HPCSE I - Fall 2022

Tutorial: Diffusion (OpenMP v2) and PCA

2 November 2022

Athena Economides

Computational **S**cience and **E**ngineering **Lab**
ETH Zürich

PART 1: Recap 1D Diffusion

first order time derivative

second order spatial derivative

$$\frac{\partial C(x,t)}{\partial t} = D \frac{\partial^2 C(x,t)}{\partial x^2}$$

diffusion coefficient

Initial Condition:

Boundary Conditions:

$C(B,t)$

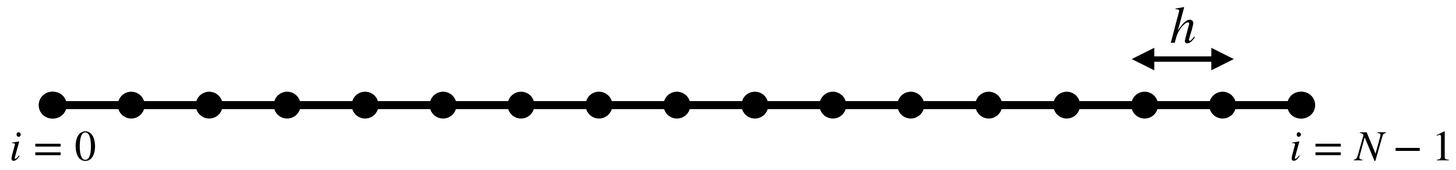
Boundary nodes: $x = 0$ and $x = L$

Recap 1D Diffusion

Discretize equation:

$$\frac{C(x, t+\Delta t) - C(x,t)}{\Delta t} = D\,\frac{C(x+h,t) - 2C(h,t) + C(x-h,t)}{h^2}$$

$$C(x, t+\Delta t) = \frac{D\,\Delta t}{h^2}\left(C(x+h,t) - 2C(h,t) + C(x-h,t)\right)$$



Discretize domain: split domain into N equidistant grid points

grid spacing: $h = \frac{L}{N-1}$

1D Diffusion with OpenMP

Starting point: `tutorials/tutorial06_PCA/diffusion_02/diffusion_naive.cpp`

```
168
169     /* Initialize diffusion system */
170     Diffusion system(D, L, N);
171
172     /* System evolution in time */
173     size_t numSteps = (T / system.dt + 1);
174     auto tstart = std::chrono::steady_clock::now();
175
176     for (size_t step = 0; step <= numSteps; ++step) {
177         double t = system.dt * step;
178         double amount = system.compute_diagnostics(t);
179
180         std::vector<int> hist(10, 0); // histogram with 10 bins
181         system.compute_histogram(hist);
182
183         system.diag.push_back(Diagnostics(t, amount, hist));
184
185         system.advance();
186     }
187     auto tend = std::chrono::steady_clock::now();
188     double ms = std::chrono::duration_cast<std::chrono::milliseconds>(tend - tstart).count();
189
```

we have parallelized *each* function independently

1D Diffusion with OpenMP

Starting point: `tutorials/tutorial06_PCA/diffusion_02/diffusion_naive.cpp`

```
168
169     /* Initialize diffusion system */
170     Diffusion system(D, L, N);
171
172     /* System evolution in time */
173     size_t numSteps = (T / system.dt + 1);
174     auto tstart = std::chrono::steady_clock::now();
175
176     for (size_t step = 0; step <= numSteps; ++step) {
177         double t = system.dt * step;
178         double amount = system.compute_diagnostics(t);
179
180         std::vector<int> hist(10, 0); // histogram with 10 bins
181         system.compute_histogram(hist);
182
183         system.diag.push_back(Diagnostics(t, amount, hist));
184
185         system.advance();
186     }
187     auto tend = std::chrono::steady_clock::now();
188     double ms = std::chrono::duration_cast<std::chrono::milliseconds>(tend - tstart).count();
189
```

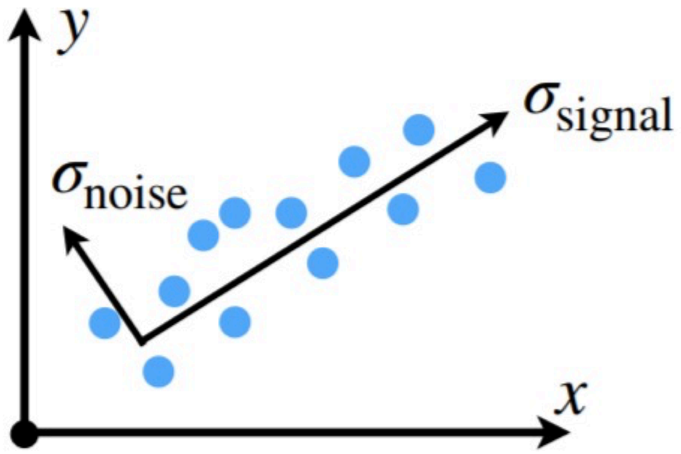
Goal: Improve OpenMP performance.

- **Open a single parallel region outside time-loop.**
- **Adapt rest of code such that all functions are parallelized.**

PART 2: Principal Component Analysis

Two main reasons:

1. Filter out irrelevant information (e.g. measurement noise, data correlations)
2. Data compression: Express data with less information (at the cost of reduced accuracy).



Principal Component Analysis

Example: The wine colour chart.

How do we describe wine?

- colour
- strength
- acidity
- age
- ....

Which of these characteristics are useful in differentiating among wines?

e.g. many of these properties are related so considering all of them can be redundant.

PCA: Construct a new set of characteristics that vary the most between different wines (i.e. have largest variance). Summarize wines with less characteristics.

Inspired by: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

Taken from: https://winefolly.com/tips/wine-color-chart/

Principal Component Analysis

Why *new characteristics* are chosen to have *largest variance* among wines?

- We are looking for wine properties that strongly differ across wines. Such properties would help **summarising** the wines. Instead properties that are the same among wines (e.g. the stillness of a wine after being poured) are useless in summarizing the wine list, since they would be the same among all wines!

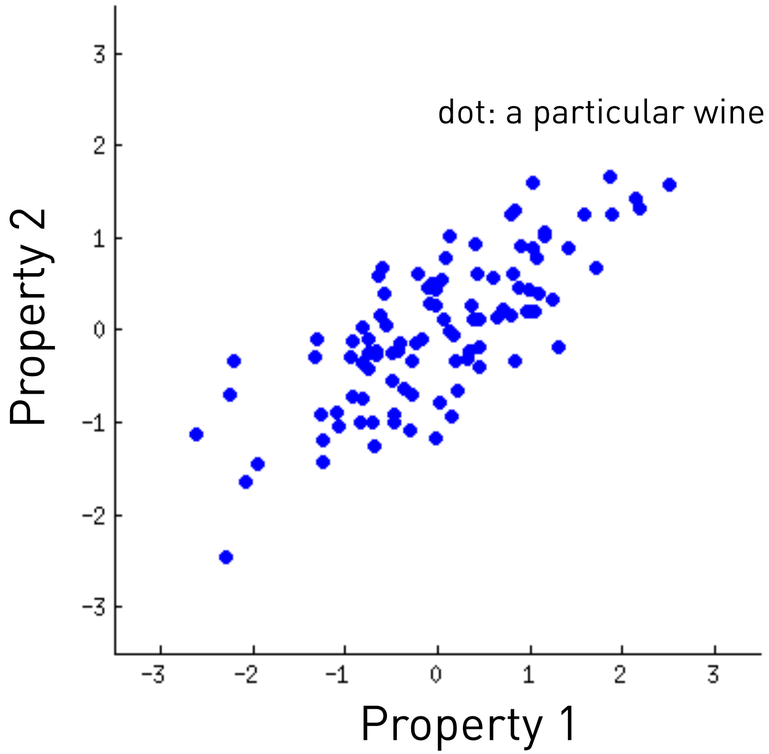
- We are looking for the properties that would allow us to **reconstruct**, the original wine characteristics. A property with no relation to the original characteristics (e.g. the shape of a wine bottle), is useless in reconstruct the original list of characteristics!

Inspired by: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

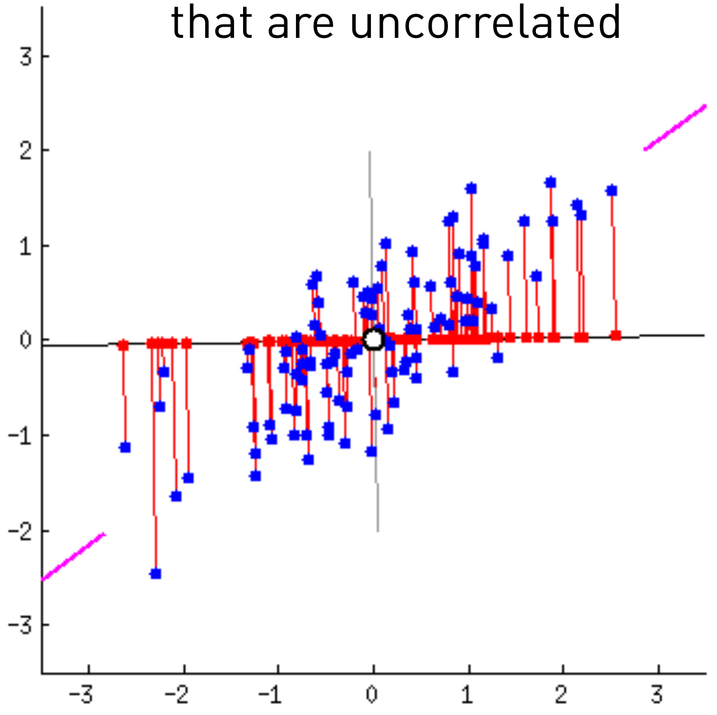
Taken from: https://winefolly.com/tips/wine-color-chart/

Principal Component Analysis

Original wine characteristics: Correlated



PCA: Find new characteristics that are uncorrelated

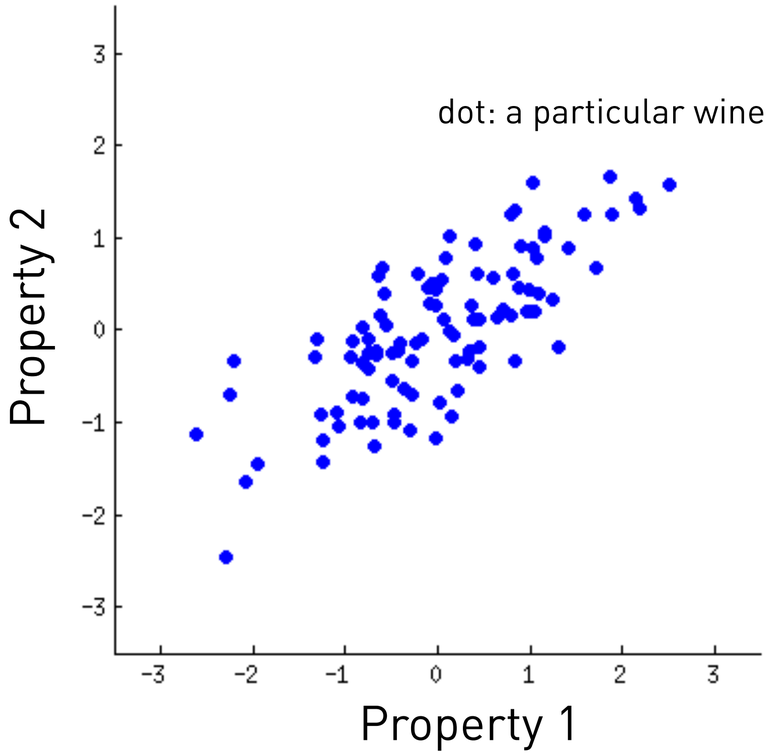


A new property can be constructed by drawing a line through the centre of this wine cloud and projecting all points onto this line. This new property will be given by a linear combination $w_1 x + w_2 y$ where each line corresponds to some particular values of w_1 and w_2.

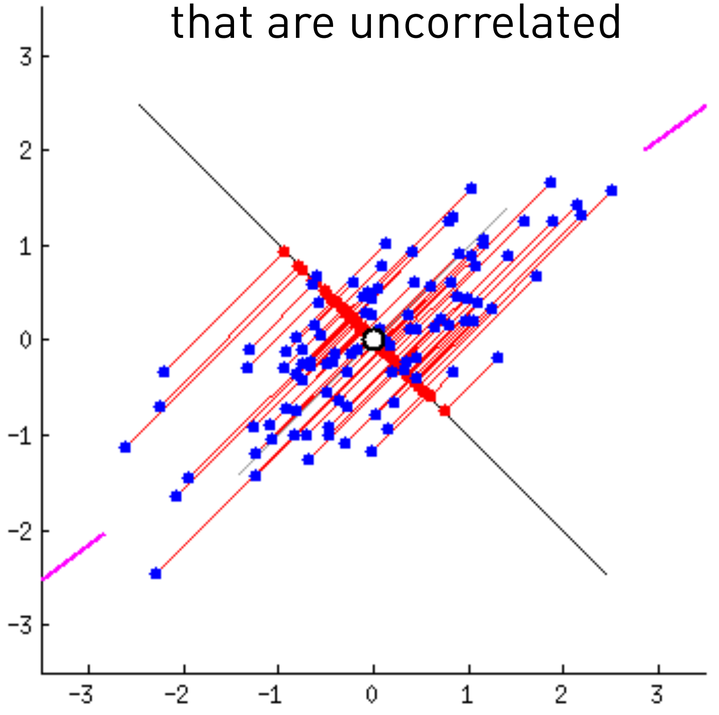
Inspired by: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

Principal Component Analysis

Original wine characteristics: Correlated



PCA: Find new characteristics that are uncorrelated



A new property can be constructed by drawing a line through the centre of this wine cloud and projecting all points onto this line. This new property will be given by a linear combination $w_1 x + w_2 y$ where each line corresponds to some particular values of w_1 and w_2.

Inspired by: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

Principal Component Analysis

PCA for dimensionality reduction

- In practice, computing PCA of a data-set X involves:

1) Subtract off the mean of each measurement type.
2) Compute the covariance matrix ($(n-1)^{-1}XX^T$).
3) Compute the eigen-vectors and -values of $(n-1)^{-1}XX^T$.
4) Sort the eigenvalues and vectors in ascending order.
5) Project the original data onto the new basis (given by the eigenvectors).

Apply in Q2 of HW 3:
- 2D dataset
- High dimensional image data set

Principal Component Analysis

Let's have a look at the skeleton code!

Principal Component Analysis

Let's have a look at the skeleton code!

TODOs: **1. Transpose data**

```
void transposeData(double* data_T, const double* const data, const int N, const int D)
{
     // Data are given in the form data(n,d)=data[n*D+d]
     // This function transposes the data to data(n,d)=data_T[d*N+n]
     // for the purpose of more efficient memory layout
     // (e.g. calculation of the mean)
     for loop over all dimensions (D) {
        for loop over all samples (N) {
             transpose data matrix: data_T[d*N + n] = data[n*D + d];
        }
     }
}
```

Principal Component Analysis

TODOs: **2. Compute mean and standard deviation of data**

```
void computeMean(double* mean, const double* const data_T, const int N, const int D)
{
    // Calculation of the mean (over samples) of the dataset
    // data(n,d)=data_T[d*N+n]
    for loop over all dimensions (D) {
        initialize mean of current dimension -> 0.0
        for loop over number of samples {
            sum samples of dth dimension
        }
        compute mean of dth dimension
    }
}

void computeStd(double* std, const double* const mean, const double* const data_T, const int N, const int D)
{
    // Calculation of the mean (over samples) of the dataset
    // data(n,d)=data_T[d*N+n]
    for loop over all dimensions{
        initialize standard deviation (std) of dth dimension -> 0
        for loop over samples{
            compute std of dth dimension
        }
        sample mean of std accross dth dimension (Hint: divide by N-1)
    }
}
```

Principal Component Analysis

TODOs: **3. Normalize the data**

```
void centerDataColMajor(double* data_T, const double* const mean, const int N, const int D)
{
    std::cout << "Centering data..." << std::endl;
    // COL MAJOR IMPLEMENTATION
    // data(n,d)=data_T[d*N + n]
    TODO: Subtract mean of each dimension ...
    for loop over dimensions{
        for loop over samples {
            subtract mean of dth dimension from sample (n,d)
        }
    }
}
```

Principal Component Analysis

TODOs: **4. Build the Covariance matrix**

$$X \in R^{m,n}, C_X \in R^{m,m} : C_X = \frac{1}{n-1} X X^T$$

```
void constructCovariance(double* C, const double* const data_T, const int N, const int D)
{
    // Construct the covariance matrix (DxD) of the data.
    // data(n,d)=data_T[d*N+n]
    // For the covariance follow the ROW major notation
    // C(j,k)=C[j*D+k]
    Considerations:
    - Dimensions of Covariance matrix is DxD.
    - Covariance matrix is symmetric (compute only the upper triangular part).
    - Remember to compute covariances using the sample mean.
    }
}
```

Principal Component Analysis

TODOs: **5. Compute eigenvectors and eigenvalues**

Use LAPACK: Linear Algebra PACKage
https://netlib.org/lapack/explore-html/d2/d8a/group__double_s_yeigen_ga442c43fca5493590f8f26cf42fed4044.html#ga442c43fca5493590f8f26cf42fed4044

Use function: `dsyev_(...)`

```
subroutine dsyev ( character                          JOBZ,
                   character                          UPLO,
                   integer                            N,
                   double precision, dimension( lda, * ) A,
                   integer                            LDA,
                   double precision, dimension( * )   W,
                   double precision, dimension( * )   WORK,
                   integer                            LWORK,
                   integer                            INFO
                 )
```

```
// First call to dsyen to determine optimal size of work array
// first call dsyev, determine optimal size of work array
dsyev_(both_eigenvectros_eigenvalues, upper_triangular,
order_of_CovMatrix, CovMatrix, dimension_of_CovMatrix, eigenvalues,
work_space, -1, info);


// Second call to dsyen
dsyev_(both_eigenvectros_eigenvalues, upper_triangular,
order_of_CovMatrix, CovMatrix, dimension_of_CovMatrix, eigenvalues,
work_space, &lwork, info);
```

Principal Component Analysis

TODOs: **6. Extract and save the principal components & eigenvalues**

```
void getEigenvectors(double* V, const double* const C, const int NC, const int D)
{
    // Extracting the last rows from matrix C containing the PCA components (eigenvectors of the
covariance matrix) that explain the highest variance.
    // Be careful to extract them in order of descending variance.
    // C(j,d)=C[j*D+d] # ROW MAJOR
    // V(k,d)=V[k*D+d] # ROW MAJOR
    TODO ...
}
```

Principal Component Analysis

TODOs: **7. Dimensionality reduction**

```
void reduceDimensionality(double* data_red, const double* const V, const double* const data_T,
                          const int N, const int D, const int NC)
{
    // data(n,d)=data_T[d*N+n] (transposed dataset)
    // V(k,d)=V[k*D+d]
    // data_red(n,k)=data_red[n*NC + k], K<<D
    for loop: Iterate through all data {
        for loop: Iterate through all dimensions {
        ...
        for loop: Iterate through components {
            Project the original data onto the new basis
        }
        ...
    }
}
```

Principal Component Analysis

TODOs: **8. Reconstruct the data from their compressed form**

```
    void reconstructDatasetRowMajor(double* data_rec, const double* const V, const double* const
data_red, const int N, const int D, const int NC)
    {
        // ROW MAJOR
        // V(c,d)=V[d + c*D]
        // data_red(n,c)=data_red[c + n*NC], C<<D   # ROW MAJOR
        // data_rec(n,d)=data_rec[d + n*D]          # ROW MAJOR
        for loop: Iterate through all data {
            for loop: Iterate through all dimensions {
                ...
                for loop: Iterate through components {
                    Reconstruct data in original dimensions
                }
                ...
            }
        }
    }
```