**UFRN – Universidade Federal do Rio Grande do Norte**

**CCET – Centro de Ciências Exatas e da Terra**

**DIMAP – Departamento de Informática e Matemática Aplicada**

**Disciplina: Calculo Numérico para as Ciências da Computação**

**Professor: Luiz Amorim Carlos**

**Detalhe Técnico dos Algoritmos da Primeira Unidade da disciplina de Cálculo Numérico**

**Aluno: Vítor Alcântara de Almeida**

**Matrícula: 2008018563**

**Natal, Março/Abril de 2010**

**1 – Algoritmos do Trabalho**

1. **Algoritmo Briot-Ruffine\_inteiro**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **public** **static** **int** briot\_Ruffine\_Inteiro(String binaryValue)  {  **int** valor = *getIntegerFromChar*(binaryValue.charAt(0));  **for** (**int** i = 1 ; i < binaryValue.length() ; i++ )  {  valor = *getIntegerFromChar*(binaryValue.charAt(i)) + 2 \* valor; }  **return** valor;  } |

Parâmetro de entrada:

*binaryValue* – Representação em String de um número binário inteiro.

Retorno da função:

O número na base decimal equivalente ao *binaryValue*

Este algoritmo percorre o número binário da esquerda para a direita. Ele armazena em *valor* o algarismo da primeira posição, e, da segunda até a última posição, *valor* é multiplicado por 2 e somado com o elemento da posição corrente. Depois, *valor* é retornado ao usuário, que possui o número na base decimal. A função *getIntegerFromChar* transforma o caractere em uma posição do número binário (que está em String) no algarismo equivalente. A função *charAt(i)* retorna o caractere da posição ‘i’ do objeto que o chamou (no caso, *binaryValue*).

1. **Algoritmo Briot-Ruffine\_fração**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **public** **static** **float** briot\_Ruffine\_Fracao(String binaryValue)  {  **float** valor = *getIntegerFromChar*(binaryValue.charAt(binaryValue.length()-1));  **for**(**int** i = binaryValue.length()-2; i >= 0 ; i--)  {  valor = *getIntegerFromChar*(binaryValue.charAt(i)) + valor/2;  }  valor = valor/2;  **return** valor;  } |

Parâmetro de entrada:

*binaryValue* – Representação em String de um número fracionário maior que 0 e menor que 1.

Retorno da função:

O número na base decimal equivalente ao *binaryValue*.

Este algoritmo percorre o número binário da direita para a esquerda. Ele armazena em *valor* o algarismo da última posição, e, da penúltima até a primeira posição, *valor* é dividido por 2 e somado com o elemento da posição corrente. Depois, *valor* é retornado ao usuário, que possui o número na base decimal.

1. **Algoritmo divisão\_sucessiva**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11** | **public** **static** String divisao\_sucessiva(**int** decimal)  {  String binaryRepresentation = "";  **while**(decimal != 0)  {  **int** value = decimal % 2;  binaryRepresentation = String.*valueOf*(value) + binaryRepresentation;  decimal = decimal / 2;  }  **return** binaryRepresentation;  } |

Parâmetro de entrada:

*decimal* –Número inteiro decimal

Retorno da função:

A representação em String do número binário equivalente a *decimal*.

Este algoritmo divide sucessivas vezes *decimal* por 2 para se obter o equivalente na base binária. É calculado o resto da divisão de *decimal* por dois, e esse resto é armazenado à esquerda do valor atual de *binaryRepresentation* (como uma pilha onde se insere pela esquerda), depois, decimal assume o valor da divisão dele por 2 e repete-se a operação anterior, até que o valor de *decimal* seja 0. Depois é retorando *binaryRepresentation*, que contém a representação em String de *decimal.*

1. **Algoritmo multiplicação\_sucessiva**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16** | **public** **static** String multiplicacao\_sucessiva(**double** fracao)  {  String binaryRepresentation = "";  **if**(fracao == 0)  {  **return** "0";  }  **while**(fracao != 0 && binaryRepresentation.length() < 2000)  {  fracao = fracao \* 2;  **int** decimal = (**int**)fracao;  binaryRepresentation = binaryRepresentation + String.*valueOf*(decimal);  fracao = fracao - decimal;  }  **return** binaryRepresentation;  } |

Parâmetro de entrada:

*fracao* –Número real maior que 0 e menor que 1.

*Limite –* Tamanho máximo da representação binária.

Retorno da função:

A representação em String do número binário equivalente a *fracao*.

Este algoritmo multiplica sucessivas vezes fracaopor 2 para se obter o equivalente na base binária. Enquanto *fração* for diferente de 0 ou o tamanho da representação binária for menor que *limite* faz-se as seguintes operações:

1. Multiplica-se *fracao* por 2 (linha 6)
2. Insere em *binaryRepresentation* pela direita o piso da multiplicação (linhas 11 e 12)
3. *fracao* recebe a parte fracionária do valor encontrado na operação 1.

Retorna-se depois binaryRepresentation, que possui a representação em binário do número fracionário (*fracao)*.

1. **Fatoração LU e Fatoração LU com pivotamento parcial**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34** | **public** **static** **double[][]** fatoracaoLU(**double**[][] a,**double**[][] b,**boolean** comPivotamentoParcial)  {  **double**[][] u = a;  **double**[][] linv = **new** **double**[a.length][a[0].length];  *setIdentidadeToMatriz*(linv);  **double**[][] m = **new** **double**[a.length][a[0].length];  **for**(**int** i = 0 ; i < a.length -1 ; i++)  {  **if**(comPivotamentoParcial)  {  *troca\_linha*(u);  }  *setIdentidadeToMatriz*(m);  **for**(**int** j = i + 1 ; j < a.length ; j++)  {  m[j][i] = -u[j][i] / u[i][i];  }  linv = *multiplicarMatrizes*(m, linv);  u = *multiplicarMatrizes*(m, u);  b = *multiplicarMatrizes*(m, b);  }  **double**[][] x = **new** **double**[a.length][1];  **for**(**int** i = a.length -1 ; i>=0 ; i--)  {  x[i][0] = b[i][0];  **for**(**int** j = i + 1; j < a.length ; j++)  {  x[i][0] = x[i][0] - u[i][j] \* x[j][0];  }  x[i][0] = x[i][0] / u[i][i];  }  **return** x;  } |

Parâmetro de entrada:

*a* – Matriz de números reais

*b* – Matriz de números reais

*comPivotamentoParcial –* parâmetro booleano que, se verdadeiro, faz-se pivotamento parcial para se achar a solução, caso contrário, executa o algoritmo sem pivotamento parcial.

Retorno da função:

A matriz *x* que é resultado do Sistema Linear Ax = b.

Este algoritmo resolve o Sistema Linear Ax = b a partir de A (que é *a* neste algoritmo) e de *b* a partir da fatoração LU*.* O código das linhas 8 a 22 calcula as matrizes U, L-1 (inversa de L) e y (L-1 \* b). As linhas 15 a 18 mostram o cálculo da matriz de Gauss, sendo neste algoritmo m. A iteração de 8 a 22 ocorre n-1 vezes, que é a quantidade de transformações de Gauss. Na matriz *b* está armazenada a matriz y da fatoração LU, usando b e u, calcula-se a matriz x, que é a solução do sistema. A busca dessa solução está no código de 24 a 32. A função *multiplicarMatrizes e setIdentidadeToMatriz* estão na seção funções úteis utilizadas. O algoritmo *troca\_linha* está mostrado abaixo. Ele é executado na linha 12 se o booleano comPivotamentoParcial for verdadeiro, antes de se calcular cada matriz gaussiana.

1. **Algoritmo troca\_linha**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31** | **public** **static** ArrayList<**double**[][]> troca\_linha(**double**[][] matriz,**double**[][] b)  {  System.*out*.println("Imprimindo matriz antes de troca linha");  *printMatriz*(matriz);  **for**(**int** i = 0 ; i < matriz[0].length;i++)  {  **for**(**int** j = i + 1 ; j < matriz.length ; j++)  {  **if**(matriz[j][i] > matriz[i][i])  {  **for**(**int** z = 0 ; z < matriz.length ; z++)  {  **double** temp = matriz[j][z];  matriz[j][z] = matriz[i][z];  matriz[i][z] = temp;  }  **for**(**int** z = 0 ; z < b.length ; z++)  {  **double** temp = b[j][0];  b[j][0] = b[i][0];  b[i][0] = temp;  }  }  }  }  System.*out*.println("Imprimindo a matriz depois de troca linha");  ArrayList<**double**[][]> AeB = **new** ArrayList<**double**[][]>();  AeB.add(matriz);  AeB.add(b);  **return** AeB;  } |

Parâmetro de entrada:

*matriz*  – A matriz A do Sistema Linear Ax = b que terá a suas linhas permutadas.

*b –* A matriz b do Sistema Linear Ax = b que terá as suas linhas permutadas.

Retorno da função:

*Um conjunto com as matrizes* matriz *e* b com as suas linhas permutadas.

Este algoritmo percorre todas as colunas da matriz, a partir da primeira coluna à esquerda. Dada uma matriz m, para cada coluna i, se houver algum elemento na mesma coluna com índice da linha que i for maior que o valor de m[i][i], então essas duas linhas são trocadas . A matriz b também tem as mesmas linhas trocadas, de acordo com a troca das matrizes *matriz*. As matrizes matriz e b são então retornadas pela função, com as linhas permutadas, dentro de um mesmo conjunto A.

A comparação entre os dois elementos de uma mesma coluna é feita na linha 9 , a permutação entre duas linhas de matriz é realizado entre as linhas 10 e 16 inclusive, e a permutação entre as duas linhas de b é realizado entre as linhas 17 e 22 inclusive.

1. **Fatoração de Cholesky**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24** | **public** **static** **double**[][] cholesky(**double**[][] a)  {  **double**[][] r = a;  **for**(**int** k=0 ; k < a.length ; k++)  {  **double** primeiroSomatorio = 0;  **for**(**int** p = 0 ; p < k ; p++)  {  primeiroSomatorio += Math.*pow*(r[k][p], 2);  }  r[k][k] = Math.*sqrt*( r[k][k] - primeiroSomatorio );  **for**(**int** i = k + 1 ; i < a.length ; i++)  {  **double** segundoSomatorio = 0;  **for**(**int** p = 0 ; p < k ; p++)  {  segundoSomatorio += r[i][p] \* r[k][p];  }  r[i][k] = (r[i][k] - segundoSomatorio) / r[k][k];  r[k][i] = r[i][k];  }  }  **return** r;  } |

Parâmetro de entrada:

*a*  – Matriz de números reais.

Retorno da função:

A matriz R e RT em uma mesma matriz da fatoração Cholesky.

Este algoritmo calcula a matriz R e RT de Cholesky, a partir da raiz A. Para se encontrar a matriz R, encontra-se primeiramente a matriz L (), depois se descobre D-1 a partir da equação M = UTD-1 , encontra a raiz da inversa dela (D1/2), e por fim faz-se o cálculo R = LD1/2. No algoritmo acima, estes passos foram feitos instantaneamente. Para cada i da iteração, faz-se, nas linhas de 4 a 11, o cálculo da posição [i][i] da matriz R, nas linhas 12 a 19, é calculado as posições seguintes na mesma coluna, e na linha 20 já se insere o valor na da posição [k][i] na posição [i][k], já para, no resultado do algoritmo, prover tanto R como sua transposta na mesma matriz.

1. **Substituição Progressiva**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | **public** **static** **void** substituicao\_progressiva(**double**[][] l, **double**[][] b)  {  **double**[][] y = **new** **double**[b.length][b[0].length];  **for**(**int** i = 0 ; i < y.length ; i++)  {  y[i][0] = b[i][0];  **for**(**int** j = 0 ; j < i ; j++)  {  y[i][0] = y[i][0] - l[i][j] \* y[j][0];  }  y[i][0] = y[i][0] / l[i][i];  }  **return** y;  } |

Parâmetro de entrada:

L – A matriz l inversa da fatoração LU de um Sistema Linear.

b – A i-ésima coluna de uma matriz identidade

Retorno da função:

A matriz y, resultado de L \* b.

A função deste algoritmo é retornar a matriz y a partir de L inversa (que está somente l no algoritmo) e de b, que é a i-ésima coluna da matriz identidade, onde 1 <= i <= (número de colunas de a). Este cálculo é usado para a substituição regressiva, onde se encotra as colunas da inversa da matriz A do sistema linear Ax = b.

1. **Substituição Regressiva**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14** | **public** **static** **void** substituicao\_regressiva(**double**[][] y,**double**[][] u)  {  **double**[][] x = **new** **double**[y.length][y[0].length];  **for**(**int** i = x.length - 1; i >= 0 ; i--)  {  x[i][0] = y[i][0];  **for**(**int** j = i+1; j < x.length ; j++ )  {  x[i][0] = x[i][0] - u[i][j] \* x[i][0];  }  x[i][0] = x[i][0] / u[i][i];  }  **return** x;  } |

Parâmetro de entrada:

y– Matriz de números reais que possui uma só coluna.

u – Matriz de números reais.

Retorno da função:

A matriz R e RT Em uma mesma matriz da fatoração Cholesky.

Este algoritmo faz o cálculo do sistema Ux = y, onde y e U são os parâmetros do algoritmo, e x é o resultado do mesmo. Y é o resultado vindo da substituição progressiva, e x é a i-ésima coluna da matriz A-1 (a matriz inversa de A), sendo 1 <= i <= (numero de colunas de a).

**2 - Algoritmos Úteis**

1. **Multiplicação de Matrizes**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28** | **public** **static** **double**[][] multiplicarMatrizes(**double**[][] multiplicadora,**double**[][] multiplicando)  {  **if**(multiplicadora[0].length != multiplicando.length)  {  System.*out*.println("O tamanho das matrizes não são compatíveis");  **return** **null**;  }  **double**[][] resultado = **new** **double**[multiplicadora.length][multiplicando[0].length];  **for**(**int** i = 0 ; i < resultado.length ; i++)  {  **for**(**int** j = 0 ; j < resultado[0].length ; j++)  {  resultado[i][j] = 0;  }  }  **for**(**int** i = 0 ; i < multiplicadora.length ; i++)  {  **for**(**int** j = 0 ; j < multiplicando[0].length ; j++)  {  **for**(**int** k = 0 ; k < multiplicando.length ; k++)  {  resultado[i][j] += multiplicadora[i][k] \* multiplicando[k][j];  }  }  }  **return** resultado;  } |

Parâmetro de entrada:

*multiplicadora*  – Matriz de números reais (Primeiro Operando da multiplicação).

m*ultiplicando -* Matriz de números reais (Segundo Operando da multiplicação).

Retorno da função:

A multiplicação de *multiplicadora* por m*ultiplicando.*

Das linhas 4 a 8, o algoritmo verifica se o tamanho das duas matrizes é compatível para a multiplicação. Se não, imprime-se uma mensagem e não retorna nada. Nas linhas 9 a 16, preenche-se a matriz resultado com 0, pois é exigido que se atribua o valor 0 nesta linguagem (a variável não começa já com 0). Nas linhas 20 a 27 é a multiplição das matrizes de fato.

1. **Atribuir identidade à matriz**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18** | **public** **static** **double**[][] setIdentidadeToMatriz(**double**[][] matriz)  {  **for**(**int** index = 0 ; index < matriz.length ; index++)  {  **for**(**int** j = 0; j < matriz[0].length ; j++ )  {  **if**(j == index)  {  matriz[index][j] = 1;  }  **else**  {  matriz[index][j] = 0;  }  }  }  **return** matriz;  } |

Parâmetro de entrada:

*matriz*  – matriz de números reais.

Retorno da função:

A matriz identidade do tamanho de matriz.

Este algoritmo simplesmente atribuir 1 para a diagonal (índice com colunas e linhas iguais), e 0 para os demais pontos da matriz. Então retorna-se a matriz identidade criada.