Sanjay Formighieri

Uma biblioteca de funções estatísticas para o apoio no desenvolvimento de aplicações com aderência de dados.

Florianópolis - SC

04 de Dezembro de 2007

Sanjay Formighieri

Uma biblioteca de funções estatísticas para o apoio no desenvolvimento de aplicações com aderência de dados.

Trabalho de conclusão do curso de Ciência da Computação, Departamento de Informática e Estatística, Centro Tecnológico, Universidade Federal de Santa Catarina.

Orientador:
Paulo José de Freitas Filho

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis - SC

04 de Dezembro de 2007

Resumo

Pensando em interagir conhecimentos da área de estatística com aplicações práticas de conhecimentos computacionais, o trabalho proposto envolve o estudo, agrupamento e implementação de algoritmos para cálculo de fórmulas matemáticas complexas necessárias para a obtenção de parâmetros das funções de distribuição de probabilidade, — através da aplicação de estimadores de Máxima Verossimilhança — o cálculo das funções de distribuição acumulada e de densidade de probabilidade, e geração de números aleatórios. Além de algoritmos para o tratamento dos dados, como por exemplo a montagem de tabelas de freqüências.

Este estudo teve como finalidade a construção de uma biblioteca em C++ flexível e portável que fornece ferramentas necessárias para teste de aderência em conjuntos de dados. Esta biblioteca está disponibilizada sob licença LGPL , no intuito de facilitar o desenvolvimento de novas funcionalidades e de melhorias por quaisquer desenvolvedores de sistemas.

Sumário

Lista de Figuras

1 Introdução p. 7

1.1 Introdução geral . . . . . p. 7

1.2 Importância e justificativa . . . . . p. 8

1.3 Objetivo geral . . . . . p. 8

1.4 Objetivos específicos . . . . . p. 8

1.5 Estrutura do trabalho . . . . . p. 9

2 Conceitos fundamentais p. 10

2.1 Testes de aderência . . . . . p. 10

2.1.1 Teste Qui-Quadrado de aderência . . . . . p. 10

2.2 Distribuições de probabilidade . . . . . p. 12

2.2.1 Uniforme . . . . . p. 13

2.2.2 Triangular . . . . . p. 13

2.2.3 Exponencial . . . . . p. 14

2.2.4 Normal . . . . . p. 15

2.2.5 Lognormal . . . . . p. 15

2.2.6 Gamma . . . . . p. 15

2.2.7 Weibull . . . . . p. 16

2.2.8 Beta . . . . . p. 17

3 Desenvolvimento p. 19

3.1 Introdução . . . . p. 19
3.2 Metodologia de trabalho . . . . p. 20
3.3 Licença . . . . p. 21
3.4 Funcionamento . . . . p. 22
3.4.1 Bibliotecas estáticas . . . . p. 22
3.4.2 Bibliotecas de vínculo dinâmico . . . . p. 23
3.5 Aspectos de implementação . . . . p. 24
3.5.1 Definições globais . . . . p. 24
3.5.2 Distribuições de probabilidade . . . . p. 28
3.5.3 Tabela de freqüência . . . . p. 30
3.5.4 Interface da biblioteca . . . . p. 36

4 Conclusão p. 40

4.1 Trabalhos futuros . . . . p. 40

Referências Bibliográficas p. 42

Apêndice A – Diagramas de classes p. 43

A.1 Diagrama de classes de distribuições de probabilidade . . . . p. 43
A.2 Diagrama de classes da tabela de freqüências . . . . p. 44

Anexo A – GNU Lesser General Public License p. 45

Anexo B – Artigo p. 49

Lista de Figuras

1 Ilustração do teste de significância de qui-quadrado . . . . p. 12

2 Ilustração do método de Czuber . . . . p. 14

3 Distribuição Weibull, parâmetros de forma . . . . p. 17

4 Ilustração do uso de bibliotecas . . . . p. 22

5 Aplicação usando bibliotecas estáticas . . . . p. 23

6 Aplicação usando biblioteca dinâmica . . . . p. 24

7 Construção da tabela de freqüência 1 . . . . p. 31

8 Construção da tabela de freqüência 2 . . . . p. 32

9 Exemplo de histograma pouco representativo . . . . p. 34

10 Função distribuição de probabilidade mal escolhida . . . . p. 36

Lista de Listagens

3.1 Arquivo de definições globais . . . . p. 25

3.2 Definição de subrotinas utilitárias da biblioteca . . . . p. 26

3.3 Implementação da função que check_values . . . . p. 28

3.4 Implementação da função expected_freq . . . . p. 28

3.5 Exemplo de enumeração dos parâmetros . . . . p. 29

3.6 Implementação da função `init_members` . . . . p. 32

3.7 Implementação da função `divide_intervals` . . . . p. 33

3.8 Implementação da função `merge_intervals` . . . . p. 35

3.9 Interface da biblioteca . . . . p. 36

3.10 Implementação da função `get_chisquare_test` . . . . p. 38

3.11 Implementação da função `get_distribution` . . . . p. 38

3.12 Implementação da função `get_freq_table` . . . . p. 39

1 *Introdução*

1.1 Introdução geral

A Simulação computacional faz parte da área que compreende as soluções por métodos numéricos de sistemas complexos de áreas como astrofísica, ecossistemas, construção civil, indústria petroquímica, entre outras em que sejam necessárias a realização de previsões ou onde os problemas a serem resolvidos são impossíveis ou inviáveis de serem reproduzidos para fins de estudo. Como alguns exemplos podemos citar o dimensionamento de tubulações de poços de petróleo; estudo do impacto de desmatamentos ambientais; simulação de ação de proteínas e de seqüências de código genético.

Para que simulações de sistemas reais possam ser realizadas, modelos físicos ou lógicos desses sistemas devem ser construídos. Os modelos físicos normalmente são uma representação do sistema em uma escala diferente. Já nos modelos lógicos suas características são expressadas por equações matemáticas.

De acordo com Law e Kelton (1991, p. 325), quase todos os sistemas reais contém uma ou mais fontes de comportamento aleatório. Durante a simulação desses tipos de sistemas, é extremamente importante garantir a aleatoriedade dessas variáveis tornando o modelo de simulação o mais fiel possível ao sistema real (precaução necessária para certificar que a solução obtida pode ser aplicada com certa previsibilidade).

Para isso, amostras dessas fontes aleatórias devem ser coletadas e analisadas para que se possa obter um padrão que exemplifique os dados dessas amostras. É nesse contexto que esse trabalho será desenvolvido. O mesmo oferecerá uma forma prática de criar funções probabilísticas que representarão o padrão de um conjunto de dados.

"Na estatística, os métodos para reconhecimento de modelos de probabilidade são conhecidos como teste de aderência." (TENÓRIO, 2005). Neste caso, os testes de aderência servirão para verificar se uma dada função densidade de probabilidade representa um respectivo conjunto de dados. Alguns exemplos destes testes são: Teste Qui-quadrado, Teste de Kolmogorov-

Smirnov, Teste de Lilliefors, Teste de Anderson-Darling, entre outros. (BARBETTA; REIS; BORNIA, 2004), (LAW; KELTON, 1991), (NETO, 1977).

1.2 Importância e justificativa

Existem algumas ferramentas já disponíveis para efetuar testes de aderência sobre um conjunto de dados, entretanto, atualmente se tratam de ferramentas dispendiosas e não flexíveis para aplicação em sistemas de simulação probabilística. Exemplos como InputAnalyser da Rockwell e o Statistica da StatSoft, realizam testes de aderência, geram gráficos e relatórios mas não dispõem de certas funcionalidades como, por exemplo, retro-alimentar um sistemas de simulação com informações necessárias para a randomicidade de suas variáveis.

Quando de frente com esse tipo de problema, os programadores desses sistemas decidem por reimplementar as funções estatísticas e, mesmo após isso, acabam engessando-as ao produto desenvolvido dificultando, também, a futura reutilização de código por outros desenvolvedores. Por se tratar de um problema bastante comum entre os projetistas de sistemas, a solução seria agrupar essas funcionalidades em um meio que facilitasse a reutilização do código e a correção de *bugs*, por exemplo, em uma biblioteca de funções.

1.3 Objetivo geral

O objetivo geral desse trabalho é o estudo e a implementação de uma biblioteca portável em C++ que agrupe funções estatísticas para que aplicações finais (software) possam tratar dados amostrais e realizar teste de aderência sobre os mesmos.

1.4 Objetivos específicos

De uma forma mais detalhada, os objetivos desse trabalho incluem:

- estudar aspectos de portabilidade de código entre sistemas operacionais Linux e Windows;
- estudar sobre diferentes aplicações das funções de distribuição estatística;
- implementar, testar e otimizar algoritmos necessários.

- construir uma biblioteca com interface simples para permitir e incentivar o reúso de código por parte dos desenvolvedores de sistemas.

1.5 Estrutura do trabalho

Este trabalho apresentará, de forma geral, conceitos necessários aos testes de aderência no capítulo 2. Neste capítulo também é explicado o teste qui-quadrado de aderência e posteriormente, há uma breve descrição das funções distribuição de probabilidade implementadas neste trabalho. Essas descrições englobam as aplicações das funções, e fórmulas matemáticas utilizadas em suas implementações.

O capítulo 3 traz aspectos da implementação da biblioteca. No início no capítulo são comentados os motivos e os desafios à implementação incial do código da biblioteca. Após o assunto tratado é a licensa do código e suas vantagens. A seção 3.4 trata sobre o funcionamento e os diferentes tipo de bibliotecas. A seguir, com o uso de conceitos de engenharia de software, são tratados aspectos de implementação e também sobre o funcionamento de partes específicas da biblioteca.

No capítulo 4 e último visa comentar sobre a aprendizagem e os objetivos atingidos. Após essas considerações, há uma breve discussão sobre possíveis trabalhos a serem realizados.

2 Conceitos fundamentais

Neste capítulo é apresentado um resumo prático da teoria utilizada para o desenvolvimento deste trabalho. É importante levar em conta que os cálculos descritos são para variáveis e distribuições contínuas.

2.1 Testes de aderência

Testes de aderência são testes não-paramétricos que visam observar se um conjunto de dados é uma amostra independente de uma distribuição de probabilidade com função distribuição F(LAW; KELTON, 1991), que será descrita mais adiante, ou de proporções definidas no problema. (BARBETTA; REIS; BORNIA, 2004). São testes que podem ser usados para testar a seguinte hipótese nula:

H_0: Os dados da amostra são variáveis aleatórias de uma função distribuição F.

2.1.1 Teste Qui-Quadrado de aderência

O teste qui-quadrado é o mais famoso pois, além de ser o mais antigo, é o que se aplica a mais tipos de distribuições discretas ou contínuas.

Para calcular sua estatística de teste é necessário que todos os dados sejam classificados em k intervalos adjacentes. Depois anotamos a freqüência observada O_j, ou seja, a quantidade desses dados que caíram em cada intervalo sendo $j = 1, 2, \ldots, k$. Então estima-se a proporção p_j de dados que seriam classificados no j-ésimo intervalo se estivéssemos gerando valores de acordo com distribuição aderida. Neste caso,

$$p_j = \int_{a_{j-1}}^{a_j} f(x)\, dx \tag{2.1}$$

onde f é a função densidade da distribuição aderida. Assim calculamos a freqüência esperada $E_j = np_j$, onde n é a quantidade de valores da amostra. Por fim a estatística do teste é uma espé-

cie de medida de distância entre as freqüências observadas (amostra) e as freqüências esperadas (distribuição de probabilidades teórica) de cada classe. (TENÓRIO, 2005). Sua expressão é:

$$\chi^2 = \Sigma_{j=1}^{k} \frac{(O_j - E_j)^2}{E_j}$$

Se as freqüências observadas são próximas das freqüências esperadas, naturalmente as diferenças $(O_j - E_j)$ serão pequenas, e conseqüentemente o valor de χ^2 também será pequeno. Se algumas ou muitas das diferenças são grandes, o valor de χ^2 também será grande. Quanto maior o valor de χ^2, maior é a probabilidade de que o grupo de freqüências observadas difira do grupo de freqüências esperadas.(SIEGEL, 1956)

Se H_0 for verdade, os possíveis valores de χ^2 seguem a distribuição qui-quadrado com $gl = k - c$ ($gl =$ graus de liberdade), $c =$ número de parâmetros estimados da distribuição escolhida $+1$. O tamanho de gl reflete o número de observações que podem variar após feitas certas restrições sobre os dados. Como k é o número de intervalos da tabela, tais restrições são dependentes da forma com que os dados são organizados. Portanto só faz sentido referir-se a um valor de χ^2 como grande ou pequeno em função do parâmetro gl.

Para decidir sobre as hipóteses, Barbetta, Reis e Bornia (2004) orientam a calcular um valor crítico χ_c^2, em função de um nível de significância α adotado, formando uma região de rejeição onde se:

$$\chi^2 < \chi^2_{k-c,\alpha} \Rightarrow \text{aceita } H_0$$
$$\chi^2 \geq \chi^2_{k-c,\alpha} \Rightarrow \text{rejeita } H_0$$

portanto o valor de $\chi^2_{k-c,\alpha} \equiv \chi_c^2$ é o ponto onde a probabilidade de ocorrerem valores maiores que ele é α. Este ponto pode ser encontrado em tabelas de valores críticos de qui-quadrado.

Outra forma de proceder com a decisão é, através da estatística χ^2 e do número de graus de liberdade gl, calcular a probabilidade de rejeitar H_0 também conhecido como valor-p. Esse valor pode ser obtido através da função $gammaq(gl/2, \chi^2/2)$ que será comentada mais adiante. Para um nível de significância α, rejeita-se H_0 se $valor\ p < \alpha$. Ou seja, a região de rejeição consiste de todos os valores de χ^2 que são tão grandes que a probabilidade associada à sua ocorrência, sob H_0, não supera α. A figura 1 ilustra esse processo.

Por exemplo para $\alpha = 0,05$ e $gl = 5$, se a estatística $\chi^2 = 13,39$, o valor-p calculado será $0,02$. Como $valor\ p < \alpha$ então rejeita-se a hipótese nula. De outra forma, se procurarmos pelo valor crítico χ_c^2, encontramos $\chi^2_{5,0.05} = 11,07$. Percebe-se também que H_0 deve ser rejeitada pois $\chi^2 \geq \chi^2_{5,0.05}$. São formas complementares para a tomada da mesma decisão.

2.2 Distribuições de probabilidade

De acordo com Tenório (2005), uma forma alternativa de representação da distribuição de probabilidades de uma variável aleatória é através da sua função de distribuição acumulada que é definida por:

$$F(x) = \int_{-\infty}^{x} f(x)\,dx \tag{2.2}$$

Ou seja, a função de distribuição acumulada $F(x)$ descreve a probabilidade de ocorrer um valor menor ou igual a x. Assim, através de propriedades das integrais, a equação 2.1 pode ser descrita como:

$$p_j = F(j_{sup}) - F(j_{inf})$$

portanto, a freqüência esperada no j-ésimo intervalo é:

$$\begin{aligned} E_i &= n * p_j \\ &= n[F(j_{sup}) - F(j_{inf})] \end{aligned} \tag{2.3}$$

sendo j_{sup} o limite superior do intervalo, j_{inf} o limite inferior, e n o tamanho da amostra.

O cálculo desta função distribuição, por se tratar de uma integral, é impossível de ser obtido trivialmente em computadores. Uma solução é utilizar-se de métodos iterativos para integração numérica — como o método adaptativo de Simpson — ou encontrar formas fechadas (abordagem analítica) do cálculo dessas funções. Algo semelhante ocorre nos estimadores de máxima verossimilhança (ou MLE do inglês *maximum likelihood estimation*) para a obtenção dos parâ-

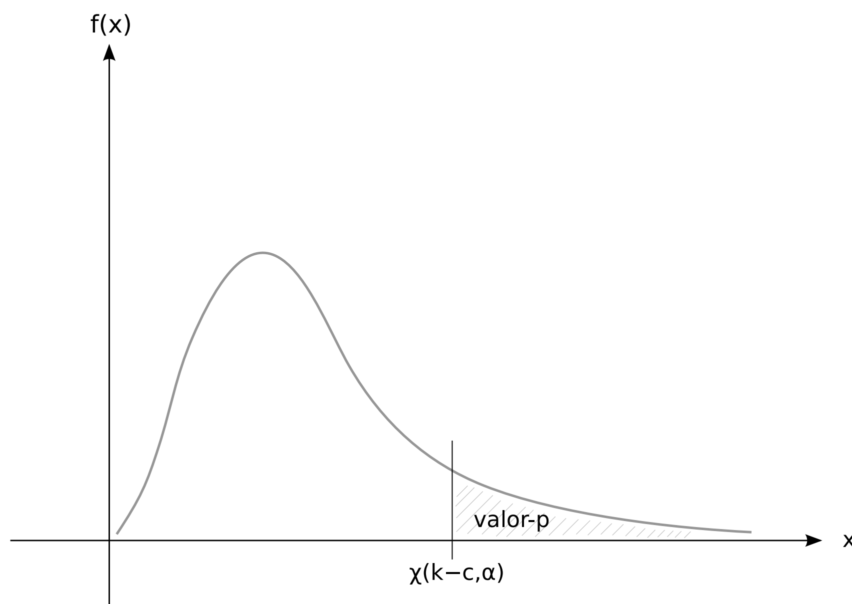


Figura 1: Ilustração do teste de significância de qui-quadrado

metros das distribuições.

Métodos iterativos requerem mais recursos computacionais reduzindo assim a performance, e nem sempre convergem para um resultado. Graças ao suporte de professores da área de estatística, foi possível o uso de formas fechadas tanto para o cálculo das funções acumuladas tanto para a estimação dos parâmetros por MLE. A seguir, será apresentado algumas distribuições contínuas implementadas neste trabalho e também as fórmulas utilizadas para esses cálculos.

2.2.1 Uniforme

Conhecida também como distribuição retangular, é a mais simples das distribuições e tem probabilidade constante em seu intervalo. O intervalo é definido por dois parâmetros, a e b que são respectivamente, o menor e o maior valor. Normalmente abreviada por $U(a,b)$, a distribuição $U(0,1)$ é a mais famosa por ser essencial na geração de números aleatórios de outras distribuições.

$$\begin{aligned} \text{Intervalo} &= [a,b] \\ F(x) &= \begin{cases} 0 & \text{se } x < a \\ \dfrac{x-a}{b-a} & \text{se } a \leq x \leq b \\ 1 & \text{se } b < x \end{cases} \\ \text{MLE} &\Rightarrow a = min_{1 \leq i \leq n}\ X_i,\ b = max_{1 \leq i \leq n}\ X_i \end{aligned}$$

2.2.2 Triangular

Normalmente denotada por $tria(a,c,b)$ a distribuição triangular é usada como um modelo bastante áspero em ausência de dados. Se for possível estimar de forma subjetiva o intervalo $[a,b]$ e o valor mais provável (moda), c, então já é possível construir um gráfico de uma função de densidade triangular.

$$\begin{aligned} \text{Intervalo} &= [a,b] \\ F(x) &= \begin{cases} 0 & \text{se } x < a \\ \dfrac{(x-a)^2}{(b-a)(c-a)} & \text{se } a \leq x \leq c \\ 1 - \dfrac{(b-x)^2}{(b-a)(b-c)} & \text{se } c < x \leq b \\ 1 & \text{se } b < x \end{cases} \\ \text{MLE} &\Rightarrow a = min_{1 \leq i \leq n}\ X_i,\ b = max_{1 \leq i \leq n}\ X_i \end{aligned}$$

A moda da distribuição triangular pode ser estimada da seguinte forma $Mo = 3 \cdot \overline{X} - (X_{min} + X_{max})$ um método mais preciso é o método proporcional de Czuber:

$$Mo = X + \frac{h(f_{max} - f_{ant})}{2 \cdot f_{max} - (f_{ant} + f_{pos})}$$

sendo

X	Limite inferior da classe modal.
h	Intervalo da classe (amplitude).
f_{max}	Freqüência da classe modal.
f_{ant}	Freqüência da classe anterior.
f_{pos}	Freqüência da classe posterior.

O método de Czuber consiste em estimar a moda de uma distribuição triangular através da proporção das freqüências dos intervalos anterior e posterior ao intervalo modal. A figura 2 mostra o funcionamento do método de forma visual.

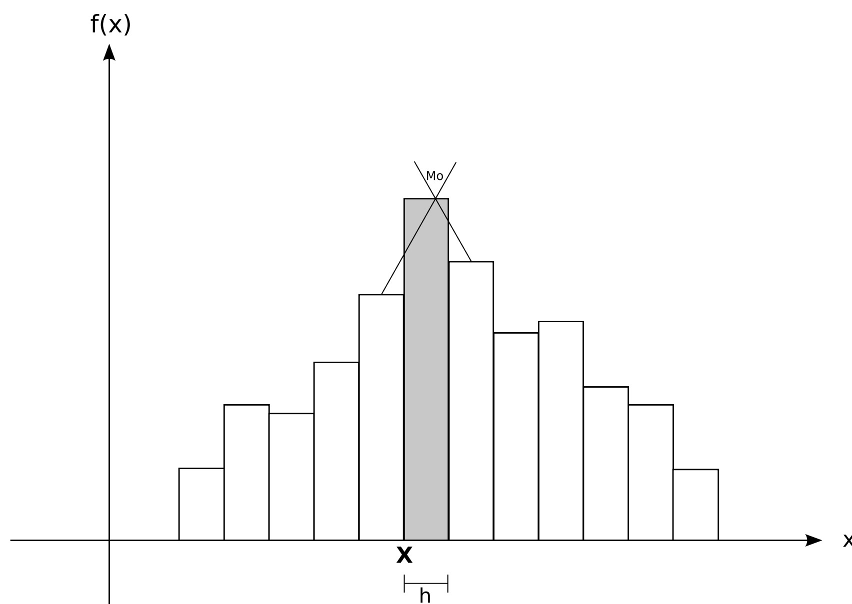


Figura 2: Estimação da moda de uma distribuição triangular através do método de Czuber

2.2.3 Exponencial

A distribuição exponencial, também denotada por $expo(\beta)$, é bastante usada para modelar processos de Poisson. Por exemplo, para expressar tempo entre chegadas de “consumidores” à um sistema, que ocorre a uma taxa constante, ou o número de ligações que chegam à um tronco telefônico. Seu parâmetro, β, é o parâmetro de escala e é freqüentemente referido como sendo λ, que é igual à $\frac{1}{\beta}$; $\beta > 0$.

$$\begin{aligned} \text{Intervalo} &= [0, \infty) \\ F(x) &= \begin{cases} 1 - e^{\frac{-x}{\beta}} & \text{se } x \geq 0 \\ 0 & \text{caso contrário} \end{cases} \\ \text{MLE} &\Rightarrow \beta = \overline{X}(n) \end{aligned}$$

2.2.4 Normal

Bastante conhecida por sua forma de sino, a distribuição normal, denotada por $norm(\mu, \sigma)$, tem grande importância como modelo quantitativo em fenômenos físicos e das ciências naturais e de comportamento devido às suas propriedades no teorema do limite central. Não há forma fechada para o cálculo de sua função distribuição e seus parâmetros, μ e σ, são a média e o desvio padrão da amostra.

$$\begin{aligned} \text{Intervalo} &= (-\infty, \infty) \\ \text{MLE} &\Rightarrow \mu = \overline{X}(n), \quad \sigma = \left[\frac{n-1}{n} S^2(n) \right]^{\frac{1}{2}} \end{aligned}$$

2.2.5 Lognormal

A distribuição Lognormal, denotada por $LogNorm(\mu, \sigma)$, pode ser usada para expressar tempo para realização de alguma tarefa. Outras variáveis que podem ser representadas pela Lognormal incluem o tempo de sobrevivência da bactéria em desinfetantes e o peso e a pressão sanguínea em humanos.

É um caso particular da distribuição normal pois $X \sim LogNorm(\mu, \sigma)$ se e somente se $\ln X \sim N(\mu, \sigma)$. Portanto se uma distribuição tem dados $X_1, X_2, \ldots, X_n$ e é dita ser uma lognormal, os logaritmos dos dados, $\ln X_1, \ln X_2, \ldots, \ln X_n$, podem ser tratados como uma distribuição normal para estimar parâmetros e realizar testes de aderência.

2.2.6 Gamma

A distribuição gamma, $gamma(\alpha, \beta)$, é bastante usada para representar a soma de variáveis aleatórias distribuídas exponencialmente. Quando nesta forma, α representa o número de variáveis da soma e β, a média da distribuição exponencial. Exemplos de uso incluem modelagem de fila, o fluxo de itens em processos de manufatura e distribuição e carga em servidores *web*.

Pode ser usada também para expressar tempo para realização de tarefas como tempo de reparo de máquinas.

Seus parâmetros α e β, respectivamente, definem a forma e a escala da distribuição; $\alpha > 0$ e $\beta > 0$.

$$\begin{aligned}
\text{Intervalo} &= [0,\infty) \\
F(x) &= \begin{cases} \Gamma_p\left(\alpha, \dfrac{x}{\beta}\right) & \text{se } x > 0 \\ 0 & \text{caso contrário} \end{cases} \\
\text{MLE} &\Rightarrow \alpha = \frac{\overline{X}^2}{S^2}, \quad \beta = \frac{S^2}{\overline{X}} \qquad (S^2 \neq 0 \ e \ \overline{X} \neq 0)
\end{aligned}$$

Onde Γ_p é a função gamma incompleta definida como

$$\Gamma_p(a,x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{\alpha-1}\, dt \qquad (a > 0)$$

melhor descrita em Press et al. (1992, p. 216).

Casos particulares da distribuição gamma incluem:

Qui-Quadrado – A distribuição Qui-quadrado é uma distribuição gamma onde o parâmetro de forma é o número de graus de liberdade divido por dois e o parâmetro de escala é igual à dois.

Erlang – A distribuição Erlang é usada para modelar o intervalo total associado à múltiplos eventos de Poisson. O parâmetro de forma representa o número de eventos e o de escala o intervalo médio entre eventos.

2.2.7 Weibull

A distribuição Weibull, $weibull(\alpha,\beta)$, tem como seus parâmetros respectivamente os de forma e escala ($\alpha > 0$ e $\beta > 0$). É uma distribuição bastante versátil pois diferentes valores do parâmetro de forma podem fazer com que a distribuição Weibull tome características de outras distribuições (ver figura 3). Por exemplo, quando $\alpha = 1$, a distribuição $weibull(1,\beta)$ é a mesma que a $expo(\beta)$. (WEISSTEIN, 2007)

Na prática a weibull é usada para dois tipos de fenômenos. O tempo de vida de objetos que é normalmente usado em controle de qualidade. A fábrica provê os parâmetros da Weibull para um produto e o usuário pode calcular a probabilidade que ele falhe após um certo tempo. Outro

uso associado à Weibull é descrever fenômenos naturais como por exemplo, a velocidade do vento.

$$\begin{aligned} \text{Intervalo} &= [0,\infty) \\ F(x) &= \begin{cases} 1-e^{-\left(\frac{x}{\beta}\right)^{\alpha}} & \text{se } x>0 \\ 0 & \text{caso contrário} \end{cases} \end{aligned}$$

O parâmetro β da Weibull é obtido de forma direta através da solução de uma equação. Já o α deve ser resolvido iterativamente através do método de Newton. As fórmulas estão descritas em Law e Kelton (1991, p. 334).

2.2.8 Beta

Por fim, a função distribuição beta, $beta(\alpha,\beta)$, é usada para modelar eventos que se restringem à um intervalo definido por um valor mínimo e outro máximo. Por esta razão, a distribuição Beta é largamente usada em PETR (Técnica de Avaliação e Revisão de Projetos), CPM (Método do Caminho Crítico) e em outros sistemas de controle e planejamento para descrever o tempo para completar tarefas.

Seus dois parâmetros, α e β, são parâmetros de escala.

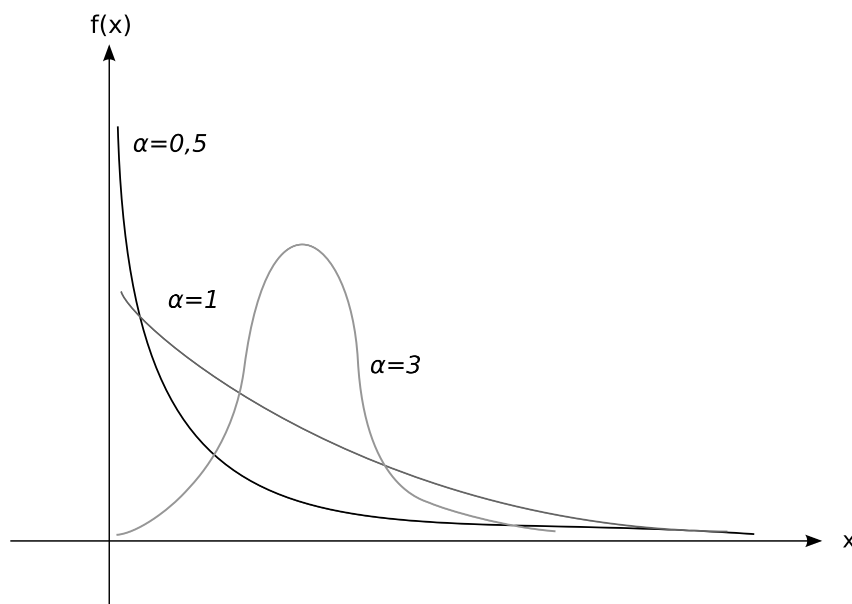


Figura 3: Função densidade de probabilidade da Weibull para $0<\alpha<1$, $\alpha=1$ e $\alpha>1$

$$\begin{aligned}
\text{Intervalo} &= [0,1] \\
F(x) &= \frac{1}{B(\alpha,\beta)} \int_0^x t^{\alpha-1}(1-t)^{\beta-1}\, dt \\
\text{MLE} &\Rightarrow \alpha = \overline{X}\left[\left[\frac{\overline{X}(1-\overline{X})}{S^2}\right]-1\right], \qquad \beta = (\overline{X}-1)\left[\left[\frac{\overline{X}(1-\overline{X})}{S^2}\right]-1\right]
\end{aligned}$$

A função $F(x)$ é conhecida como função beta incompleta ($I_x(a,b)$), outra forma de escreve-la é

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)}$$

onde $B(a,b)$, a função beta, pode ser encontrada através de sua relação com a função gamma, ou através da resolução de uma integral. (PRESS et al., 1992, p. 226)

$$B(z,w) = B(w,z) = \frac{\Gamma(z)\,\Gamma(w)}{\Gamma(z+w)} = \int_0^1 t^{z-1}(1-t)^{w-1}\, dt$$

3 *Desenvolvimento*

3.1 Introdução

Como comentado no capítulo 1, a idéia inicial do estudo foi fornecer um meio de aproveitar melhor variáveis coletadas de ambientes para, assim, permitir uma modelagem matemática mais fiel desses ambientes.

Anteriormente aos motivos que me fizeram decidir pela construção desta biblioteca, essas funcionalidades foram exigidas em um projeto maior que se utilizava de simulação computacional para otimizar um discador de *call center*.

Estamos acostumados em acreditar que *call centers* servem somente para atendimento ao consumidor, mas é bastante comum encontrarmos *call centers* que realizam, também, ligações para fora. Alguns deles, como os de venda de cartão de crédito ou de prospecção, funcionam inteiramente com ligações denominadas saíntes. Nesses casos, para automatizar o processo, existe um elemento chamado discador que em contato com um banco de dados realiza as chamadas e, quando completa, as encaminha para uma atendente. Normalmente esses discadores realizam uma nova ligação no momento em que uma atendente do *call center* fica livre. Entretanto essas ligações nem sempre eram completadas o que forçava ao discador tentar novas ligações deixando as atendentes ociosas.

A proposta de estudo era criar um discador que pudesse prever a taxa de ligações que deveriam ser realizadas para que, quando um atendente do *call center* ficasse livre, uma ligação recém atendida pudesse ser encaminhada à esse atendente diminuindo assim, seu tempo ociosidade.

A lógica desta aplicação consiste em coletar variáveis do ambiente como o tempo da duração de ligação, tempo para atendimento, número de chamadas ocupadas ou com falha em relação às tentativas, entre outros... e dessa forma, através de um modelo, poder simular o ambiente de forma mais próxima à realidade e então prever seu comportamento (como por exemplo, o momento em que um atendente ficaria livre) para assim, estimar quando e quantas

chamadas realizar.

Devido ao ambiente não determinístico, essa simulação se repete constantemente em intervalos de tempo para que novas variáveis continuem sendo coletadas não permitindo que o modelo simulado divirja do ambiente real.

Para que o ambiente de simulação pudesse gerar valores aleatórios às variáveis necessárias, um modelo matemático delas deveria ser criado. Neste momento viu-se necessário a implementação de uma ferramenta que dado um conjunto de valores, retornasse uma função geradora de números aleatórios da distribuição estatística que melhor o representasse.

Tecnicamente e resumidamente, o processo consiste em supor distribuições de probabilidade; estimar seus parâmetros através dos estimadores de máxima verossimilhança; realizar o teste de aderência e posteriormente escolher pela distribuição que melhor aderiu aos dados.

Percebe-se que este processo é bastante útil não somente para o uso em simulações mas também no estudo e análise de dados amostrais. Aspectos técnicos de implementação do processo de aderência será explicado no decorrer do capítulo.

3.2 Metodologia de trabalho

O desenvolvimento desta biblioteca consistiu basicamente em reescrever o código implementado neste projeto comentado anteriormente para que pudesse ser reaproveitado por outros desenvolvedores.

Para o desenvolvimento deste projeto, foi alocado uma equipe formada por dois alunos bolsistas e alguns professores da área de estatística. O fluxo normal das atividades iniciava-se com o repasse de documentos (geralmente, algoritmos em linguagem matemática) que descreviam um ou mais subprocessos a serem implementados. Os algoritmos descritos nos documentos eram então estudados e, solucionadas dúvidas eventuais junto aos professores, traduzidos e adaptados para código C++ de modo a montar a aplicação.

Logo após, era realizada uma etapa de testes que consistiam em comparar os resultados obtidos de nossa implementação com os obtidos em outras ferramentas como o *InputAnalyser* e o *OpenOffice Spreadsheet*.

3.3 Licença

A filosofia software livre define a liberdade aos usuários rodar, copiar, distribuir, estudar, mudar e melhorar o software. Mais precisamente, ela se refere à quatro tipos de liberdade para os usuários do software:

- Liberdade para rodar o programa para qualquer propósito.
- A liberdade para estudar como o programa funciona e adaptá-lo à suas necessidades.
- Liberdade para redistribuir cópias.
- Liberdade para melhorar o programa e liberar essas melhorias ao público, de modo que toda comunidade se beneficie.

Um programa é software livre se os usuários têm todos esses tipos de liberdade.

Liberdade para rodar o programa significa a liberdade para cada pessoa ou organização usá-lo em qualquer tipo de sistema, para qualquer propósito, sem que seja obrigado a informar o desenvolvedor ou outra entidade específica. Nesta liberdade, é o propósito do usuário que importa, não o propósito do desenvolvedor;

Liberdade para redistribuir cópias deve incluir binários ou a forma executável do programa, também do código fonte, tanto nas versões modificadas ou não do programa. O código fonte é necessário para que possam ser feitas alterações e melhorias.

É possível pagar ou não para obter cópia de um software livre, mas não importando a forma que a cópia foi obtida, existe a liberdade para copiar e modificar o software, até mesmo de vender cópias. Também, é possível o desenvolvimento, distribuição e uso comercial.

Para que o software seja livre, ele deve ser liberado sob uma licença. Uma licença bastante usada em softwares livres é a GNU GPL (*GNU General Public License*) que além de garantir os direitos de liberdade do software livre, garante também que, ao um usuário redistribuir o software livre, os direitos de liberdade não podem ser alterados. Isto é conhecido na comunidade como *copyleft*. No entanto, a licença escolhida para esta biblioteca foi a GNU LGPL (*GNU Lesser General Public License*). A LGPL foi especialmente criada para ser usada (não necessariamente) por bibliotecas. Em contraste com a GPL, que obriga à programas que usem bibliotecas GPL também estar sob licença GPL, a LGPL não obriga que um programa que use uma biblioteca sob sua licença seja livre. Portanto, uma biblioteca sob licença LGPL pode ser usada em programas proprietários.

Cópias desta licença podem ser encontradas no código fonte da biblioteca, nos anexos deste relatório, ou em Free Software Foundation (2007).

3.4 Funcionamento

Nesta seção será discutido sobre o funcionamento de bibliotecas. O funcionamento interno dos procedimentos implementados será discutido na próxima seção.

Uma biblioteca é um conjunto de funções (subprogramas) usadas no desenvolvimento de software. Deste modo o código e os dados podem ser compartilhados e usados de forma modular. Em geral, bibliotecas não são executáveis. Executáveis e bibliotecas se referenciam entre eles por ligações (*links*) através do processo conhecido por “linkagem” que é efetuado pelo *linker*.

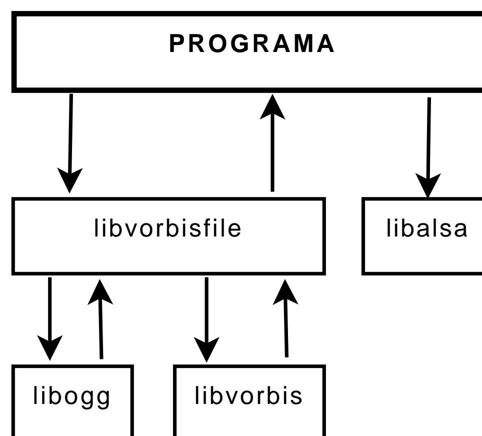


Figura 4: Ilustração de um programa que usa a biblioteca libvorbisfile.so para tocar arquivos no formato .ogg

Bibliotecas normalmente são classificadas como estáticas ou dinâmicas. A seguir encontra-se uma breve descrição desses tipos.

3.4.1 Bibliotecas estáticas

Uma biblioteca estática é um conjunto de precedimentos, funções externas e variáveis que são resolvidas em tempo de compilação e copiadas para dentro da aplicação alvo pelo compilador ou pelo linker, produzindo um arquivo objeto e um executável sem dependências. Entretanto o tamanho do executável gerado é maior que comparado à um que usa biblioteca dinâmica.

Antigamente só existiam bibliotecas estáticas. Uma de suas vantagens é que é permitido ao usuário linkar um programa com a biblioteca sem que seu código precise ser recompilado.

Devido a velocidade atual dos computadores e dos compiladores, esta característica não é mais tão importante. Todavia, na teoria, o código de uma biblioteca estática linkado à um executável deveria rodar levemente mais rápido que comparado ao de uma biblioteca compartilhada ou de vínculo dinâmico, o que acaba não sendo verdade na prática pois linkagem com bibliotecas estáticas podem aumentar em muito o tamanho do executável, e executáveis grandes, por sua vez, demoram mais para serem carregados e ocupam mais memória.

Bibliotecas estáticas podem ser fundidas com outras bibliotecas estáticas ou arquivos de objeto, durante a compilação/linkagem, para formar um único executável, ou elas podem ser carregadas durante a execução no espaço de memória do aplicativo em um deslocamento estático definido durante o processo de compilação e linkagem. Além de usar a memória de forma menos eficiente, quando uma biblioteca estática é atualizada, as aplicações não se beneficiam das melhorias feitas. Para ganhar acesso à essas melhorias, o desenvolvedor da aplicação deve linkar, novamente, sua aplicação com a nova versão da biblioteca, e não obstante, os usuários da aplicação devem substituir suas cópias pela última versão.

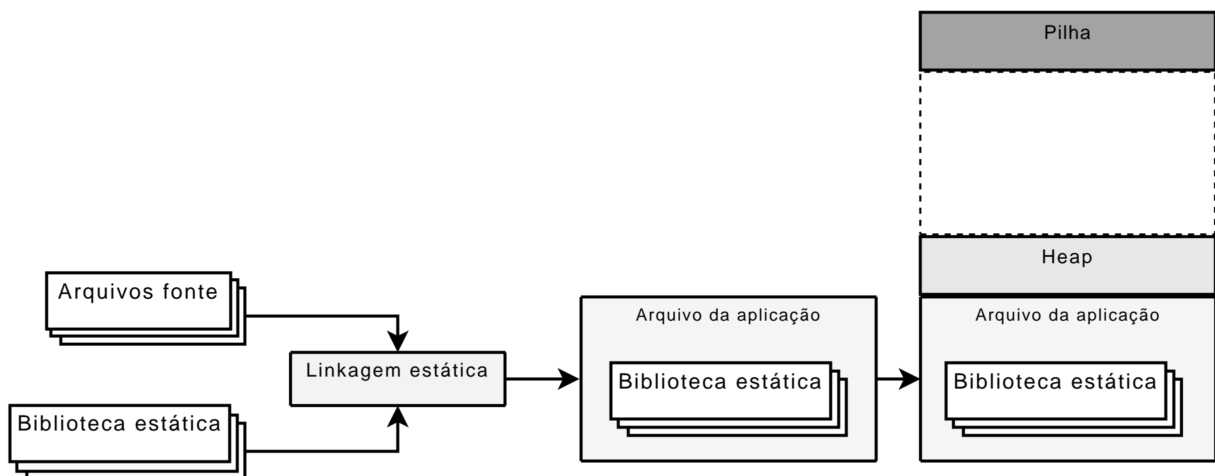


Figura 5: Aplicação usando bibliotecas estáticas

3.4.2 Bibliotecas de vínculo dinâmico

Bibliotecas dinâmicas ajudam a distribuir as funcionalidades de uma aplicação em módulos distintos que podem ser carregados no momento em que for necessário. Bibliotecas dinâmicas podem ser carregadas durante a iniciação de um programa ou durante sua execução. Bibliotecas que são carregadas durante a iniciação são chamadas de bibliotecas dependentes. As que são carregadas em tempo de execução são chamadas de bibliotecas de vínculo dinâmico.

Aplicações que usam bibliotecas dinâmicas têm um executável de menor tamanho. Se essas aplicações carregarem as bibliotecas somente quando for necessário ao invés de quando forem

iniciadas, além de sua iniciação se tornar mais rápida, ela contribuirá também para um uso mais eficiente da memória. Em alguns casos, o sistema operacional descarrega bibliotecas carregadas dinamicamente quando determina que não estão mais sendo usadas.

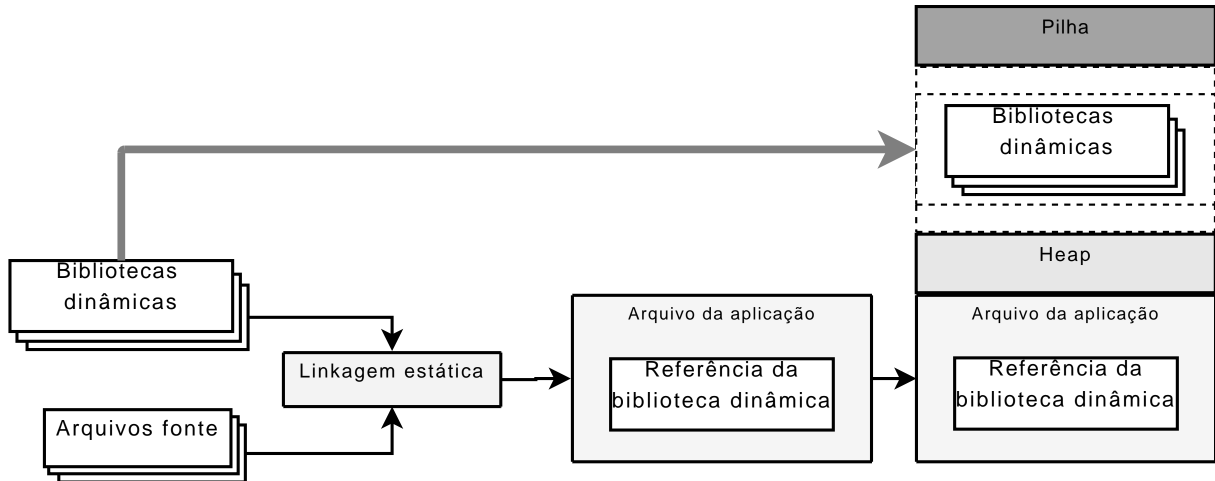


Figura 6: Aplicação usando biblioteca dinâmica

Devido à essas vantagens, decidiu-se por desenvolver essa biblioteca de aderência de forma que pudesse ser usada dinamicamente.

3.5 Aspectos de implementação

Durante o desenvolvimento de uma biblioteca deve-se levar em conta alguns aspectos que não são tratados no desenvolvimento de uma aplicação final. Por servir como utilitário para o desenvolvimento de outros sistemas, uma biblioteca deve conter uma completa e simples interface de comunicação. No âmbito de ser carregada e utilizada por programas e/ou outras bibliotecas externas, um tratamento especial deve ser aplicado à símbolos que necessitam ou não ter visibilidade externa. Nesta seção será comentado desde como foi construída internamente, como foi definida a interface até como pode ser usada externamente.

3.5.1 Definições globais

Basicamente a estrutura de implementação da biblioteca pode ser descrita em três partes. A primeira é onde se encontra definições globais de implementação e algoritmos de funções básicas para uso interno. A segunda engloba a implementação das funções distribuição de probabilidade e finalmente, a terceira parte é a que define a montagem da tabela de freqüências para ser usada no teste Qui-quadrado de aderência.

Para facilitar o uso da biblioteca por parte de desenvolvedores, decidiu-se por implementá-la completamente dentro de um espaço de nomes chamado `fit`. Dessa forma, nomes de funções e variáveis definidas na biblioteca têm menor chance de conflitar com identificadores definidos pelo desenvolvedor do sistema.

Foi criado então, um arquivo que contém definições globais de uso da biblioteca como os tipos, as exceções e parâmetro atrelados à montagem da tabela de freqüências. Este arquivo é descrito na listagem 3.1.

Listagem 3.1: Arquivo de definições globais

```
namespace fit
{
typedef double                  data_type;
typedef std::list<data_type>    data_set;

enum _bin_criteria
{
    STURGES,
    SQUAREROOT,
    OTHER
};
const unsigned int _MAX_BINS = 40;

struct value_exception
{
    std::string msg;
    value_exception(std::string _msg) : msg(_msg) {}
};
struct freq_table_exception
{
    std::string msg;
    freq_table_exception(std::string _msg) : msg(_msg) {}
};
struct distribution_exception
{
    std::string msg;
    distribution_exception(std::string _msg) : msg(_msg) {}
};
}
```

É importante notar o uso do ambiente de nomes `fit`. Como todas as definições de classes e funções da biblioteca foram descritas dentro deste ambiente de nomes, este não será mais mostrado em listagens posteriores.

Logo no início da listagem 3.1, percebe-se que o tipo `data_type` foi definido com um `double`. Este é o tipo que será usado para representar os dados que serão tratados nos algoritmos. A escolha de dupla precisão já o suficiente para obter resultados satisfatoriamente corretos. Esta definição serve para facilitar possíveis alterações na precisão escolhida para os

dados e os cálculos. Logo abaixo é definido o tipo `data_set` que será largamente usado para representar um conjunto qualquer desses dados.

A enumeração `_bin_criteria` define os métodos possíveis para a estimação do número de intervalos da tabela de freqüências. A constante `_MAX_BINS` define o número máximo padrão de intervalos, independentemente do número calculado por algum desses métodos.

Para o tratamento de exceções foram criadas três estruturas que servem para descrever os três tipos de exceções possíveis de serem geradas na biblioteca.

value_exception representa as possíveis exceções geradas por valor impróprio em alguma rotina. Por exemplo de precisão não atingida em rotinas de cálculo iterativo.

freq_table_exception representa exceções ligadas à tabela de freqüência como por exemplo, uma tabela de menos de cinco intervalos no cálculo da estatística qui-quadrado.

distribution_exception representa exceções geradas especificamente nas distribuições como por exemplo, valores negativos para estimar parâmetros de uma LogNormal

Além das definições de tipos, há também um arquivo (listagem 3.2) que contém definição de constantes e de subrotinas que são utilizadas em vários pontos internos da biblioteca. Por esse motivo foi criado um novo espaço de nomes denominado `utils`.

Listagem 3.2: Definição de subrotinas utilitárias da biblioteca

```
#ifndef BORLAND
const data_type DATAMIN = std::numeric_limits<data_type>::denorm_min();
#else
const data_type DATAMIN = 1.0e-30; // It's approx. float min. No problem.
#endif

const long double PI = 3.1415926535897932384626433832795028841968;

unsigned int _bin_width (_bin_criteria _criteria, int _samples_number, unsigned int _max_bins
    = _MAX_BINS);

data_type _chisquare_statistic(const freq_table& _table);
data_type _chisquare_pf(data_type chisquare_stat, unsigned int df);

template < class T > T _mean (const typename std::list < T >& values)
{
    if (values.empty ())
        return 0;
    T sum = 0;
    for (typename std::list < T >::const_iterator i = values.begin ();
    i != values.end (); i++)
        sum += *i;
```

```
    return sum / values.size ();
}

template < class T > T _variance (const std::list < T >& values)
{
    if (values.empty ())
        return 0;
    T temp = 0;
    T mean_value = mean (values);
    for (typename std::list < T >::const_iterator i = values.begin ();
    i != values.end (); i++)
        temp += (*i − mean_value) * (*i − mean_value);

    return temp / (values.size () − 1);
}

template < class T > T _variance (T mean, const std::list < T >& values)
{
    T temp = 0;
    for (typename std::list < T >::const_iterator i = values.begin ();
    i != values.end (); i++)
        temp += (*i − mean) * (*i − mean);

    return temp / (values.size () − 1);
}

template < class T > T _std_dev (const std::list < T >& values)
{
    if (values.empty ())
        return 0;
    T var_value = _variance (values);
    return std::sqrt (var_value);
}
```

Desta listagem podemos citar o método `_bin_width` que calcula quantos intervalos são necessárias para a tabela de freqüência dados o critério, o tamanho da amostra e o número máximo de intervalos possíveis.

Depois são definidos os métodos `_chisquare_statistic` e `_chisquare_pf`. O primeiro calcula o valor da estatística de teste qui-quadrado e o segundo realiza o teste de significância de um valor da estatística qui-quadrado em função dos graus de liberdade descrita na seção 2.1.1.

Os demais métodos servem para calcular a média, variância e desvio padrão de um conjunto de dados.

3.5.2 Distribuições de probabilidade

A implementação de distribuições de probabilidade envolveu um estudo das possíveis funcionalidades que objetos dessas distribuições deveriam dispor. Devido à finalidade de uso desta biblioteca, decidiu-se implementar o máximo de funcionalidade possível para cada distribuição de probabilidade. Para facilitar o entendimento dessas funcionalidades, uma classe abstrata foi criada como base para as outras classes de distribuições. Os membros dessa classe, tanto quanto a relação dela com suas implementações podem ser vistas no diagrama de classes do apêndice A.1

Percebe-se que o único atributo desta classe é uma lista dos parâmetros da distribuição (`_parameters`). Existem também outros membros protegidos na classe como por exemplo seu construtor padrão que inicializa a semente de gerador de números aleatórios do sistema; a função `random_number` que retorna um número aleatório distribuído uniformemente no intervalo $(0,1)$.

Há certas funções finais que estão implementadas nesta classe base. Essas funções servem para garantir a integridade das distribuições. A função `check_values`, da listagem 3.3, serve para checar se um conjunto de dados está de acordo com os limites da função distribuição. Para isso, basta que a classe que descrever uma distribuição, implemente o método `in_range`, que verifica se um dados valor pertence ou não ao intervalo da distribuição.

Listagem 3.3: Implementação da função que check_values

```
bool base_distrib::check_values(const data_set& values) const
{
    for (data_set::const_iterator it = values.begin(); it != values.end(); it++)
        if (!in_range(*it))
            return false;

    return true;
}
```

Outra função implementada na própria classe base é a `expected_freq`, descrita na listagem 3.4, que realiza o cálculo da freqüência esperada absoluta do intervalo [from_number, to_number] dado a quantidade total de valores do conjunto de dados. Se este último valor não for fornecido, então a função `expected_freq` retornará a proporção no intervalo (ver equação 2.1;

Listagem 3.4: Implementação da função expected_freq

```
data_type base_distrib::expected_freq(data_type from_number, data_type to_number, unsigned int
    total) const
{
```

```
    return total * (cumulative_function(to_number) - cumulative_function(from_number));
}
```

Esta função implementa a equação 2.3 e é necessário que nas subclasses seja implementado o método que calcula a função acumulada da distribuição, `cumulative_function`.

Dentre as funções já citadas acima, existe também um "protótipo" de outras funções que são implementadas em cada distribuição:

- As funções auto-explicativas `get_mean`, `get_variance`, `get_mode`, onde esta última pode gerar uma exceção caso a distribuição não tenha um valor modal único.
- A função `density_function` que implementa a função densidade de probabilidade da distribuição.
- As funções `get_params` e `get_params_number` que retornam, respectivamente, a lista de parâmetros e a quantidade de parâmetros da distribuição (útil na hora de calcular o número de graus de liberdade no teste qui-quadrado)
- A função `check_parameters` que faz a checagem de consistência dos parâmetros de uma distribuição.
- A função `get_random` que retorna um número aleatório gerado de acordo com a distribuição.
- Finalmente a função `set_params_mle` que estima os parâmetros da distribuição a partir de um conjunto de dados, através de estimadores de máxima verossimilhança.

Cada distribuição de probabilidade implementada nesta biblioteca, especializa esta classe base. Alguns ações foram tomadas durante a implementação das distribuições para facilitar a programação. Uma delas foi criar uma enumeração que serve como índice do vetor de parâmetros da distribuição. Pelo motivo do número de parâmetros ser variável entre os tipos de distribuição, esta é de fato, uma solução elegante para indexar a lista de parâmetros. A listagem 3.5 mostra o exemplo para a distribuição normal. Deste modo, para acessar a média da distribuição, basta somente escrever `_parameters[MEAN]`.

Listagem 3.5: Enumeração dos parâmetros da distribuição normal

```
enum _norm_params
{
    MEAN,
    STDDEV,
    PARAMS_NUMBER
};
```

Existem duas formas para a criação de um objeto que representa uma distribuição. Uma delas é fornecendo a lista de parâmetros e a outra é fornecendo o conjunto de dados que será usado para a estimação dos parâmetros no método `set_params_mle`.

A primeira forma descrita normalmente é usada quando o desejado é gerar números aleatórios de uma função pré-definida ou fazer uso da função densidade de probabilidade da distribuição (`density_function`). No entanto a distribuição pode ser criada também com parâmetros pré-definidos para posteriormente ser usada em testes de aderência com alguns conjunto de dados.

A segunda forma geralmente é utilizadas para criar distribuições que serão usadas em testes de aderência. Com somente um conjunto de dados disponíveis, um procedimento que utiliza bastante esta forma de construção é criar todas as distribuições usando o conjunto de dados para posteriormente realizar testes de aderência e decidir qual distribuição gerada representa melhor o conjunto de dados.

3.5.3 Tabela de freqüência

Normalmente na estatística, pouco se absorve somente do conjunto cru de dados. É possível calcular os valores de média, variância do conjunto mas para que se possa melhor interpretá-los é necessário que esses dados sejam organizados, resumidos e apresentados de certa forma. No caso da realização de testes Qui-quadrado de aderência, a forma de apresentação usada é através da tabela de freqüências.

O diagrama de classes implementadas para a construção da tabela de freqüências pode ser visto em A.2. É possível criar tabelas de freqüências de dados qualitativos e também de variáveis discretas mas esta implementação focou em criar tabelas de dados contínuos.

Como descrito no diagrama, `freq_table_entry` é uma classe que representa a entrada ou o intervalo na tabela de freqüências. A tabela de freqüências em si é um agrupamento desses intervalos. Portanto, a classe `freq_table` especializa uma lista da STL (*Standart Library* – Biblioteca padrão) do C++. A STL fornece um conjunto de estruturas de dados e algoritmos para tipos de dados genéricos através de *templates*. A classe `freq_table` estende uma lista de dados genéricos mas com o tipo dos dados pré-definido. Assim, como em todos os *templates*, as dependências são resolvidas em tempo de compilação.

Existem dois modos de construir um objeto da tabela de freqüência. Percebe-se que nos dois casos é necessário fornecer um objeto de alguma distribuição de probabilidade devidamente construído (ilustrado na figura 7) pois é necessário a disponibilidade de uma função que calcule

a freqüência esperada nos intervalos, caso contrário a tabela ficaria incompleta.

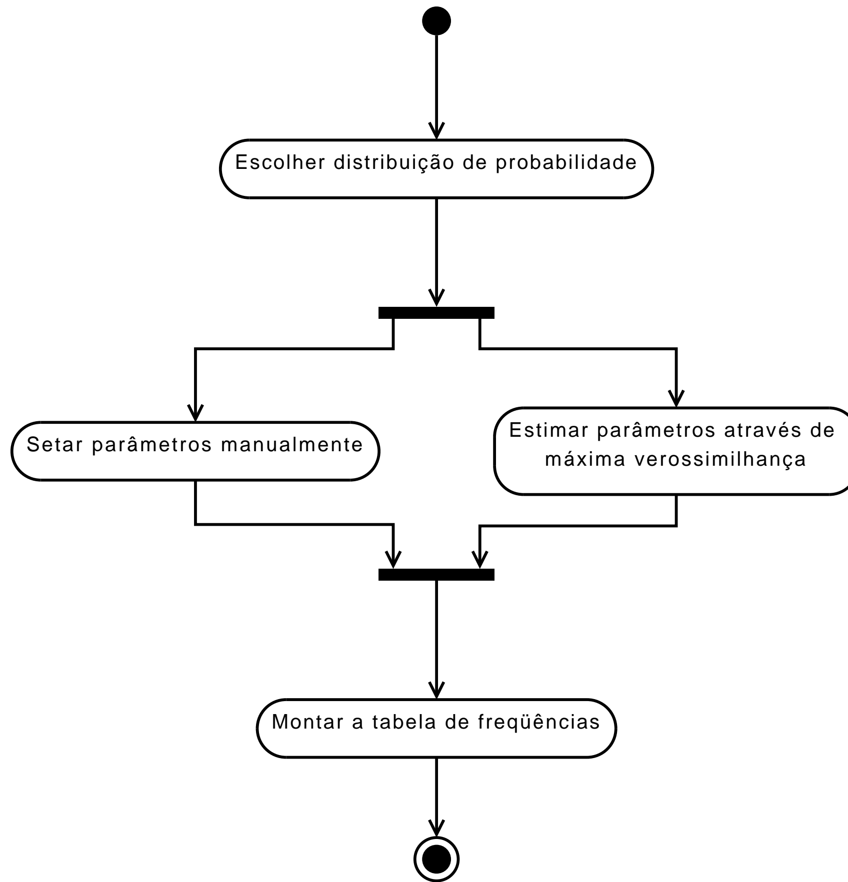


Figura 7: Diagrama de atividades para a construção da tabela de freqüência

A diferença entre os dois modos de construção consiste na forma de escolher o número de intervalos para a classificação inicial dos dados na tabela. Um construtor recebe o número de intervalos que devem ser usados e o outro construtor recebe o critério para o cálculo do número de intervalos mais o número o número total de intervalos possíveis (com valor padrão definido como `_MAX_BINS`). Os critérios implementados foram o de Sturges ($N_{intervalos} = 1.5 + 3.222 \cdot log_{10}(s)$) e o da raiz ($N_{intervalos} = \sqrt{s}$) onde s é a quantidade de dados do conjunto.

Um diagrama de atividades mais detalhado da construção da tabela de freqüência pode ser visto na figura 8.

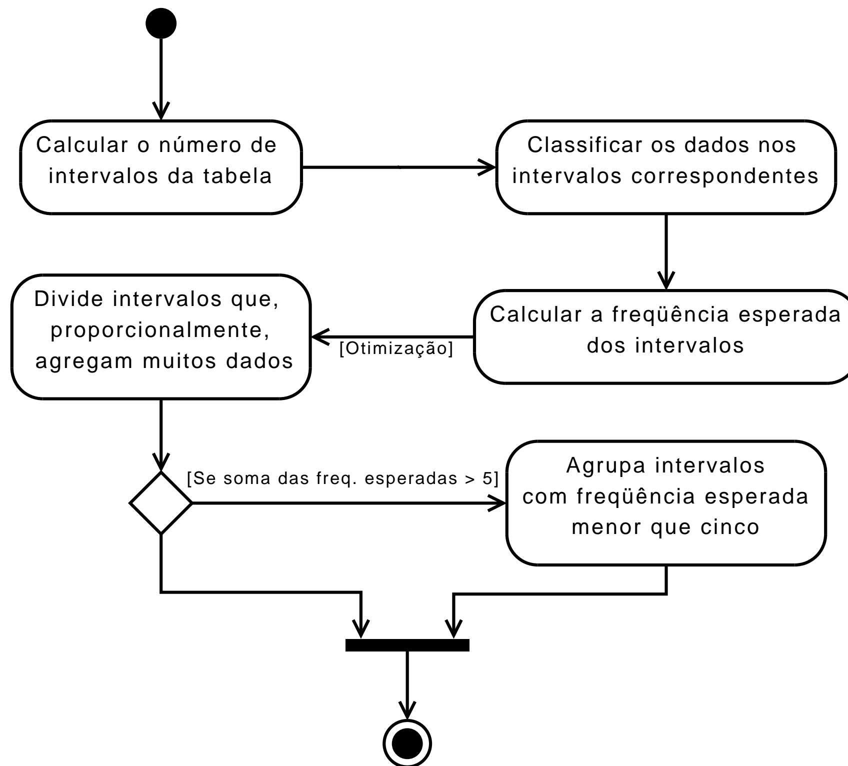


Figura 8: Diagrama de atividades mais detalhado da construção da tabela de freqüência

A listagem 3.6 mostra a implementação das atividades anteriores à otimização descritas neste diagrama.

Listagem 3.6: Implementação da função que constrói a tabela de freqüência

```
void freq_table::init_members(const base_distrib& distrib, unsigned int intervals)
{
if (_values.empty())
    throw freq_table_exception("No␣values␣on␣frequency␣table␣construction");

if (intervals <= 0)
    throw freq_table_exception("Negative␣number␣of␣intervals␣on␣frequency␣table␣construction␣:
        " + intervals);

data_type minor = *min_element(_values.begin(), _values.end());
data_type major = *max_element(_values.begin(), _values.end());

data_type difference = major − minor;

if (std::isnan(difference)) //difference != difference
    throw freq_table_exception("Constant␣values␣on␣frequency␣table␣construction");

data_type step = (difference * 1.01) / intervals;
```

```
std::vector<data_set> intervals_values(intervals);

for (data_set::const_iterator i = _values.begin(); i != _values.end(); i++)
{
    int auxtmp = std::floor( (*i - minor) / step );
    intervals_values.at(auxtmp).push_back(*i);
}

for (unsigned int i = 0; i < intervals; i++)
{
    data_type interv_min = minor + (i * step);
    data_type interv_max = minor + ((i * step) + step);
    data_type interv_exp_freq = distrib.expected_freq(interv_min, interv_max, _values.size());

    freq_table_entry new_entry(intervals_values.at(i), interv_min, interv_max, interv_exp_freq
        );

    push_back(new_entry);
}
}
```

Logo após o cálculo das freqüências esperadas nos intervalos, é necessário fazer duas otimizações na tabela recém criada. A primeira delas visa estimar e dividir intervalos que agregam valores demasiadamente se comparado com os outros intervalos. Isso normalmente acontece quando a amostra é composta por muitos dados pertencendo à uma faixa e alguns dados, possivelmente de “ruídos”, localizados longe desta faixa predominante. Inicialmente os dados são classificados em intervalos de tamanho fixo; isso acarreta em dados desta faixa predominante na quantidade de valores, serem classificados em poucos intervalos, fazendo com que esta tabela seja pouco representativa, o que dificulta a aderência dos dados à alguma distribuição. Uma ilustração desta classificação inconveniente pode ser vista no histograma da figura 9.

O método desenvolvido para atenuar esse problema consiste em verificar, em cada intervalo da tabela, se a freqüência observada não é dez vezes maior que a média estimada nos intervalos[1], se em algum intervalo isso for verdade, então divide-se o intervalo ao meio e o testa novamente. E assim sucessivamente.

A implementação deste método de teste e divisão dos intervalos, está descrita na listagem 3.7.

Listagem 3.7: Implementação da função que teste e divide intervalos da tabela de freqüência

```
void freq_table::divide_intervals(const base_distrib& distrib)
{
    for (freq_table_t::iterator it = begin(); it != end() ; )
    {
```

[1] A freqüência observada média é obtida dividindo-se o número de dados da amostra pelo número de intervalos da tabela

```
if ( it->get_observed_freq() > (10.0 * (_values.size()/size())) ) //Barbetta criteria.
{
    data_type minor = it->get_min_value();
    data_type major = it->get_max_value();
    data_type medium = (minor + major) / 2.0;
    data_set actual_interval_v = it->get_entry_values();
    data_set previous_interval_v;
    data_set subsequent_interval_v;
    for (data_set::const_iterator itV = actual_interval_v.begin(); itV !=
        actual_interval_v.end(); itV++)
    {
        if (*itV < medium)
        {
            previous_interval_v.push_back(*itV);
        }
        else
        {
            subsequent_interval_v.push_back(*itV);
        }
    }
    if (previous_interval_v.size() == 0)
    //this happens when there are too many values repeated.
    {
        it++;
    }
    else
    {
        freq_table_entry previous_interval(previous_interval_v, minor, medium, distrib
            .expected_freq(minor, medium, _values.size()));
        freq_table_entry subsequent_interval(subsequent_interval_v, medium, major,
            distrib.expected_freq(medium, major, _values.size()));
        it = erase(it);
```

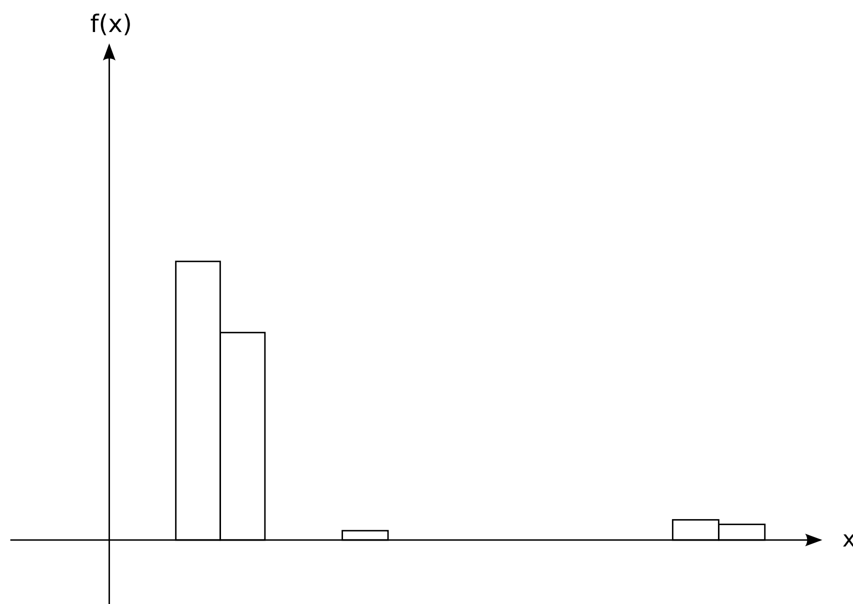


Figura 9: Exemplo de histograma pouco representativo

```
                it = insert(it, subsequent_interval);
                it = insert(it, previous_interval);
            }
        }
        else { it++; }
    }
}
```

A segunda otimização feita após a construção da tabela de freqüência consiste em analisar possíveis intervalos em que a freqüência esperada seja menor que cinco, e uní-los à um intervalo adjacente. A condição em que nenhum intervalo da tabela de freqüência tenha freqüência esperada menor que cinco, é necessária para o cálculo da estatística de qui-quadrado (χ^2). A implementação desta rotina de otimização pode ser vista na listagem 3.8.

Listagem 3.8: Implementação da função une intervalos cuja freqüência esperada é menor que cinco

```
void freq_table::merge_intervals()
{
    freq_table_t::iterator it_temp;
    freq_table_t::iterator end_it = end();
    end_it--;

    data_type expected_freq_sum = 0;
    for (freq_table_t::iterator it = begin(); it != end(); it++)
    {
        expected_freq_sum += it->get_expected_freq();
    }

    if (expected_freq_sum < 5) return; //return if it is impossible to merge.

    //Merge entry whose expected frequency is less than 5
    for (freq_table_t::iterator it = begin() ; it != end() ; )
    {
        if (it->get_expected_freq() < 5)
        {
            if (it == end_it)
            {
                it_temp = it;
                it_temp--;
                it->merge(*it_temp);
                erase(it_temp);
            }
            else
            {
                it_temp = it;
                it_temp++;
                it_temp->merge(*it);
                it = erase(it);
            }
        }
```

```
            else
            {
                it++;
            }
        }
}
```

É bem possível que a soma da freqüência esperada de todos os intervalos da tabela seja menor que cinco, isso acontece quando não foi escolhida uma boa distribuição de probabilidade. Este caso, que resulta em uma péssima aderência aos dados, pode ser observado na figura 10.

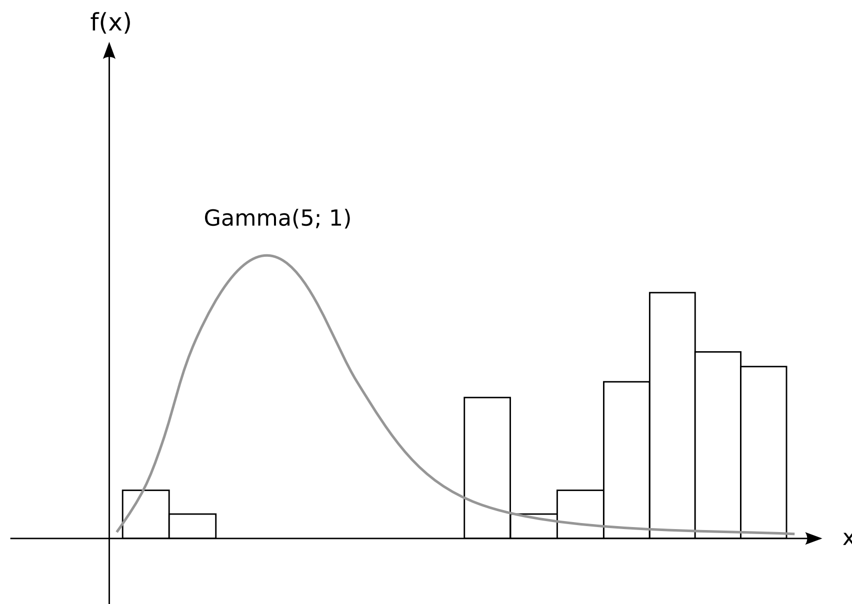


Figura 10: Exemplo de um caso onde a função distribuição escolhida não representa bem os dados

3.5.4 Interface da biblioteca

Se a biblioteca estivesse implementada somente da forma descrita até o momento, já seria possível criar funções distribuição, gerar números aleatórios, criar tabelas de freqüência e realizar teste de aderência, mas executando passos como o descrito na figura 7. Mesmo após implementada as funcionalidades, os desenvolvedores costumam criar uma interface de acesso às funções básicas da biblioteca para que sua utilização seja possível caso seu carregamento ocorra durante o tempo de execução do programa (ver seção 3.4.2).

A interface criada para esta biblioteca é a descrita na listagem 3.9.

Listagem 3.9: Interface da biblioteca

```
#if __GNUC__ >= 4
#pragma GCC visibility push(default)
```

```
#endif //__GNUC__

namespace fit
{
    enum _distribution_type
    {
        BETA,
        EXPONENTIAL,
        GAMMA,
        LOGNORMAL,
        NORMAL,
        TRIANGULAR,
        UNIFORM,
        WEIBULL
    };

    struct chisquare_test_result
    {
        data_type chi_statistic; //the test statistic
        unsigned int degrees_freedom; //the number of degrees of freedom
        data_type chi_p_value; //the significance probability of the test
    };

    typedef std::auto_ptr<base_distrib> distrib_ptr;

    DLLEXPORT distrib_ptr get_distribution(const data_set& values, int dist_type);
    DLLEXPORT distrib_ptr get_distribution(const params_list& params, int dist_type);

    DLLEXPORT freq_table get_freq_table(const data_set& values, int dist_type, int criteria,
        unsigned int max_bins = _MAX_BINS);
    DLLEXPORT freq_table get_freq_table(const data_set& values, const base_distrib& distrib,
        int criteria, unsigned int max_bins = _MAX_BINS);

    DLLEXPORT chisquare_test_result get_chisquare_test(const freq_table& table);

#if __GNUC__ >= 4
#pragma GCC visibility pop
#endif //__GNUC__
}
```

As funções definidas na interface são `get_distribution`, `get_freq_table` e a função `get_chisquare_test`. O parâmetro `dist_type` presente em algumas dessas funções representa a função distribuição escolhida de acordo com a enumeração `_distribution_type`. A implementação dessas funções descrevem uma forma com que a biblioteca poderia ser usada no código fonte da aplicação.

As diretivas de pré-compilação `#pragma GCC visibility`, servem para, caso a biblioteca seja compilada com GCC versão 4 ou superior e opção `-fvisibility=hidden`, explicitar que os símbolos definidos neste arquivo devem permanecer visíveis durante o processo de linkagem.

A função `get_chisquare_test` (listagem 3.10) retorna uma estrutura contendo, respectivamente, o resultado da estatística de teste qui-quadrado, o número de graus de liberdade, e o a probabilidade de significância do teste (ou valor-p).

Listagem 3.10: Implementação da função `get_chisquare_test`

```
fit::chisquare_test_result fit::get_chisquare_test(const freq_table& table)
{
    chisquare_test_result result = {0, 0, 0};

    result.chi_statistic = utils::_chisquare_statistic(table);
    result.degrees_freedom = table.get_df();
    result.chi_p_value = utils::_chisquare_pf(result.chi_statistic, result.degrees_freedom);

    return result;
}
```

As funções sobrecarregadas `get_distribution`, diferem somente na forma de construção do objeto que representa a distribuição. Na listagem 3.11, está descrita a função que estima os parâmetros por MLE para a construção da distribuição.

Listagem 3.11: Implementação da função `get_distribution`

```
fit::distrib_ptr fit::get_distribution(const data_set& values, int type)
{
    base_distrib* distrib_pointer = 0;

    switch (type)
    {
        case BETA:
            distrib_pointer = new beta_distrib(values);
            break;
        case EXPONENTIAL:
            distrib_pointer = new expo_distrib(values);
            break;
        case GAMMA:
            distrib_pointer = new gamm_distrib(values);
            break;
        case LOGNORMAL:
            distrib_pointer = new lognorm_distrib(values);
            break;
        case NORMAL:
            distrib_pointer = new norm_distrib(values);
            break;
        case TRIANGULAR:
            distrib_pointer = new tria_distrib(values);
            break;
        case UNIFORM:
            distrib_pointer = new unif_distrib(values);
            break;
        case WEIBULL:
            distrib_pointer = new weib_distrib(values);
```

```
            break;
        default:
            throw distribution_exception("Distribution␣not␣found");
            break;
    }

    return distrib_ptr(distrib_pointer);
}
```

A função get_freq_table (listagem 3.12) retorna a tabela de freqüência montada com os dados de values, a distribuição escolhida em dist_type, o critério para o cálculo do número de intervalos criteria, e o número máximo de intervalos max_bins.

Listagem 3.12: Implementação da função get_freq_table

```
fit::freq_table fit::get_freq_table(const data_set& values, int dist_type, int criteria,
    unsigned int max_bins)
{
    distrib_ptr dist = get_distribution(values, dist_type);
    return freq_table(values, *dist, (_bin_criteria) criteria, max_bins);
}
```

Percebe-se que o retorno da função get_distribution se dá através do tipo distrib_ptr que é definido como std::auto_ptr<base_distrib>. O tipo auto_ptr é uma classe genérica da STL que serve para abstrair o conceito de alocação dinâmica de memória em C++. Ela mantém um ponteiro que foi criado dinamicamente através do comando new, e o apaga quando ela mesma é destruída. No caso da função get_freq_table, o objeto criado dentro da função get_distribution é apagado ao fim do contexto. Além disso, a classe auto_ptr fornece a semântica estrita de um ponteiro.

4 *Conclusão*

Apesar do título deste trabalho ser bastante geral, a motivação para desenvolvê-lo nasceu da necessidade de sua aplicação na área de simulação computacional. Contudo, aderência dos dados à uma função distribuição de probabilidade é equivalente à criação de hipóteses sobre os dados em questão. Útil em qualquer momento que se desejar obter uma função matemática que represente um conjunto de dados para a geração de variáveis aleatórias.

Este trabalho teve como diferencial a criação de uma biblioteca especializada em realizar testes de aderência sobre um conjunto de dados. Entretanto, durante o desenvolvimento, muitas outras funcionalidades foram adicionadas, permitindo que a gama de aplicações na área estatística aumentasse consideravelmente.

A experiência de implementar vários conceitos estatísticos foi bastante enriquecedora. Além disso, o desenvolvimento de uma biblioteca dinâmica fez com que eu me aprofundasse mais nas áreas de compiladores e portabilidade do código além de muitas outras áreas da Ciência da Computação.

4.1 Trabalhos futuros

Muitos trabalhos hão de ser feitos no futuro. Devido à filosofia do código livre, outros desenvolvedores poderão acrescentar novas funcionalidades e fornecer melhorias ao código. Para este efeito, um primeiro trabalho futuro deveria ser montar a documentação completa do código implementado até este instante.

Após isso, outras funções de distribuição de probabilidade poderiam ser implementadas. O suporte para funções contínuas já existe. Deve-se então verificar possíveis alterações para que distribuições de probabilidade de variáveis discretas também possam ser implementadas. Ainda nas distribuições de probabilidade, a variável de deslocamento (*offset*) também pode ser adicionada.

Por enquanto o único teste disponível é o de qui-quadrado. Um trabalho seria estudar da

estatística descritiva, outras formas de organização e tratamento dos dados para oferecer suporte à aplicação de outros testes de aderência como o de Kolmogorov-Smirnov, Lilliefors, Anderson-Darling, etc, e também suas implementações.

Referências Bibliográficas

BARBETTA, P. A. **Estatística aplicada às Ciências Sociais**. Florianópolis: Ed. da UFSC, 2006.

BARBETTA, P. A.; REIS, M. M.; BORNIA, A. C. **Estatística para cursos de engenharia e informática**. São Paulo: Atlas, 2004.

Free Software Foundation. **GNU Lesser General Public License**. 5 out. 2007. <http://www.gnu.org/licenses/lgpl.html>.

LAW, A. M.; KELTON, W. D. **Simulation modeling and analysis**. 2ª. ed. USA: McGraw-Hill, 1991.

LEVIN, J. **Estatística aplicada a Ciências Humanas**. 2ª. ed. São Paulo: Harbra, 1978.

NETO, P. L. d. O. C. **Estatística**. São Paulo: Edgard Blüncher, 1977.

NIST/SEMATECH. **e-Handbook of Statistical Methods**. 5 out. 2007. <http://www.itl.nist.gov/div898/handbook/>.

PRESS, W. H. et al. **Numerical Recipes in C: The Art of Scientific Computing**. 2ª. ed. [S.l.]: Cambridge University, 1992.

SIEGEL, S. **Estatística não-paramétrica para siências do comportamento**. São Paulo: McGraw-Hill, 1956.

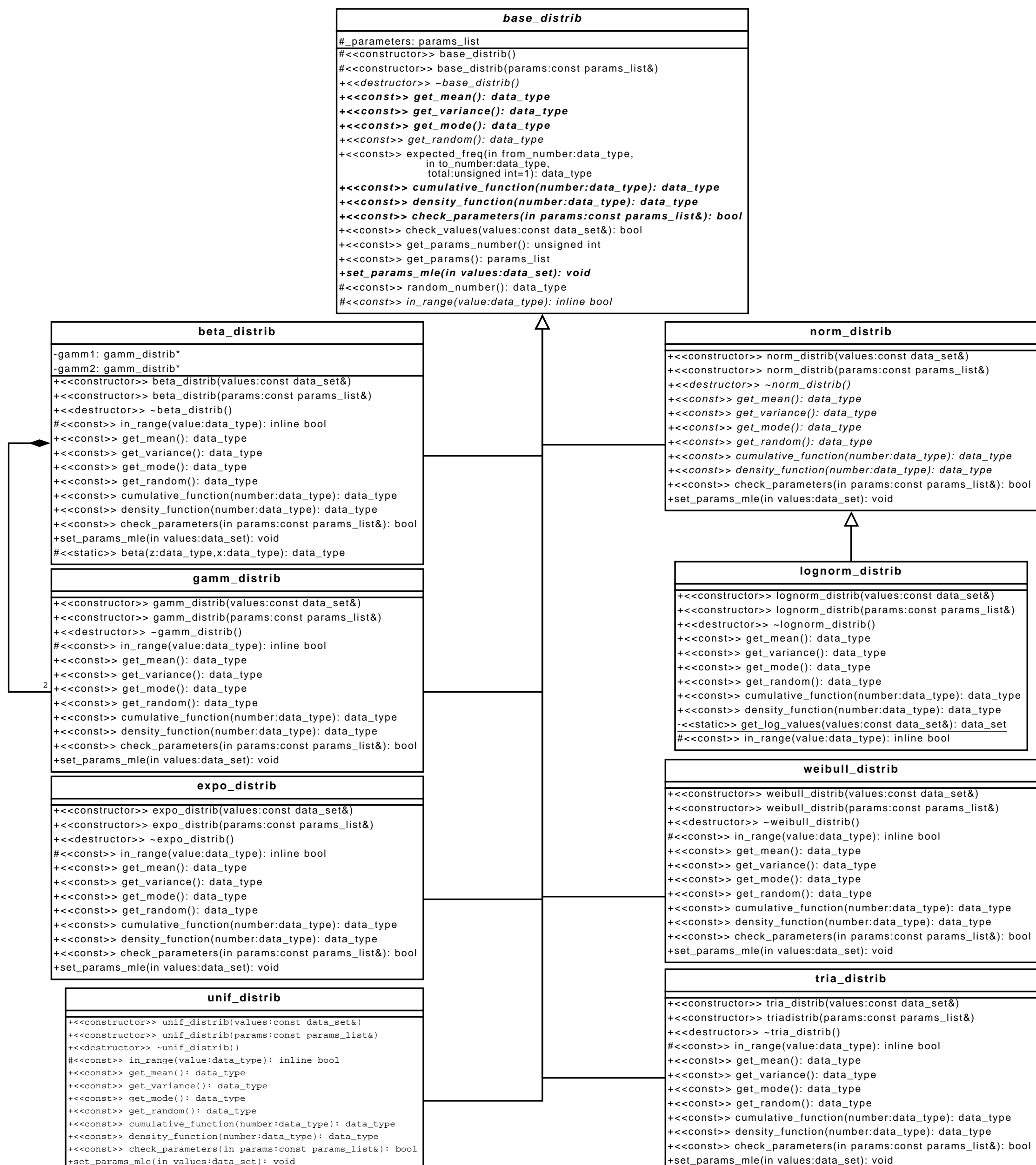
STROUSTRUP, B. **A linguagem de programação C++**. 3ª. ed. Porto Alegre: Bookman, 2000.

TENÓRIO, M. B. **Reconhecimento de modelos de probabilidade**. Dissertação (Mestrado) — Departamento de Informática e Estatística - UFSC, mar. 2005.

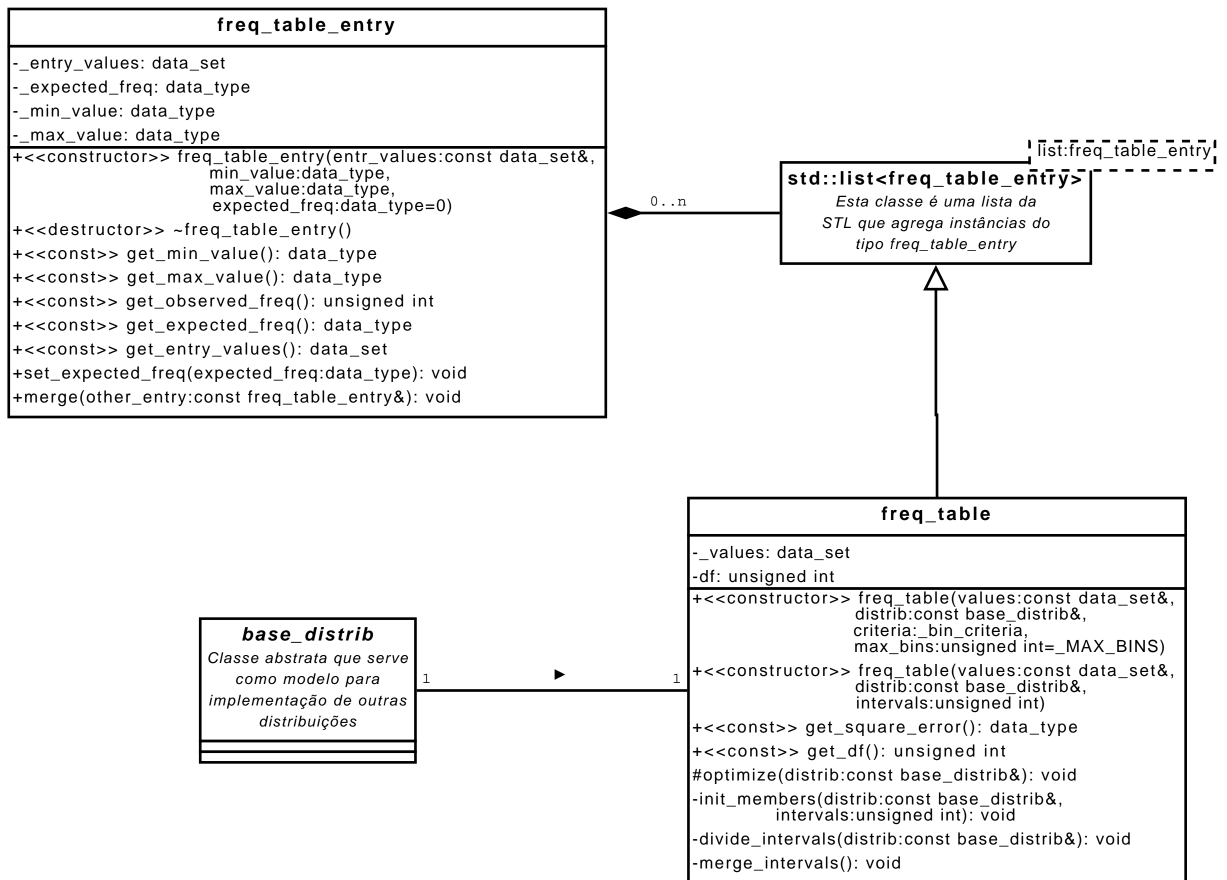
WEISSTEIN, E. W. **Wolfram MathWorld**. 1 out. 2007. <http://mathworld.wolfram.com/topics/ContinuousDistributions.html>.

APÊNDICE A – Diagramas de classes

A.1 Diagrama de classes de distribuições de probabilidade



A.2 Diagrama de classes da tabela de freqüências



ANEXO A – GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

    1) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

    2) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

ANEXO B – Artigo

Uma biblioteca de funções estatísticas para o apoio no desenvolvimento de aplicações com aderência de dados.

Sanjay Formighieri

Departamento de Informática e Estatística
Centro Tecnológico
Universidade Federal de Santa Catarina

`sanjay@inf.ufsc.br`

Abstract. *This work involves the study, group and implementation of algorithms that calculate probability distribution's parameters, probability density function and distribution function as well as random number generating and data treatment like the construction of a frequency table. This study proposes the construction of a C++ library flexible and portable which supplies necessary tools for goodness-of-fit tests.*

Resumo. *Este trabalho envolve o estudo, agrupamento e implementação de algoritmos para cálculo de fórmulas matemáticas complexas necessárias para a obtenção de parâmetros das funções de distribuição de probabilidade, o cálculo das funções de distribuição acumulada e de densidade de probabilidade, e geração de números aleatórios. Além de algoritmos para o tratamento dos dados, como por exemplo a montagem de tabelas de freqüências. Este estudo teve como finalidade a construção de uma biblioteca em C++ flexível e portável que fornece ferramentas necessárias para teste de aderência em conjuntos de dados.*

1. Introdução

A Simulação computacional faz parte da área que compreende as soluções por métodos numéricos de sistemas complexos de áreas como astrofísica, ecossistemas, construção civil, indústria petroquímica, entre outras em que sejam necessárias a realização de previsões ou onde os problemas a serem resolvidos são impossíveis ou inviáveis de serem reproduzidos para fins de

estudo. Como alguns exemplos podemos citar o dimensionamento de tubulações de poços de petróleo; estudo do impacto de desmatamentos ambientais; simulação de ação de proteínas e de seqüências de código genético.

Para que simulações de sistemas reais possam ser realizadas, modelos físicos ou lógicos desses sistemas devem ser construídos. Os modelos físicos normalmente são uma representação do sistema em uma escala diferente. Já nos modelos lógicos suas características são expressadas por equações matemáticas.

De acordo com Law e Kelton (1991, p. 325), quase todos os sistemas reais contém uma ou mais fontes de comportamento aleatório. Durante a simulação desses tipos de sistemas, é extremamente importante garantir a aleatoriedade dessas variáveis tornando o modelo de simulação o mais fiel possível ao sistema real (precaução necessária para certificar que a solução obtida pode ser aplicada com certa previsibilidade).

Para isso, amostras dessas fontes aleatórias devem ser coletadas e analisadas para que se possa obter um padrão que exemplifique os dados dessas amostras. É nesse contexto que esse trabalho foi desenvolvido. O mesmo oferece uma forma prática de encontrar funções probabilísticas que representam o padrão de um conjunto de dados.

"Na estatística, os métodos para reconhecimento de modelos de probabilidade são conhecidos como teste de aderência." (TENÓRIO, 2005). Neste caso, os testes de aderência servem para verificar se uma dada função densidade de probabilidade representa um respectivo conjunto de dados. Alguns exemplos destes testes são: Teste Qui-quadrado, Teste de Kolmogorov-Smirnov, Teste de Lilliefors, Teste de Anderson-Darling, entre outros. (BARBETTA; REIS; BORNIA, 2004), (LAW; KELTON, 1991), (NETO, 1977).

2. Importância

Existem algumas ferramentas já disponíveis para efetuar testes de aderência sobre um conjunto de dados, entretanto, atualmente se tratam de ferramentas dispendiosas e não flexíveis para aplicação em sistemas de simulação probabilística. Exemplos como InputAnalyser da Rockwell e o Statistica da StatSoft, realizam testes de aderência, geram gráficos e relatórios mas não dispõem de certas funcionalidades como, por exemplo, retro-alimentar um sistemas de simulação com informações necessárias para a randomicidade de suas variáveis.

Quando de frente com esse tipo de problema, os programadores desses sistemas decidem por reimplementar as funções estatísticas e, mesmo após isso, acabam engessando-as ao produto desenvolvido dificultando, também, a futura reutilização de código por outros desenvolvedores.

Por se tratar de um problema bastante comum entre os projetistas de sistemas, a solução foi agrupar essas funcionalidades em um meio que facilitasse a reutilização do código e a correção de *bugs*, neste caso, em uma biblioteca de funções.

3. Trabalho

Uma biblioteca é um conjunto de funções (subprogramas) usadas no desenvolvimento de software. Deste modo o código e os dados podem ser compartilhados e usados de forma modular. Em geral, bibliotecas não são executáveis. Executáveis e bibliotecas se referenciam entre eles por ligações (*links*) através do processo conhecido por “linkagem” que é efetuado pelo *linker*.

A biblioteca chamada *libfit*, foi desenvolvida para prover funcionalidades na área da estatística. Além de implementar rotinas básicas como o cálculo da média, variância e estatística qui-quadrado, é possível também construir uma tabela de freqüência de conjuntos de dados, e realizar o teste qui-quadrado de aderência nestes conjuntos.

As distribuições de probabilidade implementadas neste trabalho incluem uniforme, triangular, exponencial, normal, lognormal, gamma, weibull e beta. Para cada uma dessas distribuições é possível realizar o cálculo da função densidade de probabilidade; função acumulada; estimar parâmetros através dos estimadores de máxima verossimilhança; calcular a média, desvio padrão e moda da amostra com base nos parâmetros da distribuição; gerar números aleatórios.

A motivação inicial para o agrupamento dessas funcionalidades foi a de, dado um conjunto de valores, era necessário encontrar uma função geradora de números aleatórios da distribuição de probabilidade que melhor o representa. Resumidamente este processo consiste em supor distribuições de probabilidade; estimar seus parâmetros através dos estimadores de máxima verossimilhança; realizar o teste de aderência e posteriormente escolher pela distribuição que melhor aderiu aos dados. Além de ser bastante útil na análise de dados amostrais, este processo é bastante útil em simulações computacionais onde é necessário gerar números aleatórios de variáveis de um ambiente.

4. Conclusão

Apesar do título deste trabalho ser bastante geral, a motivação para desenvolvê-lo nasceu da necessidade de sua aplicação na área de simulação computacional. Contudo, aderência dos dados à uma função distribuição de probabilidade é equivalente à criação de hipóteses sobre os dados em questão. Útil em qualquer momento que se desejar obter uma função matemática que

represente um conjunto de dados para a geração de variáveis aleatórias.

5. Bibliografia

BARBETTA, P. A.; REIS, M. M.; BORNIA, A. C. **Estatística para cursos de engenharia e informática**. São Paulo: Atlas, 2004.

LAW, A. M.; KELTON, W. D. **Simulation modeling and analysis**. 2 ed. USA: McGraw-Hill, 1991.

NETO, P. L. d. O. C. **Estatística**. São Paulo: Edgard Blüncher, 1977.

TENÓRIO, M. B. **Reconhecimento de modelos de probabilidade**. Dissertação (Mestrado) – Departamento de Informática e Estatística - UFSC, mar. 2005.