1. Introduction.

STATLIB
Probability and Statistics Library

John Walker

This program provides facilities for computations involving probability and statistics. A variety of probability distributions, both discrete and continuous, are provided along with a template class *dataTable* which implements a variety of descriptive statistical measures on an arbitrary numeric data type.

This program requires none of the other components of the analysis software suite and may be extracted as a general statistics library. It uses the `DCDFLIB` package for its low-level computations.

References

Abramowitz, Milton and Irene A. Stegun. *Handbook of Mathematical Functions* (Chapter 26). New York: Dover, 1974. ISBN 0-486-61272-4.

Montgomery, Douglas C. and George C. Runger. *Applied Statistics and Probability for Engineers.* New York: John Wiley **S**ons, 1994. ISBN 0-471-54041-2.

Wolfram Research. *Mathematica 4.0 Standard Add-On Packages.* Champaign, IL: Wolfram Media, 1999. ISBN 1-57955-007-X.

⟨ `statlib_test.cc` 1 ⟩ ≡
```
#define REVDATE "15th␣February␣2003"
```
See also section 49.

2. Program global context.

```
#include "config.h"     /* System-dependent configuration */
  ⟨ Preprocessor definitions ⟩
  ⟨ Application include files 4 ⟩
  ⟨ Class implementations 5 ⟩
```

3. We export the class definitions for this package in the external file `statlib.h` that programs which use this library may include.

```
⟨ statlib.h 3 ⟩ ≡
#ifndef STATLIB_HEADER_DEFINES
#define STATLIB_HEADER_DEFINES
#include <math.h>     /* Make sure math.h is available */
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
#include <vector>
#include <algorithm>
  using namespace std;
  ⟨ Class definitions 6 ⟩
#endif
```

4. The following include files provide access to external components of the program not defined herein.

```
⟨ Application include files 4 ⟩ ≡
#include "dcdflib.h"     /* DCDFlib Cumulative Distribution Function library */
#include "statlib.h"     /* Class definitions for this package */
```

This code is used in section 2.

5. The following classes are defined and their implementations provided.

```
⟨ Class implementations 5 ⟩ ≡
  ⟨ Probability distributions 11 ⟩
```

This code is used in section 2.

6. Probability distributions.

We provide the class definitions for computations involving random variables with the following distributions:

normalDistribution	Normal (Gaussian) distribution
poissonDistribution	Poisson distribution
chiSquareDistribution	Chi-Square (χ^2) distribution
gammaDistribution	Gamma distribution
betaDistribution	Beta distribution
tDistribution	Student's t distribution
FDistribution	F distribution
binomialDistribution	Binomial distribution
negativeBinomialDistribution	Negative binomial distribution

Computations are performed using the `DCDFLIB` Double Precision Cumulative Distribution Function Library developed by the Section of Computer Science, Department of Biomathematics, University of Texas M. D. Anderson Hospital. Source code for `DCDFLIB` may be downloaded via:

anonymous FTP

The abstract superclass *probabilityDistribution* is the parent of all of the specific distributions, discrete and continuous. It implements properties common to all distributions and defines pure virtual functions which all specific distributions must implement.

Quantities named in the the following methods are as follows:

`P`	P	Cumulative probability distribution function
`Q`	Q	"Upper tail" probability distribution ($Q = 1 - P$)
`x`	x	Value of random variable
`mean`	μ	Mean value of distribution
`stdev`	σ	Standard deviation
`variance`	σ^2	Variance
`skewness`	γ_1	Coefficient of skewness
`kurtosisExcess`	γ_2	Kurtosis excess
`kurtosis`	β_2	Kurtosis

⟨ Class definitions 6 ⟩ ≡

```
class probabilityDistribution {
private:
  virtual string distributionName(void) = 0;      /* Return distribution name */
public:
  static double Q_from_P(double x)
  {     /* Upper tail probability: Q = 1 - P */
    return 1 - x;
  }
  static double P_from_Q(double x)
  {     /* CDF from upper tail probability: P = 1 - Q */
    return 1 - x;
  }
  virtual double mean(void) = 0;      /* Return mean of distribution */
  virtual double stdev(void) = 0;      /* Return standard deviation σ of distribution */
  double variance(void)
  {     /* Return variance σ^2 of deviation */
    return stdev( ) * stdev( );
  }
  virtual double skewness(void) = 0;      /* Skewness γ1 of distribution */
  virtual double kurtosisExcess(void) = 0;      /* Kurtosis excess γ2 of distribution */
```

```
  double kurtosis(void)
  {     /* Kurtosis (β2 = γ2 + 3) of distribution */
    return kurtosisExcess( ) + 3;
  }
  virtual double CDF_P(double x) = 0;     /* Compute CDF P = ∫_{-∞}^{x} f(x) dx */
  double CDF_Q(double x)
  {     /* Q = 1 − P */
    return Q_from_P(CDF_P(x));
  }     /* Write description of distribution to output stream */
  virtual void writeParameters(ostream &of)
  {
    of ≪ "Distribution:␣" ≪ distributionName( ) ≪ "\n";
    of ≪ "␣␣␣␣Mean␣=␣" ≪ mean( ) ≪ "␣␣Stdev␣=␣" ≪ stdev( ) ≪ "␣␣Variance␣=␣" ≪ variance( ) ≪
        "␣␣Skewness␣=␣" ≪ skewness( ) ≪ "\n";
    of ≪ "␣␣␣␣Kurtosis␣=␣" ≪ kurtosis( ) ≪ "␣␣KurtosisExcess␣=␣" ≪ kurtosisExcess( ) ≪ "\n";
  }
};
```

See also sections 7, 13, 16, 19, 22, 25, 28, 31, 34, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, and 48.

This code is used in section 3.

7. Normal (Gaussian) distribution.

A *normal distribution* describes a random variable x with probability distribution function

$$f_x(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\left(\frac{(x-\mu)^2}{2\sigma^2}\right)}, \qquad -\infty < x < \infty$$

where μ is the *mean value* of the random variable and σ is the *standard deviation*. A *standard normal distribution* is one with $\mu = 0$ and $\sigma = 1$.

A binomial distribution approaches the normal distribution as a limit as the number of trials becomes large. For n trials with probability 0.5, the binomial distribution is approximated by a normal distribution with $\mu = n/2$ and $\sigma = \sqrt{n/4}$.

⟨ Class definitions 6 ⟩ +≡

```
class normalDistribution : public probabilityDistribution {
private:
  virtual string distributionName(void)
  {
    return "normal";
  }
  double mu;      /* Mean value */
  double sigma;     /* Standard deviation */
public:     /* The default constructor creates a standard normal distribution: μ = 0, σ = 1. */
  normalDistribution( )
  {
    set_mu_sigma( );
  }     /* The following constructor creates a normal distribution with a given mean and standard
       deviation. */
  normalDistribution(double c_mu, double c_sigma)
  {
    set_mu_sigma(c_mu, c_sigma);
  }     /* Accessors */
  void set_mu(double c_mu = 0)
  {
    mu = c_mu;
  }
  void set_sigma(double c_sigma = 1)
  {
    sigma = c_sigma;
  }
  void set_mu_sigma(double c_mu = 0, double c_sigma = 1)
  {
    set_mu(c_mu);
    set_sigma(c_sigma);
  }
  double get_mu(void)
  {
    return mu;
  }
  double get_sigma(void)
  {
    return sigma;
```

```
  }     /* Implementations of abstract virtual functions */
  double mean(void)
  {
    return get_mu( );
  }
  double stdev(void)
  {
    return get_sigma( );
  }
  double skewness(void)
  {
    return 0;      /* γ1 = 0 for normal distribution */
  }
  double kurtosisExcess(void)
  {
    return 0;      /* γ2 = 0 for normal distribution */
  }     /* Auxiliary accessors */
  ⟨ Set standard deviation from variance 8 ⟩;
  ⟨ Approximate binomial distribution 9 ⟩;      /* Static transformation functions */
  static double p_from_mu_sigma_x(double mu, double sigma, double x);
  static double x_from_p_mu_sigma(double p, double mu, double sigma);
  static double mu_from_p_x_sigma(double p, double x, double sigma);
  static double sigma_from_p_x_mu(double p, double x, double mu);      /* Analysis methods */
  ⟨ Compute z score 10 ⟩;
  double CDF_P(double x);      /* P = ∫_{-∞}^{x} f(x) dx */
};
```

8. The standard deviation, σ, is the square root of the variance. You can set the standard deviation of the distribution from the variance with the *set_variance* method.

```
⟨ Set standard deviation from variance 8 ⟩ ≡
  void set_variance(double var)
  {
    set_sigma(sqrt(var));
  }
```

This code is used in section 7.

9. A binomial distribution for a large number of trials n each with probability 0.5 is approximated by a normal distribution with $\mu = n/2$ and $\sigma = \sqrt{n/4}$. The *approximateBinomial* method allows you to create such a normal distribution by specifying the number of trials n.

```
⟨ Approximate binomial distribution 9 ⟩ ≡
  void approximateBinomial(unsigned int n)
  {
    set_mu_sigma(n/2, sqrt(n/4.0));
  }
```

This code is used in section 7.

10. The z score for a given value in a normal distribution is given by

$$z = \frac{m - \mu}{\sigma}.$$

These z values obey the χ^2 distribution as does the summation of a number of independent z scores.

⟨ Compute z score 10 ⟩ ≡

```
double z_score(double m)
{
  return (m − mean( ))/stdev( );
}
```

This code is used in sections 7 and 31.

11. The following transformation functions are **static** methods of the **normalDistribution** class. They permit calculation of properties of the normal distribution without reference to a particular **normalDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`mu`	μ	Mean value
`sigma`	σ	Standard deviation

⟨ Probability distributions 11 ⟩ ≡

```
double normalDistribution::p_from_mu_sigma_x(double mu, double sigma, double x)
{
  double p, q, bound;
  int which = 1, status;
  cdfnor(&which, &p, &q, &x, &mu, &sigma, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("normalDistribution::p_from_mu_sigma_x: Result out of bounds"));
  }
  return p;
}
double normalDistribution::x_from_p_mu_sigma(double p, double mu, double sigma)
{
  double q, x, bound;
  int which = 2, status;
  q = 1 − p;
  cdfnor(&which, &p, &q, &x, &mu, &sigma, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("normalDistribution::x_from_p_mu_sigma: Result out of bounds"));
  }
  return x;
}
double normalDistribution::mu_from_p_x_sigma(double p, double x, double sigma)
{
  double q, mu, bound;
  int which = 3, status;
  q = 1 − p;
  cdfnor(&which, &p, &q, &x, &mu, &sigma, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("normalDistribution::mu_from_p_x_sigma: Result out of bounds"));
  }
  return mu;
}
double normalDistribution::sigma_from_p_x_mu(double p, double x, double mu)
{
  double q, sigma, bound;
  int which = 4, status;
  q = 1 − p;
  cdfnor(&which, &p, &q, &x, &mu, &sigma, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("normalDistribution::sigma_from_p_x_mu: Result out of bounds"));
```

```
    }
    return sigma;
  }
```

See also sections 12, 14, 15, 17, 18, 20, 21, 23, 24, 26, 27, 29, 30, 32, 33, 35, and 36.

This code is used in section 5.

12. The *cumulative distribution function* (CDF) P value is the probability a given value x of the random variable will occur. It is computed by integrating the probability density function $f(x)$ from $-\infty$ to x:

$$P = \int_{-\infty}^{x} f(x)\,dx.$$

⟨ Probability distributions 11 ⟩ +≡

```
double normalDistribution::CDF_P(double x)
{
  return p_from_mu_sigma_x(mu, sigma, x);
}
```

13. Chi-Square (χ^2) distribution.

A random variable has a *chi-square* (χ^2) distribution if it follows the probability distribution function

$$f(x) = \frac{1}{2^{k/2}\Gamma\left(\frac{k}{2}\right)} x^{(k/2)-1} e^{-x/2}, \qquad x > 0$$

where k is the *degrees of freedom* of the distribution and the Γ is the gamma function—the generalisation of factorial to real arguments

$$\Gamma(n) = \int_0^\infty x^{n-1} e^{-x}\, dx, \qquad n > 0.$$

The χ^2 distribution is a special case of the *gamma distribution*

$$f_x(x; \lambda, r) = \frac{\lambda^r x^{r-1} e^{-\lambda x}}{\Gamma(r)}, \qquad x > 0, \lambda > 0, r > 0$$

when $\lambda = 1/2$ and r is the degrees of freedom k divided by 2.

The χ^2 distribution is applicable when combining the results of a series of independent experiments whose outcomes are normally distributed. If $Z_1, Z_2, \ldots Z_k$ are the results of k independent experiments normalised so that their mean $\mu = 0$ and standard deviation $\sigma = 1$, then the sum X of the squares of these outcomes:

$$X = \sum_{i=1}^{k} {Z_i}^2$$

will be χ^2 distributed with k degrees of freedom, χ^2_k.

⟨ Class definitions 6 ⟩ +≡

```
class chiSquareDistribution : public probabilityDistribution {
private:
  virtual string distributionName(void)
  {
    return "chi-square";
  }
  double k;      /* Degrees of freedom */
public:      /* The default constructor creates a χ² distribution with k = 1. */
  chiSquareDistribution( )
  {
    set_k( );
  }
  chiSquareDistribution(double c_k = 1)
  {
    set_k(c_k);
  }      /* Accessors */
  void set_k(double c_k = 1)
  {
    k = c_k;
  }
  double get_k(void)
  {
    return k;
  }      /* Implementations of abstract virtual functions */
  double mean(void)
```

```
  {
    return get_k( );
  }
  double stdev(void)
  {
    return sqrt(2 * k);      /* σ = √(2k) */
  }
  double skewness(void)
  {
    return sqrt(8/k);      /* γ1 = √(8/k) */
  }
  double kurtosisExcess(void)
  {
    return 12/k;      /* γ2 = 12/k */
  }     /* Static transformation functions */
  static double p_from_k_x(double k, double x);
  static double x_from_p_k(double p, double k);
  static double k_from_p_x(double p, double x);      /* Analysis methods */
  double CDF_P(double x);      /* P = ∫_{-∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "␣␣␣␣Degrees␣of␣freedom␣=␣" << k << "\n";
  }
};
```

14. The following transformation functions are **static** methods of the **chiSquareDistribution** class. They permit calculation of properties of the χ^2 distribution without reference to a particular **chiSquareDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`k`	k	Degrees of freedom

⟨ Probability distributions 11 ⟩ +≡

```
double chiSquareDistribution::p_from_k_x(double k, double x)
{
  double p, q, bound;
  int which = 1, status;
  cdfchi(&which, &p, &q, &x, &k, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("chiSquareDistribution::p_from_k_x: Result out of bounds"));
  }
  return p;
}
double chiSquareDistribution::x_from_p_k(double p, double k)
{
  double q, x, bound;
  int which = 2, status;
  q = 1 − p;
  cdfchi(&which, &p, &q, &x, &k, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("chiSquareDistribution::x_from_p_k: Result out of bounds"));
  }
  return x;
}
double chiSquareDistribution::k_from_p_x(double p, double x)
{
  double q, k, bound;
  int which = 3, status;
  q = 1 − p;
  cdfchi(&which, &p, &q, &x, &k, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("chiSquareDistribution::k_from_p_x: Result out of bounds"));
  }
  return k;
}
```

15. The *cumulative distribution function* (CDF) P value is the probability a given value x of the random variable will occur. It is computed by integrating the probability density function $f(x)$ from $-\infty$ to x:

$$P = \int_{-\infty}^{x} f(x)\,dx.$$

⟨ Probability distributions 11 ⟩ +≡

```
double chiSquareDistribution::CDF_P(double x)
{
  return p_from_k_x(k, x);
}
```

16. Gamma distribution.
The probability density function of the Gamma distribution is

$$f_x(x;\alpha,\lambda) = \frac{\lambda^\alpha x^{\alpha-1} e^{-\lambda x}}{\Gamma(\alpha)}, \qquad x > 0.$$

α is referred to as the *shape parameter* and λ the *scale parameter*. Some references use r instead of α for the shape parameter, and others specify the scale parameter as the reciprocal of the value used here. A chi-square distribution with k degrees of freedom is a special case of a gamma distribution with $\lambda = 1/2$ and $r = k/2$.

```
⟨ Class definitions 6 ⟩ +≡
  class gammaDistribution : public probabilityDistribution {
  private:
    virtual string distributionName(void)
    {
      return "gamma";
    }
    double alpha;      /* Shape parameter α */
    double lambda;     /* Scale parameter λ */
  public:     /* The default constructor creates a gamma distribution with α = 10, λ = 10. */
    gammaDistribution( )
    {
      set_alpha_lambda( );
    }
    gammaDistribution(double c_alpha = 10, double c_lambda = 10)
    {
      set_alpha_lambda(c_alpha, c_lambda);
    }     /* Accessors */
    void set_alpha(double c_alpha = 10)
    {
      alpha = c_alpha;
    }
    double get_alpha(void)
    {
      return alpha;
    }
    void set_lambda(double c_lambda = 10)
    {
      lambda = c_lambda;
    }
    double get_lambda(void)
    {
      return lambda;
    }
    void set_alpha_lambda(double c_alpha = 10, double c_lambda = 10)
    {
      set_alpha(c_alpha);
      set_lambda(c_lambda);
    }     /* Implementations of abstract virtual functions */
    double mean(void)
```

```
  {
    return alpha/lambda;      /* μ = α/λ */
  }
  double stdev(void)
  {
    return sqrt(alpha)/lambda;      /* σ = √α/λ */
  }
  double skewness(void)
  {
    return 2/sqrt(alpha);      /* γ1 = 2/sqrtα */
  }
  double kurtosisExcess(void)
  {
    return 6/alpha;      /* γ2 = 6/α */
  }     /* Static transformation functions */
  static double p_from_alpha_lambda_x(double alpha, double lambda, double x);
  static double x_from_p_alpha_lambda(double p, double alpha, double lambda);
  static double alpha_from_p_lambda_x(double p, double lambda, double x);
  static double lambda_from_p_alpha_x(double p, double alpha, double x);
  double CDF_P(double x);      /* P = ∫_{-∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "␣␣␣␣Shape␣(alpha)␣=␣" << alpha << "␣␣Scale␣(lambda)␣␣=␣" << lambda << "\n";
  }
};
```

17. The following transformation functions are **static** methods of the **gammaDistribution** class. They permit calculation of properties of the Gamma distribution without reference to a particular **gammaDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`alpha`	α	Shape parameter
`lambda`	λ	Scale parameter

⟨ Probability distributions 11 ⟩ +≡

```
double gammaDistribution::p_from_alpha_lambda_x(double alpha, double lambda, double x)
{
  double p, q, bound;
  int which = 1, status;
  cdfgam(&which, &p, &q, &x, &alpha, &lambda, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("gammaDistribution::p_from_alpha_lambda_x: Result out of bou\
        nds"));
  }
  return p;
}
double gammaDistribution::x_from_p_alpha_lambda(double p, double alpha, double lambda)
{
  double q, x, bound;
  int which = 2, status;
  q = 1 − p;
  cdfgam(&which, &p, &q, &x, &alpha, &lambda, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("gammaDistribution::x_from_p_alpha_lambda: Result out of bou\
        nds"));
  }
  return x;
}
double gammaDistribution::alpha_from_p_lambda_x(double p, double lambda, double x)
{
  double q, alpha, bound;
  int which = 3, status;
  q = 1 − p;
  cdfgam(&which, &p, &q, &x, &alpha, &lambda, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("gammaDistribution::alpha_from_p_lambda_x: Result out of bou\
        nds"));
  }
  return alpha;
}
double gammaDistribution::lambda_from_p_alpha_x(double p, double alpha, double x)
{
  double q, lambda, bound;
  int which = 4, status;
  q = 1 − p;
```

```
    cdfgam(&which, &p, &q, &x, &alpha, &lambda, &status, &bound);
    if (status ≠ 0) {
      throw (out_of_range("gammaDistribution::lambda_from_p_alpha_x: Result out of bou\
          nds"));
    }
    return lambda;
  }
```

18. The *cumulative distribution function* (CDF) P value is the probability of n successes in x trials.

```
⟨ Probability distributions 11 ⟩ +≡
  double gammaDistribution::CDF_P(double x)
  {
    return p_from_alpha_lambda_x(alpha, lambda, x);
  }
```

19. Beta distribution.

If X_a and X_b are independent gamma distributions with shape (α) parameters a and b respectively and identical scale (λ) parameters, then the random variable

$$\frac{X_a}{X_a + X_b}$$

will obey the *beta distribution* whose probability density function is

$$f(x; a, b) = \frac{(1-x)^{b-1} x^{a-1}}{B(a, b)}$$

where $B(a, b)$ is the *Euler beta function*

$$B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

⟨ Class definitions 6 ⟩ +≡

```
class betaDistribution : public probabilityDistribution {
private:
  virtual string distributionName(void)
  {
    return "beta";
  }
  double a;      /* Shape parameter of gamma variable X_1 */
  double b;      /* Scale parameter of gamma variable X_2 */
public:      /* The default constructor creates a beta distribution with a = 5, b = 10. */
  betaDistribution( )
  {
    set_a_b( );
  }
  betaDistribution(double c_a = 5, double c_b = 10)
  {
    set_a_b(c_a, c_b);
  }      /* Accessors */
  void set_a(double c_a = 5)
  {
    a = c_a;
  }
  double get_a(void)
  {
    return a;
  }
  void set_b(double c_b = 10)
  {
    b = c_b;
  }
  double get_b(void)
  {
    return b;
  }
```

```
  void set_a_b(double c_a = 5, double c_b = 10)
  {
    set_a(c_a);
    set_b(c_b);
  }     /* Implementations of abstract virtual functions */
  double mean(void)
  {
    return a/(a + b);       /* μ = a/(a + b) */
  }
  double stdev(void)
  {
    return sqrt(a * b)/((a + b) * sqrt(a + b + 1));       /* σ = √(ab) / ((a+b)√(a+b+1)) */
  }
  double skewness(void)
  {
    return (2 * (b − a) * sqrt(a + b + 1))/(sqrt(a * b) * (a + b + 2));       /* γ1 = 2(b−a)√(a+b+1) / (√(ab)(a+b+2)) */
  }
  double kurtosisExcess(void)
  {
    return ((3*(a+b+1)*((a*b*(a+b−6))+(2*(a+b)*(a+b))))/(a*b*(a+b+2)*(a+b+3)))−3;
        /* γ2 = 3(a+b+1)((ab(a+b−6))+(2(a+b)^2)) / ab(a+b+2)(a+b+3) − 3 */
  }     /* Static transformation functions */
  static double p_from_a_b_x(double a, double b, double x);
  static double x_from_p_a_b(double p, double a, double b);
  static double a_from_p_b_x(double p, double b, double x);
  static double b_from_p_a_x(double p, double a, double x);
  double CDF_P(double x);       /* P = ∫_{−∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "␣␣␣␣a␣=␣" << a << "␣␣b␣=␣" << b << "\n";
  }
};
```

20. The following transformation functions are **static** methods of the **betaDistribution** class. They permit calculation of properties of the Beta distribution without reference to a particular **betaDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`a`	a	Shape parameter of first Gamma distribution
`b`	b	Shape parameter second Gamma distribution

⟨ Probability distributions 11 ⟩ +≡

```
double betaDistribution::p_from_a_b_x(double a, double b, double x)
{
  double p, q, y, bound;
  int which = 1, status;
  y = 1 - x;
  cdfbet(&which, &p, &q, &x, &y, &a, &b, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("betaDistribution::p_from_a_b_x: Result out of bounds"));
  }
  return p;
}
double betaDistribution::x_from_p_a_b(double p, double a, double b)
{
  double q, x, y, bound;
  int which = 2, status;
  q = 1 - p;
  cdfbet(&which, &p, &q, &x, &y, &a, &b, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("betaDistribution::x_from_p_a_b: Result out of bounds"));
  }
  return x;
}
double betaDistribution::a_from_p_b_x(double p, double b, double x)
{
  double a, q, y, bound;
  int which = 3, status;
  q = 1 - p;
  y = 1 - x;
  cdfbet(&which, &p, &q, &x, &y, &a, &b, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("betaDistribution::a_from_p_b_x: Result out of bounds"));
  }
  return a;
}
double betaDistribution::b_from_p_a_x(double p, double a, double x)
{
  double b, q, y, bound;
  int which = 4, status;
  q = 1 - p;
  y = 1 - x;
```

```
    cdfbet(&which, &p, &q, &x, &y, &a, &b, &status, &bound);
    if (status ≠ 0) {
      throw (out_of_range("betaDistribution::b_from_p_a_x: Result out of bounds"));
    }
    return b;
  }
```

21. The *cumulative distribution function* (CDF) P value is the probability of n successes in x trials.

```
⟨ Probability distributions 11 ⟩ +≡
  double betaDistribution::CDF_P(double x)
  {
    return p_from_a_b_x(a, b, x);
  }
```

22. Student's t distribution.

If Z is a random variable with a standard normal distribution ($\mu = 0, \sigma = 1$) and V is a chi-square random variable, the random variable

$$T = \frac{Z}{\sqrt{V/k}}$$

will have the probability density function

$$f(x) = \frac{\Gamma((k+1)/2)}{\sqrt{\pi k}\Gamma(k/2)} \cdot \frac{1}{((x^2/k)+1)^{(k+1)/2}}, \qquad -\infty < x < \infty, k > 2$$

where k is the *degrees of freedom* of the distribution V and Γ is the gamma function. This is referred to as a t distribution with k degrees of freedom.

The t distribution is useful in determining confidence intervals and testing hypotheses about the mean of samples taken from a normal distribution.

⟨ Class definitions 6 ⟩ +≡

```
class tDistribution : public probabilityDistribution {
private:
  virtual string distributionName(void)
  {
    return "t";
  }
  double k;      /* Degrees of freedom */
public:      /* The default constructor creates a t distribution with k = 3. */
  tDistribution( )
  {
    set_k( );
  }
  tDistribution(double c_k = 3)
  {
    set_k(c_k);
  }      /* Accessors */
  void set_k(double c_k = 3)
  {
    k = c_k;
  }
  double get_k(void)
  {
    return k;
  }      /* Implementations of abstract virtual functions */
  double mean(void)
  {
    return 0;
  }
  double stdev(void)
  {
    return sqrt(k/(k - 2));      /* σ = √(k/(k − 2)) */
  }
  double skewness(void)
  {
    return 0;      /* γ1 = 0 */
```

```
  }
  double kurtosisExcess(void)
  {
    return 6.0/(k − 4);      /* γ2 = 6/(k − 4) */
  }     /* Static transformation functions */
  static double p_from_k_x(double k, double x);
  static double x_from_p_k(double p, double k);
  static double k_from_p_x(double p, double x);      /* Analysis methods */
  double CDF_P(double x);      /* P = ∫_{−∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of ≪ "␣␣␣␣Degrees␣of␣freedom␣=␣" ≪ k ≪ "\n";
  }
};
```

23. The following transformation functions are **static** methods of the **tDistribution** class. They permit calculation of properties of the t distribution without reference to a particular **tDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`k`	k	Degrees of freedom

⟨ Probability distributions 11 ⟩ +≡

```
double tDistribution::p_from_k_x(double k, double x)
{
  double p, q, bound;
  int which = 1, status;
  cdft(&which, &p, &q, &x, &k, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("tDistribution::p_from_k_x: Result out of bounds"));
  }
  return p;
}
double tDistribution::x_from_p_k(double p, double k)
{
  double q, x, bound;
  int which = 2, status;
  q = 1 − p;
  cdft(&which, &p, &q, &x, &k, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("tDistribution::x_from_p_k: Result out of bounds"));
  }
  return x;
}
double tDistribution::k_from_p_x(double p, double x)
{
  double q, k, bound;
  int which = 3, status;
  q = 1 − p;
  cdft(&which, &p, &q, &x, &k, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("tDistribution::k_from_p_x: Result out of bounds"));
  }
  return k;
}
```

24. The *cumulative distribution function* (CDF) P value is the probability a given value x of the random variable will occur. It is computed by integrating the probability density function $f(x)$ from $-\infty$ to x:

$$P = \int_{-\infty}^{x} f(x)\,dx.$$

⟨ Probability distributions 11 ⟩ +≡

```
double tDistribution::CDF_P(double x)
{
  return p_from_k_x(k, x);
}
```

25. F distribution.

If A and B are independent random variables with chi-square distributions with u and v degrees of freedom respectively, the ratio of these variables divided by their degrees of freedom

$$\frac{A/u}{B/v}$$

follows the F distribution with probability density function

$$f_x(x;u,v) = \frac{\Gamma\left(\frac{u+v}{2}\right)\left(\frac{u}{v}\right)^{u/2} x^{u/2-1}}{\Gamma\left(\frac{u}{2}\right)\Gamma\left(\frac{v}{2}\right)\left[\left(\frac{u}{v}\right)x+1\right]^{(u+v)/2}}, \qquad 0 < x < \infty.$$

⟨ Class definitions 6 ⟩ +≡

```
class FDistribution : public probabilityDistribution {
private:
  virtual string distributionName(void)
  {
    return "F";
  }
  double u;      /* Numerator degrees of freedom */
  double v;      /* Denominator degrees of freedom */
public:      /* The default constructor creates an F distribution with u = 8, v = 8. */
  FDistribution( )
  {
    set_u_v( );
  }
  FDistribution(double c_u = 8, double c_v = 8)
  {
    set_u_v(c_u, c_v);
  }      /* Accessors */
  void set_u(double c_u = 8)
  {
    u = c_u;
  }
  double get_u(void)
  {
    return u;
  }
  void set_v(double c_v = 8)
  {
    v = c_v;
  }
  double get_v(void)
  {
    return v;
  }
  void set_u_v(double c_u = 8, double c_v = 8)
  {
    set_u(c_u);
```

```
    set_v(c_v);
  }    /* Implementations of abstract virtual functions */
  double mean(void)
  {
    return v/(v - 2);
  }
  double stdev(void)
  {
    return sqrt((2 * (v * v) * (u + v - 2))/(u * ((v - 2) * (v - 2)) * (v - 4)));    /* σ = √(2v²(u+v−2)/(u(v−2)²(v−4))) */
  }
  double skewness(void)
  {
    return (2 * sqrt(2.0) * sqrt(v - 4) * (2 * u + v - 2))/(sqrt(u) * (v - 6) * sqrt(u + v - 2));
      /* γ1 = 2√2√(v−4)(2u+v−2)/(√u(v−6)√(u+v−2)) */
  }
  double kurtosisExcess(void)
  {
    return (12 * ((v - 4) * (v - 2) * (v - 2) + u * (u + v - 2) * (5 * v - 22)))/(u * (v - 8) * (v - 6) * (u + v - 2));
      /* γ2 = 12(v−4)(v−2)²+u(u+v−2)(5v−22)/(u(v−8)(v−6)(u+v−2)) */
  }    /* Static transformation functions */
  static double p_from_u_v_x(double u, double v, double x);
  static double x_from_p_u_v(double p, double u, double v);
  static double u_from_p_v_x(double p, double v, double x);
  static double v_from_p_u_x(double p, double u, double x);
  double CDF_P(double x);    /* P = ∫_{−∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "␣␣␣␣Numerator␣DF␣=␣" << u << "␣␣Denominator␣DF␣␣=␣" << v << "\n";
  }
};
```

26. The following transformation functions are **static** methods of the **FDistribution** class. They permit calculation of properties of the F distribution without reference to a particular **FDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`u`	u	Numerator degrees of freedom
`v`	v	Denominator degrees of freedom

⟨ Probability distributions 11 ⟩ +≡

```
double FDistribution::p_from_u_v_x(double u, double v, double x)
{
  double p, q, bound;
  int which = 1, status;
  cdff(&which, &p, &q, &x, &u, &v, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("FDistribution::p_from_u_v_x: Result out of bounds"));
  }
  return p;
}
double FDistribution::x_from_p_u_v(double p, double u, double v)
{
  double q, x, bound;
  int which = 2, status;
  q = 1 − p;
  cdff(&which, &p, &q, &x, &u, &v, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("FDistribution::x_from_p_u_v: Result out of bounds"));
  }
  return x;
}
double FDistribution::u_from_p_v_x(double p, double v, double x)
{
  double q, u, bound;
  int which = 3, status;
  q = 1 − p;
  cdff(&which, &p, &q, &x, &u, &v, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("FDistribution::u_from_p_v_x: Result out of bounds"));
  }
  return u;
}
double FDistribution::v_from_p_u_x(double p, double u, double x)
{
  double q, v, bound;
  int which = 4, status;
  q = 1 − p;
  cdff(&which, &p, &q, &x, &u, &v, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("FDistribution::v_from_p_u_x: Result out of bounds"));
  }
```

```
  return v;
}
```

27. The *cumulative distribution function* (CDF) P value is the probability of n successes in x trials.

```
⟨ Probability distributions 11 ⟩ +≡
double FDistribution::CDF_P(double x)
{
  return p_from_u_v_x(u, v, x);
}
```

28. Poisson distribution.

The *Poisson distribution* gives the probability of an event with an arrival rate λ for a given interval of occurring in an period of x intervals. The the probability density function of the Poisson distribution is

$$f(x;\lambda) = \frac{e^{-\lambda}\lambda^x}{x!}, \qquad x = 0, 1, 2 \ldots .$$

```
⟨ Class definitions 6 ⟩ +≡
  class poissonDistribution : public probabilityDistribution {
  private:
    virtual string distributionName(void)
    {
      return "poisson";
    }
    double lambda;      /* Arrival rate */
  public:      /* The default constructor creates a poisson distribution with λ = 0.5. */
    poissonDistribution( )
    {
      set_lambda( );
    }
    poissonDistribution(double c_lambda = 0.5)
    {
      set_lambda(c_lambda);
    }      /* Accessors */
    void set_lambda(double c_lambda = 0.5)
    {
      lambda = c_lambda;
    }
    double get_lambda(void)
    {
      return lambda;
    }      /* Implementations of abstract virtual functions */
    double mean(void)
    {
      return lambda;
    }
    double stdev(void)
    {
      return sqrt(lambda);      /* σ = √λ */
    }
    double skewness(void)
    {
      return 1/sqrt(lambda);      /* γ1 = 1/√λ */
    }
    double kurtosisExcess(void)
    {
      return 1/lambda;      /* γ2 = 1/λ */
    }      /* Static transformation functions */
    static double p_from_lambda_x(double lambda, double x);
    static double x_from_p_lambda(double p, double lambda);
```

```
  static double lambda_from_p_x(double p, double x);      /* Analysis methods */
  double CDF_P(double x);      /* P = ∫_{-∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "    Arrival rate = " << lambda << "\n";
  }
};
```

29. The following transformation functions are **static** methods of the **poissonDistribution** class. They permit calculation of properties of the Poisson distribution without reference to a particular **poissonDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`x`	x	Value of random variable
`lambda`	λ	Arrival rate

⟨ Probability distributions 11 ⟩ +≡

```
double poissonDistribution::p_from_lambda_x(double lambda, double x)
{
  double p, q, bound;
  int which = 1, status;
  cdfpoi(&which, &p, &q, &x, &lambda, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("poissonDistribution::p_from_lambda_x: Result out of bounds"));
  }
  return p;
}
double poissonDistribution::x_from_p_lambda(double p, double lambda)
{
  double q, x, bound;
  int which = 2, status;
  q = 1 − p;
  cdfpoi(&which, &p, &q, &x, &lambda, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("poissonDistribution::x_from_p_lambda: Result out of bounds"));
  }
  return x;
}
double poissonDistribution::lambda_from_p_x(double p, double x)
{
  double q, lambda, bound;
  int which = 3, status;
  q = 1 − p;
  cdfpoi(&which, &p, &q, &x, &lambda, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("poissonDistribution::lambda_from_p_x: Result out of bounds"));
  }
  return lambda;
}
```

30. The *cumulative distribution function* (CDF) P value is the probability a given value x of the random variable will occur. It is computed by integrating the probability density function $f(x)$ from $-\infty$ to x:

$$P = \int_{-\infty}^{x} f(x)\,dx.$$

⟨ Probability distributions 11 ⟩ +≡

```
double poissonDistribution::CDF_P(double x)
{
  return p_from_lambda_x(lambda, x);
}
```

31. Binomial distribution.

An experiment consisting of n independent trials, each with probability r of success, will follow a *binomial distribution* with probability density function

$$f_x(x; r, n) = \binom{n}{x} r^x (1-r)^{n-x}, \qquad n = 1, 2, \ldots \; x = 0, 1, \ldots, n$$

where the notation

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

represents the number of ways of choosing n items from a set of x. (We use r to represent the probability in the above equations instead of the customary p to avoid confusion with P, the value of the cumulative distribution function.) For an experiment with equal probability of success or failure such as flipping a coin, the probability r is 0.5.

As the number of trials becomes large, the binomial distribution approaches the normal distribution; in such circumstances you may wish to use the *approximateBinomial* method of the **normalDistribution** class.

⟨ Class definitions 6 ⟩ +≡

```
class binomialDistribution : public probabilityDistribution {
private:
  virtual string distributionName(void)
  {
    return "binomial";
  }
  double n;      /* Number of trials */
  double r;      /* Probability of success in each trial */
public:      /* The default constructor creates a binomial distribution with n = 2, r = 0.5. */
  binomialDistribution( )
  {
    set_n_r( );
  }
  binomialDistribution(double c_n = 2, double c_r = 0.5)
  {
    set_n_r(c_n, c_r);
  }      /* Accessors */
  void set_n(double c_n = 2)
  {
    n = c_n;
  }
  double get_n(void)
  {
    return n;
  }
  void set_r(double c_r = 0.5)
  {
    r = c_r;
  }
  double get_r(void)
  {
    return r;
```

```
  }
  void set_n_r(double c_n = 2, double c_r = 0.5)
  {
    set_n(c_n);
    set_r(c_r);
  }     /* Implementations of abstract virtual functions */
  double mean(void)
  {
    return n * r;
  }
  double stdev(void)
  {
    return sqrt(n * r * (1 - r));       /* σ = √(nr(1 − r)) */
  }
  double skewness(void)
  {
    return (1 - (2 * r))/stdev( );      /* γ1 = (1−2r)/σ */
  }
  double kurtosisExcess(void)
  {
    return (1 - (6 * (1 - r) * r))/(n * (1 - r) * r);      /* γ2 = (1−6(1−r)r)/(n(1−r)r) */
  }     /* Static transformation functions */
  static double p_from_n_r_s(double n, double r, double s);
  static double s_from_p_r_n(double p, double r, double n);
  static double n_from_p_r_s(double p, double r, double s);
  static double r_from_p_n_s(double p, double n, double s);     /* Analysis methods */
  ⟨ Compute z score 10 ⟩;
  double CDF_P(double x);      /* P = ∫_{−∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "␣␣␣␣Trials␣=␣" << n << "␣␣Probability␣␣=␣" << r << "\n";
  }
};
```

32. The following transformation functions are **static** methods of the **binomialDistribution** class. They permit calculation of properties of the binomial distribution without reference to a particular **binomialDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`n`	n	Number of trials
`r`	r	Probability of success in each trial
`s`	s	Number of successes

⟨ Probability distributions 11 ⟩ +≡

```
double binomialDistribution::p_from_n_r_s(double n, double r, double s)
{
  double p, q, ri, bound;
  int which = 1, status;
  ri = 1 - r;
  cdfbin(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("binomialDistribution::p_from_n_r_s: Result out of bounds"));
  }
  return p;
}
double binomialDistribution::s_from_p_r_n(double p, double r, double n)
{
  double q, ri, s, bound;
  int which = 2, status;
  q = 1 - p;
  ri = 1 - r;
  cdfbin(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("binomialDistribution::s_from_p_r_n: Result out of bounds"));
  }
  return s;
}
double binomialDistribution::n_from_p_r_s(double p, double r, double s)
{
  double q, n, ri, bound;
  int which = 3, status;
  q = 1 - p;
  ri = 1 - r;
  cdfbin(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("binomialDistribution::n_from_p_r_s: Result out of bounds"));
  }
  return n;
}
double binomialDistribution::r_from_p_n_s(double p, double n, double s)
{
  double q, r, ri, bound;
  int which = 4, status;
  q = 1 - p;
```

```
    cdfbin(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
    if (status ≠ 0) {
      throw (out_of_range("binomialDistribution::r_from_p_n_s: Result out of bounds"));
    }
    return r;
  }
```

33. The *cumulative distribution function* (CDF) P value is the probability a given value x of the random variable will occur. It is computed by integrating the probability density function $f(x)$ from $-\infty$ to x:

$$P = \int_{-\infty}^{x} f(x)\,dx.$$

```
⟨ Probability distributions 11 ⟩ +≡
  double binomialDistribution::CDF_P(double x)
  {
    return p_from_n_r_s(n, r, x);
  }
```

34. Negative binomial distribution.

The *negative binomial distribution* for a series of independent trials each with a probability of success r gives the probability of n successes in a sequence of x trials. The probability density function is

$$f_x(x; n, r) = \binom{x-1}{n-1}(1-r)^{x-n}r^n, \qquad n = 1, 2, \ldots \; x = n, n+1, \ldots .$$

(We use r to represent the probability in the above equations instead of the customary p to avoid confusion with P, the value of the cumulative distribution function.) For an experiment with equal probability of success or failure such as flipping a coin, the probability r is 0.5. Note that the probability is necessarily zero when $x < n$ (you can't have more successes than trials)!

```
⟨ Class definitions 6 ⟩ +≡
  class negativeBinomialDistribution : public probabilityDistribution {
  private:
    virtual string distributionName(void)
    {
      return "negative␣binomial";
    }
    double n;      /* Number of successes */
    double r;      /* Probability of success in each trial */
  public:      /* The default constructor creates a negative binomial distribution with n = 2, r = 0.5. */
    negativeBinomialDistribution( )
    {
      set_n_r( );
    }
    negativeBinomialDistribution(double c_n = 2, double c_r = 0.5)
    {
      set_n_r(c_n, c_r);
    }      /* Accessors */
    void set_n(double c_n = 2)
    {
      n = c_n;
    }
    double get_n(void)
    {
      return n;
    }
    void set_r(double c_r = 0.5)
    {
      r = c_r;
    }
    double get_r(void)
    {
      return r;
    }
    void set_n_r(double c_n = 2, double c_r = 0.5)
    {
      set_n(c_n);
      set_r(c_r);
    }      /* Implementations of abstract virtual functions */
```

```
  double mean(void)
  {
    return (n * (1 - r))/r;
  }
  double stdev(void)
  {
    return sqrt(n * (1 - r))/r;        /* σ = √(n(1−r)) / r */
  }
  double skewness(void)
  {
    return (2 - r)/sqrt(n * (1 - r));        /* γ1 = (2−r) / √(n(1−r)) */
  }
  double kurtosisExcess(void)
  {
    return (6 * (1 - r) + (r * r))/(n * (1 - r));        /* γ2 = (6(1−r)+r²) / (n(1−r)) */
  }      /* Static transformation functions */
  static double p_from_n_r_s(double n, double r, double s);
  static double s_from_p_r_n(double p, double r, double n);
  static double n_from_p_r_s(double p, double r, double s);
  static double r_from_p_n_s(double p, double n, double s);
  double CDF_P(double x);        /* P = ∫_{−∞}^{x} f(x) dx */
    /* Write description of distribution to output stream */
  void writeParameters(ostream &of)
  {
    probabilityDistribution::writeParameters(of);
    of << "␣␣␣␣Successes␣=␣" << n << "␣␣Probability␣␣=␣" << r << "\n";
  }
};
```

35. The following transformation functions are **static** methods of the **negativeBinomialDistribution** class. They permit calculation of properties of the negative binomial distribution without reference to a particular **negativeBinomialDistribution** object.

The functions operate on the following quantities:

`p`	P	Cumulative probability distribution
`s`	s	Number of successes
`r`	r	Probability of success in each trial
`n`	n	Number of trials

⟨ Probability distributions 11 ⟩ +≡

```
double negativeBinomialDistribution::p_from_n_r_s(double n, double r, double s)
{
  double p, q, ri, bound;
  int which = 1, status;
  ri = 1 - r;
  cdfnbn(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("negativeBinomialDistribution::p_from_n_r_s: Result out of b\
        ounds"));
  }
  return p;
}
double negativeBinomialDistribution::s_from_p_r_n(double p, double r, double n)
{
  double q, ri, s, bound;
  int which = 2, status;
  q = 1 - p;
  ri = 1 - r;
  cdfnbn(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("negativeBinomialDistribution::s_from_p_r_n: Result out of b\
        ounds"));
  }
  return s;
}
double negativeBinomialDistribution::n_from_p_r_s(double p, double r, double s)
{
  double q, n, ri, bound;
  int which = 3, status;
  q = 1 - p;
  ri = 1 - r;
  cdfnbn(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
  if (status ≠ 0) {
    throw (out_of_range("negativeBinomialDistribution::n_from_p_r_s: Result out of b\
        ounds"));
  }
  return n;
}
double negativeBinomialDistribution::r_from_p_n_s(double p, double n, double s)
{
  double q, r, ri, bound;
```

```
    int which = 4, status;
    q = 1 - p;
    cdfnbn(&which, &p, &q, &s, &n, &r, &ri, &status, &bound);
    if (status ≠ 0) {
      throw (out_of_range("negativeBinomialDistribution::r_from_p_n_s: Result out of b\
          ounds"));
    }
    return r;
  }
```

36. The *cumulative distribution function* (CDF) P value is the probability of n successes in x trials.

```
⟨ Probability distributions 11 ⟩ +≡
  double negativeBinomialDistribution::CDF_P(double x)
  {
    return p_from_n_r_s(n, r, x);
  }
```

37. Descriptive statistics.

Descriptive statistics refers to empirical tests applied to data sets, measuring quantities such as their mean value, variance, and shape. We implement this via the template class *dataTable*, which is a generalisation of the STL **vector** template with additional methods which compute statistical metrics. The elements in the *dataTable* may be any type which can be converted to a **double**. The elements of a *dataTable* with n items are referred to as $x_1, \ldots x_n$.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ class dataTable : public vector⟨T⟩ {
  public:
    double mean(void);      /* Arithmetic mean value (μ) */
    double geometricMean(void);      /* Geometric mean */
    double harmonicMean(void);      /* Harmonic mean */
    double median(void);      /* Median (central value) */
    double RMS(void);      /* Root mean square */
    T mode(void);      /* Mode (most frequent value) */
    double percentile(double k);      /* Percentile (0 ≤ k ≤ 1) */
    double quartile(int q)      /* Quartiles (implemented with percentile) */
    {
      assert(q ≥ 1 ∧ q ≤ 3);
      return percentile(0.25 * q);
    }
    double variance(void);      /* Variance (σ²) */
    double varianceMLE(void)      /* Variance Maximum Likelihood Estimate */
    {
      return (variance( ) * (size( ) − 1))/size( );
    }
    double stdev(void)      /* Standard deviation (σ) */
    {
      return sqrt(variance( ));
    }
    double stdevMLE(void)      /* Standard deviation Maximum Likelihood Estimate */
    {
      return sqrt(varianceMLE( ));
    }
    double centralMoment(int k);      /* kth central moment */
    double skewness(void);      /* Skewness */
    double kurtosis(void);      /* Kurtosis */
  };
```

38. Arithmetic mean.
The arithmetic **mean** is the average value of the elements in the **dataTable**

$$\frac{1}{n}\sum_{i=1}^{n} x_i.$$

The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::mean(void)
  {
    typename dataTable⟨T⟩::iterator p;
    double m = 0;
    for (p = begin(); p ≠ end(); p++) {
      m += *p;
    }
    return m/size();
  }
```

39. Geometric mean.
The **geometric mean** of a set of n values x_i is

$$\prod_{i=1}^{n} x_i^{\frac{1}{n}}.$$

The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::geometricMean(void)
  {
    typename dataTable⟨T⟩::iterator p;
    double g = 1, ni = 1.0/size();
    for (p = begin(); p ≠ end(); p++) {
      g *= pow(*p, ni);
    }
    return g;
  }
```

40. Harmonic mean.
The **harmonic mean** of a set of n values x_i is

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}.$$

The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::harmonicMean(void)
  {
    typename dataTable⟨T⟩::iterator p;
    double d = 0;
    for (p = begin(); p ≠ end(); p++) {
      d += 1.0/(*p);
    }
    return size()/d;
  }
```

41. Root mean square.
The **root mean square** of a set of n values x_i is

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::RMS(void)
  {
    typename dataTable⟨T⟩::iterator p;
    double sum = 0;
    for (p = begin(); p ≠ end(); p++) {
      sum += (*p) * (*p);
    }
    return sqrt(sum/size());
  }
```

42. Median.

The **median** or central value of a distribution is the value for which half the elements are smaller and half are greater. For a distribution with an even number of elements, the median is defined as the arithmetic mean of the two elements in the middle of the sorted distribution. We implement the median by using the *sort* algorithm to create an ordered copy of the distribution, then extract the central value. The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::median(void)
  {
    dataTable⟨T⟩ v(*this);
    sort(v.begin( ), v.end( ));
    if (v.size( ) & 1) {
      return (double) v[(v.size( ) + 1)/2];
    }
    else {
      return (double)((v[(v.size( )/2) − 1] + v[v.size( )/2])/2.0);
    }
  }
```

43. Mode.

The **mode** is the value which occurs most frequently in the collection of data. This really only makes sense when the **dataTable** is instantiated with an integral data type, but we blither away regardless of the data type; the user may be employing **double**s to represent integers too large for the widest integral type. The type returned is the same as the the elements of the **dataTable**.

We compute the mode by sorting the data table in ascending order, then scanning it linearly for repeated values, keeping track of the item with the most repeats. If two or more values share the largest number of occurrences, the largest value is returned as the mode.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩T dataTable⟨T⟩::mode(void)
  {
    dataTable⟨T⟩ v(*this);
    T cmode, bmode = 0;
    int n = 0, most = 0;
    typename dataTable⟨T⟩::iterator p;
    sort(v.begin( ), v.end( ));
    p = v.begin( );
    cmode = *p++;
    while (p ≠ v.end( )) {
      T cval = *p++;
      if (cval ≡ cmode) {
        n++;
      }
      else {
        if (n > most) {
          most = n;
          bmode = cmode;
        }
        n = 1;
        cmode = cval;
      }
    }
    if (n > most) {      /* Mode may be last item in table */
      most = n;
      bmode = cmode;
    }
    return bmode;
  }
```

44. Percentile.

The kth **percentile** is the value such that $100k\%$ of the data are less than or equal to that value. The result returned is always a **double**. If the result of multiplying the size of the **dataTable** by k is an integer, that item is returned. Otherwise, the mean of the two adjacent items is returned. Note that *percentile*(0.5) is a synonym for *median*( ).

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::percentile(double k)
  {
    dataTable⟨T⟩ v(*this);
    double index, result;
    int i;
    assert(k ≥ 0 ∧ k ≤ 1);
    sort(v.begin( ), v.end( ));
    index = v.size( ) * k;
    if (index ≠ floor(index)) {
      i = ((int) index);
      result = v[i];
    }
    else {
      i = ((int) index) − 1;
      result = (v[i] + v[i + 1])/2.0;
    }
    return result;
  }
```

45. Variance.

The **variance** (σ^2) of a set of n values x_i is

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\overline{x})^2$$

where $\overline{x}$ is the *mean*( ) value of the **dataTable**. The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::variance(void)
  {
    typename dataTable⟨T⟩::iterator p;
    double mu = mean( ), sum = 0;
    for (p = begin( ); p ≠ end( ); p++) {
      double t = ((*p) − mu);
      sum += t * t;
    }
    return sum/(size( ) − 1);
  }
```

46. Central moments.
The kth **central moment** of a set of n values x_i is

$$\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{x})^k$$

where $\overline{x}$ is the *mean*( ) value of the **dataTable**. The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::centralMoment(int k)
  {
    typename dataTable⟨T⟩::iterator p;
    double mu = mean( ), sum = 0;
    for (p = begin( ); p ≠ end( ); p++) {
      sum += pow((*p) − mu, (double) k);
    }
    return sum/size( );
  }
```

47. Skewness.
The **skewness** of a distribution is a measure of its asymmetry with respect to the mean. It is defined as the third central moment of the distribution divided by the cube of the maximum likelihood estimate of the standard deviation. The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::skewness(void)
  {
    double sigma = stdevMLE( );
    return centralMoment(3)/(sigma * sigma * sigma);
  }
```

48. Kurtosis.
Kurtosis is a measure of how sharply peaked the distribution is. It is defined as the fourth central moment of the distribution divided by the square of the maximum likelihood estimate of the variance. The result is always a **double** regardless of the data type of the **dataTable**.

```
⟨ Class definitions 6 ⟩ +≡
  template⟨class T⟩ double dataTable⟨T⟩::kurtosis(void)
  {
    double v = varianceMLE( );
    return centralMoment(4)/(v * v);
  }
```

49. Test program.

```
⟨ statlib_test.cc 1 ⟩ +≡
  ⟨ Test program include files 53 ⟩;
  ⟨ Show how to call test program 51 ⟩;
  int main(int argc, char *argv[])
  {
    extern char *optarg;      /* Imported from getopt */      /* extern int optind; */
    int opt;
    ⟨ Process command-line options 50 ⟩;
    ⟨ Test descriptive statistics template 52 ⟩;
#if 1      /* Statistical library tests */
    {
      normalDistribution nd(100, 7.07);
      cout << "Normal dist: mu = " << nd.get_mu() << " sigma = " << nd.get_sigma() <<
          " variance = " << nd.variance() << "\n";
      cout << "  P = " << nd.CDF_P(110) << " Q = " << nd.CDF_Q(110) << "\n";
      nd.writeParameters(cout);
      nd.approximateBinomial(200);
      cout << "Normal dist: mu = " << nd.get_mu() << " sigma = " << nd.get_sigma() <<
          " variance = " << nd.variance() << "\n";
      cout << "  P = " << nd.CDF_P(110) << " Q = " << nd.CDF_Q(110) << "  z = " << nd.z_score(110) <<
          "\n";
    }
    {
      poissonDistribution pd(10);
      pd.writeParameters(cout);
      cout << "Poisson dist: lambda = " << pd.get_lambda() << "\n";
      cout << "  P = " << pd.CDF_P(5) << " Q = " << pd.CDF_Q(5) << "\n";
    }
    {
      chiSquareDistribution xd(32);
      xd.writeParameters(cout);
      cout << "Chi-square dist: k = " << xd.get_k() << "\n";
      cout << "  P = " << xd.CDF_P(36) << " Q = " << xd.CDF_Q(36) << "\n";
    }
    {
      gammaDistribution gd(7.5, 3.75);
      gd.writeParameters(cout);
      cout << "Gamma dist: alpha = " << gd.get_alpha() << "  lambda = " << gd.get_lambda() << "\n";
      cout << "  P = " << gd.CDF_P(2.2) << " Q = " << gd.CDF_Q(2.2) << "\n";
    }
    {
      betaDistribution bd(3, 4);
      bd.writeParameters(cout);
      cout << "Beta dist: a = " << bd.get_a() << "  b = " << bd.get_b() << "\n";
      cout << "  P = " << bd.CDF_P(0.4) << " Q = " << bd.CDF_Q(0.4) << "\n";
    }
    {
      tDistribution xd(32);
      xd.writeParameters(cout);
```

```
    cout ≪ "t␣dist:␣k␣=␣" ≪ xd.get_k( ) ≪ "\n";
    cout ≪ "␣␣P␣=␣" ≪ xd.CDF_P(0.8) ≪ "␣Q␣=␣" ≪ xd.CDF_Q(0.8) ≪ "\n";
  }
  {
    FDistribution fd(12, 16);
    fd.writeParameters(cout);
    cout ≪ "F␣dist:␣u␣=␣" ≪ fd.get_u( ) ≪ "␣␣v␣=␣" ≪ fd.get_v( ) ≪ "\n";
    cout ≪ "␣␣P␣=␣" ≪ fd.CDF_P(0.8) ≪ "␣Q␣=␣" ≪ fd.CDF_Q(0.8) ≪ "\n";
  }
  {
    binomialDistribution bd(200, 0.5);
    cout ≪ "Binomial␣dist:␣n␣=␣" ≪ bd.get_n( ) ≪ "␣r␣=␣" ≪ bd.get_r( ) ≪ "␣variance␣=␣" ≪
        bd.variance( ) ≪ "\n";
    cout ≪ "␣␣P␣=␣" ≪ bd.CDF_P(110) ≪ "␣Q␣=␣" ≪ bd.CDF_Q(110) ≪ "␣␣z␣=␣" ≪ bd.z_score(110) ≪
        "\n";
    bd.writeParameters(cout);
  }
  {
    negativeBinomialDistribution bd(10, 0.5);
    cout ≪ "Negative␣binomial␣dist:␣n␣=␣" ≪ bd.get_n( ) ≪ "␣r␣=␣" ≪ bd.get_r( ) ≪
        "␣variance␣=␣" ≪ bd.variance( ) ≪ "\n";
    cout ≪ "␣␣P␣=␣" ≪ bd.CDF_P(15) ≪ "␣Q␣=␣" ≪ bd.CDF_Q(15) ≪ "\n";
    bd.writeParameters(cout);
  }
#endif
  return 0;
}
```

50. We use *getopt* to process command line options. This permits aggregation of options without arguments and both `-d`*arg* and `-d` *arg* syntax.

```
⟨ Process command-line options 50 ⟩ ≡
  while ((opt = getopt(argc, argv, "nu-:")) ≠ −1) {
    switch (opt) {
    case 'u':     /* -u Print how-to-call information */
      case '?': usage( );
      return 0;
    case '-':     /* -- Extended options */
      switch (optarg[0]) {
      case 'c':     /* --copyright */
        cout ≪ "This␣program␣is␣in␣the␣public␣domain.\n";
        return 0;
      case 'h':     /* --help */
        usage( );
        return 0;
      case 'v':     /* --version */
        cout ≪ PRODUCT ≪ "␣" ≪ VERSION ≪ "\n";
        cout ≪ "Last␣revised:␣" ≪ REVDATE ≪ "\n";
        cout ≪ "The␣latest␣version␣is␣always␣available\n";
        cout ≪ "at␣http://www.fourmilab.ch/eggtools/eggshell\n";
        return 0;
      }
    }
  }
```

This code is used in section 49.

51. Procedure *usage* prints how-to-call information.

```
⟨ Show how to call test program 51 ⟩ ≡
  static void usage(void)
  {
    cout ≪ PRODUCT ≪ "␣␣--␣␣Analyse␣eggsummary␣files.␣␣Call:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ PRODUCT ≪ "␣[options]␣[infile]␣[outfile]\n";
    cout ≪ "\n";
    cout ≪ "Options:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣␣␣Print␣copyright␣information\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣-u,␣--help␣␣␣␣␣␣␣␣Print␣this␣message\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣␣␣␣Print␣version␣number\n";
    cout ≪ "\n";
    cout ≪ "by␣John␣Walker\n";
    cout ≪ "http://www.fourmilab.ch/\n";
  }
```

This code is used in section 49.

52. The following test the descriptive statistics facilities implemented by the **dataTable** template class.

```
⟨ Test descriptive statistics template 52 ⟩ ≡
  { int i;
  dataTable⟨int⟩ zot;
  dataTable⟨int⟩::iterator z;
#define PR
  for (z = zot.begin(); z ≠ zot.end(); z++) {
    cout ≪ *z ≪ "␣";
  }
  cout ≪ "\n";
  for (i = 0; i < 10; i++) {
    zot.push_back(rand() & #3F);
  }
  PR;
  sort(zot.begin(), zot.end());
  PR;
  reverse(zot.begin(), zot.end());
  PR;
  cout ≪ "Mean␣=␣" ≪ zot.mean() ≪ "\n";
  cout ≪ "Geometric␣mean␣=␣" ≪ zot.geometricMean() ≪ "\n";
  cout ≪ "Harmonic␣mean␣=␣" ≪ zot.harmonicMean() ≪ "\n";
  cout ≪ "RMS␣=␣" ≪ zot.RMS() ≪ "\n";
  cout ≪ "Median␣=␣" ≪ zot.median() ≪ "\n";
  cout ≪ "Mode␣=␣" ≪ zot.mode() ≪ "\n";
  cout ≪ "Percentile(0.5)␣=␣" ≪ zot.percentile(0.5) ≪ "\n";
  cout ≪ "Quartile(1)␣=␣" ≪ zot.quartile(1) ≪ "\n";
  cout ≪ "Quartile(3)␣=␣" ≪ zot.quartile(3) ≪ "\n";
  cout ≪ "Variance␣=␣" ≪ zot.variance() ≪ "\n";
  cout ≪ "Standard␣deviation␣=␣" ≪ zot.stdev() ≪ "\n";
  cout ≪ "CentralMoment(3)␣=␣" ≪ zot.centralMoment(3) ≪ "\n";
  cout ≪ "Skewness␣=␣" ≪ zot.skewness() ≪ "\n";
  cout ≪ "Kurtosis␣=␣" ≪ zot.kurtosis() ≪ "\n"; }
```

This code is used in section 49.

53. We need the following definitions to compile the test program.

⟨ Test program include files 53 ⟩ ≡

```
#include "config.h"      /* Our configuration */      /* C++ include files */
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
  using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef HAVE_GETOPT
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#else
#include "getopt.h"      /* No system getopt—use our own */
#endif
#include "statlib.h"      /* Class definitions for this package */
```

This code is used in section 49.

54. Index. The following is a cross-reference table for `statlib`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

⟨ Application include files 4 ⟩ Used in section 2.
⟨ Approximate binomial distribution 9 ⟩ Used in section 7.
⟨ Class definitions 6, 7, 13, 16, 19, 22, 25, 28, 31, 34, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48 ⟩ Used in section 3.
⟨ Class implementations 5 ⟩ Used in section 2.
⟨ Compute z score 10 ⟩ Used in sections 7 and 31.
⟨ Probability distributions 11, 12, 14, 15, 17, 18, 20, 21, 23, 24, 26, 27, 29, 30, 32, 33, 35, 36 ⟩ Used in section 5.
⟨ Process command-line options 50 ⟩ Used in section 49.
⟨ Set standard deviation from variance 8 ⟩ Used in section 7.
⟨ Show how to call test program 51 ⟩ Used in section 49.
⟨ Test descriptive statistics template 52 ⟩ Used in section 49.
⟨ Test program include files 53 ⟩ Used in section 49.
⟨ `statlib.h` 3 ⟩
⟨ `statlib_test.cc` 1, 49 ⟩

STATLIB