Universidade de São Paulo

Instituto de Geociências

AUTTITUDE: MODELO COMPUTACIONAL ORIENTADO A OBJETOS PARA DADOS DIRECIONAIS COM ÊNFASE EM TECTÔNICA E GEOLOGIA ESTRUTURAL

MONOGRAFIA DE TRABALHO DE FORMATURA

TF-14/01

Aluno: Arthur Endlein Correia

Orientador: Ginaldo Ademar da Cruz Campanha

Co-Orientadora: Camila Duelis Viana

São Paulo

2014

Universidade de São Paulo

Instituto de Geociências

AUTTITUDE: MODELO COMPUTACIONAL ORIENTADO A OBJETOS PARA DADOS DIRECIONAIS COM ÊNFASE EM TECTÔNICA E GEOLOGIA ESTRUTURAL

MONOGRAFIA DE TRABALHO DE FORMATURA

TF-14/01

Aluno: Arthur Endlein Correia

Orientador: Ginaldo Ademar da Cruz Campanha

Co-Orientadora: Camila Duelis Viana

São Paulo

2014

Universidade de São Paulo

Instituto de Geociências

AUTTITUDE: MODELO COMPUTACIONAL ORIENTADO A OBJETOS PARA DADOS DIRECIONAIS COM ÊNFASE EM TECTÔNICA E GEOLOGIA ESTRUTURAL

MONOGRAFIA DE TRABALHO DE FORMATURA

TF-14/01

Aluno: Arthur Endlein Correia

Orientador: Ginaldo Ademar da Cruz Campanha

Co-Orientadora: Camila Duelis Viana

São Paulo

2014

A área do meu TF é

6.283185307179586476925286766559005768394338798750211641949889184615632812572417997256069650684234135964296173026564613294187689219101164463450718816256962234900568205403877042211119289245897909860763928857621951331866892256951296467573566330542403818291297133846920697220908653296426787214520498282547449174013212631176349763041841925658508183430728735785180720022661061097640933042768293903883023218866114540731519183906184372234763865223586210237096148924759925499134703771505449782455876366023898259667346724881313286172042789892790449474381404359721887405541078434352586353504769349636935338810264001136254290527121655571542685515579218347274357442936881802449906860293099170742101584559378517847084039912224258043921728068836319627259549542619921037414422699999996745956099902119463465632192637190048918910693816605285044616506689370070523862376342020006275677505773175066416762841234355338294607196506980857510937462319125727764707575187503915563715561064342453613226003855753222391818432840397876190514402130971726557731872306763655936460603904070603705937991547245198827782499443550566958263031149714484908301391901659066233723455711778150196763509274929878638510120801855403342278019697648025716723207127415320209420363885911192397893535674898896510759549453694208095069292416093368518138982586627354057978304209504324113932048116076300387022506764860071175280494992946527828398545208539845593564709563272018683443282439849172630060572365949111413499677010989177173853991381854421595018605910642330689974405511920472961330998239763669595507132739614853085055725103636835149345781955545587600163294120032290498384346434429544700282883947137096322722314705104266951483698936877046647814788286669095524833725037967138971124198438444368545100508513775343580989203306933609977254465583572171568767655935953362908201907767572721901360128450250410234785969792168256977253891208483930570044421322372613488557244078389890094247427573921912728743834574935529315147924827781731665291991626780956055180198931528157902538936796705191419651645241044978815453438956536965202953981805280272788874910610136406992504903498799302862859618381318501874443392923031419716774821195771919545950997860323507856936276537367737885548311983711850491907918862099945049361691974547289391697307673472445252198249216102487768780902488273099525561595431382871995402..**Ouch!**

Pô, Urtiga, não me bate.

RESUMO

Dados direcionais (aqueles que podem ser representados na forma de vetores unitários), possuem grande aplicação em Geologia, seja na geologia estrutural, representando a orientação de estruturas planares e lineares ou na sedimentologia, analisando direções de paleocorrentes. Tradicionalmente sua representação é feita na forma de projeções estereográficas, porém sua tradução na forma de vetores permite maior facilidade de operação. O trabalho teve por objetivo a produção de uma biblioteca de análise de dados direcionais orientada a objetos com ênfase em tratamento de dados de geotectônica e geologia estrutural, de fácil utilização e extensão, e sua utilização como base matemática para a remodelagem do *software* OpenStereo. O projeto foi desenvolvido em sua maioria na linguagem Python série 2.7.x com o auxílio do pacote numérico Numpy, com algumas funções escritas em Fortran 90 e compiladas posteriormente. O desenvolvimento se deu com a criação do sistema de entrada de dados, seguido do sistema para sua tradução na forma de cossenos diretores. Posteriormente, deu-se a criação do modelo de objeto para dados direcionais genéricos, que deu origem aos modelos específicos para dados esféricos, circulares, direcionais e axiais, com seus parâmetros estatísticos auxiliares calculados automaticamente. Também foram desenvolvidas em paralelo funções auxiliares para tarefas como a paralelização automática de operações simples. Considerando ser uma biblioteca Python, o Auttitude serve tanto para o desenvolvimento de outras aplicações quanto para o uso em ambiente interativo, como plataforma para a análise de dados direcionais. Os testes realizados deram resultados positivos. Sua integração ao OpenStereo permite que dados complexos sejam visualizados e facilita sua organização, ao mesmo tempo que confere rapidez, robustez e facilidade de manutenção.

Palavras-chave: Dados direcionais, Python, estereograma

ABSTRACT

Directional data (that is, data that may be represented as unitary vectors) have many different uses in the geological sciences, either on structural geology, representing the attitudes of planar or linear structures or in sedimentology, used for paleocurrent direction analysis. They are usually represented graphically, but translating them into vectors have many advantages. This work aims to create a object oriented directional data analysis library, with emphasis on geotectonics and structural geology related data, ease of use, and as a mathematical engine for OpenStereo, a directional data analysis *software*. This project was mainly developed using the Numpy package for numerical processing, with a few methods written in Fortran 90 and compiled with Python support. Initially the data input system was created, together with the data translation into direction cosines module. Afterwards, the generic object oriented module for directional data was developed, which gave birth to the specific models for spherical, circular, directional and axial data, with their respective statistical parameters automatically calculated. Auxiliary methods for automatic parallelizing of simple task were also developed. Considering that it is a Python library, it may either be used for creating new directional data analysis *software* but also in a interactive shell as an analysis system. Test results were positive. Its integration to OpenStereo allows complex data sets to be easily visualized, at the same time giving it speed, stability and ease of maintenance.

Keywords: Directional Data, Python, stereonet

SUMÁRIO

1. INTRODUÇÃO

Dados direcionais são aqueles que podem ser representados como vetores unitários, tipicamente em duas ou três dimensões, podendo representar pontos na superfície de uma esfera (3D) ou numa circunferência (2D), a orientação de linhas ou rotações. São às vezes também referidos como dados de orientação, angulares ou esféricos. Podem ser vetoriais, se possuírem direção e sentido, ou axiais, caso possuam apenas direção. (Fisher, Lewis e Embleton 1987, Jupp e Mardia 1989).

Existem diversos exemplos de seu uso dentro da geologia, seja na cristalografia, representando a orientação relativa das faces de um cristal; na geologia estrutural, representando a orientação de estruturas planares e lineares; em paleomagnetismo, como pólos de magnetização remanescente; na sedimentologia, analisando direções de paleocorrentes; e na geotectônica, tanto no estudo da cinemática de placas tectônicas, representando traços de hotspot, como no de falhas transformantes.

Sua representação em Geologia é tradicionalmente feita com uso de projeções estereográficas (Schmidt-Lambert e Wullf principalmente, e mais raramente Ortográfica e Gnomônica) para dados tridimensionais, ou com diagramas em roseta para dados bidimensionais. No entanto, distribuições e métodos estatísticos numéricos são conhecidos há bastante tempo e estão disponíveis em quase todos os *software* voltado ao tratamento de dados estruturais de orientação. A representação destes dados na forma de vetores, permite uma maior clareza e facilidade de operação, facilitando a manipulação dos mesmos por meios de programação.

A disponibilidade de linguagens de programação de alto nível orientadas a objeto, como Python - que possui uma extensa biblioteca padrão de funções, além do constante desenvolvimento de bibliotecas de funções especializadas (módulos), que permitem expandir as capacidades base da linguagem -, o tornam uma ferramenta poderosa para tratamento de dados direcionais. Sendo assim, sua aplicação na análise de dados geológicos é muito vantajosa, tornando o processamento mais rápido e completo.

2. OBJETIVOS

O objetivo fundamental deste trabalho foi a produção de uma biblioteca de análise de dados direcionais orientada a objetos com ênfase em tratamento de dados de geotectônica e geologia estrutural, de fácil utilização e extensão. Construindo-se esta, objetivou-se então a remodelagem do *software* OpenStereo, utilizando-a como base matemática para este.

3. FUNDAMENTAÇÃO TEÓRICA

O trabalho estatístico com dados direcionais no âmbito da geologia pode ser traçado desde Schmidt (1917), o qual utilizou-os para a análise de estruturas em ardósias, desenvolvendo tanto o uso do diagrama em roseta quanto a distribuição normal envolta no círculo. A maior parte do avanço em tratamento de dados circulares nesta época ocorreu na área da geologia, com a exceção notável de Von Mises em 1918, que o aplicou na modelagem de erro na determinação de pesos atômicos, desenvolvendo a distribuição estatística que leva seu nome, e que até hoje é a mais usada neste tipo de dado (Fisher 1993). Já a análise estatística para dados direcionais tridimensionais iniciou-se com Fisher (1953), ao definir uma distribuição para concentrações de vetores próximos a um vetor médio equivalente à distribuição normal, desenvolvida para tratar dados de paleomagnetismo. Consagrada como distribuição de *Fisher*, dela surgiram várias outras, como a de *Watson* (Bingham 1964, Watson 1965), para concentração de dados axiais em torno de uma moda ou em círculos máximos, *Bingham-Mardia* (Mardia e Gadsden 1977, Bingham e Mardia 1978), para concentrações em guirlanda de círculo menor e *Bingham* para dados axiais (Bingham 1964, 1974) ou *Kent* para dados vetoriais (Kent 1982), que são capazes de descrever vários padrões diferentes, variando entre axiais, guirlanda de círculo máximo e uniformes, com parâmetros obtidos a partir dos autovalores e autovetores de um tensor de orientação calculado a partir dos dados direcionais.

Além destas distribuições estatísticas mais formais e seus parâmetros associados, vários outros métodos numéricos *ad-hoc* foram criados, muitos deles bastante úteis para a análise de dados geológicos. Temos como exemplo o trabalho de Kamb (1959), que definiu critérios para detecção de desvio da uniformidade em malhas de contagem, posteriormente revisitado e ampliado por Robin e Jowett (1986). Além deste, Woodcock (1977) e posteriormente Vollmer (1990) desenvolveram métodos para classificação da forma geral dos dados a partir dos autovalores e autovetores de sua matriz de dispersão, servindo como base inicial não paramétrica para tratamento dos dados. Dados circulares são geralmente representados na forma de rosácea, porém alguns trabalhos (e.g., Fisher 1993, Munro 2012) propõem uma curva contínua baseada em médias móveis, semelhante ao que é feito há mais tempo com as malhas de contagem de dados esféricos.

3.1. Análise de dados direcionais

3.1.1. Notação de atitudes medidas em campo

Há uma grande diversidade de métodos para a representação de atitudes em geologia medidas com a bússola, especialmente para dados planares. Isto se mostra um obstáculo adicional ao tratamento deste tipo de dado numericamente, pois em geral é necessária a sua conversão manual. Portanto, torna-se vantajoso um método robusto e rápido para a detecção e conversão automática de atitudes em diferentes sintaxes.

Basicamente, usa-se ou o rumo do mergulho da camada ou a direção da camada para representar sua orientação horizontal, e o mergulho para representar sua inclinação vertical. Entretanto, o ângulo da direção ou rumo do mergulho pode ser apresentado na forma de azimute, indo de 0 a 360 (ou 0 a 180, ou 270 a 90, no caso de direções), ou como quadrante, indo de N0E a N90E ou N0W a N90W ou S0E a S90E ou S0W a S90W, dependendo do quadrante que ele se encontra, e dependendo também se é utilizado rumo do mergulho ou direção. Especificamente para direções, pode-se escrevê-las através da regra da mão direita, onde a direção é sempre o ângulo em azimute ou quadrante que está noventa graus a esquerda do rumo do mergulho, ou adicionando-se o quadrante do rumo do mergulho a uma direção medida entre 0 e 180 ou 270 e 90, em azimute ou N90W e N90E. Algumas vezes ainda utilizam-se medidas em outros quadrantes que não estes esperados, como medidas de direção contadas a partir do sul, seja por inexperiência ou gosto particular do geólogo que coletou os dados, ou anota-se medidas aproximadas como NS ou EW, ou apenas N, S, E ou W.

Não é possível em absoluto determinar se uma medida em particular foi feita em rumo do mergulho ou em direção, porém a partir desta informação básica o restante pode ser convertido sem grandes dificuldades por *software*. A seguinte expressão regular é a base desta análise:

```
([NSEW]{0,2})(\d*)([NSEW]{0,2})[^NSEW0-9]*(\d+)([NSEW]{0,2})
```

Expressões regulares (Kleene 1956, Pilgrim 2009) são um método para se validar e analisar textos de forma robusta e rápida, permitindo que se defina quais variações são aceitáveis e ainda extraindo pedaços específicos do texto analisado. No caso acima, cada parêntese indica um dos pedaços analisados, sendo eles, na ordem, ([NSEW]{0,2}), a primeira letra do azimute, que deve ser uma dentre N, S, E e W, NE, NW, SE, SW ou vazio; (\d*), o azimute em si, que deve ser um número ou vazio; ([NSEW]{0,2}), a segunda letra do azimute, que deve ser também uma entre N, S, E e

W, NE, NW, SE, SW ou vazio; [^NSEW0-9]*, um separador, que pode ser qualquer caractere que não um número ou as letras N, S, E ou W ou pode ser vazio; (\d+), o mergulho, que deve ser um número; e ([NSEW]{0,2}), o quadrante do mergulho, que deve ser um entre N, S, E, W, NE, NW, SE, SW ou vazio. A partir da presença (algum valor) ou ausência (vazio) de cada uma dessas partes (excetuando o separador e o valor do mergulho) segue-se então para a seguinte tabela verdade, com casos omissos sendo considerados erros:

Tabela 1 - Tabela verdade da expressão regular para determinação do tipo de notação de atitude geológica.

Exemplo	[NSEW]{0,2}	\d*	[NSEW]{0,2}	[NSEW]{0,2}	Tipo
N30E/50NW	S	S	S	S	Direção, mergulho, quadrante
140/50	N	S	N	N	Regra da mão direita
140/50NE	N	S	N	N	Direção, mergulho, quadrante
NW/50	S	N	N	N	Regra da mão direita
NS/50E	S	N	S	S	Direção aproximada, mergulho, quadrante

Obs.: S indica presença de caractere e N indica ausência de caractere.

A partir desta informação e do fato de tratarem-se de rumo de mergulho ou direção, torna-se possível converter estas atitudes para um único formato, que é então utilizado durante o resto da análise, transparente para o usuário. Neste caso, é utilizado rumo do mergulho / mergulho.

3.1.2. Vetores e álgebra

Apesar de grande parte das operações entre dados direcionais poder ser feita por outros métodos não vetoriais (sejam eles métodos gráficos, como projeções, ou numéricos, através de trigonometria plana ou esférica), é difícil que estes superem os vetoriais em termos de clareza de sintaxe e consequentemente facilidade de programação. Vetores são conjuntos de números, um para cada dimensão representada, ordenados como uma única linha ou coluna. São um caso especial de matrizes, que possuem *m* linhas por *n* colunas (Fergusson 1994).

Para representar dados direcionais como vetores torna-se necessária a conversão das atitudes (ou coordenadas, ou ângulos em geral) de entrada em cossenos diretores, que

recebem este nome por serem numericamente o cosseno do ângulo entre a atitude em questão e os eixos x, y e z (tipicamente, Norte, Leste e Cima, para atitudes em geologia).

Para que se entenda a conversão de, tomando como exemplo mais simples, uma lineação, divide-se primeiramente o vetor resultante da atitude em seus componentes vertical e horizontal (Kim 2005). Sendo o caimento (*plunge*) o ângulo entre a linha e o plano horizontal, seu componente vertical será o seno deste ângulo, restando como componente horizontal o seu cosseno. Sendo o rumo do caimento (*trend*) o ângulo entre o eixo Norte (X) e o componente horizontal do vetor, este se dividirá entre o cosseno do ângulo do rumo de caimento para o eixo X e o seno do ângulo de caimento para o eixo Y (Figura 1). Em suma, a seguinte fórmula é válida:

$$u = \begin{bmatrix} \cos(trend)\cos(plunge) \\ sen(trend)sen(plunge) \\ sen(plunge) \end{bmatrix}$$

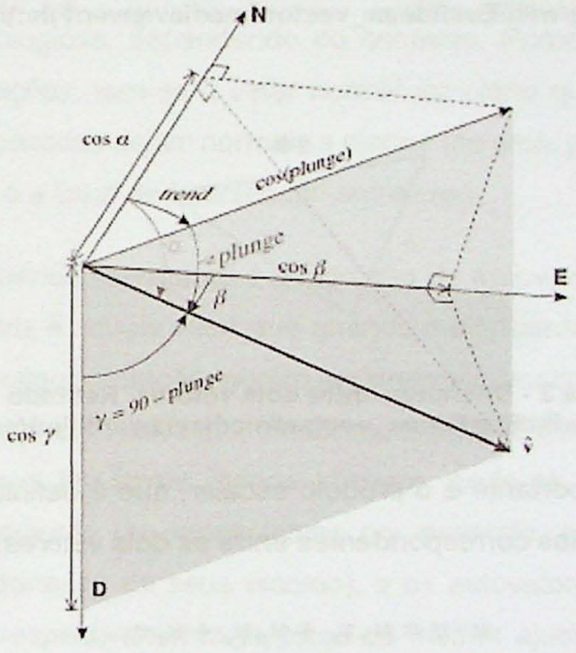

Figura 1 - Relação entre os cossenos diretores e rumo do caimento e caimento para uma linha $\hat{v}$. Retirado de Allmendinger, Cardozo e Fisher 2012.

Fórmulas equivalentes para atitudes de planos são facilmente construídas de forma similar, sendo necessário apenas lembrar que utiliza-se o vetor normal do plano como representação vetorial deste.

A conversão de volta em atitudes a partir de cossenos diretores se dá também de forma simples, bastando se calcular o arcosseno (ou arcocoseno, no caso de planos) do componente vertical e o arcotangente da razão entre os componentes y e x (ou x e y,

no caso de planos), restando ainda comparar os sinais destes para se determinar o quadrante do rumo do caimento.

A partir destes vetores, algumas operações úteis se tornam possíveis. Primeiramente, vetores podem ser somados, adicionando-se cada coordenada correspondente e obtendo-se um terceiro vetor (Figura 2). Pode-se também subtrair um vetor de outro, de forma semelhante, resultando no vetor diferença entre os dois (Figura 3).

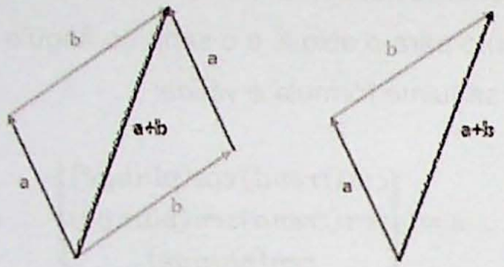

Figura 2 - Soma de dois vetores. Retirado de http://en.wikipedia.org/wiki/Euclidean_vector#mediaviewer/File:Vector_addition.svg

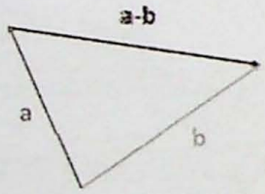

Figura 3 - Diferença entre dois vetores. Retirado de http://en.wikipedia.org/wiki/Euclidean_vector#mediaviewer/File:Vector_subtraction.svg

A terceira operação importante é o produto escalar, que é definido como a soma dos produtos das coordenadas correspondentes entre os dois vetores, ou seja:

$$u \cdot v = u_x v_x + u_y v_y + u_z v_z$$

É possível então obter-se a magnitude de um vetor u através do produto escalar dele consigo mesmo, que pode ser representada por |u|. Além disso, o produto escalar de dois vetores quaisquer é igual ao produto entre suas magnitudes e o cosseno do ângulo (teta) entre os dois vetores:

$$u \cdot v = \|u\|\|v\|cos\theta$$

Dividindo-se então o valor do produto escalar entre dois vetores pelas magnitudes dos mesmos, é possível obter-se o ângulo entre eles. No caso de vetores unitários, o cálculo

é direto. Para o caso geral de matrizes, é necessário que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz, e o produto escalar dá-se pela soma do produto dos números respectivos das linhas da primeira matriz pelas colunas da segunda matriz, ou seja,

$$(A \cdot B)_{i,j} = \sum a_{i,k} * b_{k,j}$$

A última operação fundamental entre dois vetores é o produto vetorial, cujo resultado é um terceiro vetor perpendicular aos dois originais e com magnitude igual ao produto entre o seno do ângulo entre os dois vetores e as magnitudes dos dois vetores. Numericamente pode ser obtido através da seguinte fórmula:

$$u \times v = w = \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

O produto vetorial pode ter atribuído ao menos dois significados importantes para análise de dados geológicos, dependendo do contexto. Primeiro, considerando-se o produto de duas lineações, tem-se o vetor normal ao plano que as contém. Caso os vetores a serem multiplicados sejam normais a planos (ou seja, polos de planos), o vetor resultante do produto é a linha de intersecção entre eles.

A última operação genérica importante é a extração de autovalores e autovetores. Um autovetor de uma matriz é aquele vetor que quando multiplicado por ela mantém a sua direção, sem sofrer qualquer rotação, porém podendo ter seu comprimento modificado. Autovalor será então, por esta definição, o fator multiplicativo que será aplicado por esta matriz a este autovetor. De forma menos abstrata, pode-se montar uma matriz de dispersão para um conjunto de dados (como por exemplo, a matriz de covariância relacionando as coordenadas de seus vetores), e os autovalores e autovetores desta matriz representarão respectivamente os eixos de melhor ajuste para o elipsoide que descreve os dados (Figura 4), e os autovalores os comprimentos relativos destes eixos (Pearson 1901).

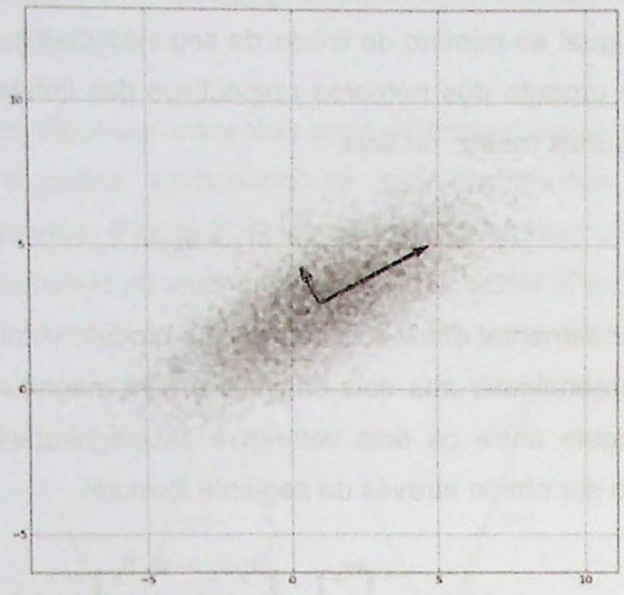

Figura 4 - Autovetores associados a um conjunto de pontos, como eixos do elipsoide de melhor ajuste.

A relação entre o comprimento destes eixos leva a três formas extremas possíveis para dados tridimensionais: linear, planar e esférica:

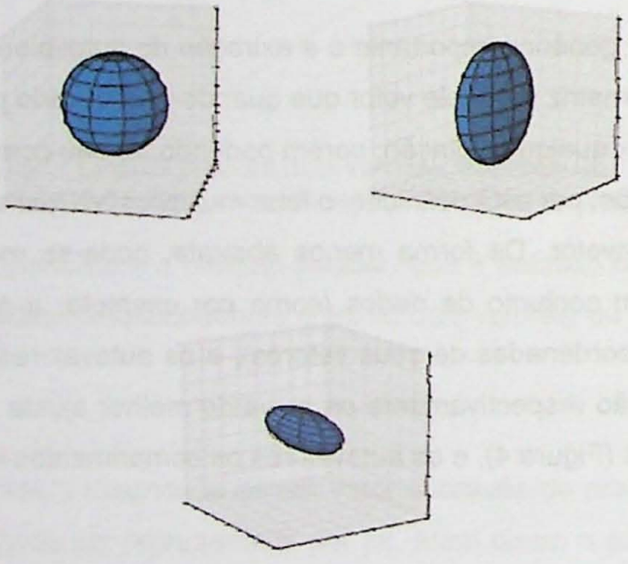

Figura 5 - Formas extremas mostrando a relação entre os três autovalores.

A classificação destas formas pode ser feita de forma numérica aproximada, por métodos explicados abaixo.

3.1.3. *Estatísticas descritivas*

A partir destes blocos básicos, Fisher (1986) define para dados direcionais esféricos algumas estatísticas descritivas. Sendo $x_i = \{x_i, y_i, z_i\}$ cada ponto de dado, inicialmente definimos o vetor resultante e sua magnitude:

$$r = \sum x_i, \qquad \bar{x} = \frac{r}{n}, \qquad R = |r|, \qquad \bar{R} = \frac{R}{n}$$

Este vetor nos fornece uma ideia básica da direção geral dos pontos. De forma parecida, o comprimento médio, $\bar{R}$, será mais próximo de 1 quanto mais concentrados forem os pontos em torno desta direção média. Entretanto, no caso de dados direcionais, dois grupos concentrados opostos na esfera, com número parecido de pontos, terão como comprimento médio algo próximo a zero. Para dados axiais, se não for tomado cuidado, o mesmo problema pode ocorrer, por limitações da representação deste tipo de dado como vetores unitários.

Uma correção razoavelmente simples, e disponível como parte das análises estatísticas realizadas pelo programa desenvolvido, é comparar os pontos de dado com o primeiro autovetor de sua matriz de dispersão, que estará próximo também a esta direção média porém não será afetado por este fenômeno, visto que os pontos x e $-x$ não são diferenciados no seu cálculo. A partir disto, é possível concentrá-los em um mesmo hemisfério relativo a este autovetor, evitando alguns problemas com o cálculo destes parâmetros e outros tratamentos.

Especificamente para dados circulares definem-se também a variância circular V e o desvio padrão circular v (Fisher 1993):

$$V = 1 - R, \qquad v = \sqrt{2\log(1 - V)}$$

3.1.4. *Eigenanalysis e parâmetros de forma*

Estes parâmetros básicos de dispersão em torno da média são então complementados pelos parâmetros de forma, extraídos da matriz de dispersão T (também chamada de matriz de orientação), definida da seguinte forma:

$$T = \frac{1}{n}\sum {x_i}^T \cdot x_i$$

Os autovetores (u_i) e autovalores (τ_i) desta matriz representam, como explicado acima, os eixos e comprimentos dos eixos do elipsoide de melhor ajuste para os dados.

Considerando os autovalores em ordem decrescente, e seus respectivos autovetores, podemos analisar descritivamente a forma da distribuição de dados, considerando a seguinte tabela:

Tabela 2 - Interpretação descritiva das formas de distribuição esféricas em termos dos autovalores $\bar{t}_1, \bar{t}_2, \bar{t}_3$ de $\bar{T}$ e o comprimento resultante $\bar{R}$. Traduzido de Mardia (2000).

Magnitudes relativas dos autovalores	Tipo de distribuição	Outras características
$\bar{t}_1 \simeq \bar{t}_2 \simeq \bar{t}_3$	Uniforme	
$\bar{t}_1$ grande; $\bar{t}_2, \bar{t}_3$ pequenos		
(i) $\bar{t}_2 \neq \bar{t}_3$	Unimodal se $\bar{R} \simeq 1$, bimodal caso contrário	Concentrado em uma extremidade de $\bar{t}_1$ Concentrado nas extremidades de $\bar{t}_1$
(ii) $\bar{t}_2 \simeq \bar{t}_3$	Unimodal se $\bar{R} \simeq 1$, bipolar caso contrário	Simetria rotacional em torno de $\bar{t}_1$
$\bar{t}_3$ pequeno; $\bar{t}_1, \bar{t}_2$ grandes		
(i) $\bar{t}_1 \neq \bar{t}_2$	Guirlanda	Concentrado em círculo maior no plano $\bar{t}_1, \bar{t}_2$
(ii) $\bar{t}_1 \simeq \bar{t}_2$	Guirlanda simétrica	Simetria rotacional em torno de $\bar{t}_3$

Entretanto, resta-nos definir o que são autovalores grandes e pequenos. Woodcock (1977) e posteriormente Vollmer (1990) definiram razões que facilitam esta inferência. Para Woodcock, estabelecendo-se as razões

$$x = \ln(\frac{\lambda_1}{\lambda_2})$$

$$y = \ln(\frac{\lambda_2}{\lambda_3})$$

pode-se analisar pelo seguinte gráfico a forma da distribuição dos pontos:

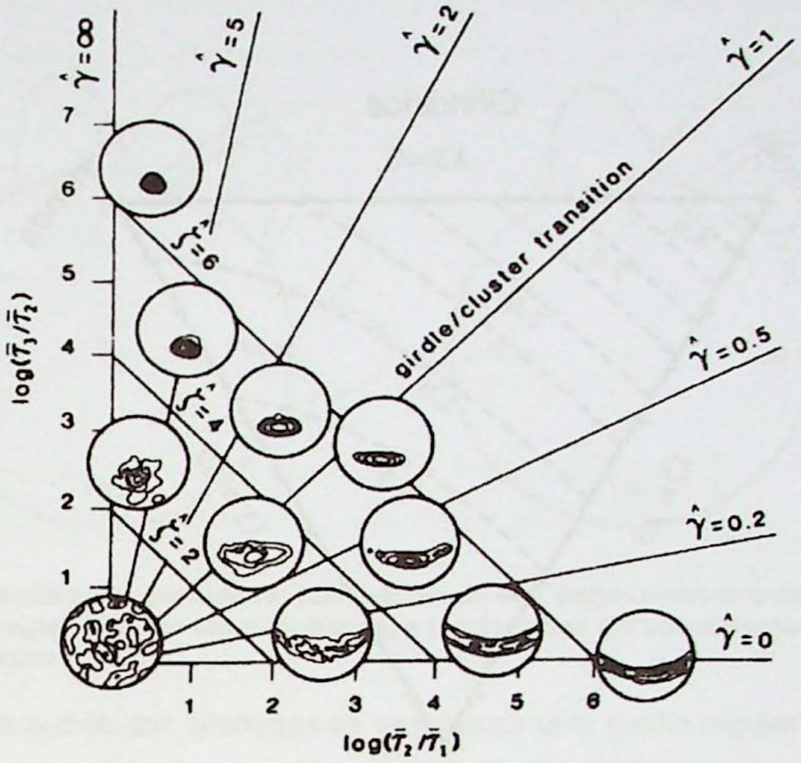

Figura 6 - Diagrama de Flinn adaptado (Woodcock 1976). Extraído de Fisher (1986).

Entretanto, este gráfico ainda abre espaço para ambiguidades, e não é de interpretação muito simples. Vollmer (1990), por outro lado, propõe as seguintes equações:

$$P = \frac{(\lambda_1 - \lambda_2)}{N} \qquad G = \frac{2(\lambda_2 - \lambda_3)}{N} \qquad R = \frac{3(\lambda_3)}{N} \qquad C = \ln(\frac{\lambda_1}{\lambda_3})$$

referentes, respectivamente, a proximidade do conjunto de dados à distribuição pontual (P, em que os autovalores seriam no extremo 1, 0 e 0), em guirlanda (G, com autovalores 0,5, 0,5 e 0) ou aleatória (R, com autovalores 1/3, 1/3 e 1/3). Além disso, define também um parâmetro de cilindricidade (C), baseado no logaritmo da razão entre o primeiro e o terceiro autovetores. O diagrama triangular resultante é de entendimento muito mais simples (Figura 7)

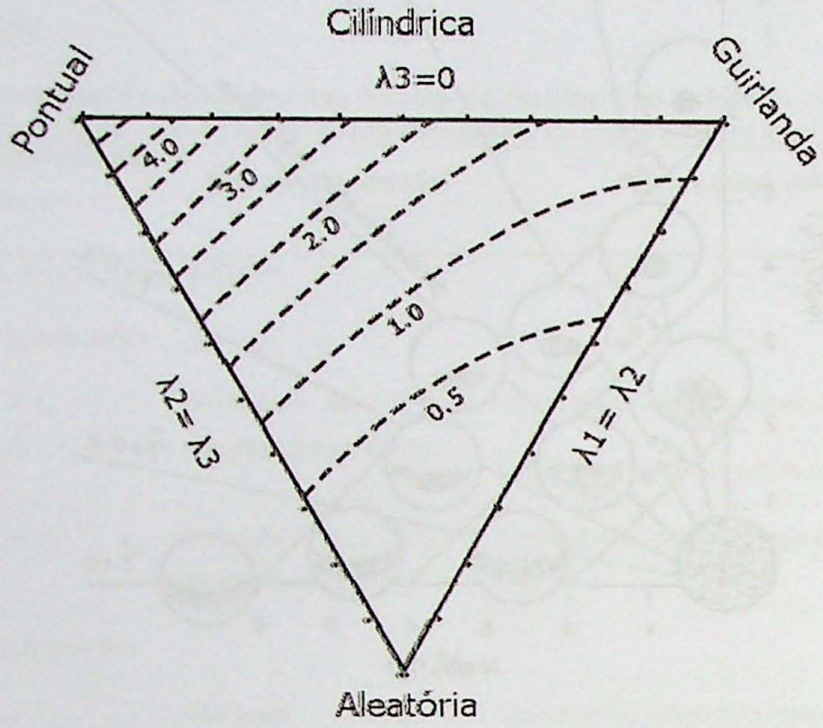

Figura 7 - Diagrama triangular para classificação de forma. Adaptado de Vollmer (1990).

3.1.5. Malhas de contagem

Malhas de contagem são malhas regulares de pontos aproximadamente equiespaçados na esfera ou círculo, onde para cada nó desta ou se conta a quantidade de pontos que estão a uma distância angular menor que um valor escolhido ou se calcula um valor total que depende da distância até cada ponto de dado. De uma forma ou outra, ela representa uma estimativa da função de densidade de probabilidade para os dados (Figura 8).

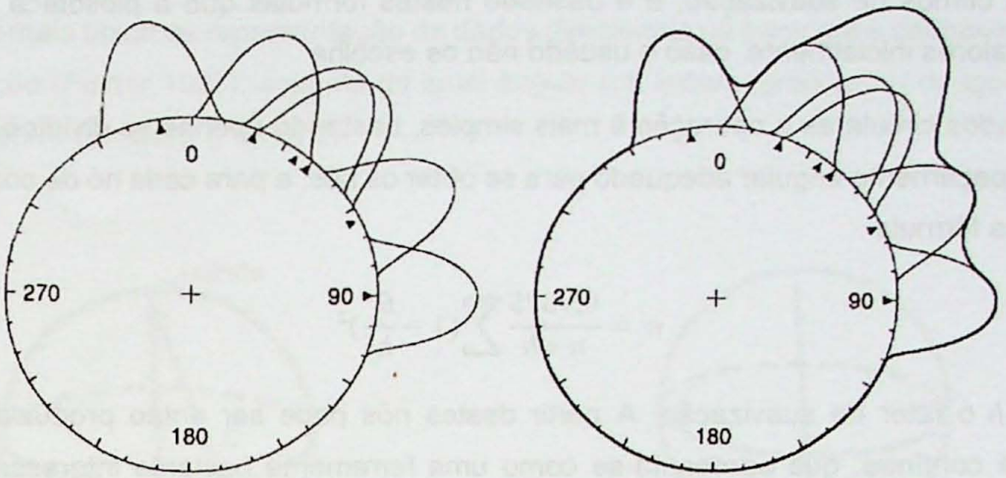

Figura 8 - Estimativa não paramétrica da densidade dos dados usando o método de Fisher. Cada ponto é espalhado em sua vizinhança, e a estimativa é a soma destas contribuições. Extraído de Fisher (1993).

Existem diversos métodos diferentes de se produzir uma malha regular para se fazer o cálculo desta estimativa (Diggle e Fisher 1985, Kamb 1956), entretanto a forma mais flexível é a sugerida por Robin e Jowett (1986), na qual a partir do espaçamento angular médio esperado t, se distribuem círculos de latitude espaçados por este valor, e dentro de cada um destes círculos se calcula qual distância longitudinal a equivale a esta distância t de círculo máximo, em um círculo separado por ângulo v do eixo vertical, segundo a relação:

$$a = \frac{2(sen(\frac{t}{2})}{sen(v)}$$

A partir dos cossenos diretores dos nós destas malhas calcula-se então o ângulo até cada ponto de dado, a partir de adaptação do produto escalar:

$$\theta = \text{acos}(\frac{v \cdot u}{|v||u|})$$

Então, ou contando-se o número de pontos com ângulo até um limite definido ou aplicando-se a fórmula:

$$w = \sum e^{k(cos\theta - 1)}$$

é calculado o valor para cada nó da malha. A escolha da constante K na fórmula acima ou do ângulo limite para contagem tem como efeito fundamental suavizar as concentrações de dados ao longo da esfera, permitindo se ver grandes tendências, porém apagando detalhes. A escolha deste fator deve ser feita, em casos extremos, de

forma experimental, porém Robin e Jowett (1986) sugerem fórmulas para estimar valores ótimos de suavização, e é baseado nestas fórmulas que a biblioteca calcula estes valores inicialmente, caso o usuário não os escolha.

Para dados circulares a operação é mais simples, bastando apenas se dividir o círculo pelo espaçamento angular adequado para se obter os nós, e para cada nó de contagem aplicar a fórmula

$$w = \frac{0{,}9375}{n * h} \sum (1 - \frac{\theta^2}{h^2})^2$$

sendo h o fator de suavização. A partir destes nós pode ser então produzida uma rosácea contínua, que apresenta-se como uma ferramenta bastante interessante de visualização de dados circulares (Fisher 1993, Munro 2012).

3.1.6. Rotações e Mudança de Eixos

Muitas vezes é conveniente ou necessário a rotação ou mudança dos eixos de coordenadas dos dados. No segundo caso, define-se o novo sistema de coordenadas a partir de três eixos ortogonais, x', y' e z', que devem ser unitários a não ser que se deseje deformar a distribuição espacial dos dados (que neste caso deixarão de pertencer a superfície de uma esfera unitária). Para se projetar dados neste sistema basta realizar o seguinte produto

$$u' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \cdot u = \begin{bmatrix} x'_x u_x + x'_y u_y + x'_z u_z \\ y'_x u_x + y'_y u_y + y'_z u_z \\ z'_x u_x + z'_y u_y + z'_z u_z \end{bmatrix}$$

Como um exemplo, a projeção de um conjunto de dados utilizando seus autovetores como novos eixos pode ser útil para a verificação de simetria.

A rotação, por outro lado, é feita a partir de um eixo unitário (v) e do ângulo de rotação anti-horária em torno dele (θ), a partir da equação matricial

$$u' = R \cdot u = \begin{bmatrix} cos\theta + v_x^2(1 - cos\theta) & v_x v_y(1 - cos\theta) - v_z sen\theta & v_x v_z(1 - cos\theta) + v_y sen\theta \\ v_x v_y(1 - cos\theta) - v_z sen\theta & cos\theta + v_y^2(1 - cos\theta) & v_y v_z(1 - cos\theta) - v_x sen\theta \\ v_x v_z(1 - cos\theta) + v_y sen\theta & v_y v_z(1 - cos\theta) - v_x sen\theta & cos\theta + v_z^2(1 - cos\theta) \end{bmatrix} \cdot u$$

Rotações serão em geral utilizadas para a restituição, como por exemplo se tentando reestabelecer o rumo de transporte sedimentar de uma estratificação cruzada que foi basculada tectonicamente.

3.1.7. Projeções

A forma mais típica de representação de dados direcionais na geologia é definitivamente a projeção (Fisher 1987), seja ela de igual ângulo (ou estereográfica) ou de igual área (ou Schmidt-Lambert) (Figura 9).

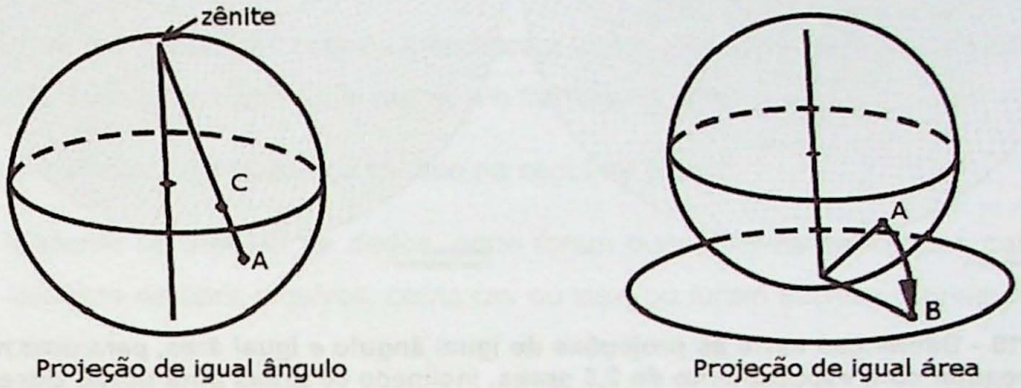

Figura 9 - Demonstração das projeções de igual ângulo e igual área, mostrando a relação entre pontos na esfera e projetados no plano. Adaptado de Fisher (1986).

A primeira tem seu uso principal na cristalografia, onde é necessária a correta representação dos ângulos entre as faces, sem distorções. Entretanto, para a visualização da distribuição de atitudes apenas a segunda presta-se de forma adequada. Enquanto a primeira preservará as formas, visto que círculos na esfera permanecerão como círculos no plano na projeção de igual ângulo, uma malha regular de pontos ao redor da esfera ficará excessivamente distorcida sem que se use uma projeção de igual área, como nesta figura:

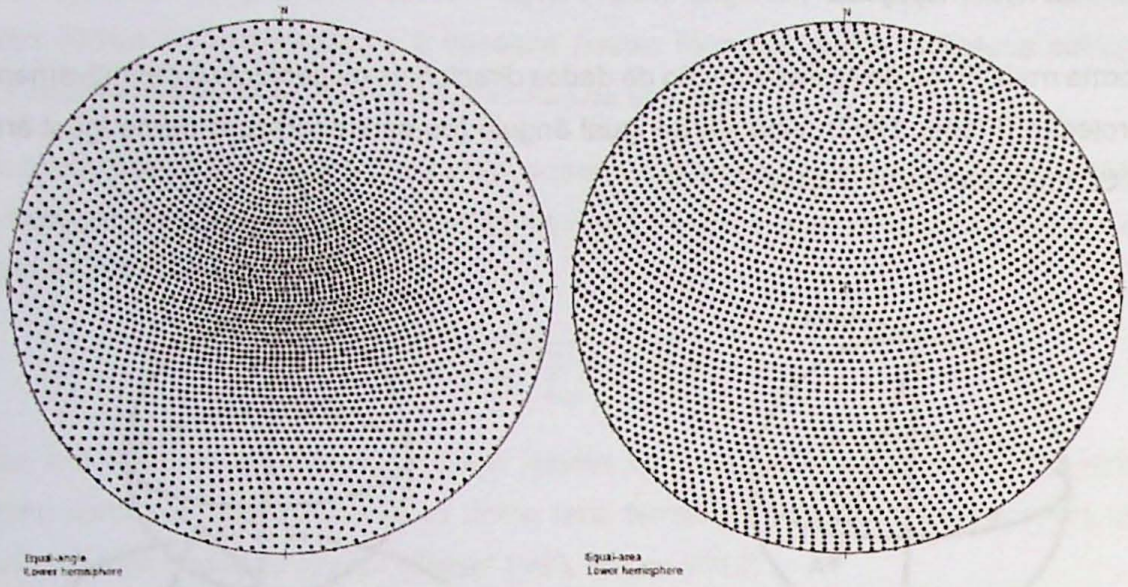

Figura 10 - Diferenças entre as projeções de igual ângulo e igual área, para uma mesma malha regular com espaçamento de 2,5 graus, inclinada 60 graus para norte. Observa-se que os círculos de latitude onde distribuem-se os nós na malha regular mantém-se como círculos no primeiro caso, entretanto a sua distribuição regular só é respeitada na segunda projeção.

Considerando-se o ponto $v = \{x, y, z\}$ pertencente a esfera, suas coordenadas X, Y no plano na projeção de igual ângulo são dadas pelas fórmulas

$$X = \frac{x}{1-z}, \qquad Y = \frac{y}{1-z}$$

Já no caso da projeção de igual área as seguintes fórmulas são utilizadas

$$X = x\sqrt{\frac{2}{1-z}}, \qquad Y = y\sqrt{\frac{2}{1-z}}$$

4. MATERIAIS E MÉTODOS

O projeto foi desenvolvido na linguagem Python série 2.7.x (Python *Software* Foundation 2010), tomando os cuidados necessários para que eventualmente seja convertido para a série 3.x. O desenvolvimento foi feito nessa série porque ainda parte considerável das bibliotecas numéricas e científicas não foi adaptada para a série 3.x. Entretanto é desejado que quando possível este trabalho seja convertido, pois esta nova série promete maior robustez e código em geral mais limpo.

Além da biblioteca padrão embutida na linguagem Python foi utilizado também o Numpy (Oliphant 2007), pacote numérico que simplifica enormemente o trabalho de

implementação com métodos rápidos e eficientes para boa parte das operações com vetores e matrizes.

Algumas partes da biblioteca foram escritas em Fortran 90, quando não for possível ou adequado fazê-las diretamente no Python com o auxílio do Numpy. Para isto, foi utilizado o compilador gfotran e o script f2py (Peterson 2009) para sua compilação com suporte integrado ao Python.

Considerando a integração com o OpenStereo, foram utilizadas também as bibliotecas wxPython, de interface gráfica, e yapsy, um *framework* de plug-ins.

O desenvolvimento da biblioteca foi feito na seguinte ordem:

1. Sistema de entrada de dados, onde foram buscadas as bibliotecas auxiliares capazes de abrir arquivos, como csv ou xlsx, ou foram escritas rotinas próprias para tal.
2. Sistema de tradução de dados, que permite que dados de diferentes tipos possam ser automaticamente traduzidos para uma única notação, gerando então cossenos diretores a partir destes.
3. Modelo de objeto para dados direcionais genéricos, que armazena os dados importados e traduzidos anteriormente, de forma genérica
4. A partir do modelo genérico, os modelos específicos para dados esféricos, circulares, direcionais e axiais, com seus parâmetros estatísticos auxiliares calculados automaticamente.

Em paralelo, funções auxiliares foram desenvolvidas para outras tarefas, como a paralelização automática de operações simples.

5. RESULTADOS OBTIDOS

Considerando ser uma biblioteca Python, o Auttitude serve tanto para o desenvolvimento de outras aplicações quanto para o uso em ambiente interativo, como plataforma para a análise de dados direcionais. Foram desenvolvidas 4 classes (Tabela 3) e 28 funções (Tabela 4) para o tratamento dos dados, além de se utilizar diversas bibliotecas externas. O código fonte dos pacotes desenvolvidos encontra-se reproduzido nos anexos de 1 a 6. Os arquivos de dados utilizados para testes encontram-se nos anexos 7 a 10.

Tabela 3 - Relação de classes desenvolvidas.

Classe	Descrição
DirectionalData	Classe base para análise de dados direcionais
SphericalGrig	Classe base para contagem de dados na esfera
PartProcessor	Parte do sistema de paralelização automática
Attitude	Sistema de tradução automática de notações de atitudes

Tabela 4 - Relação de funções desenvolvidas.

Função	Descrição
dcos	Converte polos em cossenos diretores
dcos_lines	Converte linhas em cossenos diretores
sphere	Converte cossenos diretores em polos
sphere_line	Converte cossenos diretores em linhas
invert	Converte polos em planos ou vice versa
RHR	Converte rumo do mergulho em regra da mão direita
equal_angle	Projeta os dados em projeção de igual ângulo
equal_area	Projeta os dados em projeção de igual área
concatenate	Concatena dois DirectionalData
intersect	Calcula todas as interseções entre dois DirectionalData
regular_grid	Produz uma malha regular em torno da semi esfera
sphere_regular_grid	Produz uma malha regular em torno da esfera
universal_loader	Carrega automaticamente diferentes formatos de arquivo
universal_translator	Converte automaticamente dados em rumo do mergulho / mergulho
load	Produz automaticamente DirectionalData a partir de um arquivo
calculate_axes	Calcula os autovetores e autovalores de um conjunto qualquer de dados
rotation_matrix	Produz uma matriz de rotação
rotate	Rotacional os dados
project	Projeta os dados em um novo sistema de coordenadas
parallel	Converte uma função para processamento paralelo
parallel_counter	Modifica uma função geradora de contadores para processamento paralelo
FisherCounterAxial	Gera contadores axiais segundo o método de Fisher
FisherCounter	Gera contadores segundo o método de Fisher
RobinGirdleCounter	Gera contadores para guirlandas segundo o método de Robin

A parte principal da biblioteca é a classe *DirectionalData*. Ela serve como contêiner para dados direcionais, ao mesmo tempo que oferece uma série de parâmetros estatísticos calculados automaticamente, tanto para dados circulares quanto esféricos. Ela pode ser utilizada fornecendo-se cossenos diretores como dados e quaisquer outros parâmetros adicionais disponíveis, que quando necessário serão utilizados em seus sub-módulos.

O primeiro passo para se produzir um objeto da classe *DirectionalData* é carregar os dados de entrada. Para isto, pode se utilizar a sequência de funções *universal_loader*, *universal_translator* e por fim passar os dados obtidos para a classe *DirectionalData*, ou utilizar-se da função *load*. O *universal_loader* é responsável pelo carregamento automático dos dados, lidando com diferentes formatos de arquivo. Nesta versão, ele é capaz de lidar com dados do tipo CSV, Numpy e Excel. A partir disto, o *universal_translator* traduz os dados da notação que estiverem para rumo do mergulho / mergulho, como explicado na seção de Notação de atitudes. O arquivo no exemplo abaixo, frat.dat, que encontra-se disponível no Anexo 10, exemplifica a variedade de notações encontradas. Esta sequência de métodos permite maior flexibilidade no carregamento dos dados, utilizando-se, por exemplo, de outras funções externas dentre os passos. A função *load* segue esta sequência básica automaticamente, convertendo então os dados em cossenos diretores e carregando-os em um objeto da classe *DirectionalData,* que é por ela retornado. Funciona de forma transparente ao usuário, utilizando valores padrão ou tentando extraí-los dos dados. Por exemplo,

```
>>>arquivo_de_entrada = universal_loader("frat.dat")
>>>dados_de_entrada = universal_translator(arquivo_de_entrada,
dip_direction=False)
>>>frat = DirectionalData(dcos(dados_de_entrada))
>>>b = auttitude.load("b.csv")
>>>dados = load("tocher.txt")
```

A partir de um *DirectionalData*, uma série de parâmetros estatísticos encontram-se disponíveis,

```
>>>dados.fisher_k
2.0994328692611806
>>>dados.eigenvalue[0]
109.57337905751763
>>>dados.vollmer_C
0.79271596847405035
>>>print dados
tocher.txt
n = 200
Expected Distribution:
Girdle
Eigenvectors:
1: 204.5 / 1.0
2: 295.4 / 42.2
3: 113.4 / 47.8
Shape parameter
K = 0.21
Strength parameter
C = 2.07
Normalized Eigenvalues:
S1: 0.548
```

```
S2: 0.383
S3: 0.069
Fabric (triangular diag.):
Point  = 0.165
Girdle = 0.628
Random = 0.207
>>>spherical(dados.mode)
(16.6, 3.8)
```

dentre diversos outros. Veja que o último parâmetro chamado *moda*, que equivale ao nó de contagem com maior valor, depende de uma malha de contagem. Por padrão, é criada uma malha de contagem axial, com K calculado a partir da quantidade de dados e espaçamento médio entre os nós de 2,5 graus. Como a análise da malha pode ser demorada, a malha de contagem não é criada pelo *DirectionalData* até que ela seja necessária, armazenando então o resultado obtido.

A malha de contagem é genericamente um objeto da classe *SphericalGrid*, que constrói internamente a malha regular e possui métodos para contagem pelo método de Fisher ou por pontos dentro de um ângulo limite. Adicionalmente é possível operá-la com qualquer função cujos parâmetros sejam uma malha onde ela será calculada e dados para se fazer o cálculo.

Funções adicionais que paralelizam automaticamente a contagem, ou a fazem em um submódulo em Fortran, foram também desenvolvidas. Não são utilizadas por padrão porque o gasto de tempo adicional para se preparar o processamento em paralelo ou se importar as bibliotecas externas, não compensa o ganho de tempo na quantidade típica de dados processados em problemas geológicos. Tornam-se mais interessantes para malhas de altíssima resolução, como separação média de um segundo de arco, ou quantidade de dados muito grande.

Algumas operações com os dados também são possíveis, como a concatenação (*concatenate*), que combina dois *DirectionalData,* e o produto, que calcula as intersecções (*intersect*) entre dois conjuntos de dados (Figura 12), exaustivamente:

```
>>>a = load("a.xlsx")
>>> concatenate(a, b) # OU a + b
<__main__.SphericalDataset at 0xa828080>
>>>resultados = intersect(a, b) # OU a*b
```

Os *DirectionalData* resultantes destas operações herdam os parâmetros adicionais do primeiro *DirectionalData*.

Por fim, dados podem ser projetados em um outro sistema de coordenadas (*project*), ou rotacionados (*rotate*) por um eixo e ângulo definidos pelo usuário (Figura 13). Os

DirectionalData resultantes também herdam dos dados originais os parâmetros adicionais.

```
>>>auttitude.project(dados, (x_linha, y_linha, z_linha))
<__main__.SphericalDataset at 0xa828668>
>>>dados = load("tocher.txt")
>>>dados_rotacionado = rotate(dados, eixo, angulo)
```

Considerando seu uso em ambiente interativo, foram incluídos na classe *DirectionalData* métodos auxiliares para o uso da biblioteca mplstereonet, permitindo sua visualização (Figura 11):

```
>>>import matplotlib.pyplot as plt
>>>import mplstereonet

>>> fig, ax = mplstereonet.subplots()
>>>dados.plot_poles(ax, "bo")
>>>plt.show()
```

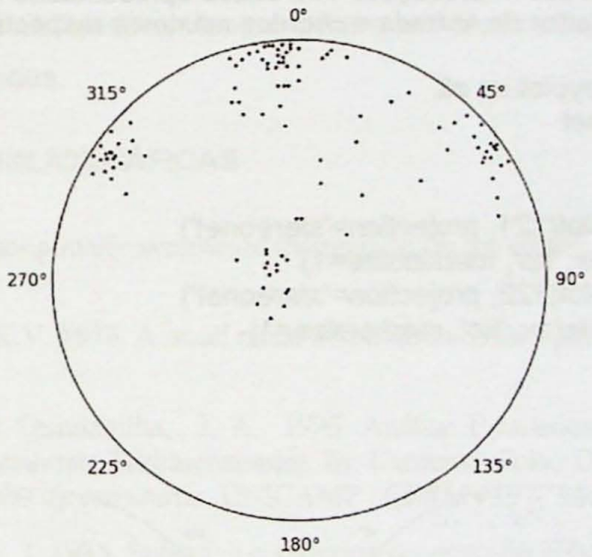

Figura 11 - Plot dos dados em projeção de igual área utilizando a biblioteca mplstereonet adaptada. Os dados utilizados são apresentados no Anexo 10.

```
>>> fig, ax = mplstereonet.subplots()
>>> a.plot_poles(ax, "bo")
>>> a_planes(ax, "b-")
>>> b.plot_poles(ax, "bo")
>>> b.plot_planes(ax, "b-")
>>> resultados.plot_lines(ax, "r+")
>>>plt.show()
```

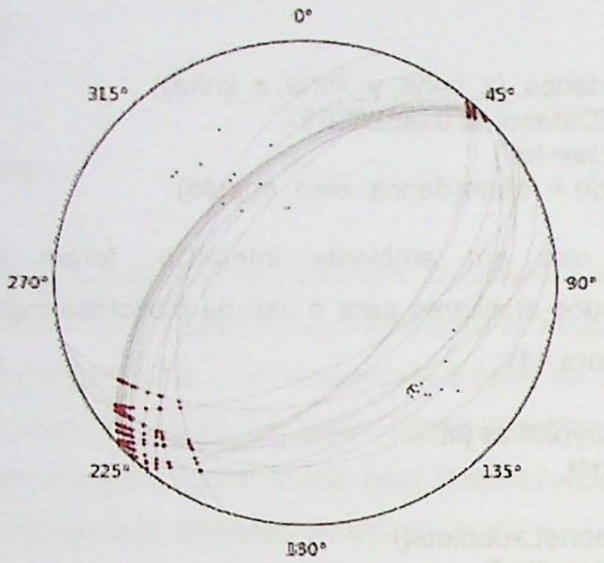

Figura 12 - Resultados das intersecções dos dados apresentados nos anexos 7 e 8, com os dados de entrada e círculos máximos respectivos.

```
>>>import matplotlib.pyplot as plt
>>>import mplstereonet

>>>fig = plt.figure()
>>>ax = fig.add_subplot(121, projection='stereonet')
>>>tocher.plot_pole(ax, 'ko', markersize=1)
>>>ax = fig.add_subplot(122, projection='stereonet')
>>>tocher_rot.plot_pole(ax, 'ko', markersize=1)
>>>plt.show()
```

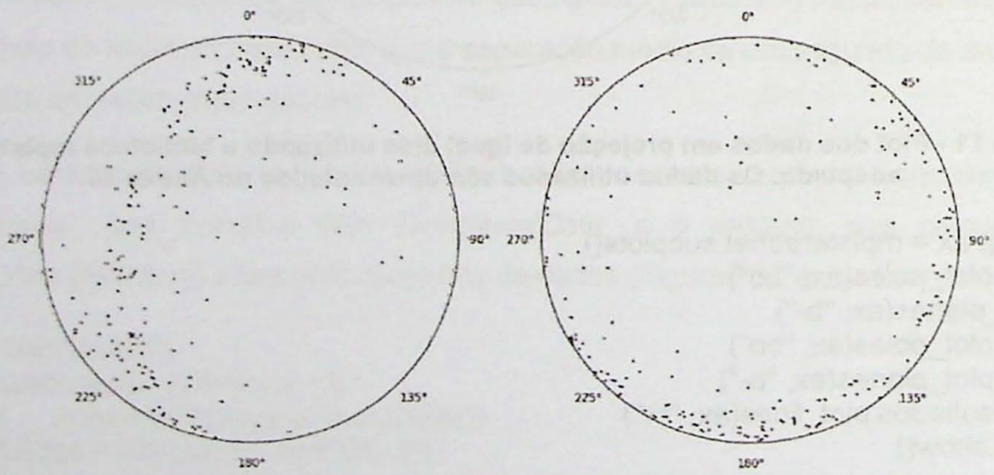

Figura 13 - Exemplo de rotação, usando o eixo 204/01 com rotação de 42,2 graus. Os dados utilizados encontram-se reproduzidos no Anexo 9.

Pode-se também obter as projeções de igual ângulo e igual área dos dados, através dos métodos *equal_angle* e *equal_area,* respectivamente.

Por fim, o uso da biblioteca como *backend* matemático e modelo de armazenamento de dados para o OpenStereo o tornou mais rápido e estável, além de facilitar sua manutenção e extensão, deixando para sua implementação apenas a interface gráfica.

6. CONCLUSÕES

Considera-se que o objetivo principal do trabalho foi atingido. A criação da biblioteca Auttitude, composta por 28 funções e 4 classes, permite que esta seja utilizada para o tratamento de dados direcionais tanto de forma direta como integrado em outras aplicações. Sua integração ao *software* OpenStereo traz vantagens aos dois programas, já que o uso do Auttitude em uma interface gráfica facilita a visualização de conjuntos de dados complexos e simplifica a sua organização, ao mesmo tempo que traz ao OpenStereo velocidade, robustez e facilidade de implementação e manutenção. Os testes realizados tiveram resultados positivos.

O código fonte é disponibilizado em anexo para que futuras adaptações e expansões possam ser desenvolvidas.

7. REFERÊNCIAS BIBLIOGRÁFICAS

Bingham, C., 1974. An antipodally symmetric distribution on the sphere. *Annals of Statistics*, **2:** 1201–1225

Bingham, C. & Mardia, K.V. 1978. A small circle distribution on the sphere. *Biometrika* **65**: 379-389

Campanha, G. A. C., & Quintanilha, J. A.. 1996. Análise Estatística de Dados Estruturais; Estatística de Dados Direcionais Tridimensionais, **In**: Carneiro, Celso Dal Re (editor) *Projeção estereográfica para análise de estruturas,* UNICAMP / CPRM / IPT, São Paulo, pp. 51-58

Diggle, P. J., & Fisher, N. I. 1985. Sphere: a contouring program for spherical data. *Computers & Geosciences*, **11**, 725-766.

Fisher, R. 1953. Dispersion on a sphere. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 217.1130 295-305.

Fisher, N.I., Lewis, T. & Embleton, B.J.J. 1987. *Statistical analysis of spherical data.* Cambridge University Press, 329p.

Fisher, N.I.1995. *Statistical analysis of circular data.* Cambridge University Press, 277p.

Ferguson, J. 1994. *Introduction to linear algebra in geology.* Springer. 224p.

Grohmann, C.H., Campanha, G.A.C. and Soares Junior, A.V., 2011. OpenStereo: um programa Livre e multiplataforma para análise de dados estruturais. **In**: XIII Simpósio Nacional de Estudos Tectônicos, *Atas ...*, Sociedade Brasileira de Geologia Núcleo Centro-Oeste, Cuiabá.

Kamb, W.B. 1959. Ice petrofabric observations from Blue Glacier, Washington, in relation to theory and experiment. *Journal of Geophysical Research* **64:** 1891-1909

Kent, J.T. 1982. The Fisher-Bingham distribution on the sphere. *Journal of the Royal Statistical Society*. **44:** 71-80.

Kim, J. M. (2005). Vectorial formulation of direction cosines for anisotropic geologic structures from their geologic angle measurements. *Mathematical geology*, **37:** 929-941.

Kleene, S.C. 1956. Representation of Events in Nerve Nets and Finite Automata. In: Shannon, Claude E.; McCarthy, John. *Automata Studies*. Princeton University Press. p.: 3–42.

Munro, M.A. & Blenkinsop T.G. 2012. MARD—A moving average rose diagram application for the geosciences. *Computers & Geosciences* **49:** 112-120.

Oliphant, T.E. 2007. Python for Scientific Computing. *Computing in Science & Engineering* **9:** 90

Pearson, K. 1901. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, **2:** 559-572.

Peterson, P. 2009. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering* **4:** 296-305.

Pilgrim, M., & Willison, S. (2009). *Dive Into Python*. Apress. 315 pp.

Python *Software* Foundation 2010. Python, version 2.7: programming language *software*. Reston, Virginia, Zope Corporation.

Robin, P.F., & Jowett E.C. 1986. Computerized density contouring and statistical evaluation of orientation data using counting circles and continuous weighting functions. *Tectonophysics* **121:** 207-223.

Schmidt, W. 1917. Statistische Methoden beim Gefügestudium Kristaliner Schiefer. *Kaiserliche Akademie der Wissenshaften in Wien, Sitzungsberichte, Mathematisch-Naturwissenschaftliche Klasse*. **126:** 515-538

Vollmer, F.W. 1990. An application of eigenvalue methods to structural domain analysis. *Geological Society of America Bulletin* **102:** 786-791.

Watson, G.S. 1965. Equatorial distributions on a sphere. *Biometrika* **52:** 193-201.

Woodcock, N.H. 1977. Specification of fabric shapes using an eigenvalue method. *Geological Society of America Bulletin* **88:** 1231-1236.

8. ANEXOS

ANEXO 1 - auttitude.py

```
import math
import itertools
import os
from csv import Sniffer, reader
sniffer = Sniffer()
import multiprocessing
from multiprocessing import cpu_count, Pipe

import numpy as np
from conversion import Attitude
translator = Attitude()

#set this for experimental multi-core support.
multicore_when_possible = False

#pole_plot_options =
#plane_plot_options =
#line_plot_options =

def dcos(data):
    """Converts poles into direction cossines."""
    theta, phi = np.radians(data.T)
    return np.array((np.sin(phi)*np.sin(theta), np.sin(phi)*np.cos(theta), -
np.cos(phi))).T

def dcos_lines(data):
    """Converts lines into direction cosines."""
    return dcos(invert(data))

def sphere(data):
    """Calculates the attitude of poles direction cossines."""
    x, y, z = data.T
    sign_z = np.copysign(1, z)
    return np.array((np.degrees(np.arctan2(sign_z*x,sign_z*y)) % 360,
np.degrees(np.arccos(np.abs(z))))).T

def sphere_lines(data):
    """Calculate the attitude of lines direction cosines."""
    return invert(sphere(data))

def invert(data):
    """Inverts poles into planes and vice versa."""
    theta, phi = data.T
    return np.array(((theta - 180) % 360, 90 - phi)).T

def RHR(data):
    """Converts data into Right Hand Rule."""
    theta, phi = data.T
    return np.array(((theta - 90) % 360, phi)).T

def equal_angle(data):
    x, y, z = data.T
    return x/(1-z), y/(1-z)
```

```
def equal_area(data):
    x, y, z = data.T
    return x*np.sqrt(2/(1-z)), y*np.sqrt(2/(1-z))

def concatenate(A, B):
    """Concatenate A and B directional datasets, retaining A's additional
attributes"""
    return DirectionalData(np.vstack((A.data, B.data)), *A.args, **A.kwargs)

def intersect(A, B):
    """Calculate all intersections between A and B directional datasets,
retaining A's additional attributes"""
    all_intersections = np.array([np.cross(a, b) for a, b in
itertools.product(A.data, B.data)])
    intersections =
all_intersections[np.nonzero(np.linalg.norm(all_intersections, axis=1))]
    return DirectionalData(intersections, *A.args, **A.kwargs)

def regular_grid(node_spacing):
    """\
Builds a regular grid over the hemisphere, with the given average node
spacing."""
    nodes = [(0., 90.),]
    spacing = math.radians(node_spacing)
    for phi in np.linspace(node_spacing, 90., 90./node_spacing,
endpoint=False):
        azimuth_spacing =
math.degrees(2*math.asin((math.sin(spacing/2)/math.sin(math.radians(phi)))))
        for theta in np.linspace(0., 360., 360./azimuth_spacing):
            nodes.append((theta+phi + node_spacing/2, 90. - phi))
    for theta in np.linspace(0., 360., 360./azimuth_spacing):
        nodes.append(((theta + 90. + node_spacing/2) % 360., 0.))
    return np.array(nodes)

def sphere_regular_grid(node_spacing):
    """\
Builds a regular grid over the sphere, with the given average node
spacing."""
    grid = dcos_lines(regular_grid(node_spacing))
    return np.vstack((grid, -grid))

def universal_loader(filename, extension=None, worksheet=0):
    """\
Loads many different possible file formats, dispatching them to
the proper specific loader."""
    extension = extension or os.path.splitext(filename)[-1] #support for
field names should be added eventually. First row in csv and xlsx, header in
geoeas, field names directly in databases and GIS files.
    if extension in [".csv", ".txt", ".dat"]: #seems ok, for now.
        f = open(filename)
        try:
            #data = f.readlines()
            #geoeas_offset = sniff_geoeas(data)
            dialect = sniffer.sniff(f.read(1024))
            f.seek(0)
        except Error:
            input_data = np.loadtxt(filename)
        else:
            input_data = reader(f, dialect=dialect)
```

```
                #Have to actually read the data inside here. It currently
returns the reader. Well, or not.
                #Maybe return a generator, that would be fun.
            #input_data = np.loadtxt(filename)
    elif extension in [".npy", ".npz"]:
        input_data = np.load(filename)
    elif extension in [".xls", ".xlsx"]:
        from xlrd import open_workbook
        data = open_workbook(filename).sheets()[worksheet]
        input_data =[data.row_values(i) for i in range(data.nrows)]
    return input_data

def universal_translator(data, longitude_column=0, colatitude_column=1,\
                                colatitude=True, dip_direction=False):
    """Translates data from many different notations into dipdirection/dip,
semi-automatically"""
    attitude_data = ((line[longitude_column], line[colatitude_column]) for
line in data)
    converted_data = np.array(translator.process_data(attitude_data,
dd=dip_direction))
    if not colatitude:
        converted_data[:,1] = 90 - converted_data[:,1]
    return converted_data


def load(filename, *args, **kwargs):
    """\
Attempts to automatically load the given filename, using whatever extra
information is
made available by the user, returning a DirectionalData object. See
universal_translator
and universal_loader signatures for additional information. Important
options:
dip_direction, defaults True:
interpret data as dip direction, or strike if set to False.
line, defaults to False:
interpret data as lines, instead of planes."""
    extension = kwargs.get('extension', os.path.splitext(filename)[-1])
    worksheet = kwargs.get('worksheet', 0)
    input_data = universal_loader(filename, extension=extension,
worksheet=worksheet)
    dip_direction = kwargs.get('dip_direction', True)
    line = kwargs.get('line', False)
    longitude_column = kwargs.get('strike_column', 0)
    colatitude_column = kwargs.get('dip_column', 1)
    converted_data = universal_translator(input_data,
                        longitude_column=longitude_column,
                        colatitude_column=colatitude_column,
                        colatitude=line,
                        dip_direction=dip_direction,)
    if not line:
        converted_data = invert(converted_data)
    return DirectionalData(dcos(converted_data),*args, **kwargs)


def calculate_axes(data):
    """Calculates the eigenvectors and eigenvalues of the dispersion matrix
of the dataset."""
    dispersion_tensor = np.cov(data.T[:3, :])
```

```
    eigenvalues, eigenvectors = np.linalg.eigh(dispersion_tensor, UPLO='U')
    eigenvalues_order = eigenvalues.argsort()[::-1]
    eigenvectors = eigenvectors[:,eigenvalues_order].T
    return eigenvectors, eigenvalues

def rotation_matrix(u, theta):
    #From openstereo development notes,
    #from http://stackoverflow.com/questions/6802577/python-rotation-of-3d-
vector
    #Using the Euler-Rodrigues formula:
    #http://en.wikipedia.org/wiki/Euler%E2%80%93Rodrigues_parameters
    """
    Return the rotation matrix associated with counterclockwise rotation
about
    the given axis u by theta degrees.
    """
    u = np.asarray(u)
    theta = math.radians(theta)
    u = u/math.sqrt(np.dot(u, u))
    a = math.cos(theta/2)
    b, c, d = -u*math.sin(theta/2)
    aa, bb, cc, dd = a*a, b*b, c*c, d*d
    bc, ad, ac, ab, bd, cd = b*c, a*d, a*c, a*b, b*d, c*d
    return np.array([[aa+bb-cc-dd, 2*(bc+ad), 2*(bd-ac)],
                     [2*(bc-ad), aa+cc-bb-dd, 2*(cd+ab)],
                     [2*(bd+ac), 2*(cd-ab), aa+dd-bb-cc]])


def rotate(data, u, theta):
    return DirectionalData(np.dot(data.data, rotation_matrix(u, theta)),
*data.args, **data.kwargs)

def project(data, new_axes):
    return DirectionalData(np.dot(data.data, new_axes.T), *data.args,
**data.kwargs)

class DirectionalData(object):
    def __init__(self, data, calculate_statistics=True, *args, **kwargs):
        """\
        Base class for directional data analysis, either 2d or 3d. Store
optionally
        additional arguments for plotting."""
        self.args, self.kwargs = args, kwargs
        self.data = data
        self.data_sphere = kwargs.get('data_sphere', None)
        if self.data_sphere is None:
            self.data_sphere = sphere(data/np.linalg.norm(data, axis=1)[:,
np.newaxis])
        self.n, self.d = data.shape

        if calculate_statistics: self.initialize_statistics()
        self._grid = None

    def initialize_statistics(self):
        self.resultant_vector = np.sum(self.data, axis=0)
        self.mean_resultant_vector = self.resultant_vector/self.n
        self.resultant_length = np.linalg.norm(self.resultant_vector)
        self.mean_resultant_length = self.resultant_length/self.n
```

```
        if self.d == 2:
            self.circular_variance = 1 - self.mean_resultant_length
            self.circular_standard_deviation = math.sqrt(-2*math.log(1 -
self.circular_variance))
        elif self.d == 3:
            self.resultant_vector_sphere = sphere(self.resultant_vector)
            self.fisher_k = (self.n - 1)/(self.n -
np.linalg.norm(self.resultant_vector))
            direction_tensor = np.dot(self.data.T, self.data)/self.n
            eigenvalues, eigenvectors = np.linalg.eigh(direction_tensor)
            eigenvalues_order = (-eigenvalues).argsort()

            self.eigenvalues = eigenvalues[eigenvalues_order]
            self.eigenvectors = eigenvectors[:,eigenvalues_order].T
            self.eigenvectors_sphere = sphere_lines(self.eigenvectors)

            #From Vollmer 1990
            self.vollmer_P = (self.eigenvalues[0] -
self.eigenvalues[1])/eigenvalues.sum()
            self.vollmer_G = 2*(self.eigenvalues[1] -
self.eigenvalues[2])/eigenvalues.sum()
            self.vollmer_R = 3*self.eigenvalues[2]/eigenvalues.sum()

            self.vollmer_classification = ("point", "girdle", "random")[\
                np.argmax((self.vollmer_P,self.vollmer_G, self.vollmer_R))]

            self.vollmer_B = self.vollmer_P + self.vollmer_G
            self.vollmer_C =
math.log(self.eigenvalues[0]/self.eigenvalues[2])

            #From Woodcock 1977
            self.woodcock_Kx =
math.log(self.eigenvalues[1]/self.eigenvalues[2])
            self.woodcock_Ky =
math.log(self.eigenvalues[0]/self.eigenvalues[1])
            self.woodcock_C =
math.log(self.eigenvalues[0]/self.eigenvalues[2])

            self.woodcock_K = self.woodcock_Ky / self.woodcock_Kx
    def __add__(self, other):
        """Concatenate A and B directional datasets, retaining A's additional
attributes"""
        return concatenate(self, other)
    def __mul__(self, other):
        """Calculate all intersections between A and B directional datasets,
retaining A's additional attributes"""
        return intersect(self, other)
    @property
    def grid(self):
        if self._grid is None:
            self._grid = SphericalGrid(**self.kwargs)
        return self._grid
    @property
    def grid_nodes(self):
        return self.grid.grid
    def grid_fisher(self, k=None):
        return self.grid.count_fisher(self, k)
    def grid_kamb(self, theta=None):
        return self.grid.count_kamb(self, theta)
```

```
    def __repr__(self):
        return "%s(%s, *%s, **%s)" %  (self.__class__, self.data, self.args,
self.kwargs)
    def __str__(self):
        if self.d == 2:
            factor = self.kwargs.get('simm_factor', 1)
            return """\
{filename}\
Number of data lines:
{self.n}
Resultant {datatype}:
{resultant}
Resultant Length:
{self.resultant_length}
Mean Resultant Length :
{self.mean_resultant_length}

Circular Variance:
{self.circular_variance}
Circular Standard Deviation:
{self.circular_standard_deviation}\
""".format(self=self, filename=self.kwargs.get('filename', ''),
          datatype=self.kwargs.get('datatype', 'Azimuth'),
          resultant=math.degrees(math.atan2(*self.resultant_vector)))/factor
        elif self.d == 3:
            return """\
{filename}\
Number of data lines= {self.n}
Expected Distribution:
{self.vollmer_classification}
Eigenvectors:
1: {self.eigenvectors_sphere[0]}
2: {self.eigenvectors_sphere[1]}
3: {self.eigenvectors_sphere[1]}
Shape parameter
K = {self.woodcock_K}
Strength parameter
C = {self.woodcock_C}
Normalized Eigenvalues:
S1: {self.eigenvalues[0]}
S2: {self.eigenvalues[1]}
S3: {self.eigenvalues[2]}
Fabric (triangular diag.):
Point  = {self.vollmer_P}
Girdle = {self.vollmer_G}
Random = {self.vollmer_R}
Cilindricity = {self.vollmer_C}\
""".format(self=self, filename=self.kwargs.get('filename', ''))
    def plot_pole(self, ax, symbol=None, **plot_kwargs):
        """Plot data as poles to planes"""
        if self.d == 3:
            strike, dip = RHR(self.data_sphere).T
            #strike = (dip_direction - 90) % 360
            if symbol is None:
                self.kwargs.get('pole_symbol', 'ko')
            if not plot_kwargs:
                plot_kwargs = self.kwargs.get('pole_plot_options', {})
            ax.pole(strike, dip, symbol, **plot_kwargs)
```

```
    def plot_plane(self, ax, symbol=None, **plot_kwargs):
        """Plot data as great circles"""
        if self.d == 3:
            strike, dip = RHR(self.data_sphere).T
            #strike = (dip_direction - 90) % 360
            if symbol is None:
                self.kwargs.get('plane_symbol', 'k-')
            if not plot_kwargs:
                plot_kwargs = self.kwargs.get('plane_plot_options', {})
            ax.plane(strike, dip, symbol, **plot_kwargs)
    def plot_line(self, ax, symbol=None, **plot_kwargs):
        """Plot data as lines"""
        if self.d == 3:
            trend, plunge = self.data_sphere.T
            #strike = (dip_direction - 90) % 360
            if symbol is None:
                self.kwargs.get('line_symbol', 'k+')
            if not plot_kwargs:
                plot_kwargs = self.kwargs.get('line_plot_options', {})
            ax.line(trend, plunge, symbol, **plot_kwargs)
    # def plot_circle(self, ax, symbol=None, **plot_kwargs):
    #   pass

class PartProcessor(multiprocessing.Process):
    def run(self):
        connection = self._kwargs.pop("connection")
        connection.send(self._target(*self._args, **self._kwargs))
        connection.close()

def parallel(function): #there should be faster and simpler ways to do this,
gosh. It works, though.
    """\
A parallelization decorator for simple functions that evaluate over a grid,
splitting it's first dimension among the available cores."""
    core_count = cpu_count()
    if core_count < 2: return function
    def parallel_function(grid, *args, **kwargs):
        output = []
        cores = []
        grid_size = grid.shape[0]
        grid_section = int(grid_size/core_count)
        for n in range(core_count - 1):
            server_p, client_p = Pipe()
            core = PartProcessor(grid[n*grid_section:(n+1)*grid_section,:],
target = function, *args, **kwargs)
            core.start()
            cores.append(core)
        core = PartProcessor(grid[(core_count - 1)*grid_section:-1,:], target
= function, *args, **kwargs)
        core.start()
        cores.append(core)
        for core in cores:
            output.append(core.recv())
            core.close()
        return np.vstack(output)
    return parallel_function

def parallel_counter(counter_factory): #yes, a factory decorator!
    """\
```

```
A factory decorator that applies @parallel on the functions returned by
the decorated factory. See parallel for more details."""
    def parallel_counter_factory(*args, **kwargs):
        if multicore_when_possible:
            return parallel(counter_factory(*args, **kwargs))
        else:
            return counter_factory(*args, **kwargs)
    return parallel_counter_factory

@parallel_counter
def FisherCounter(k):
    try:
        from grid_functions.fisher_counter import count
        def counter(grid, direction_cosines):
            return count(grid, direction_cosines, k)
    except ImportError:
        def counter(grid, direction_cosines):
            try:
                return np.exp(k*(np.dot(grid, direction_cosines.T) -
1)).sum(axis=1)
            except MemoryError:
                result = np.zeros((grid.shape[0],1))
                for input_node, output_node in zip(grid, result):
                    output_node[:] = np.exp(k*(np.dot(input_node,
direction_cosines.T) - 1)).sum()
    return counter

@parallel_counter
def FisherCounterAxial(k):
    try:
        from grid_functions.fisher_counter_axial import count
        def counter(grid, direction_cosines):
            return count(grid, direction_cosines, k)
    except ImportError:
        def counter(grid, direction_cosines):
            try:
                return np.exp(k*(np.abs(np.dot(grid, direction_cosines.T)) -
1)).sum(axis=1)
            except MemoryError:
                result = np.zeros((grid.shape[0],1))
                for input_node, output_node in zip(grid, result):
                    output_node[:] = np.exp(k*(np.abs(np.dot(input_node,
direction_cosines.T)) - 1)).sum()
    return counter

@parallel_counter
def RobinGirdleCounter(k):
    try:
        from grid_functions.robin_girdle_counter import count
        def counter(grid, direction_cosines):
            return count(grid, direction_cosines, k)
    except ImportError:
        def counter(grid, direction_cosines):
            try:
                return np.exp(k*(np.dot(grid,
direction_cosines.T)**2)).sum(axis=1)
            except MemoryError:
                result = np.zeros((grid.shape[0],1))
                for input_node, output_node in zip(grid, result):
```

```
                    output_node[:] = np.exp(k*(np.dot(input_node,
direction_cosines.T)**2)).sum()
    return counter

class SphericalGrid(object):
    def __init__(self, node_spacing=None, *args, **kwargs):
        """Creates a spherical counting grid"""
        self.args, self.kwargs = args, kwargs
        if node_spacing is None:
            node_spacing = self.kwargs.get('node_spacing', 2.5)
        self.grid_nodes = regular_grid(node_spacing)
        self.grid = dcos(self.grid_nodes)

    def count_fisher(self, data, k=None):
        """\
Performs data counting as in Robin and Jowett (1986). May either receive
as input a DirectionalData object or any numpy array-like. Will guess an
appropriate
k if not given and not available from the DirectionalData options."""
        if isinstance(data, DirectionalData):
            k = k or data.kwargs.get('counting_k', None)
            n = data.n
            direction_cosines = data.data
        else:
            direction_cosines =  data
            n = data.shape[0]
        if k is None:
            if n < 100: #This is the Recomendation made by Robin & Jowett 86
                k = 2*(n + 1)
            else:
                k = 100
        try: #It is bettter to beg forgiveness that ask for permittion.
            self.result = np.exp(k*(np.abs(np.dot(self.grid,
 direction_cosines.T)) - 1)).sum(axis=1)
            return self.result
        except MemoryError:
            result = np.zeros((self.grid.shape[0],1))
            for input_node, output_node in zip(self.grid, result):
                output_node[:] = np.exp(k*(np.abs(np.dot(input_node,
 direction_cosines.T)) - 1)).sum()
        self.result = result
        return result
    def count_kamb(self, data, theta=None):
        """\
 Performs data counting as in Robin and Jowett (1986) based on Kamb (1956),
 May either receive
 as input a DirectionalData object or any numpy array-like. Will guess an
 appropriate
 counting angle theta if not given and not available from the DirectionalData
 options."""
        if isinstance(data, DirectionalData):
            theta = theta or data.kwargs.get('counting_theta', None)
            n = data.n
            direction_cosines = data.data
        else:
            direction_cosines =  data
            n = data.shape[0]
        if theta is None:
            theta = (n-1)/(n+1)
```

```
        else:
            theta = math.cos(math.radians(theta))
        try:
            self.result = (np.abs(np.dot(self.grid, direction_cosines.T)) <
theta).sum(axis=1)
        except MemoryError:
            result = np.zeros((self.grid.shape[0],1))
            for input_node, output_node in zip(self.grid, result):
                output_node[:] = (np.abs(np.dot(input_node,
direction_cosines.T)) < theta).sum()
        self.result = result
        return result
    def count(self, data, method=None):
        """\
If method isn't given, search data for it. If method is a function, execute
it with the counting grid
and the data object as parameters, or search SphericalGrid for it, in case it
is a string."""
        if isinstance(data, DirectionalData):
            method = method or data.kwargs.get('counting_method', None)
            direction_cosines = data.data
        else:
            direction_cosines = data
        if not method is None:
            if isinstance(method, basestring):
                return self.__getattribute__(method)(direction_cosines)
            else:
                return method(self.grid, direction_cosines)
```

ANEXO 2 - conversion.py

```
#Not tremendously stupid converter for orientation data. As of now,
#hopefully will be able to convert from strike/dip+dipquadrant, whether
#the strike is in azimuth or quadrant notation.
#We can but hope.
#Ver 0.5.0
import re

from collections import defaultdict

attitude_parser = re.compile("([NSEW]{0,2})(\d*)([NSEW]{0,2})[^NSEW0-
9](\d+)([NSEW]{0,2})", re.IGNORECASE)
azimuth_parser = re.compile("([NSEW]{0,2})(\d*)([NSEW]{0,2})", re.IGNORECASE)
dip_parser = re.compile("(\d+)([NSEW]{0,2})", re.IGNORECASE)


def parse_quadrant(leading, number, trailing):
    base, multiplier = quadrants[(leading, trailing)]
    return base + number*multiplier

def parse_strike_quadrant(strike, dip_quadrant):
    return strike + dip_quadrants[((strike%180)//90,
trends[dip_quadrant//90])]

p = re.compile('(N?)(\d*)([EW ]?).*', re.IGNORECASE)
class Attitude(object):
```

```
    """Class to transcode attitude data"""
    #Dispatcher truth table. There oughta be an eleganter way to do this,
though.
    strike_coding = {(True, True, True): 'quadrant',
                     (True, True, False): 'azimuth',
                     (True, False, True): 'cardinal',
                     (True, False, False): 'cardinal',
                     (False, True, True): 'dip',
                     (False, True, False): 'azimuth',
                     (False, False, False): 'error',
                     (False, False, True): 'error'}

    #Constants for trike processing for quadrants. Maybe should reposition
this.
    strike_leading = {'N': 0, 'S': 180}
    strike_trailing = {'E': 1, 'W': -1}
    strike__leading_multiplier = {'N': 1, 'S': -1}
    #quads = {"NE":-90,
    #        "SE":90,
    #        "SW":90,
    #        "NW":-90}
    #Constants for conversion based on dip, with a default value
    quadrants = defaultdict(lambda: 90)
    quadrants.update(quads = {"NE":-90, #ALWAYS REMEMBER: UPDATE RETURNS
NOTHING
                              "SE":90,
                              "SW":90,
                              "NW":-90})

    #are regex matches fast? They seem to be...
    strike_pattern = re.compile('([NS]?)(\d*)([EW ]?).*', re.IGNORECASE)
#Regex parser for strike, separating the constituent letters and numbers
    dip_pattern = re.compile('(\d*)([NESW]*).*', re.IGNORECASE) #Regex parser
for dip, separating the dip from the dip direction quadrant, if present

    def process_data(self, attitude, dd=True):
        #print strike, dip
        self.data = []
        dd = (dd and 90.0) or 0.0
        for strike, dip in attitude:
            self.leading_letter, self.number, self.tailing_letter =
Attitude.strike_pattern.match(strike).groups()
            #coding = Attitude.strike_coding[(bool(self.leading_letter),
bool(self.number), bool(self.tailing_letter))]
            #Dispatch the strike and dip combination to the correct parsing
function.
            dip_direction, dip = self.process_strikedip(strike, dip)
            dip_direction = (dip_direction - dd) % 360
            self.data.append((dip_direction, dip))
        return self.data

    def do_quadrant(self, strike, dip):
        dip, dip_direction_quadrant =
Attitude.dip_pattern.match(dip).groups()
        if not dip:
            return 'err', 'err'
#       elif not dip_direction_quadrant:
#           return ((Attitude.strike_leading[self.leading_letter.upper()] +
Attitude.strike_trailing[self.trailing_letter.upper()]*strike__leading_multip
```

```
lier[self.leading_letter.upper()]*int(self.number)) % 360 + 90) % 360,
int(dip)
        return ((Attitude.strike_leading[self.leading_letter.upper()] +
Attitude.strike_trailing[self.trailing_letter.upper()]*Attitude.strike__leadi
ng_multiplier[self.leading_letter.upper()]*int(self.number)) % 360 +
Attitude.quadrants[dip_direction_quadrant.upper()]) % 360, int(dip)
        #pass

    def do_azimuth(self, strike, dip):
        dip, dip_direction_quadrant =
Attitude.dip_pattern.match(dip).groups()
        if not dip:
            return 'err', 'err'
        return (int(self.number) +
Attitude.quadrants[dip_direction_quadrant]) % 360, int(dip)
        #pass

    def do_cardinal(self, strike, dip):
        dip, dip_direction_quadrant =
Attitude.dip_pattern.match(dip).groups()
        return (Attitude.strike_leading[self.leading_letter.upper()] -
Attitude.strike_trailing[dip_direction_quadrant.upper()]*90) % 360.0, dip
        #pass

    def do_dip(self, dip, strike):
        #had to do somewhat of a hack... Better way, anyone?
        if reduce(lambda x, y: bool(x and y),
Attitude.strike_pattern.match(strike).groups()): #So as to prevent an
infinite recursion, there might be a better way to do this.
            return self.process_strikedip(strike, dip)
        else:
            return 'err', 'err'

    def do_error(self, strike, dip):
        return 'err', 'err'

    def process_strikedip(self, strike, dip):
        self.leading_letter, self.number, self.trailing_letter =
Attitude.strike_pattern.match(strike).groups()
        coding = Attitude.strike_coding[(bool(self.leading_letter),
bool(self.number), bool(self.tailing_letter))]
        #Dispatch the strike and dip combination to the correct parsing
function.
        #if __debug__: print strike.upper(), dip.upper()
        return getattr(self, "do_%s" % coding)(strike.upper(), dip.upper())
```

ANEXO 3 - grid_functions/count_zero.f

```
        subroutine count(grid, points, cosin, ngrid, npoints, dims)
        integer ngrid, npoints, dims
        real*8 grid(0:ngrid-1,0:dims-1), points(0:npoints-1,0:dims-1)
        real*8 cosin(0:ngrid-1,0:npoints-1)
Cf2py intent(out) cosin
Cf2py intent(in) grid
Cf2py intent(in) points
Cf2py depend(ngrid,npoints,dims) cosin

        integer i,j
```

```
        do i = 0, ngrid-1
            do j = 0, npoints-1
            cosin(i,j) = dot_product(grid(i,:), points(j,:))
            enddo
        enddo
        return
        end
```

ANEXO 4 - grid_functions/fisher_counter.f

```
        subroutine count(total,grid,points,kappa,ngrid,npoints,dims)
        integer ngrid, npoints, dims
        real*8 grid(0:ngrid-1,0:dims-1), points(0:npoints-1,0:dims-1)
        real*8 total(0:ngrid-1)
        real*8 z, kappa
Cf2py intent(in) grid
Cf2py intent(in) points
Cf2py intent(in) kappa
Cf2py intent(out) total
Cf2py depend(ngrid) total
        integer i,j
        total=0
        do i = 0, ngrid-1
            z = 0.0
            do j = 0, npoints-1
            z = z + exp(kappa*(dot_product(grid(i,:),
     +      points(j,:))-1))
            enddo
            total(i) = z
        enddo
        return
        end
```

ANEXO 5 - grid_functions/fisher_counter_axis.f

```
        subroutine count(total,grid,points,kappa,ngrid,npoints,dims)
        integer ngrid, npoints, dims
        real*8 grid(0:ngrid-1,0:dims-1), points(0:npoints-1,0:dims-1)
        real*8 total(0:ngrid-1)
        real*8 z, kappa
Cf2py intent(in) grid
Cf2py intent(in) points
Cf2py intent(in) kappa
Cf2py intent(out) total
Cf2py depend(ngrid) total
        integer i,j
        total=0
        do i = 0, ngrid-1
            z = 0.0
            do j = 0, npoints-1
            z = z + exp(kappa*(abs(dot_product(grid(i,:),
     +      points(j,:)))-1))
            enddo
            total(i) = z
        enddo
        return
        end
```

ANEXO 6 - grid_functions/robin_girdle_counter.f

```
        subroutine count(total,grid,points,kappa,ngrid,npoints,dims)
        integer ngrid, npoints, dims
        real*8 grid(0:ngrid-1,0:dims-1), points(0:npoints-1,0:dims-1)
        real*8 total(0:ngrid-1)
        real*8 z, kappa
Cf2py intent(in) grid
Cf2py intent(in) points
Cf2py intent(in) kappa
Cf2py intent(out) total
Cf2py depend(ngrid) total
        integer i,j
        total=0
        do i = 0, ngrid-1
            z = 0.0
            do j = 0, npoints-1
            z = z + exp(kappa*(dot_product(grid(i,:),
     +      points(j,:))**2))
            enddo
            total(i) = z
        enddo
        return
        end
```

ANEXO 7 - família_a.txt

```
315.3       50.54
316.22      51.18
316.19      50.32
317.39      50.29
318.4       51.76
314.69      51.4
315.32      53.12
318.6       53.89
314.65      56.51
312.15      52.64
315.3       50.11
316.26      52.47
300.32      54.25
309.17      59.72
313.92      57.35
306.87      64.0
288.29      52.85
```

ANEXO 8 - família_b.txt

```
157.96      37.85
140.65      32.0
148.34      44.03
145.21      43.01
140.97      54.41
167.47      49.56
168.74      26.67
135.47      38.91
130.83      58.96
152.4       61.26
132.03      58.37
```

ANEXO 9 - tocher.txt

#dip_dir dir

147.704 64.813
113.663 50.462
126.587 44.613
185.946 72.617
182.887 72.158
117.096 53.585
106.355 50.980
088.821 79.496
118.418 65.046
180.374 75.115
002.590 61.473
186.324 76.550
073.251 65.166
073.075 55.372
184.012 72.959
054.920 86.960
149.582 65.623
159.197 71.321
154.722 64.041
195.160 88.091
165.847 71.867
200.319 89.681
191.572 88.257
115.686 53.725
119.646 53.953
107.687 47.648
197.313 81.262
018.910 81.262
186.295 77.261
170.073 63.964
193.162 85.419
017.958 86.119
050.066 79.353
051.529 55.798
183.888 89.809
022.684 88.927
022.847 67.354
035.434 72.350
152.535 50.912
050.636 69.432
019.931 81.029
041.512 71.017
061.994 67.319
165.805 59.771
096.176 57.875
168.413 63.495
062.166 34.297
035.197 78.540
113.356 27.666
057.701 75.725
207.980 86.867
164.107 63.076
026.724 86.259
174.334 62.995
177.443 84.191
048.417 62.431
047.913 65.683
038.494 73.840
098.498 54.470
171.121 66.100
106.525 51.247
242.519 71.620
171.796 57.536
184.788 81.977
202.191 86.461
013.365 87.228
032.958 74.399
147.061 65.965
143.730 53.312
010.308 70.091

145.370 57.206
158.877 69.612
265.256 35.543
043.428 84.384
098.266 51.915
097.822 43.252
071.013 79.479
056.093 51.210
178.973 82.155
068.170 58.115
193.314 58.030
218.444 42.082
073.681 49.974
050.558 62.086
164.928 44.062
010.509 89.496
114.927 52.349
173.027 75.558
048.600 75.027
054.556 83.251
028.714 83.329
179.097 77.346
206.759 81.959
189.796 85.761
062.805 84.281
081.532 54.172
182.726 73.641
172.438 70.130
209.246 82.435
195.288 85.264
050.907 73.201
085.990 57.503
203.828 79.360
186.366 79.135
152.194 60.690
114.351 32.094
117.694 39.905
151.686 38.504
046.208 75.328
190.187 33.427
184.025 74.412
166.945 82.383
079.147 87.256
080.213 59.694
026.953 85.030
227.387 71.399
077.225 56.545
033.882 74.621
093.408 43.321
029.687 70.825
065.182 76.147
050.685 63.673
039.644 70.137
045.481 62.365
170.814 81.379
023.895 63.795
233.271 77.470
136.735 38.706
165.968 64.078
187.254 59.624
046.060 69.051
195.918 89.476
034.020 77.271
145.704 53.997
023.234 86.074
178.647 74.555
042.615 77.935
235.232 37.604
034.457 56.617
074.546 48.892

```
181.248 89.241
180.138 76.790
277.718 62.909
029.092 86.475
171.900 71.472
349.540 86.893
037.460 82.650
081.254 87.107
058.418 68.879
061.123 60.037
069.092 41.819
117.138 54.189
043.721 66.989
111.762 58.043
149.894 57.619
076.044 54.411
061.195 85.126
358.275 81.410
138.966 8.899
093.954 49.118
130.246 34.754
280.296 54.802
073.719 53.493
016.698 87.604
191.761 81.048
193.082 72.053
015.233 85.012
36.658 70.868
082.952 52.051
029.841 49.790
037.979 81.407
048.625 60.323
180.669 55.225
035.159 30.706
241.215 43.754
185.154 70.962
170.722 72.476
126.284 61.304
063.119 80.054
051.368 63.618
053.725 72.659
022.326 86.315
195.305 86.282
053.241 36.802
232.124 41.108
173.320 74.837
169.298 63.663
164.828 61.953
184.352 56.685
050.775 64.940
072.223 50.303
171.455 65.730
249.528 40.938
034.971 72.316
177.362 78.392
092.586 13.522
180.257 21.653
090.249 43.486
134.942 49.096
110.946 51.057
```

ANEXO 10 - frat.dat

```
162 74sw
152 78ne
150 84ne
150 88ne
147 78ne
147 82ne
```

146 80ne
146 86ne
145 88ne
145 82ne
145 86ne
144 86ne
143 80ne
143 90ne
140 82ne
139 88ne
138 70ne
136 22ne
135 86ne
134 80ne
126 36ne
092 21ne
100 42ne
112 52ne
120 79ne
117 72ne
120 79ne
117 72ne
132 74ne
140 54ne
157 76ne
n72e 62nw
n70e 68se
n72e 67se
n74e 82se
n80e 90se
n82e 78se
n81e 82se
n85e 54se
n85e 80se
n85e 86se
n86e 82se
n86e 88se
n87e 78se
n87e 21se
n87e 70se
n88e 90se
n33e 82se
n90e 78se
n90e 72se
98 86sw
97 84sw
96 82sw
92 90sw
110 62sw
102 34sw
102 84sw
n72e 73nw
n29e 76nw
n28e 84nw
n27e 68nw
n30e 80nw
n32e 78nw
n32e 88nw
n32e 84nw
n33e 82nw
n34e 76nw
n34e 80nw
n35e 83nw
n35e 80nw
n35e 82nw
n35e 82nw
n35e 88nw
n38e 76nw
n38e 82nw
n34e 84nw

n39e 88nw
n40e 70nw
n41e 78nw
n44e 84nw
n73e 86nw
n80e 86nw
n82e 80nw
n82e 88nw
n78e 86nw
n83e 88nw
n84e 70nw
n84e 70nw
n84e 78nw
n84e 62nw
n85e 83nw
n84e 88nw
n86e 70nw
n86e 70nw
n86e 80nw
n76e 84nw
n86e 88nw
n76e 84nw
n86e 88nw
n87e 78nw
n87e 84nw
n88e 88nw
n88e 86nw
n90e 86ne
n78e 60nw
n20e 12se
n34e 09se
n54e 10se
n43w 05sw
n44w 06sw
n60e 08se
n18e 22se
n53w 16ne
n43e 15nw
n60w 10ne
n22e 08se
n30w 16sw
n40e 14nw
n30e 12nw
n10e 10nw
n12w 12ne