Simply Connectedness
A topological space is simply connected if it is path-connected and every path between two points can be \textit{continuously} transformed into any other such path while preserving the endpoints.
A **homotopy** between two continuous functions $f, g : \mathbb{X} \rightarrow \mathbb{Y}$ is defined to be a *continuous* function $H : \mathbb{X} \times [0, 1] \rightarrow \mathbb{Y}$ such that, if $x \in \mathbb{X}$ then $H(x, 0) = f(x)$, and $H(x, 1) = g(x)$.

A **homotopy** between two continuous functions $f, g : \mathbb{X} \rightarrow \mathbb{Y}$ is a family of continuous functions $h_t : \mathbb{X} \rightarrow \mathbb{Y}$ for $t \in [0, 1]$ such that $h_0 = f$ and $h_1 = g$, and the map $(x, t) \mapsto h_t(x)$ is *continuous* from $\mathbb{X} \times [0, 1]$ to $\mathbb{Y}$.

The two versions coincide by setting $h_t(x) = H(x, t)$.

https://en.wikipedia.org/wiki/Homotopy
A topological space $\mathbb{X}$ is *simply connected* if it is path-connected and any loop in $\mathbb{X}$ defined by $f : S^1 \to \mathbb{X}$ can be contracted to a point.

$\mathbb{X}$ is *simply connected* if and only if it is *path-connected*, and whenever $p : [0, 1] \to \mathbb{X}$ and $q : [0, 1] \to \mathbb{X}$ are two paths (i.e.: continuous maps) with the same start and endpoint $p(0) = q(0)$ and $p(1) = q(1)$, then $p$ can be continuously deformed into $q$ while keeping both endpoints fixed.

Explicitly, there exists a continuous homotopy $F : [0, 1] \times [0, 1] \to \mathbb{X}$ such that $F(x, 0) = p(x)$ and $F(x, 1) = q(x)$.

$\mathbb{X}$ is *path-connected* if there is a path joining any two points in $\mathbb{X}$. 
TDA: Relation to Time Series Analysis
Choudhury et al. (2012)
A story from software visualization

- Detecting circular structures in memory reference traces
- Takens embedding
- Capture recurrent nature of the program
Visualize circular structures in memory access patterns

Example

File: sort.cpp

1: void bubblesort(std::vector<double>& v) {
2:     for(unsigned end=v.size()-1; end >= 0; end--){
3:         bool swapped = false;
4:         for(unsigned i=0; i<end; i++){
5:             if(v[i] > v[i+1]){
6:                 std::swap(v[i], v[i+1]);
7:                 swapped = true;
8:             }
9:         }
10:         if(!swapped) break;
11:     }
12: }
System pipeline

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>0x7fffac539ed8</td>
</tr>
<tr>
<td>Write</td>
<td>0x7fffac539ed0</td>
</tr>
<tr>
<td>Write</td>
<td>0x7fffac539ecc</td>
</tr>
<tr>
<td>Write</td>
<td>0x7fffac539ec8</td>
</tr>
<tr>
<td>Read</td>
<td>0x7fffac539ecc</td>
</tr>
</tbody>
</table>
Capture a memory reference trace

File: sort.cpp

```cpp
void bubblesort(std::vector<double>& v) {
    for (unsigned end=v.size()-1; end >= 0; end--){
        bool swapped = false;
        for (unsigned i=0; i<end; i++){
            if (v[i] > v[i+1]){
                std::swap(v[i], v[i+1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```
Data dependent structure

# loops = # comparisons

File: sort.cpp

```cpp
void bubblesort(std::vector<double>& v){
    for(unsigned end=v.size()-1; end >= 0; end--){
        bool swapped = false;
        for(unsigned i=0; i<end; i++){
            if(v[i] > v[i+1]){
                std::swap(v[i], v[i+1]);
                swapped = true;
            }
        }
        if(!swapped) break;
    }
}
```
Data dependent structure

\[
\# \text{ teeth} = \# \text{ swaps}
\]
Data dependent structure

# teeth = # comparisons
# loops = # comparisons and swaps
Data dependent structure
Algorithm dependent structure

File: matmult.cpp

1: `unsigned int i, j, k;
2: for (i = 0; i < N; i++)
3:   for (j = 0; j < N; j++)
4:     for (k = 0; k < N; k++)
Algorithm dependent structure

File: blocked-matmult.cpp

```cpp
1: unsigned int i, j, k, j0, k0;
2: for (k0 = 0; k0 < N; k0 += b)
3:   for (j0 = 0; j0 < N; j0 += b)
4:     for (i = 0; i < N; i++)
5:       for (k = k0; k < min(k0 + b, N); k++) {  
6:         r = linA[i*N + k];
7:         for (j = j0; j < min(j0 + b, N); j++)  
8:           linC[i*N + j] += r*linB[k*N + j];
9:   }
```
Algorithm dependent structure

Naïve Matrix Multiply

Blocked Matrix Multiply
A story from mice pregnancy detection
A story from mice pregnancy detection

Which mice are pregnant? Joint work with Benjamin Smarr. Data from Smarr et al. (2016).
Jet lagged mice

Which jet lagged mice are pregnant?
Persistent homology of time-varying networks

Visual detection of structural changes in time-varying graphs using persistent homology.

Hajij et al. (2018)
Takens embedding

Maximal persistence: Khasawneh and Munch (2016)
Chatter detection

Khasawneh and Munch (2016)
Reconstruct dynamics using witness complexes

Fig. 1. Classic Lorenz attractor ($\tau = 28, b = 8/3, \sigma = 10$): (a) A $10^5$-point trajectory in $\mathbb{R}^3$ generated using fourth-order Runge–Kutta with a time step of $T = 0.001$. (b) A time-series trace of the $x$ coordinate of that trajectory. (c) A 3D projection of a delay-coordinate embedding with dimension $m = 5$ and delay $\tau = 174T$, following (1).

Garland et al. (2016)
Reconstruct dynamics using witness complexes

Garland et al. (2016)


