Università di Roma TorVergata
Laurea Magistrale in Informatica

Corso: Modelli e Qualità del Software

Federica Magliocca, 0318517,
federica.magliocca@students.uniroma2.eu

9 CFU

Nome della prova: Prit2 MQS

Data della Prova: 22/05/2023

Domande

1) Stimare il tasso iniziale φ_0 di failure per difetto (RISPOSTA IN 1.2), simulando un **pre-run** con lifetime T_i esponenziali di tasso λi = 0.00785 failure/ora, identico per tutti gli *i*, e di durata sufficiente a produrre *n*=100 failure. Si assuma l'esistenza di N_0= 60 iniziali difetti latenti (dire come si pensa di aver stimato N_0 (RISPOSTA IN 1.1). Per il generatore pseudocasuale della sequenza di base $< X_i >$ da cui ottenere la sequenza $< T_i >$ si usino i dati indicati in NOTA.

2) Simulare poi un **run** (*) con T_i esponenziali di tasso λi (failure/ora) calcolato, ad ogni *i*, secondo MUSA (A=0.95) e di durata sufficiente a ottenere 50 failure, e rilevare la sequenza ($T_1, \ldots, T_{50}$) dei tempi di lifetime.

3) Sulla base di detta sequenza, usando maximum likelihood e modello MUSA (A=0.95) stimare i parametri N e φ necessari alla stima di affidabilità, adottando il procedimento Newton-Raphson (codice Math accluso) per la ricerca delle radici, e usando come iniziali i valori N_0 , φ_0 di cui sopra, e numero di iterazioni come indicato in NOTA.

4) Dire cosa rappresentano i parametri N e φ stimati (indicandone le metriche) e come si perviene ad esprimere le funzioni f1 e f2 del codice Math richieste dal procedimento Newton-Raphson.

5) Con i valori di N e φ sopra stimati , esprimere e calcolare con Musa (fissando A=0.95 e B=1.2) il numero *i* di failures che si dovranno sperimentare prima di eliminare il 90% dei fault, e il relativo tempo t_i necessario.

6) Dare con Musa l'espressione dell'affidabilità $R_i(t)$ del prodotto dopo l'istante t_i e con essa calcolare la probabilità che la successiva failure non si verifichi prima di 4 ore, e quella che non si verifichi prima di 9 ore.

(*) La sequenza $< X_i >$ per la simulazione del run sia il seguito immediato della sequenza del pre-run.

NOTA
- Dati per la sequenza pseudocasuale $< X_i >$ di base:
Generatore moltiplicativo con a = 1.220.703.125 , m=2.147.483.648, seme X0 = da assicurare periodo max
- Numero di iterazioni nel NewRaph = fino a convergenza con distanza tra radici successive inferiore a 10^{-6}.

Risposte

1) La risposta è articolata in due parti, nella prima parte (1.1) spiego come si ottiene N_0 e nella seconda (1.2) mostro la simulazione del prerun e il calcolo del tasso iniziale di failure.

1.1 La stima di $\mathbf{N_0}$ si ottiene attraverso i modello **statici** di affidabilità che, con un'analisi di regressione su dati da progetti esistenti, stabiliscono una relazione tra le misure di complessità process oriented (d(F), a(F), cc(F)) o object oriented (PD, CBO, LOCM, WMC, Depth, RFC, OOFI, CDC, NOC) e il numero iniziale N_0 di difetti.

Minore è la complessità, minore è N_0.
Quindi dati dati due progetti dettagliati pd1 e pd2 di uno stesso modulo, quando si passerà alla loro codifica, il codice (attendibilmente) con N_0 più piccolo sarà quello che avrà (ad es. nel caso process oriented)
– migliore d-structuredness d(F)
– minore depth of nesting a(F)
– minore cyclomatic complexity cc(F)

Per verificare l'attendibilità delle stime viene fatto un confronto tra
- numero di difetti effettivamente esistenti nel prodotto (artificiosamente introdotti)
- numero stimato (N_0).

Una modalità per effettuare questo confronto è basata sul calcolo di Errore relativo medio(**RE**) e Correlazione(**CR**).

Alcuni modelli statici:
- **Akiyama**
- **Brooks**
- **Halstead**
- **Lipow**
- **Gaffney**
- **Compton-Withrow**
- **Rodriguez-Harrison-Satpathy-Dolado**
- **Sherry-Malviya-Tripathi (per metriche object oriented)**

1.2 Simuliamo un prerun generando casualmente n = 100 lifetime esponenziali T_i (ore) di tasso λ_i = 0.00785 (failure/ora), identico per tutti gli i, calcoliamo il tempo totale τ (ore) di prerun come somma di tutti i lifetime:

$$\tau = \sum_{i=0}^{n-1} T_i$$

E calcoliamo il tasso iniziale di failure φ_0 nel seguente modo:
- calcoliamo prima il tasso complessivo di failure per difetto (failure /ora) $\Phi_0 = n/\tau$
- E poi il tasso iniziale di failure per difetto (failure /ora) $\varphi_0 = \Phi_0/N_0$

Infine stampiamo i valori dei 100 lifetime generati, il tempo totale τ di prerun e la stima del tasso iniziale φ_0 .

Di seguito la simulazione del **prerun** il cui file sorgente è
/Prit2_MQS_Vers.2_Magliocca.jar/Prit2_MQS_Vers2_Magliocca/PreRun.java

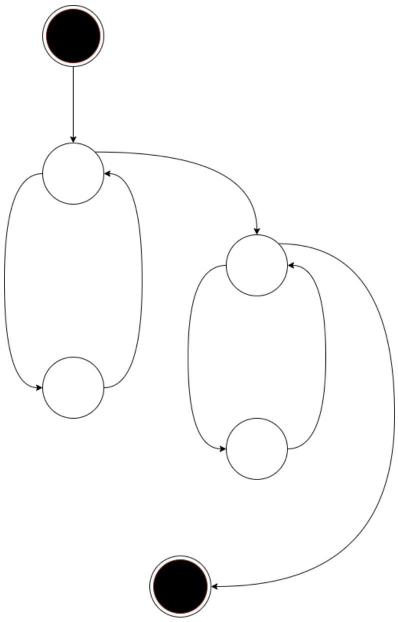
```
import java.util.Arrays;
/*
 * La classe PreRun simula un prerun con n lifetime esponenziali di tasso lambda_i utilizzando la classe
ExponentialStream
 * e calcola il tasso iniziale di failure per difetto
 */
public class PreRun {
	//genratore di numeri esponenziali casuali
	ExponentialStream generatoreSequenza;
	//numero di failure simulate
	private int numFailure;
	//sequenza di lifetime esponenziali
	double[] lifeTimes;
	//numero inizale di difetti latenti (N_0)
	int numDifettiLatenti;

	/*
	 * il costruttore prende in input il tasso lambda_i di failure/ora, il numero di failure da generare,
	 * il numero iniziale di difetti latenti e il seme da passare al generatore e setta i valori degli
attributi
	 */
	public PreRun(double lambda_i, int numFailure, int numDifettiLatenti,long seme){
		this.numFailure = numFailure;
		this.numDifettiLatenti = numDifettiLatenti;
		generatoreSequenza = new ExponentialStream(1/lambda_i,seme);
		lifeTimes = new double[numFailure];
	}

	/*
	 * funzione che genera una sequenza di lifetime esponenziali utilizzando il generatore esponenziale
	 */
	public double[] generaLifetime() {
		for(int i=0;i<numFailure;i++) {
			lifeTimes[i] = generatoreSequenza.getNumber();
		}
		return lifeTimes;
	}

	/*
	 * funzione che calcola il tempo totale di prerun come somma dei tempi di interfailure (lifetimes)
	 */
	public double calcolaTempoTotalePreRun() {
		double tempoTotale=0;
		for(int i=0;i<numFailure;i++) {
			tempoTotale = tempoTotale + lifeTimes[i];
		}
		return tempoTotale;
	}

	/*
	 * funzione che calcola il tasso inziale di failure per difetto
	 */
	public double stimaTassoInizialeFailure() {

		double tassoComplessivoFailure = numFailure/calcolaTempoTotalePreRun();
		double tassoInizialeFailure = tassoComplessivoFailure/numDifettiLatenti;
		return tassoInizialeFailure;
	}

	/*
	 *  funzione di supporto per restituire il valore del seme successivo da passare a un nuovo generatore
	 */
	public double getSuccSeme() {
		return generatoreSequenza.getSuccSeme();
	}

	/*
	 * viene simulato un prerun con con lifetime esponenziali di tasso lambda_i = 0.00785 failure/ora,
identico per tutti gli i,
	 * e di durata sufficiente a produrre 100 failure con numero di difetti iniziali = 60
	 * e stampa la sequenza di lifetime simulata e il tasso iniziale di failure per difetto phi_0
	 */
	public static void main(String[] args) {
	double lambda_i = 0.00785;
	int numFailures = 100;
	int N_0 = 60;
	long seme = 55;
```

```
        PreRun prerun = new PreRun(lambda_i,numFailures,N_0,seme);
        double[] lifeTimes = prerun.generaLifetime();
        System.out.println("LifeTimes: "+Arrays.toString(lifeTimes)+ "\nTempo totale di prerun: "+
prerun.calcolaTempoTotalePreRun()+"\nphi_0: "+ prerun.stimaTassoInizialeFailure());

    }
}
```

Di seguito è riportata la stampa dell'esecuzione:

```
LifeTimes: [169.71457444879502, 90.93602722746076, 35.339899370023964,
26.851907604626742, 180.65358493250537, 94.27641393814127, 33.270035655717535,
16.740729690192392, 103.23967804670328, 115.74873156809485, 173.49545432315315,
62.324711384214666, 80.459513552616, 27.337713982710085, 109.67833184120686,
127.77413064775645, 122.88735150229458, 213.58610614164087, 113.47933367429329,
327.3378321660405, 81.86875512372377, 57.50223566666266, 25.1923003057143,
156.08922112939842, 151.91425740596597, 113.89618145658113, 33.67363710365071,
207.63397213362163, 22.15689156721203, 423.88999251004327, 69.77879055389623,
193.29922218998087, 0.2439136788917782, 59.22602098954481, 169.64561235344132,
9.0892184086418, 115.19193922824827, 2.675709487686262, 14.286856885879207,
33.089317204216655, 45.767759501123656, 295.13852569662095, 1.4421704608039345,
160.11379907244168, 288.7633624628406, 28.337203551423126, 348.54438638173247,
95.84607954248949, 95.01916184637669, 3.602043209212463, 101.49474683440181,
16.407886571904534, 11.139935662749131, 245.34361295456412, 87.53905008898656,
351.94633063871095, 264.8313178219774, 805.5172335660374, 135.8725954893005,
421.01079116121105, 74.30524754658173, 108.78630824430661, 176.19902515447203,
137.0459384255, 72.12146584436802, 37.53163137490019, 26.28717448935173,
131.64725668764058, 60.72674927216272, 33.3391598727053, 0.43748565899378616,
94.16048193012656, 40.30758645588731, 168.83737791668582, 192.95193679929426,
30.034237067770288, 196.71647405993735, 93.92806793438038, 67.1399436702396,
20.44822204304894, 8.608932034766937, 137.63515788955894, 114.7989156377802,
189.50442048342356, 290.01851411107197, 2.9890236088456534, 359.80403264162976,
124.14890648970017, 28.052241548244723, 237.826520018621, 98.18782012929915,
73.51382525192538, 132.56368489523788, 167.85473327867223, 266.262640772697,
140.83030489339617, 58.866036043506945, 84.18719076266255, 0.8671334717104049,
160.31514534644305]
Tempo totale di prerun: 12408.941049353642
phi_0: 1.3431175634068146E-4
```

A seguire il flowgraph del codice di Prerun (Pseudo codice a pagina 13):



2) Simuliamo un run generando casualmente n = 50 lifetime esponenziali di tasso λ_i (failure/ora), calcolato, ad ogni *i*, secondo MUSA (A=0.95) ovvero calcoliamo λ_i nel seguente modo:

- $\lambda_i = 1/h_i$ dove $h_i = \varphi_0 N_0 e^{-\varphi_0 A t_i}$

Dove t_i è l'*i*-esimo istante di failure (ore) calcolato come somma cumulativa dei lifetime fino all'*i*-esimo lifetime partendo da $t_0 = 0$.
Utilizziamo λ_i come media nel generatore esponenziale per ottenere l'*i*-esimo lifetime.
Infine stampiamo i valori dei 50 lifetime generati e i corrispondenti istanti di failure.

Di seguito la simulazione del **run** il cui file sorgente è
/Prit2_MQS_Vers.2_Magliocca.jar/Prit2_MQS_Vers2_Magliocca/Run.java

```
import java.util.Arrays;

/* La classe Run simula un run con n lifetime esponenziali di tasso lambda_i, calcolato per ogni i secondo il modello MUSA,
 * e calcola gli istanti di failure
 * La sequenza degli esponenziale viene generata come conseguente della sequenza del prerun
 */
public class Run {
        //numero di failure simulate
        private int numFailures;
        //sequenza di lifetime esponenziali
    private double[] lifeTimes;
    //sequenza di istanti di failure
    private double[] failureTimes;
    //tasso di failure/ora
    private double lambda_i;
  //genratore di numeri esponenziali casuali
    private ExponentialStream generatore ;
    //tasso iniziale di failure per difetto ottenuto in prerun
    private double phi_0;
  //numero inizale di difetti latenti (N_0)
    private int n_0;


    /*
     * Il costruttore prende in input il tasso lambda_i di failure/ora, il numero di failure da generare,
         * il numero iniziale di difetti latenti e il seme per simulare un prerun.
         * Inoltre prende in input il numero di failure da generare nella simulazione del run.
         * Vengono settati i valori degli attributi della classe e viene simulato un prerun con gli input passati al costruttore.
         * Il generatore della sequenza di lifetime del run viene impostato al generatore della sequenza di lifetime del prerun,
         * in modo tale che la sequenza generata sarà il seguito della sequenza generata in fase di prerun
     */
    public Run(double lambdaPreRun,int numFailuresPreRun, int numFailures,int n_0, long seme) {
         this.n_0=n_0;
        this.numFailures = numFailures;
        lifeTimes = new double[numFailures];
        failureTimes = new double[numFailures+1];
        PreRun prerun = new PreRun(lambdaPreRun,numFailuresPreRun,n_0,seme);
        prerun.generaLifetime();
        phi_0 = prerun.stimaTassoInizialeFailure();
        generatore=prerun.generatoreSequenza;
    }

    /*
     * funzione che simula il run generando la sequenza di lifetime esponenziali di tasso lambda_i,
     * calcolato per ogni i secondo il modello MUSA, e calcolando la relativa sequenza di istanti
     * di failure come somma cumulativa dei lifetime
     */
    public void simulaRun() {
         failureTimes[0]=0;
        for (int i = 1; i < numFailures+1; i++) {
         lambda_i=1/(phi_0*n_0* Math.exp(-phi_0*0.95*failureTimes[i-1]));
         lifeTimes[i-1] = generaLifeTime(lambda_i);
         failureTimes[i] = calcolaFailureTime(i);
        }
    }

    /*
     * funzione che genera la sequenza di lifetime esponenziali
     */
    private double generaLifeTime(double lambda_i) {
```

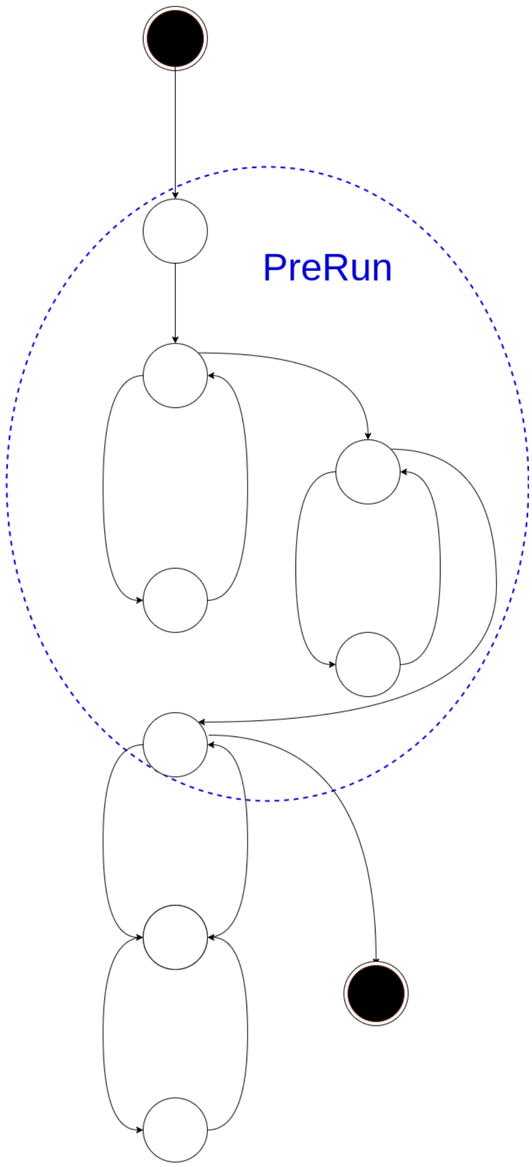
```
        generatore= new ExponentialStream(lambda_i, generatore.getSuccSeme());
        return generatore.getNumber();
    }

    /*
     * funzione che calcola la relativa sequenza di istanti
     * di failure come somma cumulativa dei lifetime
     */
    private double calcolaFailureTime(int i) {
        double failureTime = 0;
        for (int j = 0; j <= i-1; j++) {
            failureTime += lifeTimes[j];
        }
        return failureTime;
    }

    /*
     * funzione che restituisce la sequenza degli istanti di failure
     */
    public double[] getFailureTimes() {
        return failureTimes;
    }
    /*
     * funzione che restituisce la sequenza di lifetime generata
     */
    public double[] getLifetimes() {
        return lifeTimes;
    }

    /*
     * viene simulato un run con con lifetime esponenziali di tasso lambda_i, calcolato per ogni i secondo il
modello MUSA,
         * e di durata sufficiente a produrre 50 failure.
         * Stampa la sequenza di lifetime simulata e la sequenza di istanti di failure
     */
    public static void main(String[] args) {
        double lambda_i = 0.00785;
        int numFailures = 50;
        int n_0 = 60;
        int numFailuresPreRun = 100;
        long seme = 55;
        Run simulation = new Run(lambda_i,numFailuresPreRun, numFailures,n_0, seme);
        simulation.simulaRun();
        double[] failureTimes = simulation.getFailureTimes();
        double[] lifeTimes = simulation.getLifetimes();
        System.out.println("LifeTimes: "+Arrays.toString(lifeTimes)+"\nFailure Times: " +
Arrays.toString(failureTimes));

    }
}
```

Di seguito è riportata la stampa dell'esecuzione:

```
LifeTimes: [93.89726754105959, 15.375598621041716, 195.22864214639546,
49.70731928213155, 161.8907941127551, 79.2490759174173, 17.078275301676175,
80.627249495132, 458.04028265077426, 152.85224343268334, 3.432574696174488,
13.589289672664954, 73.35212016120313, 15.087214661085461, 88.63381119209168,
42.308351931771334, 24.267493519698835, 343.5192953013864, 184.9344904567581,
166.28243445369887, 214.4571226356223, 135.7123309393273, 51.8198193073251,
81.38045798302886, 94.0336676875361, 967.5323150795447, 29.958003047396353,
708.6690129590916, 338.419596607078, 162.8054352283882, 63.284128143760945,
55.751999420278544, 24.17081513948052, 347.8838927826927, 759.5737244728072,
17.441792030214533, 192.4635271317409, 213.76541714050393, 377.01792602554576,
48.597944573716184, 672.3148911231369, 83.16218878117377, 187.0438939935643,
25.685726021543697, 119.30849137421806, 666.489757316762, 242.43865769667784,
8.956357857350094, 473.93017691489297, 415.8604258245611]
Failure Times: [0.0, 93.89726754105959, 109.27286616210131, 304.5015083084968,
354.20882759062835, 516.0996217033835, 595.3486976208007, 612.4269729224769,
693.0542224176089, 1151.0945050683831, 1303.9467485010664, 1307.3793231972409,
1320.9686128699059, 1394.320733031109, 1409.4079476921945, 1498.041758884286,
1540.3501108160574, 1564.6176043357561, 1908.1368996371425, 2093.071390093901,
2259.3538245475997, 2473.810947183222, 2609.5232781225495, 2661.3430974298744,
2742.7235554129034, 2836.7572231004397, 3804.2895381799844, 3834.2475412273807,
4542.916554186472, 4881.336150793551, 5044.141586021939, 5107.4257141657,
5163.177713585978, 5187.348528725459, 5535.232421508152, 6294.806145980959,
6312.247938011174, 6504.711465142915, 6718.4768822834185, 7095.494808308964,
```

```
7144.0927528826805, 7816.407644005818, 7899.569832786991, 8086.613726780555,
8112.299452802099, 8231.607944176318, 8898.09770149308, 9140.536359189757,
9149.492717047107, 9623.422893962, 10039.283319786562]
```

A seguire il flowgraph del codice di Run (Pseudo codice a pagina 14):



3) Sulla base delle sequenze di lifetime T_i e failure time t_i generate in fase di run, usando maximum likelihood e modello MUSA (A=0.95) stimiamo i parametri N e φ adottando il procedimento Newton-Raphson per la ricerca delle radici, e usando come valori inziali $N_0 = 60$ e $\varphi_0 = 0.000134311$ (calcolato nel prerun).

A = 0.95;
t = {0.0, 93.89726754105959, 109.27286616210131, 304.5015083084968,
354.20882759062835, 516.0996217033835, 595.3486976208007,
612.4269729224769, 693.0542224176089, 1151.0945050683831,
1303.9467485010664, 1307.3793231972409, 1320.9686128699059,
1394.320733031109, 1409.4079476921945, 1498.041758884286,
1540.3501108160574, 1564.6176043357561, 1908.1368996371425,

```
  2093.071390093901, 2259.3538245475997, 2473.810947183222,
  2609.5232781225495, 2661.3430974298744, 2742.7235554129034,
  2836.7572231004397, 3804.2895381799844, 3834.2475412273807,
  4542.916554186472, 4881.336150793551, 5044.141586021939,
  5107.4257141657, 5163.177713585978, 5187.348528725459,
  5535.232421508152, 6294.806145980959, 6312.247938011174,
  6504.711465142915, 6718.4768822834185, 7095.494808308964,
  7144.0927528826805, 7816.407644005818, 7899.569832786991,
  8086.613726780555, 8112.299452802099, 8231.607944176318,
  8898.09770149308, 9140.536359189757, 9149.492717047107,
  9623.422893962, 10039.283319786562};
T = {93.89726754105959, 15.375598621041716, 195.22864214639546,
  49.70731928213155, 161.8907941127551, 79.2490759174173,
  17.078275301676175, 80.627249495132, 458.04028265077426,
  152.85224343268334, 3.432574696174488, 13.589289672664954,
  73.35212016120313, 15.087214661085461, 88.63381119209168,
  42.308351931771334, 24.267493519698835, 343.5192953013864,
  184.9344904567581, 166.28243445369887, 214.4571226356223,
  135.7123309393273, 51.8198193073251, 81.38045798302886,
  94.0336676875361, 967.5323150795447, 29.958003047396353,
  708.6690129590916, 338.419596607078, 162.8054352283882,
  63.284128143760945, 55.751999420278544, 24.17081513948052,
  347.8838927826927, 759.5737244728072, 17.441792030214533,
  192.4635271317409, 213.76541714050393, 377.01792602554576,
  48.597944573716184, 672.3148911231369, 83.16218878117377,
  187.0438939935643, 25.685726021543697, 119.30849137421806,
  666.489757316762, 242.43865769667784, 8.956357857350094,
  473.93017691489297, 415.8604258245611};

F[{c_, y_}] = {n/c - Sum[y*Exp[-y*A*t[[i]]]*T[[i]], {i, 1, n}],
  n/y - Sum[A*t[[i]], {i, 1, n}] -
   Sum[c*Exp[-y*A*t[[i]]]*T[[i]], {i, 1, n}] +
   Sum[y*c*A*t[[i]]*Exp[-y*A*t[[i]]]*T[[i]], {i, 1, n}]};

jacobian[{c_, y_}] = Transpose[{D[ F[{c, y}], c], D[ F[{c, y}], y]}];

NewtonSystem[X0_, max_] := Module[{}, n = 2;
  k = 0;
  Dp = {0, 0};
  P0 = X0;
  F0 = F[P0];
  Print["F[", P0, "]=", N[F0, 3]];
  P1 = P0;
  F1 = F0;
  While[k < max, k = k + 1;
        P0 = P1;
        F0 = F1;
        J0 = jacobian[P0];
   det = Det[J0];
   If[det == 0, Dp = {0, 0}, Dp = Inverse[Rationalize[J0, 0]] . F0];
```

```
        P1 = P0 - Dp;
   Print["Dist. radici =", EuclideanDistance[P0, P1]];
        F1 = F[P1];
   Print["F[", P1, "]=", N[F1, 3]];
   ];
  ];
NewtonSystem[{60, 0.00013431175634068146}, 6]
```

Stampa del risultato:

F[{60, 0.000134312}]={0.0522121, 5874.34}
Dist. radici =5.97085
F[{65.9708, 0.000124112}]={0.00892361, - 146.372}
Dist. radici =2.77785
F[{68.7487, 0.000117132}]={0.00201157, 330.087}
Dist. radici =0.50162
F[{69.2503, 0.000116192}]={0.0000521657, 4.38966}
Dist. radici =0.0180257
F[{69.2683, 0.000116153}]=□7.19662 × 10-8, 0.0107625□
Dist. radici =0.0000203013
F[{69.2684, 0.000116153}]=□8.62643 × 10-14, 9.31323 × 10-9□
Dist. radici =2.78533 × 10-11
F[{69.2684, 0.000116153}]=□1.11022 × 10-16, - 1.16415 × 10-10□

Nella stampa del risultato vediamo 4 dati di output:

- Ad argomento della *F* abbiamo le radici N e φ ad ogni iterazione. Nella prima riga troviamo i valori iniziali di $N_0 = 60$ e $\varphi_0 = 1.3431175634068146 \times 10^{-4}$, mentre in ogni riga successiva troviamo le radici prodotte dalla relativa iterazione del metodo.
- A membro destro, in parentesi graffe, ci sono i valori via via assunti dalle due componenti *f1* e *f2* della funzione vettoriale.

Le stime di N e φ (failure /ora) ottenute dopo 6 iterazioni (sufficienti ad avere la convergenza con distanza tra radici successive inferiore a 10^{-6}) del Metodo Newton-Raphson sono:

- N = 69
- φ = 0.000116153

4) Il parametro N rappresenta il numero di difetti presenti nel sistema al tempo considerato. Il valore stimato di N fornisce un'indicazione del numero medio di difetti nel sistema.
Il parametro φ rappresenta il tasso di failure per difetto, ovvero il numero medio di failure causate da difetti nel sistema per unità di tempo (failure/ora).

Le funzioni *f1* e *f2* utilizzate per il procedimento Newton-Raphson sono state calcolate come segue:
Data la sequenza $E = T_0, T_1, \dots T_n$ di tempi di vita generati in fase di statistical testing, secondo l'assunzione di indipendenza si ha che:

$$P(E)= f(T_1)\cdot f(T:2)\cdot \dots f(T_n)$$

Ipotizzando di voler determinare il parametro P di T_i avremo

$$\hat{P}(E|P)=\hat{f}(T_1|\hat{P})\cdot\hat{f}(T_2|\hat{P})\cdot\dots\cdot\hat{f}(T_n|\hat{P})$$

Analogamente nel caso di due parametri(N, φ).
Perciò ricordando che la funzione di densità del modello MUSA è

$$f_i(t) = -\phi N e^{-\phi A t_i} e^{-\phi N e^{-\phi A t_i} t}$$

e che tale funzione è relativa alla variabile *t* definita dall'istante t_i e all'interno del interfailure T_{i+1} che inizia all'istante t_i, lo stimatore sarà

$$\hat{f}(T_{i+1} | (\hat{N}\hat{\phi})) = -\hat{\phi}\hat{N} e^{-\hat{\phi} A t_i} e^{-\hat{\phi}\hat{N} e^{-\hat{\phi} A t_i} T_{i+1}}$$

Perciò

$$\hat{p}(E | (\hat{N}\hat{\phi})) = \prod_{i=1}^{n} -\hat{\phi}\hat{N} e^{-\hat{\phi} A t_i} e^{-\hat{\phi}\hat{N} e^{-\hat{\phi} A t_i} T_{i+1}}$$

questa espressione va derivata e annullata rispetto ai parametri N e φ, così si ottengono le funzioni *f1* e *f2*.
Prima di derivare conviene trasformare i prodotti in somme

$$\frac{d}{d\hat{N}} \sum_{i=1}^{n} \ln \hat{f}(T_i | (\hat{N}\hat{\phi})) = 0$$

$$\frac{d}{d\hat{\phi}} \sum_{i=1}^{n} \ln \hat{f}(T_i | (\hat{N}\hat{\phi})) = 0$$

Alla fine si ottiene

$$\frac{n}{\hat{N}} - \sum_{i=0}^{n-1} \hat{\phi} e^{-\hat{\phi} A t_i} T_{+1} i = 0$$

$$\frac{n}{\hat{N}} - \sum_{i=0}^{n-1} A t_i - \sum_{i=0}^{n-1} \hat{N} e^{-\hat{\phi} A t_i} T_{i+1} + \sum_{i=0}^{n-1} \phi \hat{N} A t_i e^{-\hat{\phi} A t_i} T_{i+1} = 0$$

5) Poichè secondo MUSA non c'è perfect debug, il numero *i* di failures sarà:

$$i = m(t_i) = M(t_i)/B = 0.9N/B = 0.9x69/1.2 = 51$$

Usando la notazione del modello MUSA, inoltre possiamo calcolare t_i:

$$M(t_i) = N(1 - e^{-\varphi A t_i})$$

$$0.9 N_0 = N(1 - e^{-\varphi A t_i}) \Rightarrow 0.9 = 1 - e^{-\varphi A t_i} \Rightarrow 0.1 = e^{-\varphi A t_i} \Rightarrow \ln(0.1) = -\varphi A t_i$$

Quindi

$$t_i = \frac{-\ln(0.1)}{\varphi A} = \frac{-2{,}302258}{(0.000116153 \cdot 0.95)} = 20867\, ore$$

6) L’espressione di affidabilità $R_i(t)$ dopo l’eliminazione dell’*i*-esimo fault con MUSA è:

$$R_{51}(t)= e^{-\phi N e^{-\phi A t_i} t}= e^{-\phi N e^{2.3} t}= e^{-\phi N 0.1 t}= e^{-0.0008 t}$$

Quindi la probabilità che la successiva failure non si verifichi prima di 4 ore è:

$$R_{51}(4)= e^{-0.0008\times 4}= 0.997$$

E la probabilità che la successiva failure non si verifichi prima di 9 ore è:

$$R_{51}(9)= e^{-0.0008\times 9}= 0.992$$

Da questo calcolo vediamo che i valori di affidabilità sono alti dopo 4 e 9 ore di funzionamento operativo. Infatti la probabilità che si verifichi una failure dopo 9 ore è molto bassa, ovvero 1 - 0.992 = 0.008 quindi lo 0.8% di probabilità.

PSEUDOCODICE PRERUN

```
Inizio classe PreRun
   Definisci attributi:
   generatoreSequenza: istanza di ExponentialStream
   numFailure: intero
   lifeTimes: array di numeri reali
   numDifettiLatenti: intero

    Definisci costruttore PreRun(lambda_i: numero reale, numFailure: intero,
   numDifettiLatenti: intero, seme: lang)
       numFailure ← numFailure
       numDifettiLatenti ← numDifettiLatenti
       Crea un nuovo oggetto ExponentialStream chiamato generatoreSequenza, passando
      (1/lambda_i) come media e seme come seme
       Inizializza lifeTimes come un nuovo array di numeri reali di dimensione numFailure

   Definisci funzione generaLifetime() che restituisce un array di numeri reali
      Per ogni i da 0 a numFailure-1:
      lifeTimes[i] ← generatoreSequenza.getNumber()
      Restituisci lifeTimes

   Definisci funzione calcolaTempoTotalePreRun() che restituisce un numero reale
      tempoTotale ← 0
      Per ogni i da 0 a numFailure-1:
      tempoTotale ← tempoTotale + lifeTimes[i]
      Restituisci tempoTotale

   Definisci funzione stimaTassoInizialeFailure() che restituisce un numero reale
      tassoComplessivoFailure ← numFailure/calcolaTempoTotalePreRun()
      tassoInizialeFailure ← tassoComplessivoFailure/numDifettiLatenti
      Restituisci tassoInizialeFailure

   Definisci funzione getSuccSeme() che restituisce un numero reale
      Restituisci il valore di generatoreSequenza.getSuccSeme()

   Definisci funzione main(con args: array di stringhe)
      lambda_i ← 0.00785
      numFailures ← 100
      N_0 ← 60
      seme ← 55
      Crea un nuovo oggetto PreRun chiamato prerun, passando lambda_i, numFailures, N_0
      e seme come argomenti
      lifeTimes ← prerun.generaLifetime()
      Stampa "LifeTimes: " seguito dalla rappresentazione in stringa di lifeTimes
      Stampa "phi_0: " seguito dal valore di prerun.stimaTassoInizialeFailure()
   Fine funzione main

Fine classe PreRun
```

PSEUDOCODICE RUN

Inizio classe Run

Definisci attributi:
numFailures: intero
lifeTimes: array di numeri reali
failureTimes: array di numeri reali
lambda_i: numero reale
generatore: istanza di ExponentialStream
phi_0: numero reale
n_0: intero

Definisci costruttore Run(*lambdaPreRun*: numero reale, *numFailuresPreRun*: intero, *numFailures*: intero, *n_0*: intero, *seme*: long)
n_0 ← *n_0*
numFailures ← *numFailures*
Inizializza lifeTimes come un nuovo array di numeri reali di dimensione numFailures
Inizializza failureTimes come un nuovo array di numeri reali di dimensione numFailures+1
Crea un nuovo oggetto PreRun chiamato prerun, passando *lambdaPreRun*, *numFailuresPreRun*, n_0 e *seme* come argomenti
prerun.generaLifetime()
phi_0 ← prerun.stimaTassoInizialeFailure()
generatore ← prerun.generatoreSequenza

Definisci funzione simulaRun()
failureTimes[0] ← 0
Per i da 1 a numFailures:
lambda_i ← 1 / (phi_0 * n_0 * esp(-phi_0 * 0.95 * failureTimes[i-1]))
lifeTimes[i-1] ← generaLifeTime(lambda_i)
failureTimes[i] ← calcolaFailureTime(i)

Definisci funzione generaLifeTime(lambda_i: numero reale) che restituisce un numero reale
generatore ← nuovo oggetto ExponentialStream, passando lambda_i e generatore.getSuccSeme() come argomenti
Restituisci generatore.getNumber()

Definisci funzione calcolaFailureTime(i : intero) che restituisce un numero reale
failureTime ← 0
Per j da 0 a i-1:
failureTime ← failureTime + lifeTimes[j]
Restituisci failureTime

Definisci funzione getFailureTimes() che restituisce un array di numeri reali
Restituisci failureTimes

Definisci funzione getLifetimes() che restituisce un array di numeri reali
Restituisci lifeTimes

Definisci funzione main(con args: array di stringhe)
lambda_i ← 0.00785
numFailures ← 50
n_0 ← 60
numFailuresPreRun ← 100
seme ← 55
Crea un nuovo oggetto Run chiamato simulation, passando lambda_i, numFailuresPreRun, numFailures, n_0 e seme come argomenti
simulation.simulaRun()
failureTimes ← simulation.getFailureTimes()
lifeTimes ← simulation.getLifetimes()
Stampa "**LifeTimes**: " **seguito dalla rappresentazione in stringa di lifeTimes Stampa** "**Failure Times**: " **seguito dalla rappresentazione in stringa di failureTimes**

Fine funzione main

Fine classe Run

Concluso il 15/06/2023 alle ore 12:45