Ney André de Mello Zunino

Um Ambiente para Experimentação de Tratamento de Dependência em Análise de Risco

Florianópolis

2006

UNIVERSIDADE FEDERAL DE SANTA CATARINA

CENTRO TECNOLÓGICO

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Um Ambiente para Experimentação de Tratamento de Dependência em Análise de Risco

Ney André de Mello Zunino

Florianópolis

2006

Resumo

Processos complexos como aqueles inerentes à instalação de um poço de petróleo são cercados de incertezas. O elevado custo associado a cada operação que compõe tais processos dá relevância especial às incertezas, demandando esforços no sentido de entendê-las e minimizá-las. O uso de simulações computadorizadas na estimação de variáveis críticas como o tempo de cada operação (e, conseqüentemente, seu custo) é prática bastante estabelecida. Os métodos mais freqüentemente empregados são aqueles que se baseiam em modelos probabilísticos e na geração de números aleatórios (Métodos Monte Carlo). Entretanto, existe um aspecto relevante que não é comumente ou propriamente considerado: as eventuais dependências existentes entre as operações em estudo. Este trabalho trata do desenvolvimento de um ambiente computacional que permite a realização de experimentos de simulação envolvendo controle de dependência entre operações. O tratamento de dependência é baseado no uso de uma ferramenta matemática conhecida como Cópula, que oferece uma forma relativamente simples de se modelar dependências e de se produzir resultados mais fiéis e confiáveis.

Palavras-chave: simulação, Monte Carlo, dependência, cópula.

Abstract

Complex processes like those related to the installation of oil wells are surrounded by uncertainties. The high costs associated with each of the operations that make up those processes emphasize the relevance of the uncertainties and call for efforts in order to understand and minimize them. The use of computerized simulations in the estimation of critical variables such as the time required for the completion of each operation (and, hence, its cost) is common practice. The methods employed more frequently are those based on probabilistic models and the generation of random numbers (Monte Carlo Methods). However, there is one relevant aspect which is mostly neglected or not properly considered: the eventual dependencies among the operations under study. This work focuses on the development of a software environment that allows the execution of simulation experiments which provide control over operations interdependence. The interdependence handling is made possible by the use of a mathematical tool known as Copula, which provides a relatively simple way to model dependencies and produce more faithful and reliable results.

Keywords: simulation, Monte Carlo, dependency, copula.

Ney André de Mello Zunino

Um Ambiente para Experimentação de Tratamento de Dependência em Análise de Risco

Projeto para o Trabalho de Conclusão de Curso em Ciência da Computação da Universidade Federal de Santa Catarina.

Orientador:
Paulo José de Freitas Filho, Dr.

Co-orientadores:
Dalton Francisco de Andrade, Dr.
Pedro Alberto Barbetta, Dr.

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

Florianópolis

2006

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução p. 8

1.1 Introdução Geral . . . . p. 8

1.2 Justificativa . . . . p. 9

1.3 Objetivo Geral . . . . p. 10

1.4 Objetivos Específicos . . . . p. 10

1.5 Estrutura do Trabalho . . . . p. 11

2 Conceitos Fundamentais p. 12

2.1 Simulação . . . . p. 12

2.1.1 Introdução . . . . p. 12

2.1.2 Tipos de Sistemas . . . . p. 12

2.1.3 Modelos de Simulação . . . . p. 13

2.1.3.1 Modelos Estáticos versus Dinâmicos . . . . p. 13

2.1.3.2 Modelos Determinísticos versus Não-determinísticos . . . p. 13

2.1.3.3 Modelos Contínuos versus Discretos . . . . p. 13

2.1.4 Método Monte Carlo . . . . p. 14

2.2 Cópulas . . . . p. 14

2.2.1 Introdução . . . . p. 14

2.2.2 Distribuições Marginais . . . . p. 15

2.2.3 Medida de Dependência . . . . p. 16
2.2.4 Cópula de Clayton . . . . p. 17
2.2.4.1 Função . . . . p. 17
2.2.4.2 Função Geradora . . . . p. 17
2.2.4.3 Algoritmo . . . . p. 17
2.2.4.4 Parâmetro de Dependência . . . . p. 18
2.2.5 Cópula de Frank . . . . p. 18
2.2.5.1 Função . . . . p. 18
2.2.5.2 Função Geradora . . . . p. 18
2.2.5.3 Algoritmo . . . . p. 18
2.2.5.4 Parâmetro de Dependência . . . . p. 19
2.2.6 Cópula Gaussiana . . . . p. 20
2.2.6.1 Algoritmo . . . . p. 21
2.2.7 Cópula *t* . . . . p. 22
2.2.7.1 Algoritmo . . . . p. 22
2.2.8 Cópulas Multivariadas . . . . p. 23

3 Desenvolvimento p. 25

3.1 Introdução . . . . p. 25
3.2 Metodologia de Trabalho . . . . p. 26
3.3 Visão Geral do Software . . . . p. 27
3.4 Instalação . . . . p. 29
3.5 Funcionamento . . . . p. 30
3.5.1 Configurando a Simulação . . . . p. 31
3.5.1.1 Configurando as Distribuições . . . . p. 31
3.5.1.2 Configurando a Medida de Dependência . . . . p. 32
3.5.2 Executando a Simulação . . . . p. 34

3.6 Aspectos de Implementação . . . . p. 35
3.6.1 Distribuições de Probabilidade . . . . p. 35
3.6.2 Cópulas . . . . p. 37
3.6.3 Experimentos . . . . p. 40
3.6.4 Validação de Domínio . . . . p. 41
3.6.5 Gerência de Memória . . . . p. 41
3.6.6 *Threads* . . . . p. 46

4 Conclusão p. 49
4.1 Trabalhos Futuros . . . . p. 49
4.1.1 Interface Gráfica para Cópulas Multivariadas . . . . p. 49
4.1.2 Aderência com Cópulas . . . . p. 50
4.1.3 Exportação para Excel . . . . p. 51

Referências p. 52

Apêndice A – Diagramas de Classes p. 53
A.1 Diagrama de classes de distribuições de probabilidade . . . . p. 53
A.2 Diagrama de classes de cópulas . . . . p. 54
A.3 Diagrama de classes de experimentos . . . . p. 55

Apêndice B – Esquemas dos Algoritmos de Cópulas p. 56
B.1 Esquema de Cópula Bivariada . . . . p. 56
B.2 Esquema de Cópula Multivariada . . . . p. 57

Lista de Figuras

1 Um dos ambientes de teste de algoritmos . . . . p. 26

2 O software e sua Interface MDI . . . . p. 28

3 O instalador NSIS em ação . . . . p. 30

4 Configurando uma distribuição do experimento . . . . p. 31

5 Definindo a medida de dependência entre as distribuições . . . . p. 33

6 Janela com indicação de progresso da simulação . . . . p. 34

7 Resultados de simulação com cópula . . . . p. 35

8 Esquema para geração de cópulas bivariadas . . . . p. 39

Lista de Tabelas

1 Polinômios de ajuste usados na implementação da cópula de Frank . . . . . . p. 20

2 Campos e seções da janela de experimento . . . . p. 32

3 Campos da configuração de medida de dependência . . . . p. 33

Lista de Listagens

3.1 Trecho do script NSI de instalação do software . . . . p. 29

3.2 Classe abstrata Distribuicao . . . . p. 36

3.3 Implementação da função-membro `proximo` para distribuição uniforme . . p. 37

3.4 Implementação da função-membro `proximo` para distribuição Weibull . . . p. 37

3.5 Classe abstrata Copula . . . . p. 37

3.6 Classe CopulaFrank . . . . p. 38

3.7 Implementação do algoritmo de geração para a cópula de Frank . . . . p. 39

3.8 Classe abstrata Experimento . . . . p. 40

3.9 Implementação de `geraProximo` para experimento univariado . . . . p. 41

3.10 Implementação de `geraProximo` para experimento bivariado . . . . p. 41

3.11 Construtor da classe DistribuicaoGama . . . . p. 41

3.12 Classe ReferenceCountable . . . . p. 43

3.13 Classe RefCountPtr<T> . . . . p. 43

3.14 Código de execução da simulação . . . . p. 46

3.15 Tratador do evento `Show` da janela de progresso . . . . p. 47

3.16 Laço Principal de Execução da Simulação . . . . p. 47

1 *Introdução*

1.1 Introdução Geral

São inúmeras as aplicações que, hoje, se beneficiam do uso de simulações computadorizadas para a produção de estimativas e outros resultados que dão suporte a importantes tomadas de decisão, tanto de pontos-de-vista estratégicos quanto econômicos. Simulações baseadas em modelos não-determinísticos —com ou sem uso de computadores— são coletivamente designadas método ou simulação *Monte Carlo*.

As simulações computadorizadas brilham quando soluções baseadas em modelos puramente analíticos são impossíveis ou inviáveis. Modelos de simulação não-determinísticos —que oferecem um tratamento probabilístico para a modelagem de problemas— são convertidos em modelos computacionais com relativa facilidade, permitindo uma produção de estimativas rápida e de baixo custo.

Com seu alto poder de processamento e baixo custo, os computadores tornam-se opções atraentes para a realização de experimentos baseados em algoritmos de geração de números aleatórios —a essência do método Monte Carlo. Além disso, possibilitam também a realização de ambientes gráficos que facilitam e dão apoio à modelagem e simulação de processos.

Em contraste a todo o avanço que se teve desde que a simulação passou de fato a fazer parte do arsenal de gerentes e analistas, existem aspectos que, por simplicidade ou pura imprudência, não recebem a atenção merecida. Um exemplo notório é o fato de que, em um determinado processo, nem sempre é verdade que as operações que o compõem (e sobre as quais se está produzindo estimativas através de simulações) ocorrem de maneira totalmente independente. De modo geral, uma observação mais cautelosa revelaria que as operações de um processo estariam, em maior ou menor grau, atreladas e, portanto, sujeitas aos efeitos dessa interdependência. Assim, a negligência pura e simples desse fato deve, fatalmente, levar à produção de estimativas (e conseqüentes tomadas de decisão) de acuidade duvidosa.

A matemática há muito oferece ferramentas que vêm ao auxílio do analista em situações

como essa. Medidas de dependência podem ser introduzidas em um dado modelo de simulação de mais de uma forma. Uma dessas formas, cujo emprego no tratamento de dependências em simulações e análise de risco é relativamente recente, chama-se Cópula. A Cópula oferece um mecanismo simples e eficaz de embutir a noção de dependência em simulações baseadas em modelos não-determinísticos.

Neste trabalho, será apresentado um ambiente computacional que implementa alguns dos tipos existentes de Cópulas. A ferramenta de software construída permite que experimentos de simulação sejam criados e executados, levando-se em consideração as possíveis relações de dependência entre as operações do modelo de estudo.

O trabalho foi desenvolvido durante o período de junho de 2005 a junho de 2006, em que o autor esteve vinculado ao projeto E&P Risk (Petrobras), junto ao laboratório Performance Lab., da Universidade Federal de Santa Catarina.

O ambiente computacional ora apresentado não se trata do sistema E&P Risk em si, mas sim de uma ferramenta desenvolvida em paralelo. Essa ferramenta serviu de base para uma série de experimentos e testes que foram conduzidos a medida que a pesquisa de Cópulas avançava. Foi somente na fase final do projeto E&P Risk que, validados os algoritmos e sua implementação no sistema de experimentos, o novo código e conceitos resultantes da pesquisa puderam ser incorporados ao sistema principal.

1.2 Justificativa

Sistemas de apoio à decisão são parte fundamental do planejamento estratégico da absoluta maioria das empresas de grande porte, nos ramos mais diversos. De General Motors à Petrobras, o estudo e tratamento de incertezas são atividades imprescindíveis para melhorar as chances de sucesso em mercados cada vez mais competitivos. A GM, por exemplo, usa simulações para fazer previsões sobre lucro ou risco de lançamento de determinado produto. Já a empresa do ramo de petróleo pode produzir estimativas de custo, retorno e risco de um determinado projeto de perfuração de poço, que esteja em estudo.

Dada a importância das ferramentas de simulação em tantos e diversos contextos, evidencia-se também a importância do aprimoramento de técnicas existentes e da pesquisa sobre novas técnicas que visem aumentar a qualidade dos resultados.

É natural pensar que modelos de simulação mais ricos tendam a produzir estimativas mais confiáveis e fiéis à realidade (o desafio muitas vezes está justamente em saber dosar o nível

de detalhamento versus o provável ganho). Dentre a gama de detalhes que se pode considerar diante de um sistema em estudo está a necessidade de se reconhecer as interdependências entre seus componentes de incerteza (VOSE, 2001). É interessante, portanto, que um ambiente de simulação seja capaz de tomar como entrada não apenas detalhes das operações em estudo, mas também medidas de suas interdependências.

1.3 Objetivo Geral

O objetivo geral deste trabalho é a criação de um ambiente computacional (um *software*) que permita a definição e execução de experimentos de simulação variados em que se possa expressar relações de dependência entre operações através do uso de Cópulas.

1.4 Objetivos Específicos

Um apanhado um pouco mais detalhado dos objetivos deste trabalho incluiria os itens a seguir:

- estudar conceitos de medida de dependência;
- estudar algoritmos para a geração bivariada de números aleatórios para as Cópulas selecionadas;
- implementar e testar os algoritmos encontrados ou definidos;
- criar uma interface gráfica amigável, mas que ao mesmo tempo permita um alto grau de configurabilidade na criação de experimentos;
- otimizar os algoritmos implementados;
- possibilitar o fácil aproveitamento dos resultados produzidos pelo ambiente de modo a aplicá-los em outras ferramentas de estatística;
- avaliar o efeito do emprego de cópulas com diferentes graus de correlação nos resultados produzidos.

1.5 Estrutura do Trabalho

O presente capítulo teve como objetivo introduzir o leitor aos principais assuntos e problemas relacionados ao tema do trabalho, bem como estabelecer objetivos e apresentar justificativas para sua execução. Toda essa introdução foi feita de maneira breve e superficial, ficando os detalhes reservados aos capítulos seguintes.

O capítulo 2 traz uma revisão sobre os conceitos fundamentais que deram sustentação ao trabalho. A seção 2.1 dedica-se ao tópico *simulação*, incluindo entidades, tipos de sistemas, modelos de simulação, método Monte Carlo, etc. A aplicação de simulações para análise de risco é também abordada e, por fim, o capítulo fecha com algumas idéias sobre modelagem de dependência.

Ainda no capítulo 2, a seção 2.2 apresenta a ferramenta matemática eleita para a modelagem de dependências na execução deste trabalho: a *cópula*. Após uma breve introdução geral sobre cópulas, são mencionadas as cópulas e distribuições marginais tratadas no escopo do trabalho. A seção segue com uma descrição formal de cada uma das cópulas e dos algoritmos desenvolvidos para a geração bivariada de amostras dependentes. Por fim, são apresentadas considerações sobre o uso de cópulas para a geração multivariada de valores dependentes, sendo apresentada uma solução concebida como parte da pesquisa que levou a este trabalho.

O desenvolvimento do ambiente computacional é o foco do capítulo 3. Com linguagem um pouco mais técnica e dirigida por conceitos de ciência da computação e engenharia de software, são apresentados e discutidos tópicos e desafios que acompanharam a implementação do projeto.

O capítulo 4 e último visa dar fechamento aos tópicos apresentados, oferecendo uma avaliação do progresso obtido ante aos objetivos inicialmente propostos. Após as considerações finais, são listadas e brevemente descritas algumas idéias para trabalhos subseqüentes.

2 *Conceitos Fundamentais*

2.1 Simulação

2.1.1 Introdução

Simulação é uma técnica através da qual se procura imitar algum sistema real com o intuito de estudar seu comportamento. As finalidades de tal estudo podem ser muitas mas, de maneira geral, o interesse está em avaliar diferentes possibilidades, visando à otimização ou alguma outra espécie de melhoramento.

O princípio da simulação está em se criar um modelo numérico que represente o estado de um sistema de interesse e, através de um conjunto de métodos, realizar transformações sobre esse modelo de modo a simular possíveis mudanças de estado. O mapeamento do funcionamento de um sistema para um modelo numérico permite que se obtenha estimativas para situações hipotéticas que, de outro modo, seriam de realização excessivamente difícil ou custosa.

Segundo (FILHO, 2001), um *sistema* é uma coleção de *entidades* que atuam e interagem juntas para o cumprimento de algum objetivo lógico. Dessa maneira, a criação do modelo numérico mencionado anteriormente consiste em identificar e estruturar propriedades relevantes das entidades e de suas interações. A granularidade ou nível de detalhamento aplicados ao modelo está diretamente relacionada aos objetivos do estudo em questão (LAW; KELTON, 1991). Assim, uma abstração considerada adequada para o estudo de um dado problema pode ser simples ou complexa demais quando o enfoque do estudo é diferente.

2.1.2 Tipos de Sistemas

Os sistemas são classificados em dois tipos: *contínuos* e *discretos*. Sistemas contínuos são aqueles em que o estado de suas entidades (também, suas *variáveis de estado*) está sendo continuamente modificado com relação ao tempo. Em sistemas discretos, por outro lado, as mudanças de estado ocorrem em instantes separados e identificáveis.

Na prática, muitos sistemas possuem tanto elementos contínuos quanto discretos. A relevância de um ou outro tipo de mudança é o que permite classificar o sistema.

Além disso, os objetivos específicos de um dado estudo podem permitir que, por exemplo, um sistema contínuo dê origem a um modelo discreto que produza resultados satisfatórios naquele contexto. Deve-se mencionar que a situação inversa também é uma possibilidade, i.e., um sistema normalmente tido como discreto pode ser simulado através da construção de um modelo contínuo que o represente com grau de fidelidade aceitável. Existem fatores práticos, como complexidade computacional, para se optar por um ou outro tipo de modelo.

2.1.3 Modelos de Simulação

Dado um sistema a ser estudado através de simulação, já foi dito que é necessário criar-se uma abstração matemática que seja capaz de representar seus estados, entidades e interações. Os modelos resultantes dessa abstração são chamados *modelos de simulação* e podem ser classificados segundo uma série de critérios.

2.1.3.1 Modelos Estáticos versus Dinâmicos

Um modelo de simulação estático é um modelo que representa um sistema em um instante determinado ou de forma que o tempo não seja relevante (e.g. modelos de Monte Carlo que serão vistos em breve). Um modelo de simulação dinâmico, por sua vez, permite que as variáveis de estado reflitam mudanças a medida que o tempo avança.

2.1.3.2 Modelos Determinísticos versus Não-determinísticos

A diferença fundamental entre modelos determinísticos e modelos não-determinísticos está no fato de que esse último faz emprego de números aleatórios e funções de probabilidade. Em contraste, um modelo determinístico permite que se calcule sua saída exata uma vez que se conheça a entrada. O uso de modelos não-determinísticos, apesar de produzir valores estimados no lugar de resultados precisos, é interessante porque permite modelar fatores de incerteza inerentes à uma grande parte dos processos de um sistema.

2.1.3.3 Modelos Contínuos versus Discretos

Seguindo a descrição dada na diferenciação entre sistemas contínuos e sistemas discretos, tem-se que modelos contínuos são aqueles em que as mudanças de estado ocorrem continua-

mente, enquanto que em modelos discretos, elas ocorrem em um número contável de vezes. Todavia, como também mencionado anteriormente, essas definições não implicam que exista uma correspondência direta entre sistema contínuo e modelo contínuo e sistema discreto e modelo discreto. Os interesses do estudo e as características do sistema é que ditarão qual tipo de modelo oferece vantagem.

2.1.4 Método Monte Carlo

O "Método Monte Carlo" denota, na verdade, um conjunto de técnicas que se baseia no uso de números aleatórios e funções probabilísticas para o estudo de um determinado sistema. Muitos autores consideram qualquer solução na qual se construa modelos de simulação que dependam de variáveis aleatórias como sendo uma aplicação de um método Monte Carlo.

Dentre as classificações já apresentadas para modelos de simulação, o método Monte Carlo se enquadraria como estático (a variável tempo não possui papel relevante) e não-determinístico (baseado em números aleatórios).

O uso dos métodos de Monte Carlo se estende por diversas áreas, desde o cálculo de integrais definidas até simulações que visam à análise de risco.

2.2 Cópulas

2.2.1 Introdução

Cópulas são funções que assumem valores no intervalo $[0;1]$ e cujos argumentos também só assumem valores no intervalo $[0;1]$, caso das distribuições de probabilidade. Formalmente, essas funções são definidas como:

Definição (ANJOS et al., 2004): Uma cópula é qualquer função $C : [0;1]^n \mapsto [0;1]$ que tem as seguintes propriedades:

(i) $C(u_1, u_2, \ldots, u_n)$ é crescente em cada componente u_i;

(ii) $C(1, 1, \ldots, u_i, 1, \ldots, 1) = u_i \in [0;1] \, \forall i = 1, 2, \ldots, n$;

(iii) $\forall (a_1, a_2, \ldots, a_n)$, $(b_1, b_2, \ldots, b_n) \in [0;1]^n$ com $a_i \leq b_i$, tem-se que

$$\sum_{i_1=1}^{2} \cdots \sum_{i_n=1}^{2} (-1)^{i_1+\ldots+i_n} C(u_{1i_1}, \ldots, u_{ni_n}) \geq 0$$

, com $u_{j1} = a_j$ e $u_{j2} = b_j, j = 1, \ldots, n$.

Existem várias cópulas definidas na literatura (exemplos de cópulas podem ser vistos em (NELSEN, 1999)). Delas, quatro foram escolhidas para serem implementadas neste trabalho:

- cópula de Clayton;
- cópula de Frank;
- cópula Gaussiana;
- cópula *t*.

As duas primeiras pertencem à família das Cópulas Arquimedianas. As outras duas são membros da família das Cópulas Elípticas.

2.2.2 Distribuições Marginais

O ambiente desenvolvido neste trabalho permite que as cópulas acima listadas usem como distribuições marginais combinações das seguintes distribuições de probabilidade contínuas univariadas:

- uniforme;
- triangular;
- exponencial;
- normal;
- log-normal;
- Weibull; e
- gama.

Algumas das distribuições necessitaram de adaptações e aplicação de métodos alternativos em suas implementações de modo a poderem ser usadas como marginais de uma cópula. Em especial, a distribuição gama ocasionou bastante dificuldade. O algoritmo usado para sua implementação ficou da seguinte forma:

Entrada: $u > 0, \alpha > 0, \beta > 0$, sendo u o argumento e α e β os parâmetros da distribuição.

(a) Obter os graus de liberdade v (podendo ser fracionário:

$$v = 2\alpha$$

(b) Calcular o logaritmo natural da função gama no valor α:

$$g = ln[\Gamma(\alpha)]$$

onde Γ é a função gama. Para a integração numérica, foi usado o algoritmo descrito em (PRESS et al., 1992)

(c) Calcular o ponto percentual de uma distribuição χ^2 (MILLER, 1975):

$$y = PP\chi^2(u, v, g)$$

(d) Transformar o ponto percentual de uma distribuição χ^2, y, num ponto percentual de uma distribuição Γ (LAW; KELTON, 1991):

$$x = \frac{\beta y}{2}$$

2.2.3 Medida de Dependência

Para realizar a simulação de um conjunto de observações bivariadas através das cópulas implementadas, torna-se necessário, além do fornecimento das distribuições univariadas marginais, apresentar um valor para o parâmetro θ da cópula. Como este parâmetro não tem uma interpretação prática clara, optou-se por dar a alternativa de fornecer o coeficiente de correlação τ de Kendall, que é um valor no intervalo $[-1, 1]$, sendo negativo quando se quer os dados bivariados correlacionados negativamente; e positivo quando se quer os dados bivariados correlacionados positivamente. Quanto mais próximo τ estiver de 1 (ou -1) a estrutura de correlação é mais forte.

Em se tratando da distribuição normal bivariada, a medida de dependência natural é o chamado coeficiente de correlação de Pearson, ρ. Como será visto na seção 2.2.6, essa distribuição pode ser gerada pela Cópula Gaussiana, considerando as marginais distribuições normais. Porém, se forem usadas outras distribuições como marginais, o coeficiente ρ (parâmetro da Cópula Gaussiana) deixa de representar a correlação de Pearson.

Em geral, a magnitude do coeficiente τ de Kendal é menor do que a magnitude do tradicional coeficiente de correlação de Pearson, ρ, para representar uma mesma estrutura de dependência, porém ele tem a vantagem de ser invariante a transformações estritamente crescentes

não-lineares, o que não acontece com o coeficiente de correlação de Pearson. Em se tratando de cópulas, o coeficiente τ é mais apropriado do que o coeficiente ρ, pois ele pode representar a dependência de uma cópula, independentemente de suas distribuições marginais. Na seção 2.2.6 será apresentada a relação entre τ e ρ sob a distribuição normal bivariada.

Para cada uma das cópulas implementadas será apresentada a relação entre o parâmetro de dependência da cópula e o coeficiente de correlação τ de Kendall, que é dada pela expressão:

$$\tau(X,Y) = 4 \int\int_{[0,1]^2} C_\theta(u,v) dC_\theta(u,v) - 1,$$

com $C_\theta(u,v)$ sendo a cópula geradora da distribuição conjunta de X e Y, com parâmetro de dependência θ.

Nas próximas seções, serão descritas as funções e algoritmos para cada uma das cópulas selecionadas.

2.2.4 Cópula de Clayton

2.2.4.1 Função

$$C_\theta^C(u,v) = [u^{-\theta} + v^{-\theta} - 1]^{-\frac{1}{\theta}},$$

com $u,v \in [0,1]$ e $\theta > 0$.

2.2.4.2 Função Geradora

$$\varphi(t) = \theta^{-1}(t^{-\theta} - 1),$$

com $t \in (0,1]$.

2.2.4.3 Algoritmo

Fornecidas, pelo usuário, as funções das distribuições marginais, F_1 e F_2, e o valor do parâmetro da cópula, θ, então, com base no algoritmo geral, a implementação da cópula de Clayton foi feita da seguinte maneira:

(a) Gerar u e w a partir de distribuições $U[0,1]$ independentes;

(b) Calcular:

$$v = C^{-1}_{V|U}(u,w) = \frac{\varphi'(u)}{\varphi'(\varphi^{-1}(\varphi(u)+\varphi(v)))} = \left[1 + \left(uw^{\frac{1}{\theta+1}}\right)^{-\theta} - u^{-\theta}\right]^{-1};$$

(c) Calcular $x = F_1^{-1}(u)$ e $y = F_2^{-1}(v)$. O par (x,y) gerado é um par de números aleatórios com dependência definida pelo parâmetro da cópula;

(d) Repetir (a) - (c) até obter a amostra bivariada do tamanho desejado.

2.2.4.4 Parâmetro de Dependência

A relação entre o coeficiente de correlação τ de Kendall e o parâmetro da cópula de Clayton é:

$$\theta = \frac{2\tau}{1-\tau}.$$

Como a cópula de Clayton é definida para $\theta > 0$, então, pela relação acima, deve-se ter $\tau > 0$. Assim, esta cópula é apropriada para modelar distribuições conjuntas de variáveis correlacionadas positivamente.

2.2.5 Cópula de Frank

2.2.5.1 Função

$$C^F_\theta(u,v) = \left(-\frac{1}{\theta}\right) ln\left[1 + \frac{\left(e^{-\theta u}-1\right)\left(e^{-\theta v}-1\right)}{e^{-\theta}-1}\right],$$

com $u,v \in [0,1]$ e $\theta \neq 0$.

2.2.5.2 Função Geradora

$$\varphi(t) = -ln\left[\frac{e^{-\theta t}-1}{e^{-\theta}-1}\right],$$

com $t \in (0,1]$.

2.2.5.3 Algoritmo

Fornecidas, pelo usuário, as funções das distribuições marginais, F_1 e F_2, e o valor do parâmetro da cópula, θ, então, com base no algoritmo geral, a implementação da cópula de

Frank foi feita da seguinte maneira:

(a) Gerar u e w a partir de distribuições $U[0,1]$ independentes;

(b) Calcular:

$$v = C^{-1}_{V|U}(u,w) = \frac{\varphi'(u)}{\varphi'(\varphi^{-1}(\varphi(u)+\varphi(v)))} = -\frac{1}{\theta} ln \left[\frac{we^{-\theta}+e^{-\theta u}-we^{-\theta u}}{e^{-\theta u}-we^{-\theta u}+w}\right];$$

(c) Calcular $x = F_1^{-1}(u)$ e $y = F_2^{-1}(v)$. O par (x,y) gerado é um par de números aleatórios com dependência definida pelo parâmetro da cópula;

(d) Repetir (a) - (c) até obter a amostra bivariada do tamanho desejado.

2.2.5.4 Parâmetro de Dependência

A relação entre o coeficiente de correlação τ de Kendall e o parâmetro θ da cópula de Frank é (EMBRECHTS; LINDSKOG; MCNEIL, 2001):

$$\tau = 1 - \frac{4[1-D(\theta)]}{\theta},$$

sendo $D(\theta) = \frac{1}{\theta}\int_0^\theta \frac{t}{e^t-1}\,dt$

Como essa integral não tem forma fechada, o cálculo de θ em função de τ foi feito de forma aproximada por uma função construída empiricamente por pares de dados (θ,τ) simulados a partir da expressão acima.

Como não foi possível encontrar uma função que se ajustasse bem aos pares (τ,θ), fez-se um ajuste para cada segmento do domínio, usando uma função quadrática. A seguir, as equações usadas para cada faixa do domínio:

No caso, $x = \tau$ e, portanto, deve satisfazer $0 < x < 1$. Para evitar problemas numéricos, o sistema só aceita $0{,}01 \leq x \leq 0{,}99$. Caso o usuário forneça um valor de τ fora desse intervalo, o sistema dá um alerta ao usuário. Quando o usuário fornece um valor de τ (e, conseqüentemente, tem-se o valor de $x = \tau$), o sistema calcula y. Se $\tau > 0$, então o parâmetro da cópula é $\theta = y$. Se $\tau < 0$, então $\theta = -y$. Com o valor de θ, o algoritmo de geração de uma amostra bivariada pode ser realizado.

Tabela 1: Polinômios de ajuste usados na implementação da cópula de Frank

Faixa	Expressão de $y(x)$
$0{,}01 < x \leq 0{,}20$	$0{,}00484 + 8{,}75842x + 2{,}60948x^2$
$0{,}20 < x \leq 0{,}40$	$0{,}33233 + 5{,}71441x + 9{,}66422x^2$
$0{,}40 < x \leq 0{,}60$	$3{,}8662 - 11{,}2978x + 30{,}0794x^2$
$0{,}60 < x \leq 0{,}75$	$45{,}241 - 144{,}69x + 137{,}71x^2$
$0{,}75 < x \leq 0{,}85$	$286{,}61 - 778{,}52x + 553{,}7x^2$
$0{,}85 < x \leq 0{,}91$	$1586{,}8 - 3831{,}8x + 2346x^2$
$0{,}91 < x \leq 0{,}95$	$11105 - 24633x + 13712x^2$
$0{,}95 < x \leq 0{,}975$	$83586 - 176567x + 93332x^2$
$0{,}975 < x \leq 0{,}99$	$935608 - 1919047x + 984227x^2$

2.2.6 Cópula Gaussiana

A cópula gaussiana é uma função que transforma distribuições normais univariadas em uma distribuição normal multivariada, considerando uma certa dependência. Ela não é uma cópula Archimediana, pois ela não possui função geradora.

Considerando o caso bivariado, seja ϕ a função de distribuição da normal padrão e seja ϕ_2 a função de distribuição da normal bivariada padrão com correlação de Pearson ρ, então a cópula gaussiana pode ser definida para todo z_i real $(i = 1{,}2)$ como

$$\phi_2(z_1,z_2) = C_\rho \left[\phi(z_1),\phi(z_2)\right]$$

ou

$$C_\rho(u,v) = \phi_2 \left[\phi^{-1}(u),\phi^{-1}(v)\right]$$

onde ϕ^{-1} é a inversa de ϕ, $u \in [0{,}1]$ e $v \in [0{,}1]$.

Tomando $u = \phi(z_1)$ e $v = \phi(z_2)$, a cópula gaussiana é a própria função de distribuição normal bivariada, mas, assim como as outras cópulas, a cópula gaussiana pode ser usada para construir uma distribuição conjunta com quaisquer marginais. Sendo (x,y) um vetor bidimensional, a função de distribuição conjunta de X e Y, $H(X,Y)$, pode ser definida pela cópula gaussiana e pelas marginais F_1 e F_2 por:

$$H(x,y) = C_\rho \left[F_1(x),F_2(y)\right].$$

Conforme apresentado em (SONG, 2000), a cópula gaussiana tem a seguinte expressão:

$$C_\rho^G(u,v) = \int_{-\infty}^{\theta^{-1}(u)} \int_{-\infty}^{\theta^{-1}(v)} \frac{1}{2\pi\sqrt{1-\rho^2}} e^{-\frac{s^2 - 2\rho st + t^2}{2(1-\rho^2)}} \, ds \, dt$$

onde u e v são valores do intervalo $[0{,}1]$ e ρ é o parâmetro de dependência da cópula Gaussiana.

Considerando a distribuição normal bivariada (isto é, uma distribuição conjunta gerada pela cópula gaussiana com distribuições marginais normais), o coeficiente τ de Kendall tem a seguinte relação com o coeficiente ρ:

$$\tau = \frac{2}{\pi} sen^{-1}(\rho)$$

O coeficiente τ é invariante por transformações monotônicas e, portanto, seu valor permanece o mesmo ao mudar as distribuições marginais. O mesmo não acontece com o coeficiente ρ. Ou seja, se forem geradas distribuições conjuntas com um dado valor para ρ, mas se marginais não forem normais, então não se pode dizer que o valor do coeficiente de correlação de Pearson será igual a ρ.

2.2.6.1 Algoritmo

O usuário precisa fornecer as funções das distribuições marginas F_1 e F_2 e o valor do parâmetro da cópula, ρ, sendo $-1 < \rho < 1$. Se o usuário fornecer τ, sendo $-1 < \tau < 1$, então o sistema calcula

$$\rho = sen\left(\frac{\pi}{2}\tau\right).$$

A seguir, os passos do algoritmo:

(a) Gerar um par $(z_1{,}z_2)$ de número aleatórios independentes, a partir da distribuição normal padrão;

(b) Calcular um par $(y_1{,}y_2)$ de observações normais bivariada padrão (média nula, variância unitária e correlação ρ), utilizando

$$y_1 = \sqrt{\frac{1-\rho}{2}} z_1 + \sqrt{\frac{1+\rho}{2}} z_2$$

$$y_2 = -\sqrt{\frac{1-\rho}{2}} z_1 + \sqrt{\frac{1+\rho}{2}} z_2;$$

(c) Calcular $u = \phi(y_1)$ e $v = \phi(y_2)$, onde ϕ é a função de distribuição normal padrão;

(d) Calcular $x = F_1^{-1}(u)$ e $y = F_2^{-1}(v)$. O par $(x{,}y)$ gerado é um par de números aleatórios gerados pela cópula Gaussiana;

(e) Repetir (a) - (d) até obter a amostra bivariada do tamanho desejado.

2.2.7 Cópula *t*

Seja $Z = (Z_1, Z_2)$ um vetor aleatório com distribuição normal bivariada, sendo o vetor de médias nulas, variâncias unitárias e correlação ρ. Seja S uma variável aleatória com distribuição qui-quadrado com V graus de liberdade, sendo $V > 2$ e sendo Z e S independentes. O vetor aleatório T, definido por

$$T = \mu + \sqrt{\frac{V}{S}} Z$$

tem distribuição *t* bivariada.

A cópula *t* transforma distribuições univariadas *t* numa distribuição bivariada t_2 com dependência definida por ρ. Assim, sendo t_2 uma distribuição *t* bivariada e t uma função de distribuição *t* univariada, tem-se:

$$t_2(Z_1, Z_2) = C_\rho \left[t(Z_1), t(Z_2) \right]$$

ou

$$C_\rho(u,v) = t_2 \left[t^{-1}(u), t^{-1}(v) \right].$$

Conforme (EMBRECHTS; LINDSKOG; MCNEIL, 2001), a cópula *t* tem a seguinte expressão:

$$C^t_\rho(u,v) = \int_{-\infty}^{t^{-1}(u)} \int_{-\infty}^{t^{-1}(v)} \frac{1}{2\pi\sqrt{1-\rho^2}} \left[1 + \frac{s^2 - 2\rho st + t^2}{V(1-\rho^2)} \right]^{\frac{V+2}{2}} ds\, dt$$

onde u e v são valores do intervalo $[0,1]$, ρ é o parâmetro de dependência e V é o outro parâmetro da Cópula *t*. Da mesma forma que a cópula Gaussiana, os argumentos da cópula *t* podem ser outras distribuições, gerando uma nova distribuição conjunta.

2.2.7.1 Algoritmo

O usuário precisa fornecer as funções das distribuições marginais F_1 e F_2, o parâmetro de dependência ρ $(-1 < \rho < 1)$ e o parâmetro número de graus de liberdade V, sendo $V > 2$ e inteiro. Caso o usuário opte por fornecer τ $(-1 < \tau < 1)$ no lugar de ρ, o sistema calcula ρ pela expressão aproximada:

$$\rho = sen\left(\frac{\pi}{2}\tau\right).$$

Observa-se que essa expressão é exata para distribuições normais bivariadas —aqui ela está sendo usada como uma aproximação. Seguem os passos:

(a) Gerar um par (z_1, z_2) de número aleatórios independentes, a partir da distribuição normal

padrão;

(b) Calcular um par (y_1,y_2) de observações normais bivariada padrão (média nula, variância unitária e correlação ρ), utilizando

$$y_1 = \sqrt{\frac{1-\rho}{2}}z_1 + \sqrt{\frac{1+\rho}{2}}z_2$$

$$y_2 = -\sqrt{\frac{1-\rho}{2}}z_1 + \sqrt{\frac{1+\rho}{2}}z_2;$$

(c) Gerar um número aleatório s com distribuição qui-quadrado e V graus de liberdade;

(d) Calcular

$$w1 = \sqrt{\frac{V}{s}}y_1$$

$$w2 = \sqrt{\frac{V}{s}}y_2;$$

(e) Calcular $u = t_V(w_1)$ e $v = t_V(w_2)$, onde t_V é a função de distribuição *t-Student* com V graus de liberdade;

(f) Calcular $x = F_1^{-1}(u)$ e $y = F_2^{-1}(v)$. O par (x,y) gerado é um par de números aleatórios gerados pela cópula *t* com V graus de liberdade;

(g) Repetir (a) - (e) até obter a amostra bivariada do tamanho desejado.

2.2.8 Cópulas Multivariadas

O desenvolvimento do presente trabalho visava, inicialmente, ao estudo e emprego de cópulas na geração bivariada de números aleatórios. Em outras palavras, o que se pretendia era permitir a criação e execução de experimentos que gerassem conjuntos de pares de valores correlacionados. Cada componente do par seguiria uma dada distribuição de probabilidade (distribuição marginal) e uma função-cópula seria usada para materializar a dependência entre os valores.

À medida que o trabalho progredia e que resultados positivos eram colhidos com a geração de pares correlacionados, a equipe permitiu-se avaliar estratégias para estender as capacidades do sistema, visando à geração multivariada de valores correlacionados. A literatura oferece alternativas para a modelagem de dependência nesses casos mas, em geral, tais alternativas impõem restrições quanto às distribuições marginais que podem ser usadas ou, ainda, não são sempre simples de se aplicar.

O trabalho de pesquisa da equipe resultou em uma solução satisfatória e que não sofre dos dois problemas mencionados acima. A solução é baseada em um *encadeamento de cópulas*, onde as dependências são analisadas e consideradas entre cada par de distribuições marginais, quando essas distribuições são dispostas em uma ordem linear.

De maneira um pouco mais formal, dadas as distribuições marginais $F_1, F_2, ..., F_n$, a aplicação da solução de cópulas encadeadas consiste, inicialmente, na definição dos $n-1$ coeficientes de correlação $\tau_1, \tau_2, ..., \tau_{n-1}$ entre cada par de distribuições, onde τ_1 estaria correlacionando F_1 e F_2, τ_2 estaria correlacionando F_2 e F_3 e assim por diante. Feito isso, o próximo passo consiste na construção das cópulas $C_1, C_2, ..., C_{n-1}$ correspondentes, que conjugariam as distribuições marginais e suas dependências.

O algoritmo inicia com a obtenção dos dois primeiros elementos x_1 e x_2 da tupla de resultados, usando a cópula C_1 normalmente, como se faz no caso bivariado. O diferencial vem a partir daí. Para as cópulas seguintes $(C_2, \ldots, C_{n-1})$, o valor de u usando internamente no algoritmo não deve ser um valor aleatório gerado, mas sim deve assumir o valor do último v da cópula anterior. Essa estratégia "amarra" as cópulas, formando uma corrente ou cadeia.

A partir de $C_i, 2 \leq i < n-1$, cada cópula produz um único novo elemento x_{i+1} da tupla de resultados. O diagrama B.2, nos anexos, ilustra o esquema do algoritmo de cópulas multivariadas.

3 *Desenvolvimento*

3.1 Introdução

Como já mencionado no capítulo 1, este trabalho girou em torno de um projeto maior, o software de análise de risco E&P Risk, desenvolvido pelo Performance Lab. (UFSC) para a Petrobras. Atualmente caminhando para sua 4ª versão, o software incorpora uma série de módulos que oferecem formas alternativas de se trabalhar com estimativas de tempo e custo. Uma das novidades que foram planejadas para a nova versão é a possibilidade de se introduzir informação de dependência entre operações, quando da execução de simulações.

Até a versão anterior, as estimativas de tempo para o conjunto de operações ligado a um dado poço de petróleo eram feitas em total ignorância das possíveis e prováveis relações de dependência entre elas. Os resultados, apesar de aparentemente satisfatórios, escondiam imprecisões potencialmente danosas, sobretudo em um campo de aplicação que envolve grandes somas de valores monetários.

A abordagem escolhida para a modelagem de dependência no âmbito de análise de risco do software baseia-se no uso de cópulas, uma combinação relativamente nova. Por se tratar de um assunto de pesquisa que, conseqüentemente, estaria sujeito à experimentação e modificação constantes, optou-se por construir um sistema a parte, deixando a integração com o sistema E&P Risk para os estágios finais do projeto.

O sistema de experimentação criado foi desenvolvido no sistema operacional Windows XP Professional, rodando sobre plataforma Intel/32-bit. Por questão de familiaridade, o software foi escrito em linguagem C++ e, para facilitar a posterior integração com o E&P Risk, o ambiente de desenvolvimento escolhido foi o mesmo, Borland C++ Builder 6.0.

3.2 Metodologia de Trabalho

Desde o início, este trabalho teve o suporte de uma equipe de dois professores da área de estatística —também participantes do projeto E&P Risk, que havia tomado para si o desafio de estudar a modelagem de dependências através do uso de cópulas. Incidentalmente, tais professores vieram a ser os co-orientadores deste trabalho. É deles o mérito principal pela pesquisa e fundamentação matemática para os algoritmos implementados. Por toda a extensão do projeto, houve bastante interação, desde as reuniões iniciais de planejamento até o dia-a-dia das experimentações e ajustes.

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 poderem ser incorporados ao sistema de experimentos que estava sendo construído.

O que se seguia era uma fase de testes em que a implementação das novas funcionalidades era verificada. Por diversas vezes, isso implicou na criação de pequenos ambientes de testes com interface de linha de comando, que eram passados aos professores.

```
D:\Zunino\Cópula\Testes\debug\PPChi2.exe
Entre com os 3 argumentos da funcao PPChi2 <p> <v> <g>

Onde: <p> -> o valor da area da cauda inferior
      <v> -> parametro de graus de liberdade
      <g> -> logaritmo natural de Gama(0.5 * v)

Entre com '?' para mostrar esta ajuda ou tecle CTRL-C para sair.
p v g: 2,176 5 2,5
PPChi2(p, v, g) = 29,5201232345
p v g: 9,711 8,2 4,1
PPChi2(p, v, g) = 36,3802549493
p v g: _
```

Figura 1: Um dos ambientes de teste de algoritmos

Esses ambientes eram programas escritos separadamente e com o único intuito de testar a implementação de um certo algoritmo. A figura 1 mostra um desses ambientes, criado para testar a implementação de um algoritmo de cálculo de pontos percentuais da distribuição χ^2. Com caráter interativo, os ambientes permitiam que os professores colocassem à prova a implementação, realizando uma série de testes como: validação de domínio, adequação e precisão de resultados, etc.

Alguns algoritmos secundários —a maioria não diretamente relacionada a cópulas— eram retirados da literatura e adaptados para servirem como peças na construção de algoritmos maiores. Alguns deles vinham na forma de pseudo-código; outros encontravam-se escritos em linguagens de programação, incluindo C e FORTRAN. O primeiro contato com a tradicional linguagem de programação científica demandou estudo sobre sua sintaxe e uso, de modo que a conversão para C++ se desse sem alteração semântica.

3.3 Visão Geral do Software

Do ponto de vista do usuário, um dos objetivos traçados inicialmente era de que o software tivesse uma inteface gráfica amigável e que, ao mesmo tempo, oferecesse bastante flexibilidade na configuração de experimentos de simulação. Em virtude de a ferramenta ter sido construída ao longo de um projeto de pesquisa, é natural que ela tenha sofrido alguns ajustes e modificações. De maneira geral, entretanto, a forma final preservou muito do que fora inicialmente projetado.

Por ser baseado na idéia de um *ambiente de experimentos*, o software foi concebido segundo o paradigma de aplicação MDI (*Multiple-Document Interface*). Nesse paradigma, uma aplicação é centrada em um tipo de documento (onde documento é uma designação genérica de um determinado tipo de objeto manipulado por uma aplicação) e é capaz de apresentar e gerenciar vários desses documentos simultaneamente. No contexto da aplicação ora implementada, um documento toma forma de um experimento de simulação. Assim, o paradigma escolhido permite que se trabalhe com vários experimentos simultaneamente.

A interface gráfica foi construída usando o framework VCL (*Visual Component Library*), que acompanha o Borland C++ Builder. O framework VCL encapsula um grande número de objetos e facilidades da API do Windows (e.g. janelas, botões, barras de ferramenta, barras de progresso, etc.), expondo-os através de classes hierarquicamente organizadas. Apesar da palavra "Visual" no título, o framework incorpora também facilidades não visuais (i.e., que não possuem uma parte gráfica associada). Um exemplo disso é a classe `TThread`. Essa classe encapsula a API de *threads* do Windows e facilita bastante a criação de aplicações com mais de uma linha de execução. As simulações realizadas no software ora apresentado rodam no contexto de uma linha paralela de execução, empregando a classe `TThread`, como será detalhado mais adiante, na seção 3.6.6.

Em linha com a filosofia RAD (*Rapid Application Development*), o IDE permite a manipulação gráfica de componentes de interface e oferece uma série de ferramentas que ajudam

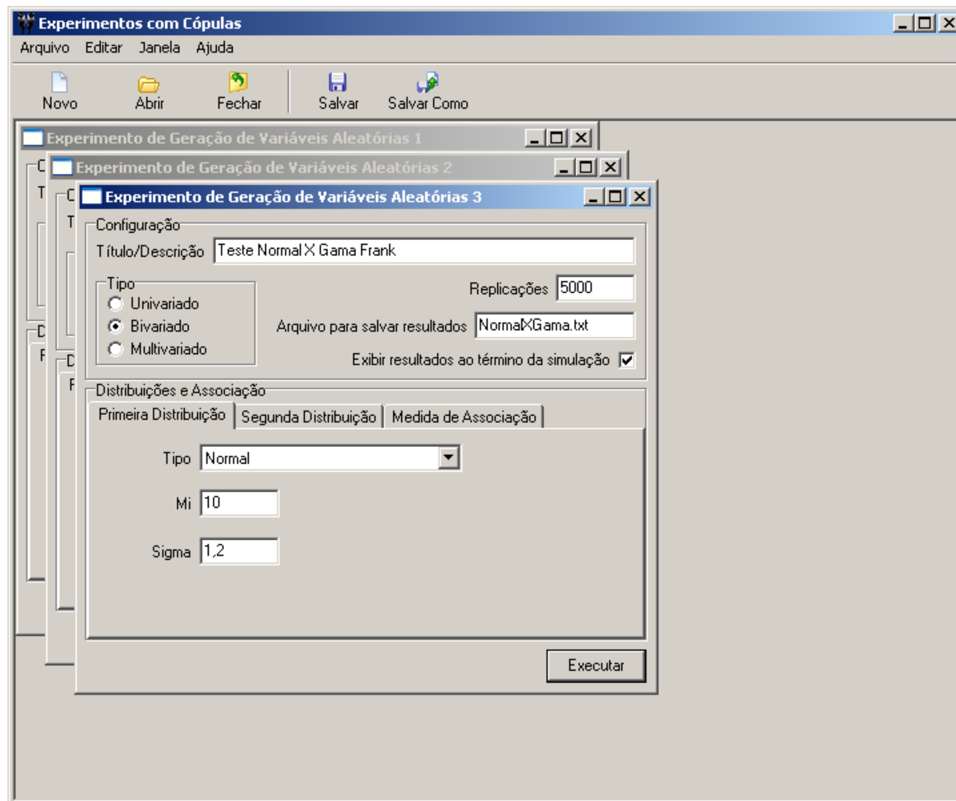


Figura 2: O software e sua Interface MDI

no desenho dos formulários. O ambiente de desenvolvimento oferece também facilidades para a depuração de erros, recurso que foi fundamental na descoberta de alguns erros numéricos insidiosos (e.g. aqueles causados por perda de significação).

O software foi originalmente concebido para rodar em plataforma Windows. Isso não implicou, todavia, na ausência de cuidado em separar responsabilidades e minimizar acoplamentos. De fato, todo o código que implementa a "lógica do negócio" (algoritmos de geração de números aleatórios, cópulas, etc.) foi escrito segundo as normas da linguagem C++ padrão. Assim sendo, seu reuso em uma plataforma diferente consistiria simplesmente em copiar os arquivos de código-fonte e usar um compilador C++ padrão que gerasse código para aquela plataforma. A única parte do software que necessitaria ser reescrita com código específico para a nova plataforma-destino é aquela concernente à interface gráfica. De maneira mais estrita, seria também preciso substituir a facilidade de *thread* usada na execução de experimentos. Em outras palavras, todo e qualquer uso da VCL —biblioteca específica para Windows— teria que ser substituído por uma alternativa equivalente na plataforma-destino.

3.4 Instalação

Como descrito anteriormente, a metodologia de trabalho tinha caráter iterativo, onde versões novas de teste eram freqüentemente criadas para serem validadas pelos professores. Inicialmente, a distribuição era feita de uma maneira pouco eficiente: o sistema lhes era enviado, por email, num arquivo compactado, e não havia processo amigável de instalação. Era necessário realizar alguns passos manuais: descompactação, cópia dos arquivos (executável do projeto, bibliotecas de tempo de execução, criação de atalhos, etc.).

Eventualmente, resolveu-se adotar uma forma um pouco mais madura e eficiente de se distribuir as novas versões. Foi aí que se passou a usar o NSIS (*Nullsoft Scriptable Install System*), um sistema gerador de programas de instalação. O NSIS foi inicialmente criado pela Nullsoft para distribuir seu famoso software tocador de arquivos multimídia, Winamp. Eventualmente, a empresa resolveu torná-lo sofware livre e o projeto é hoje gerenciado pela comunidade, tendo como base o portal SourceForge (http://nsis.sourceforge.net/). Desde então, o NSIS têm sido adotado por uma série de projetos, como uma alternativa viável a produtos comerciais como o InstallShield, da empresa Macrovision. A listagem 3.1 mostra um trecho do script de instalação NSIS criado para este projeto.

Listagem 3.1: Trecho do script NSI de instalação do software

```
; Instalador Experimentos com Copulas
; Ney Andre de Mello Zunino
; Novembro 2005
;————————————————————————————————

; Atributos do instalador

Name        "Experimentos com Copulas"
OutFile     "Instalar Experimentos com Copulas.exe"
InstallDir  "$PROGRAMFILES\Experimentos com Copulas"

;————————————————————————————————

; Paginas

Page        directory
Page        components
Page        instfiles
UninstPage  uninstConfirm
UninstPage  instfiles

;————————————————————————————————

; Secoes

Section "Programa"
  SectionIn RO

  SetOutPath $INSTDIR

  File Experimentos.exe
;   File BorlndMM.dll
;   File CC3260MT.dll
;   File STLPMT45.dll

  WriteUninstaller $INSTDIR\Desinstalar.exe
SectionEnd
```

```
Section "Atalhos no Menu de Programas"
  CreateDirectory "$SMPROGRAMS\Experimentos com Copulas"
  CreateShortCut "$SMPROGRAMS\Experimentos com Copulas\Experimentos com Copulas.lnk"
                 "$INSTDIR\Experimentos.exe"
  CreateShortCut "$SMPROGRAMS\Experimentos com Copulas\Desinstalar.lnk"
                 "$INSTDIR\Desinstalar.exe"
SectionEnd
```

O emprego do NSIS possibilitou um processo de distribuição do software muito mais simples e amigável. A cada nova versão, o script de instalação era reprocessado e o instalador resultante disponibilizado em um URL padrão, no servidor do PLab. Os professores podiam então baixar o programa e instalá-lo através de uma interface familiar e amigável, exemplificada na figura 3.

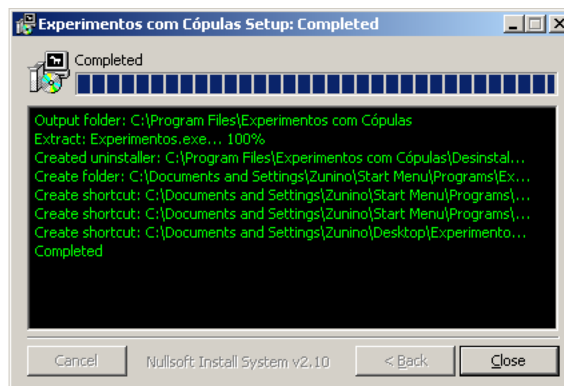


Figura 3: O instalador NSIS em ação

3.5 Funcionamento

O usuário começa seu trabalho criando um novo experimento de simulação. Isso pode ser feito através da opção ‘Novo Experimento’ no menu ‘Arquivo’ ou do botão ‘Novo’ na barra de tarefas. A criação de um novo experimento faz com que seja apresentada uma janela onde o experimento pode ser configurado e executado, como mostra a figura 4. Configurar um experimento implica em definir uma (para experimentos univariados) ou duas (para experimentos bivariados) distribuições que farão parte dele e, para o caso de duas distribuições, especificar a medida de dependência entre elas.

Após configurado o experimento, o botão ‘Executar’ dá início à simulação. É então apresentada ao usuário uma pequena janela que o permite acompanhar o progresso da simulação e, opicionalmente, interrompê-la. De acordo com a escolha feita pelo usuário quando da configuração do experimento, a aplicação ‘Bloco de Notas’ é automaticamente aberta ao término da simulação, para mostrar os resultados.

3.5.1 Configurando a Simulação

A janela de experimento é, sem dúvida, a parte da interface gráfica mais importante do sistema. Seu design procura atender aos requisitos inicialmente propostos de facilidade de uso e flexibilidade na configuração de experimentos. Sua forma final deriva de uma série de discussões com os professores acerca de como melhor colher informações sobre as distribuições marginais e sobre a medida de dependência. A tabela 2 apresenta uma breve descrição das seções e campos da janela de experimento mostrada na figura 4.

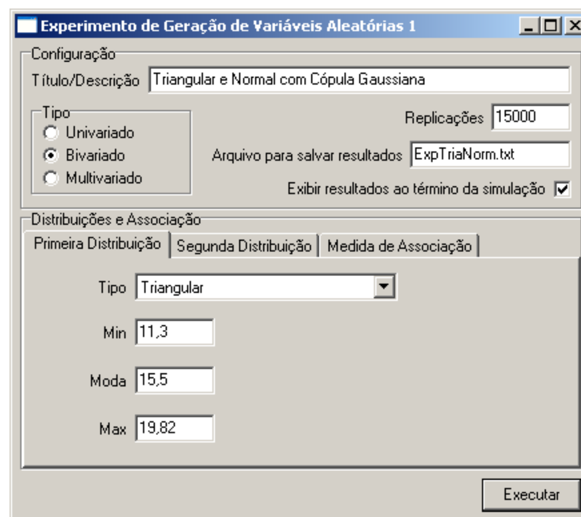


Figura 4: Configurando uma distribuição do experimento

3.5.1.1 Configurando as Distribuições

Na fase de planejamento da interface com o usuário, considerou-se algumas alternativas para a forma como se daria a configuração das distribuições usadas em experimentos.

Uma possível abordagem seria trabalhar com formas textuais das funções, que seriam submetidas a um analisador léxico/sintático para validação. Por exemplo, para se configurar uma distribuição *normal* com média 8,0 e desvio padrão 1,6, o usuário deveria entrar com `normal(8,0;1,6)`. Para uma distribuição *exponencial* com média 11,5, a entrada deveria ser, por exemplo, `exponencial(11,5)`.

Essa alternativa tem a vantagem de permitir uma simplificação da interface gráfica, sobretudo quando se considera o suporte a experimentos multivariados, uma vez que basta usar um único campo de texto para cada distribuição, independentemente de qual seja e de quantos parâmetros necessite. Por outro lado, aumenta-se a carga sobre o usuário, a medida que ele precisa

Tabela 2: Campos e seções da janela de experimento

Campo/seção	**Descrição**
Título	Texto descritivo que identifica o experimento.
Tipo	Número de variáveis aleatórias ou distribuições de probabilidade a serem definidas e usadas no experimento. Somente as opções ‘univariado’ e ‘bivariado’ podem ser usadas no momento. A opção ‘multivariado’ já possui implementação dos algoritmos, mas resta adaptar a interface gráfica para dar suporte a esse tipo de experimento.
Replicações	Número de valores, pares ou tuplas que o simulador deve gerar.
Arquivo	Nome do arquivo onde os resultados da simulação serão salvos.
Exibir resultados	Indica se o arquivo de resultados deve ser automaticamente aberto ao término da simulação.
Distribuições	Este painel permite a entrada dos parâmetros de cada distribuição selecionada.

conhecer, além dos parâmetros desejados, a sintaxe para especificação das distribuições. Como conseqüência natural, há um aumento na incidência de erros de validação.

Uma outra abordagem consiste no uso de campos individuais para a escolha da distribuição e coleta dos parâmetros de cada uma delas. Essa foi a abordagem empregada no sistema. Os campos e seus rótulos dependem da distribuição selecionada pelo usuário e adequam-se automaticamente. Por exemplo, se o usuário tiver selecionado uma distribuição *gama*, serão apresentados campos propriamente rotulados para seus parâmetros *alfa* e *beta*. No exemplo da figura 4, o usuário selecionou uma distribuição *triangular*, o que fez com que fossem oferecidos campos para coletar seus três parâmetros: *mínimo*, *moda* e *máximo*.

3.5.1.2 Configurando a Medida de Dependência

Para experimentos bivariados, há uma segunda fase de configuração. É onde se especifica a medida de dependência entre as duas distribuições selecionadas. A figura 5 mostra um exemplo de configuração de dependência.

No exemplo da figura 5, o usuário optou por modelar a dependência através de uma cópula gaussiana. Para a medida, foi escolhido o coeficiente τ de Kendall, aqui indicando uma corre-

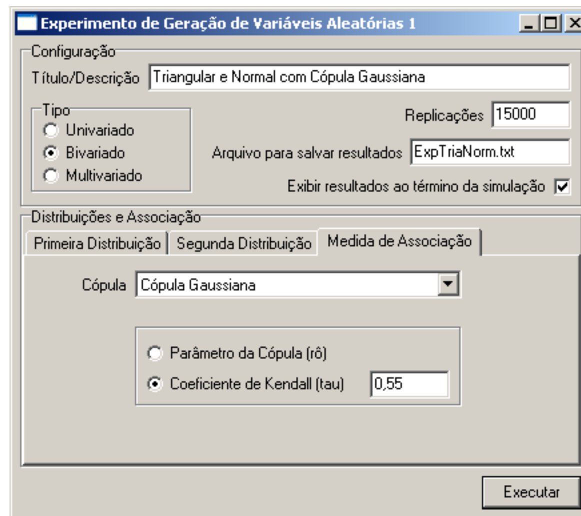


Figura 5: Definindo a medida de dependência entre as distribuições

lação positiva de $0{,}55$ entre as duas distribuições do experimento. A tabela 3 dá mais detalhes sobre os campos disponíveis no painel de configuração de dependência, rotulado 'Medida de Associação'.

Tabela 3: Campos da configuração de medida de dependência

Campo/seção	Descrição
Cópula	Permite que o usuário escolha qual cópula empregar no experimento. A versão atual do sistema implementa 4 cópulas: cópula de Clayton, cópula de Frank, cópula Gaussiana e cópula *t*. Dependendo da cópula escolhida, há restrições quanto aos valores que se pode usar na medida de dependência (e.g. a cópula de Clayton aceita apenas valores positivos para o coeficiente τ de Kendall. Detalhes na seção 2.2.
Medida de dependência	A caixa de medida de dependência permite que o usuário quantifique a dependência entre as distribuições do experimento. Para todas os tipos de cópula, há a opção de se usar o coeficiente τ de Kendall. Para tipos específicos de cópulas, a outra medida de correlação pode variar entre o coeficiente ρ de Pearson ou o parâmetro θ da cópula. Esses coeficientes e seus significados estão descritos na seção 2.2, sobre cópulas.

3.5.2 Executando a Simulação

Depois de ter concluído a configuração do experimento, o usuário pode executá-lo através do botão 'Executar', localizado na parte inferior da janela. Opcionalmente, ele pode decidir salvar o experimento. Salvar um experimento implica no registro em arquivo de todos os parâmetros de configuração, de modo que ele possa ser restaurado futuramente ou até compartilhado com outro usuário do sistema.

Dependendo do número de replicações indicado pelo usuário e da capacidade do equipamento onde o sistema estiver sendo executado, o processo de simulação pode ser demorado. Por esse motivo, o sistema inclui uma janela de progresso (figura 6) que é apresentada ao usuário enquanto a simulação é executada.

Figura 6: Janela com indicação de progresso da simulação

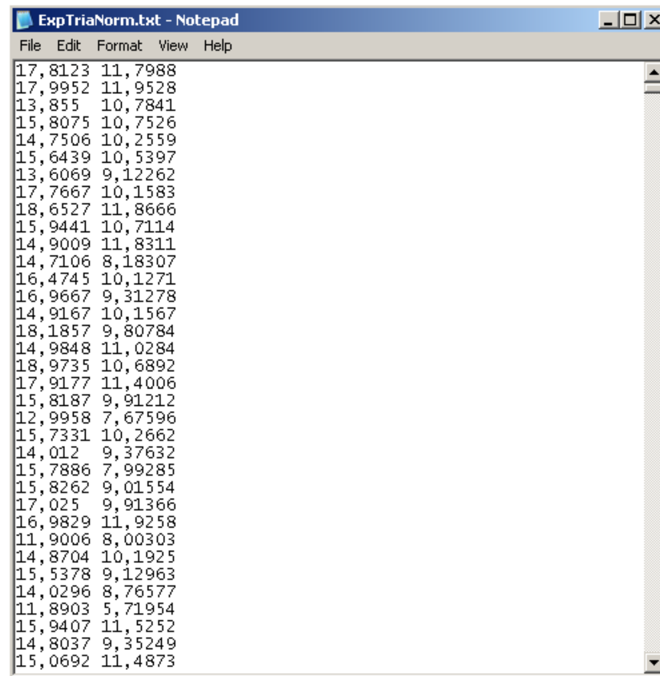
As vantagens dessa facilidade são duas:

- oferecer ao usuário uma noção de tempo para a conclusão da simulação; e
- permitir que o usuário interrompa o processo, caso deseje.

Em virtude de o processo de simulação ser de natureza ligada à CPU (*CPU-bound*), é necessário um tratamento especial para que a interface gráfica do programa possa se manter alerta à interação do usuário enquanto o processo de cálculo está em andamento. Caso contrário, o usuário teria a impressão de que o software deixou de responder ou "travou". A técnica para lidar com esse problema envolve o uso de *threads*, conforme descrito na seção 3.6.6 adiante.

Os resultados produzidos pelo simulador são salvos em um arquivo texto especificado pelo usuário na configuração da simulação. A figura 7 mostra um exemplo de um desses arquivos. A configuração da simulação permite também ao usuário escolher se ele deseja que o arquivo de resultados seja automaticamente aberto ao término do processo.

Os dados no arquivo de resultados aparecem dispostos em colunas separadas por caracteres de tabulação, onde cada linha contém os resultados de uma replicação. Isso permite que os valores sejam facilmente copiados e colados em outras suítes de estatística, como Microsoft




```
17,8123 11,7988
17,9952 11,9528
13,855  10,7841
15,8075 10,7526
14,7506 10,2559
15,6439 10,5397
13,6069 9,12262
17,7667 10,1583
18,6527 11,8666
15,9441 10,7114
14,9009 11,8311
14,7106 8,18307
16,4745 10,1271
16,9667 9,31278
14,9167 10,1567
18,1857 9,80784
14,9848 11,0284
18,9735 10,6892
17,9177 11,4006
15,8187 9,91212
12,9958 7,67596
15,7331 10,2662
14,012  9,37632
15,7886 7,99285
15,8262 9,01554
17,025  9,91366
16,9829 11,9258
11,9006 8,00303
14,8704 10,1925
15,5378 9,12963
14,0296 8,76577
11,8903 5,71954
15,9407 11,5252
14,8037 9,35249
15,0692 11,4873
```

Figura 7: Resultados de simulação com cópula

Excel, MiniTab, Statistica, etc. Obviamente, caso o experimento executado seja univariado, haverá apenas uma coluna no arquivo de resultados.

Com relação ao caractere usado como separador decimal nos resultados, é importante mencionar que a implementação do software foi feita de modo a adaptar-se às configurações regionais do ambiente onde a simulação estiver sendo executada. Em outras palavras, em um ambiente com configurações regionais americanas, o caractere de ponto (‘.’) seria usado; para configurações regionais brasileiras, o caractere de vírgula (‘,’) é empregado. Este é o caso ilustrado na figura 7.

3.6 Aspectos de Implementação

Nesta seção, serão apresentadas e discutidas algumas partes relevantes da implementação do sistema. A discussão procurará mostrar como os conceitos fundamentais do trabalho foram materializados em código e como se dá a interação entre os elementos.

3.6.1 Distribuições de Probabilidade

A caixa de ferramentas de um simulador Monte Carlo deve sempre contar com a implementação de algumas distribuições teóricas de probabilidade. No sistema de experimentos, foram

implementadas 11 delas. Tais implementações fazem uso de um gerador de números aleatórios uniformes e de expressões matemáticas específicas que produzem valores em uma distribuição desejada.

Listagem 3.2: Classe abstrata Distribuicao

```
class Distribuicao : public ReferenceCountable
{
public:
   static long double numeroAleatorio();
   static void reiniciarSemente(bool opcao) { mReiniciarSemente = opcao; }
public:
   virtual ~Distribuicao() {}
   virtual long double proximo(long double u = numeroAleatorio()) const = 0;
protected:
   Distribuicao();
private:
   static bool mReiniciarSemente;
};
```

Todas as 11 distribuições implementadas no sistema derivam de uma base comum, a classe abstrata `Distribuicao`, cuja definição está contida na listagem 3.2. O diagrama A.1, nos anexos, apresenta a hierarquia das classes de distribuições. A primeira coisa a se perceber é que a classe deriva de `ReferenceCountable`, o que lhe concede capacidades especiais com respeito à gerência de memória. Mais detalhes sobre esse aspecto serão apresentados na seção 3.6.5.

A interface pública da classe `Distribuicao` está dividida em duas partes: uma estática e outra não-estática. Os membros de uma interface estática, sejam eles variáveis ou funções, podem ser usados sem a necessidade de se ter um exemplar da classe. Em linguagens como Smalltalk, variáveis-membro e funções-membro estáticos são conhecidos como variáveis de classe e métodos de classe, respectivamente. Outra característica importante é que, ao contrário de variáveis-membro não-estáticas, variáveis-membro estáticas são compartilhadas por todos os exemplares da classe.

A função-membro estática relevante é `numeroAleatorio`. Ela implementa um gerador de números aleatórios uniformes ($u \in (0;1)$) que serve de base para toda a geração de números das distribuições teóricas de probabilidade.

Na interface pública não-estática tem-se a função-membro `proximo`. Essa é uma função declarada como virtual pura, ou seja, as classes que herdarem de `Distribuicao` devem fornecer uma implementação específica que gere e retorne um próximo valor segundo a distribuição de probabilidade representada. As listagens 3.3 e 3.4 mostram a implementação de `proximo` para as classes `DistribuicaoUniforme` e `DistribuicaoWeibull`, respectivamente. O argumento opcional `u` especifica um número aleatório uniforme a ser usado pela implementação da distribuição teórica de probabilidade. Quando não fornecido pelo usuário da

classe, a função-membro estática `numeroAleatorio` é usada para gerar um novo número.

Listagem 3.3: Implementação da função-membro `proximo` para distribuição uniforme

```
long double DistribuicaoUniforme::proximo(long double u) const
{
    return (mMin + (mMax - mMin) * u);
}
```

Listagem 3.4: Implementação da função-membro `proximo` para distribuição Weibull

```
long double DistribuicaoWeibull::proximo(long double u) const
{
    return mBeta * pow(-log(1.0 - u), 1.0 / mAlfa);
}
```

3.6.2 Cópulas

Para a introdução do conceito de cópulas, foi definida uma nova hierarquia de classes, que inicia-se com a classe abstrata `Copula`, mostrada na listagem 3.5.

Listagem 3.5: Classe abstrata Copula

```
class Copula : public ReferenceCountable
{
public:
    typedef std::pair<long double, long double> Par;
public:
    enum TipoParametro {TAU, TETA, RO};
public:
    virtual ~Copula() {}
    virtual Par proximo(long double u = Distribuicao::numeroAleatorio()) const = 0;
    virtual long double ultimoV() const = 0;
protected:
    Copula();
};
```

Observa-se que a classe `Copula` também deriva da classe `ReferenceCountable`. Como mencionado anteriormente, a razão disso é permitir que exemplares de cópulas possam gozar de um gerenciamento especial de memória, explicado mais adiante.

Por trabalhar com geração bivariada, a classe `Copula` introduz um novo tipo `Par`, que representa pares de valores reais. Esse tipo é usado por todas as classes que derivam de `Copula` e em outros trechos de código que lidam com cópulas e os resultados produzidos por elas. Um exemplo de uso do tipo `Par` está na função-membro `proximo`. Enquanto as distribuições retornam ou geram valores reais escalares, as cópulas geram pares de valores reais.

Outro tipo introduzido pela classe `Copula` —na verdade, uma enumeração— é chamado `TipoParametro`. Como o nome diz, esse tipo serve para identificar a natureza do parâmetro fornecido para uma determinada cópula. Conforme as descrições presentes na seção 2.2.3 sobre medida de dependência, diferentes parâmetros podem ser usados para essa especificação. Ao

construir uma cópula, o usuário deve não só fornecer um valor real para servir de medida de dependência, mas também especificar que tipo de parâmetro aquele valor representa.

A outra função-membro disponível na interface pública não-estática da classe é `ultimoV`. Ela deve ser implementada nas classes derivadas para retornar o valor do último número `v` usado no algoritmo da cópula. Essa facilidade foi incluída para permitir a implementação das cópulas encadeadas, no caso multivariado, que necessita desse valor.

Listagem 3.6: Classe CopulaFrank

```
class CopulaFrank : public Copula
{
public:
   CopulaFrank(RefCountPtr<Distribuicao> dist1,
               RefCountPtr<Distribuicao> dist2,
               long double parametro,
               TipoParametro tipoParametro = TAU);
   virtual ~CopulaFrank();
   virtual Par proximo(long double u = Distribuicao::numeroAleatorio()) const;
   virtual long double ultimoV() const;
private:
   RefCountPtr<Distribuicao> mDist1;
   RefCountPtr<Distribuicao> mDist2;
   long double mTeta;
   mutable long double mUltimoV;
};
```

A listagem 3.6 mostra a definição da classe `CopulaFrank`, uma das quatro classes que derivam de `Copula`. Aqui, pode-se enfim apreciar a estrutura de uma classe de cópula propriamente dita, a partir da qual se pode criar exemplares. O diagrama A.2, nos anexos, apresenta a hierarquia completa das classes de cópulas implementadas.

Os parâmetros do construtor fornecem os elementos e informação necessários para que o algoritmo da cópula possa gerar pares de valores correlacionados. Os dois primeiros parâmetros identificam as distribuições marginais. Eles são passados para o construtor encapsulados em um objeto do tipo `RefCountPtr`, que é um tipo de *smart pointer* criado para facilitar a gerência de memória e que também será discutido adiante, quando se apresenta as classes de gerência de memória.

O terceiro parâmetro é um número real que representa a medida de dependência entre as duas distribuições marginais. O parâmetro final é opcional e indica qual a natureza ou tipo da medida de dependência usada. Se não for indicado, assume-se que o usuário esteja trabalhando com o coeficiente τ de Kendall.

Ao se olhar para as variáveis-membro privadas da classe, observa-se que o parâmetro da cópula é armazenado como θ (variável-membro `mTeta`). Isso implica que, para esta cópula, uma conversão é feita caso o usuário realmente deseje entrar com o parâmetro τ de Kendall. Essas conversões são feitas através de expressões de equivalência entre os parâmetros, apresentadas como parte da descrição das cópulas, no capítulo de conceitos fundamentais.

Uma outra variável-membro que se tornou necessária é aquela que armazena o valor do último número `v` obtido no algoritmo da cópula. Esse valor é atualizado cada vez que um novo par é gerado e pode ser lido a qualquer momento através da função-membro `ultimoV`, já comentada.

Será apresentado, agora, um exemplo de implementação do algoritmo de cópulas. A listagem 3.7 contém o código usado pela classe `CopulaFrank` para gerar um par de valores correlacionado.

Listagem 3.7: Implementação do algoritmo de geração para a cópula de Frank

```
Copula::Par CopulaFrank::proximo(long double u) const
{
    long double w = Distribuicao::numeroAleatorio();
    long double a = exp(-mTeta * u);
    long double v = (-1.0 / mTeta) * log((w * exp(-mTeta) + a - w * a) / (a - w * a + w));

    mUltimoV = v;

    long double x = mDist1->proximo(u);
    long double y = mDist2->proximo(v);

    return Copula::Par(x, y);
}
```

Antes de comentar a implementação, é interessante apresentar um diagrama que ilustra o esquema para a geração bivariada com cópulas. A figura 8 dá uma visão simplificada do fluxo de informação no algoritmo e permite traçar um paralelo com o código da listagem 3.7.

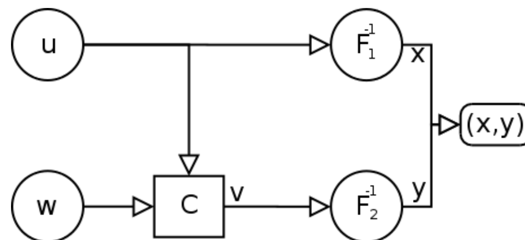


Figura 8: Esquema para geração de cópulas bivariadas

O diagrama mostra que o algoritmo começa com dois números aleatórios uniformes independentes u e w. Na implementação, tem-se o parâmetro `u` que, para o caso de cópulas bivariadas, é um número aleatório uniforme. Tem-se também uma variável local `w` que é iniciada com um valor aleatório uniforme. Seguindo o fluxo do diagrama, vê-se a aplicação de u e w em uma função-cópula, tendo como resultado um valor v. Na implementação, isso é visto na linha que declara e define a variável `v`. Analisando a expressão, identifica-se o uso dos valores de `mTeta`, `u` (através da variável local `a`) e `w`.

Na seqüência do diagrama, os valores de u e v são aplicados, respectivamente, às funções das distribuições marginais F_1^{-1} e F_2^{-1}, para produzir os componentes x e y do par que se está

gerando. Na versão de código, tem-se duas invocações da função-membro virtual `proximo` da classe `Distribuicao`, uma para cada distribuição marginal da cópula. Os resultados são armazenados em variáveis locais `x` e `y` que são usadas para construir o par a ser retornado. Embora não explicitado no diagrama, pode-se perceber, no código, que o valor de `v` é armazenado na variável-membro `mUltimoV` logo após sua expressão ter sido calculada.

3.6.3 Experimentos

Outra hierarquia importante no sistema é a de experimentos. A listagem 3.8 apresenta a definição da classe abstrata `Experimento`, que serve de base para os diferentes tipos de experimento disponíveis no sistema.

Listagem 3.8: Classe abstrata Experimento

```
class Experimento
{
public:
   virtual ~Experimento() {}
   virtual void geraProximo(std::ostream& saida) = 0;
   int replicacoes() const { return mReplicacoes; }
   const std::string& nomeArquivoSaida() const { return mNomeArquivoSaida; }
protected:
   Experimento(int replicacoes, const std::string& nomeArquivoSaida);
private:
   int mReplicacoes;
   std::string mNomeArquivoSaida; // Arquivo com os resultados do experimento
};
```

Basicamente, todo experimento tem associado seu número de replicações (variável-membro `mReplicacoes`) e um nome de arquivo onde escrever seus resultados (variável-membro `mNomeArquivoSaida`). A hierarquia completa de experimentos pode ser vista no diagrama A.3, no capítulo de anexos.

Da interface pública da classe, a função-membro mais importante é a `geraProximo`. Ela é a responsável por produzir os resultados para cada replicação do experimento. É interessante observar que, ao contrário das funções-membro equivalentes nas hierarquias de distribuições e cópulas, esta não retorna qualquer valor (tipo de retorno declarado como `void`). A razão para isso é que a hierarquia inclui experimentos univariados, bivariados e multivariados e o código que executa experimentos e registra seus resultados deve ser executado sem conhecimento do tipo específico de experimento em questão. Para tornar isso possível, foi utilizada uma estratégia que buscou uniformizar a interface da função de geração, usando um *stream* como meio comum para transferência de informação. Esse é o papel do parâmetro `saida`, passado por referência à função-membro `geraProximo`.

As listagens 3.9 e 3.10 visam ilustrar essa situação, apresentando, respectivamente, as implementações da função-membro de geração para as classes `ExperimentoUnivariado` e

`ExperimentoBivariadoCopula`.

Listagem 3.9: Implementação de `geraProximo` para experimento univariado

```
void ExperimentoUnivariado::geraProximo(std::ostream& saida)
{
    saida << mDistribuicao->proximo() << '\n';
}
```

Listagem 3.10: Implementação de `geraProximo` para experimento bivariado

```
void ExperimentoBivariadoCopula::geraProximo(std::ostream& saida)
{
    Par par = mCopula->proximo();
    saida << par.first << '\t' << par.second << '\n';
}
```

Olhando para as listagens, pode-se entender como o uso do *stream* viabiliza, de fato, a uniformização da interface da função para os diferentes casos de experimentos. No exemplo da listagem 3.9, apenas um valor é inserido no *stream*; já na listagem 3.10, tem-se a inserção de dois valores, separados por um caractere de tabulação. O importante é que o código responsável por executar experimentos pode fazê-lo sem qualquer preocupação ou conhecimento sobre o tipo real de experimento sendo simulado.

3.6.4 Validação de Domínio

Ao transformar algoritmos matemáticos em código de computador, não se deixou de lado cuidados básicos que conferem maior robustez ao software como um todo. Um exemplo disso é a verificação de adequação de parâmetros ao domínio de uma função. A listagem 3.11 traz o código do construtor da classe `DistribuicaoGama`, onde tem-se exemplificada a verificação de validade de domínio dos parâmetros. Neste caso, tanto `alfa` quanto `beta` devem ser maiores que zero. Infrações a essas regras são notificadas através do uso de exceções.

Listagem 3.11: Construtor da classe DistribuicaoGama

```
DistribuicaoGama::DistribuicaoGama(long double alfa, long double beta)
:   mAlfa(alfa), mBeta(beta)
{
    if (!((alfa > 0) && (beta > 0)))
        throw std::domain_error("Os parametros 'alfa' e 'beta' da "
                                "distribuicao Gama devem ser maiores "
                                "que zero.");
}
```

3.6.5 Gerência de Memória

Um dos aspectos mais intricados e que requerem muita atenção quando se está lidando com um ambiente não-gerenciado de programação é a gerência de memória. Por "ambiente

não-gerenciado", entende-se um ambiente onde cabe ao programador determinar, não só o momento de criação, mas também o de destruição de objetos. Em outras palavras, a alocação e liberação de blocos de memória é também responsabilidade do programador. Contraste-se isso com ambientes gerenciados, onde esse encargo fica por conta de uma entidade conhecida como coletor de lixo. O programador fica livre dessa preocupação e o risco de ocorrerem problemas sérios por acessos inválidos à memória são bastante reduzidos.

Como não poderia deixar de ser, todo esse benefício não acontece sem algum custo. Neste caso, o custo está normalmente relacionado a questões de desempenho. De um lado, há a necessidade de se manter uma espécie de monitor que faz um acompanhamento das alocações e das referências a um dado objeto, de modo a poder determinar quando esse objeto pode ser encerrado. De outro, o fato de essa gerência se dar de forma automática, poda o programador quanto a fazer um controle mais fino do uso de memória de seu programa.

Existem muitos mitos e conceitos equivocados acerca do uso de uma linguagem de programação como C++, na qual foi escrita o software ora descrito. Geralmente, assume-se que o uso de C++ implica na necessidade de se lidar com alocação de memória, ponteiros, etc. A verdade é que, em um sistema C++ moderno, o uso de ponteiros crus e de alocação explícita de memória dificilmente é necessário e, quando o é, limita-se às partes mais internas e de baixo-nível da aplicação. Além disso, há ainda a possibilidade de se definir o que se chama de *smart pointers*, entidades que facilitam bastante a gerência de memória, reduzindo o risco de erros.

Smart pointers aparecem em várias formas, mas não cabe aqui apresentar uma discussão exaustiva sobre o assunto. O foco será em tentar descrever a estratégia usada para a gerência de memória no sistema de experimentos. A essência está em se criar uma classe que encapsule um ponteiro cru e que fique responsável por ele. Para o usuário, a manipulação do ponteiro se dará da mesma forma, mas terá sido tirada dele a responsabilidade pela liberação da memória alocada.

A questão é: como essa classe-envólucro pode determinar quando a memória apontada pelo ponteiro pode ser liberada? Uma estratégia comum para isso é a *contagem de referência*. Contar referências significa manter um registro de todos os apontamentos que existem para um determinado objeto no sistema. Assim, a medida que esses apontamentos crescem, cresce junto a contagem de referência; por outro lado, a medida que diminuem os apontamentos, diminui também a contagem de referência, até que a contagem chegue a zero. Quando não há mais apontamentos, ou seja, o objeto não está mais sendo referenciado, o *smart pointer* pode encerrá-lo e devolver a memória alocada ao sistema.

A listagem 3.12 inclui a definição da classe `ReferenceCountable`, que serve como

base para todas as classes a cujos exemplares se deseja permitir gerenciamento automático de memória através de *smart pointers*. A classe que implementa o *smart pointer* em si —na verdade, um *template* de classe— chama-se `RefCountPtr` e está disposta na listagem 3.13.

Listagem 3.12: Classe ReferenceCountable

```
class ReferenceCountable
{
public:
  void incrementReferenceCount() { count++; }
  void decrementReferenceCount() { count--; }
  unsigned int referenceCount() const { return count; }
protected:
  ReferenceCountable() : count(0) {}
private:
  unsigned int count;
};
```

Como pode-se perceber, a classe `ReferenceCountable` é bastante simples. Ela é uma classe desenhada para servir como base para outras classes e, por isso, seu construtor é protegido. À construção, sua variável-membro `count` é iniciada com zero. Em sua interface pública, a classe abstrata oferece facilidades para: incrementar seu contador; decrementar seu contador; e retornar o valor do contador.

Listagem 3.13: Classe RefCountPtr<T>

```
template <class T>
class RefCountPtr
{
public:
  RefCountPtr()
  : object(0)
  {
  }

  RefCountPtr(T* pointer)
  : object(pointer)
  {
    if (object)
      object->incrementReferenceCount();
  }

  RefCountPtr(const RefCountPtr& original)
  : object(original.object)
  {
    if (object)
      object->incrementReferenceCount();
  }

  ~RefCountPtr()
  {
    if (object)
    {
      object->decrementReferenceCount();
      if (object->referenceCount() == 0)
        delete object;
    }
  }

  RefCountPtr& operator=(const RefCountPtr& original)
  {
    return operator=(original.object);
  }

  RefCountPtr& operator=(T* pointer)
  {
    // Handle self-assignment and assignmet from equivalent smart pointer
```

```
    if (this->object != pointer)
    {
      if (object)
      {
        object->decrementReferenceCount();
        if (object->referenceCount() == 0)
          delete object;
      }
      object = pointer;
      if (pointer)
        pointer->incrementReferenceCount();
    }
    return *this;
  }

  T* operator->()
  {
    return object;
  }

  const T* operator->() const
  {
    return object;
  }

  T& operator*()
  {
    return *object;
  }

  const T& operator*() const
  {
    return *object;
  }

  T* pointer()
  {
    return object;
  }

  const T* pointer() const
  {
    return object;
  }

  operator void*() const
  {
    return object;
  }
private:
  T* object;
};
```

Voltando a atenção agora à classe `RefCountPtr` ou, mais propriamente, ao *template* de classe `RefCountPtr<T>`, destaca-se alguns aspectos relevantes. Em primeiro lugar, a razão pela qual essa classe foi projetada como um template é para dar-lhe a flexibilidade de trabalhar com ponteiros para diferentes tipos de objetos. Assim, ao invés de implementá-la em termos de um tipo específico de objeto, usa-se o parâmetro `T`.

Observa-se em sua definição, que a classe possui apenas um membro de dados; o membro chama-se `object` e é do tipo “ponteiro para `T`”. Esse é o ponteiro cru para o bloco de memória alocado dinamicamente e que se deseja gerenciar.

São implementados três diferentes construtores para a classe (sobrecarga de construtores). O primeiro deles é um construtor-padrão, que simplesmente inicia a variável-membro `object`

com o valor zero. Isso pode ser entendido como um *smart pointer* que foi criado, mas que ainda não está gerenciando qualquer objeto.

O segundo construtor recebe como argumento um ponteiro cru para um objeto do tipo `T`. O ponteiro cru recebido é o que será gerenciado pela classe. Em sua lista de iniciação, o construtor inicia a variável-membro `object` com o valor do ponteiro recebido como argumento. O código no corpo do construtor primeiro certifica-se de que o ponteiro passado não é nulo e então invoca sobre ele a função-membro `incrementReferenceCount` da classe `ReferenceCountable`, da qual o objeto apontado deve descender.

O terceiro e último construtor é um construtor de cópia da classe. Ele permite que um *smart pointer* `RefCountPtr<T>` seja construído a partir de outro já existente. As ações deste construtor são idênticas a do anterior, exceto pelo fato de que a variável-membro `object` é iniciada não diretamente com o argumento recebido, mas sim com o ponteiro contido dentro dele.

Será analisado, agora, um dos aspectos mais importantes dessa classe: o destrutor. Ele corresponde a um dos pontos onde pode ocorrer o encerramento efetivo do objeto apontado. A primeira coisa que o código faz é verificar se a variável-membro `object` não é nula. Caso não seja, a função-membro `decrementReferenceCount` da classe `ReferenceCountable` é invocada sobre ele. Em seguida, usa-se a função-membro `referenceCount` da mesma classe para verificar se a contagem chegou a zero. Se tiver chegado, o operador `delete` é finalmente usado para destruir o objeto apontado e liberar a memória por ele ocupada.

Outro elemento importante na implementação da classe `RefCountPtr<T>` é o operador de atribuição. São dois na verdade, pois há uma versão que aceita como argumento um outro objeto do tipo `RefCountPtr<T>` e uma versão sobrecarregada que aceita um ponteiro cru para um objeto do tipo `T`. Um cuidado especial que se deve ter na implementação dos operadores de atribuição é verificar se não está ocorrendo auto-atribuição. Nesse caso, a função não deve realizar qualquer atualização de contagem e simplesmente retornar uma referência para o próprio objeto. Caso contrário, deve-se proceder de maneira similar ao código do destrutor, ou seja, decrementar a contagem para o ponteiro original, caso seja não-nulo. A contagem para o ponteiro novo, por sua vez, deve ser incrementada, caso o ponteiro seja não-nulo.

As demais funções-membro servem apenas para permitir que o *smart pointer* possa ser usado como um ponteiro comum, com relação à sintaxe. Em outras palavras, elas permitem que o usuário trabalhe com os operadores comumente associados a ponteiros, e.g. `*`, `->`.

3.6.6 *Threads*

Encerra-se a discussão sobre aspectos de implementação abordando o uso de *threads* no sistema. Conforme fora descrito na seção 3.3, o software utilizou a classe `TThread`, parte do framework VCL, da Borland, para possibilitar a execução de simulações em uma linha paralela. A classe `TThread` encapsula a API de *threads* do Windows, facilitando seu uso. Para criar uma *thread*, deve-se definir uma nova classe derivada de `TThread`. No sistema de experimentos, isso foi feito com a classe `ThreadExperimento`, que representa uma linha de execução paralela onde uma simulação deve rodar.

Por ser tratada como um objeto, uma *thread* precisa ser criada antes de ser usada. O sistema de experimentos cria um exemplar da classe `ThreadExperimento` assim que o usuário comanda o início da simulação. A listagem 3.14 mostra o trecho de código relevante, executado quando uma simulação é disparada.

Listagem 3.14: Código de execução da simulação

```
Experimento* experimento = constroiExperimento();

if (experimento)
{
    ThreadExperimento* threadExperimento = new ThreadExperimento(true, experimento);
    TFormProgresso* janelaProgresso = new TFormProgresso(this, threadExperimento);
    threadExperimento->janelaProgresso(janelaProgresso);

    janelaProgresso->ShowModal();
    .
    .
    .
```

O trecho de código inicia com a chamada de uma função que irá "construir" o experimento a ser executado. A função `constroiExperimento` é responsável por colher as informações de configuração entradas pelo usuário e construir o tipo de experimento adequado (univariado ou bivariado). Na seqüência, é feita uma verificação de ocorrência de erros durante a criação do experimento. Na ausência de erros, os seguintes passos são executados:

1. É criado um objeto do tipo `ThreadExperimento`, representando a linha de execução paralela, onde o código da simulação será rodado. O objeto que representa o experimento, construído anteriormente, é passado como parâmetro para o construtor do objeto-*thread*. O primeiro parâmetro passado para o construtor da classe `ThreadExperimento` faz com que o objeto-*thread* seja criado em modo *suspenso*, ou seja, sua execução deverá ser explicitamente comandada.

2. É criado um objeto do tipo `TFormProgresso`, que representa a janela usada para monitorar o avanço da simulação e permitir a interrupção do processo. A esse objeto, é passado um ponteiro para o objeto-*thread* criado no passo anterior.

3. Em contrapartida, o objeto-*thread* é informado de qual janela de progresso estará sendo usada para monitorar o avanço de seu processo.

4. Por fim, a função-membro `ShowModal` é invocada sobre a janela de progresso, fazendo com que ela seja mostrada e detenha o foco de entrada.

O processo é efetivamente iniciado quando ocorre o evento `Show` da janela de progresso. A função-membro que trata esse evento é mostrada na listagem 3.15.

Listagem 3.15: Tratador do evento `Show` da janela de progresso

```
void __fastcall TFormProgresso::FormShow(TObject *)
{
   mThreadExperimento->Resume();
}
```

Tudo que o tratador do evento `Show` faz é invocar a função-membro `Resume` sobre o objeto-*thread*. Isso irá, efetivamente, fazer com que outra função-membro, `Execute`, seja invocada, dando início ao processo de simulação. A listagem 3.16 mostra o laço central que comanda a simulação, dentro da função-membro `Execute`.

Listagem 3.16: Laço Principal de Execução da Simulação

```
   int intervaloAtualizacao = mExperimento->replicacoes() / 50;
   for (int i = 0; !Terminated && i < mExperimento->replicacoes(); ++i)
   {
      mExperimento->geraProximo(arquivoSaida);
      mProgresso = i;
      if (mProgresso % intervaloAtualizacao == 0)
      {
         mMensagemProgresso = AnsiString("Replicacao ") + AnsiString(mProgresso) + " de "
                              + AnsiString(mExperimento->replicacoes());
         Synchronize(atualizaJanelaProgresso);
      }
   }
```

A idéia do laço principal da simulação é executar cada uma de suas replicações e notificar a janela de progresso do andamento do processo, para que ela possa refletí-lo para o usuário. Todavia, essa notificação não deve ser feita a cada replicação, pois o custo seria muito alto e o desempenho sofreria bastante. Por isso, antes de entrar no laço, executa-se um simples cálculo para encontrar um intervalo de atualização. Esse número indicará de quantas em quantas replicações deverá ser feita uma notificação à interface gráfica.

O corpo do laço começa invocando a função-membro `geraProximo` (descrita na seção 3.6.3) para executar um passo ou replicação da simulação. Em seguida, é feito um teste para verificar se já foi decorrido um número suficiente de passos para que ocorra atualização da interface de monitoramento. Em caso verdadeiro, uma mensagem de texto com informação do progresso é criada e armazenada em uma variável-membro `mMensagemProgresso`. Em

seguida, é feita uma invocação um pouco curiosa. A chamada `Synchronize` tem uma peculiaridade. Ela faz com que o código passado a ela (na verdade, um ponteiro para uma função) seja executado no contexto da *thread* principal, aquela que controla a interface gráfica do sistema.

O processo continua dessa forma até que o número total de replicações do experimento tenha sido executado ou a propriedade `Terminated` da classe `Thread` tenha se tornado verdadeira. Isso acontece, por exemplo, caso o usuário decida interromper a simulação, clicando sobre o botão “Cancelar” na janela de progresso.

4 *Conclusão*

O presente trabalho abordou o desenvolvimento de uma ferramenta experimental de simulação que teve como diferencial a introdução da noção de dependência entre operações. A modelagem de dependência se deu através da ferramenta matemática *cópula* que, aliada à simulação Monte Carlo, permitiu a produção de estimativas correlacionadas.

O desenvolvimento paralelo de um ambiente de experimentação permitiu que as idéias e algoritmos fossem postos a prova antes de serem aproveitados no sistema alvo de análise de risco. A propósito, o emprego de cópulas em análise de risco mostrou-se uma área ainda pouco explorada.

As cópulas são bastante flexíveis quanto às funções de distribuição marginais, mesmo que alguma adaptação tenha sido necessária para um ou outro caso (e.g. distribuição gama). Algumas técnicas de modelagem de dependência encontradas na literatura são bastante restritivas quanto às distribuições marginais que se pode associar. Outro fator interessante no uso de cópulas é a possibilidade de se quantificar a dependência através do coeficiente τ de Kendall, que oferece uma noção intuitiva de medida de dependência, negativa ou positiva.

Finalmente, a experiência de lidar com conceitos e algoritmos matemáticos e a necessidade de transpô-los para o mundo digital foi bastante enriquecedora. Obviamente, foram aplicados na implementação do trabalho, muitos dos conhecimentos adquiridos durante a formação em Ciência da Computação.

4.1 Trabalhos Futuros

4.1.1 Interface Gráfica para Cópulas Multivariadas

Em seu estado atual, o software oferece a possibilidade de construir dois tipos de experimentos: univariados e bivariados. A opção de experimentos multivariados está presente na interface gráfica mas, ao tentar selecioná-la, o usuário recebe uma mensagem informando que

a implementação não foi concluída.

Como mencionado em outras seções deste trabalho, a pesquisa que lhe deu sustentação levou também à definição de um algoritmo de cópulas multivariadas. Tal algoritmo foi implementado, testado e validado. Sua implementação faz uso das mesmas classes de cópulas e distribuições usadas para o caso bivariado.

De fato, apesar de inacessível através da interface gráfica do sistema, a capacidade de se trabalhar com cópulas multivariadas já existe e está materializada em uma nova classe na hierarquia de experimentos, chamada `ExperimentoMultivariadoCopula`. Naturalmente, para ser útil ao usuário final, essa nova classe precisa ser complementada por uma interface gráfica amigável que facilite a interação na preparação de experimentos multivariados.

A interface gráfica atual do sistema permite a definição/configuração de duas distribuições marginais e da medida de dependência entre elas. É necessário estender o conceito para um número n de distribuições e mais os $n - 1$ valores de medida de dependência entre elas.

Uma abordagem possível consistiria no uso de um botão (e.g. na barra de ferramentas) que permitisse a adição incremental de novas distribuições. A cada distribuição adicionada, deveria ser também disponibilizado um campo onde o usuário pudesse especificar a medida de dependência entre a distribuição sendo adicionada e aquela imediatamente anterior. Para tornar o encadeamento mais intuitivo, alguma identificação da distribuição anterior poderia ser mostrada ao lado do campo de medida de dependência.

Uma outra abordagem —na verdade, uma derivação provavelmente mais interessante da abordagem anterior— seria abandonar a distinção explícita da natureza dos experimentos (univariado, bivariado ou multivariado) e empregar uma forma única de construção. Com um procedimento uniforme, o usuário preocuparia-se apenas em adicionar quantas distribuições desejasse e caberia ao sistema "entender" qual tipo de experimento construir.

4.1.2 Aderência com Cópulas

Os resultados obtidos com este trabalho permitiram a apreciação de correlação entre variáveis aleatórias geradas (simuladas), com base em restrições de dependência previamente fornecidas. Uma das questões que podem naturalmente seguir é: dada uma amostra de dados multivariada, onde supostamente ocorra interdependência entre esses dados, seria possível inferir, através de análise e de maneira automática, os parâmetros de correlação?

Em outras palavras, tendo-se percorrido o caminho da geração de valores interdependentes

através do uso de cópulas, como traçar o caminho inverso e chegar a um modelo de cópulas multivariadas a partir de uma massa de dados? Estratégias válidas para se abordar o problema podem incluir o estudo e adaptação de técnicas existentes de aderência, como testes qui-quadrado, Kolmogorov-Smirnov, etc.

4.1.3 Exportação para Excel

Atualmente, os resultados produzidos pelas simulações são direcionados para arquivos-texto, formatados como TSV (*Tab-Separated Values*). Dados nesse formato são facilmente aproveitáveis no Excel, seja via processo de importação ou mesmo através da área de transferência (copiar e colar) do Windows.

Por outro lado, seria interessante que o sistema oferecesse ao usuário a possibilidade de controlar o formato dos dados de saída. Poderia haver uma opção que produzisse resultados no mesmo formato atualmente suportado e uma nova que levasse à criação automática de uma planilha Excel, preferencialmente tirando proveito dos recursos oferecidos pelo formato, incluindo títulos, totais, médias, etc.

A implementação de tal funcionalidade é de complexidade relativamente baixa. A grande maioria dos aplicativos da Microsoft, incluindo todos os que fazem parte de sua suíte de ferramentas de escritório, incluem componentes que permitem que uma aplicação externa manipule seus objetos. Esses componentes —baseados em COM (*Component Object Model*), outra tecnologia da Microsoft— expõem uma série de objetos da aplicação e, através de um processo conhecido como *automação*, permitem que ela seja controlada a partir de uma outra aplicação.

Referências

ANJOS, U. U. et al. Modelando dependências via cópulas. 2004.

EMBRECHTS, P.; LINDSKOG, F.; MCNEIL, A. Modeling dependence with copulas and applications for risk management. 2001. Disponível em: <http://www.risklab.ch/ftp/papers/DependenceWithCopulas.pdf>.

FILHO, P. J. de F. *Introdução à Modelagem e Simulação de Sistemas*. [S.l.]: Visual Books, 2001.

LAW, A. M.; KELTON, W. D. *Simulation Modeling and Analysis*. Second edition. [S.l.]: McGraw-Hill, 1991.

MILLER, A. *Applied Statistics - AS91: percentage points of the* χ^2 *distribution*. [S.l.: s.n.], 1975. 386-387 p.

NELSEN, R. B. *An Introduction to Copulas*. [S.l.]: Springer-Verlag, 1999.

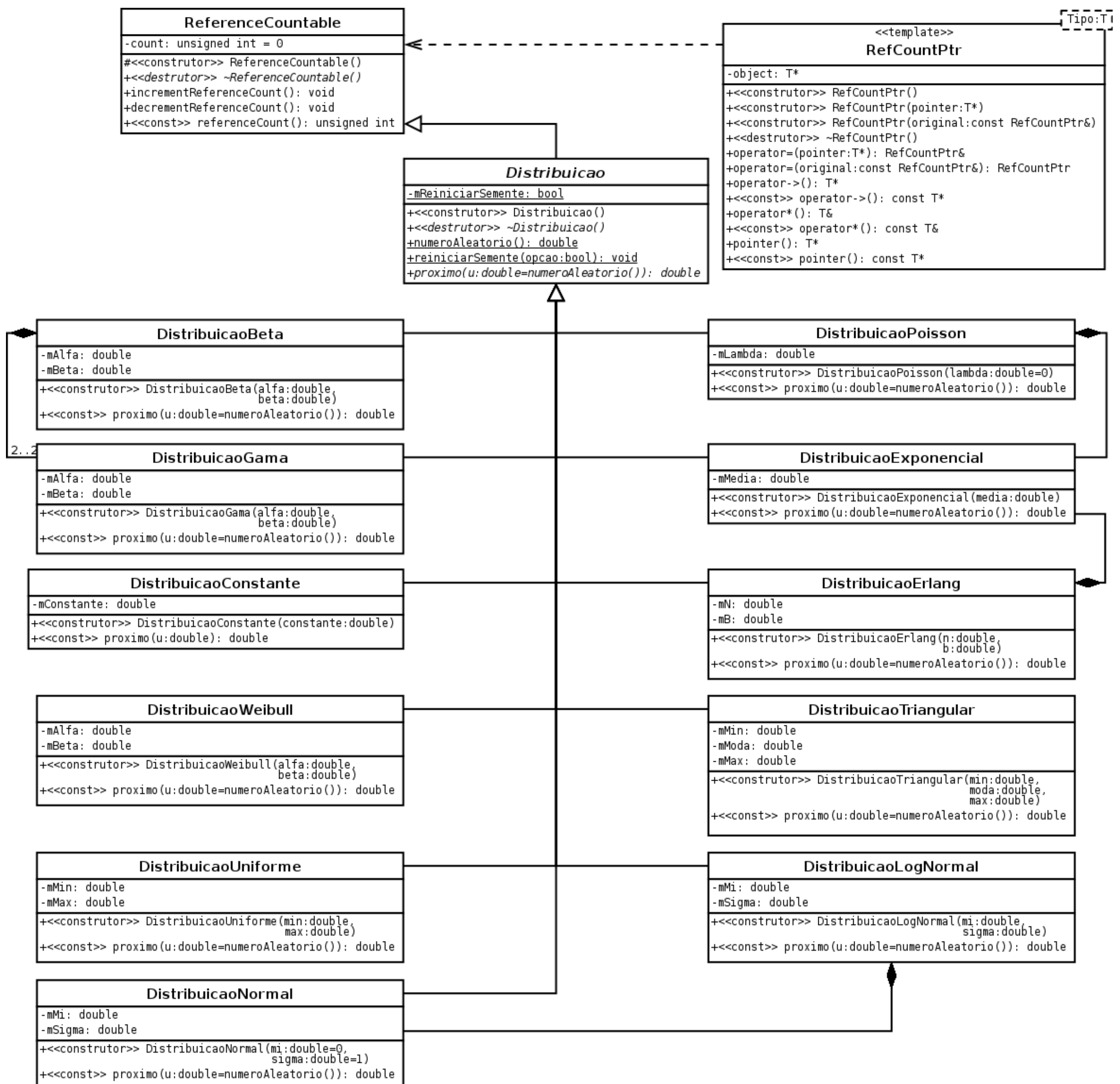
PRESS, W. H. et al. *Numerical Recipes in C*. Second edition. [S.l.]: Cambridge University Press, 1992.

SONG, P. X.-K. Multivariate dispersion models generated from gaussian copula. *Scandinavian Journal of Statistics*, v. 27, p. 305–330, 2000.

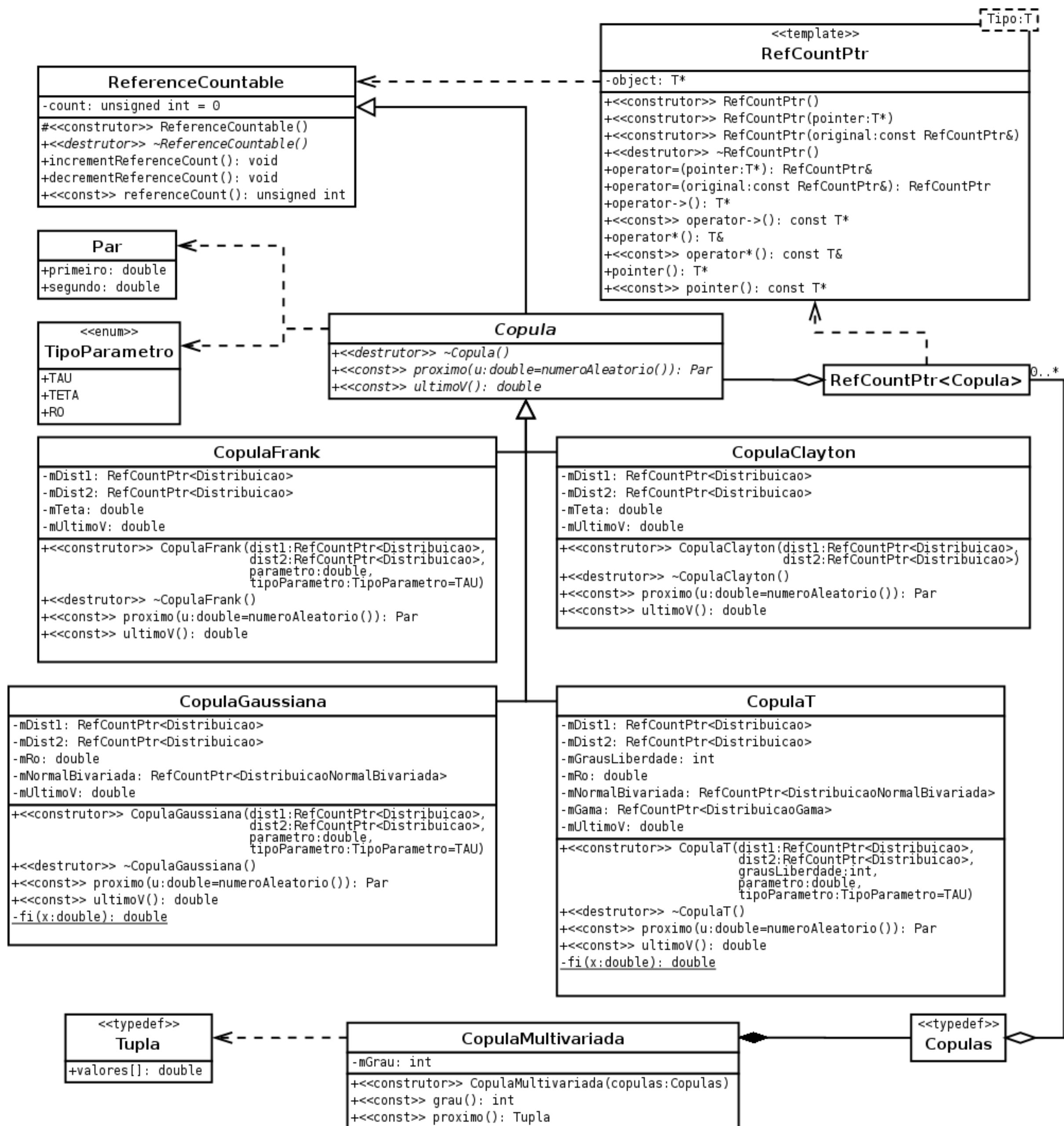
VOSE, D. *Risk Analysis: A Quantitative Guide*. Second edition. [S.l.]: John Wiley & Sons Ltd., 2001.

APÊNDICE A – Diagramas de Classes

A.1 Diagrama de classes de distribuições de probabilidade



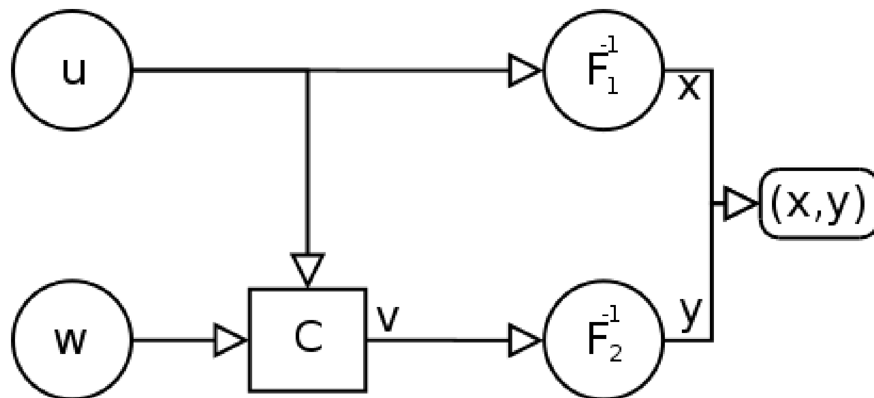
A.2 Diagrama de classes de cópulas



A.3 Diagrama de classes de experimentos

APÊNDICE B – Esquemas dos Algoritmos de Cópulas

B.1 Esquema de Cópula Bivariada



B.2 Esquema de Cópula Multivariada