

# Lim.one 技术文档

本文档以源码为准，覆盖两部分：

1. 全部参数 (APVTS 参数 ID、范围、默认值、含义、在哪一段 DSP 生效)
2. DSP 完整信号流与算法细节 (Clipper + Limiter Classic/Modern + 过采样 + Meter/LUFS)

代码入口：

- 参数定义：[PluginProcessor.cpp](#)
  - 音频主流程：[processBlock](#)
  - Clipper 入口：[Clipper.h](#)
  - Clipper Soft/Hard 实现：[ClipperSoftClipper.h](#)、[ClipperHardClipper.h](#)
  - Limiter 入口：[Limiter.h](#)
  - Limiter Classic/Modern 实现：[LimiterClassic.h](#)、[LimiterModern.h](#)
  - UI/Meter 推送：[PluginEditor.cpp](#)
  - Web UI 入口：[index.html](#)
- 

## 1 参数总表

完整参数列表见 CSV 文件：[parameters.csv](#)

ID	显示名	类型	范围	默认	单位
in_gain	Input Gain	Float	-24.0 → 24.0 (step 0.01)	0.0	dB
drive	Input Drive / Clip Gain	Float	0.0 → 18.0	0.0	dB
knee	Knee	Float	0.0 → 1.0	0.0	归一化
clipper_mode	Clipper Mode	Choice	Soft / Hard	Soft	-
limiter_drive	Limiter Drive / Limit Gain	Float	0.0 → 18.0	0.0	dB
character	Character	Float	0.0 → 1.0	0.5	归一化
ceiling	Ceiling	Float	-1.0 → 0.0	-0.1	dB
out_gain	Output Gain	Float	-24.0 → 1.0 (step 0.01)	-0.1	dB
true_peak	True Peak	Bool	Off/On	On	-
delta	Delta	Bool	Off/On	Off	-
limiter_mode	Limiter Mode	Choice	Classic / Modern	Modern	-
oversampling	Oversampling	Choice	Off / 2x / 4x / 8x / 16x	4x	-
lookahead	Lookahead	Float	0.00 → 5.00 (step 0.01)	2.00	ms
ui_mode	UI Mode	Choice	Simple / Adv	Simple	-
modern_hold_ms	Modern Hold	Float	0.00 → 10.00	4.00	ms
modern_hold_release_ms	Modern Hold Release	Float	0.05 → 10.00	1.50	ms
modern_attack_tau_div	Modern Attack Tau Div	Float	0.25 → 8.00	2.50	-
modern_release_smooth_base_ms	Modern Release Smooth Base	Float	0.00 → 20.00	1.50	ms
modern_release_smooth_range_ms	Modern Release Smooth Range	Float	0.00 → 50.00	6.00	ms
modern_adapt_fast_strength	Modern Adapt Fast Strength	Float	0.00 → 8.00	1.50	-
modern_adapt_slow_strength	Modern Adapt Slow Strength	Float	0.00 → 16.00	3.00	-
modern_sc_hpf_hz	Modern SC HPF	Float	20 → 400 (step 1)	120	Hz
modern_ratio_base	Modern Ratio Base	Float	0.10 → 3.00	1.50	-
modern_ratio_slope	Modern Ratio Slope	Float	0.00 → 1.50	0.90	-
modern_transient_mix	Modern Transient Mix	Float	0.00 → 1.00 (step 0.001)	0.30	-
modern_soft_safety_strength	Modern Soft Safety Strength	Float	0.00 → 20.00	6.00	-
modern_release_fast_ms	Modern Release Fast	Float	0.05 → 20.00	2.00	ms
modern_release_slow_ms	Modern Release Slow	Float	0.10 → 200.0 (step 0.1)	30.0	ms
modern_link_transients	Modern Link Transients	Float	0.00 → 1.00 (step 0.001)	0.50	-
modern_link_release	Modern Link Release	Float	0.00 → 1.00 (step 0.001)	0.95	-
ui_scale	UI Scale	Float	50 → 200 (step 1)	100	%

参数表源码位置: [PluginProcessor.cpp](#)

## 2 Web UI 与可视化 (实现口径)

Web UI 资源位于 [web](#) 目录, 通过 JUCE WebView 加载:

- HTML 入口: [index.html](#)
- JS 模块: [ui.js](#)、[controls.js](#)、[ipc.js](#)、[state.js](#)、[init.js](#)

### 2.1 UI 缩放 (实现口径)

UI 缩放通过 `ui_scale` 参数驱动，包含 Host 窗口尺寸调整 + Web 内部缩放两层：

- Host 侧 (Editor)：读取 `ui_scale` (50%~200%)，将插件窗口尺寸设置为 `baseUiWidth/baseUiHeight` 乘以缩放比；并在 `timerCallback()` 中实时监听变化更新。见 [PluginEditor.cpp](#)、[PluginEditor.cpp](#)
- Web 侧：`updateUiScaleUI()` 将 `scale-inner` 容器按 `scale()` 缩放，同时按当前 `window.innerWidth/innerHeight` 反向设置 inner 尺寸，避免缩放后布局裁切。见 [ui.js](#)
- 交互入口：设置面板中的下拉框 `select-ui-scale` 写回 `ui_scale` 参数。见 [index.html](#)、[controls.js](#)
- 自适应：浏览器窗口尺寸变化会触发 `updateUiScaleUI()` 重新计算 inner 尺寸。见 [ui.js](#)

## 2.2 可视化数据流

- C++ 侧把 meter/lufs/viz 数据推送到 WebView (见 [PluginEditor.cpp](#))。
- Web 侧由 `AudioPlugin.Visualizer.update(meterData)` 接收，维护环形缓冲 (`outDb/clipDb/limitDb/lufsSt` 四条历史轨)。见 [Visualizer.update](#)
- 绘制由 `requestAnimationFrame(draw)` 驱动；每帧按 Range/Ticks/Speed 渲染波形 + GR + LUFS ST 历史线。见 [draw](#)

## 2.3 停止/暂停时仍持续滚动

当宿主停止回调导致 `meterData` 不再到达时，`draw()` 会调用 `synthIfStale(ts)`：按“像素每秒 pps”做连续时间累积 (`synthFrac`)，以固定上一次的 out/gr/lufs 值补点推进时间轴，保证 UI 视觉持续滚动且不会出现抽搐。见 [synthIfStale](#)

## 2.4 Range/Ticks/Speed 的永久记忆

可视化设置使用 `localStorage` 持久化：

- `saveSettings()` 在用户切换 Ticks/Range/Speed 后写入
- `loadSettings()` 在可视化初始化时恢复；旧版本保存的 Range=6 dB 会自动映射到当前最接近的可用范围

实现见 [saveSettings/loadSettings](#) 与 [ensure](#)

## 3 Character 宏映射 (Simple UI 时)

当 `ui_mode == 0` (Simple) 时，不使用 `modern_*` 的用户值，而是用 `character` (记为  $t \in [0,1]$ ) 生成一套 Modern 高级参数，覆盖传入 Limiter。映射的目标是把一个宏旋钮转换成“更短的时间常数 + 更松的立体声链接 + 更强的瞬态偏好”，从而获得更激进的响度行为。

- 线性插值：`lerp(a,b,t) = a + (b-a)*t`
- 指数插值：`expLerp(a,b,t) = a * (b/a)^t` ( $a,b>0$ )

其中 `t = clamp(character, 0, 1)`。

映射代码见：[PluginProcessor.cpp](#)

### 3.1 映射明细 (实现精确口径)

下表给出每个被覆盖的 `used*` 参数的计算方式（最终值还会再做一次范围 clamp，与对应参数的 `NormalisableRange` 一致）：

目标参数 (Modern)	插值	公式 (以 t 为自变量)	解释 (t↑ 的趋势)
hold_ms	lerp	lerp(8.0, 0.8, t)	Hold 变短, 更快进入释放
hold_release_ms	expLerp	expLerp(4.0, 0.35, t)	Hold 回弹更快
attack_tau_div	lerp	lerp(3.2, 1.6, t)	attackTau = lookahead/tauDiv ⇒ tau 变大, 攻击更“贴峰”
release_smooth_base_ms	lerp	lerp(4.0, 0.6, t)	Release 平滑更短, 更锐利
release_smooth_range_ms	lerp	lerp(18.0, 4.0, t)	GR/低频驱动的额外平滑范围变小
adapt_fast_strength	lerp	lerp(2.2, 1.2, t)	低频/大 GR 时对“快释放”的拉长程度变弱
adapt_slow_strength	lerp	lerp(4.5, 2.0, t)	低频/大 GR 时对“慢释放”的拉长程度变弱
sc_hpf_hz	lerp	lerp(80.0, 180.0, t)	侧链 HPF 抬高 (更不吃低频)
transient_mix	lerp	lerp(0.15, 0.75, t)	瞬态判定时更偏向 fast (更“咬”瞬态)
release_fast_ms	expLerp	expLerp(6.0, 0.7, t)	快释放更快
release_slow_ms	expLerp	expLerp(90.0, 18.0, t)	慢释放更快 (更少泵感但更可能失真)
link_transients	lerp	lerp(0.95, 0.35, t)	瞬态阶段的立体声链接更弱 (更宽/更大响度)
link_release	lerp	lerp(0.99, 0.75, t)	释放阶段链接更弱

## 3.2 未被覆盖的参数 (Simple UI 下仍保持自身值)

Simple UI 的映射只覆盖上表 12 个参数; 以下参数不会被 `character` 改写 (仍来自 `modern_*` 参数当前值, 默认即 `PluginProcessor.cpp` 中的默认值) :

- `ratio_base`
- `ratio_slope`
- `soft_safety_strength`

## 4 连动系统与手势逻辑 (v0.0.9 新增)

### 4.1 连动 (Linking) 机制

每个 Modern 参数 (如 `modern_attack`) 都有一个对应的布尔参数 (如 `modern_attack_link`) 。

- **Link = On:** 参数值完全由 `character` 宏参数通过曲线映射计算得出 (只读) 。
- **Link = Off:** 参数值由用户手动指定, 不再跟随 `character` 变化。

### 4.2 手势 (Gesture) 系统

为了解决“程序自动化 (如 Reset) 与用户手动调整”的冲突, v0.0.9 引入了手势系统:

1. **前端:** 用户按下推子触发 `beginGesture`, 释放触发 `endGesture`。
2. **后端:** 维护 `activeGestures` 集合。
3. **判定:**
  - 当参数发生变化时, 检查该 ID 是否在 `activeGestures` 中。

- 是：判定为“用户手动调整”，强制将对应的 `_link` 设为 Off。
- 否：判定为“程序/宏更新”，保持 `_link` 状态不变。

这一机制确保了 "Reset Links" (程序批量修改值) 不会导致参数意外 Unlink。

### 4.3 双向同步

- **Reset Links**: 后端将所有 `_link` 设为 1，计算所有目标值，并通知前端更新 UI。
  - **Link 按钮**: 前端点击 -> 后端监听 `_link` 变化 -> 立即重算该参数值 -> 推送回前端。
- 

## 5 DSP 信号流 (完整)

音频线程入口是 `processBlock()` : [PluginProcessor.cpp](#)

整体链路 (以实现为准) :

1. Stage 1: Clipper (始终在过采样域内处理，且最小 4x)
  2. Stage 2: Limiter (按 Oversampling 选择运行: Off=1x, 2x/4x/8x/16x)
  3. 过采样补偿: 固定 +0.1 dB (位于 Delta 差分前)
  4. Delta 驱动补偿 (仅 Delta 开启) : 抵消 Clipper/Limiter 的 drive 增益
  5. Clipper 延迟对齐 (extra delay)
  6. Delta 差分 (仅 Delta 开启) : 输出 = 处理后 - 对齐干声
  7. Output Gain
  8. True Peak / ISP Safety (必要时 4x 重建 + 安全限幅)
  9. 输出峰值统计、LUFS 统计、GR 计算与 UI 原子通信
- 

## 6 Stage 1: Clipper (Knee 曲线)

实现入口: [Clipper.h](#) (Soft/Hard 实现分别在 [ClipperSoftClipper.h](#)、[ClipperHardClipper.h](#))

阈值固定为 `|x|=1.0` (0 dBFS)，Knee 记为 `k ∈ [0,1]`，过渡起点:

`start = 1 - k`

对每个采样点 (L/R 分别) 执行:

1. 输入增益: `x *= 10^(drive_dB/20)`
2. 对 `a = |x|` 做对称曲线 `y = sign(x) * f(a)`，其中 (仅写正半轴) :

$$f(x) = \begin{cases} x, & x \leq start \\ x - (x - start)^2 / (2k), & start < x < 1 \\ 1, & x \geq 1 \end{cases}$$

上式在 `x=1` 时得到 `f(1)=1-0.5k`，为保证用户体验，内部额外做一个“渐进式自动增益补偿”，确保 `x=1` 时输出仍精确为 1，同时避免在 `x=start` 处引入斜率不连续:

- `g = 1 / (1 - 0.5k)` ( $k > 0$ )
- `t = (x - start) / k` (映射到 0..1)
- `s(t) = t^2 (3 - 2t)` (smoothstep)
- `m(x) = 1 + (g - 1) * s(t)`
- 最终 `f(x) = clamp( (x - (x-start)^2/(2k)) * m(x), 0, 1 )`

3. 线性 GR (用于 UI) : 当发生削波时

`GR_linear = 1 - (|y| / |x|)`, 否则为 0

当 `k=0` 时退化为硬削波: `f(x)=min(x,1)`。

## 6.1 ADAA 抗混叠 (实现口径)

Clipper 的非线性输出不是直接 `y=f(x)`, 而是使用一阶 ADAA (Antiderivative Anti-Aliasing), 在非线性附近显著降低混叠, 尤其在强 Drive/硬拐点时更明显。

实现策略:

- 仅当 `knee > 0` 时启用 ADAA; 硬削波 (`knee=0`) 直接走 `f(x)`。
- 若当前采样与前一采样都处于线性区间 (`|x|` 与 `|xPrev|`  $\leq$  `start`) , 直接返回 `x`, 避免 ADAA 在线性段引入高频滚降。

对每个声道维护 `xPrev` (上一次输入采样)。令非线性为 `f(x)`, 其反导函数为:

`F(x) = ∫ f(x) dx`

则 ADAA 的离散输出为:

- 若 `|x - xPrev|` 很小: `y = f(x)` (避免除零与数值噪声)
- 否则: `y = (F(x) - F(xPrev)) / (x - xPrev)`

本实现对称非线性 `f(x)=sign(x)*g(|x|)` 的 `F(x)` 通过 `integralPositive(|x|)` 构造, 并在 Knee 区间使用预计算的多项式系数 (`updateADAAState()`) 避免在音频线程内做复杂积分。

实现位置:

- ADAA 主逻辑: [processADAA](#)
- 反导与积分构造: [integralPositive / antiderivative](#)
- Knee 区间积分系数预算算: [updateADAAState](#)

## 7 Stage 2: Limiter (Classic / Modern)

Limiter 入口: [Limiter.h](#)

公共步骤:

1. Limiter 输入增益: `x *= 10^(limiter_drive_dB/20)`
2. 根据 `limiter_mode`:

- Classic: `processClassic()`
  - Modern: `processModern()`
3. 1:1 补偿 (若 `one_to_one`) : `x *= 1 / (limiterDrive * clipperDrive)`

## 7.1 Classic 模式 (ADClip 风格的动态阈值 + 硬墙)

实现: [LimiterClassic.h](#)

核心行为:

- `refclip` 会在发生越界时快速下降 (等效“动态阈值”), 并以极慢速回升到 1.0
- 最终硬墙: `x = clamp(x, -refclip, +refclip)`
- 输出乘以 `ceiling_linear`
- 线性 GR 用 `rawInput` 与 clamp 后的 `inputSample` 计算:

`GR_linear = 1 - (|out|/|in|)` (若 `linl` 足够大且发生压缩)

## 7.2 Modern 模式 (前瞻 + 双包络 + Hold + 平滑 + 硬墙)

实现: [LimiterModern.h](#)

Modern 每个采样的逻辑可按 6 个阶段理解:

### 7.2.1 Phase 1: Lookahead (前瞻缓存)

- 将当前输入写入环形缓冲 `lookaheadBuffer*`
- 读出延迟 `lookaheadSize-1` 的 `delayed*` 作为“最终被处理并输出的信号”

缓冲长度由 `lookahead_ms` 与 (当前 Limiter 实际采样率=原始采样率×过采样倍率) 决定。Classic 模式也使用同一延迟线, 仅用于对齐输出延迟。

### 7.2.2 Phase 2: Sidechain 预处理

- 计算一阶 HPF 状态 (`advScHpfHz`), 用于估计 `hfRatio` (高频占比), 以驱动自适应 Release (低频占比越高, 释放越慢)
- 检测信号使用宽带峰值: `det = abs(input)`。当 True Peak 开启时, Limiter 运行在过采样域, 因此 `det` 等价于“重建后采样点”的峰值口径, 用于保证最终 `ceiling` 安全

实现见: [Limiter.h](#)

### 7.2.3 Phase 3: Overshoot(dB) + Dual Envelope (双包络)

- 若 `det > 1.0`, 则 `overshoot_dB = 20*log10(det)`, 否则为 0
- Fast/Slow envelope 保存的是“需要衰减的 dB 值”, 用指数系数衰减:

`env = max(env, overshoot_dB)` 或 `env *= coeff`

其中 `coeff = exp(-1/(T*Fs))`, `T` 由 `releaseFastMs/releaseSlowMs` 进一步被低频占比与当前 GR 强度自适应拉长。

### 7.2.4 Phase 4: 瞬态/延音判定 (Ratio 逻辑)

- 计算 `ratio = fast/slow`
- 计算门限: `ratioThresh = ratioBase * (1 - character*ratioSlope)`
- 若 `ratio > ratioThresh`, 认为更像瞬态:

```
atten_dB = max(fast, slow * transientMix)
```

否则:

```
atten_dB = max(fast, slow)
```

## 7.2.5 Phase 5: Stereo Link (立体声链接)

- Modern: 分别对 fast/slow 以 `linkTransients/linkRelease` 向左右最大值收敛
- Classic (fallback 分支) : 对衰减量 `atten` 用 `linkAmount` 链接

## 7.2.6 Phase 6: Hold + Gain Smoothing + Output

1. Peak Hold: 对 `atten_dB` 做“保持最大衰减 N ms + 指数回弹”
  2. 平滑 (两级平滑器) :
    - Attack 系数由 `attackTau = lookahead/tauDiv` 得到
    - Release 系数由 `releaseSmoothBase + releaseSmoothRange * (lowBias*grNorm)` 得到
  3. dB → 线性增益: `gain = 10^(-smoothGain_dB/20)`
  4. 对 `delayed*` 应用增益后再乘以 `ceiling`
  5. Soft safety (可调强度) + 最终硬墙 clamp ( $\pm ceiling$ )
  6. GR (用于 UI) 按增益计算: `GR_linear = 1 - gain`
- 

## 8 过采样 (Oversampling) 与延迟

实现: [PluginProcessor.cpp](#)

- Clipper 始终在过采样域内运行, 且最小为 4x:
    - 当用户选择 Off 或 2x 时: Clipper 先以 4x 处理, 随后回到宿主采样率, 再让 Limiter 按用户所选倍率运行 (1x 或 2x)。
    - 当用户选择 4x/8x/16x 时: Clipper 与 Limiter 共享同一次上采样/下采样, 在同一过采样域内串联运行。
  - 插件延迟 (latency) 由“过采样滤波群延迟 + lookahead 延迟”组成:
    - 共享同一次过采样时: `oversamplingLatency = oversampler(selected).getLatencyInSamples()`
    - 分两段过采样时: `oversamplingLatency = oversampler(clipper=4x).getLatencyInSamples() + oversampler(limiter=selected).getLatencyInSamples()`
    - `lookaheadLatency = lookahead_ms * Fs / 1000`
    - `totalLatency = oversamplingLatency + lookaheadLatency`
    - 并通过 `setLatencySamples(totalLatency)` 上报宿主, 保证 Classic/Modern 与不同 oversampling 组合下延迟一致。
- 

## 9 Meter 与 LUFS (实现口径)

## 9.1 GR (Clipper/Limiter)

- DSP 内部以线性 GR (0..1) 累计每 block 的最大值
- block 结束后将 `GR_linear` 转成 dB:

`ratio = 1 - GR_linear`, `GR_dB = 20*log10(ratio)` (负值, 越负代表压得越多)

见 [PluginProcessor.cpp](#)

- 写入方式是“原子保持最小值”（最负的 dB）：用于 UI 低频刷新也能捕捉瞬态。

## 9.2 Peak Hold (UI 上的保持线)

UI 上那条保持线由 Editor 的 `timerCallback()` 生成:

- 读取一次 `clipperGR*/limiterGR*` 原子值并平滑
- 维护 `hold*`: 当 GR 更负时立即更新, 否则以固定回弹速度回到 0

见 [PluginEditor.cpp](#)

## 9.3 输出峰值 (当前 UI 不显示 OUT bar)

- Processor 统计 block 内 `max(abs(sample))`, 并原子存储“自上次读取以来的最大值”
- Editor 将线性峰值转 dB (-100..0) 并平滑后通过 web IPC 推送

见 [PluginProcessor.cpp](#)、[PluginEditor.cpp](#)

## 9.4 LUFS (EBU R128)

- `Ebu128LoudnessMeter` 在 `prepareToPlay()` 初始化并在 `processBlock()` 处理
- Editor 读取 Integrated / Short-term / Momentary 并推送到 UI
- Reset 按钮会调用 `lufsMeter.reset()`

见 [PluginProcessor.cpp](#)、[PluginEditor.cpp](#)

---

# 10 实现备注与实时性风险 (排查清单)

以下是当前实现的“事实口径”，包含已知潜在的实时性风险点，在低 Buffer/高采样率或多实例场景下需注意：

### 1. 动态内存分配风险:

- Modern 模式下, 当 `lookahead` 参数变化时, 会触发 `std::vector::assign()` 重新分配 buffer。这发生在音频线程内, 理论上属于非实时安全操作 (RT-unsafe) 。
- 优化方向: 未来可改为预分配最大 buffer 或使用无锁环形缓冲池。
- 实现见: [Limiter.h](#)

### 2. CPU 负载波动风险:

- True Peak 模式下的后置 4x 重建检测会为每个采样点增加少量标量运算。
- 在 CPU 紧张时, 建议优先通过关闭 True Peak 来减负。

## 11 True Peak (实现口径)

True Peak 相关逻辑分两部分：

### 1. 限制器内部 Ceiling 余量 (margin)

- 当 `true_peak == On` 时，Limiter 会把用户 `ceiling` 再额外下调一小段作为内部工作天花板（当前实现是 `-0.2 dB`）。
- 目的：下采样/滤波可能产生轻微的峰值回长（peak regrowth），提前留 margin 让最终输出更稳地不超出用户 ceiling。
- 实现见：[Limiter::setParameters](#)。

### 2. True Peak 检测口径 (过采样域重建)

- `true_peak == On` 时，Limiter 的峰值检测 `det = abs(input)` 按“当前 Limiter 实际采样率”（由 `oversampling` 决定）运行；当 Limiter 处于过采样域时，该口径对应重建后的采样点峰值。
- 实现见：[processBlock](#)、[Limiter::processModern](#)

### 3. True Peak 后置 4x 重建限幅 (默认启用)

- 当 `true_peak == On` 且 Limiter 的 `oversampling < 4x` 时，Limiter 保持用户选择的过采样倍率运行；随后 Processor 会在输出端追加一次 4x 重建检测，并在发现超过 `ceiling` 时对当前 block 施加安全增益以兜底不超标（用于处理下采样/滤波造成的 peak regrowth）。
- 实现见：[processBlock](#)

---

## 12 True Peak 与过采样策略 (实现口径)

过采样由 Processor 负责分配与切换（并上报延迟），True Peak 会影响“最低过采样倍率”的策略：

- `oversampling` 参数含义：`0=Off(1x), 1=2x, 2=4x, 3=8x, 4=16x`
- Limiter 的实际过采样倍率：
  - True Peak Off：按用户选择运行（允许 1x）
  - True Peak On：按用户选择运行；当 `oversampling < 4x` 时会追加后置 4x 重建限幅兜底
- Clipper 的实际过采样倍率：
  - 若 Clipper 未旁路：始终最低 4x（即 `index>=2`）
  - 若 Clipper 旁路：Clipper 不额外强制倍率（跟随 Limiter）

实现见：

- `prepareToPlay()` 初始化与策略计算：[PluginProcessor.cpp](#)
- `processBlock()` 中的动态切换策略：[PluginProcessor.cpp](#)

### 12.1 过采样滤波器选择 (性能口径)

当前 Oversampling 初始化使用 JUCE 的半带 FIR Linear Phase 方案 (`filterHalfBandFIREquiripple`)，目的是在 True Peak 开启时确保波形重构的相位线性与精确性。

实现见：[PluginProcessor.cpp](#)