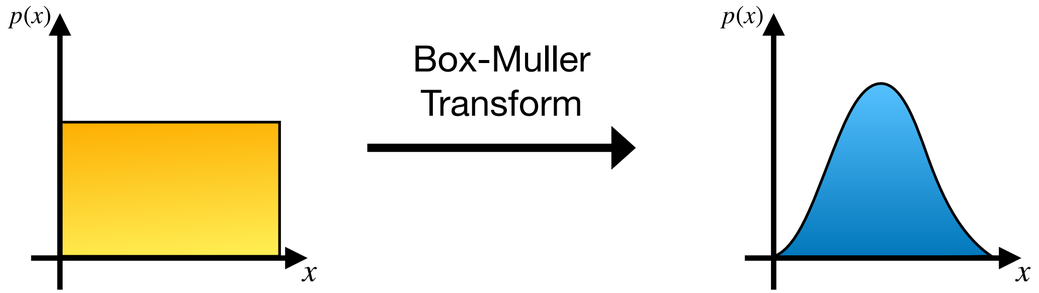
Randomness in Postscript

- Postscript has 3 operators to deal with randomness:
  - **`rand`** - generates a random integer in
  - **`srand`** - sets the random seed
  - **`rrand`** - returns the random seed
- So we can use **`rand`** to generate random numbers and **`srand`** to control our results while we are testing the system

Signature Randomization

Randomness in Postscript

- Usually we want variations to follow a normal distribution
  - Probability drops with higher magnitude
  - But PostScript samples uniformly
- We can create a procedure that transforms uniform samples into normal

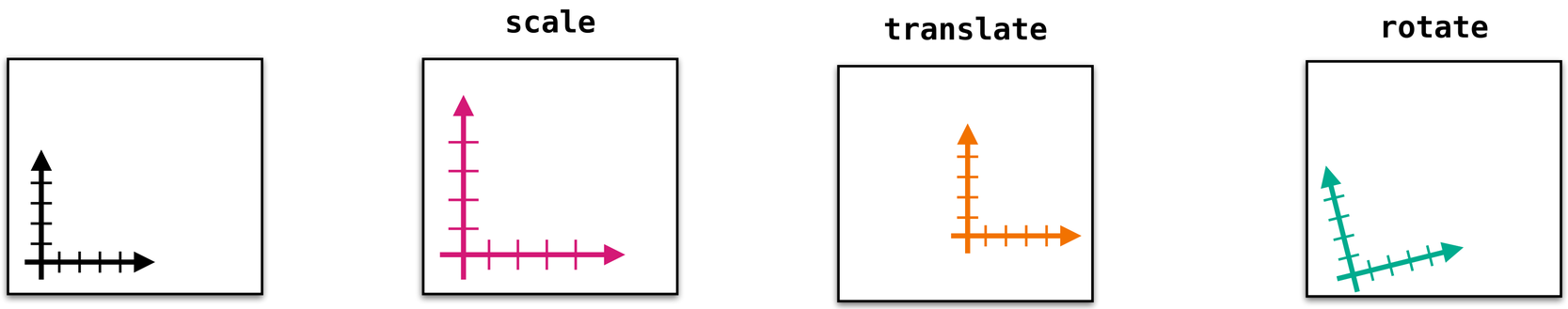



```
% Sample from a normal distribution using
% Box-Muller transform
% Z = sqrt(-2 * ln(U1)) * cos(2 * pi * U2)
% sample = mu + sigma * Z
/randomnormal {
    /sigma exch def
    /mu exch def

    % Two samples from U(0,1)
    /u1 rand 2147483647 div def
    /u2 rand 2147483647 div def

    /t1 u1 ln -2 mul sqrt def % sqrt(-2ln(u1))
    /t2 2 3.14159265 mul u2 mul cos def % cos(2 * pi * u2)

    /z t1 t2 mul def

    z sigma mul mu add
} def
```

Signature Randomization

Modifying Coordinate System

- PostScript has operators that allow us to modify the coordinate system that can be used to determine position, orientation and scale of drawings



Signature Randomization

Randomly Changing Coordinates

- Combining both ideas we can randomly change the coordinate system before drawing each stroke
- To help us PostScript has operators to save and restore the coordinate system

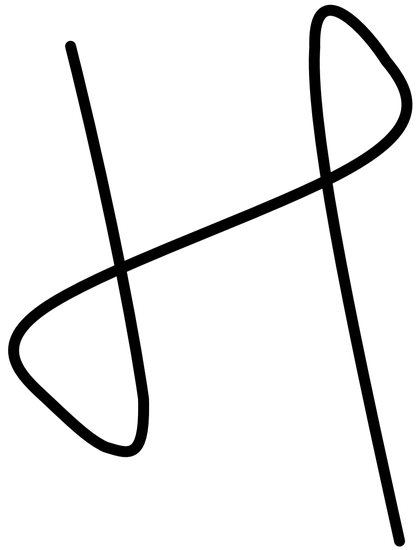
```
/drawsignature {
    /strokeslist exch def % Getting arg from the stack

    /nstrokes strokeslist length def % Number of strokes

    0 1 nstrokes 1 sub {
        /i exch def % Loop variable

        % Sampling random transformations
        /xoffset 0 0.01 randomnormal def
        /yoffset 0 0.01 randomnormal def
        /theta 0 5 randomnormal def
        /newscale 1 0.01 randomnormal def

        % Saving the current coordinate system on the stack
        gsave

        % Applying transformations
        xoffset yoffset translate
        theta rotate
        newscale newscale scale

        % Drawing the stroke
        strokeslist i get bezierspline stroke

        % Recovering the "clean" coordinate system
        grestore
    } for
} def
```

Drawing Scalable Signature

On the home stretch

- Using the method presented up to here we are ready to generate infinitely scalable signatures

```
% Recall that’s what our Python program created
/signature [
    [[0.12 0.78] … [0.17 0.54]]
    [[0.36 0.6]…[0.43 0.6]]
    [[0.41 0.84]…[0.44 0.75]]
] def

% This procedure does everything for us
signature drawsignature
```



Demo

Thank you!