**Simple Karras Exponential Scheduler**

The Simple Karras Exponential Scheduler dynamically blends Karras (smooth noise reduction) with Exponential (fast decay) methods during image generation, ensuring smooth transitions and better control.

Created by KittensX

Uploaded to: <https://github.com/Kittensx>

Last Updated:

10/27/2024

Table of Contents

[Simple Karras Exponential Scheduler (Simple KES) 4](#_Toc180913136)

[Key Features 4](#_Toc180913137)

[Installation 5](#_Toc180913138)

[License 5](#_Toc180913139)

[Configurable Parameters 5](#_Toc180913140)

[Dynamic Configuration Reloading with Watchdog 8](#_Toc180913141)

[What is Watchdog? 8](#_Toc180913142)

[How it Works in This Project 9](#_Toc180913143)

[Benefits of Watchdog in This Project 9](#_Toc180913144)

[Setting Up Watchdog 9](#_Toc180913145)

[Detailed Analysis of the Simple Karras Exponential Scheduler 9](#_Toc180913146)

[Defining Sigma Max and How it Affects Blending between Karras and Exponential 9](#_Toc180913147)

[What is it Doing? 10](#_Toc180913148)

[Theoretical Maximum and Minimum for the Multiplication Factor 10](#_Toc180913149)

[Purpose of Adjusting the Sigma Max 10](#_Toc180913150)

[Why Only a 10% Increase? 11](#_Toc180913151)

[Matching Sequence Lengths Explained 11](#_Toc180913152)

[Why is Having Matching Sequences Important? 11](#_Toc180913153)

[Matching sequence lengths 12](#_Toc180913154)

[Exception block 12](#_Toc180913155)

[Defining Progress and Sigs Variables 13](#_Toc180913156)

[Progress 13](#_Toc180913157)

[Sigs Variable 14](#_Toc180913158)

[Step Size, Blend Factor, and Noise Scale 15](#_Toc180913159)

[Step Size Calculation 15](#_Toc180913160)

[Dynamic Blend Factor Calculation 15](#_Toc180913161)

[Why Shouldn't These Values Change Much? 16](#_Toc180913162)

[Noise Scale Calculation 16](#_Toc180913163)

[Summary of Methods 16](#_Toc180913164)

[Smooth Blend and the Sigmoid Function 17](#_Toc180913165)

[Sigmoid Function 17](#_Toc180913166)

[Dynamic Blend Factor Adjustment 17](#_Toc180913167)

[Scaling the Input 18](#_Toc180913168)

[Purpose of the Sigmoid and Scaling 18](#_Toc180913169)

[Why Use the Sigmoid for Blending? 19](#_Toc180913170)

[Why Not Use Larger/Smaller Scaling Factors? 19](#_Toc180913171)

[Dynamic Blend Factor Adjustment Summary 19](#_Toc180913172)

[Blending Karras and Exponential 20](#_Toc180913173)

[Blending Concept 20](#_Toc180913174)

[Formula 20](#_Toc180913175)

[Blending Behavior 21](#_Toc180913176)

[Why This Blending Matters 21](#_Toc180913177)

[Blending Behavior Summary 22](#_Toc180913178)

[Applying Step Size and Noise Scaling 22](#_Toc180913179)

[Blended Sigma 22](#_Toc180913180)

[Step Size 22](#_Toc180913181)

[Noise Scale 23](#_Toc180913182)

[Final Sigma Value for Step i 23](#_Toc180913183)

[The Multiplication Process 23](#_Toc180913184)

[Why Apply Both Step Size and Noise Scale? 23](#_Toc180913185)

[Why Not Just Use Blended Sigma Alone? 24](#_Toc180913186)

[Applying Step Size and Noise Scaling Summary 24](#_Toc180913187)

[Adaptive Sharpening 25](#_Toc180913188)

[What is Adaptive Sharpening? 25](#_Toc180913189)

[torch.where() Function 25](#_Toc180913190)

[Multiplying the Sigmas by the Sharpen Mask 26](#_Toc180913191)

[Why This Works for Adaptive Sharpening 26](#_Toc180913192)

[Why Use 1.5 as the Threshold? 27](#_Toc180913193)

[Adaptive Sharpening Summary 27](#_Toc180913194)

[Early Stopping 27](#_Toc180913195)

[Calculate the Change Between Consecutive Sigma Values 28](#_Toc180913196)

[Check If All Changes Are Below the Early Stop Threshold 28](#_Toc180913197)

[Logging and Early Return 29](#_Toc180913198)

[Why Use Early Stopping Based on Sigma Convergence? 29](#_Toc180913199)

[Early Stopping Summary 30](#_Toc180913200)

[Randomization of Parameters 30](#_Toc180913201)

[Randomize 30](#_Toc180913202)

[Configuration Options for Randomization 30](#_Toc180913203)

[Enabling Randomization 31](#_Toc180913204)

[Benefits of Randomization 31](#_Toc180913205)

[Graph References 31](#_Toc180913206)

[20 Steps 32](#_Toc180913207)

[40 Steps 32](#_Toc180913208)

[60 Steps 32](#_Toc180913209)

[80 Steps 33](#_Toc180913210)

[100 Steps 34](#_Toc180913211)

# Simple Karras Exponential Scheduler (Simple KES)

The Simple Karras Exponential Scheduler (Simple KES) is a dynamic image generation tool that combines Karras and Exponential noise scheduling methods to produce high-quality, detailed images with smooth transitions. This scheduler leverages the strengths of each method: Karras for smooth noise reduction and Exponential for faster noise decay. Through this blending approach, Simple KES offers enhanced control and adaptability, resulting in refined and versatile images suited for artistic and experimental applications.

Created by **KittensX**, Simple KES is designed to operate within the Automatic1111 framework, with support for GPU (CUDA) or CPU processing. This scheduler also integrates with **Watchdog**, a Python library that allows real-time reloading of configurations without needing to restart the program, offering on-the-go flexibility to experiment with different settings.

### Key Features

**Blended Noise Scheduling**: Utilizes a combination of Karras and Exponential methods for noise scheduling, allowing smooth yet flexible transitions tailored to user-defined image quality and style preferences.

**Dynamic Parameter Randomization**: Provides options for randomized parameter control, supporting creative exploration of various settings for diverse visual results.

**Adaptive Sharpening and Early Stopping**: Optimizes processing time and image detail by sharpening selectively based on sigma values and stopping early when further steps would not enhance the image.

**Real-Time Configuration Reloading**: Powered by Watchdog, this feature enables on-the-fly adjustments to configurations, making it ideal for workflows that require frequent experimentation and fine-tuning.

## Installation

* 1. Install the scheduler and configuration file inside the modules folder in Automatic1111.
  2. Ensure that the sd\_schedulers.py is updated to include this line inside schedulers[] and import it at the top (or replace the file as needed with the given file – updated as version 1.10)

import modules.simple\_karras\_exponential\_scheduler as simple\_kes

Scheduler('karras\_exponential', 'Karras Exponential', simple\_kes.simple\_karras\_exponential\_scheduler),

For version 1.10 you can replace the files found in “modules\” if preferred.

See the scheduler requirements. This file uses watchdog which monitors changes in files. So if you wanted to do a config change to the yaml file you could do so without needing to restart the program, since watchdog will monitor changes then trigger a reload of the config file. For more information, see “Watchdog”

To include this scheduler with other programs, you may need technical knowledge to include this scheduler.

## License

MIT License

## Configurable Parameters

Note: Default values are included in the scheduler, so the simple\_kes\_scheduler.yaml (config file) is optional. Any values not in the config file will use default values found in the python file. Using a config file allows the user more control over using the advanced features of the scheduler.

num\_steps: 100 # Suggested range: 50 – 150

*Number of steps to use in the image generation process.Higher values result in smoother images but take longer to process.Lower values are faster but may result in lower quality.*

sigma\_min: 0.01 # Suggested range: 0.01 - 0.1

The minimum value for the noise level (sigma) during image generation. Decreasing this value makes the image clearer but less detailed. Increasing it makes the image noisier but potentially more artistic or abstract.

sigma\_max: 0.1 # Suggested range: 0.05 - 0.2

The maximum value for the noise level (sigma) during image generation. Increasing this value can create more variation in the image details. Lower values keep the image more stable and less noisy.

device: "cuda" # Options: "cuda" (GPU) or "cpu" (processor)

The device used for running the scheduler. If you have a GPU, set this to "cuda". Otherwise, use "cpu", but note that it will be significantly slower.

start\_blend: 0.1 # Suggested range: 0.05 - 0.2

Initial blend factor between Karras and Exponential noise methods. A higher initial blend makes the image sharper at the start. A lower initial blend makes the image smoother early on.

end\_blend: 0.5 # Suggested range: 0.4 - 0.6

Final blend factor between Karras and Exponential noise methods. Higher values blend more noise at the end, possibly adding more detail. Lower values blend less noise for smoother, simpler images at the end.

sharpness: 0.95 # Suggested range: 0.8 - 1.0

Sharpening factor applied to images during generation. Higher values increase sharpness but can add unwanted artifacts. Lower values reduce sharpness but may make the image look blurry.

early\_stopping\_threshold: 0.01 # Suggested range: 0.005 - 0.02

Early stopping threshold for stopping the image generation when changes between steps are minimal. Lower values stop early, saving time, but might produce incomplete images. Higher values take longer but may give more detailed results.

update\_interval: 10 # Suggested range: 5 – 15

The number of steps between updates of the blend factor. Smaller values update the blend more frequently for smoother transitions. Larger values update the blend less frequently for faster processing.

initial\_step\_size: 0.9 # Suggested range: 0.5 - 1.0

Initial step size, which controls how quickly the image evolves early on. Higher values make big changes at the start, possibly generating faster but less refined images. Lower values make smaller changes, giving more control over details.

final\_step\_size: 0.2 # Suggested range: 0.1 - 0.3

Final step size, which controls how much the image changes towards the end. Higher values keep details more flexible until the end, which may add complexity. Lower values lock the details earlier, making the image simpler.

initial\_noise\_scale: 1.25 # Suggested range: 1.0 - 1.5

Initial noise scaling applied to the image generation process. Higher values add more noise early on, making the initial image more random. Lower values reduce noise early on, leading to a smoother initial image.

final\_noise\_scale: 0.8 # Suggested range: 0.6 - 1.0

Final noise scaling applied at the end of the image generation. Higher values add noise towards the end, possibly adding fine detail. Lower values reduce noise towards the end, making the final image smoother.

randomize: true

#If true, all parameters are randomized within specified min/max ranges.

#Parameter\_rand – where parameter is the name of the below parameters (sigma\_min, sigma\_max, start\_blend, etc). If set to true it will randomize, else default config or default values are used.

#Rand\_min: The minimum value suggested

#Rand\_max: The maximum value suggested. Advanced users may change the min max. Suggested min/max are included as default values in the config yaml file.

sigma\_min\_rand: false

sigma\_min\_rand\_min: 0.001

sigma\_min\_rand\_max: 0.05

sigma\_max\_rand: false

sigma\_max\_rand\_min: 10

sigma\_max\_rand\_max: 60

start\_blend\_rand: false

start\_blend\_rand\_min: 0.05

start\_blend\_rand\_max: 0.2

end\_blend\_rand: false

end\_blend\_rand\_min: 0.4

end\_blend\_rand\_max: 0.6

sharpness\_rand: false

sharpness\_rand\_min: 0.85

sharpness\_rand\_max: 1.0

early\_stopping\_rand: false

early\_stopping\_rand\_min: 0.001

early\_stopping\_rand\_max: 0.02

update\_interval\_rand: false

update\_interval\_rand\_min: 5

update\_interval\_rand\_max: 10

initial\_step\_rand: false

initial\_step\_rand\_min: 0.7

initial\_step\_rand\_max: 1.0

final\_step\_rand: false

final\_step\_rand\_min: 0.1

final\_step\_rand\_max: 0.3

initial\_noise\_rand: false

initial\_noise\_rand\_min: 1.0

initial\_noise\_rand\_max: 1.5

final\_noise\_rand: false

final\_noise\_rand\_min: 0.6

final\_noise\_rand\_max: 1.0

smooth\_blend\_factor\_rand: false

smooth\_blend\_factor\_rand\_min: 6

smooth\_blend\_factor\_rand\_max: 11

step\_size\_factor\_rand: false

step\_size\_factor\_rand\_min: 0.65

step\_size\_factor\_rand\_max: 0.85

noise\_scale\_factor\_rand: false

noise\_scale\_factor\_rand\_min: 0.75

noise\_scale\_factor\_rand\_max: 0.95

## Dynamic Configuration Reloading with Watchdog

The Simple Karras Exponential Scheduler includes a dynamic configuration reloading feature that utilizes the **Watchdog** library. This feature enables "on-the-go" configuration changes, allowing users to adjust scheduler parameters in the configuration file without stopping and restarting the program.

### What is Watchdog?

[Watchdog](https://github.com/gorakhargosh/watchdog) is a Python library that monitors file system events. It allows the program to detect changes to files or directories in real time. In this project, Watchdog is used to monitor the configuration file (simple\_kes\_scheduler.yaml), and when any modifications are detected, the program automatically reloads the configuration.

### How it Works in This Project

1. **Configuration Watcher Setup**:
   * The program includes a ConfigWatcher class that extends Watchdog's FileSystemEventHandler. This class is responsible for monitoring the configuration file and reloading it when changes occur.
   * A ConfigManagerYaml class loads the configuration initially and holds the configuration data. When Watchdog detects changes to the config file, the ConfigWatcher triggers the ConfigManagerYaml to reload the latest settings.
2. **Automatic Reloading**:
   * Once the scheduler starts, the program initializes Watchdog to observe changes in the config file directory.
   * If a change is detected, Watchdog reloads the new configuration settings, updating the scheduler's parameters dynamically without requiring a program restart.
3. **Usage Example**:
   * Make any modifications to simple\_kes\_scheduler.yaml (such as adjusting sigma\_min, sigma\_max, or randomize settings).
   * Save the file. Watchdog will automatically detect the change, and the new settings will take effect immediately.

### Benefits of Watchdog in This Project

* **Flexibility**: Adjust parameters on the fly for experimentation or fine-tuning without stopping the image generation process.
* **Efficiency**: Saves time by avoiding the need to restart the program each time you want to test new configuration values.
* **Real-Time Adaptability**: Useful in workflows requiring multiple tests, as each new configuration can be applied immediately.

### Setting Up Watchdog

Watchdog should be installed as a dependency for the project. You can install it via pip:

Pip install watchdog

# Detailed Analysis of the Simple Karras Exponential Scheduler

## Defining Sigma Max and How it Affects Blending between Karras and Exponential

This line **increases** the value of sigma\_max by **10%**. This method benefits from having an increased maximum noise level to blend the Karras and Exponential methods.

sigma\_max = sigma\_max \* 1.1

### What is it Doing?

* **Sigma (σ)** represents the noise level in a diffusion model. Higher sigma values correspond to more noise, while lower sigma values correspond to less noise.
* In this case, **sigma\_max** refers to the maximum noise level at the beginning of the image generation process. The model will start with a high amount of noise (near sigma\_max) and gradually reduce the noise level as the process progresses.
* **Multiplying sigma\_max by 1.1** increases the starting noise slightly, expanding the range of noise levels that will be used during the generation process. This can lead to **smoother transitions** between steps by providing more headroom for the denoising process.

### Theoretical Maximum and Minimum for the Multiplication Factor

#### Theoretical Minimum

The theoretical minimum value for the multiplication factor is **1.0**.

If you multiply sigma\_max by **1.0**, you leave the value unchanged. This means the scheduler will use the exact sigma\_max as originally defined by the user or configuration.

You cannot use values below 1.0 in this context because the goal is to **expand** the sigma\_max for smoother transitions. Reducing sigma\_max would have the opposite effect. It is not advised to decrease the sigma\_max.

#### Theoretical Maximum

The theoretical maximum for this multiplication factor does not have a strict upper bound, but practical constraints apply. If you **increase sigma\_max too much**, you introduce **too much noise** at the beginning, which could overwhelm the model and result in overly chaotic, grainy images, and make it harder to generate clear results later.

In practice, values like **1.1** (a 10% increase) or slightly higher (e.g., **1.2** or **1.3**) are common for adding a small buffer to smooth out the noise schedule. Higher values might be counterproductive because the starting noise would be so large that it would be difficult for the model to recover finer details in the later stages.

### Purpose of Adjusting the Sigma Max

#### Smoother Transitions

By increasing sigma\_max slightly, the range of sigma values (from sigma\_min to sigma\_max) becomes a bit larger. This allows the noise schedule (either Karras, Exponential, or the blend of both) to **spread out the noise reduction** more smoothly over the given number of steps (n).

A larger sigma\_max means the initial steps will handle more noise, making the transitions less abrupt, leading to smoother results overall.

#### Flexibility for Noise Decay

Since the **Exponential method** tends to reduce noise more rapidly in the early stages, a higher sigma\_max allows more headroom for these rapid early transitions without sacrificing stability.

For the **Karras method**, which prefers smoother, more gradual noise reduction, this increase in sigma\_max can help ensure that the early steps are still impactful enough to create the right amount of variation while keeping the process controlled.

#### Blending Between Methods

The **Simple Karras Exponential Scheduler (Simple KES)**, blends between Karras and Exponential noise schedules and increasing sigma\_max helps balance the two methods.

Exponential methods, which make large jumps in noise reduction early on, benefit from having more noise to work with initially. This ensures the blending process doesn't create abrupt changes when transitioning between the two methods.

### Why Only a 10% Increase?

The choice of **1.1** (a 10% increase) provides a **small, controlled adjustment**. A 10% buffer gives enough additional room for the noise schedule to adjust, but not so much that it overwhelms the image generation process.

This 10% increase ensures that the sigma schedule has **enough noise** to start with, but not too much to destabilize the denoising process. It also helps avoid making the model work harder to remove excess noise, which could cause artifacts in the generated image.

## Matching Sequence Lengths Explained

We start this method by generating sigma (noise) schedules using two methods, **Karras** and **Exponential**, and then combining them for use in the **Simple Karras Exponential Scheduler**. The challenge comes from the fact that the two methods can sometimes produce sequences of different lengths, which need to be handled to ensure they can be blended smoothly.

### Why is Having Matching Sequences Important?

When blending noise schedules like Karras and Exponential, having mismatched sequence lengths would cause issues when trying to apply the blend. Specifically:

If one sequence is longer than the other, you would encounter an **indexing error** when trying to access the non-existent elements in the shorter sequence.

This truncation ensures that the two sequences align properly and that every step has corresponding sigma values from both methods for blending.

**Step 1**: Generates sigma sequences using both Karras and Exponential methods.

**Step 2**: Ensures both sequences are of the same length by truncating the longer one.

**Step 3**: Catches any errors that occur during this process and replaces the sequences with zeroes to prevent crashes.

**get\_sigmas\_karras**: Generates the Karras sigma sequence, which reduces noise logarithmically for smoother transitions, particularly beneficial toward the end of the generation process.

**get\_sigmas\_exponential**: Generates the Exponential sigma sequence, which decays the noise exponentially, making early transitions more rapid than Karras.

### Matching sequence lengths

target\_length = min(len(sigmas\_karras), len(sigmas\_exponential))

This finds the shortest sequence between the two, and ensures both sequences are truncated to this length.

sigmas\_karras = sigmas\_karras[:target\_length]

sigmas\_exponential = sigmas\_exponential[:target\_length]

Ensure that both sequences (sigmas\_karras and sigmas\_exponential) have the same length. This is crucial because, for blending to work properly, the two sequences need to have an equal number of steps. The **truncation** (using [:target\_length]) ensures that both sequences are of the same length, preventing indexing issues when blending them later.

### Exception block

This **try-except block** catches any errors that might occur during sigma generation or length matching.

If an error occurs (for example, if the get\_sigmas\_\* functions fail), it logs the error using Python’s logging module and **falls back** to creating sigma sequences filled with zeros (torch.zeros(n)). This prevents the process from crashing and provides a fallback value that still allows the program to continue.

The fallback also ensures that the program can continue running on the specified device, whether it's a GPU ('cuda') or CPU ('cpu').

**Karras** and **Exponential** methods may not always return sigma sequences of the exact same length.

This discrepancy happens because each method follows different mathematical rules for generating noise values, and depending on the parameters, they may calculate different step counts.

To fix this, the code finds the **shorter** of the two sequences and **truncates** the longer one. By using the shortest length, both sequences are aligned and can be safely blended together without errors.

## Defining Progress and Sigs Variables

The Progress and Sigs variables work together during the denoising process to control the values between the two methods. The progress variable holds the values to adjust, and the sigs variable stores the changed values. Together they ensure that the blending process between the Karras and Exponential sigma values happens smoothly, adapting as the denoising progresses. These variables form the foundation for dynamically blending the Karras and Exponential methods, ensuring that the transition from one to the other happens smoothly over the course of the denoising steps.

### Progress

The **progress variable** represents how far along the denoising process is. It starts at **0** (indicating the start of the denoising process) and ends at **1** (indicating the completion of the process).

This variable will be used to **dynamically adjust** the **blend factor** and other parameters (such as step size or noise scaling) during the denoising process. As the process progresses, the values in progress allow for **smooth transitions** between the **Karras** and **Exponential** sigma sequences by gradually changing the influence of each method.

#### # Define progress and initialize blend factor

progress = torch.linspace(0, 1, len(sigmas\_karras)).to(device) sigs = torch.zeros\_like(sigmas\_karras).to(device)

These lines of code are **initializing variables** that are critical for the process of **blending** the **Karras** and **Exponential** sigma sequences in the **Simple Karras Exponential Scheduler**.

#### torch.linspace(0, 1, len(sigmas\_karras))

This creates a **1D tensor** with values that are evenly spaced between **0 and 1**, with the total number of points being equal to the length of sigmas\_karras.

The **len(sigmas\_karras)** indicates the number of steps in the sigma sequence (which should match the number of steps n in the denoising process).

The result is a series of values that incrementally move from 0 to 1 over the course of the denoising process. This is effectively a **progress indicator**.

#### Summary of Progress

This variable tracks how far along the denoising process is, moving from **0 to 1** as the process goes through all the steps (determined by len(sigmas\_karras)).

It is used to **dynamically adjust** the blend factor and other parameters (such as noise scale) as the denoising process continues.

**progress[i]** is used to determine how much weight to give to the Karras sigma versus the Exponential sigma at each step.

### Sigs Variable

#### Purpose of sigs

The sigs tensor is initialized to store the **final blended sigma values** that are calculated by combining the Karras and Exponential sigmas based on the progress through the denoising process.

At each step, the blended sigmas are calculated and stored in this tensor, which will be used later in the denoising process to adjust the noise at each step.

Since the initial values are all **zeros**, this tensor will be updated progressively as the sigma values are calculated.

sigs = torch.zeros\_like(sigmas\_karras).to(device)

This creates a new tensor filled with zeros that has the same shape (and length) as the **sigmas\_karras** tensor. Essentially, it creates a **placeholder** tensor that will store the **blended sigma values**.

#### Summary of Sigs

This is the **storage** tensor where the final sigma values (which are a blend of Karras and Exponential sigma sequences) will be stored.

At each step in the process, this tensor is updated with the **blended sigma values** that control the noise for that step of the image generation process.

**sigs[i]** stores the result of the blend for each step in the process.

## Step Size, Blend Factor, and Noise Scale

### Step Size Calculation

Purpose:

**Purpose**: This formula defines how the step size evolves from the **initial\_step\_size** at the start to the **final\_step\_size** at the end of the denoising process.

**progress[i]**: This is the progress variable, which ranges from 0 to 1 over the entire denoising process. At the first step, progress[i] is 0, and at the last step, it is 1.

step\_size = initial\_step\_size \* (1 - progress[i]) + final\_step\_size \* progress[i]

For any intermediate step, the step size is a **linear interpolation** between initial\_step\_size and final\_step\_size based on the value of progress[i]. The formula weights the initial step size more at the beginning and the final step size more at the end.

**Why Shouldn't These Values Change Much?**

* **Initial Step Size**: If this value is too high, the scheduler makes very large jumps at the beginning, which could result in missed details or instability. If it’s too low, the process could become too slow and inefficient.
* **Final Step Size**: This should be small to allow finer adjustments toward the end of the process, ensuring that small details can be captured in the final stages. If it's too high, the final image may not be as refined.

By using **1** as the multiplier for progress[i], the interpolation is smooth and linear. If you use **other values**, it would overemphasize the final value, causing more abrupt changes at the end, which could harm the final image quality.

### Dynamic Blend Factor Calculation

dynamic\_blend\_factor = start\_blend \* (1 - progress[i]) + end\_blend \* progress[i]

**Purpose**: This controls how much weight is given to the **Karras** or **Exponential** sigma sequences at any given point in the process.

At the start (progress[i] = 0), the scheduler uses more of the **start\_blend** (which might prioritize Karras, for example).

At the end (progress[i] = 1), it uses more of the **end\_blend** (which might favor Exponential sigmas).

### Why Shouldn't These Values Change Much?

If the **start\_blend** or **end\_blend** are set too high or too low, the blending between the Karras and Exponential methods could become unbalanced, leading to unnatural transitions or abrupt changes in the sigma schedule.

A **smooth transition** between the two methods (linear interpolation with the range 0 to 1) ensures the output image will combine the benefits of both approaches without creating artifacts.

Using **1** as the multiplier ensures a **gradual, even transition** from one method to the other. Higher values would accelerate the transition, causing abrupt jumps between the Karras and Exponential methods, which could introduce undesirable effects in the image generation.

### Noise Scale Calculation

noise\_scale = initial\_noise\_scale \* (1 - progress[i]) + final\_noise\_scale \* progress[i]

**Purpose**: This defines how much noise is applied at each step, transitioning from an **initial noise scale** to a **final noise scale**.

This helps control how aggressively the noise is reduced as the generation progresses.

At the **end** (progress[i] = 1):

The noise scale transitions smoothly to the **final value**, which should be lower to refine the details at the end of the denoising process.

**Why Shouldn't These Values Change Much?**

* **Initial Noise Scale**: If it’s too high, the process might add too much randomness early on, making it harder to recover details later. If it’s too low, the denoising process might not capture enough variation early on.
* **Final Noise Scale**: If this is too high, the image could end up too noisy at the end. If it’s too low, the process might become too rigid, and small details could be lost.

Again, using **1** as the interpolation multiplier provides a **linear, smooth transition**. Higher multipliers would make the noise scale jump more dramatically, possibly introducing inconsistencies in the noise removal process.

### Summary of Methods

**Step Size**, **Blend Factor**, and **Noise Scale** all interpolate smoothly between their initial and final values based on the progress.

Using a **linear interpolation** with the multiplier set to **1** ensures a **smooth transition** over the denoising process.

Using higher multipliers (like 1.5 or 2) would lead to **non-linear transitions**, causing the process to be too abrupt, which can negatively impact the final image quality.

## Smooth Blend and the Sigmoid Function

smooth\_blend = torch.sigmoid((dynamic\_blend\_factor - 0.5) \* 11) # Increase scaling factor to smooth transitions more

**Smooth blending** occurs between two sigma sequences (Karras and Exponential) using a **sigmoid function** to control the **smoothness** and **gradualness** of the transition between the two.

### Sigmoid Function

The **sigmoid function** is a mathematical function that maps any input value to a range between **0** and **1**. The sigmoid function is useful for creating smooth transitions because it transforms large negative values to something close to 0 and large positive values to something close to 1, with the steepest change occurring near 0.

sigmoid(x)=1/ (1+e^−x1​)

This function is often used in machine learning for smooth probability estimates, but here it’s being used to smoothly transition between values (blend factors).

When x is large and positive, the sigmoid approaches 1.

When x is large and negative, the sigmoid approaches 0.

When x is near 0, the sigmoid is around 0.5, meaning this is the point where the transition happens most rapidly.

### Dynamic Blend Factor Adjustment

dynamic\_blend\_factor - 0.5

**dynamic\_blend\_factor** is the current blend factor between Karras and Exponential methods, which typically ranges between 0 and 1.

* **When dynamic\_blend\_factor = 0**: You are entirely using one method (Karras or Exponential).
* **When dynamic\_blend\_factor = 1**: You are fully using the other method.
* **Intermediate values** represent partial blending between the two methods.

**dynamic\_blend\_factor - 0.5**:

* Subtracting **0.5** from dynamic\_blend\_factor shifts its range from **[-0.5, 0.5]**.
* This is important because it places **0.5** (the midpoint, or equal blending) at the center of the sigmoid function’s steepest point.
* When dynamic\_blend\_factor is 0.5, the sigmoid input is 0, meaning the sigmoid output will be 0.5, creating a **balanced blend**.
* When dynamic\_blend\_factor is closer to 0 or 1, the output of the sigmoid will approach 0 or 1, but smoothly.

### Scaling the Input

(dynamic\_blend\_factor - 0.5) \* 11

**Why multiply by 11?**

The **scaling factor** of **11** is used to **control the sharpness** of the transition between the two sigma sequences.

By multiplying by 11, the transition between sigma sequences becomes **sharper** around the midpoint (where dynamic\_blend\_factor = 0.5), but still smooth. This gives the user more **control** over how the blending occurs.

**Smaller scaling factors (like 1 or 2)**: The transition between the two sigma sequences would be more gradual, meaning the blending would happen over more steps.

**Larger scaling factors (like 11)**: The transition becomes sharper, concentrating the blend around dynamic\_blend\_factor = 0.5. This creates a **more distinct transition** between the two methods.

The value **11** was chosen based on experimentation, as it balances the smooth transition with a sharper blending zone.

### Purpose of the Sigmoid and Scaling

The **sigmoid** with a scaling factor provides **non-linear blending** between the two sigma sequences. Here’s how it works in practice:

* At the **start** of the process, where the dynamic\_blend\_factor is close to **0**, the sigmoid output is near 0, meaning the blend is mostly weighted towards one sigma sequence (the Karras method).
* At the **end** of the process, where dynamic\_blend\_factor is close to **1**, the sigmoid output is near 1, meaning the blend is mostly weighted towards the other sigma sequence (the Exponential method).
* In the **middle** (when dynamic\_blend\_factor is near 0.5), the sigmoid output is around 0.5, creating an **equal blend** between the two sigma sequences.

### Why Use the Sigmoid for Blending?

**Smooth Transition**: The sigmoid function is chosen because it offers a **smooth, non-linear** transition between 0 and 1, making the blending process between the two sigma sequences more gradual and natural. A sharp cutoff would create **visual artifacts** or abrupt transitions in the image generation process.

**Control over the Transition Point**: By centering the sigmoid function around 0.5 (the midpoint of the blend), you ensure that the **most rapid blending** happens when the scheduler is roughly halfway through the denoising process.

**Enhancing Flexibility**: The scaling factor (11 in this case) allows the transition to be **adjustable**. If you want a very smooth transition, you might reduce the scaling factor. If you want a more distinct switch from Karras to Exponential, you increase the scaling factor.

### Why Not Use Larger/Smaller Scaling Factors?

**Larger values (>11)**: A very large scaling factor (e.g., 20 or 50) would make the transition extremely **sharp**, meaning the blending would happen very suddenly between the sigma sequences. This could create **instability** or **artifacts** in the generated images because the blending wouldn't be gradual enough.

**Smaller values (<11)**: A much smaller scaling factor (e.g., 2 or 3) would make the transition too **slow and gradual**, causing the process to feel like a blend of both methods almost throughout the entire denoising process. This would reduce the distinctiveness of using the two different sigma schedules and make it harder to capture the best parts of both.

### Dynamic Blend Factor Adjustment Summary

**torch.sigmoid((dynamic\_blend\_factor - 0.5) \* 11)** creates a **smooth, sharp transition** between two sigma sequences (Karras and Exponential).

* The **sigmoid function** controls how much of each sigma sequence is applied, based on the dynamic\_blend\_factor.
* The **scaling factor** of **11** makes the blending sharper, concentrating the transition around the midpoint of the denoising process.
* This approach ensures a **smooth but controlled blending** of sigma sequences, allowing the scheduler to effectively combine the benefits of both methods without abrupt transitions.

## Blending Karras and Exponential

blended\_sigma = sigmas\_karras[i] \* (1 - smooth\_blend) + sigmas\_exponential[i] \* smooth\_blend

This is the **blending** of two sigma sequences — **Karras sigmas** and **Exponential sigmas** — to produce a **blended sigma value** for the current step i in the denoising process. This blending is controlled by the **smooth\_blend** value, which smoothly transitions the weight between the two sequences.

### Blending Concept

**Sigma sequences** (sigmas\_karras and sigmas\_exponential) represent different noise schedules.

The **blending process** combines these two sequences to create a new sigma value (blended\_sigma) at each step of the image generation process.

**smooth\_blend** controls how much weight is given to each sequence at step i, allowing for a gradual transition between the Karras and Exponential schedules.

### Formula

blended\_sigma = sigmas\_karras[i] \* (1 - smooth\_blend) + sigmas\_exponential[i] \* smooth\_blend

This formula is a **weighted average** of the two sigma values at step i:

**sigmas\_karras[i] \* (1 - smooth\_blend)**:

* This gives more weight to the **Karras** sigma value when smooth\_blend is close to 0, meaning the process is relying more on the Karras schedule early on.
* As smooth\_blend increases, this term gradually reduces its weight, since 1 - smooth\_blend decreases.

**sigmas\_exponential[i] \* smooth\_blend**:

* This gives more weight to the **Exponential** sigma value as smooth\_blend increases. Early in the process, smooth\_blend will be close to 0, so the Exponential sigma contributes very little.
* As smooth\_blend approaches 1, the weight on the Exponential schedule increases, meaning it takes over more as the process progresses.

## Blending Behavior

### Why This Blending Matters

The **blended sigma values** allow the **Simple Karras Exponential Scheduler (Simple KES)** to combine the benefits of both Karras and Exponential methods:

* **Karras method**: Focuses on **smooth, gradual noise reduction** with fine control over the denoising process, particularly at later stages.
* **Exponential method**: Emphasizes **faster noise decay** in the early steps, making the initial denoising more aggressive.

By blending these two approaches, **Simple KES** can:

* Start with the advantages of one schedule (e.g., Karras) in the early steps.
* Gradually transition to the other schedule (e.g., Exponential) later in the denoising process.

This results in smoother transitions and more controlled noise reduction, allowing the model to achieve a balance between fast initial denoising and fine-grained control toward the end.

To understand the blending behavior, let’s examine what the formula would look like at stages of the blending process.

#### When smooth\_blend is close to 0

* At the beginning of the process (early steps), smooth\_blend is close to 0, so the formula looks like:

blended\_sigma ≈ sigmas\_karras[i] \* 1 + sigmas\_exponential[i] \* 0

≈ sigmas\_karras[i]

The result is dominated by the **Karras sigma** value at that step.

#### When smooth\_blend is close to 1

* At the end of the process (later steps), smooth\_blend is close to 1, so the formula looks like:

blended\_sigma ≈ sigmas\_karras[i] \* 0 + sigmas\_exponential[i] \* 1

≈ sigmas\_exponential[i]

The result is dominated by the **Exponential sigma** value at that step

#### At intermediate values of smooth\_blend

(e.g., when smooth\_blend = 0.5):

* The formula produces an **equal blend** of the two sigma values

blended\_sigma ≈ sigmas\_karras[i] \* 0.5 + sigmas\_exponential[i] \* 0.5

This means the noise schedule at that step is equally weighted between the Karras and Exponential sequences.

### Blending Behavior Summary

* The formula blends the **Karras** and **Exponential** sigma sequences at each step using the **smooth\_blend** factor.
* The **weighting** of each sequence changes smoothly as the process progresses, with more weight on **Karras** early on and more weight on **Exponential** later.
* This blending provides the benefits of both sigma schedules, leading to better image generation outcomes by balancing smoothness and efficiency.

## Applying Step Size and Noise Scaling

This line takes the **blended sigma value** (from the previous step) and **modifies it** by applying both the **step size** and **noise scaling factors**. The result is stored in sigs[i], which represents the final sigma value for step i in the denoising process. This essential because it ensures that the denoising process happens smoothly and adaptively, transitioning from broad noise reduction to fine-tuning as the model generates the image.

sigs[i] = blended\_sigma \* step\_size \* noise\_scale

### Blended Sigma

The **blended\_sigma** is the result of combining the **Karras** and **Exponential** sigma sequences for step i using the blending formula discussed earlier:

blended\_sigma = sigmas\_karras[i] \* (1 - smooth\_blend) + sigmas\_exponential[i] \* smooth\_blend

This value represents the sigma (noise) at step i, already blended between the two methods (Karras and Exponential).

### Step Size

**Step size** is an adaptive factor that controls the **rate of change** between successive noise values (or sigmas). It adjusts how large or small each step in the denoising process should be.

Early in the process, the **step size** might be larger (when initial\_step\_size dominates), allowing for faster reduction of noise. Toward the end, the step size shrinks (as final\_step\_size takes over), enabling more refined adjustments.

This adaptive adjustment ensures that large changes are made early, while finer tweaks are applied later on.

### Noise Scale

The **noise scale** is another factor that modifies the sigma value to **control the amount of noise** added or removed at each step.

It dynamically adjusts how much the noise is scaled as the process progresses. Early in the process, a larger **initial\_noise\_scale** may allow more noise to be present, whereas toward the end, the **final\_noise\_scale** reduces the noise more aggressively, focusing on refining details.

Like the step size, the **noise scale** is also adaptively interpolated between an initial and a final value to allow more noise to be handled early on and fine-tuned adjustments later.

### Final Sigma Value for Step i

sigs[i] = blended\_sigma \* step\_size \* noise\_scale

* **Blended Sigma**: Represents the sigma at the current step, blended between the Karras and Exponential methods.
* **Step Size**: Scales the blended sigma value based on how large or small a change in noise should be at this point in the denoising process.
* **Noise Scale**: Further scales the sigma value to control how much noise should be applied or removed.

### The Multiplication Process

* **Step Size** controls the overall magnitude of the change, dictating how large or small the step in the denoising process will be.
* **Noise Scale** adds an additional layer of control, ensuring that the noise is managed in a way that adapts to the current stage of denoising (e.g., more noise at the beginning, less at the end).

Together, these two factors modulate the **blended\_sigma** to produce a final value (sigs[i]), which is the sigma value to be used for step i in the process.

### Why Apply Both Step Size and Noise Scale?

**Step Size**:

The **step size** allows for control over how **rapidly** or **gradually** the noise is reduced between steps.

Early on, larger steps allow the model to quickly reduce noise and make broad changes to the image.

Later, smaller steps ensure that the noise reduction becomes more **delicate**, allowing the model to refine finer details as the image nears completion.

**Noise Scale**:

The **noise scale** ensures that the amount of noise present is scaled appropriately at each stage of the process.

Early in the process, a higher noise scale might be used to handle the large amounts of noise present in the initial random image.

Later on, the noise scale is reduced to allow more fine-tuned adjustments, ensuring that unnecessary noise isn't added when the image is almost fully denoised.

### Why Not Just Use Blended Sigma Alone?

**Blended sigma** alone provides a value that is blended between two methods (Karras and Exponential), but it doesn't control the **rate of noise reduction**.

By introducing **step size** and **noise scale**, the model gains **adaptive control** over the entire process:

* **Step Size** ensures that large steps are made early on, and small steps are made toward the end.
* **Noise Scale** ensures that the amount of noise is adjusted dynamically throughout the denoising process.

### Applying Step Size and Noise Scaling Summary

* **blended\_sigma** provides the noise level at step i, combining Karras and Exponential methods.
* **step\_size** controls the size of the changes in noise, making the process faster early on and more refined later.
* **noise\_scale** further adjusts the amount of noise, allowing for more flexibility in noise control at each stage.
* **sigs[i]** stores the final sigma value for step i, which will be used to adjust the image's noise in the current step of the denoising process.

## Adaptive Sharpening

These lines implement **adaptive sharpening** during the denoising process by selectively modifying the sigma values based on certain conditions.

sharpen\_mask = torch.where(sigs < sigma\_min \* 1.5, sharpen\_factor, 1.0).to(device)

sigs = sigs \* sharpen\_mask

### What is Adaptive Sharpening?

Adaptive sharpening in this context is used to **sharpen** the generated image by making adjustments to the sigma values based on their magnitude. This helps fine-tune details in certain parts of the denoising process, ensuring that regions where noise has already been sufficiently reduced get an additional sharpening effect.

### torch.where() Function

This line uses torch.where(), a conditional function that works similarly to an if-else statement in Python. The torch.where function operates element-wise on tensors, selecting values based on a condition.

torch.where(condition, x, y)

This returns a tensor where:

* If the **condition** is true, it takes the value **x**.
* If the **condition** is false, it takes the value **y**.

torch.where(sigs < sigma\_min \* 1.5, sharpen\_factor, 1.0)

**Condition**: sigs < sigma\_min \* 1.5

* This checks if the sigma values (sigs) are less than **1.5 times** the minimum sigma (sigma\_min).
* sigma\_min represents the **minimum allowable noise level** in the denoising process. By multiplying it by 1.5, you're defining a threshold for when sharpening should be applied.
* When the sigma value drops below this threshold, the sharpening effect will be applied.

**True case (sharpen\_factor)**:

* If the sigma value is less than sigma\_min \* 1.5, the **sharpen\_factor** is applied. This could be a value less than 1, designed to sharpen the image by enhancing details in regions where noise is already quite low.

**False case (1.0)**:

* If the sigma value is **greater than or equal to** sigma\_min \* 1.5, the value 1.0 is used. This means no sharpening effect will be applied, as multiplying by 1.0 keeps the sigma value unchanged.

**.to(device)**:

* This ensures the resulting tensor (sharpen\_mask) is transferred to the appropriate **device** (e.g., GPU or CPU) for computation.

#### Purpose of the sharpen\_mask

* **sharpen\_mask** is a tensor that contains either **sharpen\_factor** or **1.0** for each sigma value, depending on whether that value is below the threshold (sigma\_min \* 1.5).
  + **If the condition is met** (sigma is low), the sharpen factor is applied.
  + **If the condition is not met**, the value is left unchanged (multiplied by 1.0).

### Multiplying the Sigmas by the Sharpen Mask

sigs = sigs \* sharpen\_mask

This line **modifies the sigma values** (sigs) by multiplying them by the corresponding values in the **sharpen\_mask**.

The effect is that **if the sigma value is low** (below the threshold of sigma\_min \* 1.5), the sharpening effect (via sharpen\_factor) is applied.

* This might make the sigma smaller, which leads to a more refined and sharper output, particularly in areas where the noise has already been sufficiently reduced.

**If the sigma value is not low** (above or equal to sigma\_min \* 1.5), no sharpening occurs, because multiplying by 1.0 leaves the sigma value unchanged.

### Why This Works for Adaptive Sharpening

#### Low Sigma Values Indicate Finer Detail

When the sigma values are low, the denoising process is approaching the final steps where finer details are being resolved. At this stage, additional sharpening can help enhance details that might be lost in the earlier broad noise removal.

By applying sharpening only when the sigma values are below a certain threshold, this ensures that the sharpening effect is applied **adaptively** and only when it’s beneficial.

#### Selective Application of Sharpening

The sharpening is **selective**, meaning it only applies in situations where the noise has already been reduced to a low level (indicating that the image is nearing completion).

Applying sharpening at every step or when the noise is still high could have a negative effect, as it would sharpen regions where large noise patterns are still present, leading to artifacts. By using the condition (sigs < sigma\_min \* 1.5), sharpening is applied only when appropriate.

### Why Use 1.5 as the Threshold?

The choice of **1.5 times** sigma\_min is based on experimentation to determine when noise reduction reaches a level where sharpening can be beneficial without causing artifacts:

**If the threshold were lower** (e.g., sigma\_min or less), the sharpening might be applied too late in the process, when the noise is already very low, limiting its impact.

**If the threshold were higher** (e.g., 2 \* sigma\_min), the sharpening might be applied too early, while there is still too much noise, which could degrade image quality by sharpening noisy areas.

The factor of **1.5** balances these concerns, applying sharpening at a stage where noise is low enough for details to emerge but not so low that sharpening is ineffective.

### Adaptive Sharpening Summary

* **torch.where(sigs < sigma\_min \* 1.5, sharpen\_factor, 1.0)**: This creates a mask to selectively apply a sharpening factor when sigma values drop below 1.5 times the minimum sigma.
* **sigs = sigs \* sharpen\_mask**: This applies the sharpening effect by modifying the sigma values, sharpening the image when sigma values are low, and leaving higher sigma values unchanged.
* **Purpose**: This method adds adaptive sharpening to the denoising process, selectively refining details when noise is sufficiently reduced, which enhances the final image quality without introducing artifacts.

## Early Stopping

These lines of code implement **early stopping** based on how much the sigma values are changing between consecutive steps. The purpose of this is to **stop the denoising process early** if the sigma values converge (i.e., if the changes between consecutive sigma values become very small). Early stopping helps save computation time when further steps would not meaningfully improve the result.

change = torch.abs(sigs[1:] - sigs[:-1])

if torch.all(change < early\_stop\_threshold):

logging.info("Early stopping criteria met.")

return sigs[:len(change) + 1].to(device)

### Calculate the Change Between Consecutive Sigma Values

change = torch.abs(sigs[1:] - sigs[:-1])

**sigs[1:]** and **sigs[:-1** are slices of the sigs tensor:

* **sigs[1:]**: This is the tensor containing all sigma values except the first one.
* **sigs[:-1]**: This is the tensor containing all sigma values except the last one.

By subtracting **sigs[:-1]** from **sigs[1:]**, you get the **difference** between each consecutive pair of sigma values in the sequence.

**torch.abs()**: This computes the **absolute value** of the differences between consecutive sigma values. We care about the **magnitude** of the change, not whether it’s increasing or decreasing, so the absolute value is used.

#### Purpose of This Step

* The **change** tensor holds the absolute differences between sigma values at each step. This is used to determine whether the sigma values are **converging**, meaning that they are no longer changing significantly between steps.

### Check If All Changes Are Below the Early Stop Threshold

if torch.all(change < early\_stop\_threshold):

**torch.all()**: This checks if **all** elements of the change tensor satisfy the condition inside.

**change < early\_stop\_threshold**: This condition checks whether every value in the change tensor is less than the **early\_stop\_threshold**.

* **early\_stop\_threshold** is a small, user-defined value that represents the **convergence tolerance**. It sets the minimum amount of change between steps that is considered significant.
* If the change between sigma values at every step is smaller than this threshold, the process has **converged** sufficiently, meaning further steps won’t significantly alter the result.

#### Purpose of This Step

* The **if condition** checks whether the change in sigma values between every consecutive step has become **smaller than the early stop threshold**.
* If this is true, it means that continuing the denoising process is unlikely to produce better results because the changes are so small. Therefore, it's more efficient to **stop the process early** and save computational resources.

### Logging and Early Return

logging.info("Early stopping criteria met.")

return sigs[:len(change) + 1].to(device)

**logging.info("Early stopping criteria met.")**: This logs a message indicating that the early stopping condition has been met. This can be useful for debugging or tracking when and why the process stops early.

**return sigs[:len(change) + 1].to(device)**:

* This returns the **sigma values** up to the point where early stopping is triggered.
* The change tensor has one fewer element than the sigs tensor because it's the difference between consecutive elements. Therefore, to return the corresponding sigma values, we use **len(change) + 1** to ensure that the returned tensor includes all relevant sigma values.
* **.to(device)** ensures the returned sigma values are on the correct device (e.g., GPU or CPU).

#### Purpose of This Step

* The denoising process is **stopped early** when the change between sigma values becomes negligible (i.e., below the early\_stop\_threshold). By returning early, the process avoids wasting time and computational resources on steps that will not meaningfully improve the result.
* The **returned sigs tensor** contains the sigma values up to the point where the process was stopped, which will be used for the remainder of the image generation process.

### Why Use Early Stopping Based on Sigma Convergence?

#### Efficiency

If the sigma values are no longer changing significantly, continuing the process is unnecessary because the image is essentially as refined as it will get. Early stopping saves computation time and resources.

#### Convergence

As the sigma values converge, the changes become so small that additional steps are unlikely to improve the image’s quality. The early stop condition ensures the process halts once the sigma sequence is stable.

#### Customizable Threshold

The **early\_stop\_threshold** can be adjusted based on the desired balance between computational efficiency and image quality. A lower threshold means the process will run longer (with more refinement), while a higher threshold stops the process earlier (saving more time).

### Early Stopping Summary

* **change = torch.abs(sigs[1:] - sigs[:-1])**: Calculates the absolute differences between consecutive sigma values.
* **if torch.all(change < early\_stop\_threshold)**: Checks if all changes are below the defined threshold, indicating that the process has converged.
* **Early return**: If the sigma values have converged, the process stops early, saving computation time.

This early stopping mechanism is a way to optimize the denoising process by halting it once the sigma values stabilize, ensuring efficiency without sacrificing quality.

## Randomization of Parameters

### Randomize

The randomize feature allows for dynamic variation of key scheduler parameters to introduce diversity and adaptability into the noise generation process. When enabled, randomization provides a range for each parameter, resulting in a unique blend of values on each execution. This adds variability in the image generation process, making it more versatile and suitable for experiments where slight adjustments can reveal new artistic qualities or help identify optimal configurations.

### Configuration Options for Randomization

Each parameter in the scheduler configuration (e.g., sigma\_min, sigma\_max, start\_blend, etc.) has corresponding randomization controls. These options let you specify minimum and maximum bounds and toggle randomization on or off for each parameter individually.

For instance:

* sigma\_min\_rand: Enables randomization of sigma\_min.
  + **sigma\_min\_rand\_min**: Defines the minimum bound for sigma\_min randomization.
  + **sigma\_min\_rand\_max**: Sets the maximum bound for sigma\_min randomization.

Each parameter can be randomized with a similar set of \_rand, \_rand\_min, and \_rand\_max settings, ensuring that the process is both customizable and controlled.

### Enabling Randomization

To activate this feature globally, set randomize: true in the configuration file under the scheduler section. Individual parameter randomization is controlled by their respective \_rand flags, which must also be enabled for the parameter to vary within its defined range.

With the randomize feature, the scheduler function will:

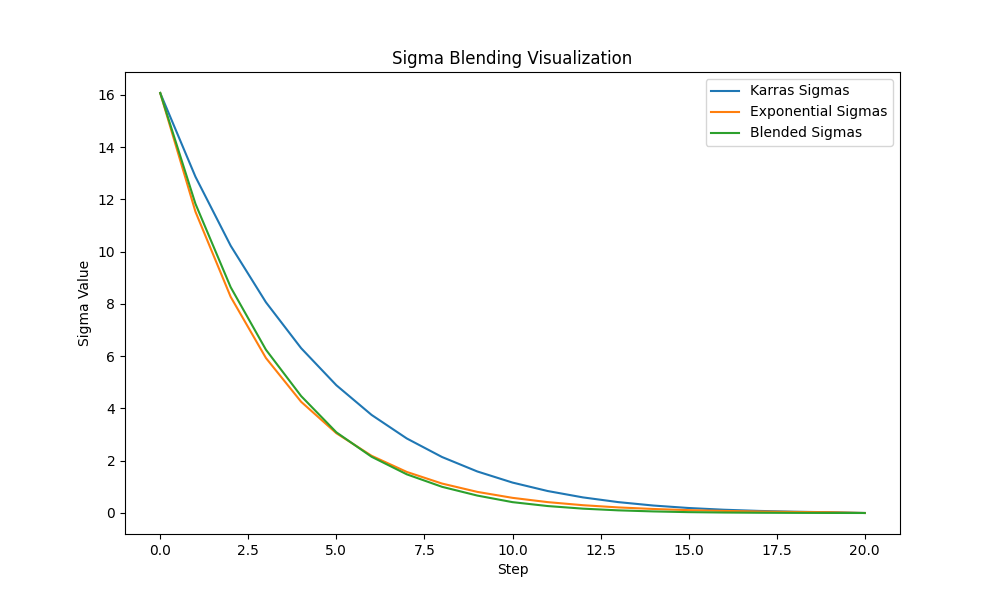
1. Check the randomize flag in the configuration.
2. Apply randomization to each parameter where both randomize and the individual \_rand flags are enabled.
3. Log each randomized value for transparency and reproducibility.

### Benefits of Randomization

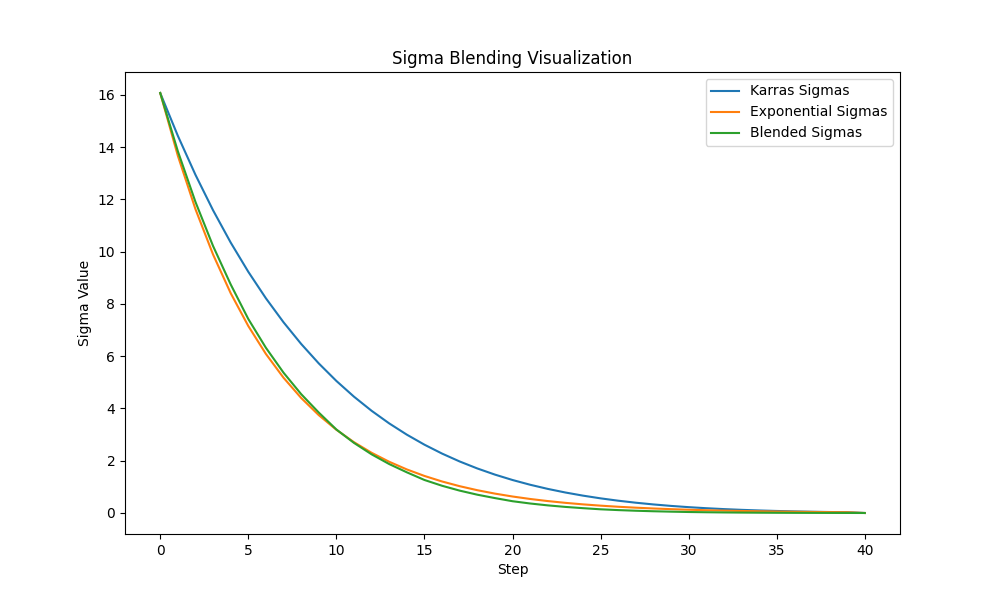
* **Diversity**: Produces a wider range of visual results from the same base image.
* **Experimental Flexibility**: Helps in exploring a broader parameter space, particularly useful in research or when tuning the scheduler.
* **Enhanced Adaptability**: Allows the scheduler to adjust dynamically, which can result in more nuanced and visually engaging images.

## Graph References

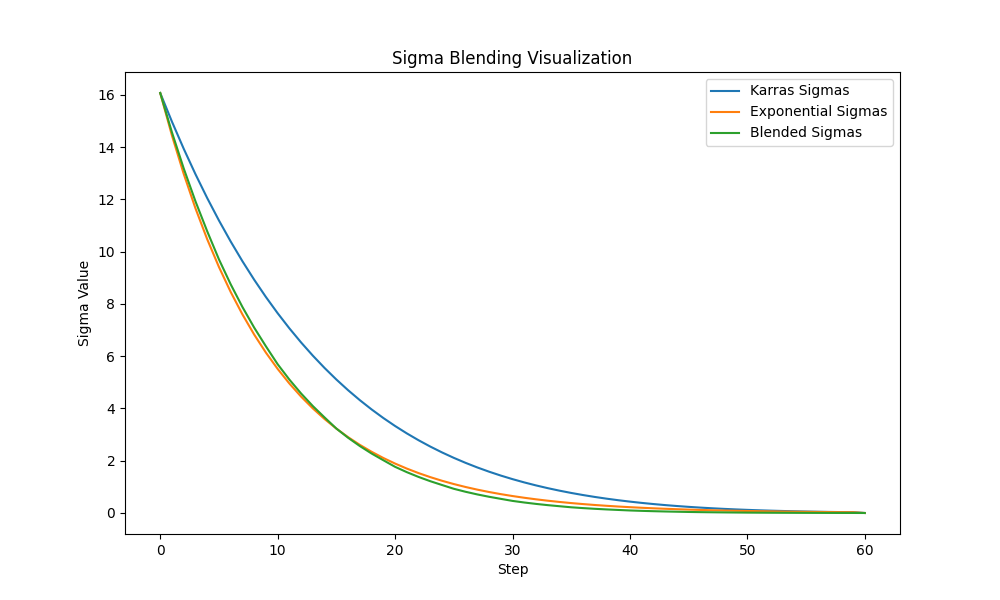
### 20 Steps



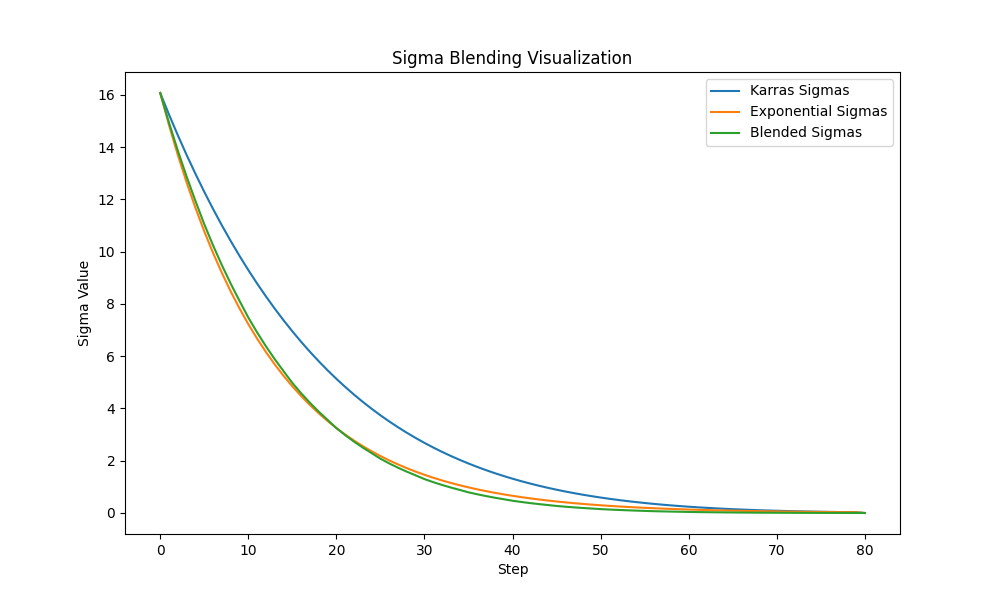
### 40 Steps



### 60 Steps



### 80 Steps



### 100 Steps

