A Stable Implementation of the Adaptive Rejection Sampler in MATLAB

Kar Wai Lim*
National University of Singapore
karwai.lim@nus.edu.sg

November 28, 2018

Abstract

We provide a tutorial on the derivative-based adaptive rejection sampler with an illustrative example and present a stable implementation of the sampler in MATLAB. The stable implementation avoids the issue of numerical instability that may appear in a naïve implementation of ARS. To validate our implementation, we perform one-sample Kolmogorov-Smirnov test for samples generated from normal distribution and gamma distribution. We find that our code produces accurate samples whereby several existing implementations fail.

1 Adaptive Rejection Sampling

Adaptive rejection sampling (ARS) [Gilks and Wild, 1992] is a form of rejection sampling where we propose a sample from a simpler distribution (called the envelope distribution) and then accept/reject the sample. The ARS uses an efficient envelope distribution and allows the envelope distribution to adapt and become better over time. Refer to Gilks and Wild [1992] for details.

We study the *derivative*-based ARS in this report, which requires the derivative of the log likelihood of the target distribution to be evaluated. However, we like to note that an alternative called the derivative free ARS exists, which may or may not be more efficient (depends on the cost of computing the derivative).

The aim of the ARS is to generate samples from a distribution where its likelihood (or probability density function) is continuous and log-concave. Notable examples include the normal distribution (Gaussian distribution) and exponential distribution.

2 Derivative-based ARS

Derivative-based ARS makes use of the derivative of the log likelihood in determining the envelope distribution. It requires the following assumptions to be met (adapted from Kohnen[1]):

1. The likelihood up to a proportionality constant, denoted $g(x)$, is continuous and differentiable everywhere in its domain $x \in D$.
2. The likelihood $g(x)$ is log-concave, or equivalently, its log, $h(x) = \log g(x)$, is concave everywhere in D. This means that the second derivative $h''(x)$ is negative everywhere in D.

[*] This work is undertaken while the author was in the Australian National University and CSIRO's Data61.
[1] http://stat.duke.edu/~cnk/Links/slides.pdf

Efficiency of ARS:

1. ARS tends to position the evaluations of $h(x)$ and $h'(x)$ optimally, because new evaluations are most likely to occur at values of x where the rejection envelope and squeezing functions are most discrepant.
2. Using two starting points were found to be sufficient for computational efficiency.
3. Empirically, the number of evaluations of $h(x)$ and $h'(x)$ required to sample n points from the target distribution $f(x)$ (normalised pdf of $g(x)$) increases approximately in proportion to $n^{\frac{1}{3}}$, even for very non-normal densities.

2.1 Demonstration of ARS

Here, we detail the sampling algorithm for ARS. We assume that the following are given: (1) $g(x)$ the likelihood (up to a proportionality constant) that we wish to sample from, (2) log likelihood $h(x) = \log g(x)$, and (3) derivative of the log likelihood $h'(x) = \frac{\mathrm{d}}{\mathrm{d}x} h(x)$. We note that $g(x)$ needs to satisfy log-concavity for all x, that is, $h''(x) < 0$ for all x.

Initialisation. The sampling algorithm starts by choosing two points x_1 and x_2 such that $h'(x_1) > 0$ and $h'(x_2) < 0$, that is, x_1 is located to the left of the mode and x_2 is located on the right. We then compute their gradients $h'(x)$ and connect their tangent lines, which form the envelope function. The point of intersection is given by

$$z_1 = \frac{h(x_2) - h(x_1) - x_2 h'(x_2) + x_1 h'(x_1)}{h'(x_1) - h'(x_2)}$$

The corresponding piecewise upper hull function $u(x)$ (for the log likelihood) is thus

$$u(x) = \begin{cases} h(x_1) + (x - x_1)h'(x_1) & \text{for } z_0 < x < z_1 \\ h(x_2) + (x - x_2)h'(x_2) & \text{for } z_1 \leq x < z_2 \end{cases}$$

where $z_0 = -\infty$ and $z_2 = \infty$ corresponds to the boundary of the support of the distribution of interest. The corresponding envelope function is thus $\exp u(x)$. The upper hull function and the envelope function are represented by the red lines in the plots below. We note that the upper hull function must always be greater or equal to the log likelihood, such that the following condition $u(x) \geq h(x)$ is satisfied.

Next, we define the squeezing function as $\exp l(x)$, where $l(x)$ is the piecewise lower hull for $h(x)$. Note that the condition $l(x) \leq h(x)$ needs to be satisfied. The lower hull is formed by joining the two initial points x_1 and x_2 by a straight line. For the support x that are not within $[x_1, x_2]$, we simply define $l(x) = -\infty$. The lower hull function is given by

$$l(x) = \begin{cases} -\infty & \text{for } x < x_1 \\ \dfrac{(x_2 - x)h(x_1) + (x - x_1)h(x_2)}{x_2 - x_1} & \text{for } x_1 \leq x < x_2 \\ -\infty & \text{for } x_2 \leq x \end{cases}$$

We note that the lower hull $l(x)$ and the squeezing function $\exp l(x)$ are represented by the green lines in the plots below.

Example. We illustrate the above with a standard normal distribution $\mathrm{Normal}(\mu = 0, \sigma^2 = 1)$. The likelihood (up to a proportionality constant), log likelihood and its derivative are:

$$\begin{aligned} g(x) &= \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2} \right) \\ h(x) &= -\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2} \\ h'(x) &= -\frac{x-\mu}{\sigma^2} \end{aligned}$$

Note that $h''(x) = -1/\sigma^2 < 0$ for all x, which confirms that the likelihood function is log-concave.

We choose starting point $x_1 = -1$ and $x_2 = 2$, with

$$h(x_1) = -0.5 \qquad h(x_2) = -2$$
$$h'(x_1) = 1 \qquad h'(x_2) = -2$$

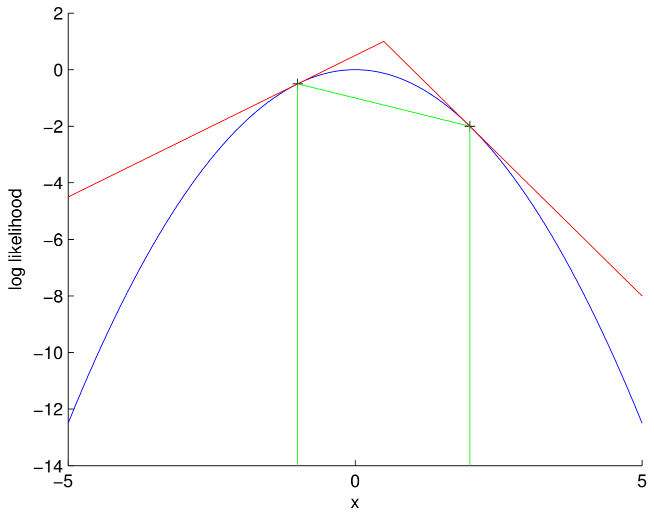
and we work out the upper hull function as

$$u(x) = \begin{cases} x + 0.5 & \text{for } -\infty < x < 0.5 \\ -2\,x + 2 & \text{for } 0.5 \le x < \infty \end{cases}$$

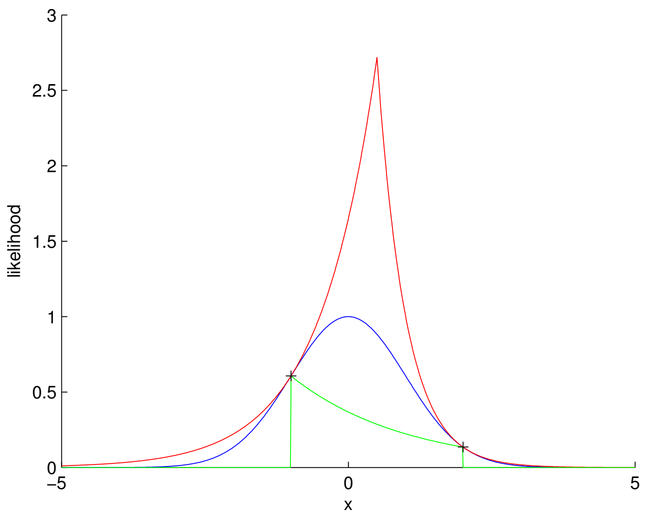
Note that the intersection of the two tangents located at $(0.5, 1)$. Similarly, we can work out the lower hull function, as follows:

$$l(x) = \begin{cases} -\infty & \text{for } x < -1 \\ -0.5\,x - 1 & \text{for } -1 \le x < 2 \\ -\infty & \text{for } 2 \le x \end{cases}$$

Below, we illustrate the upper hull $u(x)$ (red line) and the lower hull $l(x)$ (green line) superimposed on the log likelihood plot of $h(x)$ (blue line). We note that both the upper hull and the lower hull are piecewise linear functions.



Taking exponential, we get the corresponding likelihood plot $g(x)$ (blue line), with the envelope function $\exp u(x)$ (red line) and the squeezing function $\exp l(x)$ (green line). The envelope function and the squeezing function are piecewise exponential functions. We note that the envelope function and the squeezing function gets better as we perform more evaluation of the log likelihood.



Propose a sample. We now move on to the generation of samples through the acceptance/rejection sampling method using the piecewise exponential envelope function. The probability density function used for proposing a sample can be obtained by normalising the envelope function (so that the area under the curve is one):

$$s(x) = \frac{\exp u(x)}{\int_{-\infty}^{\infty} \exp u(y)\,\mathrm{d}y}$$

The normalising constant, denoted as Q, (assuming $h'(x_i) \neq 0$) can be evaluated as

$$\begin{aligned}
\int_{-\infty}^{\infty} \exp u(y)\,\mathrm{d}y &= \int_{-\infty}^{z_1} \exp\Big(h(x_1) + (y - x_1)h'(x_1)\Big)\,\mathrm{d}y + \int_{z_1}^{\infty} \exp\Big(h(x_2) + (y - x_2)h'(x_2)\Big)\,\mathrm{d}y \\
&= \left[\frac{1}{h'(x_1)} \exp\Big(h(x_1) + (y - x_1)h'(x_1)\Big)\right]_{-\infty}^{z_1} + \left[\frac{1}{h'(x_2)} \exp\Big(h(x_2) + (y - x_2)h'(x_2)\Big)\right]_{z_1}^{\infty} \\
&= \frac{1}{h'(x_1)}\Big(\exp u(z_1) - \lim_{x\to-\infty} \exp u(x)\Big) + \frac{1}{h'(x_2)}\Big(\lim_{x\to\infty} \exp u(x) - \exp u(z_1)\Big) \\
&= \left(\frac{1}{h'(x_1)} - \frac{1}{h'(x_2)}\right) \exp u(z_1)
\end{aligned}$$

Noting that for the normal likelihood the upper hull function $u(x)$, its limit toward ∞ and $-\infty$ is

$$\lim_{x\to-\infty} u(x) = -\infty$$
$$\lim_{x\to\infty} u(x) = -\infty$$

Thus, the limits for the envelope function become

$$\lim_{x\to-\infty} \exp u(x) = 0$$
$$\lim_{x\to\infty} \exp u(x) = 0$$

After computing the normalising constant, $Q = \int_{-\infty}^{\infty} \exp u(x)\,\mathrm{d}x$, we can compute the cumulative density function (cdf):

$$c(x) = \frac{1}{Q}\int_{-\infty}^{x} \exp u(y)\,\mathrm{d}y$$

which will be used for sampling (*via* the inverse cdf method). Note that the cdf is a piecewise exponential function:

$$c(x) = \begin{cases} \dfrac{1}{Q}\dfrac{1}{h'(x_1)}\exp u(x) & \text{for } z_0 < x < z_1 \\ \dfrac{1}{Q}\left[\dfrac{1}{h'(x_1)}\exp u(z_1) + \dfrac{1}{h'(x_2)}\Big(\exp u(x) - \exp u(z_1)\Big)\right] & \text{for } z_1 \le x < z_2 \end{cases}$$

To generate a sample x from the cdf $c(x)$, we sample a random standard uniform variable w_1 and find x such that $c(x) = w_1$. That is,

$$x^* = c^{-1}(w_1)$$

is a sample from the cdf. For instance, if w_1 falls in the first piece of the cdf, then

$$x^* = x_1 + \frac{\log\big(Q\,w_1 h'(x_1)\big) - h(x_1)}{h'(x_1)}$$

Otherwise if w_1 falls into the second piece then

$$x^* = x_2 + \frac{\log\big(e_2\big) - h(x_2)}{h'(x_2)}$$

where

$$e_2 = h'(x_2)\left[Q\,w_1 - \frac{1}{h'(x_1)}\exp u(z_1)\right] + \exp u(z_1)$$

Example. To illustrate using the above curated example, the normalising constant can be computed to be $Q = 4.0774$. The cdf can then be derived as

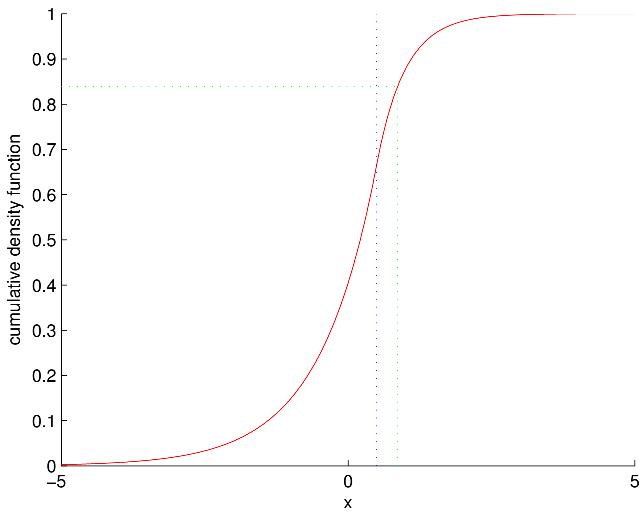
$$c(x) = \begin{cases} \dfrac{1}{Q}\exp\big(x + 0.5\big) & \text{for } -\infty < x < 0.5 \\ \dfrac{1}{Q}\left[e + -0.5\Big(\exp\big(-2\,x + 2\big) - e\Big)\right] & \text{for } 0.5 \le x < \infty \end{cases}$$

The cdf is presented as the red line in the plot below. The piecewise boundary is separated by the dotted blue line.

Now, to simulate a sample from the cdf, we draw a random variable from the standard uniform distribution. As an example, say we sample $w_1 = 0.8389$. From the cdf plot, we can see that this value (on y-axis) corresponds to the second piece of the cdf, which gives

$$x^* = 0.8635$$

This is shown by the green dotted line below.



Accept/reject the sample. After sampling x^*, we perform an acceptance/rejection test to see if the sample x^* is to be rejected. We note that the existence of the squeezing function is to improve the efficiency for the rejection test, especially if the log likelihood $h(x)$ requires high computational cost. To illustrate, the acceptance probability for accepting the sample x^* is given by

$$A(x^*) = \frac{\exp h(x^*)}{\exp u(x^*)} = \exp\big(h(x^*) - u(x^*)\big)$$

and we accept the sample x^* if a generated standard uniform variable w_2 is less than the acceptance probability:

$$w_2 < A(x^*)$$

Since the squeezing function $\exp l(x)$ is always smaller than $\exp h(x)$, we can first create a *squeezing test* (quick to evaluate)

$$w_2 < B(x^*) = \frac{\exp l(x^*)}{\exp u(x^*)} = \exp\big(l(x^*) - u(x^*)\big)$$

to test for acceptance before proceeding to the acceptance/rejection test which requires the computation of the log likelihood $h(x)$.

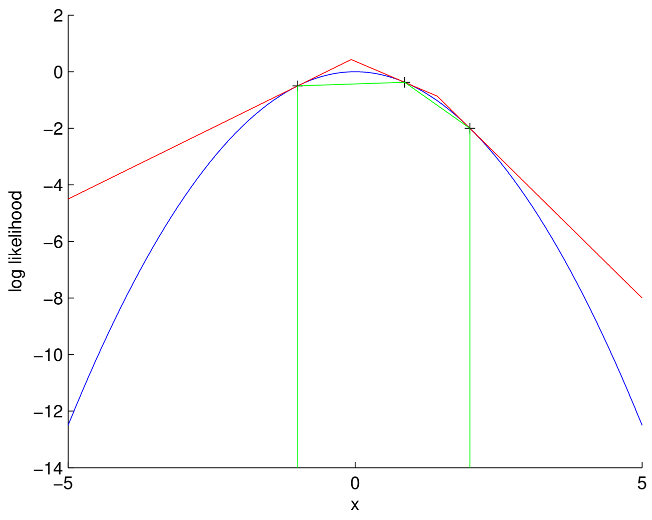
Example. Continuing with our example, we first sample $w_2 = 0.4789$ from a standard uniform distribution and perform the squeezing test. The squeezing ratio for the sampled $x^* = 0.8635$ is $B(x^*) = 0.1818$, since w_2 is greater than the squeezing ratio, we fail the squeezing test and proceed with the acceptance/rejection test.

To perform the acceptance/rejection test, we compute $h(x^*) = -0.3728$ (note that this value will be used later, so will be stored) and compute the acceptance probability $A(x^*) = 0.5242$. Since $w_2 < A(x^*)$, we accept the sample x^* as generated from the likelihood $g(x)$.

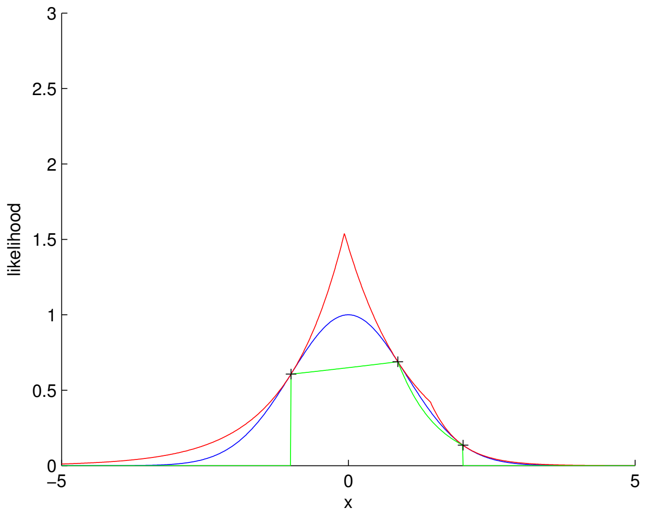
Code. The code to perform the above ARS demonstration, including the plotting of the relevant diagrams, is presented in Code Appendix A.

Update the Envelope and Squeezing Function. Now, since we have evaluated the log likelihood $h(x^*)$, we can use this information to improve the envelope function and the squeezing function.

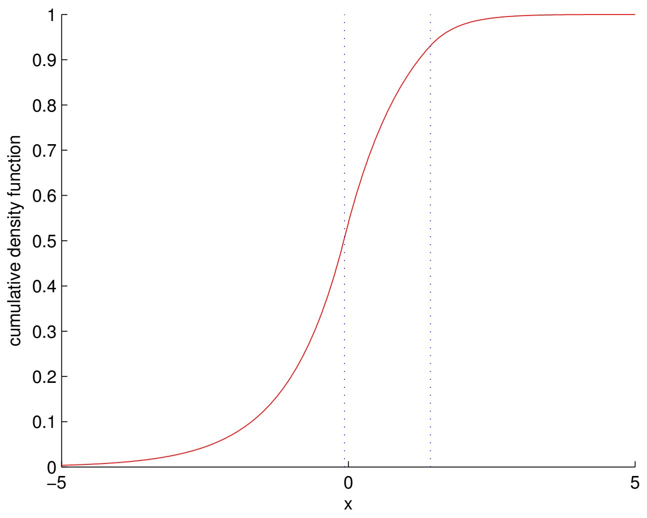
The updated plots are displayed below. As expected, the upper hull function (red) and the lower hull function (green) are now closer to the log likelihood function (blue). Note that we now have three points in the plot.



The likelihood function is plotted using the same scale as the above likelihood plot (when we use two starting points). We can see that the introduction of one extra point drastically improves the envelope function (red) and the squeezing function (green). Note, since points are introduced whenever we evaluate the log likelihood (and its derivative), and that is when the proposed sample fails the squeezing test, the introduction of new point will tend to be where the discrepancy between the upper hull and lower hull is high. This gives rise to an efficient update. “The method tends to space evaluations of $h(x)$ and $h'(x)$ optimally” [Gilks and Wild, 1992].



The updated cdf has thus become a three pieces piecewise exponential functions, as illustrated below. We note that the cdf converges to the true distribution when the number of points tend to infinity.



Code. The code to produce the updated envelope function and squeezing function is in presented Code Appendix B.

3 Algorithm for ARS

We implemented two versions of the ARS algorithm in MATLAB. The first one utilises efficient methods to update the various statistics used in the algorithm, giving a very fast sampler. However, this algorithm may encounter numerical overflow (numbers too large to be stored in the computer and treated as ∞ instead) or underflow (numbers are too small that the precision of the computer is not sufficient, thus treated as zero) if the log derivatives are high in magnitude and the initial points are chosen appropriately.

The second implementation addresses the numerical instability, however, at the cost of sacrificing some efficiency, thus slower. Fortunately, this implementation is very robust and is thus more suitable to be used in Gibbs sampling for which the posterior distributions can vary every iteration and can be unstable. We note that both versions produce the *same* samples given the same random seed.

3.1 Efficient Sampler

We first describe the efficient sampler, which is a relatively straight forward implementation of the ARS. The inputs to the algorithm are

1. Required number of samples N.
2. A concave log likelihood function $h(x)$.
3. Derivative of the log likelihood function $h'(x)$.
4. Left and right boundary of the support for x, denoted as x^- and x^+.
5. Initial points for constructing the upper and lower hull, denoted as $x_1, \ldots, x_M$.

Several conditions to ensure the functionality of the algorithm are as follows:

1. The log likelihood $h(x)$ is concave, that is, $h''(x) < 0$ for all x.
2. At least 2 initial points are given, that is, $M \geq 2$.
3. The initial points are within bounds, that is, $x^- < x_m < x^+$ for $m = 1, \ldots, M$.
4. The initial points are not repeated, that is, $x_i \neq x_j$ for $i = 1, \ldots, M$; $j = 1, \ldots, M$; and $i \neq j$.
5. The left most initial point has positive derivative and the right most initial point has negative derivative, that is, $h'(x_1) > 0$ and $h'(x_M) < 0$.
6. The initial points have non-zero derivative, that is, $h'(x_m) \neq 0$ for $m = 1, \ldots, M$.

Now we describe the algorithm in details below.

Initialisation Step. We initialise a vector of $\mathbf{x}$ as follows

$$\mathbf{x} = [\mathbf{x}_0, \ldots, \mathbf{x}_{M+1}] \coloneqq [x^-, \mathrm{sort}(x_1, \ldots, x_M), x^+]$$

and then renaming the points to $x_0, x_1, \ldots, x_M, x_{M+1}$. Note that this means $x_0 = x^-$, $x_{M+1} = x^+$, and x_i are in ascending order. In addition, we also evaluate the log likelihood and the associated derivative of the log likelihood of each point in $\mathbf{x}$. These values are stored (cached) as $\mathbf{h}$ and $\mathbf{v}$ respectively.

$$\begin{aligned}
\mathbf{h} &= [\mathbf{h}_0, \ldots, \mathbf{h}_{M+1}] \coloneqq [h(x_0), h(x_1), \ldots, h(x_M), h(x_{M+1})] \\
\mathbf{v} &= [\mathbf{v}_0, \ldots, \mathbf{v}_{M+1}] \coloneqq [h'(x_0), h'(x_1), \ldots, h'(x_M), h'(x_{M+1})]
\end{aligned}$$

Note that for $x_1, \ldots, x_M$, we can compute the *tangents* of the log likelihood function $h(x)$. These tangents are linear equations of the form

$$\mathrm{tangent}_i(x) = h(x_i) + (x - x_i)\, h'(x_i)$$

Next, we construct a vector of $\mathbf{z}$, for which each value corresponds to the intersection point of the tangents for each point in $\mathbf{x}$. These intersections are obtained by solving $\mathrm{tangent}_i(x) = \mathrm{tangent}_{i+1}(x)$. Mathematically,

$$\mathbf{z} = [\mathbf{z}_0, \ldots, \mathbf{z}_M] \coloneqq [z_0, \ldots, z_M]$$

where $z_0 = x_0 = x^-$ and $z_M = x_{M+1} = x^+$ and the z_i (for $i = 1, \ldots, M-1$) satisfies

$$\begin{aligned}
z_i &= \frac{h(x_{i+1}) - h(x_i) - x_{i+1}h'(x_{i+1}) + x_i h'(x_i)}{h'(x_i) - h'(x_{i+1})} \\
&= \frac{\mathbf{h}_{i+1} - \mathbf{h}_i - \mathbf{x}_{i+1}\mathbf{v}_{i+1} + \mathbf{x}_i\mathbf{v}_i}{\mathbf{v}_i - \mathbf{v}_{i+1}} \qquad (1)
\end{aligned}$$

Note that $\mathbf{z}$ has one less element compared to $\mathbf{x}$, that is, $\texttt{length}(\mathbf{z}) = \texttt{length}(\mathbf{x}) - 1$.

We then construct the upper hull function $u(x)$, which is a piecewise linear function of the tangents that 'envelope' the log likelihood function. This piecewise function can be seen as the minimum of all tangent lines:

$$\begin{aligned} u(x) &= \min_{i=1,\ldots,M} \text{tangent}_i(x) \\ &= \sum_{i=1}^{M} \text{tangent}_i(x)\, I(z_{i-1} \le x < z_i) \\ &= \sum_{i=1}^{M} \Big[h(x_i) + (x - x_i)\, h'(x_i)\Big]\, I(z_{i-1} < x \le z_i) \end{aligned} \tag{2}$$

Note that $u(x) \ge h(x)$ for all x.

For computational efficiency, we store the value of $u(x)$ evaluated at each point in $\mathbf{z}$:

$$\mathbf{u} = [\mathbf{u}_0, \ldots, \mathbf{u}_M] \coloneqq [u(z_0), \ldots, u(z_M)]\,.$$

In addition, we also cache their exponents

$$\mathbf{e}^u = [\mathbf{e}^u_0, \ldots, \mathbf{e}^u_M] \coloneqq \Big[e^{u(z_0)}, \ldots, e^{u(z_M)}\Big]\,.$$

The lower hull function $l(x)$ is given by straight lines connecting each consecutive points $x_1, \ldots, x_M$. For instance, the linear equation connecting x_{i-1} and x_i (for $i = 2, \ldots, M$) can be derived as

$$\begin{aligned} l_i(x) &= h(x_{i-1}) + (x - x_{i-1})\frac{h(x_i) - h(x_{i-1})}{x_i - x_{i-1}} \\ &= \frac{x_i\, h(x_{i-1}) + x_{i-1}\, h(x_{i-1}) + x\, h(x_i) - x_{i-1}\, h(x_i) - x\, h(x_{i-1}) + x_i\, h(x_{i-1})}{x_i - x_{i-1}} \\ &= \frac{(x_i - x)\, h(x_{i-1}) + (x - x_{i-1})\, h(x_i)}{x_i - x_{i-1}} \end{aligned}$$

For the region to the left of x_1 and to the right of x_M, we simply set $l(x)$ to be $-\infty$. The lower hull function is thus a piecewise linear function and can be written as

$$l(x) = \begin{cases} -\infty & \text{for } x_0 < x \le x_1 \\ \dfrac{(x_i - x)\, h(x_{i-1}) + (x - x_{i-1})\, h(x_i)}{x_i - x_{i-1}} & \text{for } x_{i-1} < x \le x_i; i = 2, \ldots, M \\ -\infty & \text{for } x_M < x \le x_{M+1} \end{cases}$$

Note that we do not explicitly store the lower hull function, and instead computing them as needed.

The envelope function and the squeezing function are computed as required. Note that they are given by $\exp u(x)$ and $\exp l(x)$ respectively. To sample a value from the envelope function, we employ the inverse cumulative density function (CDF) method, which requires the CDF of the proposal distribution, $s(x)$, that is proportional to $\exp u(x)$. The proposal distribution $s(x)$ can be obtained by normalisation, as follows:

$$s(x) = \frac{\exp u(x)}{\int_{x^-}^{x^+} \exp u(y)\, \mathrm{d}y}$$

We denote the normalisation constant as Q, and can be computed as

$$\begin{aligned}
Q &= \int_{x^-}^{x^+} \exp u(y)\,\mathrm{d}y \\
&= \int_{x^-}^{x^+} \sum_{i=1}^{M} \exp\Big[h(x_i) + (y - x_i)\,h'(x_i)\Big]\, I(z_{i-1} < y \leq z_i)\,\mathrm{d}y \\
&= \sum_{i=1}^{M} \int_{x^-}^{x^+} \exp\Big[h(x_i) + (y - x_i)\,h'(x_i)\Big]\, I(z_{i-1} < y \leq z_i)\,\mathrm{d}y \\
&= \sum_{i=1}^{M} \int_{z_{i-1}}^{z_i} \exp\Big[h(x_i) + (y - x_i)\,h'(x_i)\Big]\,\mathrm{d}y \\
&= \sum_{i=1}^{M} \left[\frac{1}{h'(x_i)} \exp\Big[h(x_i) + (y - x_i)\,h'(x_i)\Big]\right]_{z_{i-1}}^{z_i} \\
&= \sum_{i=1}^{M} \frac{1}{h'(x_i)}\Big(\exp u(z_i) - \exp u(z_{i-1})\Big) \qquad (3)
\end{aligned}$$

Note that each part in the summation in Equation (3), denoted as $\mathbf{q}_i$ (for $i = 1, \dots, M$), can be evaluated efficiently using the cached values:

$$\begin{aligned}
\mathbf{q}_i &:= \frac{1}{h'(x_i)}\Big(\exp u(z_i) - \exp u(z_{i-1})\Big) \\
&= \frac{1}{\mathbf{v}_i}\big(\mathbf{e}_i^u - \mathbf{e}_{i-1}^u\big) \qquad (4)
\end{aligned}$$

We store these values as

$$\mathbf{q} = [\mathbf{q}_1, \dots, \mathbf{q}_M]$$

After computing $Q = \sum_i \mathbf{q}_i$, we can derive the CDF $c(x)$ of the proposal distribution $s(x)$.

$$\begin{aligned}
c(x) &= \int_{x^-}^{x} s(y)\,\mathrm{d}(y) \\
&= \frac{1}{Q}\int_{x^-}^{x} \exp u(y)\,\mathrm{d}(y) \\
&= \frac{1}{Q}\left[\int_{z_k}^{x} \exp u(y)\,\mathrm{d}(y) + \sum_{i=1}^{k} \mathbf{q}_i\right] \\
&= \frac{1}{Q}\left[\frac{1}{h'(x_{k+1})}\Big(\exp u(x) - \exp u(z_k)\Big) + \sum_{i=1}^{k} \mathbf{q}_i\right]
\end{aligned}$$

where $k = \sup\{i \,|\, z_i < x\}$ is the largest index k such that $z_k < x$ is satisfied. We store the unnormalised CDF, $Q\,c(x)$, evaluated at each point in $\mathbf{z}$ to speed up computation, this is given by

$$\mathbf{Qc} = [\mathbf{Qc}_0, \dots, \mathbf{Qc}_M] := [Q\,c(z_0), \dots, Q\,c(z_M)] \qquad (5)$$

(defining $c(z_0) = 0$) and can easily be computed using the `cumsum` function in MATLAB, that is, $\mathbf{Qc} = \texttt{cumsum}([0, \mathbf{q}])$, since

$$\begin{aligned}
\mathbf{Qc}_j = Q\,c(z_j) &= \frac{1}{h'(x_j)}\Big(\exp u(z_j) - \exp u(z_{j-1})\Big) + \sum_{i=1}^{j-1} \mathbf{q}_i \\
&= \mathbf{q}_j + \sum_{i=1}^{j-1} \mathbf{q}_i \\
&= \sum_{i=1}^{j} \mathbf{q}_i \qquad (6)
\end{aligned}$$

Noting that $k = \sup\{i \,|\, z_i < z_j\} = j - 1$ above.

Sampling Step. With the CDF defined, we can now sample a value x^* using the inverse CDF method. This sampling routine is implemented in a loop until N number of samples are *accepted*.

We first sample a random number w_1 from the standard uniform distribution $\mathrm{U}(0, 1)$, then find the value x^* that satisfies (inverse CDF method)

$$c(x^*) = w_1$$

The solution to the above equation is simply

$$x^* = c^{-1}(w_1)$$

where $c^{-1}(x)$ is the inverse CDF, which is also a piecewise function. To solve for x^*, we find k such that $k = \sup\{i \,|\, z_i < c^{-1}(w_1)\}$ is satisfied. This gives

$$\begin{aligned} k &= \sup\{i \,|\, z_i < c^{-1}(w_1)\} \\ &= \sup\{i \,|\, c(z_i) < w_1\} \\ &= \inf\{i \,|\, c(z_{i+1}) \geq w_1\} \\ &= \inf\{i \,|\, w_1 \leq c(z_{i+1})\} \\ &= \inf\{i \,|\, Q\, w_1 \leq Q\, c(z_{i+1})\} \end{aligned}$$

In particular, we iterate over $i = 0, 1, 2, \ldots$ to find the first index i such that the inequality $Q\, w_1 \leq Q\, c(z_{i+1})$ is satisfied. Note that $Q\, c(z_j)$ is cached as $\mathbf{Qc}_j$ as defined in Equation (5). After having k, before solving for x^*, we compute $u(x^*)$:

$$\begin{aligned} c(x^*) &= w_1 \\ \frac{1}{Q}\left[\frac{1}{h'(x_{k+1})}\Big(\exp u(x^*) - \exp u(z_k)\Big) + \sum_{i=1}^{k} \mathbf{q}_i\right] &= w_1 \\ \frac{1}{h'(x_{k+1})}\Big(\exp u(x^*) - \exp u(z_k)\Big) &= Q\, w_1 - \sum_{i=1}^{k} \mathbf{q}_i \\ u(x^*) &= \log\left(h'(x_{k+1})\left[Q\, w_1 - \sum_{i=1}^{k} \mathbf{q}_i\right] + \exp u(z_k)\right) \\ u(x^*) &= \log\Big(\mathbf{v}_{k+1}\big(Q\, w_1 - \mathbf{Qc}_k\big) + \mathbf{e}^u_k\Big) \end{aligned} \tag{7}$$

Note that $u(x^*)$ is stored for computing the acceptance probability later.

We can then compute x^* from $u(x^*)$ as follows:

$$\begin{aligned} u(x^*) &= h(x_{k+1}) + (x^* - x_{k+1})\, h'(x_{k+1}) \\ (x^* - x_{k+1})\, h'(x_{k+1}) &= u(x^*) - h(x_{k+1}) \\ x^* &= \frac{u(x^*) - h(x_{k+1})}{h'(x_{k+1})} + x_{k+1} \\ x^* &= \frac{u(x^*) - \mathbf{h}_{k+1}}{\mathbf{v}_{k+1}} + \mathbf{x}_{k+1} \end{aligned}$$

Having sampled x^*, we then move on to deciding if the sample is to be accepted as a true sample from the distribution of interest. The decision is governed by the rejection test. The acceptance probability to accepting the sample x^* is

$$A(x^*) = \frac{\exp h(x^*)}{\exp u(x^*)}$$

The rejection test can be carried out by first generating a standard uniform variable w_2 and then check if $w_2 < A(x^*)$. If the above condition is satisfied, we would accept the sampled x^*, otherwise we reject the sample.

We note that the acceptance probability is lower bounded by

$$A(x^*) = \frac{\exp h(x^*)}{\exp u(x^*)} \geq \frac{\exp l(x^*)}{\exp u(x^*)} = B(x^*)$$

since $l(x) \leq h(x)$ for all x. Thus, rather than performing the rejection test that may require expensive computation of $h(x^*)$, we instead start with the following *squeezing* test that checks if $w_2 < B(x^*)$, if this condition is satisfied, then we know for sure $w_2 < B(x^*) \leq A(x^*)$ and we would accept the sample x^*. Otherwise, if the condition is not satisfied, we continue with the rejection test.

Note that whenever we perform the rejection test, we compute the log likelihood $h(x^*)$. Regardless of whether the sample x^* is accepted or not, we will update (improve) the upper hull $u(x)$, the lower hull $l(x)$, and the associated variables using the newly computed $h(x^*)$. This allows us to sample x more efficiently in the following iterations. We present the update procedure in detail below.

Update Step. For j such that $x^* > x_j$ and $x^* < x_{j+1}$, we update the caches (and renaming the indices as necessary) as follows:

$$\begin{aligned}
\mathbf{x} &:= [x_0, \dots, x_j, x^*, x_{j+1}, \dots, x_{M+1}] \\
&= [\mathbf{x}_0, \dots, \mathbf{x}_j, x^*, \mathbf{x}_{j+1}, \dots, \mathbf{x}_{M+1}] \\
\mathbf{h} &:= [h(x_0), \dots, h(x_j), h(x^*), h(x_{j+1}) \dots, h(x_{M+1})] \\
&= [\mathbf{h}_0, \dots, \mathbf{h}_j, h(x^*), \mathbf{h}_{j+1}, \dots, \mathbf{h}_{M+1}] \\
\mathbf{v} &:= [h'(x_0), \dots, h'(x_j), h'(x^*), h'(x_{j+1}) \dots, h'(x_{M+1})] \\
&= [\mathbf{v}_0, \dots, \mathbf{v}_j, h'(x^*), \mathbf{v}_{j+1}, \dots, \mathbf{v}_{M+1}]
\end{aligned}$$

Since adding a new point x^* affects the location of both z_i on the left of x^* and on the right of x^*, we update $\mathbf{z}$ by adding z_j^* and z_{j+1}^* (removing original z_j) as follows:

$$\begin{aligned}
\mathbf{z} &:= [z_0, \dots, z_{j-1}, z_j^*, z_{j+1}^*, z_{j+1}, \dots, z_M] \\
&= [\mathbf{z}_0, \dots, \mathbf{z}_{j-1}, z_j^*, z_{j+1}^*, \mathbf{z}_{j+1}, \dots, \mathbf{z}_M]
\end{aligned}$$

where the new z_i^* (for $i \in \{j, j+1\}$) are computed with Equation (1) using the new $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{v}$. Note that for the boundary cases $j = 0$ and $j = M$, we would fix z_0^* to x^- the left bound and z_{M+1}^* to x^+ the right bound.

We also update the cache $\mathbf{u}$ and $\mathbf{e}^u$ accordingly:

$$\begin{aligned}
\mathbf{u} &:= [u(z_0), \dots, u(z_{j-1}), u(z_j^*), u(z_{j+1}^*), u(z_{j+1}), \dots, u(z_M)] \\
&= [\mathbf{u}_0, \dots, \mathbf{u}_{j-1}, u(z_j^*), u(z_{j+1}^*), \mathbf{u}_{j+1}, \dots, \mathbf{u}_M] \\
\mathbf{e}^u &:= \left[e^{u(z_0)}, \dots, e^{u(z_{j-1})}, e^{u(z_j^*)}, e^{u(z_{j+1}^*)}, e^{u(z_{j+1})}, \dots, e^{u(z_M)}\right] \\
&= \left[\mathbf{e}_0^u, \dots, \mathbf{e}_{j-1}^u, e^{u(z_j^*)}, e^{u(z_{j+1}^*)}, \mathbf{e}_{j+1}^u, \dots, \mathbf{e}_M^u\right]
\end{aligned}$$

where $u(z_i^*)$ are computed with Equation (2). We do not worry about the boundary for $\mathbf{u}$ and $\mathbf{e}^u$.

Excepting for the boundary cases, when we modify two values in $\mathbf{z}$, three values in $\mathbf{q}$ would need to be modified. In particular, we change $\mathbf{q}_i$, where $i \in \{j, j+1, j+2\}$, such that

$$\mathbf{q} := [\mathbf{q}_1, \dots, \mathbf{q}_{j-1}, \mathbf{q}_j^*, \mathbf{q}_{j+1}^*, \mathbf{q}_{j+2}^*, \mathbf{q}_{j+2}, \dots, \mathbf{q}_M]$$

where $\mathbf{q}_i^*$ is computed using Equation (4). Note that in contrast to the updating rule for other caches, in this case we have replaced two $\mathbf{q}_i$ with three $\mathbf{q}_i^*$ excepting for the boundary cases. For the boundary case $j = 0$, we replace $\mathbf{q}_1$ with $\mathbf{q}_1^*$ and $\mathbf{q}_2^*$. While for the boundary case $j = M$, we instead replace $\mathbf{q}_M$ with $\mathbf{q}_M^*$ and $\mathbf{q}_{M+1}^*$.

Finally, we also update $Q = \sum_i \mathbf{q}_i$ and $\mathbf{Qc} = \texttt{cumsum}([0, \mathbf{q}])$.

Code. An implementation of the efficient sampler is represented in Code Appendix C.

3.2 Stable Sampler

Here, we discuss an implementation variant of the ARS algorithm, which we name as the stable sampler. This implementation addresses numerical instability that may present in the efficient sampler described above. This problem arises when the $u(z_i)$ is too large in magnitude (either positive or negative), making its exponent unstable.

In MATLAB, the largest real value that can be stored is 1.7977e+308, and its log is 709.7827. Thus, the highest $u(z_i)$ we can have is 709.7827 before we run into numerical overflow. On the other hand, the smallest (most negative) real value is 2.2251e-308 with its log -708.3964. If we have $u(z_i)$ that is smaller than -708.3964, then MATLAB will treat its exponent $\exp u(z_i)$ as zero. We will run into an error when all the $\exp u(z_i)$ are zero.

To bypass this issue, we compute and store the variables in log format. Below, we will detail the modification to the efficient sampler that gives us the stable sampler. We note that the inputs to the algorithm remain the same.

Initialisation Step. We initialise $\mathbf{x}$, $\mathbf{h}$, $\mathbf{v}$, $\mathbf{z}$, and $\mathbf{u}$ in the exact same way as in the efficient sampler. Additionally, we also store an additional cache for the log of the derivative:

$$\mathbf{ln}^v = [\mathbf{ln}^v_0, \ldots, \mathbf{ln}^v_{M+1}] \coloneqq [\log h'(x_0), \ldots, \log h'(x_{M+1})] = [\log \mathbf{v}_0, \ldots, \log \mathbf{v}_{M+1}]$$

We note that since the derivatives can be negative, the log of the derivatives may not be defined. Note that in such cases, we would store only the log of their absolute value. To illustrate, say the derivative is $-\nu$ where ν is a positive number, then we store $\log \nu$ instead of $\log(-\nu)$. Note that this will not pose any problem since the negative sign in the gradient will be cancelled out later by the multiplication of another negative number:

$$\log(-\nu \times -\mu) = \log(\nu \times \mu) = \log \nu + \log \mu$$

where $\mu > 0$. In MATLAB, however, we do not need to perform the above since the log of a negative number will be expressed as a complex number:

$$\log(-\nu) = \log \nu + \log(-1) = \log \nu + \pi i$$

We note that the complex number will be cancelled out later when we compute $\log \mathbf{q}_i$.

Another distinction compared to the efficient sampler is that we do not store $\mathbf{e}^u$ directly. We instead store $\mathbf{e}^o$ that corresponds to an adjusted version of $\mathbf{e}^u$.

We first define

$$\mathbf{u}_{\max} \coloneqq \max_i u(z_i)$$

the maximum of the $u(z_i)$. We then define $o(z_i) \coloneqq u(z_i) - \mathbf{u}_{\max}$ the adjusted value so we have

$$\mathbf{o} = [\mathbf{o}_0, \ldots, \mathbf{o}_M] \coloneqq [o(z_0), \ldots, o(z_M)] = [u(z_0) - \mathbf{u}_{\max}, \ldots, u(z_M) - \mathbf{u}_{\max}]$$

We store the exponents of $o(z_i)$:

$$\mathbf{e}^o = [\mathbf{e}^o_0, \ldots, \mathbf{e}^o_M] \coloneqq \left[e^{o(z_0)}, \ldots, e^{o(z_M)}\right].$$

instead of $\mathbf{e}^u$. In this form, the largest value in $\mathbf{e}^o$ is 1 since the largest value in $\mathbf{o}$ is 0. Note that $\mathbf{e}^u$ can easily be recovered by

$$\mathbf{e}^u_i = \exp u(z_i) = \exp\big(o(z_i) + \mathbf{u}_{\max}\big) = e^{\mathbf{u}_{\max}} \exp o(z_i) = e^{\mathbf{u}_{\max}}\, \mathbf{e}^o_i$$

Next, rather than storing Q and $\mathbf{q}_i$, which can be large, we store their log values: $\log Q$ and $\log \mathbf{q}_i$. Recall that

$$\mathbf{q}_i \coloneqq \frac{1}{h'(x_i)}\Big(\exp u(z_i) - \exp u(z_{i-1})\Big)$$

Its log can be computed as

$$\begin{aligned}
\log \mathbf{q}_i &= \log\left(\frac{1}{h'(x_i)}\Big(\exp u(z_i) - \exp u(z_{i-1})\Big)\right) \\
&= -\log h'(x_i) + \log\Big(\exp u(z_i) - \exp u(z_{i-1})\Big) \\
&= -\log h'(x_i) + \log\Big(e^{\mathbf{u}_{\max}} \exp o(z_i) - e^{\mathbf{u}_{\max}} \exp o(z_{i-1})\Big) \\
&= -\log h'(x_i) + \mathbf{u}_{\max} + \log\Big(\exp o(z_i) - \exp o(z_{i-1})\Big) \\
&= -\mathbf{ln}_i^v + \mathbf{u}_{\max} + \log\left(\mathbf{e}_i^o - \mathbf{e}_{i-1}^o\right)
\end{aligned}$$

Here, care must be taken to ensure the log is defined, this is because both $h'(x_i)$ and the difference $\exp o(z_i) - \exp o(z_{i-1})$ can be negative. Fortunately, we note that $h'(x_i)$ and $\exp o(z_i) - \exp o(z_{i-1})$ will always have the same sign:

$$\begin{aligned}
h'(x_i) > 0 &\implies u(z_i) - u(z_{i-1}) > 0 \\
&\implies o(z_i) - o(z_{i-1}) > 0 \\
&\implies \exp o(z_i) - \exp o(z_{i-1}) > 0
\end{aligned}$$

The same can be shown for $h'(x_i) < 0$. Thus, we can simply remove the negative sign (if presence) inside both log when computing $\log \mathbf{q}_i$, that is, an alternative way to compute $\log \mathbf{q}_i$ is

$$\log \mathbf{q}_i = -\log|h'(x_i)| + \mathbf{u}_{\max} + \log\Big(\big|\exp o(z_i) - \exp o(z_{i-1})\big|\Big)$$

In MATLAB, however, this is taken care of automatically since the positive πi and the negative πi cancel out each other. To illustrate, if $h'(x_i) < 0$ then

$$\begin{aligned}
&-\log h'(x_i) + \log\Big(\exp o(z_i) - \exp o(z_{i-1})\Big) \\
&= -\log\big(-h'(x_i)\big) - \pi i + \log\Big(-\big(\exp o(z_i) - \exp o(z_{i-1})\big)\Big) + \pi i \\
&= -\log\big(-h'(x_i)\big) + \log\Big(-\big(\exp o(z_i) - \exp o(z_{i-1})\big)\Big)
\end{aligned}$$

The $\log \mathbf{q}_i$ is stored in $\mathbf{ln}^q$:

$$\mathbf{ln}^q = [\mathbf{ln}_1^q, \ldots, \mathbf{ln}_M^q] := [\log \mathbf{q}_1, \ldots, \log \mathbf{q}_M]$$

Before computing Q, we first derive the cdf $c(z_j)$ evaluated at z_j (see Equation (6)):

$$\begin{aligned}
c(z_j) &= \frac{1}{Q}\sum_{i=1}^{j} \mathbf{q}_i \\
&= \frac{1}{Q}\sum_{i=1}^{j} \exp \mathbf{ln}_i^q \\
&= \frac{1}{Q}\sum_{i=1}^{j} \exp \mathbf{ln}_{\max}^q \times \exp\left(\mathbf{ln}_i^q - \mathbf{ln}_{\max}^q\right) \\
&= \frac{\exp \mathbf{ln}_{\max}^q}{Q}\sum_{i=1}^{j} \exp\left(\mathbf{ln}_i^q - \mathbf{ln}_{\max}^q\right)
\end{aligned}$$

where

$$\mathbf{ln}_{\max}^q = \max_i \mathbf{ln}_i^q$$

Since we know $c(z_M) = 1$, we have

$$\begin{aligned} 1 &= \frac{\exp \mathbf{ln}^q_{\max}}{Q} \sum_{i=1}^{M} \exp\left(\mathbf{ln}^q_i - \mathbf{ln}^q_{\max}\right) \\ Q &= \exp \mathbf{ln}^q_{\max} \sum_{i=1}^{M} \exp\left(\mathbf{ln}^q_i - \mathbf{ln}^q_{\max}\right) \\ \log Q &= \mathbf{ln}^q_{\max} + \log\left(\sum_{i=1}^{M} \exp\left(\mathbf{ln}^q_i - \mathbf{ln}^q_{\max}\right)\right) \end{aligned}$$

and thus

$$\begin{aligned} c(z_j) &= \frac{\exp \mathbf{ln}^q_{\max}}{Q} \sum_{i=1}^{j} \exp\left(\mathbf{ln}^q_i - \mathbf{ln}^q_{\max}\right) \\ &= \exp\left(\mathbf{ln}^q_{\max} - \log Q\right) \sum_{i=1}^{j} \exp\left(\mathbf{ln}^q_i - \mathbf{ln}^q_{\max}\right) \\ &\equiv \sum_{i=1}^{j} \exp\left(\mathbf{ln}^q_i - \log Q\right) \end{aligned}$$

In the stable implementation, we store the cache

$$\mathbf{c} = [\mathbf{c}_0, \ldots, \mathbf{c}_M] \coloneqq [c(z_0), \ldots, c(z_M)]$$

rather than $\mathbf{Q}\mathbf{c}$.

Sampling Step. The sampling procedure for the stable sampler is very similar to the efficient sampler, only the caches used are different.

We first find k by iterating through i such that

$$\begin{aligned} k &= \inf\{i \,|\, Q\, w_1 \le Q\, c(z_{i+1})\} \\ &= \inf\{i \,|\, w_1 \le c(z_{i+1})\} \end{aligned}$$

where $w_1 \sim \mathrm{U}(0, 1)$ a realised standard uniform random variable.

We then compute $u(x^*)$, from Equation (7):

$$\begin{aligned} u(x^*) &= \log\left(h'(x_{k+1})\left[Q\, w_1 - \sum_{i=1}^{k} \mathbf{q}_i\right] + \exp u(z_k)\right) \\ &= \log\left(h'(x_{k+1})\, Q\left[w_1 - \frac{1}{Q}\sum_{i=1}^{k} \mathbf{q}_i\right] + \frac{Q}{Q}\exp\left(o(z_k) + \mathbf{u}_{\max}\right)\right) \\ &= \log Q + \log\left(h'(x_{k+1})\left(w_1 - \mathbf{c}_k\right) + \exp\left(\mathbf{u}_{\max} - \log Q\right)\exp o(z_k)\right) \\ &= \log Q + \log\left(\mathbf{v}_{k+1}\left(w_1 - \mathbf{c}_k\right) + \exp\left(\mathbf{u}_{\max} - \log Q\right)\mathbf{e}^o_k\right) \end{aligned}$$

Note that $u(x^*)$ is stored for computing the acceptance probability later.

The sample x^* can be computed in the same way as the efficient sampler.

$$x^* = \frac{u(x^*) - \mathbf{h}_{k+1}}{\mathbf{v}_{k+1}} + \mathbf{x}_{k+1}$$

We then accept or reject the sample using the same routine as in the efficient sampler.

Update Step. The variable $\mathbf{x}$, $\mathbf{h}$, $\mathbf{v}$, $\mathbf{z}$, and $\mathbf{u}$ are updated in the same manner as in the efficient sampler. Additionally, we also update the log derivative cache $\mathbf{ln}^v$ as follows:

$$\begin{aligned} \mathbf{ln}^v \coloneqq\ & [\log h'(x_0), \ldots, \log h'(x_j), \log h'(x^*), \log h'(x_{j+1}) \ldots, \log h'(x_{M+1})] \\ & [\mathbf{ln}^v_0, \ldots, \mathbf{ln}^v_j, \log h'(x^*), \mathbf{ln}^v_{j+1} \ldots, \mathbf{ln}^v_{M+1}] \end{aligned}$$

The other caches $\mathbf{u}_{\max}$, $\mathbf{e}^o$, $\mathbf{ln}^q$, $\mathbf{ln}^q_{\max}$, $\log Q$, and $\mathbf{c}$ are recomputed from the formulae outlined above. Due to this, the stable sampler is less efficient compared to the efficient sampler.

Code. A MATLAB implementation of the stable sampler is presented in Code Appendix D.

4 Testing the Implemented ARS Algorithm

We test our implementation on normal distribution and gamma distribution.

Standard Normal. We use the implemented sampler to generate standard normal distribution variables for testing purpose. The implemented sampler took only 5.88 seconds to generate *one million* independent random variables $x_i \sim \mathrm{Normal}(\mu = 0, \sigma^2 = 1)$ (this is very fast for rejection sampling). From the samples, we compute the sample mean and sample variance as a sanity check to assess the sampler. The sample mean is found to be $\bar{x} = 0.001145$ and the sample variance is $s_x^2 = 1.0014$, these values are very close to the ground truth.

In addition, we perform the one-sample Kolmogorov-Smirnov (KS) test to test the samples' normality. From the KS test, we fail to reject the null hypothesis that the samples are drawn from standard normal distribution. Thus we conclude that our sampler correctly generates standard normal random variables.

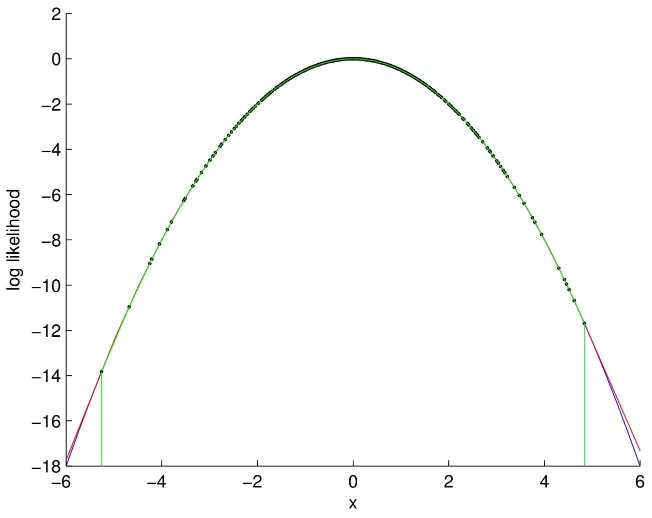
Normal Distribution. We then proceed to generating variables from the normal distribution $x_i \sim \mathrm{Normal}(\mu = 3, \sigma^2 = 5)$. We found that the sample mean and sample variance are

$$\bar{x} = 3.00262$$
$$s_x^2 = 5.00744$$

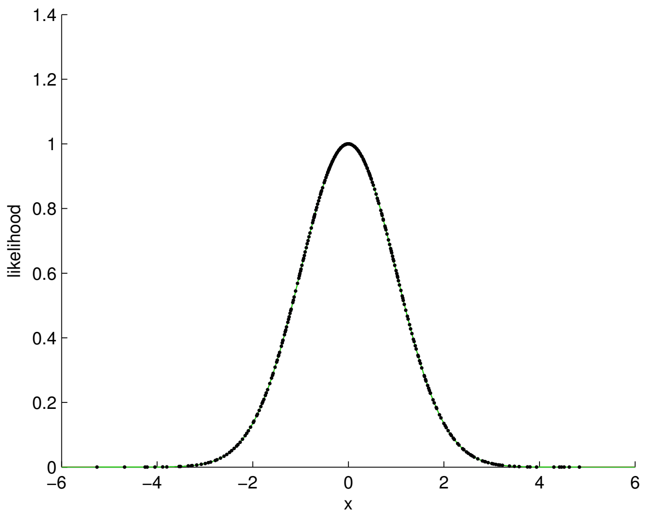
Applying the KS test, we find that the *p*-value for rejecting the null hypothesis that the samples are from $\mathrm{Normal}(\mu = 3, \sigma^2 = 5)$ is 0.4092. Thus we fail to reject the null hypothesis and conclude that the implemented ARS algorithm is correct.

Code. The test script is presented in Code Appendix E.

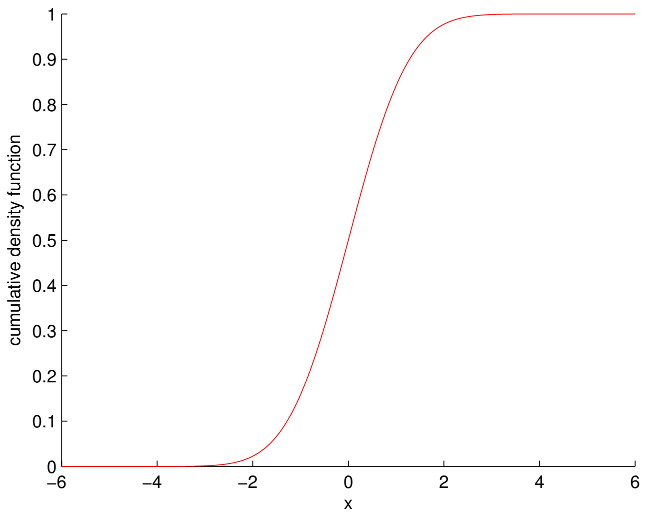
Below, we present the final envelope function and the squeezing function after sampling $N = 1{,}000{,}000$ standard normal variables. Note that there are 277 points of x (represented by black dots) where $h(x)$ and $h'(x)$ is evaluated, this is way below the number of samples N. We can see that the upper hull function closely resembles the true log likelihood function, while the lower hull function closely resembles the log likelihood function in the range $[x_1, x_{277}]$.



We can see that the envelope function and the squeezing function closely resembles the true likelihood function (un-normalised).



Additionally, we note that the cdf is also very close to the cdf of standard normal distribution:



Gamma Distribution. Besides the normal distribution, we also test the implemented sampler's ability to generate gamma random variables. We first note that the probability density function (pdf) of a gamma distribution $x_i \sim \mathrm{Gamma}(k, \theta)$ with shape parameter k and scale parameter θ is log-concave if and only if $k > 1$. Let the pdf be $f(x)$, we have

$$f(x) = \frac{1}{\Gamma(k)\,\theta^k}\, x^{k-1} \exp\Big(-\frac{x}{\theta}\Big)$$

and the unnormalised pdf is thus

$$\begin{aligned} g(x) &\propto f(x) \\ &\propto x^{k-1} \exp\Big(-\frac{x}{\theta}\Big) \end{aligned}$$

The log likelihood and its derivatives are

$$\begin{aligned} h(x) &= \log g(x) \\ &= (k-1)\log x - \frac{x}{\theta} \\ h'(x) &= \frac{k-1}{x} - \frac{1}{\theta} \\ h''(x) &= -\frac{k-1}{x^2} \end{aligned}$$

Hence we can see that $f(x)$ is log-concave when $k > 1$. Enforcing the constraint so that $f(x)$ is log-concave, we set $k = 3$ and $\theta = 2$. The mean and variance of the gamma distribution are

$$\begin{aligned} \mathbb{E}[X] &= k\theta = 6 \\ \mathbb{V}[X] &= k\theta^2 = 12 \end{aligned}$$

With the implemented ARS algorithm, we simulate $N = 1{,}000{,}000$ samples and compute their sample mean and sample variance. We found that the sample mean and the sample variance are very close to the theoretical counterparts:

$$\begin{aligned} \bar{x} &= 6.00691 \\ s_x^2 &= 11.983 \end{aligned}$$

In addition, we also perform the one sample KS test for gamma distribution, with null hypothesis being that the samples are generated from $\mathrm{Gamma}(k = 3, \theta = 2)$. The result from this is that we fail to reject the null hypothesis (with p-value 0.06202). This gives confidence that the ARS algorithm is implemented correctly.

Code. The script to sample and test the gamma random variables is presented in Code Appendix F.

5 Using the Code

Please cite this report if you are using our ARS implementation:

> Lim, K. W. (2018). A Stable Implementation of the Adaptive Rejection Sampler in MATLAB. Technical Report, National University of Singapore.

The code is available at the author's website.[2]

References

Gilks, W. R. and Wild, P. (1992). Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 41(2):337–348.

[2] `https://karwailim.github.io/`

Code

Code is released under MIT License:

A Demo: Generating a Sample

Part of `example.m`

```
%%% This script provides step-by-step tutorial to illustrate the adaptive
%%% rejection sampling (ARS) method.
%%% In this example, we sample from normal (gaussian) distribution.

% clc;
clear all; close all; % clear console

% set seed
rng(11111);


%% Settings

% parameters for normal distribution
mu    = 0;
sigma2 = 1;

% support of the distribution
left_bound = -inf;
right_bound = inf;

% likelihood up to a proportionality constant
g = @(x) exp( -1/2 .* (x - mu).^2 ./ sigma2 );

% log of likelihood g(x)
h = @(x) -1/2 .* (x - mu).^2 ./ sigma2;

% first derivative of h(x)
d = @(x) -(x - mu) ./ sigma2;


%% ARS

% choose two starting points x1, x2
x1 = -1;
x2 = 2;

x = [x1, x2];

% evaluate the log likelihood h(x)
llik1 = h(x1);
llik2 = h(x2);

llik = [llik1, llik2];

% evaluate the gradients h'(x)
grad1 = d(x1);
grad2 = d(x2);

grad = [grad1, grad2];

% find intersections of the tangents for x1 and x2
```

```
z0 = left_bound;
z1 = (llik2 - llik1 - x2*grad2 + x1*grad1) / (grad1 - grad2);
z2 = right_bound;

z = [z0, z1, z2];

% piecewise upper hull (for envelope) function u(x) follows this form
% u_k(x) = h(x_j) + (x - x_j)*d(x_j)
% for j = 1,2; x in [z_(j-1), z_j]

% find u(z), upper hull at the intersection
u_z0 = -inf;
u_z1 = llik1 + (z1 - x1)*grad1;
u_z2 = llik2 + (z2 - x2)*grad2;

u_z = [u_z0, u_z1, u_z2];

% piecewise lower hull (for squeezing) function l(x) follows this form
% l_k(x) = ((x_(j+1) - x)*h(x_j) + (x - x_j)*h(x_(j+1))) / (x_(j+1) - x_j)
% for j = 1; x in [x_j, x_(j+1)]
% note l_k(x) = -inf for all other x (x < x1, x > x2)

% to find the envelope function, we first compute the normalising constant Q
cdf_part_z1 = 1/grad1 * (exp(u_z1) - exp(u_z0)); % area under the curve for the first piece
cdf_part_z2 = 1/grad2 * (exp(u_z2) - exp(u_z1));

cdf_part_z = [cdf_part_z1, cdf_part_z2];

Q = sum(cdf_part_z);
Q_ = (1/grad1 - 1/grad2)*exp(u_z1); % for checking

% piecewise cdf for the envelope function, \int exp u(x) follows this form
% c_k(x) = 1/norm_const * (\int_(u<k) c_u(x) + \int_x c_k(x) )
% for j = 1,2; x in [z_{j-1}, z_j]

% sample a standard uniform random variable to generate a sample x via the
% inverse cdf method
w1 = rand; % 0.8389 with seed 11111

% find x corresponds to w1, we times w1 by Q since it is easier to work on
% the un-normalised space
Qw1 = Q*w1;

cdf_z = cumsum(cdf_part_z);

u_x = nan;
if Qw1 < cdf_z(1) % belong to first piece
   u_x = log(Qw1 * grad1);
   sampled_x = x1 + (u_x - llik1)/grad1;
elseif Qw1 < cdf_z(2)
   A2 = (Qw1 - cdf_z(1))*grad2 + exp(u_z1);
   u_x = log(A2);
   sampled_x = x2 + (u_x - llik2)/grad2;
end

% squeezing test
w2 = rand; % 0.4789

squeezing_test = false;
for j = 1:(length(x)-1)
   if and(sampled_x > x(j), sampled_x < x(j+1))
      l_x = ((x(j+1) - sampled_x)*llik(j) + (sampled_x - x(j))*llik(j+1)) / (x(j+1) - x(j));
      squeeze_ratio = exp(l_x - u_x);
      squeezing_test = w2 < squeeze_ratio; % 0.4789 < 0.1818, therefore reject
   end
end

% display test result
squeezing_test %#ok<NOPTS>
% if the test pass we do not have to perform the acceptance/rejection test

% acceptance rejection test
rejection_test = false;
h_x = nan;
for j = 1:(length(x)-1)
   if and(sampled_x > x(j), sampled_x < x(j+1))
      h_x = h(sampled_x);
      acceptance_ratio = exp(h_x - u_x);
      rejection_test = w2 < acceptance_ratio; % 0.4789 < 0.5242, therefore accept
   end
end

% display test result
rejection_test %#ok<NOPTS>
% if the test pass then we accept the sample, in any case we update the
% envelope function and squeezing function since we have evaluated h(x*)

%% PLOT

left_xlim = -5;
right_xlim = 5;
```

```
granularity = 0.01;

% draw log likelihood, upper hull and lower hull function
figure;
hold on;

% plot log likelihood
plot_x = left_xlim:granularity:right_xlim; % range of x for plot
plot_y = h(plot_x);
plot(plot_x,plot_y,'color', 'blue');

% plot upper hull function
plot_u = inf(size(plot_x)); % initialise
for j = 1:length(x)
   indice = and(z(j) <= plot_x, plot_x < z(j+1)); % indice of plot_x in abscissae k
   plot_u(indice) = llik(j) + (plot_x(indice) - x(j)).*grad(j);
end
plot(plot_x,plot_u,'color','red');

% mark tangent points
plot(x, llik, 'k+');

% plot lower hull function
plot_l = -inf(size(plot_x)); % initialise
for j = 1:(length(x)-1)
   indice = and(x(j) <= plot_x, plot_x < x(j+1)); % indice of plot_x in abscissae k
   plot_l(indice) = ((x(j+1) - plot_x(indice)).*llik(j) + (plot_x(indice) - x(j)).*llik(j+1)) ./ (x(j+1) - x(j
        ));
end
plot(plot_x,plot_l,'color','green'); % squeezing function between x's

YL = ylim; % get the y limits
line([x(1) x(1)], [YL(1) llik(1)], 'color','green'); % squeezing function outside x's = -inf
line([x(end) x(end)], [YL(1) llik(end)], 'color','green'); % squeezing function outside x's = -inf

% axis([10,inf,-0.005,0.015]);
xlabel('x'); % x-axis label
ylabel('log␣likelihood'); % y-axis label

% draw the likelihood, envelope function and the squeezing function
figure;
hold on;

% plot likelihood
plot_f = g(plot_x);
plot(plot_x,plot_f,'color', 'blue');

% envelope function
plot(plot_x,exp(plot_u), 'color', 'red');

% squeezing function
plot(plot_x,exp(plot_l), 'color', 'green');

% mark tangent points
plot(x, exp(llik), 'k+');

% axis([10,inf,-0.005,0.015]);
xlabel('x'); % x-axis label
ylabel('likelihood'); % y-axis label

% draw the cdf of the envelope function
figure;
hold on;

% plot cdf
plot_c = zeros(size(plot_x)); % initialise
cum_temp = 0; % temporary variable to store the cumulative
for j = 1:length(x)
   indice = and(z(j) <= plot_x, plot_x < z(j+1)); % indice of plot_x in abscissae k
   plot_c(indice) = cum_temp + 1./grad(j).*(exp(plot_u(indice)) - exp(u_z(j)));
   cum_temp = cum_temp + 1/grad(j)*(exp(u_z(j+1)) - exp(u_z(j)));
end
plot_c = plot_c / cum_temp;
plot(plot_x,plot_c,'color','red');

% plot the piecewise changing point
YL = ylim; % get the y limits
for j = 1:length(z)
   line([z(j) z(j)], YL, 'color','blue', 'linestyle', ':');
end

% plot the sample_x
XL = xlim;
line([sampled_x sampled_x], [YL(1) w1], 'color','green', 'linestyle', ':');
line([XL(1) sampled_x], [w1 w1], 'color','green', 'linestyle', ':');

xlabel('x'); % x-axis label
ylabel('cumulative␣density␣function'); % y-axis label
```

B Demo: Updating Envelope and Squeezing Function

Part of `example.m`

```matlab
%% UPDATE ENVELOPE AND SQUEEZING FUNCTIONS (NAIVE WAY)

% We add the sampled x* to the list of x and update the envelope and
% squeezing function

x = sort([left_bound, x, sampled_x, right_bound]);
% x = sort([left_bound, x, right_bound]); % for testing (check with above demo)

% evaluate the log likelihood h(x)
llik = h(x);

% evaluate the gradients h'(x)
grad = d(x);

% number of points
K = length(x);

% find intersections of the tangents
ind = 2:(K-2);

z = [left_bound, ...
   (llik(ind+1) - llik(ind) - x(ind+1).*grad(ind+1) + x(ind).*grad(ind)) ...
      ./ (grad(ind) - grad(ind+1)), ...
   right_bound];

% piecewise upper hull (for envelope) function u(x) follows this form
% u_k(x) = h(x_j) + (x - x_j)*d(x_j)
% for j = 1,2; x in [z_(j-1), z_j]

% find u(z), upper hull at the intersection
indz = 2:(K-1);

u_z = [ llik(2) + (z(1) - x(2))*grad(2) , ... % the first u_z use the gradient of the next piece
      llik(indz) + (z(indz) - x(indz)).*grad(indz) ];

exp_u_z = exp(u_z);

% to find the envelope function, we first compute the normalising constant Q
cdf_part_z = 1./grad(indz) .* (exp_u_z(indz) - exp_u_z(indz-1));

Q = sum(cdf_part_z);

%% PLOT

left_xlim = -5;
right_xlim = 5;

granularity = 0.01;

% draw log likelihood, upper hull and lower hull function
figure;
hold on;

% plot log likelihood
plot_x = left_xlim:granularity:right_xlim; % range of x for plot
plot_y = h(plot_x);
plot(plot_x,plot_y,'color', 'blue');

% plot upper hull function
plot_u = inf(size(plot_x)); % initialise
for j = 1:length(z)-1
   indice = and(z(j) <= plot_x, plot_x < z(j+1)); % indice of plot_x in abscissae k
   plot_u(indice) = llik(j+1) + (plot_x(indice) - x(j+1)).*grad(j+1);
end
plot(plot_x,plot_u,'color','red');

% mark tangent points
plot(x, llik, 'k+');

% plot lower hull function
plot_l = -inf(size(plot_x)); % initialise
for j = 2:length(z)-1
   indice = and(x(j) <= plot_x, plot_x < x(j+1)); % indice of plot_x in abscissae k
   plot_l(indice) = ((x(j+1) - plot_x(indice)).*llik(j) + (plot_x(indice) - x(j)).*llik(j+1)) ./ (x(j+1) - x(j
      ));
end
plot(plot_x,plot_l,'color','green'); % squeezing function between x's

YL = ylim; % get the y limits
line([x(2) x(2)], [YL(1) llik(2)], 'color','green'); % squeezing function outside x's = -inf
line([x(end-1) x(end-1)], [YL(1) llik(end-1)], 'color','green'); % squeezing function outside x's = -inf

% axis([10,inf,-0.005,0.015]);
xlabel('x'); % x-axis label
ylabel('log_likelihood'); % y-axis label
```

```
% draw the likelihood, envelope function and the squeezing function
figure;
hold on;

% plot likelihood
plot_f = g(plot_x);
plot(plot_x,plot_f,'color', 'blue');

% envelope function
plot(plot_x,exp(plot_u), 'color', 'red');

% squeezing function
plot(plot_x,exp(plot_l), 'color', 'green');

% mark tangent points
plot(x, exp(llik), 'k+');

XL = xlim;
YL = ylim;
axis([XL(1), XL(2), YL(1), 3]);
xlabel('x'); % x-axis label
ylabel('likelihood'); % y-axis label

% draw the cdf of the envelope function
figure;
hold on;

% plot cdf
plot_c = zeros(size(plot_x)); % initialise
cum_temp = 0; % temporary variable to store the cumulative
for j = 1:length(z)-1
   indice = and(z(j) <= plot_x, plot_x < z(j+1)); % indice of plot_x in abscissae k
   plot_c(indice) = cum_temp + 1./grad(j+1).*(exp(plot_u(indice)) - exp(u_z(j)));
   cum_temp = cum_temp + 1/grad(j+1)*(exp(u_z(j+1)) - exp(u_z(j)));
end
plot_c = plot_c / cum_temp;
plot(plot_x,plot_c,'color','red');

% plot the piecewise changing point
YL = ylim; % get the y limits
for j = 1:length(z)
   line([z(j) z(j)], YL, 'color','blue', 'linestyle', ':');
end

xlabel('x'); % x-axis label
ylabel('cumulative␣density␣function'); % y-axis label
```

C Efficient Sampler for ARS

In ARS_sample_fast.m

```
function [ samples ] = ARS_sample_fast( N, llik_func, ldev_func, init, bounds )
%ARS_SAMPLE Sample using ARS
%  Derivative-based ARS method
%
% Variables:
%         N = number of samples
%  llik_func = log likelihood function
%  ldev_func = derivative of log likelihood function
%      init = vector of initial points (for envelope & squeezing function)
%    bounds = boundary of the distribution support
%
% Author: KW Lim
% Last modified: 12 April 2016


% initialise generated values
samples = nan(N,1);

% initial points for envelope and squeezing function
x = sort([init, bounds]); % include left right bounds

% evaluate the log likelihood h(x)
llik = llik_func(x); % include left right bounds

% evaluate the gradients h'(x)
grad = ldev_func(x); % include left right bounds

% number of points
K = length(x);       % include left right bounds


% check correctness of initial points before proceeding
if length(init) <= 1
   error('ERROR:␣Not␣enough␣initial␣points!')
end
if or(min(init) < min(bounds), max(init) > max(bounds))
```

```
   error('ERROR:␣Initial␣points␣out␣of␣bound!')
end
if ~and(grad(2) > 0, grad(K-1) < 0)
   error('ERROR:␣Initial␣points␣invalid␣(derivatives␣same␣sign)')
end
if any(grad == 0)
   error('ERROR:␣Initial␣point␣have␣derivative␣zero!')
end
if length(unique(init)) ~= length(init)
   error('ERROR:␣Initial␣points␣repeated␣more␣than␣once!')
end

% find intersections of the tangents
ind = 2:(K-2);

z = [min(bounds), ...
   (llik(ind+1) - llik(ind) - x(ind+1).*grad(ind+1) + x(ind).*grad(ind)) ...
      ./ (grad(ind) - grad(ind+1)), ...
   max(bounds)];

% piecewise upper hull (for envelope) function u(x) follows this form
% u_k(x) = h(x_j) + (x - x_j)*d(x_j)
% for j = 1,2; x in [z_(j-1), z_j]

% find u(z), upper hull at the intersection
indz = 2:(K-1);

u_z = [ llik(2) + (z(1) - x(2))*grad(2) , ... % the first u_z use the gradient of the next piece
      llik(indz) + (z(indz) - x(indz)).*grad(indz) ];

exp_u_z = exp(u_z);

% piecewise lower hull (for squeezing) function l(x) follows this form
% l_k(x) = ((x_(j+1) - x)*h(x_j) + (x - x_j)*h(x_(j+1))) / (x_(j+1) - x_j)
% for j = 1; x in [x_j, x_(j+1)]
% note l_k(x) = -inf for all other x (x < x1, x > x2)

% to find the envelope function, we first compute the normalising constant Q
cdf_part_z = [0, ...
          1./grad(indz) .* (exp_u_z(indz) - exp_u_z(indz-1))];

cdf_z = cumsum(cdf_part_z);
Q = cdf_z(end);

iter = 1;
while iter <= N

   % sample a standard uniform random variable to generate a sample x via the
   % inverse cdf method
   w1 = rand; % 0.8389 with seed 11111

   % find x corresponds to w1, we times w1 by Q since it is easier to work on
   % the un-normalised space
   Qw1 = Q*w1;

   u_x = nan; % initialise
   for j = 2:length(cdf_z)
      if Qw1 < cdf_z(j)
         u_x = log((Qw1 - cdf_z(j-1))*grad(j) + exp_u_z(j-1));
         sampled_x = x(j) + (u_x - llik(j))/grad(j);
         break
      end
   end

   % squeezing test
   w2 = rand; % 0.4789


   % find j such that [sampled_x > x(j)] AND [sampled_x < x(j+1)]
   for k = 1:(length(x)-1)
      if and(sampled_x > x(k), sampled_x < x(k+1))
         j = k;
         break
      end
   end

   % compute l_x for squeezing test
   if and(j > 1, j < length(x)-1)
      l_x = ((x(j+1) - sampled_x)*llik(j) + (sampled_x - x(j))*llik(j+1)) / (x(j+1) - x(j));
   else
      l_x = -inf;
   end


   squeeze_ratio = exp(l_x - u_x);

   % if success
   if w2 < squeeze_ratio
      samples(iter) = sampled_x;
      iter = iter + 1;
      % if fail
   else
```

```
        % perform rejection test if failed squeezing test
        h_x = llik_func(sampled_x);
        acceptance_ratio = exp(h_x - u_x);

        % if success
        if w2 < acceptance_ratio
           samples(iter) = sampled_x;
           iter = iter + 1;
        end

        %%% update envelope and squeezing function if evaluate h_x
        % update x
        x = [x(1:j), sampled_x, x(j+1:end)];
        % update llik
        llik = [llik(1:j), h_x, llik(j+1:end)];
        % update grad
        grad = [grad(1:j), ldev_func(sampled_x), grad(j+1:end)];
        % update z
        if j == 1
           % boundary case
           ind = j+1;
        elseif j == length(x)-2 % note x already increase in size
           % boundary case
           ind = j;
        else
           ind = j:j+1;
        end

        z = [z(1:ind(1)-1), ...
           (llik(ind+1) - llik(ind) - x(ind+1).*grad(ind+1) + x(ind).*grad(ind)) ...
              ./ (grad(ind) - grad(ind+1)), ...
           z(ind(end):end)];
        % update u_z
        indu = j:j+1;
        u_z = [u_z(1:indu(1)-1), ...
           llik(indu) + (z(indu) - x(indu)).*grad(indu), ...
           u_z(indu(end):end)];

        exp_u_z = [exp_u_z(1:indu(1)-1), exp(u_z(indu)), exp_u_z(indu(end):end)];
        % update cdf_part
        if j == 1
           % boundary case
           indc = j+1:j+2;
        elseif j == length(x)-2 % note x already increase in size
           % boundary case
           indc = j:j+1;
        else
           indc = j:j+2;
        end

        cdf_part_z = [cdf_part_z(1:indc(1)-1), ...
           1./grad(indc) .* (exp_u_z(indc) - exp_u_z(indc-1)), ...
           cdf_part_z(indc(end):end)];

        cdf_z = cumsum(cdf_part_z);
        Q = cdf_z(end);
    end
end
end
```

D Stable Sampler for ARS

In ARS_sample_stable.m

```
function [ samples ] = ARS_sample_stable( N, llik_func, ldev_func, init, bounds )
%ARS_SAMPLE Sample using ARS
%  Derivative-based ARS method
%
% Variables:
%        N = number of samples
%  llik_func = log likelihood function
%  ldev_func = derivative of log likelihood function
%      init = vector of initial points (for envelope & squeezing function)
%    bounds = boundary of the distribution support
%
% Author: KW Lim
% Last modified: 13 April 2016


% initialise generated values
samples = nan(N,1);

% initial points for envelope and squeezing function
x = sort([init, bounds]); % include left right bounds

% evaluate the log likelihood: h(x)
llik = llik_func(x); % include left right bounds
```

```
% evaluate the gradients: h'(x)
grad = ldev_func(x); % include left right bounds

% log of gradients: log h'(x)
logGrad = log(grad); % can be complex number in MATLAB (warning if porting to other languages)

% number of points
K = length(x);       % include left right bounds


% check correctness of initial points before proceeding
if length(init) <= 1
   error('ERROR:␣Not␣enough␣initial␣points!')
end
if or(min(init) < min(bounds), max(init) > max(bounds))
   error('ERROR:␣Initial␣points␣out␣of␣bound!')
end
if ~and(grad(2) > 0, grad(K-1) < 0)
   error('ERROR:␣Initial␣points␣invalid␣(derivatives␣same␣sign)')
end
if any(grad == 0)
   error('ERROR:␣Initial␣point␣have␣derivative␣zero!')
end
if length(unique(init)) ~= length(init)
   error('ERROR:␣Initial␣points␣repeated␣more␣than␣once!')
end

%%% INITIALISE
% find intersections of the tangents
ind = 2:(K-2);

z = [min(bounds), ...
   (llik(ind+1) - llik(ind) - x(ind+1).*grad(ind+1) + x(ind).*grad(ind)) ...
      ./ (grad(ind) - grad(ind+1)), ...
   max(bounds)];

% find u(z), upper hull at the intersection
indz = 2:(K-1);

u_z = [ llik(2) + (z(1) - x(2))*grad(2) , ... % the first u_z use the gradient of the next piece
   llik(indz) + (z(indz) - x(indz)).*grad(indz) ];

max_u_z = max(u_z); % find the largest u_z for normalisation
norm_u_z = u_z - max_u_z; % normalise u_z

exp_norm_u_z = exp(norm_u_z);

% to find the envelope function, we first compute the normalising constant Q
indz = 2:(length(x)-1);
log_cdf_part_z = [-inf, ...
   - logGrad(indz) ...
   + max_u_z + log(exp_norm_u_z(indz) - exp_norm_u_z(indz-1))]; % complex number cancelled out

max_log_cdf_part_z = max(log_cdf_part_z);
norm_log_cdf_part_z = log_cdf_part_z - max_log_cdf_part_z;

norm_cdf_z = cumsum(exp(norm_log_cdf_part_z));
Q2 = norm_cdf_z(end);
norm_cdf_z = norm_cdf_z / Q2;
logQ = log(Q2) + max_log_cdf_part_z;

%%% SAMPLING
iter = 1;
while iter <= N

   % sample a standard uniform random variable to generate a sample x via the
   % inverse cdf method
   w1 = rand; % 0.8389 with seed 11111

   % find x corresponds to w1
   u_x = nan; % initialise
   for j = 2:length(norm_cdf_z)
      if w1 < norm_cdf_z(j)
         u_x = logQ + log( (w1 - norm_cdf_z(j-1))*grad(j) + exp_norm_u_z(j-1)*exp(max_u_z - logQ) );
         sampled_x = x(j) + (u_x - llik(j))/grad(j);
         break
      end
   end

   % squeezing test
   w2 = rand; % 0.4789

   % find j such that [sampled_x > x(j)] AND [sampled_x < x(j+1)]
   for k = 1:(length(x)-1)
      if and(sampled_x > x(k), sampled_x < x(k+1))
         j = k;
         break
      end
   end

   % compute l_x for squeezing test
   if and(j > 1, j < length(x)-1)
      l_x = ((x(j+1) - sampled_x)*llik(j) + (sampled_x - x(j))*llik(j+1)) / (x(j+1) - x(j));
   else
      l_x = -inf;
   end

   squeeze_ratio = exp(l_x - u_x);

   % if success
   if w2 < squeeze_ratio
      samples(iter) = sampled_x;
      iter = iter + 1;
      % if fail
   else
      % perform rejection test if failed squeezing test
      h_x = llik_func(sampled_x);
      acceptance_ratio = exp(h_x - u_x);

      % if success
      if w2 < acceptance_ratio
         samples(iter) = sampled_x;
         iter = iter + 1;
      end

      %%% update envelope and squeezing function if evaluate h_x
      % update x
      x = [x(1:j), sampled_x, x(j+1:end)];
      % update llik
      llik = [llik(1:j), h_x, llik(j+1:end)];
      % update grad
      grad_sampleX = ldev_func(sampled_x);
      grad = [grad(1:j), grad_sampleX, grad(j+1:end)];
      % update log grad
      logGrad = [logGrad(1:j), log(grad_sampleX), logGrad(j+1:end)];
      % update z
      if j == 1
         % boundary case
         ind = j:j+1;
      elseif j == length(x)-2 % note x already increase in size
         % boundary case
         ind = j;
      else
         ind = j:j+1;
      end
      z = [z(1:ind(1)-1), ...
         (llik(ind+1) - llik(ind) - x(ind+1).*grad(ind+1) + x(ind).*grad(ind)) ...
            ./ (grad(ind) - grad(ind+1)), ...
          z(ind(end):end)];

      % update u_z
      u_z = [u_z(1:ind(1)-1), ...
         llik(ind) + (z(ind) - x(ind)).*grad(ind), ...
         u_z(ind(end):end)];

      max_u_z = max(u_z); % find the largest u_z for normalisation
      norm_u_z = u_z - max_u_z; % normalise u_z

      exp_norm_u_z = exp(norm_u_z);

      % update the rest
      indz = 2:(length(x)-1);
      log_cdf_part_z = [-inf, ...
         - logGrad(indz) ...
         + max_u_z + log(exp_norm_u_z(indz) - exp_norm_u_z(indz-1))]; % complex number cancelled out

      max_log_cdf_part_z = max(log_cdf_part_z);
      norm_log_cdf_part_z = log_cdf_part_z - max_log_cdf_part_z;

      norm_cdf_z = cumsum(exp(norm_log_cdf_part_z));
      Q2 = norm_cdf_z(end);
      norm_cdf_z = norm_cdf_z / Q2;
      logQ = log(Q2) + max_log_cdf_part_z;

   end
end
end
```

E Test: Normal Distribution

In testARS_fast_normal.m and testARS_stable_normal.m

```
%%% This script test the implemented ARS method.

clc; clear all; close all; % clear console

% set seed
rng(151891);
```

```
%% SETTINGS

% number of samples
N = 1000000;

% parameters for normal distribution
mu    = 3.0;
sigma2 = 5.0;

% support of the distribution
left_bound = -inf;
right_bound = inf;
bounds = [left_bound, right_bound];

% log of likelihood, h(x) = log g(x)
llik_func = @(x) -1/2 .* (x - mu).^2 ./ sigma2;

% first derivative of d(x) = d/dx h(x)
ldev_func = @(x) -(x - mu) ./ sigma2;

% initial points
init = [-3, -1, 2, 4];


% generate samples using ARS
tic % start timer
samples = ARS_sample(N, llik_func, ldev_func, init, bounds);
toc % end timer

mu_est = mean(samples)
var_est = var(samples)

% KS test
% h = 0 => standard normal, h = 1 => not standard normal, p is p-value
[h,p] = kstest((samples - mu)./sqrt(sigma2))
[h,p] = kstest(samples, [samples normcdf(samples, mu, sqrt(sigma2))])
```

F Test: Gamma Distribution

In `testARS_fast_gamma.m` and `testARS_stable_gamma.m`

```
%%% This script test the implemented ARS method by simulating gamma dist.

clc; clear all; close all; % clear console

% set seed
rng(2848428);

%% SETTINGS

% number of samples
N = 1000000;

% parameters for normal distribution
shape = 3.0;
scale = 2.0;

% support of the distribution
left_bound = 0;
right_bound = 9e99;
bounds = [left_bound, right_bound];

% log of likelihood, h(x) = log g(x)
llik_func = @(x) (shape-1).*log(x) - x./scale;

% first derivative of d(x) = d/dx h(x)
ldev_func = @(x) (shape-1)./x - 1./scale;

% initial points
init = [1, 2, 5, 7];


% generate samples using ARS
tic % start timer
samples = ARS_sample(N, llik_func, ldev_func, init, bounds);
toc % end timer

mu_est = mean(samples)
var_est = var(samples)

% KS test
% h = 0 => correct distribution, h = 1 => not correct, p is p-value
[h,p] = kstest(samples, [samples, gamcdf(samples, shape, scale)])
```