My implementation of Fearnhead and Meligkotsidou.

This is from their 2004 paper "Exact filtering for partially observed continuous time models".

Our goal is to sample $X_0,\dots,X_{K-1}|T=t$ according to the density

$$f(x_0,\dots,x_{K-1}|t) = 1_{\{\sum_{k=0}^{K-1} x_k = t\}} \prod_{k=0}^{K-1} f_{\Gamma(a_k,b_k)}(x_k)$$

where $f_{\Gamma(a,b)}(x) \propto x^{a-1}\exp(-bx)$ is a gamma density.

We do this in several steps.

0.1 Compute $P(N_k = n)$

First, we define the independent random variables N_k, $k=0,\dots,K-1$, by (Cor. 2)

$$P(N_k=n) = \frac{\Gamma(a_k+n)}{\Gamma(a_k)\Gamma(n+1)}\left(\frac{b_k}{b_{K-1}}\right)^{a_k}\left(1-\frac{b_k}{b_{K-1}}\right)^n$$

for $n=0,1,2,\dots$. Note that

$$\log P(N_k=n) = \log(\Gamma(a_k+n)) - \log(\Gamma(a_k)) - \log(\Gamma(n+1)) + a_k\log(b_k) + n\log(b_{K-1}-b_k) - (a_k+n)\log(b_{K-1})$$

If $b_k = b_{K-1}$, the above formula includes a term $n\log 0$ which we define as 0 if $n=0$ and $-\infty$ if $n>0$. In other words, we define $0^0=1$ and $0^n=0$ for $n>0$. We treat this as a special case in the code below.

Recall that we require $b_{K-1} \geq b_{K-2} \geq \cdots \geq b_1 \geq b_0 \geq 0$.

Altogether, we have

$$P(N_k=n) = \begin{cases} 0 & \text{if } b_k = b_{K-1} \text{ and } n>0 \\ \exp\big(\log(\Gamma(a_k+n)) - \log(\Gamma(a_k)) - \log(\Gamma(n+1)) & \\ \quad + a_k\log(b_k) - (a_k+n)\log(b_{K-1})\big) & \text{if } b_k = b_{K-1} \text{ and } n=0 \\ \exp\big(\log(\Gamma(a_k+n)) - \log(\Gamma(a_k)) - \log(\Gamma(n+1)) & \\ \quad + a_k\log(b_k) + n\log(b_{K-1}-b_k) - (a_k+n)\log(b_{K-1})\big) & \text{if } b_k \neq b_{K-1} \end{cases}$$

In code:

⟨*Function to compute the distribution of N_k*⟩≡

```
double dist_N_compute(dist_N *d, size_t k, size_t n)
{
  size_t K = d->K;
  double *a = d->a;
  double *b = d->b;

  if (approx_eq(b[k], b[K-1]) && n > 0) {
    return 0.0;
  } else if (approx_eq(b[k], b[K-1]) && n == 0) {
    return exp(
      lgamma(a[k]+n) - lgamma(a[k]) - lgamma(n+1)
      + a[k]*log(b[k]) - (a[k]+n)*log(b[K-1]));
  } else {
    return exp(
      lgamma(a[k]+n) - lgamma(a[k]) - lgamma(n+1)
      + a[k]*log(b[k]) + n*log(b[K-1]-b[k]) - (a[k]+n)*log(b[K-1]));
  }
}
```

We would like to memoize our computation of the $P(N_k = n)$, since we will need to use these quantities repeatedly later on.

⟨*Function to memoize the distribution of N_k*⟩≡
```
double dist_N_get(dist_N *d, size_t k, size_t n)
{
  assert(k < d->K);
  if (n >= d->max_n) {
    if (!d->is_truncated) {
      d->is_truncated = 1;
      fprintf(stderr, "dist_N_get: warning: at least one n is too big for our table, "
        "expect a slowdown\n");
    }
    return dist_N_compute(d, k, n);
  } else {
    if (gsl_matrix_get(d->table, k, n) == -1.0)
      gsl_matrix_set(d->table, k, n, dist_N_compute(d, k, n));
    return gsl_matrix_get(d->table, k, n);
  }
}
```

We also need some auxiliary definitions:

⟨*Auxiliary definitions for the distribution of N_k*⟩≡
```
typedef struct {
  size_t K;
  size_t max_n;
  char is_truncated;
  double *a;
  double *b;
  gsl_matrix *table;
} dist_N;
dist_N *dist_N_alloc(size_t K, size_t max_n, double *a, double *b)
{
  dist_N *d = malloc(sizeof(dist_N));
  d->K = K;
  d->max_n = max_n;
  d->is_truncated = 0;
  d->a = a;
  d->b = b;
  d->table = gsl_matrix_alloc(K, max_n);
  gsl_matrix_set_all(d->table, -1.0);
  return d;
}
void dist_N_free(dist_N *d)
{
  gsl_matrix_free(d->table);
  free(d);
}
```

⟨*All functions to compute the distribution of N_k*⟩≡
⟨*Auxiliary definitions for the distribution of N_k*⟩
⟨*Function to compute the distribution of N_k*⟩
⟨*Function to memoize the distribution of N_k*⟩
⟨*Function to test our computation of the distribution of N_k*⟩

Finally, we run two sanity checks. (i) We check that (i) $\sum_{n=0}^{n_0} P(N_k = n) \approx 1$. (ii) Note that

$$\begin{aligned}
&\frac{d}{dn}\left(n \mapsto \log P(N_k = n)\right) \\
&= \frac{d}{dn}\log(\Gamma(a_k+n)) - \frac{d}{dn}\log(\Gamma(n+1)) + \frac{d}{dn} n \log(b_{K-1} - b_k) - \frac{d}{dn}(a_k+n)\log(b_{K-1}) \\
&\approx \log(a_k+n) - \log(n+1) + \log(b_{K-1} - b_k) - \log(b_{K-1})
\end{aligned}$$

which is 0 iff

$$n = \frac{a_k(b_K - b_k) - b_K}{b_k}$$

One also can see that $(n \mapsto P(N_k = n))$ is monotone on each side of its maximum. Therefore we check that (a) $P(N_k = n)$ is maximized near the above value of n, (b) it is increasing to the left of this value, and (c) it is decreasing to the right of this value.

⟨*Function to test our computation of the distribution of* N_k⟩≡

```
void dist_N_test(gsl_rng *rng)
{
  size_t K = 3;
  //size_t K = gsl_ran_flat(rng, 3, 10);
  printf("K=%d\n", K);
  size_t max_n = 100000;

  double a[K], b[K];
  for (size_t k = 0; k < K; k++) {
    a[k] = gsl_ran_flat(rng, 0.1, 10.0);
    b[k] = gsl_ran_flat(rng, 0.1, 10.0);
  }
  qsort(b, K, sizeof(double), compare_doubles);
  printf("a: "); dprint(K, a, 1); printf("\n");
  printf("b: "); dprint(K, b, 1); printf("\n");

  dist_N *d = dist_N_alloc(K, max_n, a, b);

  // print
  /*
  for (size_t k = 0; k < K; k++) {
    for (size_t n = 0; n < max_n + 2; n++)
      printf("%f ", dist_N_get(d, k, n));
    printf("\n");
  }
  */

  // (i) check that sum approx 1
  for (size_t k = 0; k < K; k++) {
    double sum = 0.0;
    for (size_t n = 0; n < max_n + 2; n++)
      sum += dist_N_get(d, k, n);
    if (!(sum >= 0.95)) {
      fprintf(stderr, "for k=%zu, sum=%f, sum >= 0.95 fails\n", k, sum);
      exit(1);
    }
  }

  // (ii) check the behavior of P(N_k = n) around its maximimum
```

```
  size_t fudge = 2;

  // (a) check that the maximum is approximately at n_star
  for (size_t k = 0; k < K; k++) {
    double max = -INFINITY;
    size_t maximizer;
    for (size_t n = 0; n < max_n + 2; n++) {
      if (dist_N_get(d, k, n) > max) {
        max = dist_N_get(d, k, n);
        maximizer = n;
      }
    }
    double n_star = GSL_MAX(0, (a[k]*(b[K-1] - b[k]) - b[K-1])/b[k]);
    if (!(abs(maximizer - n_star) <= fudge)) {
      fprintf(stderr, "for k=%zu, n_star=%f, maximizer=%zu, fudge=%zu, n_star is close to maximizer
        k, n_star, maximizer, fudge);
      exit(1);
    }
  }

  // (b) check that the function is monotone increasing below n_star
  for (size_t k = 0; k < K; k++) {
    double prev = -INFINITY;
    double n_star = GSL_MAX(0, (a[k]*(b[K-1] - b[k]) - b[K-1])/b[k]);
    for (size_t n = 0; n < n_star - fudge; n++) {
      if (!(prev <= dist_N_get(d, k, n))) {
        fprintf(stderr, "for k=%zu, n=%zu, prev=%f, cur=%f, prev <= cur fails\n",
          k, n, prev, dist_N_get(d, k, n));
          exit(1);
      }
    }
  }

  // (c) check that the function is monotone decreasing below n_star
  for (size_t k = 0; k < K; k++) {
    double prev = INFINITY;
    double n_star = GSL_MAX(0, (a[k]*(b[K-1] - b[k]) - b[K-1])/b[k]);
    for (size_t n = n_star + fudge; n < max_n + 2; n++) {
      if (!(prev >= dist_N_get(d, k, n))) {
        fprintf(stderr, "for k=%zu, n=%zu, prev=%f, cur=%f, prev >= cur fails\n",
          k, n, prev, dist_N_get(d, k, n));
          exit(1);
      }
    }
  }

  dist_N_free(d);
}
```

1 Main

Finally, we put it all together:

⟨*⟩≡

```
#include <stdio.h>
#include <time.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sf.h>
#include <gsl/gsl_monte_plain.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_math.h>

#define MAX_K 50
#define MAX_N 10000
#define EPSILON 0.01

⟨Auxiliary functions⟩
⟨All functions to compute the distribution of N_k⟩

int main()
{
  gsl_rng *rng;
  init_rng(&rng);

  for (int iter = 0; iter < 100; iter++)
    dist_N_test(rng);
  printf("passed dist_N_test 100 times\n");

  gsl_rng_free(rng);
}
```

2 Auxiliary functions

⟨*Auxiliary functions*⟩≡

```
void die(char *msg)
{
  fprintf(stderr, "fm: fatal error: %s\n", msg);
  exit(1);
}
double dsum(size_t N, double *x, size_t incx) // sum a vector of doubles
{
  double s = 0.0;
  for (size_t i = 0; i < N; i += incx)
    s += x[i];
  return s;
}
void dprint(size_t N, double *x, size_t incx) // print a vector of doubles
{
  size_t i;
  for (i = 0; i+incx < N; i += incx)
    printf("%f ", x[i]);
  printf("%f", x[i]);
}
void init_rng(gsl_rng **rng)
{
  gsl_rng_env_setup();
  const gsl_rng_type *rng_type = gsl_rng_default;
  *rng = gsl_rng_alloc(rng_type);
  gsl_rng_set(*rng, time(0));
}
size_t sample_discrete(gsl_rng *rng, size_t n, double *v)
{
  gsl_ran_discrete_t *dt = gsl_ran_discrete_preproc(n, v);
  size_t sample = gsl_ran_discrete(rng, dt);
  gsl_ran_discrete_free(dt);
  return sample;
}
#define lgamma gsl_sf_lngamma
#define approx_eq(x, y) (abs((x) - (y)) < 1e-5)
int compare_doubles(const void *a, const void *b)
{
  double temp = *((double *)a) - (*(double *)b);
  if (temp > 0)
    return 1;
  else if (temp < 0)
    return -1;
  else
    return 0;
}
```