Adaptive rejection sampling

Rejection sampling (RS) is a useful method for sampling intractable distributions. It defines an envelope function which upper-bounds the target unnormalised probability density to be sampled. It then proceeds to sample points in the area under the envelope, rejecting those points which fall above the target and accepting the rest. The accepted points are independent and identically distributed samples from the target distribution. There are two important issues with RS. The first is that if the envelope is a very loose upper bound, then most samples will be rejected and the scheme will be slow. The second is that for rejection sampling to work, we must be certain that the envelope is an upper bound to the target, which in practice may be a challenging task.

Adaptive rejection sampling (ARS) [GW92] is an efficient method for sampling log-concave targets, which deals with both of these issues. It is origially defined for univariate distributions, but can also be extended to multivariate distributions via Gibbs sampling.[Bis06] ARS maintains an envelope which adapts as more points are sampled, becoming a progressively tighter bound to the target, thereby avoiding the inefficiency of regular RS. Further, the way that ARS constructs this envelope guarantees that the envelope is in fact an upper bound to the target, which sidesteps the second difficulty described above.

An adaptive envelope function

Suppose we wish to sample from a log-concave univariate distribution with unnormalised distribution function f. Whereas RS defines a fixed envelope, ARS will define an envelope g_u that upper bounds f and adapts its shape as the sampling procedure progresses. By adapting its shape, using the infromation that f is log-concave, the envelope reduces the probability of future rejections. In addition the envelope function, ARS can use an optional function g_l which lower bounds f, called the squeezing function. The squeezing function can be used to avoid evaluating f in the rejection step, which can be especially useful if f is computationally expensive to evaluate.

Given the an ordered set of points $x_1 < x_2 < \ldots < x_K$, ARS defines the log-envelope $\log\ g_u$ to be the minimum over the tangents to $h = \log f$ at these points. The log-squeezing function is defined to be the piecewise linear function which joins the points $(x_k, h(x_k))$ inside the interval $[x_1, x_K]$ and is equal to $-\infty$ outside this innterval. Examples of envelope and squeezing functions are shown below.

Definition (Abscissa set, envelope function and squeezing function) Let $f(x)$ be a univariate log-concave function, with non-zero domain $D = \{x : f(x) > 0\}$. An abscissa set T_K is an ordered set of points in D such that

$$T_k = \{x_1 < x_2 < \ldots < x_K\}.$$

The envelope function $g_u(x)$ defined by T_k is

$$g_u(x) = \min_k g_{u,k}(x)$$

where $g_{u,k}(x), k = 1, 2, \ldots, K$ are piecewise exponential functions such that

$$g_{u,k}(x_k) = f(x_k) \text{ and } g'_{u,k}(x_k) = \log f'(x_k).$$

The squeezing function $g_l(x)$ defined by T_k is

$$g_l(x) = \begin{cases} \min_k g_{l,k}(x) & \text{if } x_1 \leq x \leq x_k, \\ 0 & \text{otherwise.} \end{cases}$$

where $g_{u,k}(x), k = 1, 2, \ldots, K-1$ are piecewise exponential functions such that

$$g_{l,k}(x_k) = f(x_k) \text{ and } g_{l,k}(x_{k+1}) = \log f(x_{k+1}).$$

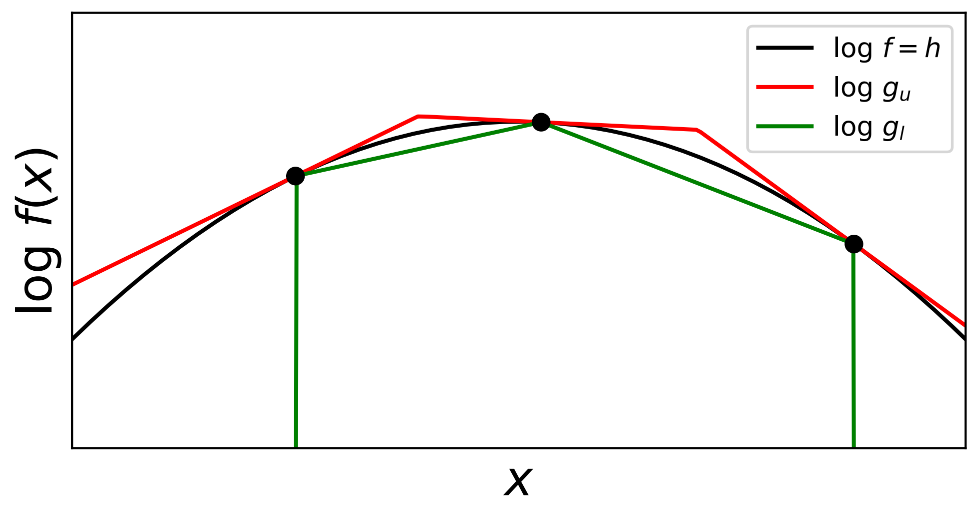
Below are functions implementing the necessary calculations to determine the envelope and squeezing functions, all of which are cheap operations. The functions take in points at input locations x and corresponding h and h' values and carry out computations in log-space, before exponentiating the result at the end.

```python
def g_u(x, xs, hs, dhdxs):

    z, _ = compute_points_of_intersection_and_intercepts(xs, hs, dhdxs)
    i = np.searchsorted(z, x)

    return np.exp(dhdxs[i] * (x - xs[i]) + hs[i])


def g_l(x, xs, hs):

    if all(x < xs) or all(x > xs):
        return 0.

    else:
        i = np.searchsorted(xs, x)
        m = (hs[i] - hs[i-1]) / (xs[i] - xs[i-1])

        return np.exp(hs[i-1] + (x - xs[i-1]) * m)


def compute_points_of_intersection_and_intercepts(x, h, dhdx):

    # y-intercepts c of envelope function line segments, intersection points z
    c = h - dhdx * x
    z = (c[1:] - c[:-1]) / (dhdx[:-1] - dhdx[1:])

    return z, c
```

Now let’s define a log unnormalised Gaussian log density, and use this to illustrate the envelope and squeezing function defined by an abcissa set with three points.

```python
def log_gaussian(mean, variance):
    return lambda x : (- 0.5 * (x - mean) ** 2 / variance, - (x - mean) / variance)
```

```
# The log unnormalised density to illustrate
log_prob = log_gaussian(0., 1.)

# Points in the abcissa set and corresponding log-probabilities and gradients
xs = np.array([-1., 0.1, 1.5])
hs, dhdxs = log_prob(xs)

# Locations to plot the log unnorm. density and envelope/squeezing functions
x_plot = np.linspace(-2, 2, 200)
log_probs = [log_prob(x)[0] for x in x_plot]
gu = [g_u(x, xs, hs, dhdxs) for x in x_plot]
gl = [g_l(x, xs, hs) for x in x_plot]

# Plot the log unnormalised density, the envelope and squeezing functions
plt.figure(figsize=(6, 3))
plt.scatter(xs, hs, color='k', zorder=3)
plt.plot(x_plot, log_probs, color='black', label='$\log~f = h$')
plt.plot(x_plot, np.log(gu), color='red', label='$\log~g_u$')

# Handle the case of negatively infinite gl, for plotting presentation
floored_log_gl = np.log(np.maximum(np.array(gl), np.ones_like(gl) * 1e-9))
plt.plot(x_plot, floored_log_gl, color='green', label='$\log~g_l$')

# Plot formatting
plt.xlim([-2, 2])
plt.ylim([-3, 1])
plt.xticks([])
plt.yticks([])
plt.xlabel('$x$', fontsize=18)
plt.ylabel('$\log~f(x)$', fontsize=18)
plt.legend()
plt.show()
```



Adaptive rejection sampling

As observed above, by vitrue of the log-concavity of f, the envelope and squeezing functions defined in this way are upper and lower bounds to f. If we then sample a point at random from the area under g_u, and this point also happens to be in the area under f, then the point is uniformly distributed in the area under f, and is an exact sample from the target distribution. Further, if the point happened to lie in the area under the squeezing function g_l, it is certain to also lie in the area under f, so we need not check this latter condition explicitly. This shortcut is particularly useful if the function f is expensive to evaluate, because it avoids some of the evaluations of f. Combining these checks, we arrive at the ARS algorithm below.

Algorithm (Adaptive Rejection Sampling) Given a univariate un-normalised probability density $f(x)$, perform the following initialisation, sampling and update steps:

1. Initialise an abscissa set T_k, such that $f'(x_1) > 0$ and $f'(x_k) < 0$, as well as the corresponding envelope and squeezing functions g_u and g_l. This can be efficiently achieved by starting from an initial guess and stepping out in steps of exponentially increasing size.
2. Sample

$$x' \sim \frac{g_u(x)}{\int g_u(x')dx'} \text{ and } z \sim \text{Unifrom}(0, 1),$$

and perform the following squeezing and rejection tests. If

$$z \leq \frac{g_l(x')}{g_u(x')}$$

holds, then accept x' otherwise perform the following rejection test

$$z \leq \frac{h(x')}{g_u(x')}$$

If this holds, accept the point and otherwise reject it.

3. If x' was accepted at the squeezing test, go to step 2 immediately. Otherwise insert x' into T_k to obtain T_{k+1}, update the piecewise exponential functions g_l and g_u accordingly and then return to step 2.

Below are functions which implement envelope sampling, that is drawing

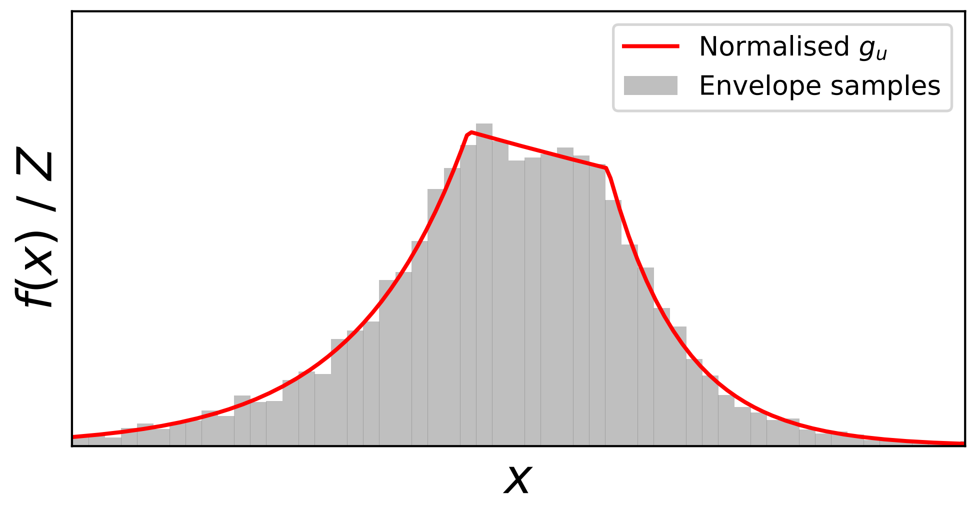
$$x' \sim \frac{g_u(x)}{\int g_u(x')dx'}.$$

The first function determines the left and right limits as well as the the unnormalised probabilities $\int g_{u,k}(x')dx'$ of each piecewise exponential. The second samples uniformly from the area under the envelope function g_u.

```python
def envelope_limits_and_unnormalised_probabilities(xs, hs, dhdxs):

    # Compute the points of intersection of the lines making up the envelope
    z, c = compute_points_of_intersection_and_intercepts(xs, hs, dhdxs)

    # Left-right endpoints for each piece in the piecewise envelope
    limits = np.concatenate([[float('-inf')], z, [float('inf')]])
    limits = np.stack([limits[:-1], limits[1:]], axis=-1)

    probs = (np.exp(dhdxs * limits[:, 1]) - np.exp(dhdxs * limits[:, 0])) * np.exp(c)

    # Catch any intervals where dhdx was zero
    idx_nonzero = np.where(dhdxs != 0.)
    probs[idx_nonzero] = probs[idx_nonzero] / dhdxs[idx_nonzero]

    idx_zero = np.where(dhdxs == 0.)
    probs[idx_zero] = ((limits[:, 1] - limits[:, 0]) * np.exp(c))[idx_zero]

    return limits, probs


def sample_envelope(xs, hs, dhdxs):

    limits, probs = envelope_limits_and_unnormalised_probabilities(xs, hs, dhdxs)
    probs = probs / np.sum(probs)

    # Randomly chosen interval in which the sample lies
    i = np.random.choice(np.arange(probs.shape[0]), p=probs)

    # Sample u = Uniform(0, 1)
    u = np.random.uniform()

    # Invert i^th piecewise exponential CDF to get a sample from that interval
    if dhdxs[i] == 0.:
        return u * (limits[i, 1] - limits[i, 0]) + limits[i, 0]

    else:
        x = np.log(u * np.exp(dhdxs[i] * limits[i, 1]) \
                   + (1 - u) * np.exp(dhdxs[i] * limits[i, 0]))
        x = x / dhdxs[i]

        return x
```

If we draw samples from the envelope defined by the three previous points without the rejection step, we obtain the following distribution.

```
x_plot = np.linspace(-4., 4., 200)
_, probs = envelope_limits_and_unnormalised_probabilities(xs, hs, dhdxs)

samples = [sample_envelope(xs, hs, dhdxs) for i in range(10000)]
gu = [g_u(x, xs, hs, dhdxs) / np.sum(probs) for x in x_plot]

# Plot samples and envelope
plt.figure(figsize=(6, 3))
plt.plot(x_plot,
         gu,
         color='red',
         label='Normalised $g_u$')

plt.hist(samples,
         density=True,
         bins=100,
         color='gray',
         alpha=0.5,
         label='Envelope samples')

# Plot formatting
plt.title('', fontsize=20)
plt.xlim([-4, 4])
plt.ylim([0, 0.5])
plt.xticks([])
plt.yticks([])
plt.xlabel('$x$', fontsize=18)
plt.ylabel('$f(x)~/~Z$', fontsize=18)
plt.legend()
plt.show()
```



We still need to add the initialisation of the abcissa set, the (optional) squeezing test and the rejection test. For the initialisation step, we can start from an initial point, and search to the left and to the right in exponentially increasing step sizes, until we find a point on the left side with positive h' and a point on the right with negative h', and use these as end-points of the abscissa set. The following function `adaptive_rejection_sampling` implements this initialisation step together with the squeezing and rejection tests.

```python
def initialise_abcissa(x0, log_unnorm_prob):

    # Expand to the left/right until the abcissa is correctly initialised
    xs = np.array([x0])
    hs, dhdxs = log_unnorm_prob(xs)

    dx = -1.

    while True:

        if dx < 0. and dhdxs[0] > 0.:
            dx = 1.

        elif dx > 0. and dhdxs[-1] < 0.:
            break

        insert_idx = 0 if dx < 0 else len(xs)

        x = xs[0 if dx < 0 else -1] + dx

        h, dhdx = log_unnorm_prob(x)

        xs = np.insert(xs, insert_idx, x)
        hs = np.insert(hs, insert_idx, h)
        dhdxs = np.insert(dhdxs, insert_idx, dhdx)

        dx = dx * 2

    return xs, hs, dhdxs
```

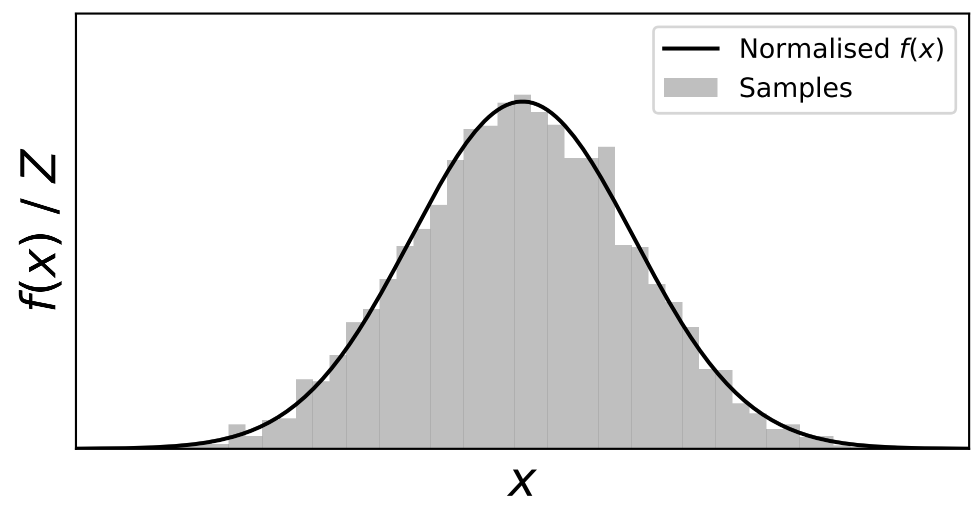
This approach to initialising the abcissa set does not have any tunable parameters, except `x0`. Any initialisation method which guarantees $h'(x_1) > 0$ and $x'(x_K) < 0$ will give a valid abcissa set and this method is only a specific choice. Changing the `x0` value used to the does not significantly affect the efficiency of ARS, since the initialisation method will terminate quickly because of the exponentially increasing step sizes. This implementation assumes that the domain D of f, that is the set of points where f is non-zero, is all of $\mathbb{R}$. If this is not the case, then this initialisation function will fail. A more robust initialisation method could use boolean comparisons of h' values instead of h values, setting $h = -\infty$ outside D, but for the purposes of exposition, this illustration assumes that $D = \mathbb{R}$ and does not bother further with this technical point. Putting the initialisation step together with the squeezing and rejection steps, we arrive at the complete ARS algorithm below.

```python
def adaptive_rejection_sampling(x0, log_unnorm_prob, num_samples):

    xs, hs, dhdxs = initialise_abcissa(x0=x0, log_unnorm_prob=log_unnorm_prob)

    samples = []

    while len(samples) < num_samples:

        x = sample_envelope(xs, hs, dhdxs)

        gl = g_l(x, xs, hs)
        gu = g_u(x, xs, hs, dhdxs)

        # Squeezing test
        u = np.random.rand()

        if u * gu <= gl:
            samples.append(x)

        h, dhdx = log_unnorm_prob(x)

        # Rejection test
        if u * gu <= np.exp(h):
            samples.append(x)

        i = np.searchsorted(xs, x)

        xs = np.insert(xs, i, x)
        hs = np.insert(hs, i, h)
        dhdxs = np.insert(dhdxs, i, dhdx)

    return samples
```

Now we can finally use this function to sample from the example standard Gaussian distribution.

```
np.random.seed(0)

target_mean = 0.
target_variance = 1.

x0 = 1.
num_samples = 10000

log_unnorm_prob = log_gaussian(mean=target_mean, variance=target_variance)

samples = adaptive_rejection_sampling(x0=x0, log_unnorm_prob=log_unnorm_prob,
num_samples=num_samples)
```

```
# Log probabilites for plotting the target
x_plot = np.linspace(-4, 4, 200)
log_probs = [np.exp(log_prob(x)[0]) / (2 * np.pi) ** 0.5 for x in x_plot]

# Plot samples and target
plt.figure(figsize=(6, 3))

plt.hist(samples,
         density=True,
         bins=50,
         color='gray',
         alpha=0.5,
         label='Samples')
plt.plot(x_plot,
         log_probs,
         color='black',
         label='Normalised $f(x)$')

# Plot formatting
plt.xlim([-4, 4])
plt.ylim([0, 0.5])
plt.xticks([])
plt.yticks([])
plt.xlabel('$x$', fontsize=18)
plt.ylabel('$f(x)~/~Z$', fontsize=18)
plt.legend()
plt.show()
```



Conclusions

ARS is an efficient method for sampling log-concave univariate distributions. Although very effective for log-concave one-dimensional differentiable distributions, the algorithm presented here has two shortcomings. First, this algorithm requires gradients of the objective with respect to the input variable. These may be expensive to compute or perhaps even may not exist if f is nowhere differentiable. For this, there exists a modified version of ARS[Gil92] which builds the envelope in a way that does not require gradients. The present page presented the gradient-based method because this is nicer for illustrative purposes. Second, although many distributions of practical interest are log-concave, there are many others which are not. In this case, the ARS algorithm does not apply since the envelope is not guaranteed to entirely contain the probability distribution. For this, there exists an extension of ARS for non-log-concave distributions called the adaptive rejection Metropolis algorithm[GBT95] (ARMS). ARMS also builds an envelope and uses it to propose samples, which are then accepted or rejected using a Metropolis-Hastings step to ensure that the samples are distributed according to the target. Note that in general, ARMS does not produce independent samples from the target, due to the Metropolis-Hastings accept/reject step. For log-concave functions, ARMS reduces to ARS.

References

[Bis06]

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[GW92]

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

[Gil92]

Wally R Gilks. Derivative-free adaptive rejection sampling for gibbs sampling. *Bayesian Statistics*, 1992.

[GBT95]

Wally R Gilks, Nicky G Best, and KKC Tan. Adaptive rejection metropolis sampling within gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 44(4):455–472, 1995