Projeto e Programação do Mapa Auto-Organizável para Apoio à Análise Espacial

Marcos A.S. da Silva[1,2] , Antônio M.V. Monteiro[1]
[1] *Instituto Nacional de Pesquisas Espaciais*
Divisão de Processamento de Imagens
São José dos Campos, SP
miguel @dpi.inpe.br
[2] *Laboratório de Geotecnologias Aplicadas*
Embrapa Tabuleiros Costeiros, Aracaju, SE
aurelio@cpatc.embrapa.br

Resumo

Este trabalho apresenta a biblioteca de classes SOMLib, que implementa os algoritmos e encapsula os dados relativos ao Mapa Auto-Organizável de Kohonen. Também mostra o sistema CΛSΛ *(**C**onnectionist **A**pproach for **S**patial **A**nalysis of **A**rea), um sistema para ambiente gráfico para análise espacial de área baseado na SOMLib.*

Abstract

This paper presents the SOMLib library that implements algorithms and encapsulate data related to the Kohonen Self-organizing Map. It also shows the CΛSΛ *(**C**onnectionist **A**pproach for **S**patial **A**nalysis of **A**rea) a SOMLib based GUI interface for spatial analysis of area.*

1. Introdução e Motivação

Para o caso de análise de dados espaciais multivariados é necessário que os resultados gerados a partir do Mapa Auto-Organizável [7], possam ser visualizados graficamente através de um SIG. Para que isto seja possível, sem a necessidade de importação/exportação de arquivos, é necessária a conexão do algoritmo SOM à biblioteca de acesso ao banco de dados geográficos, Terralib, desenvolvida no INPE/DPI, [2]. A Terralib é uma biblioteca de classes voltada para o desenvolvimento de sistemas de informação geográfica otimizados. A Terralib foi desenvolvida com a linguagem de programação C++ através da aplicação de modernas técnicas de programação como padrões de projeto [6], programação genérica [13], biblioteca padrão (STL) [10] e programação multi-paradigma [4].

Embora o algoritmo padrão de treinamento da rede SOM seja conceitualmente simples, sua implementação requer uma série de cuidados. Kohonen, [7], afirma que a maioria das implementações não se preocupa com os detalhes do processo de construção do algoritmo. Ciente deste problema a equipe de pesquisas em Mapas Auto-Organizáveis da universidade da Filândia desenvolveu dois pacotes de software que implementam a rede SOM. O SOM PAK, desenvolvido em C [8] e o SOM ToolBox, desenvolvido em MatLab [14]. Ambos possuem código fonte aberto e são gratuitos. Estes pacotes possuem características importantes como a confiabilidade, disponibilidade do código fonte e funcionalidade.

Todavia, após a análise dos pacotes SOM PAK e SOM ToolBox verificou-se que ambos demandariam um esforço muito grande de conexão com a biblioteca Terralib, uma vez que estes pacotes foram desenvolvidos em linguagens distintas da C++ e não usam conceitos de programação moderna, o que acarreta sérias dificuldades de manutenção. Portanto, apesar das vantagens em termos de confiabilidade e funcionalidade decidiu-se desenvolver um novo código para o algoritmo SOM.

Outros pacotes foram analisados, mas não atendiam simultaneamente os requisitos de disponibilidade do código fonte, confiabilidade e manutenabilidade, [5], [12], [11]. O pacote SOM ToolBox foi usado neste projeto como mecanismo de comparação e teste do algoritmo SOM desenvolvido.

Como colocado por [9] o desenvolvimento de qualquer simulador neural exige preocupações nas áreas de depuração do código, processamento de alto desempenho, com ou sem paralelização da implementação, e projeto. Este trabalho concentrou-se na elaboração do projeto de

implementação baseado no paradigma de Orientação a Objetos, [6]. Os pacotes SOM PAK e ToolBox auxiliaram na depuração do código projetado e implementado. O projeto consistiu no desenho e construção de uma biblioteca de classes, ***SOMLib***, que implementa algoritmos e encapsulam dados relativos ao uso da rede SOM para a análise exploratória de dados multivariados.

2. Mapa Auto-Organizável

O Mapa Auto-Organizável de Kohonen é uma rede neural de aprendizagem competitiva organizada em duas camadas, [7]. A primeira camada representa o vetor dos dados de entrada, x_k, a segunda corresponde a uma grade de neurônios, geralmente bidimensional, totalmente conectada aos componentes do vetor de entrada. Cada neurônio possui um vetor de código associado, w_j.

O processo de aprendizagem consiste de três fases. Na primeira fase, competitiva, cada padrão de entrada é apresentado a todos os neurônios para que aquele mais próximo do padrão apresentado seja o vencedor. Na segunda fase, cooperativa, é definida a vizinhança relativa ao neurônio vencedor. Na terceira fase, adaptativa, os vetores de código do neurônio vencedor e dos seus vizinhos serão alterados segundo algum critério de atualização.

Após a aprendizagem os vetores de código do SOM corresponderam a uma aproximação não-linear dos padrões de entrada. O SOM também preserva a formação topológica dos padrões de entrada, ou seja, padrões próximos no conjunto amostral estarão relacionados a neurônios próximos na grade neural.

O SOM pode variar em algoritmos de aprendizagem, estrutura topológica da grade, função de vizinhança, parametrização inicial etc.

3. Projeto e Implementação

Segundo [6], qualquer pacote SOM deve apresentar um conjunto mínimo de características. Este pacote deve permitir que a grade da rede possa ter qualquer dimensão e arranjo, disposição hexagonal e retangular, aprendizagem em lote e seqüencial, função de vizinhança gaussiana e bolha, iniciação linear, tratamento de dados ausentes, algoritmos de visualização e cálculo dos erros de quantização e topológico.

Como observado na seção 2 a rede neural SOM pode variar de diferentes formas. Pode-se ter redes de dimensões variadas, cada uma com um formato diferente da grade de neurônios, funções de vizinhança distintas etc. Representando este conjunto de variações num diagrama de classes (Figura 1) pode-se observar a proliferação de classes. Observe que todas as características relativas à grade estão encapsuladas nas classes de topologia (2D, 3D ...).

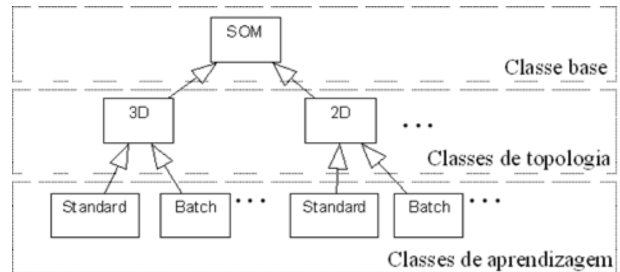


Figura 1. **Diagrama de classes para representação das famílias de Mapas Auto-Organizáveis**

As classes foram agrupadas em três categorias. Classe base (SOM), classes de topologia (2D, 3D ...) e classes de aprendizagem (Standard, Batch ...). Implementar a biblioteca com base nesta estrutura de classes não configura uma boa idéia, pois além da duplicação de classes observa-se um forte acoplamento entre as classes de topologia e de aprendizagem.

Para resolver esta questão dividiu-se o problema em dois: projeto e implementação das classes de aprendizagem e de topologia. Para o problema relativo às classes de aprendizagem tem-se que, a depender do contexto ou necessidade do usuário, deve ser possível variar entre os vários algoritmos de aprendizagem implementados. Pode-se, então, usar o padrão *Strategy* para resolver este problema. O padrão *Strategy* define uma família de algoritmos, encapsula cada um, e os faz interoperáveis, [6]. O diagrama da Figura 2 mostra que uma classe abstrata foi criada (*LearningAlgorithm*), pela qual as classes de aprendizagem serão derivadas. O trecho de código seguinte ilustra a implementação da classe base SOM, considerando a estrutura do diagrama de classes (Figura 2 (b)).

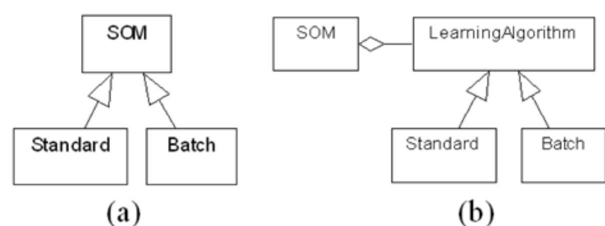


Figura 2. **Diagrama de classe: (a) representação da classe base e das classes de aprendizagem; (b) nova estrutura do diagrama (a) baseada no padrão *Strategy***

Como demonstrado através da Figura 3 as questões de topologia e aprendizagem estão misturadas de forma que a adição de mais uma classe de topologia implica na reconstrução das classes de aprendizagem relacionadas com a mesma. Este problema foi solucionado com o uso do padrão de projeto *Bridge*. Este padrão desacopla uma abstração de sua implementação de forma que ambas possam variar independentemente, [6]. Assim, criou-se

mais uma classe abstrata, *TopologyImp*. Será a partir dessa classe que se originaram as classes concretas de topologia. A Figura 4 mostra a nova estrutura do relacionamento entre as classes de topologia e as classes de aprendizagem.

```
class SOM {
public:
 SOM( Params par )
 {
  switch (par.type) {
   case BATCH: _learningAlgorithm = new Batch;
   case STAND: _learningAlgorithm = new
Standard;
  }
}
   ~SOM();
protected:
   LearningAlgorithm * _learningAlgorithm

   LearningAlgorithm * getLearningAlgorithm() {
       return _learningAlgorithm;  }
public:
  void Learning() {
     getLearningAlgorithm()->Learning( netParams
);}
```

Com esta nova estrutura uma mesma implementação de uma classe de topologia pode servir a mais de uma classe de aprendizagem sem a necessidade de duplicação de código. Em seguida tem-se mais um trecho de código da classe abstrata *LearningAlgorithm*.

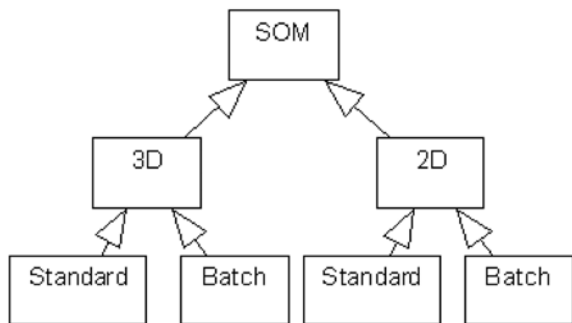


Figura 3. **Diagrama de classes: aqui se observa o alto acoplamento entre as classes de topologia e de aprendizagem.**

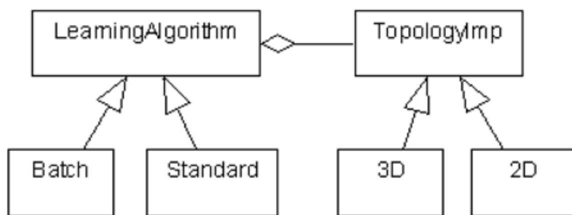


Figura 4. **Através do padrão *Bridge* separou-se os detalhes de topologia e aprendizagem.**

```
class LearningAlgorithm {
public:
  virtual int Learning( Params& par ) = 0;

  TopologyImp * getTopology(){return _topology;
};

       ~LearningAlgorithm();

protected:
  LearningAlgorithm(const TopolParams& par) {
       switch( par.type ) {
        case TWO : _topology = new TwoD;
        case THREE : _topology = new ThreeD; }
  }
private:
  TopologyImp * _topology;
}
```

Optou-se pela mesma estratégia de implementação para os dois padrões de projeto usados, mas observe que ambas foram motivadas por razões distintas. A Figura 5 mostra a configuração final do diagrama de classes após o uso dos padrões. Esta estrutura permitirá uma maior manutenabilidade e possibilidade de reuso de código para a biblioteca SOMLib.

Na implementação das classes base SOM e da classe abstrata *LearningAlgorithm* percebe-se que cada uma deve decidir qual objeto criar de acordo com os parâmetros passados no construtor de cada classe. Após a passagem de parâmetro a cláusula *switch* definirá qual objeto construir. Embora seja um método válido, cria a necessidade de se alterar todas as classes que contenham este tipo de cláusula toda vez que uma nova classe de aprendizagem ou de topologia seja implementada. Para este caso usou-se o padrão de projeto *Abstract Factory*. Este padrão estrutural provê uma interface para criação de famílias de objetos sem especificar as respectivas classes concretas. Usou-se uma implementação específica deste padrão, [3]. Nesta implementação o autor empregou a programação genérica para definir um *Factory* genérico cuja função é construir qualquer classe concreta, de um conjunto pré-definido, dispensando o uso de cláuslas do tipo *if..then* e *switch*.

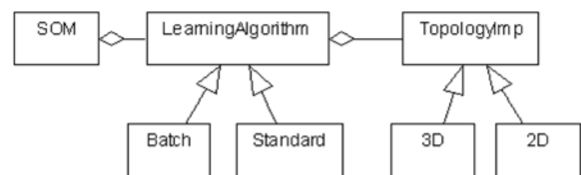


Figura 5. **Diagrama de classe final.**

A Figura 6 mostra como ficou a estrutura de classes do diagrama da Figura 5 após o uso do padrão Abstract Factory. Note que para cada classe de aprendizagem do diagrama (Figura 5) foi criada uma classe construtora, *LearningFactory*, *StandardFactory* e *BatchFactory*. A

função das classe concretas *StandardFactory* e *BatchFactory* é a de implementar a função *build* da classe *Factory*. O mesmo método foi aplicado no diagrama de classe da Figura 4.

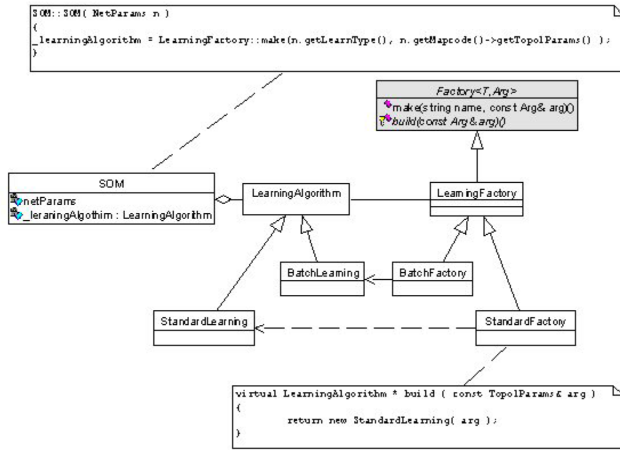


Figura 6. **Representação do uso do padrão *Abstract Factory* sobre o diagrama de classes da Figura 5.**

```
class LearningFactory : public Factory
<LearningAlgorithm,TopolParams> {

public:
LearningFactory(const string& name): Factory <
LearningAlgorithm, TopolParams> (name) {}
}

class BatchLearning : public LearningAlgorithm {
public:
    BatchLearning(const TopolParams&
s):LearningAlgorithm(s) {};

    int Learning ( NetParams& net );
}

class BatchFactory : public LearningFactory {
public:
   BatchFactory( const string& name ) :
LearningFactory( name ) {};

   virtual LearningAlgorithm * build ( const
TopolParams& arg )
       {  return new BatchLearning( arg );  } }
```

A Figura 7 mostra a estrutura de classes para implementação das rotinas de leitura e gravação dos dados, *SOMData*, que alimentarão a rede neural. Optou-se por criar uma classe concreta, *SOMDataCadastre*, para isolar completamente os dados dos detalhes de armazenamento. Assim, a classe *SOMData* transfere todas as responsabilidades de gerenciamento dos dados para a classe *SOMDataCadastre*. Como há varias formas de armazenamento de dados, usou-se o padrão *Strategy* de forma a facilitar o processo de implementação de novos algoritmos de acesso. Assim, surge a classe abstrata de interface, *ISOMDataRepository*, e as classes concretas derivadas desta e que implementam os métodos de acesso aos dados, *RepositorySOMDataFile*, sistemas de arquivos, e *RepositorySOMDataTerralib*, banco de dados formato Terralib.

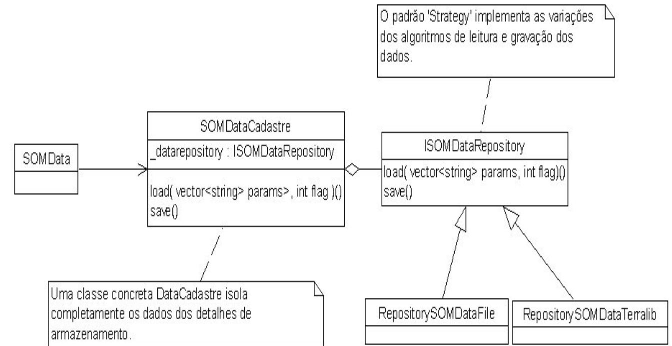


Figura 7. **Representação da estrutura de classes relativas aos dados e algoritmo de leitura e gravação dos dados de entrada da rede neural.**

4. Funcionalidades da SOMLib

No projeto SOMLib implementou-se os algoritmos de aprendizagem em lote e seqüencial; as funções de vizinhança gaussiana, bolha, gaussiana cortada e EP; grade com arranjo hexagonal e retangular; cálculo do erro de quantização e topológico; iniciação por interpolação simples e linear; grade bidimensional.

5. Avaliação da Biblioteca

Para avaliação da SOMLib usou-se dois conjuntos de dados da base [1]: Íris e Wine. As análises de separabilidade das classes e comparação com os resultados gerados pelo SOM ToolBox validaram a biblioteca para estes casos.

6. Uso da SOMLib

Abaixo se tem um exemplo, em C++, do uso da SOMLib. Neste exemplo os padrões são lidos a partir de um arquivo de dados, “dados.pat”. Após a leitura um SOM com valores *default* é criado, bidimensional com aprendizagem em lote. Em seguida os parâmetros da rede são ajustados: dimensão 20x20, disposição hexagonal da grade de neurônios, função de vizinhança gaussiana, raio inicial iguala 15, iniciação linear, 2000 épocas de treinamento. As funções de iniciação, `InitMapcode()`, e de aprendizagem, `Learning()`, são então chamadas. Finalmente, os vetores de código da rede treinada serão gravados no arquivo “mapa_treinado.cod”.

```
void main() {
vector<string> params;
RepositorySOMDataFile repD;
SOMDataCadastre cadD( repD );
SOMData * data = new SOMData;
Params.push_back("dados.pat");

SOM * mysom = new SOM;

mysom->setData( data );
mysom->getMapcode()->setNumVar( d );
mysom->getMapcode()->setDimensions(0,20);
mysom->getMapcode()->setDimensions(1,20);
mysom->getMapcode()->setLattice( "hexa");
mysom->getMapcode()->setNeighborType("gaussian")
mysom->getMapcode()->CreateCodebook( 20*20, d );

mysom->setInitNeighbor( 15 );
mysom->setInitType( LINEAR );
mysom->setNumIteractions( 2000 );

mysom->InitMapcode();
mysom->Learning();

RepositoryMapcodeFile repM;
MapcodeCadastre cadM( repM );
CadM.save(mysom->netParams.getMapcode(),
"mapa_treinado.cód"); }
```

7. O sistema CΛSΛ

A fim de tornar possível a produção visual dos resultados obtidos pelo SOM quanto ao processamento de dados geográficos foi desenvolvido o sistema CΛSΛ - (*Connectionist Approach for Spatial Analysis of Area*) .

O sistema CΛSΛ foi construído sobre as bibliotecas SOMLib e Terralib. No anexo I são mostradas algumas telas do sistema e suas funcionalidades.

8. Conclusões

Embora o algoritmo SOM seja sensível aos detalhes de construção os resultados gerados pela SOMLib mostraram-se compatíveis com os gerados pelo SOMTool Box.

A alta coesão e o baixo acoplamento entre as classes da biblioteca só poderão ser verificados ao longo das próximas versões da SOMLib, quando mais funcionalidades e classes serão adicionadas.

O código fonte da SOMLib estará disponível no seguinte endereço eletrônico: www.dpi.inpe.br/~aurelio/somlib/. O sistema CΛSΛ estará disponível em: www.dpi.inpe.br/~aurelio/casaa/.

9. Referências

[1] Blake, C.L. & Merz, C.J. ***UCI repository of machine learning databases***, Department of Information and Computer Science, University of California at Irvine, CA, 1998. Disponível em: http://www.ics.uci.edu/~mlearn/MLRepository.

[2] Câmara, G.; Neves, M.; Monteiro, A.; Souza, R.; Paiva, J. A.; Vinhas, L. SPRING and TerraLib: Integrating Spatial Analysis and GIS. ***Specialist Meeting on Spatial Data Analysis Software Tools***. Santa Barbara, CA, 2002.

[3] Câmara, G.; Vinhas, L.; Souza, R.; Paiva, J.; Monteiro, A.; Carvalho, M.; Raoult, B. Design Patterns in GIS Development: The Terralib Experience. ***III Workshop Brasileiro de Geoinformática***. Rio de Janeiro, 2001.

[4] Coplien, J. ***Multi-paradigm design for C++***. Tese – Vrije Universiteit Brussel, May 2000.

[5] Gahegan, M.; Takatsuka, M.; Wheeler, M.; Hardisty, H. Introducing GeoVISTA Studio: an integrated suite of visualization and computational methods for exploration and knowledge construction in geography. ***Computers, Environment and Urban Systems***, v. 26, p. 267–292, 2002.

[6] Gamma, E.; Helm, R.; Johson, R.; Vlissides, J. ***Design patterns : elements of reusable object-oriented software***. Addison Wesley, 1995.

[7] Kohonen, T. ***Self-organizing maps***. Springer, 1995. Third Edition 2001.

[8] Kohonen, T.; Hynninen, J.; Kangas, J.; Laaksonen, J. ***SOM PAK: the self-organizing map program package***, April 1995. Version 3.1.

[9] Lawrence, S.; Tsoi, A. C.; Giles, C. L. Correctness, E_ciency, Extendability and Maintanability in Neural Network Simulation. ***Proceeding of International Conference on Neural Networks***. IEEE Press, 1996. p. 474–479.

[10] Musser, D. R.; Saini, A. ***STL Tutorial and Reference Guide***. Addison-Wesley, 1996.

[11] Nenet. ***Neural Networks Tool***. Disponível em http://www.mbnet.fi/~phodju/nenet/. Consultado em 2003.

[12] SNNS- ***Stuttgart Neural Network Simulator***. Disponível em http://www-ra.informatik.uni-tuebingen.de/SNNS/. Consultado em 2003.

[13] Stroustrup, B. ***A Linguagem de programação C++***. Bookman, 2000.

[14] Vesanto, J.; Himberg, J.; Alhoniemi, E.; Parhankangas, J. Self-Organizing Map in Matlab: the SOM Toolbox. ***Proceeding of the Matlab DSP Conference***. 1999. p. 35–40.

ANEXO I

Informações sobre os parâmetros iniciais de treinamento e índices de avaliação

Log Info

	Value
Data File	
Learning type	Batch
Training epochs	1000
Initial radius	16
Lattice	Hexagonal
Neighbourhood	Gaussian
Dimension	18x16
Quantization error	0.195594
Topological error	0.0497076
Mapcode File	
Number Cluster	20
Davies-Bouldin(p=2,q=1.0)	0
Davies-Bouldin(p=2,q=2.0)	0
Davies-Bouldin (Data)(p=2,q=1)	3.91791
Davies-Bouldin(Data)(p=2,q=2)	1.91794
CDbw	110.142

OK

É possível ler os dados de entrada do banco *Terralib* ou de arquivos. O Mapa treinado pode ser salvo.

SOM Lib GUI

File Map Help

Open Map... Ctrl+O
Save Map Ctrl+S
Save Map As...
Open Data Ctrl+Y
Data from Terralib ...

SOM Lib GUI

File Map Help

Learn: Batch / Standard
Learn rate
Num iteractions 1000
Init Radius 16
Learning in two fases
Lattice: Hexagonal / Rectangular
Neighborhood Function: Gaussian / Bubble / Cut Gauss / EP
Dimension: Num Lines 18; Num colummns 16
Default Values
Clustering parameters
Min nº of neuros per cluster 3 — Force All Neurons
Min nº of vectors data per cluster 2 — Cluster
Field name: — Table Name E_revisto_1991_pol
Horizontal — Create 1-1 Clustering — Create n-1 Clustering

Canvas especializados em visualização da detecção de agrupamentos, U-matriz e visualização simples dos neurônios especializados, respectivamente.

Data Label

	Label
1	0310
2	0319
3	0320
4	0322
5	0324
6	0325

0

OK

Cada neurônio (círculo) estará associado a um diálogo contendo os Ids dos padrões de entrada relacionados ao mesmo.

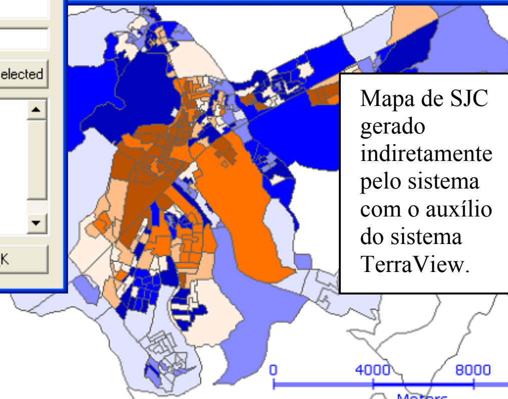
Terralib Database Params

Host localhost
Database File D:/BancosDeDados/sjc.mdb — Load File...
User
Password
Table IBGE_revisto_1991_pol
WHERE statement CATEGORIA <> 0
Data Label Variable ROTULO
Variable — Add — Delete Selected

ARENDR
DESEDUCR
ESTEDUCR
LONGR
QAMBR
QDOMR
MANALFR

Cancel — OK

Diálogo de entrada dos dados para conexão e extração dos dados através da Terralib.



Mapa de SJC gerado indiretamente pelo sistema com o auxílio do sistema TerraView.