Exchange rate dependence using copulae

Master Thesis submitted to

Prof. Dr. Wolfgang Härdle

Institute for Statistics and Econometrics

CASE - Center for Applied Statistics and Economics

Humboldt-Universität zu Berlin

by

Alexandru Isar

(185016)

in partial fulfillment of the requirements
for the degree of **Master of Sciences in Economics**

Berlin, February 14, 2009

Declaration of Authorship

I hereby confirm that I have authored this master thesis independently and without use of other than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such.

Berlin, March 17, 2009,

Alexandru Isar

Acknowledgments

I would like to thank Professor Dr. Wolfgang Härdle for this opportunity. I am grateful for his patience and support.

In that which concerns guidance, I am grateful to Dr. Enzo Giacomini for his ongoing support. Thank you is not enough.

I would like to thank my girlfriend Irina who has ever been by my side especially in this the difficult endeavor that is over today. I would like to thank my parents for the past nearly 30 years as well as my friends in Berlin and those who are still of Berlin though not there any more.

I am grateful for the experience that I have been able to gather while working at the European Central Bank. To my friends and my mentors: Joan Paredes, Wim Haine, Hans Olsson, Dieden Heinz Christian to name only a few - thank you.

For programming help, I would like to acknowledge Valentina Vulcanov.

Last but most definitely not least, I would like to thank Prof. Dr. med. Neuhaus and Dr. med. Kalmuk for their expertise and care.

Contents

1 Introduction **13**

2 Methodology **17**

2.1 Different Copulae and Dependence 19

3 Copula parameter fitting **25**

4 Data and computational challenges **29**

4.1 The dataset 29

4.2 Data treatment 31

5 Results **37**

6 Conclusion **47**

A C++ code **51**

List of Figures

2.1 Standard deviation of HUF and SKK exchange rate returns over a 200 day window 18

4.1 Log returns and $\hat{\varepsilon}_{j,t}$ - exchange rates versus the USD 33

4.2 Marginal distributions - estimated degrees of freedom 34

4.3 t copula d.f. - returns of USD vs EUR, GBP, JPY, SGD, NOK, CHF 35

5.1 Actual returns vs treated data (treated data refers to $\hat{\varepsilon}$ from equation 4.4) 38

5.2 Time dependent variance $\hat{\sigma}_t^2$ as estimated in equation 4.3 . . . 39

5.3 Marginal distributions' degrees of freedom 40

5.4 Copula degrees of freedom 41

5.5 Estimated kernel density of the elements of the rank coefficient transformed covariance matrix 42

5.6 Returns scatter plots 44

List of Tables

4.1 Eastern European countries experiences with pegs 30
4.2 Eastern European countries economic development 31

5.1 Quasi-correlation matrix elements' average and standard deviation 43

Chapter 1

Introduction

Univariate methods and analysis have often been used to describe the evolution of economic variables. The non-normal behavior of such variables has been observed as far back as Mills (1927) yet this characteristic has been assumed away in economic analysis. While this empirical observation may be an inconvenient fact it is also somewhat inconsequential for analysis concerned with the middle of a distribution rather than with its extremes. As such, due to the fact that much economic analysis is indeed concerned with the observation and prediction (i.e. expected value) of certain indicators, deviations from the standard normal model may not impact the conclusions of such analysis.

On the other hand, risk measurements are concerned with the occurrence of unlikely events. By definition, this implies that particular attention has to be paid to the functional form of the univariate or multivariate distributions of the variables under investigation. Conventionally, industry risk measurements are conducted using the multi-normal distribution (see J.P. Morgan/Reuters 1996 Group (1996)). Empirical results show however that there are certain inconsistencies with the assumption of normally distributed risk factors. Amongst other things, this assumption implies the following contradictions to stylized facts:

- constant volatility as pointed out by Giacomini & Härdle (2004), Giacomini & Härdle (2007)

- symmetrical distribution around mean as discussed in A. & J. (2002), Embrechts & Dias (2002)
- thinner tails than empirical evidence points to
- no tail dependence

The main issue with the above mentioned model comes from the fact that this is the standard tool used by financial institutions to evaluate the market risk they expose themselves to. This consequently leads to sub-optimal results.

The volatility as implied by the standard model is indeed less than the one observed empirically. Furthermore, when one enters a period of high volatility, the standard model will underestimate standard deviation and thus lead to an underestimation of risk. Asymmetry in the distribution of a variable means that one side - here the left side indicating negative returns - is heavier than the other. Again, the standard symmetrical distribution model may lead us to underestimate risk precisely because it may understate the area under the left hand side of the probability distribution function.
If we are looking at a multivariate distribution, the standard model implies no correlation in the extremes. Put plainly, this mens that extreme returns are not correlated under the standard model whereas empirical observations are at odds with this point as pointed out in Patton (2006). As explained later in the thesis, the standard normal model implies that tail dependence goes to 0 as one moves to extreme values of the involved variables. Empirical observations not only show the existence of such correlation in the extremes, but also point to the fact that the joint distribution is asymmetrical allowing for more correlated negative returns than positive ones.

There are a number of methods better suited to analyzing risk factors than the standard tool. One such method is the copula. Perhaps conceptually it is best to think of copulae as functions that determine the relationship between random variables. More precisely, they are the link between the marginal and the joint (cumulative) distribution functions.

In this thesis, I will aim to analyze the relationships between returns on foreign exchange positions with a particular attention to Eastern European currencies' exchange rates vis-a-vis the Euro. In the second and third chapters I will outline the methodology used, the reasons for choosing such method-

ology and the estimation process respectively. The fourth chapter will be dedicated to the data, the economic context and computational challenges. The fifth chapter will be dedicated to results whereas the conclusion will take up chapter six.

In the process of computing and reporting results, a number of software packages have been used including C++, Matlab and obviously LaTeX.

As the statistical device used in the analysis is a copula, a lengthy C++ code has been developed specifically for the task of computing its parameters. The code is available for inspection in the appendix.

Chapter 2

Methodology

The statistical process of interest here is the movement of daily returns due to holding a particular asset. Returns are defined as:

$$R_t = \frac{P_t}{P_{t-1}} - 1 \tag{2.1}$$

where R_t is the return at time t and P_t is the price at time t. When dealing with a portfolio of many assets with prices P_{it}, the return on the portfolio may be expressed as:

$$R_t = w^\top R_{i,t} \tag{2.2}$$

where $w^\top$ is the vector of weights of each asset i in the portfolio.

More commonly however, log returns are used:

$$r_{i,t} = \log\left(P_{i,t}\right) - log\left(P_{i,t-1}\right) \tag{2.3}$$

The later form of returns is an approximation of the former. One must be careful though: aggregating over time may produce inconsistencies. Log returns will nevertheless be used in the analysis to come.

Looking at the return process, one immediately identifies a non-constant volatility. As exemplified below in figure 2.1, the volatility of the returns of HUF and SKK exchange rates vis-a-vis the USD pass through periods of high and low volatility. In order to analyze the relationships between two or more variables, one must first bring them to common ground. This implies not only adjusting for the average, but also homogenizing their variance.

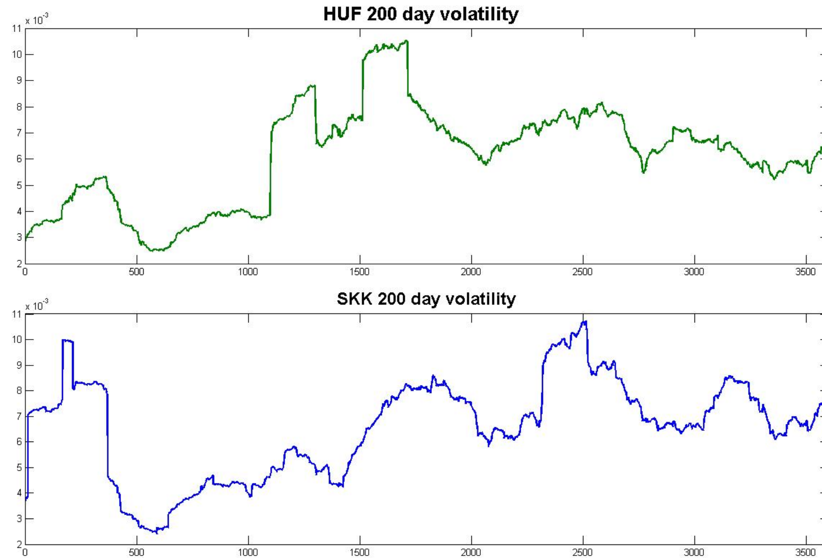

Figure 2.1: Standard deviation of HUF and SKK exchange rate returns over a 200 day window

Since the focus of this thesis is the exploration of the joint behavior of variables rather than the homogenization of variance, I will not go into great detail here and leave the explanation of this particular method for a later chapter. Subsequently, one needs to move from the univariate exploration to the multivariate context.

As previously mentioned, there are a number of methods one may use when describing the joint behavior of financial variables in general and return series in particular. One of those methods is called the copula. In order to better understand this construct, one may wish to have a look at the theorem underlying the functioning of the copula. This is called Sklar's theorem and it is outlined below.

Theorem:
Let F be a d-dimensional cdf with margins F_1,...,F_d then there exists a copula C such that for all $x \in \mathbb{R}^d$:

$$H(x_1, ..., x_d) = C\{F_1(x_1), ..., F_d(x_d)\} \tag{2.4}$$

Further, if $F_1, ..., F_d$ are continuous, then C is unique.

Basically, for every joint cdf one can think of there exists a corresponding copula. This is a very powerful result. It implies that a joint density can

be expressed as a combination between a marginal distribution and the relationship function that is the copula.
While one can model many kinds of dependence, one can also model independence using copulae.

$$\Pi\left(u_1, ..., u_d\right) = \prod_{i=1}^{d} u_i \tag{2.5}$$

Recalling that for two independent variables X_1 and X_2,

$$P(X_1 < x_1, X_2 < x_2) = P(X_1 < x_1)P(X_2 < x_2)$$

it is understandable why the product copula is also referred to as the independence copula shown in equation (1.2) above.

There are a variety of copulae and the nature of the dependency ranges from none to very complex. In the following section, I will give an overview of the nature of dependence as modeled by copulae. A suitable copula will be chosen based on its ability to model behavior consistent with stylized facts. On the basis of this theory, empirical work has been be performed and will be presented later in the thesis.

2.1 Different Copulae and Dependence

Before starting to give specific examples of copulae, it is necessary to introduce a few concepts of dependence. Seeing as it is in the area of extreme values where standard models fail, it might be useful to investigate dependence in the tails of the distribution.

Definition:
Given two random variables X_1 and X_2 with respective marginal cdf's F_1 and F_2, the (lower) tail dependence coefficient (if it exists) is defined as:

$$\lambda_L \stackrel{\text{def}}{=} \lim_{u \to 0} P\left\{X_2 \leq F_2^{-1}\left(u\right) | X_1 \leq F_1^{-1}\left(u\right)\right\} \tag{2.6}$$

Upper tail dependence is defined in the same way. Using some basic probability theory and the definition of copula, the limit can be expressed as in Nelsen (2006):

$$\lambda_L = C(u, u)/u \tag{2.7}$$

Using L'Hôpital's rule one can get an expression for this limit that may evaluate to a finite number. The value naturally depends on which functional form is assumed for the copula. We say that the copula has lower tail dependence if $\lambda_L \in (0, 1]$ and that it has no such dependence if $\lambda_L = 0$.
It is useful to have some measure of the dependence of variables in the tails. As expected it is in the copula function rather than in the margins where this dependence can be identified. That is of course due to the fact that no dependence at all is expressed in the margins. As we will see later however, there are some copulae with a tail dependence parameter equal to zero. This is an important property to keep in mind especially when applying our theory to modeling extreme values.

One copula lacking tail dependence is the Gaussian copula. This class is important not only due to the lack of tail dependence, but also due to the fact that its IFM (inference from margins) and MLE estimators coincide for the case where the margins are also Gaussian as mentioned in Giacomini (2005). The importance of this fact lies in that while the MLE is a precise estimator, the IFM often is not. The IFM is much less computationally intensive. Knowing that for this particular case the two estimators coincide, one may use the IFM in estimating the copula parameters and gain computing time.

Definition

$$C_{\Sigma}^{Gauss}\left(u_1, ..., u_d\right) = \int_{-\infty}^{\phi_1^{-1}(u_1)} ... \int_{-\infty}^{\phi_d^{-1}(u_d)} f_{\Sigma}\left(x_1, ..., x_d\right) dx_1 ... dx_d \tag{2.8}$$

where $f_{\Sigma}(x_1, ..., x_d)$ is the joint normal pdf.

As can be seen here, for the bivariate case the relationship of X_i and X_j depends on the correlation. Further, so does λ_U. As per Paul Embrechts & McNeil (2003)

$$\lambda_U^{i,j} = \lim_{x \to \infty} \bar{\Phi}\left(x \frac{\sqrt{1 - \Sigma_{i,j}^2}}{\sqrt{1 + \Sigma_{i,j}^2}}\right) \tag{2.9}$$

And further building on Paul Embrechts & McNeil (2003), it can be shown that for the Gaussian copula $\lambda_U^{i,j} = \lambda_L^{i,j}$. Inevitably, we see that for $\rho \neq 1$, the tail dependence is zero. This result puts into question the use of the Gaussian copula in modeling certain types of dependence in financial statistics as its behavior contradicts certain stylized facts as pointed out in the introduction to this thesis. One may imagine for instance the case of a market crash where all or most returns take on highly negative values. Such an event could not

be easily simulated with this (lack of) dependence.
It is noteworthy that at $\rho = 1$, the limit evaluates to 0.5 and so the expression evaluates to 1. It is thus that dependence exists only for one correlation value. Further, since the copula has to do with the relationship between the variables and not with margins, the result of course holds regardless of the margins used.

Another class of copulae is the t Copulae family. While the t Copulae exhibit tail dependence, they do converge to the Gaussian as one of their parameters, namely degrees of freedom goes to infinity.

Definition

$$C^t_{\nu,R}(u_1, ..., u_d) = \int_{-\infty}^{t_\nu^{-1}(u_1)} ... \int_{-\infty}^{t_\nu^{-1}(u_d)} f_{\nu,R}(x_1, ..., x_d) dx_d ... dx_1 \tag{2.10}$$

Where $f_{\nu,R}(x_1, ..., x_d)$ is the multivariate t pdf with coefficients ν and matrix P such that:

$$f_{\nu,R}(x_1, ..., x_d) = \frac{\Gamma\left(\frac{\nu+d}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{(\pi\nu)^d|P|}} \left(1 + \frac{x^\top P^{-1} x}{\nu}\right)^{-\frac{\nu+d}{2}} \tag{2.11}$$

And where $t_\nu^{-1}(u_1)$ is the quantile function *i.e.* the inverse of the t_ν univariate t cdf.

Two clarifications are perhaps necessary here. Firstly, with respect to the Γ(z) function - it is noteworthy that given its definition, it has regions where its behavior is non-standard. Namely, where z<0 the function is one-to-many and for positive real numbers it reaches minimum at $z_0 \approx 1.46163$ (Mathworld.com). It is a one-to-one function thereafter (*i.e.* $\forall\ z \geq z_0$). The implication of this property is that one can rest assured that $\Gamma\left(\frac{\nu+d}{2}\right)$ is increasing in ν and d for all multivariate cases with d.f. $\nu \geq 1$ and that of course $\Gamma\left(\frac{\nu}{2}\right)$ behaves the same for d.f. ≥ 3.
Another clarification pertains to margins versus copula degrees of freedom. The t copula $C^t_{\nu,R}$ used in conjunction with marginal t distributions and same d.f. yields a multivariate t distribution. It is nevertheless possible to use whatever margins one wishes. It is therefore possible to have d margins with ν_1,..., ν_d different d.f. and an additional ν parameter - copula degrees of freedom. The resulting distribution when using t or any other margins is then called Meta-t_ν distribution (Paul Embrechts & McNeil (2003), Embrechts & Dias (2003), Demarta & McNeil (2004)).
As will be shown later, the advantage of the Meta-t over the multivariate

t distribution is that it allows for the easier construction of IFM (inference from margins) estimators. This is simply because in the latter case one needs to simultaneously optimize for all parameters while in the former case we can take the IFM step by step approach. Put another way, a multivariate t distribution imposes the condition that the copula d.f. equal each individual margin's d.f. . This complicates the numeric computation of the parameters and forces us to use MLE which turns out to be very computationally intensive.

For the purpose of exemplification it is perhaps useful to have a look at the tail dependence measure in the bivariate t copula case. The tail dependence coefficients for upper and lower dependence are the same. As solved for in Demarta & McNeil (2004), they are:

$$\lambda_U = \lambda_L = \lambda = 2t_{\nu+1}\left(\frac{-\sqrt{\nu+1}\sqrt{1-\rho}}{\sqrt{1+\rho}}\right) \tag{2.12}$$

Note that as shown in Table 1, page 5 of Demarta & McNeil (2004), λ is increasing in ρ and decreasing in ν. This is somewhat intuitive since $C^t_{\nu,\rho}$ converges to a Gaussian copula as d.f. go to infinity.

Another interesting family of copulae is the Gumbel-Hougaard family. While the two previous families belonged to the elliptical class, this particular one does not. One parameter determines the relationship between variables in this case, namely θ. The functional form is:

$$C_\theta^{G-H}(u_1, ..., u_d) \stackrel{\text{def}}{=} \exp\left[-\left\{-\log(u_1)^\theta - ... - \log(u_d)^\theta\right\}^{1/\theta}\right] \tag{2.13}$$

$$\text{with } \theta \in [1, \infty)$$

Two notable properties of this copula are more easily noticed as the parameter θ approaches either one or infinity. While it is clear that at $\theta = 1$ the Gumbel-Hougaard is equivalent to the product copula (just by replacing the parameter by 1), the opposite limit requires a more complicated approach. In fact:

$$\lim_{\theta\to\infty} C_\theta = \min(u_1, ..., u_d) \tag{2.14}$$

Intuitively one may understand that as the parameter is allowed to go to infinity, this function will converge to the minimum function.

The minimum function $M(u_1, ..., u_d)$ is a copula with very interesting properties. In fact, it is an upper bound for all copulae:

$$C(u_1, ..., u_d) \leq M(u_1, ..., u_d) \qquad \forall\, C,\ (u_1, ..., u_d) \tag{2.15}$$

As previously stated, the Gumbel-Hougaard family is not elliptical class. Due to this, the upper and lower tail dependence is not the same. This is a useful property since as per stylized facts metioned amongst others in Giacomini & Härdle (2004), A. & J. (2002), Embrechts & Dias (2002), negative returns happen together more often than positive ones. In bivariate case:

$$\lambda_U = 2 - 2^{1/\theta} \tag{2.16}$$

Unfortunately, as the Gumbel-Hougaard copula has no lower tail dependence ($\lambda_L = 0$), it is not very useful for our purposes. Note however that (upper) tail dependence is increasing in θ. Naturally at $\theta = 1$, since the Gumbel-Hougaard becomes the product copula, there is no dependence at all. Therefore there is no tail dependence.

There are many other classes of copulae, however for our purposes the t copula is the right choice. While it does not allow for asymmetry in the returns process, it is rather heavier in the tails than the Gaussian and thus may be useful for our purposes. Further, as opposed to the Gumbel-Hougaard copula, it allows for a multitude of relationships between the variables under consideration via its quasi correlation matrix P.

Chapter 3

Copula parameter fitting

A copula together with its margins may have many parameters depending on the number of dimensions used. If one undertakes an estimation of both the copula and margin parameters simultaneously, this implies a very heavy computational burden. To ease such a burden, the inference from margins (IFM) method has been developed. Put simply, this framework allows the user to estimate the margin parameters separately from the parameters of the copula, thus cutting down the computing time. Another advantage is that with a multiple argument optimization there may be no unique solution. A sequential algorithm eliminates this impractical problem.

Assuming the margins already fitted, one can proceed to the fitting of the copula parameters. Copula parameter fitting is an argmax problem involving the copula parameters as arguments on one hand and the implied copula likelihood function on the other. Using the c copula likelihood function shown below I have done just that.

Thus let us look at Gaussian and t copula cases. Their respective copula - probability density functions (pdf's) are given by the following:

$$c^t_{\nu,P} = \frac{f_{\nu,P}\left\{t_\nu^{-1}(u_1),...,t_\nu^{-1}(u_d)\right\}}{\prod_d^{i=1} f_\nu\left\{t_\nu^{-1}(u_i)\right\}} \tag{3.1}$$

$$c_R^{Gauss} = \frac{f_R\left\{\phi^{-1}(u_1),...,\phi^{-1}(u_d)\right\}}{\prod_d^{i=1}\left\{f(\phi^{-1}(u_i))\right\}} \tag{3.2}$$

where $f_{\nu,P}$ and f_ν are the standard respectively joint and univariate t pdf. The t_v^{-1} is the quantile of the univariate standard t distribution with d.f. ν. Correspondingly f_R and f are the standard joint and respectively univariate Gaussian pdf. The ϕ^{-1} is the quantile of the univariate standard Gaussian distribution.
Looking at the c^{Gauss}, we can see that the maximization of the log likelihood $\sum_{t=1}^{T} \log\ c^{Gauss}$ over the relevant time span with respect to R involves only the $f_R\,[\phi^{-1}(u_1), ..., \phi^{-1}(u_d)]$.

Consequently a really straightforward algorithm can be constructed:

- transform $(u_1, ..., u_d)$ into $(x_1, ..., x_d) = \{\phi^{-1}(u_1), ..., \phi^{-1}(u_d)\}$
- calculate $R\ =\ \mathrm{Corr}(x_1, ..., x_d)$

Subsequently, one has the copula parameter matrix R. We can now use this copula.

When it comes to the t copula things become a bit more complicated. Due to the presence of the degrees of freedom parameter in the denominator, the optimization problem is not that simple. It can nevertheless be approached in two manners. For one, we can do a joint optimization for ν and P. The other approach involves estimating the quasi-covariance matrix *a priori*.

The first approach is definitely more computationally intense. It involves simultaneously optimizing for a number of parameters, number that increases exponentially with the number of variables under joint investigation. That is because the amount of below diagonal elements of a d by d matrix is $d(d-1)/2$. The second approach is more sensible. The matrix may be estimated piece-wise and this leaves a relatively simple optimization for the degrees of freedom parameter. In comparing the fit for different degrees of freedom, one has to keep in mind that ν is only a scalar and that there is for all practical purposes an upper bound to ν. The upper bound is there because for large ν the t distribution converges to the Gaussian. It therefore does not make sense to use t copula for large degrees of freedom: might as well save the computing time and use the Gaussian.

It is my suggestion to 'cycle' through different ν values to find the optimal one as follows:

- calculate the P matrix piece by piece
- set a maximum ν_{max}
- set a minimum ν_{min}
- find ν_{argmax} that maximizes likelihood using a numerical algorithm

Since the previous chapter mentioned it, we now turn our attention to the Gumbel-Hougaard copula and the estimation of its parameters. While this is a one parameter copula, it is easy to see that there is no analytical solution for the maximization problem. Subsequently it becomes necessary to do the optimization using a numerical algorithm. Implementation in the bivariate case is done in Xplore and I will not insist much on it here.

As mentioned above, the issue of margins is treated separately in the IFM framework. Consequently, for all three classes discussed here, fitting of parameters simply becomes a matter of estimating parameters as an argmax with one argument.

While it is more computationally challenging, it is the author's intuition that the t copula will yield the most meaningful results. It is thus concluded to proceed using this particular tool.

Chapter 4

Data and computational challenges

4.1 The dataset

The dataset under investigation consists of returns of Eastern European currencies' exchange rates against the Euro. The source of this data is DataStream. While the Euro has become the official currency of countries in the euro area on January 1, 2002, back data may be obtained via a variety of methods. While some artificial ECU and Euro back data is published by DataStream, the bulk of the back data was obtained by using US dollar exchange rates of the Eastern European currencies together with USD/EUR(ECU) exchange rates.
This data has been tested against actual EUR to Eastern European exchange rates. The signs of the built return series as compared to the actual return series for the overlap periods (periods for which both the actual and built exchange rates exist) verify that this is a valid approach.

It is worth mentioning here that the dataset contains only trading days data. Consequently, this analysis may be seen as an analysis of returns in consecutive trading days rather than an analysis of returns over equally spaced time periods. Another way of putting this is that time periods for which at least one currency does not show data are eliminated.

When analyzing Eastern European currency returns, one must also keep in

Country	First Peg	First event	Second event
Bulgaria	5 July 1999 pegged to DM		
Czech Republic	Jan 1991 peg to basket	May 1997 abandoned peg	
Estonia	20 June 1992 pegged to DM	27 June 2004 revaluation/ERM II	
Hungary	1995 Crawling peg	28 February 2008 peg abandoned	
Lithuania	1 April 1994 peg to USD	1 February 2002 peg to Euro	28 June 2004 ERM II
Latvia	2 May 2005 ERM II		
Poland	1991 crawling peg	1 January 1995 redenomination	April 2000 float
Romania	1 July 2005 redenomination		
Slovenia	28 June 2004 ERM II	1 January 2007 Euro Area	
Slovakia	October 1998 peg to basket abandoned	28 November 2005 ERM II	17 March 2007 revaluation

Table 4.1: Eastern European countries experiences with pegs

mind the time span one looks at. As our reference period is 1995 to 2008, it is worth pointing out that an important part of this time span is characterized by depreciations in all currencies. As seen in table 4.1, some countries have experimented with pegs in the 1990s, some of which were abandoned at one point or another. Further, for the latter part of the period, at least some of the currencies prepare to or enter the Exchange Rate Mechanism as those countries prepare for the adoption of the Euro. This information is important for our analysis as a peg requires the authorities to intervene and defend it while the ERM II mechanism presupposes a 15 percent variance band for the exchange rate.

	GDP			Unemployment			CPI		
Country	95-99	00-04	05-08	95-99	00-04	05-08	95-99	00-04	05-08
Bulgaria	-1.2	5.1	6.3	14.2	16.0	8.0	252.7	6.4	8.3
Czech Republic	2.0	3.1	5.9	5.5	8.0	6.4	7.8	2.7	3.4
Estonia	5.1	7.5	6.1	10.0	11.0	5.9	15.0	3.5	6.4
Hungary	3.3	4.3	2.7	8.8	6.0	7.6	18.9	7.2	5.4
Lithuania	4.6	7.8	7.1	9.6	14.1	5.8	15.8	0.5	6.0
Latvia	3.9	7.6	8.1	16.6	11.9	7.1	11.6	3.2	9.8
Poland	6.0	3.0	5.5	12.0	18.6	12.2	16.3	4.4	2.5
Romania	-0.2	6.1	6.7	5.5	7.6	6.8	66.1	26.0	7.1
Slovenia	4.3	3.5	5.4	7.1	6.4	5.5	9.1	6.8	3.7
Slovakia	4.3	4.5	8.1	13.1	18.5	12.7	7.8	7.8	3.7
Germany	1.7	0.6	2.0	8.7	8.5	9.1	1.3	1.5	2.1

Table 4.2: Eastern European countries economic development

Eastern European countries have by in large joined the European Union. In itself this implies that at some point the countries in question will join the ERM II and subsequently the euro area. Those currencies' evolution since the mid 1990s can be characterized by a rapid depreciation in some members of the group and a subsequent stabilization of the exchange rate. Stabilization comes about either together with some form of peg or not, however it is always accompanied by improvements in the macroeconomic situation.

The improvement in the economic situation since the mid 1990s is apparent in the table 4.2. It is far more likely that this improvement is responsible for more stable exchange rates than the introduction of the Euro in 1999.

Given the uncertain nature of our future results, it would perhaps be wise first to look at relationships using a standard hard currency dataset first. In that which follows, the method of analysis will be exemplified using EUR, GBP, JPY, SGD, NOK, CHF exchange rate returns against the US dollar.

4.2 Data treatment

In order to investigate the relationship between asset returns and taking into account empirical observations stated in the previous chapters, a t copula with t margins (meta-t copula) seems suitable.

In that which is to follow, log return calculated from changes in USD exchange rates against six major currencies will be used for an exemplification of the methodology.

As the input data for copula analysis is assumed to be drawn from the same distribution (iid), time varying volatility must be first filtered out as in Giacomini & Härdle (2004). To this end, a very common approach is used in order to homogenize and standardize the volatility of the input data for the copula and associated marginal distributions. As in Risk Metrics methodology Group (1996), we have the returns process $X_{j,t}$ assumed to be:

$$X_{j,t} = \sigma_{j,t}\varepsilon_{j,t} + \mu_{j,t} \tag{4.1}$$

and consequently estimating $\hat{\sigma}^2_{j,t}$:

$$\hat{\sigma}^2_{j,t} = \left(e^{\lambda} - 1\right) \sum_{s<t} e^{-\lambda(t-s)} \left(X_{s,j} - \mu_{j,t}\right)^2 \tag{4.2}$$

with:

$$\hat{\mu}_{j,t} = \frac{1}{t} \sum_{s \le t} X_{s,j} \tag{4.3}$$

and thus:

$$\hat{\varepsilon}_{j,t} = \frac{X_{s,j} - \hat{\mu}_{j,t}}{\sqrt{\hat{\sigma}^2_{j,t}}} \tag{4.4}$$

The λ is the "decay factor" of a shock to volatility: the larger λ, the faster the decay. For our purposes $\lambda = 0.94$ has been used. This parameter value is suggested by Risk Metrics Group (1996). Due to difficulties encountered in obtaining robust estimators for GARCH models with higher than one lag we use the equivalent of GARCH(1,1) as indicated above.

Some of the results obtained are shown in 4.1.

The second stage of the estimation procedure involves solving for the parameters of both the marginal distribution and of the copula. A window size of 250 observations is used for the moment. Although this is a disputed choice in the literature, this seems appropriate as it contains a relatively large amount of data points. Furthermore, the window size choice also depends on the dimension of the dataset. As we investigate more variables jointly, a larger window size seems more appropriate and compensates for the larger

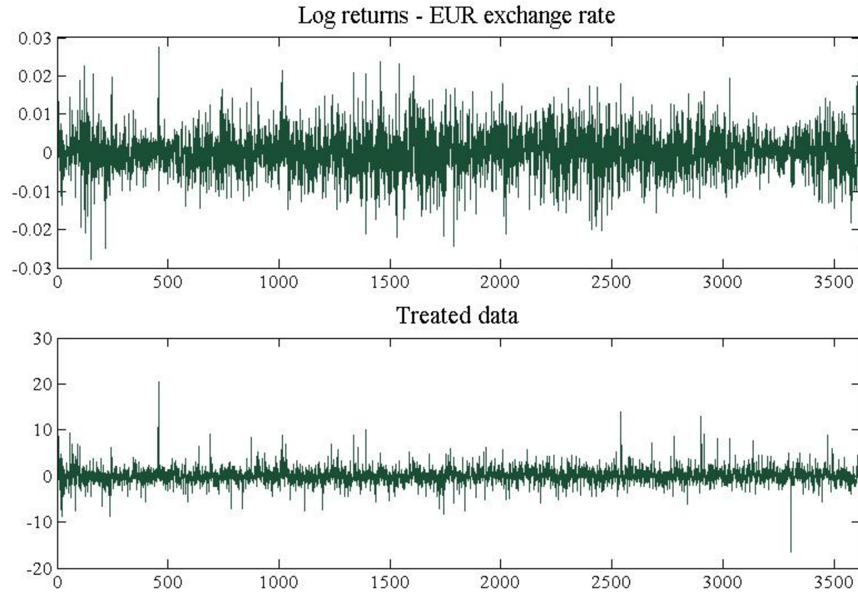

Figure 4.1: Log returns and $\hat{\varepsilon}_{j,t}$ - exchange rates versus the USD

dimension. Obviously, the trade off concerned with window size pertains to amount of data versus whether or not all data in the window are drawn from the same distribution. With a larger window size there is indeed more data, however the probability that the data is not drawn from only one distribution increases. Taking the example of the meta t copula, the IFM method will be followed in order to yield results in a timely manner as the alternative simultaneous estimation of all parameters can prove to be extremely computationally intensive.

The parameters (degrees of freedom) of the marginal t distributions are estimated from the above $\hat{\varepsilon}_{j,t}$ and subsequently the parameters of the copula (the quasi correlation matrix as well as degrees of freedom) are solved for.

The marginal distributions' estimated degrees of freedom are calculated using a widely available algorithm (Golden rule) applied to the maximization of the implicit log likelihood derived from the t distribution. Lower and upper bounds are set for the degrees of freedom between 2 and 200 respectively. Some of the results obtained while applying this method are shown in 4.2.

Furthermore, the copula parameters (i.e. the correlation matrix R_t and the

degrees of freedom term ν_t) are estimated independently. Starting off with a piecewise estimation of R_t one must use the methodology outlined in Demarta & McNeil (2004).

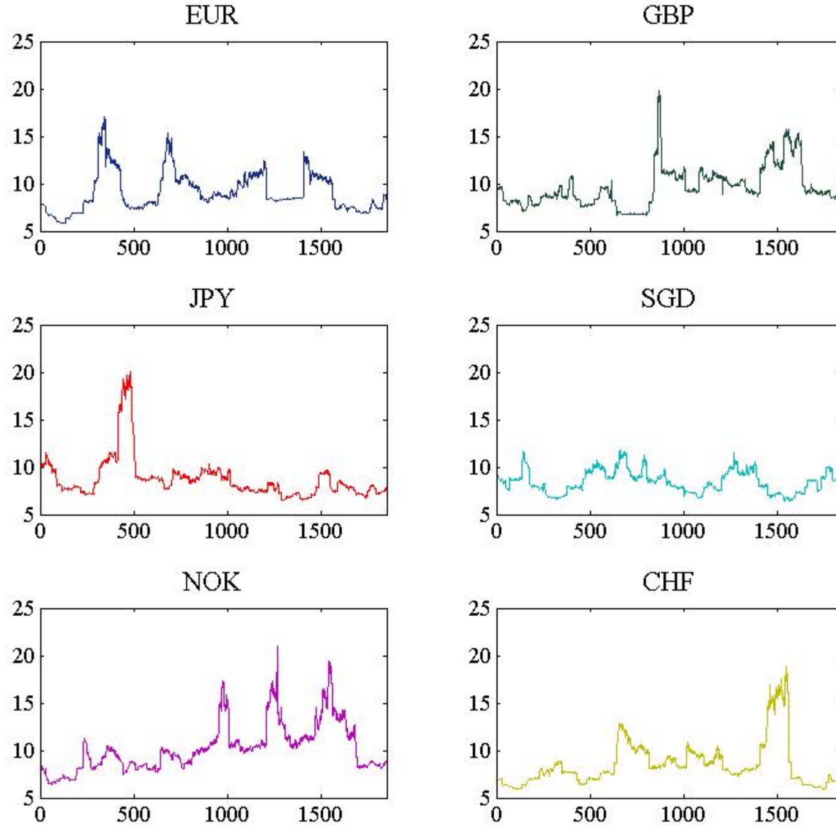

Figure 4.2: Marginal distributions - estimated degrees of freedom

Using a Kendall's tau $\hat{\rho}_\tau(X_j, X_k)$ estimate for each pair of variables j and k, it is possible to estimate the (j, k) item of R_t:

$$\hat{R}_{j,k} = \sin\left\{\frac{\pi}{2}\hat{\rho}_\tau(X_j, X_k)\right\} \tag{4.5}$$

At this point, one may realize that such a method may yield a non-positive semi-definite matrix $\hat{R}_t$. To this end, methods outlined in Rousseeuw & Molenberghs (1993) are used to find a positive semi-definite matrix close to

the one estimated by way of Kendall's tau. Minimizing the distance between the two matrices, one obtains a unique solution to this problem. The matrix has been computed for the example of the six exchange rates, however the results shall not be presented here. Such results will however be presented for the case of the Eastern European exchange rates later in the thesis.

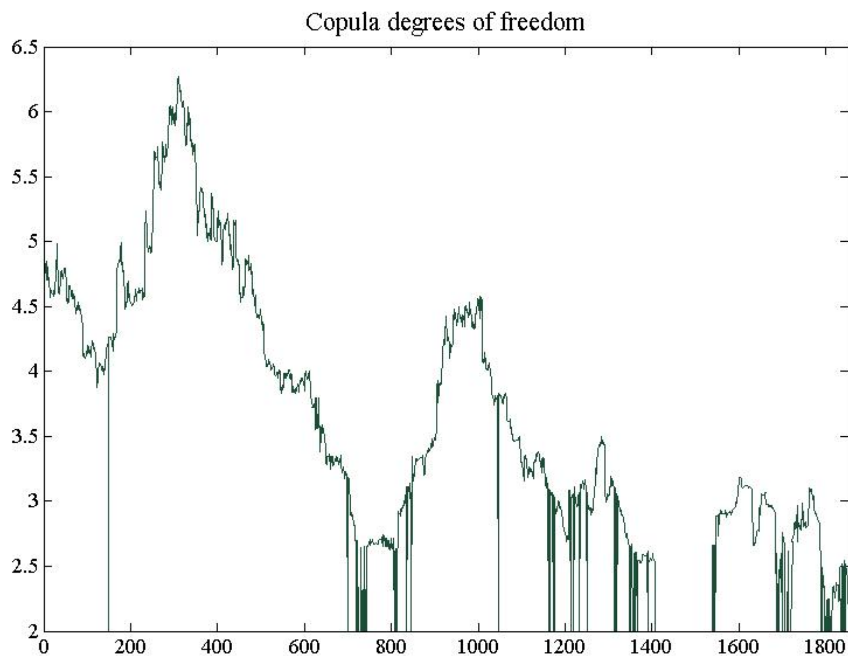

Figure 4.3: t copula d.f. - returns of USD vs EUR, GBP, JPY, SGD, NOK, CHF

With the correlation matrix estimated, one may now procede to estimate the remaining copula parameter ν. This is done using once again the Golden rule algorithm applied to the the implicit log likelihood derived from the copula function. The parameter has been calculated for the case of five exchange rates against the Euro (CZK, EEK, HUF, PLN, SKK). The result is shown in 4.3.

The results seem to indicate strong dependence in the tails (ν parameter approaches the lower limit quite often) over certain time periods. For now, the method having been exemplified, one can move on to performing the analysis on the core data and trying to evaluate whether the results have any

meaningful economic implications.

Chapter 5

Results

In this section I will aim to present and explain the results of the estimation. I will start by introducing the results of the univariate GARCH estimation followed by the results for degrees of freedom from the marginal distributions. The results of the copula degrees of freedom will also be presented together with some results from the quasi correlation matrix.

As mentioned previously, the first step in the estimation process involves a standardization of the data to mean zero and time invariant volatility. In order to achieve this, one needs to assume that the residuals follow a time variant volatility process as implied by (4.1-4.4). Consequently, the time dependent standard deviations of the residuals can be solved for and used to extract its time invariant component. The results will be presented below in figure 5.1.

As can be observed from figure 5.1, while there is a tendency for volatility clustering in the left column of graphs, this tendency disappears in the right column. Particularly, one can notice a considerable decrease in the volatility of the EEK and SKK exchange rates over time. The fact that the former has been pegged for some time to the Euro and both currencies were until recently in the European Exchange Rate Mechanism II partially explains this observation. A currency where the central bank actively intervenes in the market to stabilize it is obviously less volatile than one where the central bank does not take up this role. While in the past the Kroon's peg to first the Deutsche Mark and later the Euro was hard to maintain, its entry into the ERM II both requires and causes it to be a more stable currency vis-a-vis

the Euro. There is another problem that may affect volatility and namely the fact that the original data related to the US dollar and is rounded to four decimals leading as mentioned before to errors in the derived returns data. This phenomenon persists more in the beginning of the dataset than in the later part.

Treatment of the data has been undertaken using a $\lambda = 0.94$

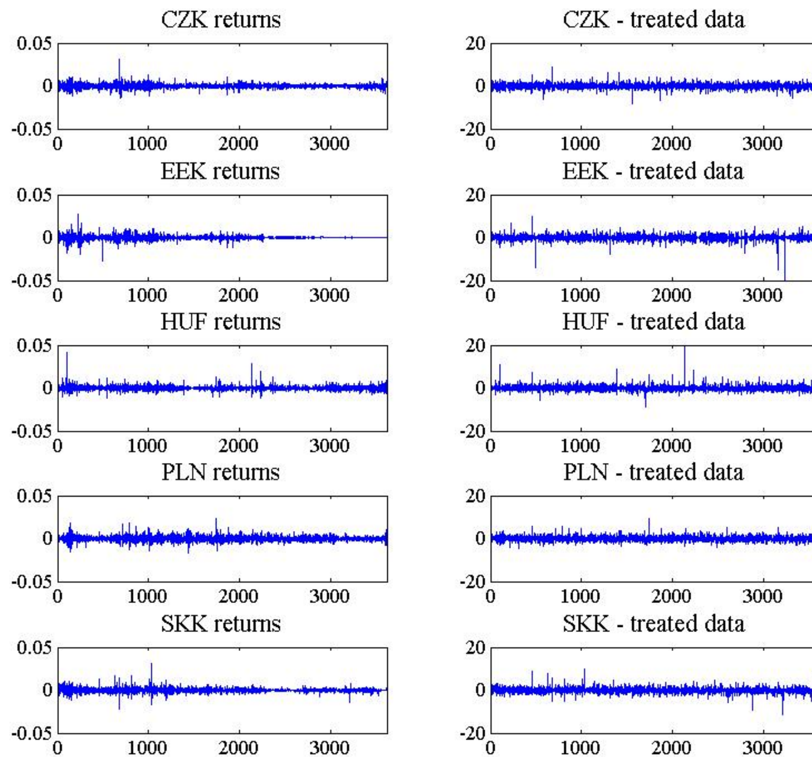

Figure 5.1: Actual returns vs treated data (treated data refers to $\hat{\varepsilon}$ from equation 4.4)

Conversely, one may look at this issue by observing the time dependent variance estimated as per equation 4.2. Note the EEK time dependent volatility which drops to very small values as well as the diminishing albeit not zero SKK volatility. It is obvious once again that countries in ERM II have less volatile currencies (see figure 5.2). Periods of high volatility are more easily

identifiable and show up as sustained peaks in the graphs. It is to be expected that volatility is prominent in the beginning of the sample, the period corresponding to mid 1990s - a time when most of Eastern Europe faced prologued instability.

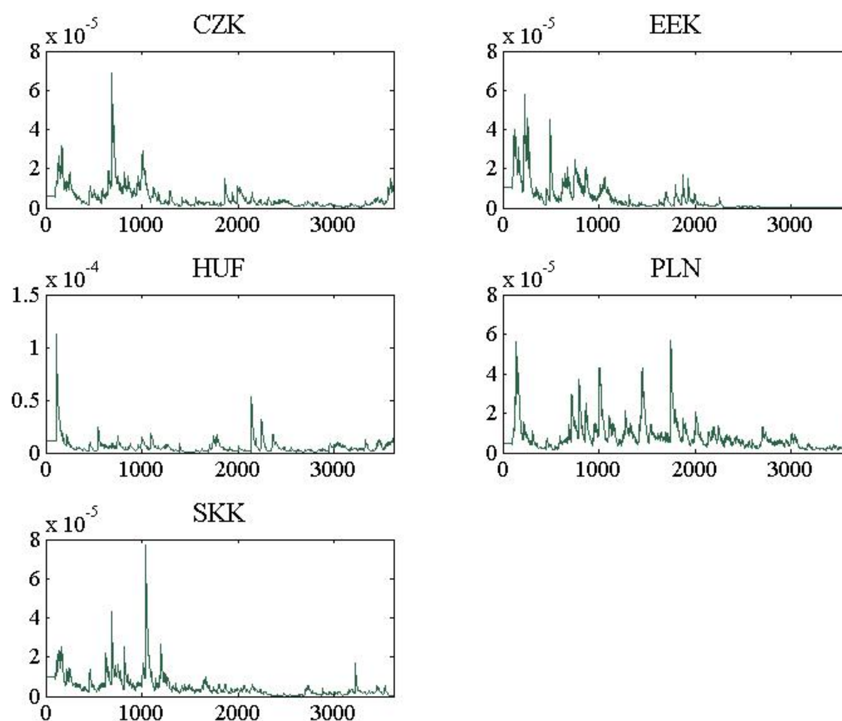

Figure 5.2: Time dependent variance $\hat{\sigma}_t^2$ as estimated in equation 4.3

The next step is the copula parameters' estimation. As stated in the previous chapters this is a two stage process involving margin followed by joint parameter fitting. Both the margins and the copula follow the functional forms of t distributions. The univariate margin degrees of freedom differ from the copula degrees of freedom.

As the data has been treated for time dependent variance, one can infer that the instability of the degrees of freedom of the marginal distributions as apparent in figure 5.3 reflects a changing of the shape of the distribution function toward and away from a normal distribution rather than a change in volatility. As low values of the degrees of freedom indicate departure from

the normal distribution to a distribution with more mass in the tails, the graphs can be indeed taken as indication of how much mass is available at the extremes. Seeing as the degrees of freedom are relatively small and high values are the exception rather than the rule, one can conclude that the data presents us with non-normal behavior and just as expected this implies a higher proportion of either large positive or negative values.
As mentioned in previous chapters, the results of a two stage estimation process is found to be close to that of the simultaneous estimation. Consequently, the conclusions drawn from looking at the marginal distribution remain valid and close to what they would have been had a simultaneous optimization of the joint distribution been undertaken.

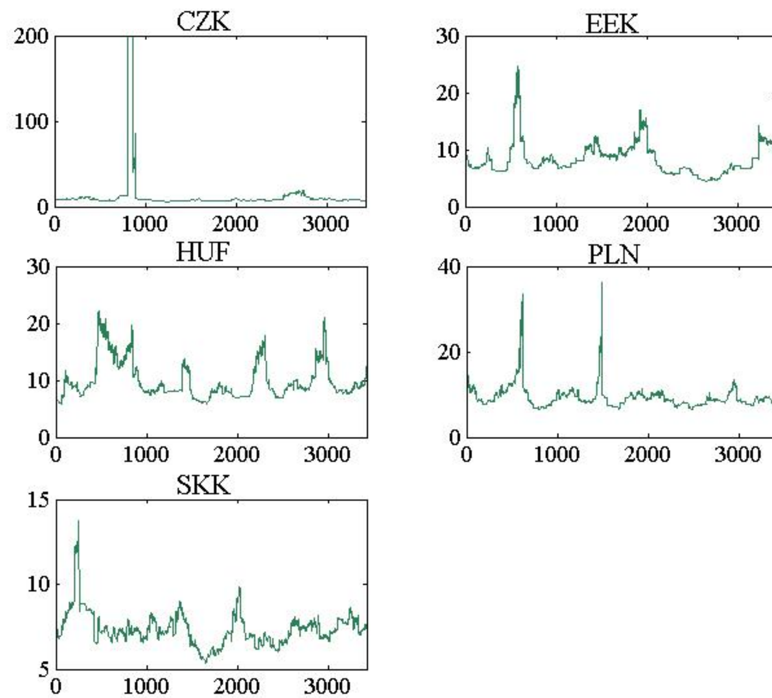

Figure 5.3: Marginal distributions' degrees of freedom

Subsequently, attention needs to be turned to the copula degrees of freedom parameter. As specified in the earlier chapters, the optimization procedure is set to maximize the copula implied maximum likelihood for values of the degrees of freedom parameter in the interval [2, 200]. Again, low values of the parameter indicate a more acute tendency of the probability mass toward

extreme values.

It is interesting to observe the parameter value going to the lower bound in the early part of the sample. Two factors contribute to this result. The first is the fact that the original data is rounded as well as the fact that one may introduce extra

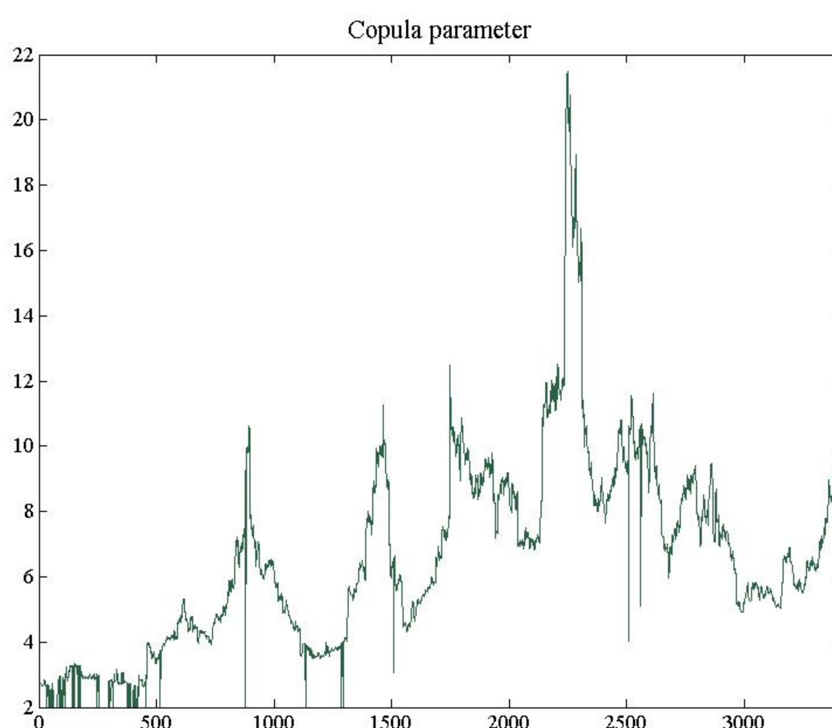

Figure 5.4: Copula degrees of freedom

noise by dividing the currency's exchange rate to the dollar by the dollar's exchange rate to the Euro. What is meant here is that this rounding may in itself affect the shape of the empirical or observed distribution function of returns. With rounding, the mass at the extremes may be overstated as rounding up or down exaggerates the magnitude of returns. Mass around the center of the distribution may also be overstated due to returns that are ignored and recorded as zero due to rounding. The second factor is explained by the fact that for Eastern European countries, the mid 1990s were a time of depreciation of the national currencies vis-a-vis what one may call hard

currencies. With all currencies devaluating, the number of joint incidents in the extreme increases leading to a low copula degrees of freedom parameter.

Finally, inspection of the quasi-correlation matrix is called for. As explained in previous chapters, this is not a correlation matrix per se but rather a measure of rank correlation. The difference is that while in the former only linear relationships between variables are captured, the latter is a much broader measure of co-movement. Naturally, since the estimation of the $\hat{R}_{j,k}$ elements of $\hat{R}$ from eq. 4.5 are further transformed in the procedure making $\hat{R}$ into a positive semidefinite matrix, looking at these elements may not yield much relevant information about the dependence structure of the copula.

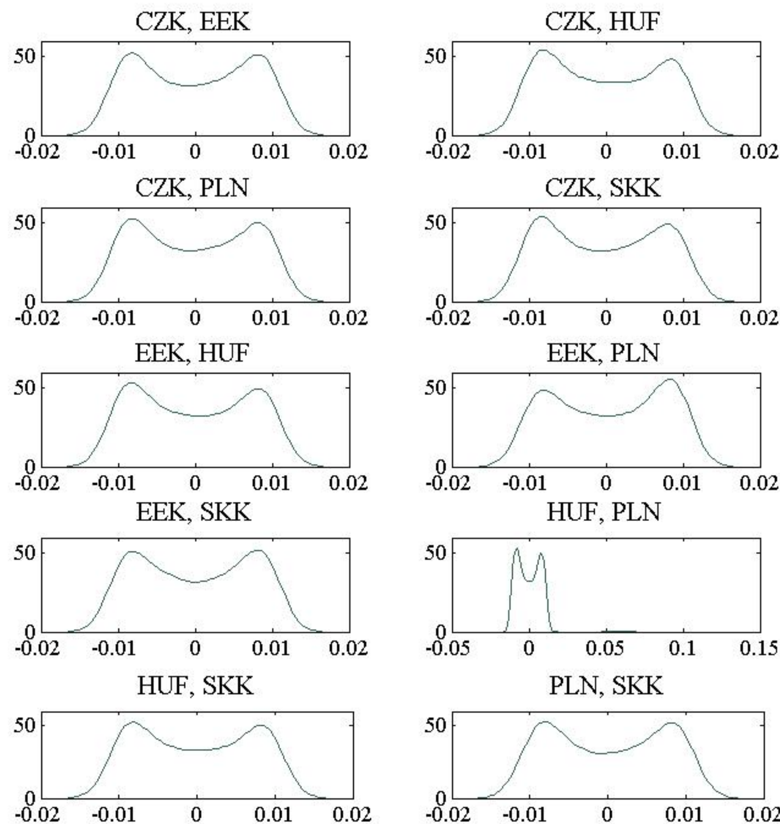

Figure 5.5: Estimated kernel density of the elements of the rank coefficient transformed covariance matrix

Currency pair	avg. ($\times 10^{-3}$)	std. dev. ($\times 10^{-1}$)
CZK, EEK	-0.25	0.23
CZK, HUF	-0.63	0.15
CZK, PLN	-0.36	0.16
CZK, SKK	0.03	0.21
EEK, HUF	0.03	0.16
EEK, PLN	-0.25	0.21
EEK, SKK	-0.41	0.17
HUF, PLN	-0.32	0.13
HUF, SKK	-0.19	0.22
PLN, SKK	-0.14	0.12

Table 5.1: Quasi-correlation matrix elements' average and standard deviation

Please see fig. 5.5 for results involving the elements of the matrix. The empirical densities in question were estimated using a standard kernel density estimator in Matlab. The empirical distributions are largely bi-modal with mean quite close to zero. This would seem to indicate no particular clear tendency overall in an either positive or negative association between any of the variables. This points to the fact that the currencies are either not hit equally and in the same manner by external shocks or that they respond differently to the same external stimuli. This is not an unexpected result seeing as the history of the countries involved points to a diversity of exchange rate policies pursued in the early to late 1990s. It is however surprising that as the countries become more stable, their currencies' behavior on the exchange rates markets do not converge.

Furthermore, looking at table 5.1 one sees not only sample averages approaching zero, but also a rather low variance. This only goes to confirm the above mentioned observations.

While in general and over the time span observed the elements of the matrix do not show a clearly positive or negative sign, it may be that such behavior manifests itself at a local level.

In order to inspect the behavior of the elements over time, it may be useful to have a look at fig 5.6. One can easily observe a time homogeneous behavior in the elements here as well, with a concentrated mass of observations dismissing the possibility of localized behavior. What is meant here is that were there to be two regimes in terms of relationships between variables, one might expect

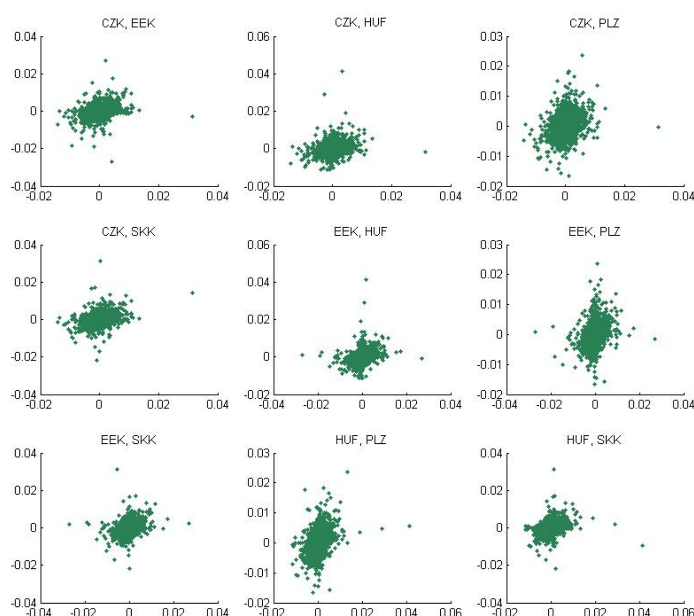

Figure 5.6: Returns scatter plots

to see two distinct masses in the scatter plot. This is evidently not the case.

One may however remember that what we are looking at is not a correlation matrix, but rather a transformation thereof. Furthermore, what we are interested in are co-movements in the extreme. Due to the nature of such events, their frequency being low, their weight when computing the Kendall tau coefficients will be outweighed by data points closer to the middle of the distribution. Furthermore, there is if not an evidence of co-movement then an indication of weight in the tails of the multivariate distribution and that is the relatively low copula degrees of freedom coefficient. It is noteworthy that despite also relatively low values of the degrees of freedom coefficients in the margins, this particular copula parameter remains low especially when we would expect it to - that is during the turbulent early to mid 1990s. This implies that while we cannot infer co-movement neither from the densities of the quasi correlation matrix elements nor from the scatter plots, there exists evidence for co-movement in the extremes.

The increase in both the copula and the margins degrees of freedom on the other hand seems to go together hand in hand with an improved macroeconomic outlook for the countries in question. It is difficult to say whether this

effect can be disentangled from the effect of the introduction of the Euro.

Chapter 6

Conclusion

In this thesis, an implementation of the t copula with t margins has been undertaken. The reasons for choosing this tool have been laid out in the earlier chapters and it has been shown that such a decision is both in line with the literature and with stylized facts in the data.

The method has been thoroughly explained and the reader has been taken through the implementation and algorithm. At each step a standard example using hard currency exchange rates has been used. The validity of the method is underlined by the results.

As the aim of the thesis was an investigation of the relationship between Eastern European exchange rates, data has been collected and the method has been applied to it. The main question to ask here is whether the introduction of the Euro affects the co-movement of these countries exchange rates.

Results seem to hint at:

- high volatility in the early 1990s followed by a period of stabilization across the board
- non-normal marginal distributions for all variables in question
- a low copula degrees of freedom parameter value that eventually increases albeit to still low levels

- quasi-correlation matrix elements close to zero

Consequently, one can say for sure that the distribution of the variables is consistent with joint high positive and joint negative behavior as pointed out by the low copula degrees of freedom parameter. Further, such behavior seems to ease off in the later part of the sample as the previously mentioned parameter increases.

The results are also consistent with the improving economic situation of the countries in question.

Further work may be carried out with the purpose of disentangling the effect of the introduction of the Euro from the effect of the improved economic situation.

Bibliography

A., A. & J., C. (2002), 'Asymmetric Correlations of Equity Portfolios', *Journal of Financial Economics* **63**, 443–494.

Demarta, S. & McNeil, A. J. (2004), 'The t Copula and Related Copulas', *Department of Mathematics ETHZ* **www.math.ethz.ch/finance**.

Embrechts, P. & Dias, A. (2002), 'Asymmetric Correlations of Equity Portfolios', *Journal of Financial Economics* **63**, 443–494.

Embrechts, P. & Dias, A. (2003), 'Dynamic copula models for multivariate high-frequency data in finance', *Department of Mathematics ETHZ* **www.math.ethz.ch/finance**.

Giacomini, E. (2005), 'Risk Management with Copulae', *Master Thesis - Humboldt Universität zu Berlin* **http://lehre.wiwi.hu-berlin.de/Professuren/quantitativ/statistik/research/dmb**.

Giacomini, E. & Härdle, W. (2004), 'Value-at-Risk Calculations with Time Varying Copulae', *CASE Working Paper* **www.case.hu-berlin.de**.

Giacomini, E. & Härdle, W. (2007), 'Inhomogenous Dependence Modelling with Time Varying Copulae', *WIAS Berlin* **Working Paper preprint no. 1273**.

Group, R. (1996), 'Riskmetrics technical document', *New York: J.P Morgan/Reuters* **www.riskmetrics.com/**.

Mills, F. (1927), 'The behaviour of prices', *New York: National Bureau of Economic Research* .

Nelsen, R. B. (2006), *An Introduction to Copulas*, Sprnger Verlag, New York.

Patton, A. J. (2006), 'Modelling assymetric exchange rate dependence', *International Economic Review* **vol. 47 no. 2**, p. 527–556.

Paul Embrechts, F. L. & McNeil, A. (2003), 'Modelling Dependence with Copulas and Applications to Risk Management', *Department of Mathematics ETHZ* **www.math.ethz.ch/finance**.

Rousseeuw, P. J. & Molenberghs, G. (1993), 'Transformation of non positive semidefinite correlation matrices', *Communications in Statistics - Theory and Methods* **2(4)**, 965–984.

Appendix A

C++ code

This appendix contains the C++ code that was used to obtain estimates for the parameters of the t copula in the previous chapters.

```
$

#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <malloc.h>
#include "pdfcdfran.h"

#define TRUE   1
#define FALSE  0
#define TINY 1.0e-20
#define PI 3.14159265358979323846
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
```

```
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
#define ITMAX 100
#define CGOLD 0.3819660
#define ZEPS 1.0e-10
#define TOL 1.0e-7
/*
Here ITMAX is the maximum allowed number of iterations;
CGOLD is the golden ratio; ZEPS is
a small number that protects against trying to achieve
fractional accuracy for a minimum that
happens to be exactly zero.
*/
#define GOLD 1.618034
#define GLIMIT 100.0
#define TINY 1.0e-20

/*Here GOLD is the default ratio by which successive
intervals are magnified; GLIMIT is the maximum
magnification allowed for a parabolic-fit step.
*/

//the shift function used for computing the min of
//a funtion given as parameter
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
static double maxarg1,maxarg2;
#define FMAX(a,b) (maxarg1=(a),maxarg2=(b),
(maxarg1) > (maxarg2) ?\ (maxarg1) : (maxarg2))

double **miu1,**sigma1;
int pos;

void reading_one_slice(char file_name[],
double ** matrice,int d, int n,int n1,long seed);
```

```
void Computing(double **matrice,int& n,int d,int n1,
long seed);

int factorial(int n);
int sign(double x);
double ro_theta(int i, int k, int n, double **matrice);
void P_star(double **P, double **matrice, int d, int n);
double gammaln(double xx);
double vector_mult(double *vect1, double *vect2, int dim);
void siftDown(double *numbers, int root, int bottom);
void heapSort(double *numbers, int array_size);
double choose_scalar(double *numbers,int size, double risk);
void ludcmp(double  **a, int n, int *indx, double *d);
void lubksb(double **a, int n, int *indx, double * b);
void inverse_computing(int dim, double **matrice, double ** inverse);
double determinant(int size, double **matrice);
int choldc(double **matrice, int n, double *p);
void scalar_multiplication(double **matrice, int n, double scalar);
void matrice_adding(double **matrice1, double **matrice2, int n);
void reset_identity(double **matrice, int n);
void transforming_semiposdef(double **matrice,int n, double *p);
double average(int indice, int n, double **matrice);
double deviation(double miu,int indice, int n, double **matrice);
double f_down(double **matrice, double niu, double x,
int indice, int n,double *miu, double *sigma);

double f_up(double niu, double **inverse_P,
double **matrice, int indice, int d,double det);
void transf_prices_u(double **matrice, double **U, int n,
int d, double *miu, double *sigma, double *niu_vector);
double sqfn_init(double p, int *prec);
double gammacf(double x, double a, double *gln);
double gammaser(double x, double a, double *gln);
double gammap(double x, double a);
double scdfn(double x);
double spdfn(double x);
double sqfn(double p);
double sqft_init(double p, double df, int *prec);
double betacf(double x, double a, double b);
double betai(double x, double a, double b);
```

```
double t(double x, double v);
double t_inverse(double p, double df);
double c(double niu, double **P_inverse,double **matrice,
int n,int d,int indice, double *miu, double *sigma,double det);
double l(int n, int d, double **matrice, double niu,
double **P_inverse,double *miu, double *sigma,double det);
double vector_matrix_multiplication(int size,
double **matrice, double *vector);
void vector_matrix_one_way_multiplication(int size, double **matrice,
double *vector1, double *vector2);
double ran2(long *idum);
void random_generator(int k, double *numbers,long seed);
double z_transform(double p);
double ppnd16_(double *p, int *ifault);
double ppchi2_(double *p, double *v, double *g, int *ifault);
double s_transform(double p, double v);
void  take_one_frame(int size_destination,int position,
double *source,double *destination);
void generator(double **matrice, int d, int n1, double niu,double *niu_vector,
double *miu,double *sigma,double *simulation_vector,long seed);
void  matrix_multiplication(double **P,double **L, int d);
double l_down(double **matrice, double niu,int n,
int indice,double *miu, double *sigma);

void mnbrak_niu_vector(double& ax, double& bx, double& cx,
double& fa, double& fb, double& fc,
double **matrice,int n,int indice,double *miu, double *sigma);

double brent_niu_vector(double ax, double bx, double cx, double tol,
double& xmin,double **matrice,int n,int indice,double *miu, double *sigma);
void de_Garch(int dim1,int monstra,int number_rows);
void q_sort(double *numbers, int left, int right);

double n_function(double x)
{
     // check extreme value of x
     if (XisNaN(x))
          return dnan;
     else if (XisPosInf(x))
          return 1.0;
     else if (XisNegInf(x))
```

```
          return 0.0;

     if (x >= 0.0){
//   printf("here\n %f \n", (1 + gammap(x*x/2.0, 0.5))/2.0);
          return (1.0 + gammap(x*x/2.0, 0.5)) / 2.0;
 }
     else return (1.0 - gammap(x*x/2.0, 0.5)) / 2.0;
}

double n_inverse(double p)
{
     double initapp, qcur, dx;
     int i, prec;

     initapp = sqfn_init(p, &prec);
     if (prec)
          return initapp;

     // iterate
     qcur = initapp;
     for(i = 1; i <= NEWTON_ITMAX; i++)
     {
          dx = (scdfn(qcur) - p) / spdfn(qcur);
          qcur -= dx;
          if (abs(dx / qcur) < NEWTON_EPS)
               return qcur;
     }

     // iterations not successful
     return initapp;
}

//Function used to bracket a local maximum of the function l_down
//teh first row parameters are necessarly to ruturn the bracket maximum and the
//values of the functions
//the second row are the parameters used by the l_down function
void mnbrak_niu_vector(double& ax, double& bx, double& cx, double& fa,
double& fb, double& fc,double **matrice,int n,int indice,double *miu,
double *sigma)
{
     double ulim,u,r,q,fu,dum;
```

```
     fa=-l_down(matrice,ax,n,indice,miu,sigma);
     fb=-l_down(matrice,bx,n,indice,miu,sigma);

 if (fb > fa) {
         SHFT(dum,ax,bx,dum)
         SHFT(dum,fb,fa,dum)
 }

     cx=(bx)+GOLD*(bx-ax);
     fc=-l_down(matrice,cx,n,indice,miu,sigma);

 while (fb > fc) {
            r=(bx-ax)*(fb-fc);
            q=(bx-cx)*(fb-fa);
            u=(bx)-((bx-cx)*q-(bx-ax)*r)/(2.0*SIGN(FMAX(fabs(q-r),TINY),q-r));
            ulim=(bx)+GLIMIT*(cx-bx);

            if ((bx-u)*(u-cx) > 0.0) {
                fu=-l_down(matrice,u,n,indice,miu,sigma);
                if (fu < fc) {
                    ax=(bx);
                    bx=u;
                    fa=(fb);
                    fb=fu;
                    return;
} else if (fu > fb) {
                           cx=u;
                           fc=fu;
                           return;
}
                u=(cx)+GOLD*(cx-bx);
                fu=-l_down(matrice,u,n,indice,miu,sigma);
} else if ((cx-u)*(u-ulim) > 0.0) {
                       fu=-l_down(matrice,u,n,indice,miu,sigma);
                       if (fu < fc) {
                           SHFT(bx,cx,u,cx+GOLD*(cx-bx))
                           SHFT(fb,fc,fu,-l_down(matrice,u,n,indice,miu,sigma))
   }
} else if ((u-ulim)*(ulim-cx) >= 0.0) {
                        u=ulim;
                        fu=-l_down(matrice,u,n,indice,miu,sigma);
```

```
} else {
                    u=(cx)+GOLD*(cx-bx);
                    fu=-l_down(matrice,u,n,indice,miu,sigma);
}
            SHFT(ax,bx,cx,u)
            SHFT(fa,fb,fc,fu)
 }
}



//Function for computing the  global maximum of the l_down function
//This is necessarly to find the niu_vector values by maximising the l_down funtion
//as above the parameters are also from the l_down function
double brent_niu_vector(double ax, double bx, double cx, double tol,double& xmin,
 double **matrice,int n,int indice,double *miu, double *sigma)
{
     int iter;
     double a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
     double e=0.0;

    a=(ax < cx ? ax : cx);
    b=(ax > cx ? ax : cx);
    x=w=v=bx;
    fw=fv=fx=-l_down(matrice,x,n,indice,miu,sigma);;

for (iter=1;iter<=ITMAX;iter++) {
         xm=0.5*(a+b);
         tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
         if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
             xmin=x;
             return fx;
 }
         if (fabs(e) > tol1) {
             r=(x-w)*(fx-fv);
             q=(x-v)*(fx-fw);
             p=(x-v)*q-(x-w)*r;
             q=2.0*(q-r);
             if (q > 0.0) p = -p;
             q=fabs(q);
```

```
            etemp=e;
            e=d;
            if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
            d=CGOLD*(e=(x >= xm ? a-x : b-x));

            else {
                  d=p/q;
                  u=x+d;
                  if (u-a < tol2 || b-u < tol2)
                      d=SIGN(tol1,xm-x);
 }
 } else {
                d=CGOLD*(e=(x >= xm ? a-x : b-x));
 }
        u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
        fu=-l_down(matrice,u,n,indice,miu,sigma);

        if (fu <= fx) {
            if (u >= x) a=x;
 else b=x;
            SHFT(v,w,x,u)
            SHFT(fv,fw,fx,fu)
 } else {
        if (u < x) a=u;
 else b=u;
        if (fu <= fw || w == x) {
            v=w;
            w=u;
            fv=fw;
            fw=fu;
 } else
 if (fu <= fv || v == x || v == w) {
                v=u;
                fv=fu;
 }
 }
}
   printf("Too many iterations in brent");
   xmin=x;
   return fx;
}
```

```
//Function used to bracket a local maximum of the function l
//the first row parameters are necessarly to return the bracket
//maximum and the values of the functions
//the second row are the parameters used by the l function
void mnbrak(double& ax, double& bx, double& cx, double& fa,
double& fb, double& fc,int n,int d,double **matrice,
double **P_inverse, double *miu,double *sigma,double det)
{
     double ulim,u,r,q,fu,dum;

// printf("in mnbrak\n%f %f %f\n",fa,fb,fc);
     fa=-l(n,d,matrice,ax,P_inverse,miu,sigma,det);

     fb=-l(n,d,matrice,bx,P_inverse,miu,sigma,det);

 if (fb > fa) {
         SHFT(dum,ax,bx,dum)
         SHFT(dum,fb,fa,dum)
 }

     cx=(bx)+GOLD*(bx-ax);

     fc=-l(n,d,matrice,cx,P_inverse,miu,sigma,det);

 while (fb > fc) {
            r=(bx-ax)*(fb-fc);
            q=(bx-cx)*(fb-fa);
            u=(bx)-((bx-cx)*q-(bx-ax)*r)/(2.0*SIGN(FMAX(fabs(q-r),TINY),q-r));
            ulim=(bx)+GLIMIT*(cx-bx);

            if ((bx-u)*(u-cx) > 0.0) {
                fu=-l(n,d,matrice,u,P_inverse,miu,sigma,det);
                if (fu < fc) {
                    ax=(bx);
                    bx=u;
                    fa=(fb);
                    fb=fu;
                    return;
} else if (fu > fb) {
                           cx=u;
```

```
                          fc=fu;
                          return;
}
                u=(cx)+GOLD*(cx-bx);
                fu=-l(n,d,matrice,u,P_inverse,miu,sigma,det);
} else if ((cx-u)*(u-ulim) > 0.0) {
                       fu=-l(n,d,matrice,u,P_inverse,miu,sigma,det);
                       if (fu < fc) {
                           SHFT(bx,cx,u,cx+GOLD*(cx-bx))
                           SHFT(fb,fc,fu,-l(n,d,matrice,u,P_inverse,miu,sigma,det))
   }
} else if ((u-ulim)*(ulim-cx) >= 0.0) {
                        u=ulim;
                        fu=-l(n,d,matrice,u,P_inverse,miu,sigma,det);
} else {
                    u=(cx)+GOLD*(cx-bx);
                    fu=-l(n,d,matrice,u,P_inverse,miu,sigma,det);
}
            SHFT(ax,bx,cx,u)
            SHFT(fa,fb,fc,fu)
//printf("in mnbrak\n%f %f %f\n",ax,bx,cx);
 }

//printf("in mnbrak\n%f %f %f\n",ax,bx,cx);

}

double brent(double ax, double bx, double cx, double tol,double& xmin,
int n,int dd,double **matrice,double **P_inverse, double *miu,
double *sigma,double det)
{
    int iter;
    double a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
    double e=0.0;

    a=(ax < cx ? ax : cx);
    b=(ax > cx ? ax : cx);
    x=w=v=bx;
//printf("initializare in brent\n%f %f %f \n",ax,bx,cx,x);

    fw=fv=fx=-l(n,dd,matrice,x,P_inverse,miu,sigma,det);
```

```
//printf("dupa apel l in brent\n%f %f %f %f \n",ax,bx,cx,x);

for (iter=1;iter<=ITMAX;iter++) {
         xm=0.5*(a+b);
         tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
  //printf("initializare in brent\n%f %f %f %f \n",ax,bx,cx,x);
         if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
             xmin=x;

// printf("in brent\n%f %f %f %f \n",ax,bx,cx,xmin);

 return fx;
 }
         if (fabs(e) > tol1) {
             r=(x-w)*(fx-fv);
             q=(x-v)*(fx-fw);
             p=(x-v)*q-(x-w)*r;
             q=2.0*(q-r);
             if (q > 0.0) p = -p;
             q=fabs(q);

             etemp=e;
             e=d;
             if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
             d=CGOLD*(e=(x >= xm ? a-x : b-x));

             else {
                   d=p/q;
                   u=x+d;
                   if (u-a < tol2 || b-u < tol2)
                       d=SIGN(tol1,xm-x);
 }
 } else {
                 d=CGOLD*(e=(x >= xm ? a-x : b-x));
 }
         u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
         fu=-l(n,dd,matrice,u,P_inverse,miu,sigma,det);

         if (fu <= fx) {
             if (u >= x) a=x;
```

```
 else b=x;
             SHFT(v,w,x,u)
             SHFT(fv,fw,fx,fu)
 } else {
         if (u < x) a=u;
 else b=u;
         if (fu <= fw || w == x) {
             v=w;
             w=u;
             fv=fw;
             fw=fu;
 } else
 if (fu <= fv || v == x || v == w) {
                 v=u;
                 fv=fu;
 }
 }
}
    printf("Too many iterations in brent");
    xmin=x;

// printf("\nin brent%f %f %f %f\n",ax,bx,cx,xmin);

    return fx;
}
//Function to be called by the main program
//This function reads slice by slice matrices in the given file
//and computes the algorith for them by calling the Computing function
//the matrix is given as a parameter to be filled with values
//also the dimension of the matrix are given
//n number of rows and d number of coloumns
//the reading procedure starts with reading one matrix then succesivly reading line
//by line the rest of the file until the end
//the new line is added at the end of the already read matrix by moving all the rows
//one position up and calling again the Computing function for the new obtained
//matrix
void reading_one_slice(char file_name[],double ** matrice,int d,
 int n,int n1,long seed)
{

int i,j;
```

```
float number;
//calcul
    FILE *fisier;
    if ( (fisier = fopen(file_name,"r")) == NULL)

{
printf("Can't open %s ",file_name);
        exit(1);
}

    for (i=1;i<=n;i++)
    for (j=1;j<=d;j++)
{
            fscanf(fisier,"%f", &number);
        matrice[i][j]=number;
}

    Computing(matrice,n,d,n1,seed);

 fclose(fisier);

}

//Funtion used to compute the multiplication of L and L
//transpose and return the values in P
void  matrix_multiplication(double **P,double **L, int d)
{   int i,j,k;
    double sum;

for (i = 1;i<=d; i++)
{
                    for (j=1;j<=d;j++)
{
                            sum = 0;
                            for (k=1;k<=d;k++)
{
                                   sum += L[i][k] * L[j][k];
                           }
                           P[i][j] = sum;
                    }
}
```

```
}
//Function which makes the computation for one matrix
//all the results are written in an output file
//fisrt compute the sigma and the miu
//compute the vector of niu's by maximization using the l_down function
//compute the P matrix and his inverse
//compute the NIU by maximazing the l function
//run the number generator
//this one writes 1000 computed numbers in the output file
//at last free the memory so we can start another computation for another matrix
void Computing(double **matrice, int& n , int d,int n1,long seed)
{


    FILE *fisier;

int i,j;


//define alloc and compute the average and the deviation
    double *miu,*sigma;
    miu= (double *)malloc((d+1)*sizeof(double));
    sigma= (double *)malloc((d+1)*sizeof(double));

/* Since the miu and the sigmas have been computed before and
the Computing functions has been fed a modified matrix
we can replace here by
*/ for (i=1;i<=d;i++)
    {
   miu[i]=average(i,n,matrice);
   sigma[i]=deviation(miu[i],i,n,matrice);
}




//define and aloc memory for niu_vector
double *niu_vector;
    niu_vector= (double *)malloc((d+1)*sizeof(double));
```

```
//compute the niu values by maximization of f_down
    double ax,cx,bx;
    double fa,fb,fc;
    double xmin;






for (i=1;i<=d;i++)
{
        ax=2.00;
bx=200.0;
        fa=-l_down(matrice,ax,n,i,miu,sigma);
        fb=-l_down(matrice,bx,n,i,miu,sigma);
        fc=-l_down(matrice,cx,n,i,miu,sigma);
        mnbrak_niu_vector(ax,bx,cx,fa,fb,fc,
              matrice,n,i,miu,sigma);
        brent_niu_vector(ax,bx,cx,TOL,xmin,
 matrice,n,i,miu,sigma);
        niu_vector[i]=xmin;
}

//write the niu's in the file

    if ( (fisier = fopen("output_file_niu.txt","a")) == NULL)

{
printf("Can't open %s ","output_file_niu.txt");
        exit(1);
}
 //   fprintf(fisier,"niu vector\n");
for (j=1;j<=d;j++)
  fprintf(fisier,"%f  ", niu_vector[j]);
    fprintf(fisier,"\n");
    fclose(fisier);

//compute and alloc the U
     double **U;
     U = (double **)malloc((n+1) * sizeof(double *));
     if(U == NULL)
```

```
 {
         printf("out of memory\n");
 }
     for( i = 1; i <= n; i++)
 {
        U[i] = (double *)malloc((d+1) * sizeof(double));
        if(U[i] == NULL)
{
           printf("out of memory\n");
}
 }

transf_prices_u(matrice, U,n,d,miu,sigma, niu_vector);
double *miu_0,*sigma_0;
    miu_0= (double *)malloc((d+1)*sizeof(double));
    sigma_0= (double *)malloc((d+1)*sizeof(double));
for (i=1;i<=d;i++)
    {
   miu_0[i]=0;
   sigma_0[i]=1;
}

//define and aloc memory for P
  double **P;
P = (double **)malloc((d+1) * sizeof(double *));
    if(P == NULL)
{
         printf("out of memory\n");
}
  for( i = 1; i <= d; i++)
{
        P[i] = (double *)malloc((d+1) * sizeof(double));
        if(P[i] == NULL)
{
           printf("out of memory\n");

}
}

//define and compute P_star
    double **P_s;
```

```
P_s = (double **)malloc((d+1) * sizeof(double *));
    if(P_s == NULL)
{
         printf("out of memory\n");

}
    for(i = 1; i <= d; i++)
{
        P_s[i] = (double *)malloc((d+1) * sizeof(double));
        if(P_s[i] == NULL)
{
           printf("out of memory\n");

}
}

P_star(P_s,matrice, d, n);

//compute P
//define the auxiliary p
//define the L factor of decomposition of P
//also computed here is the determinant of P
     double *p;

     if ((p= (double *)malloc((d+1) * sizeof(double)))==NULL)
 printf("out of memory");

     double **L;
 L = (double **)malloc((d+1) * sizeof(double *));
     if(L == NULL)
 {
         printf("out of memory\n");

 }
     for( i = 1; i <= d; i++)
 {
        L[i] = (double *)malloc((d+1) * sizeof(double));
        if(L[i] == NULL)
{
           printf("out of memory\n");
}
```

```
 }

transforming_semiposdef(P_s,d, p);
//take the L from P_star
     double det=1;
     for (i=1;i<=d;i++)
for(j=1;j<=d;j++)
if (j<i)
L[i][j]=P_s[i][j];
else L[i][j]=0;

for (i=1;i<=d;i++) {

L[i][i]=p[i];
det=det*L[i][i]*L[i][i];
}

//use LL^T to compute P and write it in the output file
 matrix_multiplication(P,L,d);
 if ( (fisier = fopen("output_file_P.txt","a")) == NULL)
 {
printf("Can't open %s ","output_file_P.txt");
        exit(1);
 }
     for (i=1;i<=d;i++)
 {
  for (j=1;j<=d;j++)
            fprintf(fisier,"%f ", P[i][j]);
        fprintf(fisier,"\n");
 }

//one free line between matrices
fprintf(fisier,"\n");

fclose(fisier);


//compute  the P inverse using the L inverse
double **inverse_P_computed;

inverse_P_computed= (double **)malloc((d+1) * sizeof(double *));
```

```
  if(inverse_P_computed == NULL)
 {
         printf("out of memory\n");

 }
     for( i = 1; i <= d; i++)
 {
        inverse_P_computed[i] = (double *)malloc((d+1) * sizeof(double));
        if(inverse_P_computed[i] == NULL)
{
           printf("out of memory\n");

}
 }

double **inverse_L_computed;
 inverse_L_computed= (double **)malloc((d+1) * sizeof(double *));
    if(inverse_L_computed == NULL)
{
         printf("out of memory\n");

}
    for( i = 1; i <= d; i++)
{
        inverse_L_computed[i] = (double *)malloc((d+1) * sizeof(double));
        if(inverse_L_computed[i] == NULL)
{
           printf("out of memory\n");

}
}

double **L_aux;
L_aux= (double **)malloc((d+1) * sizeof(double *));
    if(L_aux == NULL)
{
         printf("out of memory\n");

}
    for( i = 1; i <= d; i++)
{
```

```
        L_aux[i] = (double *)malloc((d+1) * sizeof(double));
        if(L_aux[i] == NULL)
{
           printf("out of memory\n");

}
}
for(i=1;i<=d;i++)
for(j=1;j<=d;j++)
L_aux[i][j]=L[i][j];

  inverse_computing(d,L_aux,inverse_L_computed);



//computing the P_inverse
double sum;
    int k;

for (i = 1;i<=d; i++)
{
    for (j=1;j<=d;j++)
{
             sum = 0;
             for (k=1;k<=d;k++)
{
                sum +=inverse_L_computed[j][k] * inverse_L_computed[i][k];
              }
              inverse_P_computed[i][j] = sum;
           }
}

//define niu and compute it by maximization
double niu;
      ax=2.0;
      bx=200.0;

  fa=-l(n,d,U,ax,inverse_P_computed,miu_0,sigma_0,det);
fb=-l(n,d,U,bx,inverse_P_computed,miu_0,sigma_0,det);
  fc=-l(n,d,U,cx,inverse_P_computed,miu_0,sigma_0,det);
```

```
//printf("in mnbrak\n%e %e %e\n",fa,fb,fc);
//printf("initial\n%f %f %f\n",ax,bx,cx,niu);

mnbrak(ax, bx,cx,fa,fb,fc,n,d,U,inverse_P_computed,miu_0,sigma_0,det);
//printf("dupa nmbrak\n%f %f %f\n",ax,bx,cx,niu);

brent(ax,bx,cx,TOL,niu,n,d,U,inverse_P_computed,miu_0,sigma_0,det);
//printf("dupa brent\n%f %f %f\n",ax,bx,cx,niu);


//write the niu in the output file
if ( (fisier = fopen("output_file.txt","a")) == NULL)
{
printf("Can't open %s ","output_file.txt");
        exit(1);
}

  //  fprintf(fisier,"NIU\n");
    fprintf(fisier,"%f\n ",niu);

// fprintf(fisier,"simulated vectors\n");
fclose(fisier);

//simulation part
    double *simulation_vector;
    simulation_vector= (double *)malloc((d+1) * sizeof(double));


printf("Computing one matrix finished; entering generator \n");

generator(L,d,n1,niu,niu_vector,miu,sigma,simulation_vector,seed);
free(miu);
free(sigma);
free(miu_0);
free(sigma_0);

//free(niu_vector);

free(P);
free(P_s);
free(U);
```

```
free(p);
free(L);
free(simulation_vector);
free(inverse_P_computed);
free(L_aux);
free(inverse_L_computed);
}

//function to compute the sign of one real number
int sign(double x)
{
 int result;
  if (x==0) result=0;
  else
  if (x<0) result=-1;
  else result=1;
return result;
}


//used for computing P matrix
double ro_theta(int i, int k, int n, double **matrice)
{
   double result=0;
   int i1,i2;
   double sume=0;
//for the diagonal to be equal to 1
   if (i==k)
    return 1;
   for (i1=1;i1<=n;i1++)
   for(i2=i1;i2<=n;i2++)
     sume=sume+sign((matrice[i1][i]-matrice[i2][i])*
     (matrice[i1][k]-matrice[i2][k]));
//factorial computed inside the formula
   result=2*sume/(n-1)*n;
   return result;
}
//used for computing P matrix
void P_star(double **P, double **matrice, int d, int n)
{
  int i,j;
```

```
  for (i=1;i<=d;i++)
  for (j=1;j<=i;j++)
  P[i][j]=sin((PI/2)*ro_theta(i,j,n,matrice));
  for (i=1;i<=d;i++)
  for (j=i+1;j<=d;j++)
  P[i][j]=P[j][i];
}

// logarithm of gamma function for xx > 0
//used int computing the l_down and l of a niu value
double gammaln(double xx)
{
     double cof[6] = {
          76.18009172947146, -86.50532032941677, 24.01409824083091,
          -1.231739572450155, 0.1208650973866179e-2, -0.5395239384953e-5
                    };
     double fact = 2.5066282746310005, x, result, temp, ser;
     int i;
     // checks
     if ((dllislessorequal(xx, 0.0)) || XisNegInf(xx))
          return dnan;
     if (XisPosInf(xx))
          return xx;
     if (dllisequal(xx, 1.0))
          return 0.0;
     if (dllisequal(xx, 0.5))
          return -x4dlllog(M_1_SQRTPI);
     // computation
     if (xx > 1.0)
     { // here we use gamma(x+1)=x*gamma(x)
          x = xx - 1.0;
          result = 0.0;
     }
     else
     {  // use approximation directly
          x = xx;
          result = -x4dlllog(xx);
     }
     temp = x + 5.5;
     temp -= (x + 0.5) * x4dlllog(temp);
     ser = 1.000000000190015;
```

```
     for (i = 0; i < 6; i++)
     {
          x = x + 1.0;
          ser += cof[i] / x;
     }
     result += -temp + x4dlllog(fact * ser);
 return result;
}

//function for multiplying two vectors
double vector_mult(double *vect1, double *vect2, int dim)
{
  double result=0;
  int i;

// for (i=1;i<=dim;i++)
//   printf("%f %f\n",vect1[i],vect2[i]);

  for (i=1;i<=dim;i++)
  result=result+vect1[i]*vect2[i];
  //printf("\n in vector mul %f",result);
  return result;
}

//used by heapSort
void siftDown(double *numbers, int root, int bottom)
{
  int done, maxChild;
  double temp;
  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;
    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
```

```
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}

//Function for sorting in a increasing order a vector of numbers
void heapSort(double *numbers, int array_size)
{
  int i;
  double temp;
  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);
  for (i = array_size-1; i >= 1; i--)
  {
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}


//choses the scalar which has risk % smaller than him
double choose_scalar(double *numbers,int size, double risk)
{
 double result;
     result=risk * size/100;
 return numbers[(int) result];
}

//function to obtain the LU decomposition of a matrix
//used in computing the inverse of a matrix
//for us in computing the inverse of L
void ludcmp(double **a, int n, int *indx, double *d)
{
     int i,imax,j,k;
     double big,dum,sum,temp;
```

```
    double *vv;
vv= (double *)malloc((n+1) * sizeof(double));
    *d=1.0;
for (i=1;i<=n;i++) {
         big=0.0;
         for (j=1;j<=n;j++)
              if ((temp=fabs(a[i][j])) > big) big=temp;
         if (big == 0.0) printf("Singular matrix in routine ludcmp");
         vv[i]=1.0/big;
}
    for (j=1;j<=n;j++) {
         for (i=1;i<j;i++) {
              sum=a[i][j];
              for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
              a[i][j]=sum;
 }
    big=0.0;
    for (i=j;i<=n;i++) {
         sum=a[i][j];
         for (k=1;k<j;k++)

              sum -= a[i][k]*a[k][j];
         a[i][j]=sum;
         if ( (dum=vv[i]*fabs(sum)) >= big) {
               big=dum;
               imax=i;
 }
}
    if (j != imax) {
        for (k=1;k<=n;k++) {
             dum=a[imax][k];
             a[imax][k]=a[j][k];
             a[j][k]=dum;
}
        *d = -(*d);
        vv[imax]=vv[j];
}
    indx[j]=imax;
    if (a[j][j] == 0.0) a[j][j]=TINY;
    if (j != n) {
        dum=1.0/(a[j][j]);
```

```
        for (i=j+1;i<=n;i++) a[i][j] *= dum;
 }
 }
     free(vv);
}

//Function which solves the Ax=B system of equations
//used for computing the inverse of a matrix
void lubksb(double **a, int n, int *indx, double b[])
{
     int i,ii=0,ip,j;
     double sum;
     for (i=1;i<=n;i++) {
          ip=indx[i];
          sum=b[ip];
          b[ip]=b[i];
          if (ii)
              for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
          else if (sum) ii=i;
          b[i]=sum;
 }
     for (i=n;i>=1;i--) {
          sum=b[i];
          for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
          b[i]=sum/a[i][i];
 }
}

//Function for computing the inverse of a matrix
//with this we compute the inverse of L for obtaining the inverse of P
void inverse_computing(int dim, double **matrice, double ** inverse)
{
   double *col,d;
   int i,j,*indx;
   indx= (int *)malloc((dim+1)*sizeof(int));
   col= (double *)malloc((dim+1)*sizeof(double));
   ludcmp(matrice,dim,indx,&d);      //Decompose the matrix just once.
   for(j=1;j<=dim;j++)
   {                                //Find inverse by columns.
       for(i=1;i<=dim;i++) col[i]=0.0;
       col[j]=1.0;
```

```
       lubksb(matrice,dim,indx,col);
       for(i=1;i<=dim;i++) inverse[i][j]=col[i];
   }
   free(indx);
   free(col);
}

//Function for computing the determinant of a mtrix
//not used for the moment in the program
double determinant(int size, double **matrice)
{
   double d;
   int j,*indx;
   indx= (int *)malloc((size+1)*sizeof(int));
   ludcmp(matrice,size,indx,&d);
   for(j=1;j<=size;j++) d *= matrice[j][j];
   free(indx);
   return d;
}

//Function for computing the LL^T decomposition of a matrix
//used to obtain the L matrix from the P
int choldc(double **matrice, int n, double *p)
/*Given a positive-de.nite symmetric matrix a[1..n][1..n], this routine constructs it
decomposition, A = L · LT . On input, only the upper triangle of a need be given; it i
modi.ed. The Cholesky factor L is returned in the lower triangle of a, except for its
elements which are returned in p[1..n].
*/
{
 // void nrerror(char error_text[]);
  int i,j,k,result;
  double sum;
  result=0;
  for (i=1;i<=n;i++)
  {
     for (j=i;j<=n;j++)
 {
         for (sum=matrice[i][j],k=i-1;k>=1;k--)
 sum -= matrice[i][k]*matrice[j][k];
         if (i == j)
 {
```

```
            if (sum <= 0.0)
{ //a, with rounding errors, is not positive de.nite.
                 // printf("choldc failed");
 result=1;
}
p[i]=sqrt(sum);
 }
 else matrice[j][i]=sum/p[i];
 }
  }
return result;
}

//Function for computing the multiplication of a matrix with a scalar
//returns the new matrix in the same matrix given as parameter
void scalar_multiplication(double **matrice, int n, double scalar)
{
int i,j;
for (i=1;i<=n;i++)
for (j=1;j<=n;j++)
matrice[i][j]=matrice[i][j]*scalar;
}

//Funtion for computing the sum of 2 matrices
//the result is obtain in the first matrix given as parameter
void matrice_adding(double **matrice1, double **matrice2, int n)
{
int i,j;
    for (i=1;i<=n;i++)
for (j=1;j<=n;j++)
matrice1[i][j]=matrice1[i][j]+matrice2[i][j];
}

//Funtion to reobtain the identity matrix
void reset_identity(double **matrice, int n)
{
int i,j;
    for (i=1;i<=n;i++)
for (j=1;j<=n;j++)
if (i==j)
    matrice[i][j]=1;
```

```
else
matrice[i][j]=0;
}

//Function for transforming a metix in a positive definite one
//uses the test of Cholesky decomposition
//if a mtrix is positive definite then it can be decomposed with
//the Cholesky method

void transforming_semiposdef(double **matrice,int n, double *p)
{
double **identity;
    int i;
identity= (double **)malloc((n+1) * sizeof(double *));
    if(identity == NULL)
{
         printf("out of memory\n");
}
    for( i = 1; i <= n; i++)
{
        identity[i] = (double *)malloc((n+1) * sizeof(double));
        if(identity[i] == NULL)
{
           printf("out of memory\n");
}
}
double lambda,result,delta=0.05;
  int test_redundanta=0;
reset_identity(identity,n);
    result=choldc(matrice,n,p);
    lambda=0.01;
while ((result==1) && (test_redundanta<=100))
{
scalar_multiplication(matrice,n,lambda);
scalar_multiplication(identity,n,1-lambda);
matrice_adding(matrice,identity,n);
reset_identity(identity,n);
lambda=lambda+0.01;
        result=choldc(matrice,n,p);
    test_redundanta++;
}
```

```
}

//Funtion used to compute the miu as average of a coloumn in the matrix of prices
double average(int indice, int n, double **matrice)
{
  int j;
  double result=0;
  for (j=1;j<=n;j++)
     result=result+matrice[j][indice];
  return result/n;
}

//Function used to compute the sigma as the deviation from the matrix of prices
//returns sigma squared since we only use that
double deviation(double miu,int indice, int n, double **matrice)
{
  double result=0;
  int j;
  for (j=1;j<=n;j++)
  result=result+pow(matrice[j][indice]-miu,2);
  result=result/n;
  return result;
}

//Function for computing the l_down for a given niu and an indice which is
//the coloumn representive for that niu
//if the values computed are not in the range of 2 and 200 then it
//returns a small value since
//we need to compute the value for which the maximum is obtained
//in this range

double l_down(double **matrice, double niu,int n, int indice,
double *miu, double *sigma)
{
   int j;
   double sum=0;
   if ((niu>200)||(niu<2)) sum=-1E10;
   else
   for (j=1;j<=n;j++)
    sum+=f_down(matrice,niu,matrice[j][indice],indice,n,miu,sigma);
   return sum;
```

```
}

//Funtion used by the l_down
//statistic function
double f_down(double **matrice, double niu, double x, int indice,
int n,double *miu, double *sigma)
{
double result,miu_x,sigma_x;
miu_x=miu[indice];
sigma_x=sigma[indice];
result=gammaln((niu+1)/2)-gammaln(niu/2)-log(PI*niu*sigma_x)/2;
result=result-((1+niu)/2)*log(1+(x-miu_x)*(x-miu_x)/(niu*sigma_x));
return result;
}

//Statistic funtion used for computing the l for NIU
double f_up(double niu, double **inverse_P,double **matrice,
int indice, int d,double det)
{
double result;
    double *vector;
    vector= (double *)malloc((d+1)*sizeof(double));
int i;
for (i=1;i<=d;i++)
vector[i]=t_inverse(matrice[indice][i],niu);
//for (i=1;i<=d;i++)
  //printf("%f\n",vector[i]);
result=gammaln((niu+d)/2)-gammaln(niu/2)-(d*log(PI*niu)/2)-(log(det)/2);
//printf("\ngammaln %f \n",result);
//--------------------------------------aici se fute meciul
float info=vector_matrix_multiplication(d,inverse_P,vector);
result=result -
log(1+vector_matrix_multiplication(d,inverse_P,vector)/niu)*(d+niu)/2;
//printf("\ninfo %f \n",info);
free(vector);
return result;
}

//Funtion for obtaining the U matrix from the matix of prices
void transf_prices_u(double **matrice, double **U, int n,
int d, double *miu, double *sigma, double *niu_vector)
```

```
{
   int i,j;
   for (i=1;i<=d;i++)
   for (j=1;j<=n;j++)
   {
  //printf("\n%f",sqrt(sigma[i]));
   U[j][i]=t((matrice[j][i]- miu[i])/sqrt(sigma[i]), niu_vector[i]);
   }
}

// initial approximation of the quantile
// prec - returns 1 if the result is precise
double sqfn_init(double p, int *prec)
{
     double p0, p1, p2, p3, p4, y, q0, q1, q2, q3, q4, xp, k;
     *prec = 1;     // the following answers are precise
     k   = p;
     // constants for approximation
     p0  = -0.322232431088;
     p1  = -1.0;
     p2  = -0.342242088547;
     p3  = -0.0204231210245;
     p4  = -0.453642210148e-4;
     q0  = 0.0993484626060;
     q1  = 0.588581570495;
     q2  = 0.531103462366;
     q3  = 0.103537752850;
     q4  = 0.38560700634e-2;
     // check for special values
     if ( XisNaN(p) || XisNegInf(p) || XisPosInf(p) )
          return dnan;
     if (dlliszero(p))
          return -dinf;
     if (dllisequal(p, 1.0))
          return dinf;
     if (dllislessorequal(p, 0.0) || dllisgreaterorequal(p, 1.0))
          return dnan;
     if (dllisequal(p, 0.5))
          return 0.0;
     *prec = 0;     // the following answers are approximation
     // approximation itself
```

```
    if (p > 0.5)
        p = 1.0 - p;
    y = sqrt(x4dlllog(1.0 / (p*p)));
    xp = y + ((((y * p4 + p3) * y + p2) * y + p1) * y + p0) /
    ((((y * q4 + q3) * y + q2) * y + q1) * y + q0);
    if (k < 0.5)
        xp = - xp;
    return(xp);
}

double gammacf(double x, double a, double *gln)
{
    int i;
    double an, b, c, d, del, h;
    // checks
    if (dlliszero(x))
        return (dllisgreater(a, 0.0)) ? 0.0 : dnan;
    if (dllislessorequal(x, 0.0))
        return dnan;
    // computation
    *gln = gammaln(a);
    b = x + 1.0 - a;
    c = 1.0 / G_DBLMIN;
    d = 1.0 / b;
    h = d;
    for (i = 1; i <= G_ITMAX; i++)
    {
        an = -i * (i - a);
        b += 2.0;
        d = an * d + b;
        if (abs(d) < G_DBLMIN)
            d = G_DBLMIN;
        c = b + an / c;
        if (abs(c) < G_DBLMIN)
            c = G_DBLMIN;
        d = 1.0 / d;
        del = d * c;
        h *= del;
        if (abs(del - 1.0) < G_EPS)
            return h * x4dllexp(-x + a*x4dlllog(x) - *gln);
    }
```

```
    return dnan;
}
double gammaser(double x, double a, double *gln)
{
    int n;
    double sum, del, ap;
    // checks
    if (dlliszero(x))
        return 0.0;
    if (dllislessorequal(x, 0.0))
        return dnan;
    // init
    *gln = gammaln(a);
    ap  = a;
    del = sum = 1.0 / a;
    // computation
    for (n = 1; n <= G_ITMAX; n++)
    {
        ap += 1.0;
        del *= x / ap;
        sum += del;

        if (abs(del) < abs(sum) * G_EPS)
            return sum * x4dllexp(-x + a*x4dlllog(x) - *gln);
    }

    if (dllisgreaterorequal(a, 0.0))
        return dinf;
    else return dnan;
}

double gammap(double x, double a)
{
    double gln;

    if ( dllislessorequal(a, 0.0)  || XisNaN(x) )
        return dnan;
    if ( XisPosInf(x) )
        return 1.0;
    if ( dllisless(x, 0.0)  || XisNegInf(x) )
        return 0.0;
```

```
    if ( x < a + 1.0 )
        return gammaser(x, a, &gln);
    else return 1.0 - gammacf(x, a, &gln);
}

double scdfn(double x)
{
    // check extreme value of x
    if (XisNaN(x))
        return dnan;
    else if (XisPosInf(x))
        return 1.0;
    else if (XisNegInf(x))
        return 0.0;
    if (x >= 0.0)
        return (1.0 + gammap(x*x/2.0, 0.5)) / 2.0;
    else return (1.0 - gammap(x*x/2.0, 0.5)) / 2.0;
}

// normal pdf
double spdfn(double x)
{
    return x4dllexp(-x*x/2) * M_1_SQRTPI * M_SQRT_2;
}

// normal quantile via Newton method
double sqfn(double p)
{
    double initapp, qcur, dx;
    int i, prec;
    initapp = sqfn_init(p, &prec);
    if (prec)
        return initapp;
    // iterate
    qcur = initapp;
    for(i = 1; i <= NEWTON_ITMAX; i++)
    {
        dx = (scdfn(qcur) - p) / spdfn(qcur);
        qcur -= dx;
        if (abs(dx / qcur) < NEWTON_EPS)
```

```
                return qcur;
     }
     // iterations not successful
     return initapp;
}

// Student distribution inversion
// initial approximation
// prec - returns 1 if the result is precise
double sqft_init(double p, double df, int *prec)
{
     double q, c, t, res;
     int dummy;
     *prec = 1;
     // check for special values
     if ( XisNaN(p) || XisNegInf(p) || XisPosInf(p) ||
          XisNaN(df) || dllislessorequal(df, 0.0) || XisNegInf(df))
//  {
//printf("ramura0%f \n",dnan);
 return dnan;
//  }
if (dlliszero(p))
// {
// printf("ramura 1%f \n",-dinf);
return -dinf;
// }
if (dllisequal(p, 1.0))
// {
// printf("ramura2%f \n",dinf);
return dinf;
// }
     if (dllislessorequal(p, 0.0) || dllisgreaterorequal(p, 1.0))
//  {
        // printf("ramura3 %f  %f \n",p,dnan);
 return dnan;
//  }
     if (dllisequal(p, 0.5))
 return 0.0;
     if (XisPosInf(df))
//  {
// printf("ramura4%f \n",sqfn(p));
```

```
 return sqfn(p);
//  }
     *prec = 0;
     if (dllisequal(df, 1.0))
  //   {
//  printf("ramura5%f \n",(tan(PI*(p-0.5))));
          return (tan(PI*(p-0.5)));
     //}
     if (dllisequal(df, 2.0))
     {
          t = 2.0*p - 1.0;
       //   printf("ramura6%f \n",(M_SQRT2 * t / sqrt(1.0 - t*t)));
          return (M_SQRT2 * t / sqrt(1.0 - t*t));
     }
     q = sqfn_init(p, &dummy);
     c = df - 2.0/3.0 + 1.0/(10.0*df);
     c = (df - 5.0/6.0) / (c*c);
     res = sqrt( df * x4dllexp(c*q*q) - df );
     if (p <= 0.5)
          res = -res;
     //printf("final%f \n",res);
     return res;
}

double betacf(double x, double a, double b)
{
     double aa, c, d, del, h, qab, qam, qap;
     int m, m2;
     qab = a + b;
     qap = a + 1.0;
     qam = a - 1.0;
     c = 1.0;
     d = 1.0 - qab*x/qap;
     if (abs(d) < B_DBLMIN)
               d = B_DBLMIN;
     d = 1.0 / d;
     h = d;
     for (m = 1; m <= B_ITMAX; m++)
     {
          m2 = 2 * m;
          aa = m * (b-m) * x / ((qam+m2) * (a+m2));
```

```
        d = 1.0 + aa*d;
        if (abs(d) < B_DBLMIN)
             d = B_DBLMIN;
        c = 1.0 + aa/c;
        if (abs(c) < B_DBLMIN)
             c = B_DBLMIN;
        d = 1.0 / d;
        h *= d * c;
        aa = -(a+m) * (qab+m) * x / ((a+m2) * (qap+m2));
        d = 1.0 + aa * d;
        if (abs(d) < B_DBLMIN)
             d = B_DBLMIN;
        c = 1.0 + aa/c;
        if (abs(c) < B_DBLMIN)
             c = B_DBLMIN;
        d = 1.0 / d;
        del = d * c;
        h *= del;
        if (abs(del - 1.0) < B_EPS)
             return h;
    }

    return dnan;
}

double betai(double x, double a, double b)
{
    double bt;
    // checks
    if (dllisless(x, 0.0) || dllisgreater(x, 1.0)
    || XisNaN(x) || XisPosInf(x) || XisNegInf(x))
         return dnan;
    if (dlliszero(x) || dllisequal(x, 1.0))
         bt = 0.0;
    else bt = x4dllexp(gammaln(a+b) - gammaln(a) -
    gammaln(b) + a*x4dlllog(x) + b*x4dlllog(1.0-x));
    if (x < (a+1.0) / (a+b+2.0))
         return bt * betacf(x, a, b) / a;
    else return 1.0 - bt * betacf(1.0-x, b, a) / b;
}
```

```
//the t function used for thransforming the matrix
//of prices and for lot more
double t(double x, double v)
{
     // checkV extreme value of x
     if (XisNaN(x) || XisNaN(v) || dllislessorequal(v, 0.0) || XisNegInf(v))
          return dnan;
     else if (XisPosInf(x))
          return 1.0;
     else if (XisNegInf(x))
          return 0.0;
     else if (XisPosInf(v))
          return scdfn(x);
     if (x <= 0)
          return betai(v / (v + x*x), v / 2.0, 0.5) / 2.0;
     else return 1.0 - betai(v / (v + x*x), v / 2.0, 0.5)/2.0;
}

//the inverse of t function
double t_inverse(double p, double df)
{
     double initapp, qcur, dx;
     int i, prec;

// printf(" \n p %f",p);
     initapp = sqft_init(p, df, &prec);
    // printf("%f \n",initapp);
 if (prec) {
 //printf("%f \n",initapp);
          return initapp;
 }
     // iterate
     qcur = initapp;
     for(i = 1; i <= NEWTON_ITMAX; i++)
     {
          dx = (t(qcur, df) - p) / t(qcur, df);
          qcur -= dx;
          if (abs(dx / qcur) < NEWTON_EPS)
  {
  //printf("%f \n",qcur);
  return qcur;
```

```
  }
  }
     // iterations not successful
    // printf("%f \n",initapp);
     return initapp;
}

//Function used for computing the l for a given niu
double c(double niu, double **P_inverse,double **matrice,int n,
int d,int indice, double *miu, double *sigma,double det)
{
  double suma=0;

double aux=0;
    int i;
for (i=1;i<=d;i++)
{
suma=suma+f_down(matrice,niu,t_inverse(matrice[indice][i],niu),i,n, miu,sigma);
}
aux=f_up(niu,P_inverse,matrice,indice,d,det)-suma;
return aux;
}

//The l function used for obtaining the NIU value in which
//this function obtaines a maximum
//is computed as the sum of c funtions on a row
//same as in l_down we have the test for the point of
//extreme to be obtain in the range 2 - 200
double l(int n, int d, double **matrice, double niu,double
//**P_inverse,double *miu, double *sigma,double det)
{
  int i;
  double result=0;
 if ((niu>200) || (niu<2))
 {
 result= -1.7E308;
 //printf("in l\nniu: %f %e\n\n",niu,result);
 }
  else
  {
  for(i=1;i<=n;i++)
```

```
    result=result+c(niu,P_inverse,matrice,n,d,i, miu,sigma,det);
 // printf("in l\n%e\n\n",result);
  }
  return result;
}


//Funtion used in the f_up function for computing the x^tP^(-1)x product
//returns a scalar
//computes first the first matrix multiplication between the x^t and the matrix
//and after that it calls the vector multiplication from above
double vector_matrix_multiplication(int size, double **matrice, double *vector)
{
   double *aux,result;
   aux= (double *)malloc((size+1)*sizeof(double));
   int i,j;
   for (i=1;i<=size;i++)
   {
   aux[i]=0;
       for (j=1;j<=size;j++)
   aux[i]=aux[i]+matrice[j][i]*vector[j];
   }
   result=vector_mult(aux,vector,size);
   free(aux);
   return result;
}

//Function for computing the multiplication between a matrix and a vector
//returns the value in vector2
//used in the random_generator function
void vector_matrix_one_way_multiplication(int size,
double **matrice, double *vector1, double *vector2)
{
   int i,j;
   for (i=1;i<=size;i++)
   {
   vector2[i]=0;
       for (j=1;j<=size;j++)
   vector2[i]=vector2[i]+matrice[j][i]*vector1[j];
   }
}
```

```
//Function for generating a random number using a seed given as a parameter
double ran2(long *idum)
{
     int j;
     long k;
     static long idum2=1234567891023;
     static long iy=0;
     static long iv[NTAB];
     double temp;
 if (*idum <= 0)
 {

 if (-(*idum) < 1) *idum=1;
         else *idum = -(*idum);
         idum2=(*idum);
         for (j=NTAB+7;j>=0;j--)
 {
              k=(*idum)/IQ1;
              *idum=IA1*(*idum-k*IQ1)-k*IR1;
              if (*idum < 0) *idum += IM1;
              if (j < NTAB) iv[j] = *idum;
 }
         iy=iv[0];
 }
     k=(*idum)/IQ1;
     *idum=IA1*(*idum-k*IQ1)-k*IR1;

     if (*idum < 0) *idum += IM1;
         k=idum2/IQ2;
         idum2=IA2*(idum2-k*IQ2)-k*IR2;
     if (idum2 < 0) idum2 += IM2;
 j=iy/NDIV;
     iy=iv[j]-idum2;

 iv[j] = *idum;
     if (iy < 1) iy += IMM1;
     if ((temp=AM*iy) > RNMX) return RNMX;
     else return temp;
}
```

```
//Function for generating k ramndom numbers using the same seed
//used for geneting our ramndom numbers from the simulation part
void random_generator(int k, double *numbers,long seed)
{
  int i;
  //long idum;
  for (i=1;i<=k;i++)
  {
  //printf("%d\n",seed);
  numbers[i]=ran2(&seed);
  }
}

//used in the random_generator for transforming the numbers generated into
//a sequence of numbers to fulfill the required distribution
double z_transform(double p)
{
     double p0, p1, p2, p3, p4, y, q0, q1, q2, q3, q4, xp, k;
     k   = p;
     // constants for approximation
     p0  = -0.322232431088;
     p1  = -1.0;
     p2  = -0.342242088547;
     p3  = -0.0204231210245;
     p4  = -0.453642210148e-4;
     q0  = 0.0993484626060;
     q1  = 0.588581570495;
     q2  = 0.531103462366;
     q3  = 0.103537752850;
     q4  = 0.38560700634e-2;
     // check for special values
     if (dllisequal(p, 0.5))
          return 0.0;
     // approximation itself
     if (p > 0.5)
          p = 1.0 - p;
     y = sqrt(x4dlllog(1.0 / (p*p)));
     xp = y + ((((y * p4 + p3) * y + p2) * y + p1) * y + p0) /
     ((((y * q4 + q3) * y + q2) * y + q1) * y + q0);
     if (k < 0.5)
          xp = - xp;
```

```
     return(xp);
}

// auxiliary function for Chi2 inverse
double ppnd16_(double *p, int *ifault)
{
    /* System generated locals */
    double ret_val;
    /* Local variables */
    static double q, r;
/*  ALGORITHM AS241  APPL. STATIST. (1988) VOL. 37, NO. 3 */
/*  Produces the normal deviate Z corresponding to a given lower */
/*  tail area of P; Z is accurate to about 1 part in 10**16. */
/*  The hash sums below are the sums of the mantissas of the */
/*  coefficients.   They are included for use in checking */
/*  transcription. */
/*  Coefficients for P close to 0.5 */
/*  HASH SUM AB    55.88319 28806 14901 4439 */
/*  Coefficients for P not close to 0, 0.5 or 1. */
/*  HASH SUM CD    49.33206 50330 16102 89036 */
/*  Coefficients for P near 0 or 1. */
/*  HASH SUM EF    47.52583 31754 92896 71629 */
    *ifault = 0;
    q = *p - .5;
    if (abs(q) <= .425) {
r = .180625 - q * q;
ret_val = q * (((((((r * 2509.0809287301226727 +
33430.575583588128105) * r + 67265.770927008700853) * r +
45921.953931549871457) * r + 13731.693765509461125) * r +
1971.5909503065514427) * r + 133.14166789178437745) * r +
3.387132872796366608) / (((((((r * 5226.495278852854561 +
28729.085735721942674) * r + 39307.89580009271061) * r +
21213.794301586595867) * r + 5394.1960214247511077) * r +
687.1870074920579083) * r + 42.313330701600911252) * r + 1.);
return ret_val;
    } else {
if (q < 0.) {
    r = *p;
} else {
    r = 1. - *p;
}
```

```
if (r <= 0.) {
    *ifault = 1;
    ret_val = 0.;
    return ret_val;
}
r = sqrt(-log(r));
if (r <= 5.) {
    r += -1.6;
    ret_val = (((((((r * 7.7454501427834140764e-4 +
    .0227238449892691845833) * r + .24178072517745061177) * r
    + 1.27045825245236838258) * r + 3.64784832476320460504) *
    r + 5.7694972214606914055) * r + 4.6303378461565452959) *
    r + 1.42343711074968357734) / (((((((r *
    1.05075007164441684324e-9 + 5.475938084995344946e-4) * r
    + .0151986665636164571966) * r + .14810397642748007459) *
    r + .68976733498510000455) * r + 1.6763848301838038494) *
    r + 2.05319162663775882187) * r + 1.);
} else {
    r += -5.;
    ret_val = (((((((r * 2.01033439929228813265e-7 +
    2.71155556874348757815e-5) * r + .0012426609473880784386)
    * r + .026532189526576123093) * r + .29656057182850489123)
     * r + 1.7848265399172913358) * r + 5.4637849111641143699)
     * r + 6.6579046435011037772) / (((((((r *
    2.04426310338993978564e-15 + 1.4215117583164458887e-7) *
    r + 1.8463183175100546818e-5) * r +
    7.868691311456132591e-4) * r + .0148753612908506148525) *
    r + .13692988092273580531) * r + .59983220655588793769) *
    r + 1.);
}
if (q < 0.) {
    ret_val = -ret_val;
}
return ret_val;
    }
    return ret_val;
}

double ppchi2_(double *p, double *v, double *g, int *ifault)
{
    static double aa = .6931471806;
```

```
static double six = 6.;
static double c1 = .01;
static double c2 = .222222;
static double c3 = .32;
static double c4 = .4;
static double c5 = 1.24;
static double c6 = 2.2;
static double c7 = 4.67;
static double c8 = 6.66;
static double c9 = 6.73;
static double e = 5e-7;
static double c10 = 13.32;
static double c11 = 60.;
static double c12 = 70.;
static double c13 = 84.;
static double c14 = 105.;
static double c15 = 120.;
static double c16 = 127.;
static double c17 = 140.;
static double c18 = 175.;
static double c19 = 210.;
static double pmin = 2e-6;
static double c20 = 252.;
static double c21 = 264.;
static double c22 = 294.;
static double c23 = 346.;
static double c24 = 420.;
static double c25 = 462.;
static double c26 = 606.;
static double c27 = 672.;
static double c28 = 707.;
static double c29 = 735.;
static double pmax = .999998;
static double c30 = 889.;
static double c31 = 932.;
static double c32 = 966.;
static double c33 = 1141.;
static double c34 = 1182.;
static double c35 = 1278.;
static double c36 = 1740.;
static double c37 = 2520.;
```

```
    static double c38 = 5040.;
    static double zero = 0.;
    static double half = .5;
    static double one = 1.;
    static double two = 2.;
    static double three = 3.;
    double ret_val, d__1, d__2;
#define pow_dd(a,b) pow(*a, *b)
    /* Local variables */
    extern double ppnd_();
    static double a, b, c;
    static int i;
    static double q, t, x, p1, p2, s1, s2, s3, s4, s5, s6;
    static double ch, xx;
    static int if1;
/*          Algorithm AS 91   Appl. Statist. (1975) Vol.24, P.35 */

/*          To evaluate the percentage points of the chi-squared */
/*          probability distribution function. */

/*          p must lie in the range 0.000002 to 0.999998, */
/*          v must be positive, */
/*          g must be supplied and should be equal to */
/*            ln(gamma(v/2.0)) */

/*       Incorporates the suggested changes in AS R85 (vol.40(1), */
/*       pp.233-5, 1991) which should eliminate the need for the limited */
/*       range for p above, though these limits have not been removed */
/*       from the routine. */
/*       If IFAULT = 4 is returned, the result is probably as accurate as */
/*       the machine will allow. */
/*       Auxiliary routines required: PPND = AS 111 (or AS 241) and */
/*       GAMMAD = AS 239. */
/*          test arguments and initialise */
    ret_val = -one;
    *ifault = 1;
    if (*p < pmin || *p > pmax) {
return ret_val;
    }
    *ifault = 2;
    if (*v <= zero) {
```

```
return ret_val;
    }
    *ifault = 0;
    xx = half * *v;
    c = xx - one;
    if (*v >= -c5 * log(*p)) {
goto L1;
    }
    d__1 = *p * xx * exp(*g + xx * aa);
    d__2 = one / xx;
    ch = pow_dd(&d__1, &d__2);
    if (ch < e) {
goto L6;
    }
    goto L4;
L1:
    if (*v > c3) {
goto L3;
    }
    ch = c4;
    a = log(one - *p);
L2:
    q = ch;
    p1 = one + ch * (c7 + ch);
    p2 = ch * (c9 + ch * (c8 + ch));
    t = -half + (c7 + two * ch) / p1 - (c9 + ch * (c10 + three * ch)) / p2;
    ch -= (one - exp(a + *g + half * ch + c * aa) * p2 / p1) / t;
    if ((d__1 = q / ch - one, abs(d__1)) > c1) {
goto L2;
    }
    goto L4;
/*        call to algorithm AS 111 - note that p has been tested above. */
/*   AS 241 could be used as an alternative. */
L3:
    x = ppnd16_(p, &if1);
/*        starting approximation using Wilson and Hilferty estimate */
    p1 = c2 / *v;
/* Computing 3rd power */
    d__1 = x * sqrt(p1) + one - p1, d__2 = d__1;
    ch = *v * (d__2 * (d__1 * d__1));
/*        starting approximation for p tending to 1 */
```

```
    if (ch > c6 * *v + six) {
ch = -two * (log(one - *p) - c * log(half * ch) + *g);
    }
/*        call to algorithm AS 239 and calculation of seven term */
/*        Taylor series */
L4:
    for (i = 1; i <= 20; ++i) {
q = ch;
p1 = half * ch;
p2 = *p - gammap(p1, xx);  // ???
if (if1 == 0) {
    goto L5;
}
*ifault = 3;
return ret_val;
L5:
t = p2 * exp(xx * aa + *g + p1 - c * log(ch));
b = t / ch;
a = half * t - b * c;
s1 = (c19 + a * (c17 + a * (c14 + a * (c13 + a * (c12 + c11 * a))))) /
 c24;
s2 = (c24 + a * (c29 + a * (c32 + a * (c33 + c35 * a)))) / c37;
s3 = (c19 + a * (c25 + a * (c28 + c31 * a))) / c37;
s4 = (c20 + a * (c27 + c34 * a) + c * (c22 + a * (c30 + c36 * a))) /
c38;
s5 = (c13 + c21 * a + c * (c18 + c26 * a)) / c37;
s6 = (c15 + c * (c23 + c16 * c)) / c38;
ch += t * (one + half * t * s1 - b * c * (s1 - b * (s2 - b * (s3 - b *
 (s4 - b * (s5 - b * s6))))));
if ((d__1 = q / ch - one, abs(d__1)) > e) {
    goto L6;
}
    }
    *ifault = 4;
L6:
    ret_val = ch;
    return ret_val;
}

//Funtion to transfor the s from an arbitrary number generated in random_generator
//in a number which we can use for further computation in the simluation part
```

```
double s_transform(double p, double v)
{
     double g;
     int ifault;
     // check for special values
     if ( XisNaN(p) || XisNegInf(p) || XisPosInf(p) ||
          XisNaN(v) || dllislessorequal(v, 0.0) || XisNegInf(v) || XisPosInf(v))
          return dnan;
     if ( dlliszero(p) || ((0.0 <= p) && (p <= 0.000002)) )
          return 0.0;
     if ( dllisequal(p, 1.0) || ((0.999998 <= p) && (p <= 1.0)) )
          return dinf;
     if (dllislessorequal(p, 0.0) || dllisgreaterorequal(p, 1.0))
          return dnan;
     g = gammaln(v / 2.0);
     return ppchi2_(&p, &v, &g, &ifault);
}

//Function to substract just a small part of the long sequence
//of random numbers we generate
void  take_one_frame(int size_destination,int position,
double *source,double *destination)
{
    int i;
for (i=1;i<=size_destination;i++)
destination[i]=source[position+(i-1)];
}

void quickSort(double *numbers, int array_size)
{
  q_sort(numbers, 0, array_size - 1);
}

void q_sort(double *numbers, int left, int right)
{
  double pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
```

```
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}

//the main simulation function
void generator(double **matrice, int d, int n1, double niu,double *niu_vector,
double *miu,double *sigma,double *simulation_vector,long seed)
{
FILE *fisier;
double *big_z;
big_z= (double *)malloc(((d+1)*n1+1) * sizeof(double));
    double *z;
    z= (double *)malloc((d+2) * sizeof(double));
    int i,j,k;
double s;
double *x,*y;
    x= (double *)malloc((d+1) * sizeof(double));
    y= (double *)malloc((d+1) * sizeof(double));
double *u;
```

```
u= (double *)malloc((d+1) * sizeof(double));
double *sim_sum_vector;
sim_sum_vector=(double *)malloc((n1+1) * sizeof(double));
for (i=1;i<=n1;i++)
sim_sum_vector[i]=0;
//generating the numbers
random_generator((d+1)*n1,big_z,seed);
    if ( (fisier = fopen("output_file_quantiles.txt","a")) == NULL)
{
printf("Can't open %s ","output_file_quantiles.txt");
        exit(1);
}

    for (k=1;k<=n1;k++)
{
    take_one_frame(d+1,(k-1)*(d+1)+1,big_z,z);
     for (i=1;i<=d;i++)
    z[i]=z_transform(z[i]);
   s=s_transform(z[d+1],niu);
 vector_matrix_one_way_multiplication(d,matrice,z,y);
        for(i=1;i<=d;i++)
     x[i]=(sqrt(niu)/sqrt(s))*y[i];
     for(i=1;i<=d;i++)
     u[i]=t(x[i],niu);
    for (j=1;j<=d;j++)
{
  simulation_vector[j]=(((t_inverse(u[j], niu_vector[j])*
  sqrt(sigma[j]))+miu[j])*sqrt(sigma1[pos][j]))+miu1[pos][j];
sim_sum_vector[k]+=simulation_vector[j];
}
//fprintf(fisier,"%f ",sim_sum_vector[k]);
    //fprintf(fisier,"\n");
}
quickSort(sim_sum_vector,n1);
/*
for (k=1;k<=n1;k++)
{
     printf("%f\n", sim_sum_vector[k]);
}
*/fprintf(fisier,"%f %f %f\n",sim_sum_vector[(int)(0.01*n1)],
sim_sum_vector[(int)(0.05*n1)],sim_sum_vector[(int)(0.1*n1)]);
```

```
sim_sum_vector[10],sim_sum_vector[100],sim_sum_vector[1000],
sim_sum_vector[5000],sim_sum_vector[10000]);
fclose(fisier);
free(big_z);
free(sim_sum_vector);
free(z);
free(u);
free(x);
free(y);
 }


void de_Garch(int dim1,int monstra,int  number_rows)
{
    double **matrice1;
float number;
    FILE *fisier;
//de-GARCHing PART
//alloc space for the matrice (data matrix)
matrice1 = (double **)malloc((number_rows+1)* sizeof(double *));
    if(matrice1 == NULL)
{
         printf("out of memory\n");
}
for(int i = 1; i <= (number_rows+1); i++)
{
        matrice1[i] = (double *)malloc((dim1+1) * sizeof(double));
        if(matrice1[i] == NULL)
{
           printf("out of memory\n");
}
}
if ( (fisier = fopen("C:\\data_for_project.txt","r")) == NULL)
{
printf("Can't open %s ","C:\\data_for_project.txt");
        exit(1);
}
    //data matrix being read
    while (!feof(fisier))
{
```

```
        for (int i=1;i<=number_rows;i++)
  for (int j=1;j<=dim1;j++)
{
        fscanf(fisier,"%f", &number);
      matrice1[i][j]=number
  }

}
fclose(fisier);
printf("am terminat citire\n");
//size of initial sample for de-garching
double sum;
    double *miu_monstra,*sigma_monstra;
miu_monstra= (double *)malloc((dim1+1)*sizeof(double));
    sigma_monstra= (double *)malloc((dim1+1)*sizeof(double));
for (int j=1;j<=dim1;j++)
{
        sum=0;
for (int p=1;p<=monstra;p++)
{
  sum=sum+matrice1[p][j];
}
      miu_monstra[j]=sum/monstra;
}
for (j=1;j<=dim1;j++)
for (int p=1;p<=monstra;p++)
        miu1[p][j]=miu_monstra[j];
    for ( j=1;j<=dim1;j++)
{
for (int p=monstra+1;p<=number_rows;p++)
{
         sum=0;
     for (int k=1;k<=p;k++)
 {

      sum=sum+matrice1[k][j];
 }
       miu1[p][j]=sum/p;
}
}
printf("am terminat de calculat miu\n");
```

```
//write the miu file
if ( (fisier = fopen("output_miu.txt","a")) == NULL)
 {
printf("Can't open %s ","output_miu.txt");
        exit(1);
 }
     for (i=1;i<=number_rows;i++)
 {
  for (j=1;j<=dim1;j++)
            fprintf(fisier,"%15.12f ", miu1[i][j]);
        fprintf(fisier,"\n");
 }
fclose(fisier);

//compute sigma
double lambda=0.94;
for (j=1;j<=dim1;j++)
{
        sum=0;
for (int p=1;p<=monstra-1;p++)
{
    sum=sum+(exp(-lambda*(monstra-p)))*
    ((matrice1[p][j]-miu1[p][j])*(matrice1[p][j]-miu1[p][j]));
}
      sigma_monstra[j]=(exp(lambda)-1)*sum;

}
for (j=1;j<=dim1;j++)
for (int p=1;p<=monstra;p++)
      sigma1[p][j]=sigma_monstra[j];
    for ( j=1;j<=dim1;j++)
{
for (int p=monstra+1;p<=number_rows;p++)
{
    sum=0;
  for (int k=1;k<=p-1;k++)
{
sum=sum+exp(-lambda*(p-k))*
(matrice1[k][j]-miu1[k][j])*(matrice1[k][j]-miu1[k][j]);
}
```

```
    sigma1[p][j]=(exp(lambda)-1)*sum;
}
}
printf("am terminat calculat sigma\n");
//write the sigma file
if ( (fisier = fopen("output_sigma.txt","a")) == NULL)
 {
printf("Can't open %s ","output_sigma.txt");
        exit(1);
 }
     for (i=1;i<=number_rows;i++)
 {
  for (j=1;j<=dim1;j++)
    fprintf(fisier,"%15.12f ", sigma1[i][j]);
    fprintf(fisier,"\n");
 }
fclose(fisier);

//write the new file
if ( (fisier = fopen("output_garch.txt","a")) == NULL)
 {
printf("Can't open %s ","output_garch.txt");
        exit(1);
 }
     for (i=1;i<=number_rows;i++)
 {
  for (j=1;j<=dim1;j++)
fprintf(fisier,"%f ", (matrice1[i][j]-miu1[i][j])/
sqrt(sigma1[i][j]));
fprintf(fisier,"\n");
 }
fclose(fisier);
if(!matrice1)
free(matrice1);
free(miu_monstra);
free(sigma_monstra);
}

void main(void)
{
int dim1=5;      //number of columns
```

```
int dim2=200;
int n1=100000;
long seed;
    float number;
    int monstra=100;
    int number_rows=0;
    FILE *fisier;
if ( (fisier = fopen("C:\\data_for_project.txt","r")) == NULL)
{
printf("Can't open %s ","C:\\data_for_project.txt");
        exit(1);
}
    int i=0;
    while (!feof(fisier))
{
        i++;
for (int j=1;j<=dim1;j++)
{
       fscanf(fisier,"%f", &number);
  }
}
number_rows=i;
fclose(fisier);

//alloc space for the miu1 (vector of avgs for de-garching)
miu1 = (double **)malloc((number_rows+1)* sizeof(double *));
if(miu1 == NULL)
 {
   printf("out of memory\n");
 }

for( i = 1; i <= (number_rows+1); i++)
 {
  miu1[i] = (double *)malloc((dim1+1) * sizeof(double));
  if(miu1[i] == NULL)
 {
printf("out of memory\n");
}
}
//alloc space for the sigma1
sigma1 = (double **)malloc((number_rows+1)* sizeof(double *));
```

```
if(sigma1 == NULL)
 {
  printf("out of memory\n");
 }

for( i = 1; i <= (number_rows+1); i++)
 {
  sigma1[i] = (double *)malloc((dim1+1) * sizeof(double));
  if(sigma1[i] == NULL)
 {
    printf("out of memory\n");
 }
 }
de_Garch(dim1,monstra,number_rows);

//reading the miu1 sigma1 from the files
if ( (fisier = fopen("C:\\output_miu.txt","r")) == NULL)
 {
printf("Can't open %s ","C:\\output_garch.txt");
  exit(1);
 }
for ( i=1;i<=number_rows;i++)
 for (int j=1;j<=dim1;j++)
  {
   fscanf(fisier,"%f", &number);
 miu1[i][j]=number;
  }
fclose(fisier);

if ((fisier = fopen("C:\\output_sigma.txt","r")) == NULL)
 {
printf("Can't open %s ","C:\\output_garch.txt");
  exit(1);
 }
for ( i=1;i<=number_rows;i++)
 for (int j=1;j<=dim1;j++)
  {
   fscanf(fisier,"%f", &number);
   sigma1[i][j]=number;
  }
```

```
fclose(fisier);
double **matrice,**matrice_modificata;
printf("terminat degauch");

//alloc space for the matrice (data matrix)
matrice_modificata = (double **)malloc((number_rows+1)* sizeof(double *));
if(matrice == NULL)
 {
  printf("out of memory\n");
 }

for( i = 1; i <= (number_rows+1); i++)
 {
  matrice_modificata[i] = (double *)malloc((dim1+1) * sizeof(double));
  if(matrice_modificata[i] == NULL)
 {
printf("out of memory\n");
}
}

if ( (fisier = fopen("C:\\output_garch.txt","r")) == NULL)
{
 printf("Can't open %s ","C:\\output_garch.txt");
   exit(1);
}
for ( i=1;i<=number_rows;i++)
 for (int j=1;j<=dim1;j++)
  {
   fscanf(fisier,"%f", &number);
   matrice_modificata[i][j]=number;
  }

printf("am citit noua matrice\n");
fclose(fisier);
pos=dim2;
printf("%d, intram in while \n",number_rows);
while (pos<=number_rows)
{
  printf("Begining of while\n");

//alloc space for the matrice (data matrix)
```

```
matrice = (double **)malloc((dim2+1)* sizeof(double *));
 if(matrice == NULL)
{
   printf("out of memory\n");
}

 for(int i = 1; i <= dim2; i++)
{
   matrice[i] = (double *)malloc((dim1+1) * sizeof(double));
   if(matrice[i] == NULL)
   {
    printf("out of memory\n");
   }
}
 for ( i=pos-dim2+1;i<=pos;i++)
 for (int j=1;j<=dim1;j++)
{
   matrice[i-pos+dim2][j]=matrice_modificata[i][j];
  }

Computing(matrice,dim2,dim1,n1,seed);
pos++;
free(matrice);
//printf("Tried to free matrice\n");
//printing for checking purposes
}
}

$
```