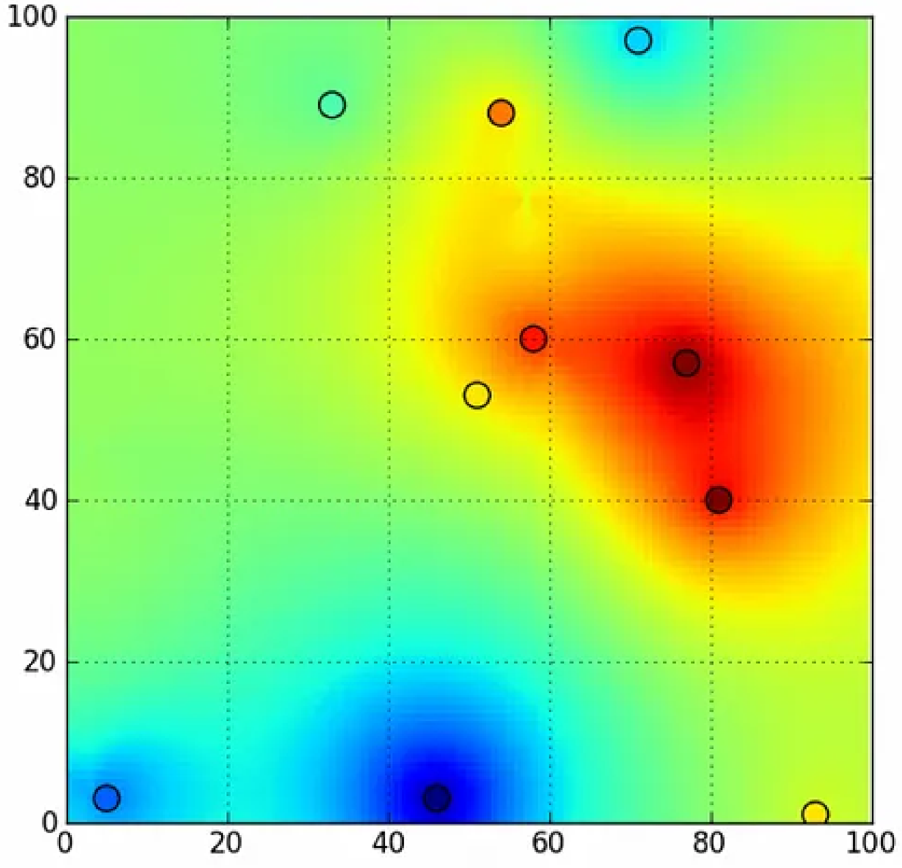
Nearest-neighbor and linear interpolations in 1D with CUDA.

Vitality Learning · Follow
7 min read · Oct 8

Listen Share



The CPU and GPU codes regarding this post are available at the Vitality Learning GitHub repository.

We present some theory behind interpolation and, in particular, nearest-neighbor and linear interpolation and then present CPU (python) and GPU (PyCUDA) implementations thereof.

The interpolation problem

The interpolation problem can be stated as follows: knowing a function *f*(*x*) at the *N*+1 discrete points

$$a = x_0, x_1, \ldots, x_n = b$$

retrieve the values of the function at any point of the (*a*, *b*) interval. Typically, an interpolation scheme sets

$$f(x) = g(x; \alpha_0, \alpha_1, \ldots, \alpha_N)$$

where *g* is the interpolation function and

$$(\alpha_0, \alpha_1, \ldots, \alpha_N)$$

are *N*+1 *interpolation parameters* which are fixed by enforcing that the function to interpolate *f* and the interpolating function *g* are coincident at the interpolation points, namely

$$f(x_j) = g(x_j; \alpha_0, \alpha_1, \ldots, \alpha_N), \quad j = 0, 1, \ldots, N$$

Often, the interpolating function *g* is expressed as a combination of basis functions:

$$g(x; \alpha_0, \alpha_1, \ldots, \alpha_N) = \sum_{i=0}^{N} \alpha_i \phi_i(x)$$

Many times, the function *f* can be well approximated by polynomials, i.e.

$$g(x; \alpha_0, \alpha_1, \ldots, \alpha_N) = \sum_{i=0}^{N} \alpha_i x^i$$

In the latter case, the interpolation parameters are worked out by constraining that

$$f(x_j) = g(x_j; \alpha_0, \alpha_1, \ldots, \alpha_N) = \sum_{i=0}^{N} \alpha_i x_j^i, \quad j = 0, 1, \ldots, N$$

so that the following matrix relation is established

$$\underline{f} = \underline{\underline{A}}\,\underline{\alpha}$$

where the relevant matrix is a Vandermonde matrix which is invertible.

Lagrange interpolation

In the case of Lagrange interpolation, we set

$$g(x) = \sum_{i=0}^{N} f(x_i)\phi_i(x)$$

where the *i*-th basis function is a polynomial of degree *N* whose zeros are the interpolation points except for the *i*-th. In other words,

$$\phi_i(x_j) = \delta_{ij}$$

where the right hand side is the Kronecker delta.

By the fundamental algebra theorem, thanks to the knowledge of its zeros, each relevant polynomial can be factored as

$$\phi_i(x) = C \prod_{j=0, j\neq i}^{N} (x - x_j)$$

where *C* is a constant to be determined. The constant *C* can be found by enforcing that at the non-zero sampling point, the relevant polynomial is unitary, namely

$$\phi_i(x_j) = C \prod_{j=0, j\neq i}^{N} (x_i - x_j) = 1$$

so that

$$C = \frac{1}{\prod_{j=0, j\neq i}^{N} (x_i - x_j)}$$

As a consequence, the interpolating function *g* can be expressed as

$$g(x)=\sum_{i=0}^{N} f(x_i)\prod_{j=0,j\neq i}^{N}\frac{(x-x_j)}{(x_i-x_j)}$$

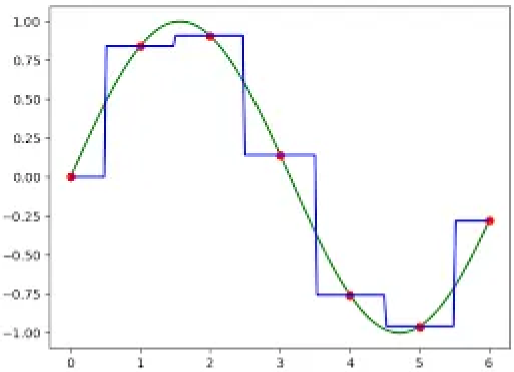
The remainder of the interpolation process is such that the function *f* can be expressed as

$$f(x)=g(x)+\underbrace{\frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)\cdot(x-x_1)\cdot\ldots\cdot(x-x_N)}_{\text{Lagrange remainder}}$$

where

$$\xi(x)\in(a,b)$$

Nearest-neighbor interpolation



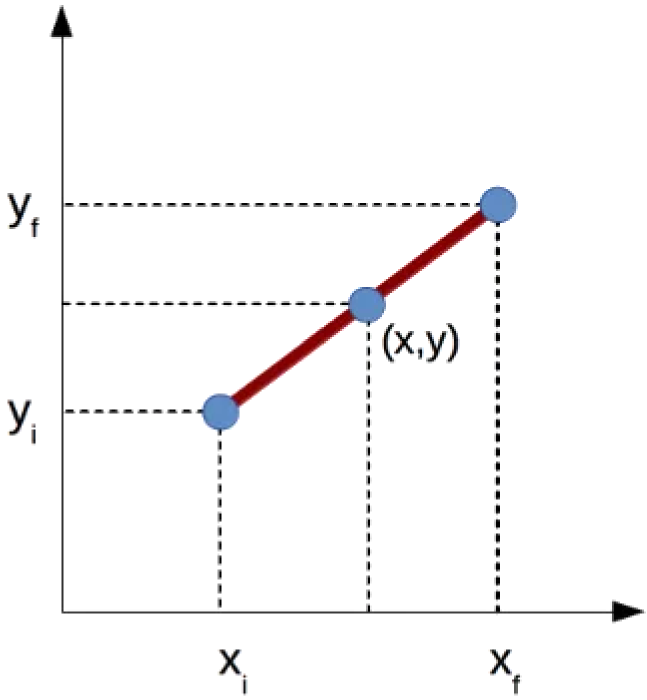
Nearest-neighbor interpolation.

Nearest-neighbor interpolation is a zero-th order polynomial interpolation that assigns the value of the nearest neighbor in the original data to each interpolation point. It is a simple method that does not involve any weighted averaging like linear interpolation (see below).

Linear interpolation

Linear interpolation can be drawn from general Lagrange interpolation in the case $N = 1$. In particular

$$g(x)=f(x_0)\frac{(x-x_1)}{(x_0-x_1)}+f(x_1)\frac{(x-x_0)}{(x_1-x_0)}$$



Linear interpolation.

CPU implementation

In the following, we present the CPU (python) implementations of nearest-neighbor and linear interpolations. The common parts of the implementations will be presented referring to nearest-neighbor interpolation only.

Nearest-neighbor CPU interpolation: python implementation

Let us begin by breaking down the code implementing nearest-neighbor interpolation in 1D in PyCUDA.

The code inputs are the vector `x` of the sampling points, the vector `y` of the sample values and the vector `yi` of the interpolated vector.

In the following, the main steps.

1. *Reshaping Vectors*

The following lines ensure that the vectors `x`, `xi`, and `y` are column vectors.

```
np.reshape(x, (len(x), 1))
np.reshape(xi, (len(xi), 1))
np.reshape(y, (len(y), 1))
```

NNLinearInterpolation1DPyCUDA_1 hosted with ❤ by GitHub view raw

2. *Calculate Spacing*

The reciprocal of the spacing between adjacent sample points in the `x` vector is then calculated as in the following snippet.

```
ndx = 1 / (x[1] - x[0])
```

NNLinearInterpolation1DPyCUDA_2 hosted with ❤ by GitHub view raw

3. *Adjust xi values*

The values in `xi` are shifted by subtracting the minimum value of `x`. This is done to make sure that the indices calculated later are positive.

```
xi = xi - x[0]
```

NNLinearInterpolation1DPyCUDA_3 hosted with ❤ by GitHub view raw

4. *Initialize Output*

The output vector `yi` is initialized with NaN values.

```
yi = float('nan') * np.ones_like(xi)
```

NNLinearInterpolation1DPyCUDA_4 hosted with ❤ by GitHub view raw

5. *Calculate Nearest-Lower-Neighbors Indices*

The indices of the nearest-lower-neighbors for each value in `xi` are calculated.

```
fxi = np.floor(xi * ndx)
```

NNLinearInterpolation1DPyCUDA_5 hosted with ❤ by GitHub view raw

6. *Perform Interpolation*

The code finally checks for out-of-bounds indices and performs linear interpolation for the valid indices. The result is stored in the `yi` vector.

```
flag = np.where((rxi >= 0) & (rxi <= (len(x) - 1)))
yi[flag] = y[np.int32(rxi[flag])]
```

NNLinearInterpolation1DPyCUDA_6 hosted with ❤ by GitHub view raw

Linear CPU interpolation: python implementation

Let us now turn to breaking down the code implementing linear interpolation in 1D in PyCUDA.

The code inputs are the vector `x` of the sampling points, the vector `y` of the sample values and the vector `yi` of the interpolated vector.

Steps 1–5 are the same as for the nearest-neighbor interpolation.

6. *Perform Interpolation*

The out-of-bounds indices are checked and linear interpolation is performed for the valid indices according to the above interpolation formula. The result is once again stored in the `yi` vector.

```
flag = np.where((fxi >= 0) & (fxi <= (len(x) - 2)))
yi[flag] = -(xi[flag] * ndx - (fxi[flag] + 1)) * y[np.int32(fxi[flag])] + (xi[flag] * nd
```

NNLinearInterpolation1DPyCUDA_7 hosted with ❤ by GitHub view raw

GPU implementation

The GPU implementation available at the GitHub repository of Vitality Learning is written in PyCUDA language. Examples on setting up a PyCUDA code are available in the post Five different ways to sum vectors in PyCUDA. In the sequel, we will iluustrate the only CUDA kernels of interest.

Concerning the CUDA kernels, we first present three different versions for linear interpolation: the first uses data stored in global memory, the second exploits texture fetching and the third texture filtering. Finally, we will present also a version performing nearest-neighbor with texture filtering.

In all the considered implementations, each thread is responsible for interpolating one interpolation point.

Linear GPU interpolation: global memory

In the following, an overview of the kernel:

```
__global__ void linear_interpolation_kernel_function_GPU(const float* __restrict__ d_xi

  // Global thread identifier
  int j = threadIdx.x + blockDim.x * blockIdx.x;

  // Ensure each thread works on a valid output point
  if (j < M) {

    // Calculate the position relative to the first input position
    float xi = d_xout[j] - d_xin[0];

    // Calculate the index of the nearest lower neighbor
    int fxi = __float2int_rz(xi * ndx_const);

    // Get the values of the nearest input neighbors
    float dk = d_yin[fxi];
    float dkp1 = d_yin[fxi + 1];

    // Calculate weights for linear interpolation
    float a = xi * ndx_const - truncf(xi * ndx_const);

    // Calculate linear interpolation and assign the result
    d_yout[j] = -(a - 1.f) * dk + a * dkp1;
  }
}
```

NNLinearInterpolation1DPyCUDA_8 hosted with ❤ by **GitHub** **view raw**

1. *Thread identification*

Each thread is associated to a unique global identifier based on its position within blocks and the current block.

```
int j = threadIdx.x + blockDim.x * blockIdx.x;
```

NNLinearInterpolation1DPyCUDA_9 hosted with ❤ by **GitHub** **view raw**

2. *Boundary check*

Ensures that each thread works on a valid output (interpolation) point (within the range `0,...,M-1`).

```
if (j < M) {
```

NNLinearInterpolation1DPyCUDA_10 hosted with ❤ by **GitHub** **view raw**

3. *Relative position calculation*

Calculates the position relative to the first input position. See step #3 of the CPU case.

```
float xi = d_xout[j] - d_xin[0];
```

NNLinearInterpolation1DPyCUDA_11 hosted with ❤ by GitHub view raw

4. *Nearest lower neighbor index calculation*

Calculates the index of the nearest lower neighbor using rounding towards zero.

```
int fxi = __float2int_rz(xi * ndx_const);
```

NNLinearInterpolation1DPyCUDA_12 hosted with ❤ by GitHub view raw

5. *Input values retrieval*

Retrieves the values of the nearest input neighbors needed for interpolation.

```
float dk = d_yin[fxi];
float dkp1 = d_yin[fxi + 1];
```

NNLinearInterpolation1DPyCUDA_13 hosted with ❤ by GitHub view raw

6. *Weight calculation and linear interpolation*

Calculates weights for linear interpolation and assigns the result to the corresponding interpolation point.

```
float a = xi * ndx_const - truncf(xi * ndx_const);
d_yout[j] = -(a - 1.f) * dk + a * dkp1;
```

NNLinearInterpolation1DPyCUDA_14 hosted with ❤ by GitHub view raw

The developed python code sets up constant memory for `ndx_const`. The use of constant memory ensures that the `ndx_const` value is efficiently shared among all threads in the GPU. This is achieved by the following lines:

```
__device__ __constant__ float ndx_const;
```

NNLinearInterpolation1DPyCUDA_15 hosted with ❤ by GitHub view raw

in CUDA.

Furthermore, the python function calling the CUDA kernel is the following:

```
def linear_interpolation_function_GPU(d_xin, d_yin, d_xout, d_yout, ndx, N, M):

  ndx1      = np.array(ndx)
  ndx_ref,_ = mod.get_global("ndx_const")
  cuda.memcpy_htod(ndx_ref, ndx1)

  blockDim                                  = (BLOCKSIZE, 1, 1)
  gridDim                                   = (iDivUp(N, BLOCKSIZE), 1, 1)
  linear_interpolation_kernel_function_GPU(d_xin, d_yin, d_xout, d_yout, np.int32(N), np
```

NNLinearInterpolation1DPyCUDA_16 hosted with ❤ by GitHub view raw

The value of `ndx` is computed from outside that function and moved to constant memory by the following lines:

```
ndx1 = np.array(ndx)
ndx_ref, _ = mod.get_global("ndx_const")
cuda.memcpy_htod(ndx_ref, ndx1)
```

NNLinearInterpolation1DPyCUDA_17 hosted with ❤ by GitHub view raw

Linear GPU interpolation: texture fetch

We now present the solution using texture fetching in an incremental way as before. Using texture memory for data fetching in CUDA can lead to improved memory access patterns and caching benefits, especially for scenarios where memory access patterns have spatial locality. It is a powerful feature that CUDA provides for optimizing memory access in GPU kernels.

The relevant kernel function is reported below:

```
// Define a texture object for 1D float data
texture<float, 1> d_yintexture;

// CUDA kernel for linear interpolation using texture fetching
__global__ void linear_interpolation_kernel_function_GPU_texture(const float* __restric

  // Global thread identifier
  int j = threadIdx.x + blockDim.x * blockIdx.x;

  // Ensure each thread works on a valid output point
  if (j < M) {

    // Calculate the position relative to the first input position
    float xi = d_xout[j] - d_xin[0];

    // Calculate the index of the nearest lower neighbor
    int fxi = __float2int_rz(xi * ndx_const);

    // Fetch the values of the nearest input neighbors from the texture
    float dk = tex1Dfetch(d_yintexture, fxi);
    float dkp1 = tex1Dfetch(d_yintexture, fxi + 1);

    // Calculate weights for linear interpolation
    float a = xi * ndx_const - truncf(xi * ndx_const);

    // Perform linear interpolation and assign the result
    d_yout[j] = -(a - 1.f) * dk + a * dkp1;
  }
}
```

NNLinearInterpolation1DPyCUDA_18 hosted with ❤ by GitHub view raw

In addition to before, it is necessary to work out a texture object declaration `d_yintexture` for 1D float data, namely

```
texture<float, 1> d_yintexture;
```

NNLinearInterpolation1DPyCUDA_19 hosted with ❤ by GitHub view raw

Finally, the rows

```
float dk = tex1Dfetch(d_yintexture, fxi);
float dkp1 = tex1Dfetch(d_yintexture, fxi + 1);
```

NNLinearInterpolation1DPyCUDA_20 hosted with ❤ by GitHub view raw replace the above used direct memory access with texture fetching. The values of the nearest input neighbors from the texture are fetched using the calculated indices.

Moreover, the function calling the compiled CUDA kernel needs to use the below additional rows:

```
# Get the texture reference for d_yintexture from the CUDA module
d_yintexture = mod.get_texref('d_yintexture')

# Bind the d_yin array to the d_yintexture texture reference
d_yin.bind_to_texref(d_yintexture)
```

NNLinearInterpolation1DPyCUDA_21 hosted with ❤ by GitHub view raw

The first instruction retrieves the texture reference object associated with the texture named `d_yintexture` from the CUDA module (`mod`). The second, binds the CUDA array `d_yin` to the texture reference `d_yintexture` . This establishes the association between the texture reference and the specific CUDA array, allowing subsequent texture fetching in CUDA kernels to use the data from `d_yin` .

Linear GPU interpolation: texture filtering

To conclude the roundup of approaches for linear interpolation, we now present the case of employing linear texture filtering.

The CUDA kernel is the following:

```
// Define a texture object for 1D float data with linear filtering
texture<float, 1> d_yintexture_filtering;

// CUDA kernel for linear interpolation using texture filtering
__global__ void linear_interpolation_kernel_function_GPU_texture_filtering(const float*

  // Global thread identifier
  int j = threadIdx.x + blockDim.x * blockIdx.x;

  // Ensure each thread works on a valid output point
  if (j < N) {

    // Calculate the position relative to the first input position and apply scaling
    float xi = (d_xout[j] - d_xin[0]) * ndx_const;

    // Perform texture filtering for linear interpolation and assign the result
    d_yout[j] = tex1D(d_yintexture_filtering, (float)xi + 0.5f);
  }
}
```

NNLinearInterpolation1DPyCUDA_22 hosted with ❤ by **GitHub** **view raw**

The first line:

```
texture<float, 1> d_yintexture_filtering;
```

NNLinearInterpolation1DPyCUDA_23 hosted with ❤ by **GitHub** **view raw**

declares a texture object named `d_yintexture_filtering` for 1D float data with linear filtering.

The direct memory access with texture fetching used before is changed to texture filtering.

```
d_yout[j] = tex1D(d_yintexture_filtering, (float)xi + 0.5f);
```

NNLinearInterpolation1DPyCUDA_24 hosted with ❤ by **GitHub** **view raw**

The `(float)xi + 0.5f` part applies a half-pixel offset, which is used to achieve the right alignment in texture filtering.

```
# Set texture flags to read as integers (needed for linear filtering)
d_yintexture_filtering.set_flags(cuda.TRSF_READ_AS_INTEGER)

# Set texture filter mode to linear
d_yintexture_filtering.set_filter_mode(cuda.filter_mode.LINEAR)
```

NNLinearInterpolation1DPyCUDA_25 hosted with ❤ by GitHub view raw

The two above lines appear in the python function calling the CUDA kernel. In particular, the first line sets texture flags to read data as integers. This is necessary when using linear filtering to indicate that the texture contains data at integer locations. Moreover, the second line sets the texture filter mode to be linear. This enables linear interpolation between texels during texture fetching.

Nearest-neighbor GPU interpolation: texture filtering

To perform a nearest-neighbor interpolation, instead of a linear one, using texture filtering, it is enough to employ the same last CUDA kernel exploited for linear texture filtering, changing the set up instructions to

```
d_yintexture_filtering.set_flags(cuda.TRSF_READ_AS_INTEGER)
d_yintexture_filtering.set_filter_mode(cuda.filter_mode.POINT)
```

NNLinearInterpolation1DPyCUDA_26 hosted with ❤ by GitHub view raw

Interpolation Pycuda Cuda Nearest Neighbors Linear Interpolation

Follow

Written by Vitality Learning

74 Followers

We are teaching, researching and consulting parallel programming on Graphics Processing Units (GPUs) since the delivery of CUDA. We also play Matlab and Python.

More from Vitality Learning

Vitality Learning in CodeX

Understanding the architecture of a GPU

Recently, in the story The evolution of a GPU: from gaming to computing, the hystorical evolution of CPUs and GPUs has been discussed and...

11 min read · Mar 25, 2021

102

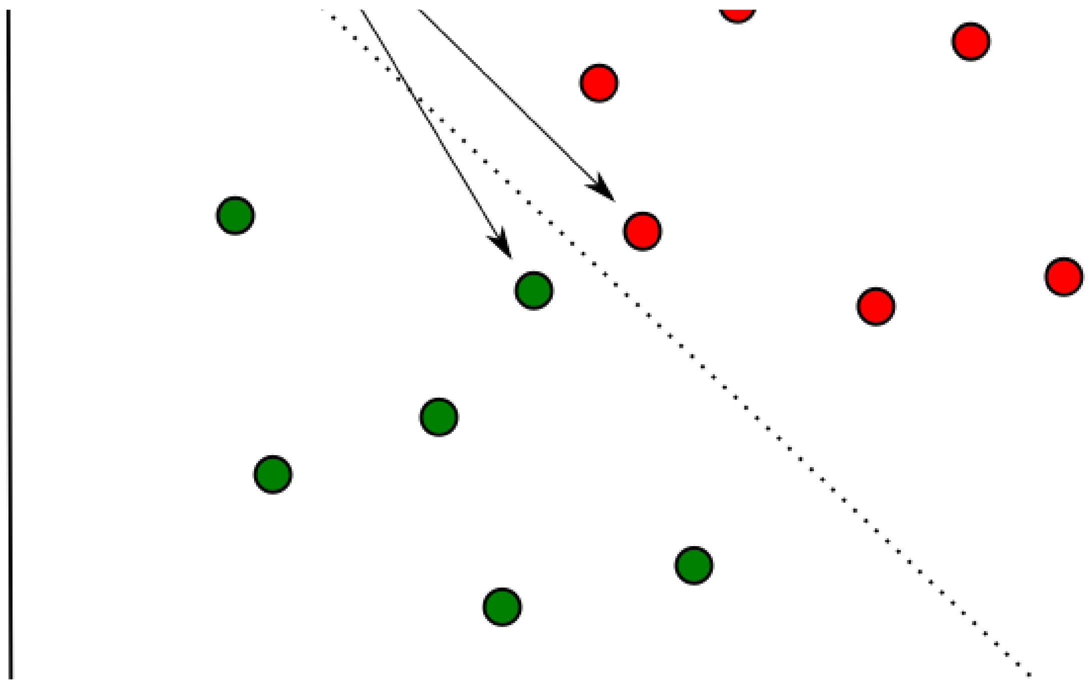
Vitality Learning

Running CUDA in Google Colab

Before reading the lines below, immediately change the runtime to use the GPU as hardware accelerator. This step is easily forgotten.

3 min read · Mar 4, 2021

21 2



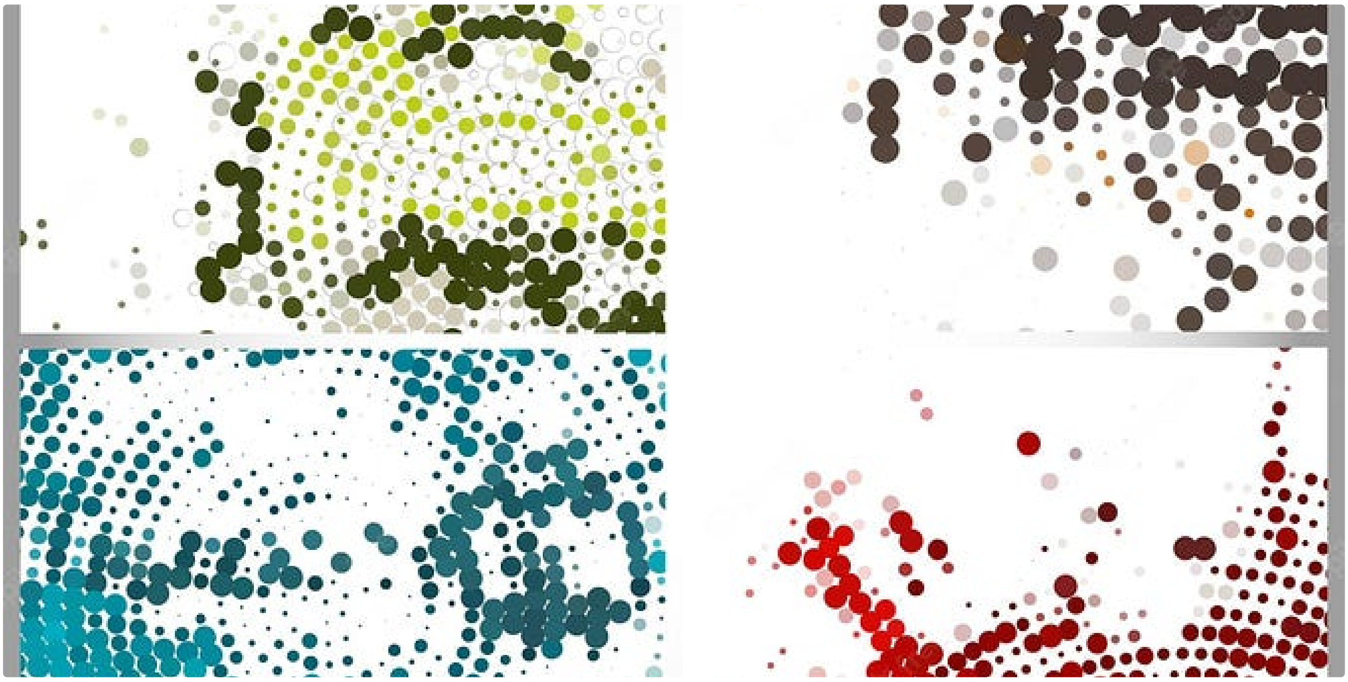
Vitality Learning

Support Vector Machines with TensorFlow

The Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for both classification and regression...

7 min read · Oct 8, 2022

12



Vitality Learning

K-Means algorithm in TensorFlow

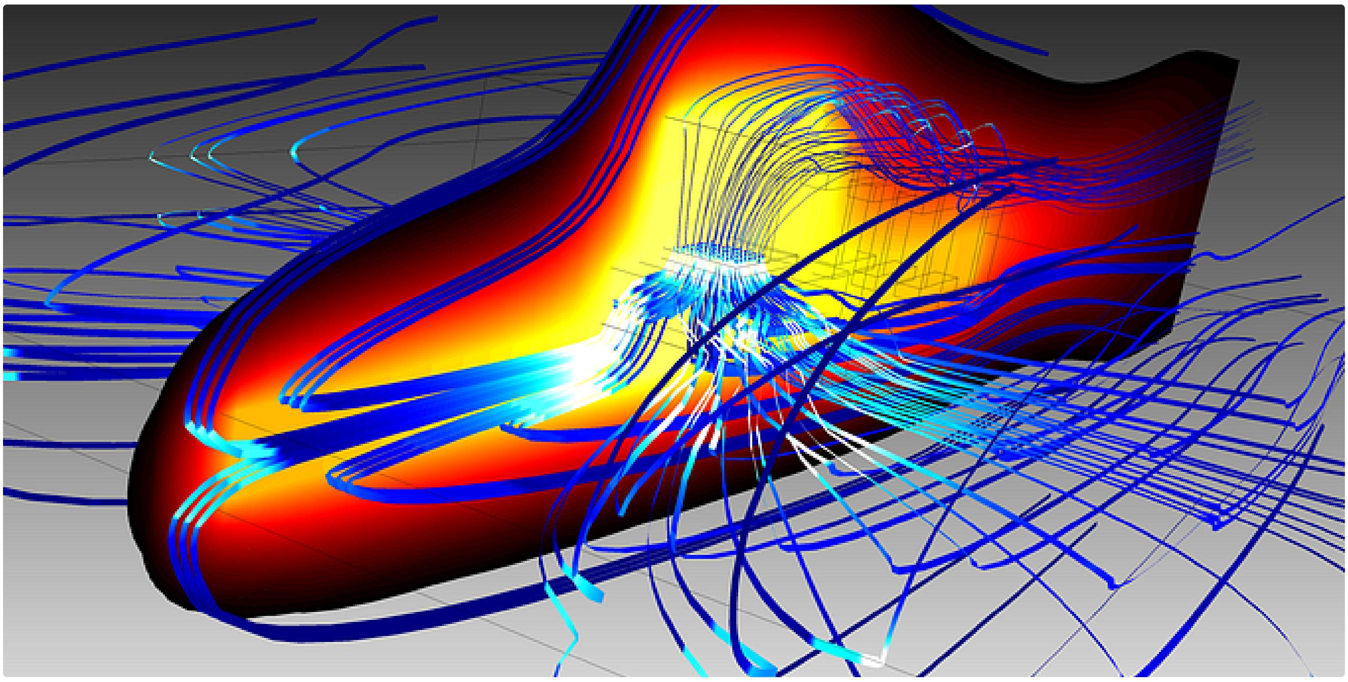
In everyday life, we often group objects according to a certain similarity, from clothes in the closet to food on the shelves in the...

7 min read · Sep 26, 2022

4

See all from Vitality Learning

Recommended from Medium



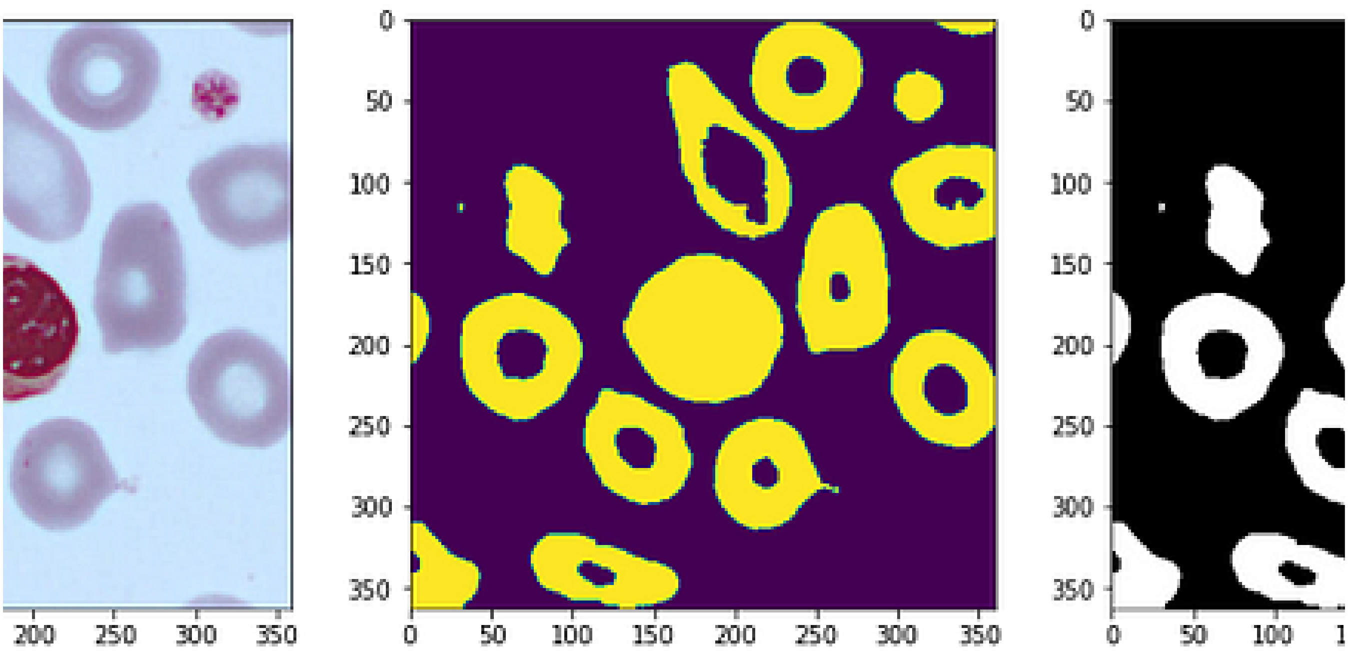
Antonio Pappaterra

Python in Action: Simulating Fluid Dynamics and Structural Analysis with CFD and FEA

In the ever-evolving landscape of scientific and engineering simulations, Python has emerged as a powerhouse for researchers and engineers...

3 min read · Oct 18

10



Sasani S. Perera

OpenCV: Object Masking

Object masking is very important in Image processing. This article discusses the theories behind object masking and a basic demonstration.

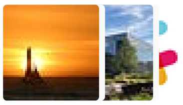
6 min read · Jul 13

10

Lists

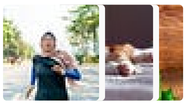
Staff Picks

533 stories · 522 saves



Stories to Help You Level-Up at Work

19 stories · 361 saves



Self-Improvement 101

20 stories · 1031 saves

Productivity 101

20 stories · 937 saves

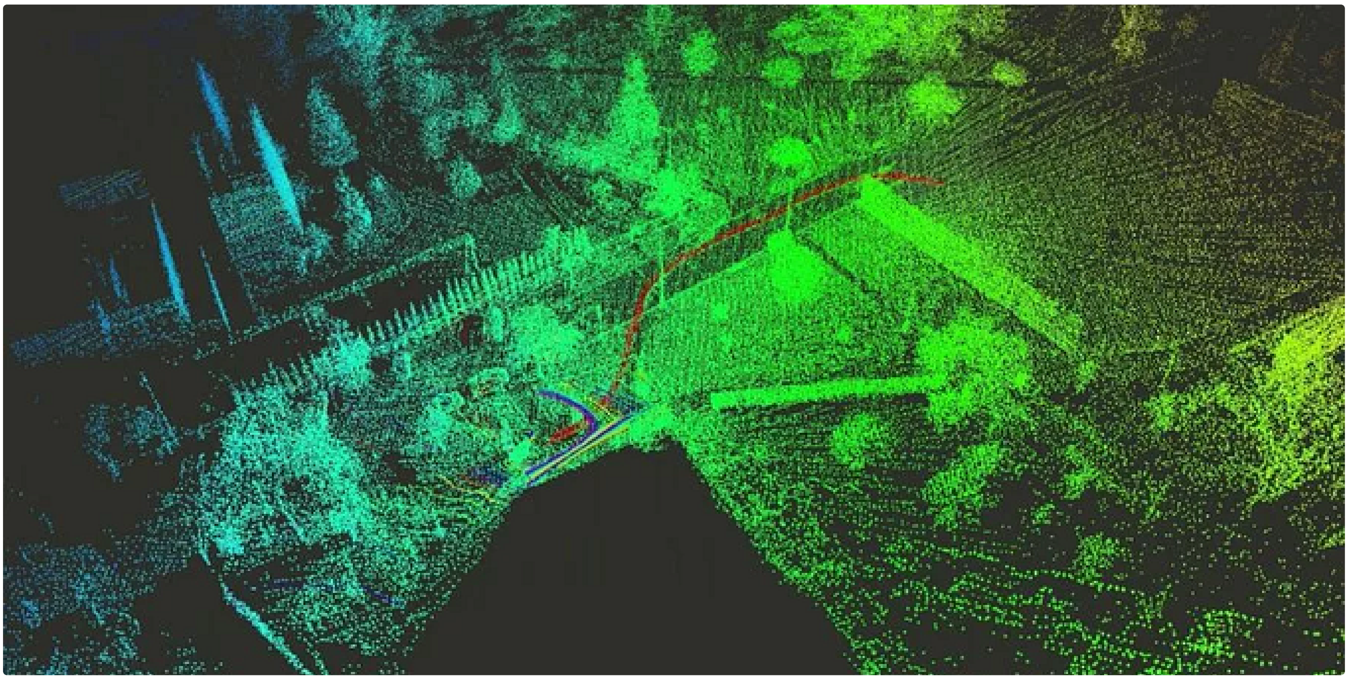
Nikhil Nair

2D Feature Tracking — Part 3: Feature Matching

In the first two parts of this series, we learned about detecting keypoints in an image, which are significant landmarks, and describing...

9 min read · Sep 17

9



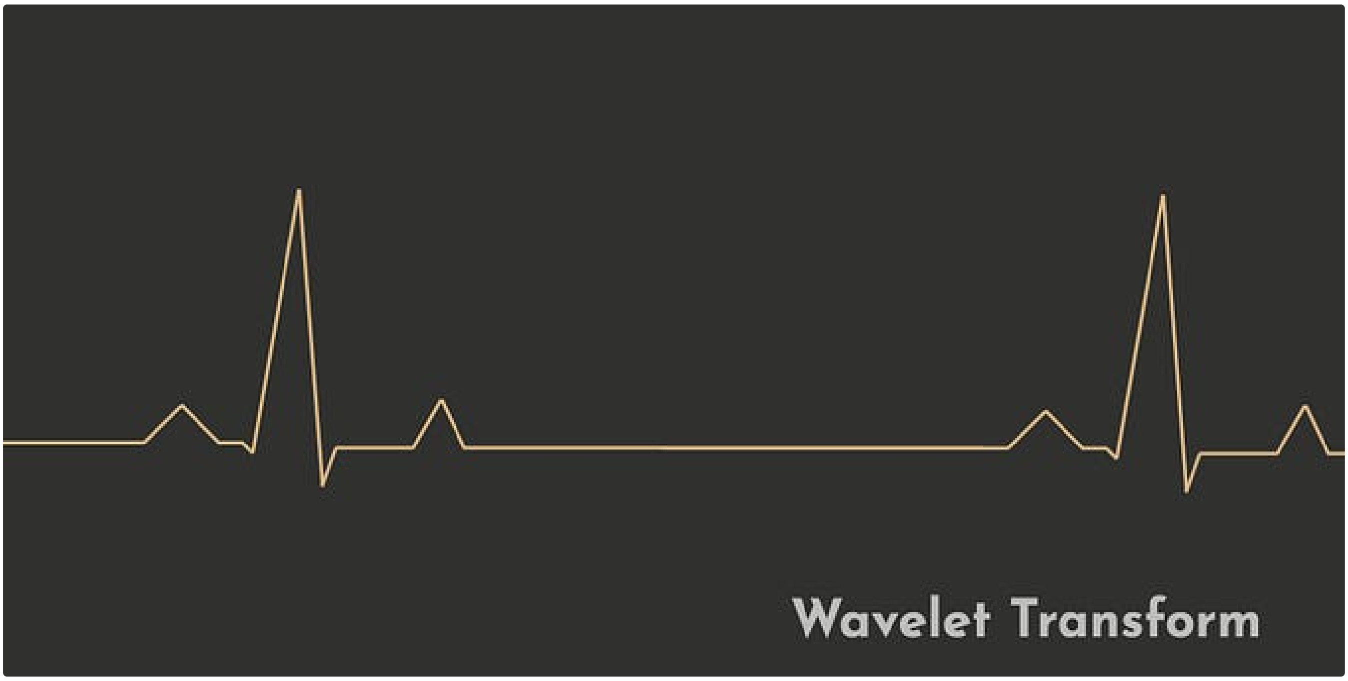
Panagiotis Liampas

Using SLAM to Autonomously Navigate in Unknown Environments

Robots can be deployed in an unknown environment, navigate on it, map it, and perform tasks autonomously. But, how do they do that?

15 min read · Oct 15

37



Shawhin Talebi in Towards Data Science

The Wavelet Transform

An Introduction and Example

· 6 min read · Dec 21, 2020

525 5

Uditha Atukorala

Frustrations of creating a gRPC server in C++

Despite the gRPC GitHub repo indicating over 65% of the code is C++, it's so frustrating to use C++ to create a gRPC server.

5 min read · Aug 7

13

See more recommendations