An Exploratory Analysis for Studying Software Evolution: Time-Delayed Correlation Analysis

Yosuke Yamatani
Graduate School of Systems Engineering,
Wakayama University
Email: s151049@sys.wakayama-u.ac.jp

Masao Ohira
Faculty of Systems Engineering,
Wakayama University
Email: masao@sys.wakayama-u.ac.jp

***Abstract*—We propose an exploratory analysis method for software repositories so that you can study software evolution without an explicit goal. The method extends the traditional correlation analysis to consider a time lag between any pair of metrics. Using data collected from three kinds of repositories, the method automatically calculates optimal parameters to deal with a time lag between metrics and returns the highest correlation coefficient for each pair of metrics. Our case study using one hundred metrics data from three OSS projects shows that the method allows us to find unexpected relationships in some pairs of metrics if we only use the traditional correlation analysis and that using such relationships can be used to build an analysis goal to understand software evolution.**

Keywords*—*software evolution; delayed correlation analysis; empirical software engineering

I. Introduction

Software evolution [1], [2] refers to changes to software products and process over time in order to adapt users' various needs to circumstances of the moment. Since software systems are essential for our daily life, they must be maintained for a long time in adapting to environmental changes, adding necessary functions at the time and keeping the quality of the software systems [3]. It is also necessary to understand the impact of software evolution on software quality and user satisfaction. In the modern, large-scale software development, it is hard to correctly understand the impact of software evolution on software quality and user satisfaction.

Many studies have addressed the issues of understanding software evolution. For instance, Zimmermann et al. developed a tool to recommend files which are likely to be modified at the same moment when developers modify files [4]. The tool uses association-rules mined from a version control system. Understanding software evolution can be a valuable knowledge for project managers and developers in developing and maintaining software systems.

The field of software evolution have provided a significant advantage. However tools or methods are not still enough for practitioners to analyze software evolution. For example, in order to understand how software modifications influences software quality, we need investigate a relationship between the modifications and software quality by measuring the number of bugs or bug density in step after the delay from the programing development process. The traditional correlation analysis is not occasionally suitable to analyze long-lived software systems because it cannot extract a timely-delayed correlation between metrics. It is hard for practitioners who lack a full understanding of past software development context to determine the candidate for analyze and the metrics. Under that circumstance, it is difficult to apply a top-down approach like GQM [5]. Practitioners need exploratory data analysis [6] by using a bottom-up approach such as a boxplot or a scatter plot. However, we cannot analyze software evolution sufficiently with bottom-up approaches because in general they do not consider the existence of time-delayed relationships among metrics.

In this paper, we propose an exploratory analysis method for software repositories so that you can study software evolution without an explicit goal. The method extends the traditional correlation analysis to consider a time lag between any pair of metrics (e.g., additional developers to a project have an impact on productivity several weeks or months later). Using data collected from three kinds of repositories (version control, issue tracking, and mailing list archive), the method automatically calculates optimal parameters to deal with a time lag between metrics and returns the highest correlation coefficient for each pair of metrics. Our case study using one hundred metrics data from three OSS projects (Eclipse Platform, Apache HTTP Server, and GIMP) shows that the method allows us to find unexpected relationships in some pairs of metrics if we only use the traditional correlation analysis and that an analysis goal for understanding software evolution can be derived from such the relationships.

The paper is organized as follows. Section II discusses the motivation of this study. Section III introduces our proposal method, delay correlation analysis. Section IV describes our case study in 3 projects (Eclipse Platform, Apache HTTP Server, and GIMP). Section V shows the result of our case study, section VI discusses implications obtained from the case study. Section VII mentions conclusion and future work.

II. Motivation

Our study is motivated because of the following two issues in studying software evolution through mining time-series data extracted from software repositories.

A. Exploratory data mining for software repositories

Recent advances in the field of Mining Software Repositories (MSR) have provided a significant advantage to researchers and practitioners who dedicate to study and understand software evolution. A variety of MSR techniques offer a means to automatically analyze large-scale software engineering data, predict software quality, localize faults (e.g., defect prediction) in a file, visualize software evolution and so forth.

However most of the techniques are based on a top-down approach that forces you to have a clear goal in advance for analysis and/or prediction. If we would like to analyze software evolution from more diverse perspectives, the current top-down methodology in MSR occasionally is not applicable. In fact, a general procedure of mining software engineering data often starts with an exploratory process to determine what software engineering (SE) data can be used to support a SE task [7]. In order to help the exploratory process efficiently, not only using traditional tools such as a histogram and a box plot for each metric but also using Brute-force-like, bottom up approach would be useful to find interesting relationships between metrics measured from software repositories.

B. *Time-series data analysis in software engineering*

Analyzing two types of time-series data is essential to understand how software evolves over time. For instance, measuring the growth of lines of code and the number of implemented functions over time in a project and analyzing the relation between the two metrics would be helpful for a project manager to understand the productivity of the project. In case of analyzing such a relation between two types of time-series data, cross-correlation [8] is used as a measure to know the similarity between the two types of time-series data especially in the domain of signal processing. Cross-correlation is also useful to analyze a lagged relation between two time-series data.

Cross-correlation is supposed to be used for two time-series data obeying stationary stochastic processes. Other models for time-series data such as autoregressive (AR) [9] and autoregressive moving average (ARMA) [9] is also based on stochastic processes. In the domain of software engineering, however, many types of time-series data cannot assume stochastic processes excepting some examples (e.g., reliability growth model), since they are strongly affected by various events in software development such as delivery date and unexpected change requests from customers. Due to this reason, we often cannot apply existing analysis methods for time-series data to metrics measured in the domain of software development.

III. Time-Delayed Correlation Analysis

In this paper we propose an analysis method for software evolution using time-series data (metrics) extracted from software repositories. The method consists of three steps: (1) preprocessing for outliers, missing values, data normalization, (2) delayed correlation analysis for calculating Spearman's rank-correlation coefficient between an explanatory variable and a time-delayed objective variable, and (3) test for no correlation.

A. *Preprocessing*

Software repositories for software configuration management and issue tracking include a wealth of time-series data to analyze software evolution. However, it is well-known that data observed in software development sometimes has outliers which have great impacts on analysis results. In oder to remove outliers in time-series data, detecting outliers based on distance in the two diminutional space is required [**?**]. In this study, Mahalanobis' generalized distance [10] is used to detect and remove outliers.

After removing outliers, missing values are created in time-series data. Since it is not appropriate to analyze correlation for time-series data with missing values, it is necessary to complement data for the missing values. In this study, spline interpolation [11] is used.

In addition to the preprocessing for outliers and missing values, all metrics measured as time-series are normalized between 0 and 1 since they include different scales and units. In this study the following equation (1) is used to normalize data.

$$x_{i_{nor}} = \frac{x_i - min(x)}{max(x) - min(x)} \quad (1)$$

In (1), x represents input time-series data (metric). x_i and $x_{i_{nor}}$ means a value of x at time i and a normalized value at time i respectively. $max(x)$ and $min(x)$ are the maximum value and minimum value in time-series data x respectively.

B. *Delayed Correlation*

Delayed Correlation Analysis (DCA) extends correlation analysis to capture a relation between an explanatory variable and a time-delayed objective variable. As shown in Figure 1, an explanatory variable and a time-delayed objective variable can include several data points while sliding the set of data points point by point. A basic assumption of DCA is that the amount of changes in the explanatory variable for certain periods affects the amount of changes in the objective variable for certain periods after certain periods.

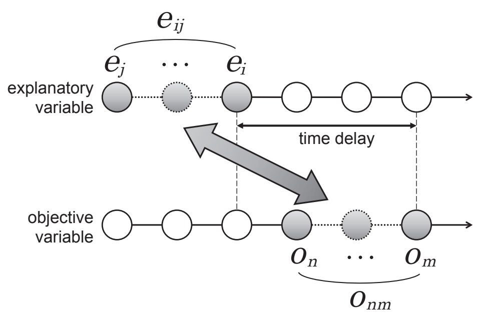

Fig. 1. A concept of delayed correlation analysis

Table I shows parameters used for DCA. In our study, a data point is supposed to be a month. The three kinds of parameters (coefficient values) range from 0 to 12 months. If all coefficient values are 0, it means that DCA works as regular coefficient analysis. In this settings, our DCA method calculates correlation coefficients for all pairs of measured metrics using all combinations of the three kinds of parameters, which means that our method outputs $\frac{N(N-1)}{2} \times 12^3$ kinds of correlation coefficients for N kinds of metrics.

The calculation formulas for the explanatory variable and the objective variable are defined as follows.

$$e_{ij} = e_i + \cdots + e_j \quad (2)$$

TABLE I. PARAMETERS USED IN DCA

parameter	definition
accumulative coefficient for an explanatory variable	$i-j$
accumulative coefficient for an objective variable	$m-n$
time-delay coefficient	$n-i$

In (2), e_i and e_j are values of the explanatory variable at time i and j respectively. Based on the accumulative coefficient for an explanatory variable ($i-j$), data points for the explanatory variable (from 0 to 11) are put together into a single data point. A value of the explanatory variable e_{ij} is calculated as an accumulative value from e_i to e_j at time i. In the same way, a value of the objective variable is calculated as follows.

$$o_{nm} = o_n + \cdots + o_m \tag{3}$$

After all accumulative values of the explanatory and objective variables at each time i and n are calculated, data points in the explanatory and objective variables are overlapped using the time-delay coefficient ($n-i$) in order to enable regular correlation analysis.

C. Test for No Correlation

Because of a large number of the combinations of measured metrics and the three kinds of parameters, DCA will also produce a large number of correlation coefficients between any pair of metrics. In order to reduce analysts' efforts, our method allows analysts to determine a threshold for correlation coefficient (e.g., 0.7) to only look at strong relationships between metrics.

After specifying the threshold, our method tests the filtered correlations using Mann–Whitney U test and removes correlations with no statistical significance ($p < 0.05$).

IV. CASE STUDY

This section describes a case study of applying delayed correlation analysis (DCA) to three OSS projects (Eclipse Platform, Apache HTTP Server, and GIMP). The goal of the case study is to show that DCA can extract unexpected correlations between metrics, which cannot be extracted through regular correlation analysis.

A. Scenario for the Case Study

Our case study assumes that an analyst tries to find interesting relationships among various metrics, in particular, to understand evolution of software quality in OSS development. In order to do so, the analyst first selects a source repository and a bug repository to measure product and process metrics since they could include rich information to know evolutionary relationships between software quality and bug management process in OSS development. Then she mines relationships between any metrics using DCA. Note that regular correlation analysis is included in DCA since DCA works as a special case of regular correlation analysis when the all parameters used in DCA are 0. Figure 2 shows a relation between two sets of extracted relationships between measured metrics, using regular correlation analysis and DCA. The analyst analyze extracted relationships in $\overline{CA} \wedge DCA$ (grayed part in Figure 2) while comparing with relationships in CA.

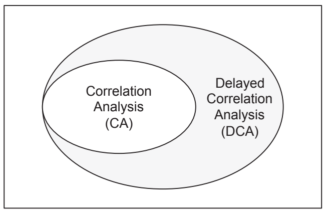

Fig. 2. A relation between extracted relationships (sets of relations between metrics) using regular correlation analysis (CA) and DCA

B. Datasets and Measured Metrics

For our case study, we collected data from version control systems (i.e., SVN and Git) and bug tracking systems (i.e., Bugzilla) which are used in the Eclipse Platform, Apache HTTP Server, and GIMP projects. Table II shows our datasets for our case study. In Table II, *# of target files (lang.)* means the number of files to measure product (source code) metrics and main languages to write source codes. *# of committed changes* is the number of commits in version control systems to measure process metrics. *# of bugs* and *# of comments on bugs* respectively mean the number of reported bugs in Bugzilla and the total number of comments which are added to a bug report for discussing the bug among developers.

In our case study, one hundred metrics are used to measure product and its quality, and bug management process in OSS development. Table III shows metrics used in the case study. Of the metrics, 28 kinds of source code metrics are measured by using a source code analysis tool called Understand[1]. 25 kinds of change metrics from commit logs, 37 kinds of bug information metrics from bug reports, and 10 kinds of communication metrics from comments on bugs are respectively measured by using our own making scripts. Note that we do not include object-oriented metrics for source codes written in Java (i.e., we do not use object-oriented metrics for Eclipse Platform) because programming languages are different among the target projects and we would like to make obtained results comparable.

C. Study Method

As described in the previous section, delayed correlation analysis (DCA) produces $\frac{N(N-1)}{2} \times 12^3$ kinds of correlation coefficients between any pair of measured metrics. Since one hundred metrics (i.e., $N = 100$) are used in our case study, a huge number of correlation coefficients are calculated and showed to an analyst. To prevent the analyst from consuming a lot of time to check all correlation coefficients, in the case study we set 0.7 as a threshold so that DCA can provides the analyst relationships only with correlation coefficients over 0.7.

V. RESULTS

A. Extracted Correlations: Statistical Result

Table IV shows the number of correlations using DCA and CA. As described in table IV, DCA extracts 7,749 correlations in Platform project, 4,885 correlations in Apache project and

[1]Understand Source Code Analysis & Metrics, http://ww.scitools.com/

TABLE II. DATASETS USED IN THE CASE STUDY

Project	target periods	# of target files (lang.)	# of committed changes	# of bugs	# of comments on bugs
Platform	2003/07–2012/06	940 (Java)	5,240	74,202	401,609
Apache	2002/06–2011/05	258 (C)	12,556	6,174	23,066
Gimp	2001/01–2011/12	1,596 (C)	25,515	10,609	54,245

TABLE III. MEASURED METRICS FOR OUR DATASETS

Data Sources	Metrics
source code	# of (All Lines, Blank Lines, Lines Containing Source Code, Lines Containing Comment), Average # of (All Lines, Blank Lines, Lines Containing Source Code, Lines Containing Comment) for All Nested Functions or Methods, # of Functions, # of Semicolons, # of Lines Containing (Declarative, Executable) Source Code, # of Statements, # of (Declarative, Executable) Statements, Average (Cyclomatic, Modified Cyclomatic, Strict Cyclomatic, Essential) Complexity for All Nested Functions or Methods, Maximum (Cyclomatic, Modified Cyclomatic, Strict Cyclomatic, Essential) Complexity for All Nested Functions or Methods, Sum of (Cyclomatic, Modified Cyclomatic, Strict Cyclomatic, Essential) Complexity for All Nested Functions or Methods, Ratio of Comment Lines to Code Lines
commit log	# of Committers, # of New Committers, Ratio of Commits by Code Owner to All Commits, # of Commits, # of Lines Added, # of Lines Deleted, # of Commits that Added Lines, # of Commits that Deleted Lines, # of Files Changed, Average # of Comment Words of Commit, (# of Commits, # of Lines Added, # of Lines Deleted, # of Commits that Added Lines, # of Commits that Deleted Lines, # of Files Changed, Average # of Comment Words of Commit) by New Committer, (# of Commits, # of Lines Added, # of Lines Deleted, # of Commits that Added Lines, # of Commits that Deleted Lines, # of Files Changed, Average # of Comment Words of Commit) by Code Owner
bug report	# of (Reported, Assigned, Fixed, Resolved, Verified) Bugs, # of Developers have (Reported, Assigned, Fixed) Bugs, # of Bugs that Priority are (P1, P2, P3, P4, P5), # of Bugs that Severity are (Blocker, Critical, Major, Normal, Minor, Trivial, Enhancement), # of Bugs that Resolution are (WontFix, Invalid), Average # of (CC List, Description Words), # of (Reopened, Reassigned) Bugs, # of (Reopened, Reassigned) Times, Ratio of (Reopened, Reassigned) Bugs to All Bugs, Median Time of Assigning Bugs, Median Time of Fixing Bugs, Median Time to be Fixed Since been Reported
comments on bugs	# of Comments, # of Commentators, # of New Commentators, Average # of Comments per Bug, Average # of Comment Words, # of Domain, Ratio of Maximum # of Commentator to All Commentators, Ratio of Maximum # of Domain to All Domain, Ratio of New Commentators to All Commentators, Median Time of Replying

TABLE V. A PART OF RESULT OF DELAYED CORRELATION ANALYSIS

Explanatory Variable	Objective Variable	correlation coefficients		
		Platform	Apache	Gimp
# of reported bugs	# of verified bugs	0.97	0.70	0.82
# of reassigned times	average # of blank lines for all nested functions or methods	-0.75	-0.87	0.72
# of bugs that severity are blocker	# of commentators	0.99	0.92	0.87
average # of blank lines for all nested functions or methods	# of comment words of commit by code owner	0.74	0.74	0.73
# of committers	# of bugs that priority are P3 (Medium)	0.89	0.7	0.92
# of bugs that resolution are WontFix	# of new commentators	0.92	0.82	0.87
# of statements	# of bugs that severity are critical	-0.85	-0.95	-0.91

TABLE IV. THE NUMBER OF EXTRACTED CORRELATIONS USING DCA

	DCA	CA	$\overline{CA} \wedge DCA$
Eclipse	7,749	1,012	6,737
Apache	4,885	434	4,451
Gimp	5,679	660	5,019
Common	783	28	127

C: correlation analysis, DC: delayed correlation analysis

TABLE VI. EACH PARAMETER OF CORRELATION BETWEEN RB AND VB WITH DCA

project parameter	Platform	Apache	Gimp
accumulative coefficient for an explanatory variable	6	11	7
accumulative coefficient for an objective variable	8	11	11
time-delay coefficient	4	12	5
correlation coefficient with DCA	0.97	0.7	0.82
correlation coefficient with CA	0.62	0.22	0.21

5,679 correlations in Gimp project. DCA also extracts 783 common correlations among 3 projects.

In this paper, we focus on common correlations among 3 projects out of correlations that CA doesn't extract ($\overline{CA} \wedge DCA$). The # of common correlations with $\overline{CA} \wedge DCA$ is 127. Table V shows a part of results of $\overline{CA} \wedge DCA$ among 3 projects. For example, the top of correlation in tableV means increased the # of reported bugs means increased # of verified bugs. Table VI shows each parameter and correlation coefficient with DCA and CA of correlation between # of reported bugs (RB) and # of verified bugs (VB). As described in table VI, increased RB means increased VB 4 months after in Platform project, 12 months after in Apache project and 5 months after in Gimp project.

TABLE VII. EACH PARAMETER OF CORRELATION BETWEEN RT AND ALB WITH DCA

project parameter	Platform	Apache	Gimp
accumulative coefficient for an explanatory variable	8	11	11
accumulative coefficient for an objective variable	10	11	11
time-delay coefficient	1	6	6
correlation coefficient with DCA	-0.75	-0.87	-0.7
correlation coefficient with CA	-0.21	-0.07	-0.2

The second top correlation in table V means increased the # of reassigned times means decreased average # of blank lines for all nested functions of methods. Table 3 shows each parameter and correlation coefficient with DCA and CA of correlation between the # of reassigned times (RT) means decreased average # of blank lines for all nested functions of methods (ALB). As described in table 3, increased RT means

TABLE VIII. THE TIME FROM REPORTING BUGS TO VERIFYING BUGS

project	median (day)	average (day)
Platform	29.60	111.37
Apache	77.41	381.68
Gimp	2732.58	2515.62

decreased ALB 1 months after in Platform project, 6 months after in Apache project and 6 months after in Gimp project. This correlation coefficients -0.07 with CA, and -0.87 with DCA in Apache project. As just described, DCA can extract correlations that CA doesn't extract. Next we mention why the correlations with time-delayed between RB and VB and between RT and ALB are extracted.

B. Extracted Correlations: Result of detailed analysis

1) Correlation with time-delayed between RB and VB: We mention why the correlation with time-delayed between RB and VB is extracted. In OSS development, when developers and users find bugs, they report bugs to bug tracking system (BTS) like Bugzilla. Developers check a symptom of the reported bug, developers create patches to fix the bug. After that, the other developers verify the fixed bug. In this way, reported bugs are closed after fixed and verified. Table VIII shows the median and average time from reporting bugs to verifying bugs in 3 projects. As described in table VIII, the average time from reporting bugs to verifying bugs is 111.37 days (about 4 months) in Platform project and 381.68 (about 12 months) days in Apache project. We compare the average time from reporting bugs to verifying bugs and time-delay coefficient, it can be seen that DCA extracts the correlation at the correct intervals.

2) Correlation with time-delayed between RT and ALB: We describe as an example the Apache project of why the correlation with time-delayed between RT and ALB is extracted. Figure 3 shows change of RT and ALB over time. As shown in Figure 3, it can be seen that ALB shows a marked drop around December 2008. The reason is that Apache distribute packages without PCRE (Perl Compatible Regular Expressions) library from December 2008. PCRE library is a set of functions that implement regular expression pattern matching. It was revealed that PCRE library conflicted the other library installed in Apache, therefore PCRE library was deleted from Apache packages. PCRE library had $prec.c$ file which contained 1,267 blank lines (ALB is 88.59 in December 2008). To delete Files which contain a lot of blank lines like $prec.c$ decreases ALB. DCA extract the correlation with time-delayed between RT and ALB because 6 months ago that ALB was decreased, RT was high.

VI. Discussion

It is necessary for researcher and project manager who just taken over a project to understand software evolution. However, they don't know a context of software development project. It is also hard for them to understand events that happened in the past and the impact of product and software development process by past events.

In this paper, we implement a case study with 100 metrics calculated from 3 projects over 9 years. As described in previous section, it is revealed unexpected events such as deleting PCRE library by DCA and detailed analysis. It is useful for understanding software evolution to detect unexpected events and to understand the impact of product and software development process by them. A proposal method can assist for researcher and project manager who just taken over a project to understand software evolution. We expect to use a proposal method by researchers who analyze a long-term software development project and manager of a project which is taken over manager with regularity.

VII. Conclusion and Future Work

In this paper, we propose an exploratory analysis method for software repositories so that you can study software evolution without an explicit goal. The method extends the traditional correlation analysis to consider a time lag between any pair of metrics (e.g., additional developers to a project have an impact on productivity several weeks or months later). Using data collected from three kinds of repositories (version control, issue tracking, and mailing list archive), the method automatically calculates optimal parameters to deal with a time lag between metrics and returns the highest correlation coefficient for each pair of metrics. Our case study using one hundred metrics data from three OSS projects (Eclipse Platform, Apache HTTP Server, and GIMP) shows that the method allows us to find unexpected relationships (eg. between # of reassigned times and average # of bank lines for all nested functions or methods) in some pairs of metrics if we only use the traditional correlation analysis and that using such relationships can be used to build an analysis goal to understand software evolution. In the future, we'll conduct a detailed analysis to reveal relationships which don't be mentioned in this paper.

Acknowledgment

This research is conducted as part of Grant-in-Aid for Scientific Research (C) 24500041 by Japan Society for the Promotion of Science (JSPS).

References

[1] T. Mens and S. Demeyer, *Software Evolution*. Springer, 2008.
[2] J. Fernandez-Ramil and D. Perry, *Software Evolution and Feedback: Theory and Practice*, N. H. Madhavji, Ed. Wiley, 2006.
[3] M. M. Lehman, "Software engineering, the software process and their support," *Software Engineering Journal*, vol. 6, pp. 243–258, 1991.

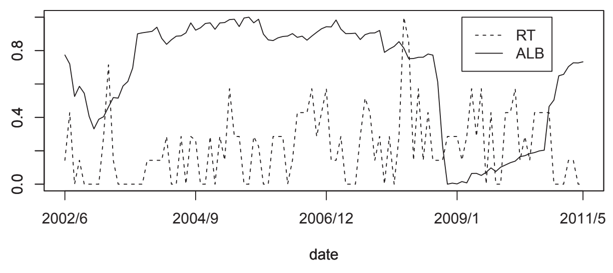

Fig. 3. Change of RT and ALB over time

[4] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 429–445, 2005.

[5] V. R. Basili, “Using measurement to build core competencies in software,” in *Seminar sponsored by Data and Analysis Center for Software*, 2005.

[6] J. W. Tukey, *Exploratory Data Analysis*, 1977.

[7] T. Xie, S. Thummalapenta, D. Lo, and C. Liu., “Data mining for software engineering,” *IEEE Computer*, vol. 42, no. 8, pp. 35–42, August 2009. [Online]. Available: http://www.cs.illinois.edu/homes/taoxie/publications/ieeecomputer09-dmse.pdf

[8] R. H. Shumway and D. S. Stoffer, *Time Series Analysis and Its Applications With R Examples*, 3rd ed. Springer, 2010.

[9] G. Box and G. Jenkins, *Time series analysis: Forecasting and control*, 1970, san Francisco, Holden-Day.

[10] P. C. Mahalanobis, “On the generalised distance in statistics,” in *Proceedings National Institute of Science Calcutta*, vol. 2, no. 1, 1936, pp. 49–55.

[11] I. J. Schoenberg, “Contributions to the problem of approximation of equidistant data by analytic functions,” in *Quarterly of Applied Mathematics*, vol. 4, 1946, pp. 45–88.