CMSC 425: Lecture 14
Procedural Generation: Perlin Noise

Reading: The material on Perlin Noise based in part by the notes *Perlin Noise*, by Hugo Elias. (The link to his materials seems to have been lost.) This is not exactly the same as Perlin noise, but the principles are the same.

Procedural Generation: Big game companies can hire armies of artists to create the immense content that make up the game's virtual world. If you are designing a game without such extensive resources, an attractive alternative for certain natural phenomena (such as terrains, trees, and atmospheric effects) is through the use of *procedural generation*. With the aid of a random number generator, a high quality procedural generation system can produce remarkably realistic models. Examples of such systems include *terragen* (see Fig. 1(a)) and *speedtree* (see Fig. 1(b)).

terragen speedtree

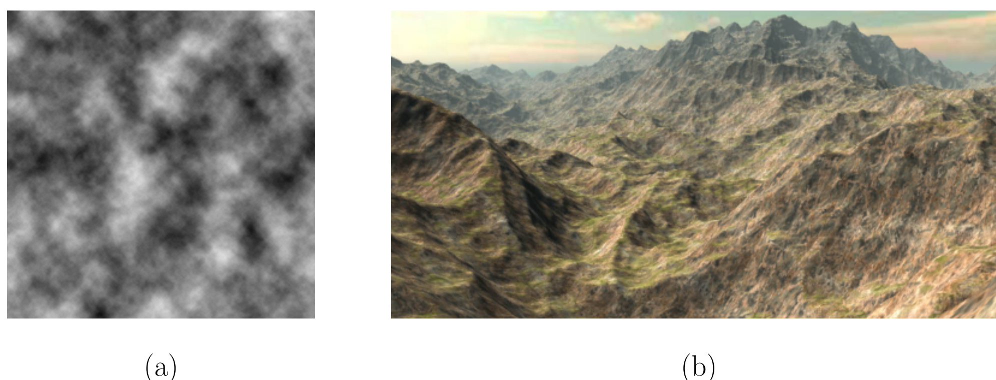
(a) (b)

Fig. 1: (a) A terrain generated by *terragen* and (b) a scene with trees generated by *speedtree*.

Before discussing methods for generating such interesting structures, we need to begin with a background, which is interesting in its own right. The question is how to construct random noise that has nice structural properties. In the 1980's, Ken Perlin came up with a powerful and general method for doing this (for which he won an Academy Award!). The technique is now widely referred to as Perlin Noise (see Fig. 2(a)). A terrain resulting from applying this is shown in Fig. 2(b). (The terragen software uses randomized methods like Perlin noise as a starting point for generating a terrain and then employs additional operations such as simulated erosion, to achieve heightened realism.)

Perlin Noise: Natural phenomena derive their richness from random variations. In computer science, pseudo-random number generators are used to produce number sequences that appear to be random. These sequences are designed to behave in a totally random manner, so that it is virtually impossible to predict the next value based on the sequence of preceding values. Nature, however, does not work this way. While there are variations, for example, in the elevations of a mountain or the curves in a river, there is also a great deal of structure present as well.

One of the key elements to the variations we see in natural phenomena is that the magnitude of random variations depends on the scale (or size) at which we perceive these phenomena.



(a) (b)

Fig. 2: (a) Two-dimensional Perlin noise and (b) a terrain generated by Perlin noise.

Consider, for example, the textures shown in Fig. 3. By varying the frequency of the noise we can obtain significantly different textures.

Fig. 3: Perlin noise used to generate a variety of displacement textures.

The tendency to see repeating patterns arising at different scales is called *self similarity* and it is fundamental to many phenomena in science and nature. Such structures are studied in mathematics under the name of *fractals*, as we have seen in an earlier lecture. Perlin noise can be viewed as a type of random noise that is self similar at different scales, and hence it is one way of modeling random fractal objects.

Noise Functions: Let us begin with the following question:

> *How do you convert the output of a pseudo-random number generator into a smooth (but random looking) function?*

To start, let us consider a sequence of random numbers in the interval $[0, 1]$ produced by a random number generator (see Fig. 4(a)). Let $Y = \langle y_0, \ldots, y_n \rangle$ denote the sequence of random values, and let us plot them at the uniformly places points $X = \langle 0, \ldots, n \rangle$.

Next, let us map these points to a continuous function, we could apply linear interpolation between pairs of points (also called *piecewise linear interpolation.* As we have seen earlier this semester, in order to interpolate linearly between two values y_i and y_{i+1}, we define a parameter α that varies between 0 and 1, the interpolated value is

$$\text{lerp}(y_i, y_{i+1}, \alpha) \;=\; (1 - \alpha)y_i + \alpha y_{i+1}.$$

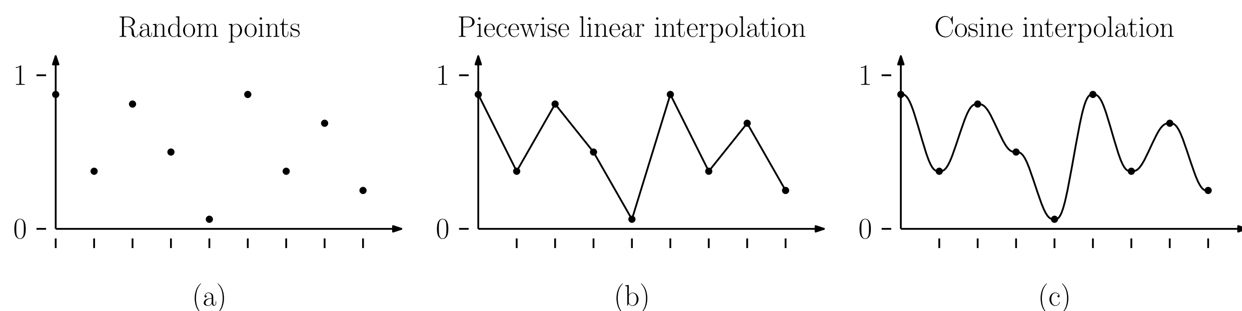


Fig. 4: (a) Random points, (b) connected by linear interpolation, and (c) connected by cosine interpolation.

To make this work in a piecewise setting we need to set α to the fractional part of the x-value that lies between i and $i+1$. In particular, if we define $x \bmod 1 = x - \lfloor x \rfloor$ to be the fractional part of x, we can define the linear interpolation function to be

$$f_\ell(x) \;=\; \mathrm{lerp}(y_i, y_{i+1}, \alpha), \qquad \text{where } i = \lfloor x \rfloor \text{ and } \alpha = x \bmod 1.$$

The result is the function shown in Fig. 4(b).

While linear interpolation is easy to define, it will not be sufficient smooth for our purposes. There are a number of ways in which to define smoother interpolating functions. (This is a topic that is usually covered in computer graphics courses.) A quick-and-dirty way to define such an interpolation is to replace the linear blending functions $(1-\alpha)$ and α in linear interpolation with smoother functions that have similar properties. In particular, observe that α varies from 0 to 1, the function $1-\alpha$ varies from 1 down to 0 while α goes the other way, and for any value of α these two functions sum to 1 (see Fig. 5(a)). Observe that the functions $(\cos(\pi\alpha)+1)/2$ and $(1-\cos(\pi\alpha))/2$ behave in exactly this same way (see Fig. 5(b)). Thus, we can use them as a basis for an interpolation method.

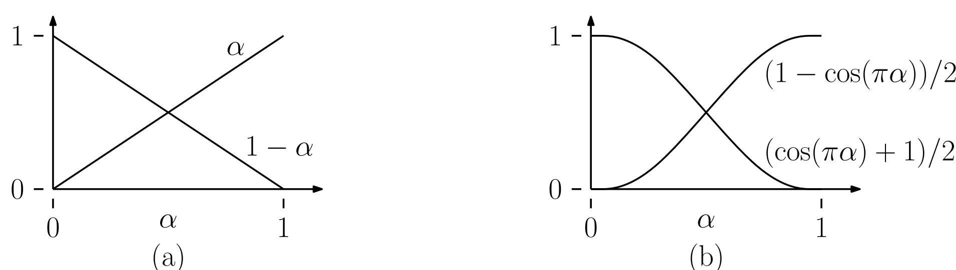


Fig. 5: The blending functions used for (a) linear interpolation and (b) cosine interpolation.

Define $g(\alpha) = (1-\cos(\pi\alpha))/2$. The cosine interpolation between two points y_i and y_{i+1} is defined:

$$\mathrm{cerp}(y_i, y_{i+1}, \alpha) \;=\; (1-g(\alpha))y_i + g(\alpha)y_{i+1},$$

and we can extend this to a sequence of points as

$$f_c(x) \;=\; \mathrm{cerp}(y_i, y_{i+1}, \alpha), \qquad \text{where } i = \lfloor x \rfloor \text{ and } \alpha = x \bmod 1.$$

The result is shown in Fig. 4(c). While cosine interpolation does not generally produce very good looking results when interpolating general data sets. (Notice for example the rather artificial looking flat spot as we pass through the fourth point of the sequence.) Interpolation methods such as cubic interpolation and Hermite interpolate are preferred. It is worth remembering, however, that we are interpolating random noise, so the lack of "goodness" here is not much of an issue.

Layering Noise: Our noise function is continuous, but there is no self-similarity to its structure. To achieve this, we will need to combine the noise function in various ways. Our approach will be similar to the approach used in the harmonic analysis of functions.

Recall that when we have a periodic function, like $\sin t$ or $\cos t$, we define (see Fig. 6)

Wavelength: The distance between successive wave crests

Frequency: The number of crests per unit distance, that is, the reciprocal of the wavelength

Amplitude: The height of the crests

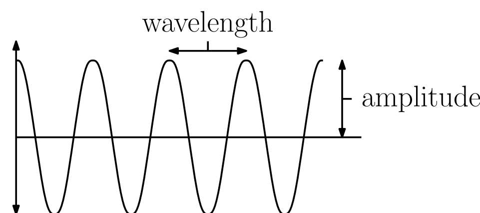


Fig. 6: Properties of periodic functions.

If we want to decrease the wavelength (equivalently increase the frequency) we can scale up the argument. For example $\sin t$ has a wavelength of 2π, $\sin(2t)$ has a wavelength of π, and $\sin(4t)$ has a wavelength of $\pi/2$. (By increasing the value of the argument we are increasing the function's frequency, which decreases the wavelength.) To decrease the function's amplitude, we apply a scale factor that is smaller than 1 to the value of the function. Thus, for any positive reals ω and α, the function $\alpha \cdot \sin(\omega t)$ has a wavelength of $2\pi/\omega$ and an amplitude of α.

Now, let's consider doing this to our noise function. Let $f(x)$ be the noise function as defined in the previous section. Let us assume that $0 \leq x \leq n$ and that the function repeats so that $f(0) = f(n)$ and let us assume further that the derivatives match at $x = 0$ and $x = n$. We can convert f into a periodic function for all $t \in \mathbb{R}$, which we call $\text{noise}(t)$, by defining

$$\text{noise}(t) \;=\; f(t \bmod n).$$

(Again we are using the mod function in the context of real numbers. Formally, we define $x \bmod n = x - n \cdot \lfloor x/n \rfloor$.) For example, the top graph of Fig. 7 shows three wavelengths of $\text{noise}(t)$.

In order to achieve self-similarity, we will sum together this noise function, but using different frequencies and with different amplitudes. First, we will consider the noise function with exponentially increasing frequencies: $\text{noise}(t)$, $\text{noise}(2t)$, $\text{noise}(4t)$, $\ldots$, $\text{noise}(2^i t)$ (see Fig. 8).

Note that we have not changed the underlying function, we have merely modified its frequency. In the jargon of Perlin noise, these are called *octaves*, because like musical octaves, the frequency doubles.[1] Because frequencies double with each octave, you do not need very many octaves, because there is nothing to be gained by considering wavelengths that are larger than the entire screen nor smaller than a single pixel. Thus, the logarithm of the window size is a natural upper bound on the number of octaves.

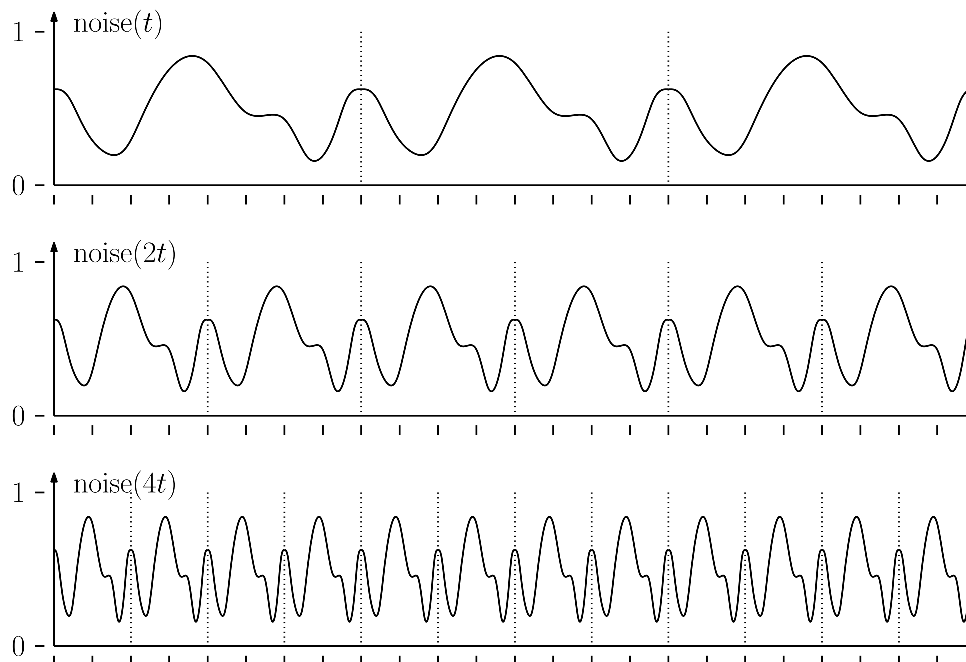


Fig. 7: The periodic noise function at various frequencies.

High frequency noise tends to be of lower amplitude. If we were in a purely self-similar situation, when the double the frequency, we should halve the amplitude. In order to provide the designer with more control, Perlin noise allows the designer to specify a separate amplitude for each frequency. A common way in which to do this is to define a parameter, called *persistence*, that specifies how rapidly the amplitudes decrease. Persistence is a number between 0 and 1. The larger the persistence value, the more noticeable are the higher frequency components. (That is, the more "jagged" the noise appears.) In particular, given a persistence of p, we define the amplitude at the ith stage to be p^i. The final noise value is the sum, over all the octaves, of the persistence-scaled noise functions. In summary, we have

$$\mathrm{perlin}(t) \;=\; \sum_{i=0}^{k} p^i \cdot \mathrm{noise}(2^i \cdot t),$$

where k is the highest-frequency octave.

It is possible to achieve greater control over the process by allowing the user to modify the octave scaling values (currently 2^i) and the persistence values (currently p^i).

[1] In general, it is possible to use factors other than 2. Such a factor is called the *lacunarity* of the Perlin noise function. For example, a lacunarity value of ℓ means that the frequency at stage i will be ℓ^i.

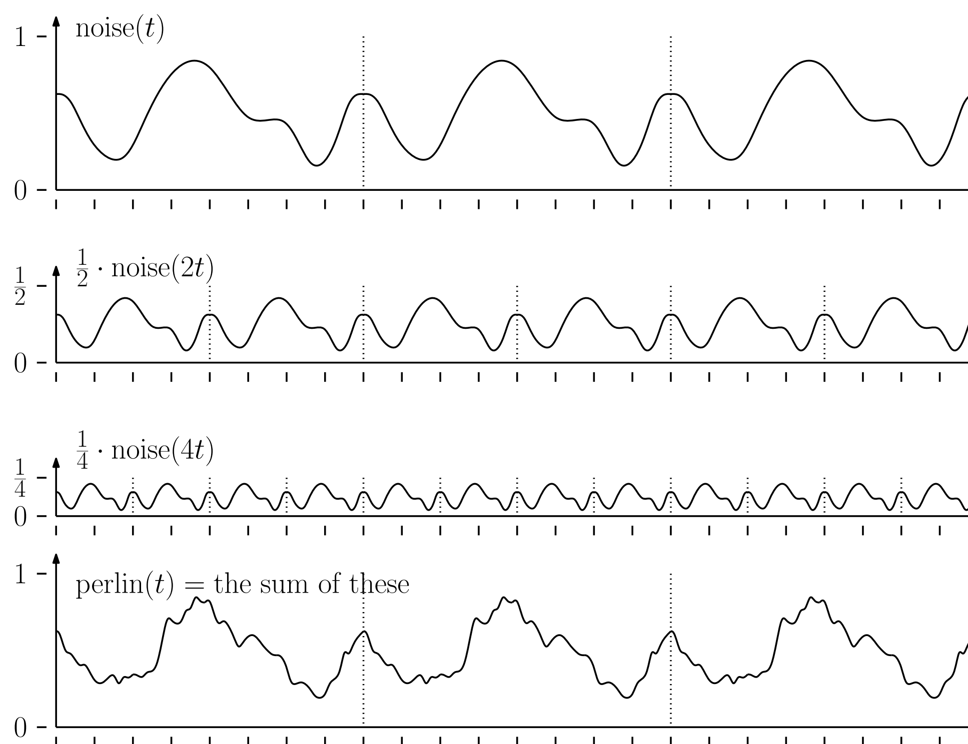


Fig. 8: Dampened noise functions and the Perlin noise function (with persistence $p = 1/2$).

Perlin Noise in 2D: (The material from here until the end of the lecture is optional)

In the previous lecture we introduced the concept of Perlin noise, a structured random function, in a one-dimensional setting. In this lecture we show how to generalize this concept to a two-dimensional setting. Such two-dimensional noise functions can be used for generating pseudo-random terrains and two-dimensional pseudo-random textures.

The general approach is the same as in the one-dimensional case:

- Generate a finite sample of random values
- Generate a noise function that interpolates smoothly between these values
- Sum together various octaves of this function by scaling it down by factors of 1/2, and then applying a dampening persistence value to each successive octave, so that high frequency variations are diminished

Let's investigate the finer points. First, let us begin by assuming that we have an $n \times n$ grid of unit squares (see Fig. 9(a)), for a relatively small number n (e.g., n might range from 2 to 10). For each vertex $[i, j]$ of this grid, where $0 \leq i, j \leq n$, let us generate a random scalar value $z_{[i,j]}$. (Note that these values are actually not very important. In Perlin's implementation of the noise function, these value are all set to 0, and it still produces a remarkably rich looking noise function.) As in the 1-dimensional case, it is convenient to have the values wrap around, which we can achieve by setting $z_{[i,n]} = z_{[i,0]}$ and $z_{[n,j]} = z_{[0,j]}$ for all i and j.

Given any point (x, y), where $0 \leq x, y < n$, the corner points of the square containing this point are (x_0, y_0), (x_1, y_0), (x_1, y_1), (x_0, y_1) where:

$$x_0 = \lfloor x \rfloor \quad \text{and} \quad x_1 = (x_0 + 1) \bmod n$$
$$y_0 = \lfloor y \rfloor \quad \text{and} \quad y_1 = (y_0 + 1) \bmod n$$

(see Fig. 9(a)).

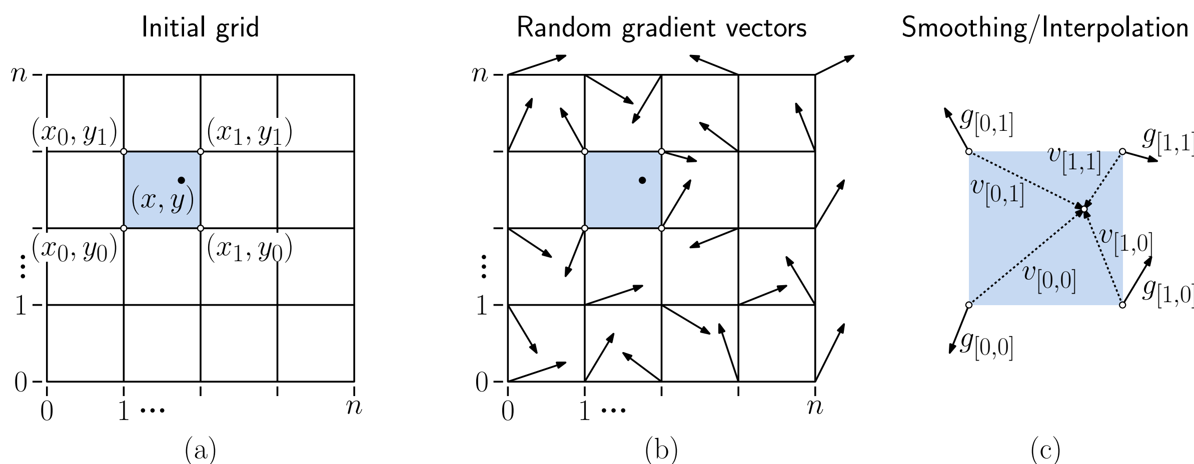


Fig. 9: Generating 2-dimensional Perlin noise.

We could simply apply smoothing to the random values at the grid points, but this would produce a result that clearly had a rectangular blocky look (since every square would suffer the same variation). Instead, Perlin came up with a way to have every vertex behave differently, by creating a random gradient at each vertex of the grid.

Noise from Random Gradients: Before explaining the concept of gradients, let's recall some basics from differential calculus. Given a continuous function $f(x)$ of a single variable x, we know that the derivative of the function df/dx yields the tangent slope at the point $(x, f(x))$ on the function. If we instead consider a function $f(x, y)$ of two variables, we can visualize the function values $(x, y, f(x, y))$ as defining the height of a point on a two-dimensional terrain. If f is smooth, then each point of the terrain can be associated with tangent plane. The "slope" of the tangent plane passing through such a point is defined by the partial derivatives of the function, namely $\partial f/\partial x$ and $\partial f/\partial y$. The vector $(\partial f/\partial x, \partial f/\partial y)$ is a vector in the (x, y)-plane that points in the direction of *steepest ascent* for the function f. This vector changes from point to point, depending on f. It is called the *gradient* of f, and is often denoted ∇f.

Perlin's approach to producing a noisy 2-dimensional terrain involves computing a random 2-dimensional gradient vector at each vertex of the grid with the eventual aim that the smoothed noise function have this gradient value. Since these vectors are random, the resulting noisy terrain will appear to behave very differently from one vertex of the grid to the next. At one vertex the terrain may be sloping up to the northeast, and at a neighboring vertex it may be sloping to south-southwest. The random variations in slope result in a very complex terrain. But how do we define a smooth function that has this behavior? In the one dimensional case we used cosine interpolation. Let's see how to generalize this to a two-dimensional setting.

Consider a single square of the grid, with corners (x_0, y_0), (x_1, y_0), (x_1, y_1), (x_0, y_1). Let $g_{[0,0]}$, $g_{[1,0]}$, $g_{[1,1]}$, and $g_{[0,1]}$ denote the corresponding randomly generated 2-dimensional gradient vectors (see Fig. 9(c)). Now, for each point (x, y) in the interior of this grid square, we need to blend the effects of the gradients at the corners. To do this, for each corner we will compute a vector from the corner to the point (x, y). In particular, define

$$
\begin{aligned}
v_{[0,0]} &= (x, y) - (x_0, y_0) \quad \text{and} \quad v_{[0,1]} = (x, y) - (x_0, y_1) \\
v_{[1,0]} &= (x, y) - (x_1, y_0) \quad \text{and} \quad v_{[1,1]} = (x, y) - (x_1, y_1)
\end{aligned}
$$

(see Fig. 9(c)).

Next, for each corner point of the square, we generate an associated *vertical displacement*, which indicates the height of the point (x, y) due to the effect of the gradient at this corner point. How should this displacement be defined? Let's fix a corner, say $(x, 0, y_0)$. Intuitively, if $v_{[0,0]}$ is directed in the same direction as the gradient vector, then the vertical displacement will increase (since we are going uphill). If it is in the opposite direction, the displacement will decrease (since we are going downhill). If the two vectors are orthogonal, then the vector $v_{[0,0]}$ is directed neither up- or downhill, and so the displacement is zero. Among the vector operations we have studied, the *dot product* produces exactly this sort of behavior. (When two vectors are aligned, the dot-product is maximized, when they are anti-aligned it is minimized, and it is zero when they are orthogonal. It also scales linearly with the length, so that a point that is twice as far away along a given direction has twice the displacement.) With this in mind, let us define the following scalar displacement values:

$$
\begin{aligned}
\delta_{[0,0]} &= (v_{[0,0]} \cdot g_{[0,0]}) \quad \text{and} \quad \delta_{[0,1]} = (v_{[0,1]} \cdot g_{[0,1]}) \\
\delta_{[1,0]} &= (v_{[1,0]} \cdot g_{[1,0]}) \quad \text{and} \quad \delta_{[1,1]} = (v_{[1,1]} \cdot g_{[1,1]}).
\end{aligned}
$$

Fading: The problem with these scalar displacement values is that they are affected by all the corners of the square, and in fact, as we get farther from the associated corner point the displacement gets larger. We want the gradient effect to apply close to the vertex, and then have it drop off quickly as we get closer to another vertex. That is, we want the gradient effect of this vertex to *fade* as we get farther from the vertex. To do this, Perlin defines the following *fade function*. This is a function of t that will start at 0 when $t = 0$ (no fading) and will approach 1 when $t = 1$ (full fading). Perlin originally settled on a cubic function to do this, $\varphi(t) = 3t^2 - 2t^3$. (Notice that this has the desired properties, and further its derivative is zero at $t = 0$ and $t = 1$, so it will smoothly interpolate with neighboring squares.) Later, Perlin observed that this function has nonzero second derivatives at 0 and 1, and so he settled on the following improved fade function:

$$
\psi(t) = 6t^5 - 15t^4 + 10t^3
$$

(see Fig. 10). Observe again that $\psi(0) = 0$ and $\psi(1) = 1$, and the first and second derivatives are both zero at these endpoints.

Because we want the effects to fade as a function of both x and y, we define the *joint fade function* to be the product of the fade functions along x and y:

$$
\Psi(x, y) = \psi(x)\psi(y).
$$

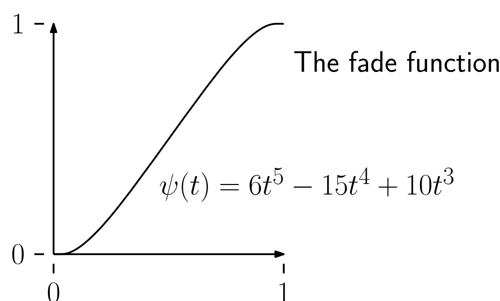


Fig. 10: The fade function.

The final noise value at the point (x, y), arises by taking the weighted average of gradient displacements, where each displacement is weighted according to the fade function.

We need to apply the joint fade function differently for each vertex. For example, consider the fading for the displacement $\delta_{[1,0]}$ of the lower right corner vertex. We want the influence of this vertex to increase as x approaches 1, which will be achieved by using a weight of $\psi(x)$. Similarly, we want the influence of this vertex to increase as y approaches 0, which will be achieved by using a weight of $\psi(1-y)$. Therefore, to achieve both of these effects, we will use the joint weight function $\Psi(x, 1-y)$. By applying this reasoning to the other corner vertices, we obtain the following 2-dimensional noise function.

$$\text{noise}(x, y) \;=\; \Psi(1-x, 1-y)\delta_{[0,0]} + \Psi(x, 1-y)\delta_{[1,0]} + \Psi(1-x, y)\delta_{[0,1]} + \Psi(x, y)\delta_{[1,1]}.$$

Adding Back the Random Heights: We have left one little bit out of our noise function. Remember that we started off by assigning random scalar values to each of the of the grid. We never made use of these (and indeed, Perlin's formulation of the noise function does not either). In order to achieve this extra degree of randomness, we can add these back into the vertical displacements. Suppose, for example that we are considering the grid square whose lower left corner has the indices $[i, j]$. When defining the vertical displacements, let us add in the associated random scalar values associated with each of the vertices:

$$\begin{aligned} \delta_{[0,0]} &= z_{[i,j]} + (v_{[0,0]} \cdot g_{[0,0]}) \quad \text{and} \quad & \delta_{[0,1]} &= z_{[i,j+1]} + (v_{[0,1]} \cdot g_{[0,1]}) \\ \delta_{[1,0]} &= z_{[i+1,j]} + (v_{[1,0]} \cdot g_{[1,0]}) \quad \text{and} \quad & \delta_{[1,1]} &= z_{[i+1,j+1]} + (v_{[1,1]} \cdot g_{[1,1]}). \end{aligned}$$

The rest of the noise computation is exactly the same as described above.

Octaves and Persistence: After all of that work, we still have only a single smooth noise function, but not one that demonstrates the sort of fractal-like properties we desire. To add this, we need to perform the same scaling that we used for the 1-dimensional case. In this case, the process is entirely analogous. As before, let p be a value between 0 and 1, which will determine how quickly things dampen down. Also, as before, at each level of the process, we will double the frequency. This leads to the following final definition of the 2-dimensional Perlin noise:

$$\text{perlin}(x, y) \;=\; \sum_{i=0}^{k} p^i \cdot \text{noise}(2^i \cdot x, 2^i \cdot y).$$

(As before, recall that the value 2^i can be replaced by some parameter ℓ^i, where $\ell > 1$.) This applies to each square individually. We need to perform the usual "modding" to generalize this to any square of the grid. (An example of the final result is shown in Fig. 2(a).)

Source Code: While the mathematical concepts that we have discussed are quite involved, it is remarkable that Perlin noise has a very simple implementation. The entire implementation can be obtained from Perlin's web page, and is shown nearly in its entirety in the code block below.

Perlin's Implementation of Perlin Noise

```
public final class ImprovedNoise {
   static public double noise(double x, double y, double z) {
      int X = (int)Math.floor(x) & 255,           // Fine unit cube that
          Y = (int)Math.floor(y) & 255,           //   contains point
          Z = (int)Math.floor(z) & 255;
      x -= Math.floor(x);                          // Find relative x,y,z
      y -= Math.floor(y);                          //   of point in cube
      z -= Math.floor(z);
      double u = fade(x),                          // Compute fade curves
             v = fade(y),                          //   for each of x,y,z
             w = fade(z);
      int A = p[X  ]+Y, AA = p[A]+Z, AB = p[A+1]+Z,// Hash coordinates of
          B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;//   the 8 cube corners
      // ...and add blended results from 8 corners of cube
      return lerp(w, lerp(v, lerp(u, grad(p[AA  ], x  , y  , z   ),
                                     grad(p[BA  ], x-1, y  , z   )),
                             lerp(u, grad(p[AB  ], x  , y-1, z   ),
                                     grad(p[BB  ], x-1, y-1, z   ))),
                     lerp(v, lerp(u, grad(p[AA+1], x  , y  , z-1 ),
                                     grad(p[BA+1], x-1, y  , z-1 )),
                             lerp(u, grad(p[AB+1], x  , y-1, z-1 ),
                                     grad(p[BB+1], x-1, y-1, z-1 ))));
   }
   static double fade(double t) { return t*t*t*(t*(t*6 - 15) + 10); }
   static double lerp(double t, double a, double b)
      { return a + t*(b - a); }
   static double grad(int hash, double x, double y, double z) {
      int h = hash & 15;                // Convert low 4 bits of hash code
      double u = h<8 ? x : y,           //   into 12 gradient directions
             v = h<4 ? y : h==12||h==14 ? x : z;
      return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
   }
   static final int p[] = new int[512], permutation[] = {
      151,160,137,91,90,15,  // ... remaining 506 entries omitted
   };
   static { for (int i=0; i < 256 ; i++)
      p[256+i] = p[i] = permutation[i]; }
}
```