CRANFIELD UNIVERSITY

SCHOOL OF MECHANICAL ENGINEERING

Ph.D. THESIS

Academic Year 1995

P. C. ESCHER

Pythia: An Object-Oriented Gas Path Analysis Computer Program for General Applications

Supervisor: R. Singh

October 1995

Summary

For both civil and military aero gas turbines, technological advances and high cost of ownership have resulted in considerable interest in advanced maintenance techniques. Some of these techniques are now ready for application to industrial gas turbines. This thesis attempts to give an overview of engine maintenance and engine health monitoring techniques.

One way to tackle the high cost is to employ Gas Path Analysis techniques. Gas Path Analysis helps to identify deteriorated components of a gas turbine in terms of performance parameter changes with respect to each other. The changes can be analysed and actions taken to minimise the life cycle costs of a gas turbine.

A generalised Gas Path Analysis computer program Pythia has been developed that incorporated new techniques such as a non-linear multiple fault diagnostics scheme. In order to develop reliable software a structured methodology, conforming to quality standards, has been introduced. The program Pythia is based on an object-oriented programming method that can be run with a modern PC.

Pythia has been applied to a wide range of gas turbine engines. As a result, the Gas Path Analysis technique showed statistically significant improvements with the non-linear solution. The non-linear GPA technique was able to successfully identify sets of instrumentation for all engines.

Finally, the thesis presents further developments of the non-linear GPA technique. In particular, instrumentation error, creep life estimation and low cycle fatigue estimation are some of the techniques.

Acknowledgements

"...Ιξεις αφιξεις ου εν πολεμω θνηξεις..."
Pythia - Greek Goddess of the Oracle of Delphi

The preparation of the present investigation has been long but exciting and one which would not have been possible without the help and encouragement of my family, friends, colleagues and many professional acquaintances. In particular I would like to express my gratitude to my supervisor Professor Riti Singh, who always provided me with the appropriate motivation and guidance at the right time.

I also would like to thank those who have provided me with research grants: ABB Turbo Systems Ltd., Pfauen-Escher Foundation and the Swiss government's SATW grant.

The contribution of the following in validating and checking the program Pythia is gratefully acknowledged: Gauthier DuBois, Lee English, Nik Mustapha, Spiros Pendedekas and in particular Don Waldock for his many suggestions to improve the program Pythia.

My warmest thanks also go to the four reviewers of the thesis whose comments I found very helpful: Geoff Engelbrecht, Ying Hang Lee, Cecil Stewart and Don Waldock. My partner Olga Pardo read and criticised the thesis in all its stages.

To all of them I would like to offer my very best thanks.

List of Contents

LIST OF FIGURES VI

LIST OF TABLES IX

NOTATION X

GLOSSARY OF TERMS XIII

PART I BACKGROUND AND OVERVIEW 1

1 **Introduction** 2

2 **Engine Maintenance and Engine Health Monitoring** 7

2.1 Maintenance Methods 7
2.2 The Development of Engine Health Monitoring Systems 13
2.3 Engine Health Monitoring Techniques 19

PART II METHODS 29

3 **Review of Traditional Gas Path Analysis Techniques** 30

3.1 Performance Analysis 31
3.2 Performance Deterioration 32
3.3 Performance Trend Monitoring and Analysis 43

4 **Gas Path Analysis Principles** 53

4.1 Linear Gas Path Analysis 53
4.2 Non-Linear Gas Path Analysis 56

5 **The Development of GPA Diagnostics Program Pythia** 65

5.1 Software Quality Assurance 65
5.2 Structured System Analysis and Design Methodology 66
5.3 Programming 87

PART III CASE STUDIES 98

6 **Pythia Applied to Industrial and Aero Gas Turbine Engines** 99

6.1 Single and twin Spool Gas Generator With Free Power Turbine 99
6.2 Solid Shaft Single Spool Engine 130
6.3 Single Shaft Turboprop Engine 132

PART IV DISCUSSION & CONCLUSION 138

7 **Discussion** 139

- Potential Use of GPA Diagnostics
- Limitations
- Further Applications

8 **Conclusion** 148

REFERENCES 151

APPENDICES

A **Component Map Scaling** A1

B **Pythia Class Library Version 1.1** B1

C **Turbomatch Simulation** C1

D **Pythia Application** D1

List of Figures

1.1	Overview Engine Health Monitoring Systems	2
1.2	Availability vs Down-Time	4
2.1	Types of Failures (Saravanamuttoo, 1983)	10
2.2	Plant Availability (Engine Availability = 90 %)	11
2.3	Engine Monitoring Systems General Description (Hess, 1983)	17
2.4	Flexible Endoscope System Capability (Lynch, 1989)	21
2.5	Borescope Access Points (Lynch, 1989)	22
2.6	Efficiency of Debris analysis versus Wear Debris Particle Size (Lynch, 1989)	25
3.1	Performance Analysis	32
3.2	Effect of Compressor Erosion on the Compressor Characteristic (Tabakoff, 1990)	37
3.3	Compressor Fouling	41
3.4	Turbine Erosion	42
3.5	Data Non-Repeatability Effects	48
3.6	Typical Performance Deterioration of a Compressor	49
3.7	Example of a Fault Tree	50
3.8	Example of Engine Fault Matrix for a Single Spool Gas Generator with a Free Power Turbine (Saravanamuttoo, 1974)	51
4.1	Performance Degradation	53
4.2	Performance Diagnostics	54
4.3	Influence Coefficient Matrix of a Single Spool Free Turbine Engine (Urban, 1983)	55
4.4	Simplified Illustration of Non-Linear Gas Path Analysis Method	59
5.1	Stages in System Analysis and Design Projects	68
5.2	Basic Symbols Used in Data Flow Diagrams	71
5.3	Data Flow Diagramming Process (SIMS, 1992)	72
5.4	Current DFD of Program Detem (Level 1)	75
5.5	Required DFD of Program Pythia (Level 1)	80
5.6	Required DFD of Program Pythia Level 2 (Activity 2: Engine Model Design Mode)	82
5.7	Required DFD of Program Pythia Level 2 (Activity 3: Clean Performance Mode)	83

5.8	Required DFD of Program Pythia Level 2 (Activity 4: GPA Mode)	84
5.9	Required DFD of Program Pythia Level 3 (Activity 4.2: GPA Modelling)	86
5.10	Document and View Relationship (Microsoft, 1993)	91
5.11	The Programmers Code in the Application Framework (Microsoft, 1993)	92
5.12	Dialog and Collection Class Hierarchy	94
5.13	Module Class Hierarchy	95
5.14	Views, Frame Windows, Control Bars, Maps, Document Architecture and CCmdTarget Class Hierarchies	96
6.1	Schematic Layout of STB5000S	104
6.2	Diagnostics Approaches in the Case A (STB5000S)	106
6.3	Diagnostics Approaches in the Case B (STB5000S)	108
6.4	Diagnostics Approaches in the Case C (STB5000S)	108
6.5	Diagnostics Approaches in the Case D (STB5000S)	109
6.6	Diagnostics Approaches in the Cases E and F (STB5000S)	112
6.7	Diagnostics Approaches in the Cases G and H (STB5000S)	113
6.8	Diagnostics Approaches in the Cases I to L (STB5000S)	115
6.9	Schematic Layout of SRB211S	117
6.10	Diagnostics Approaches in the Case A Part I (SRB211S)	120
6.10	Diagnostics Approaches in the Case A Part II (SRB211S)	121
6.11	Diagnostics Approaches in the Case B Part I (SRB211S)	122
6.11	Diagnostics Approaches in the Case B Part II (SRB211S)	123
6.12	Diagnostics Approaches in the Case C (SRB211S)	124
6.13	Diagnostics Approaches in the Case D (SRB211S)	125
6.14	Diagnostics Approaches in the Case E (SRB211S)	126
6.15	Diagnostics Approaches in the Case F (SRB211S)	127
6.16	Diagnostics Approaches in the Cases G and H (SRB211S)	128
6.17	Diagnostics Approaches in the Cases I to L (SRB211S)	129
6.18	Schematic Layout of SGE MS9001ES	131
6.19	Schematic Layout of the SAllison T56S	133
6.20	Validation of Deteriorated Performance: Air Mass Flow versus TET (Waldock, 1995)	134

6.21	Validation of Deteriorated Performance: Turbine Efficiency versus TET (Waldock, 1995)	134
6.22	Effect of Independent Parameter Changes: Comparison of Linear and Non-Linear GPA (English, 1995)	136
7.1	Return on Investment (ROI) of Engine Monitoring System (EMS)	141
7.2	Effect of Compressor Degradation on 1st Stage Turbine Rotor Blade Creep Life	143
7.3	Low Cycle Fatigue Life. Effect of Degradation of LP Compressor & Variation of Inlet Temperature	144
A1	Compressor Scaling in Turbomatch (MacMillan, 1974)	A4
A2	Turbine Scaling in Turbomatch (MacMillan, 1974)	A5
D1	Start of Pythia Application	D1
D2	Credits of Pythia and Open Project File	D2
D3	Select Operating Mode; Build New Engine by Assembling Components	D3
D4	Prepare Clean Performance by Modifying Component Data	D4
D5	Define Off-design Point Settings and Compare Performance Results (DP-OD)	D5
D6	Implant Deterioration and Compare Baseline to Deteriorated Performance	D6
D7	Define Sets of Monitored Parameter and Conduct Non-Linear GPA	D7

List of Tables

1.1	System Ranking by Forced Outage Rate and by Forced Outage Total Down-Time (Singh, 1993)	3
3.1	Physical Faults Expressed as Independent Parameter Changes	43
3.2	Simple Turbine Diagnostics Matrix	
5.1	Problems and Requirements of the System Detem/Pythia	77
5.2	Scaling Factors in Turbomatch	97
6.1	Case studies that establish the ideal set of instrumentation for each engine	100
6.2	Statistical Analysis of RMS errors (STB5000S)	107
6.3	A Non-Acceptable FCM	110
6.4	An Acceptable FCM	110
6.5	Diagnostics Approaches in the Cases M and N (STB5000S)	116
6.6	Statistical Analysis of RMS errors (SRB211S)	119
6.7	Diagnostics Approaches in the Cases M and N (SRB211S)	130
C1	Compared Input Files for Turbomatch Simulation of SRB211S and STB5000S	C1

Notation

SYMBOL

#	Number of...
A	Case
C	Data store of C++ program
δ	Difference between...
Δ	Normalised...
F	Data store of Fortran program or function of...
Γ	Non-dimensional massflow
η	Efficiency
J	Jacobean matrix
λ	Damping factor
N	Spool speed
P	Pressure
p	Step in iterative process
T	Temperature
x	Independent parameter
y	Dependent parameter

SUBSCRIPT

1,2...	Number of inlet or outlet of component or number of particular diagnostic approach
B	Burner/Combustor
C	Compressor
Clean	Clean performance value
Deter	Deteriorated performance value
i	i^{th} row in matrix
j	j^{th} column in matrix
M	Module or component
meas	Measured value
new	New calculated value for next iteration
obs	Observed or detected value

old	Value of previous iteration
sum	Sum of...
T	Turbine
TM	Turbomatch value

ACRONYMS

AIDS	Aircraft integrated data system
ARP	Aerospace recommended practice
BURSF	Combustion efficiency scaling factor
CASE	Computer aided software engineering
C++	Object-oriented programming language C++
CN	Rotational speed
DFD	Data flow diagram
DH	Enthalpy drop
DHTSF	Turbine enthalpy drop scaling factor
DP	Design point
ECM	Engine condition monitoring
EDMS	Engine distress monitoring system
EHM	Engine health monitoring
EMS	Engine monitoring system
ETACSF	Compressor efficiency scaling factor
ETATSF	Turbine efficiency scaling factor
FCM	Fault coefficient matrix
FOD	Foreign object damage
GPA	Gas path analysis
ICM	Influence coefficient matrix
IDMS	Ingested debris monitoring system
LCC	Life cycle costs
LCF	Low-cycle fatigue
OD	Off-design point
PR	Pressure ratio
PRCSF	Compressor pressure ratio scaling factor
RCM	Reliability centered maintenance
RMS	Root mean square

ROI	Return on investment
SAE	Society of automotive engineers
SCM	Sensor coefficient matrix
SF	Scaling factor
SHP	Shaft horse power
SOAP	Spectrometric oil analysis program
SQA	Software quality assurance
SSADM	Structured system analysis and design methodology
TFTSF	Turbine non-dimensional massflow scaling factor
WACSF	Compressor non-dimensional massflow scaling factor

Glossary of Terms

Availability	There are various definitions used but generally the ratio of the time the unit is available to be used to the total time, or: $Availability = \frac{(hours\ in\ period)\ -\ (all\ downtime\ hours)}{(hours\ in\ period)}$
Base-load	The power production level that is required continuously, thus it is the minimum power level on the load profile. A base-load power plant is required to operate nearly continuously. The definition of a base-load power plant is formalised by ISO as 6000 to 8760 hr/yr.
Baseline	A quantifiable physical condition or level of performance from which changes are measured.
Class	A collection of similar →*objects* is called a class.
Class Library	A group of C++ →*classes* collectively known as an application framework. These classes provide the framework and essential components of an application for the Windows graphical environment.
Degradation	The condition or status indicating impaired or →*deteriorating* condition, function or physical state.
Dependent Parameter	Dependent →*parameters* describe the engine's performance. They are measurable and they typically include parameters such as pressures, temperatures, power output and fuel flow.
Deterioration	Alternative term to →*degradation*.
Discrepancy	Deviation from an expected condition.
Engine Component	A part or assembly that is fundamental to the operating of the engine.
Engine Health Monitoring	The general discipline or technique for indication of status of the mechanical or functional condition of an engine or →*engine components*; sometimes referred to as Engine Condition Monitoring.
Engine Monitoring System	An Engine Monitoring System is a complete system approach to define engine, →*engine component* and sub-system health status through the use of sensor inputs, data collection, data processing, data analysis and human decision process. This system approach can consist of an integrated set of hardware and software and several separate engine monitoring system elements and can be manual, computer aided or automated.

Failure	A functional status or physical condition characterised by the inability of an engine, →*engine component* or sub-assembly to fulfil its design purpose; the most severe degree of →*malfunction.*
Fault	A change in one of the →*independent parameters.*
Fault Index	Used to define the magnitude of →*deterioration* for component fault sets. Changes in each of the →*independent parameters* in a component set will be a function of the fault index.
Fixed Time Maintenance	→*Maintenance* actions that are performed at certain points of time regardless of how well the engine is operating.
Free Turbine	Another term for →*power turbine*.
Gas Generator	A gas turbine that produces hot pressurised gasses with no mechanical output.
Gas Path Analysis	Gas Path Analysis is defined as a mathematical →*engine health monitoring* technique which uses the changes in →*dependent parameters* to identify changes in the →*independent parameters* which describe each →*engine component's* condition.
Handle	A set operating point or →*parameter* that is held constant, which all other parameters are measured relative to. The parameters are normally the measured dependent variables.
Hard Time Maintenance	A →*maintenance* scheme that carries out operations at specific time intervals even if the engine is still functioning well. Sometimes hard time maintenance is referred to as fixed time maintenance.
Incipient Failure	A functional status or condition before actual failure of engine, →*engine component*, or subsystem.
Independent Parameter	Independent →*parameters* are determined by the gas turbine engine's configuration and they are usually not measured. They include parameters such as flow capacity and efficiency.
Maintenance	Measures required to maintain and re-establish a specified condition, as well as to assess the actual condition of the technical capabilities of a given system.
Malfunction	Abnormal condition or status of the engine, component, or subsystem.
Modular Engine	The engine can be readily separated into subassemblies.

Monitoring	The act or technique of establishing functional status or condition.
Object	A combination of both data and the functions that operate on that data into a single unit.
On Condition Maintenance	A term used to indicate where →*maintenance* is not scheduled on a strict time interval or running time but on the actual condition of the engine.
Overhaul	The restoration of an item in accordance with the instructions defined in the relevant manual.
Parameter	A measurable or calculated quantity which varies over a set of values.
Physical Fault	A physical fault is represented or simulated by a component →*fault* set. This set is defined by one or more →*independent parameter* changes. Physical faults can be fouling, erosion, corrosion etc.
Power Turbine	A turbine mounted on a separate shaft from the compressor, from which power is extracted. Usually associated with a →*gas generator*.
Repeatability	Ability of an instrument of process to reproduce a certain output.

PART I

BACKGROUND AND OVERVIEW

1 Introduction

For both civil and military aero gas turbines, technological advances and the high cost of ownership have resulted in considerable interest in advanced maintenance techniques. Some of these techniques are now ready for application to industrial gas turbines. Combined cycle and combined heat and power gas turbines are now being widely adopted because of their acceptable levels of availability achieved, the low pollutant emissions, and their competitive capital and operational costs. The very large investment in combined cycle plants is putting increasing pressure on the need to address issues of availability and life cycle costs.

The interest in aircraft gas turbine monitoring resulted in an aerospace recommended practice guide (SAE, 1981). At Cranfield University work on engine health monitoring was started after a review of technical and economic possibilities in the application of engine health monitoring (Bourassa, 1984). Over the years the work has covered engine health monitoring systems (see Figure 1.1) including the role of Gas Path Analysis (Grewal 1988 and House 1992), other monitoring techniques such as oil and vibration (Markwell, 1985), and the possible application of knowledge based systems (Vivian, 1993).

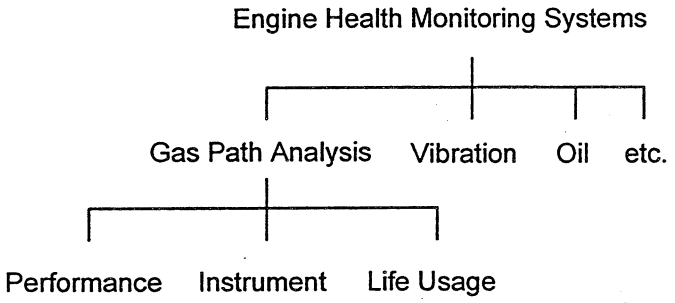


Figure 1.1: Overview Engine Health Monitoring Systems

The greatest emphasis at Cranfield over the years has been laid on the application of GPA for the identification of performance degradation (Grewal 1988 and House 1992) and of appropriate instrument sets for performance monitoring (LaGrandeur 1986), and also for studying life usage (Devereux 1994 and Wu 1994), particularly of hot section components (Ortiz 1987).

The present thesis concentrates on how GPA contributes to the management of availability with emphasis on industrial gas turbine engines.

Availability Concerns

Current gas turbines offered for base load operation have high levels of availability (ABB, 1993). However, when a forced outage is experienced, the down-time incurred depends on the time required to complete the necessary repair or maintenance action.

The largest contributors to forced outage rates are often engine support systems such as control and fuel systems. The down-time associated with these systems can be managed to acceptable levels by design redundancy and the holding of appropriate spares. Advances in instrumentation and microprocessor based controllers can be expected to contribute to further improvements in the availability of engine support systems.

In contrast, the major gas path components such as compressors and turbines have a high reliability. However, when a forced outage is caused by the failure of one of these components, the down-time experienced can be large. Both, the high cost of such components and the low likelihood that they will be required means that they are often not held as spares. For large base load combined cycle gas turbines, the manufacture of such spare components can take many weeks. Competitive pressures have resulted in a number of design advances. A number of large new machines may be suitable for a given specification. Hence, the number of machines of a particular model in use can be small, and it may not be attractive for even the manufacturer to hold a wide range of spares.

Table 1.1: System Ranking by Forced Outage Rate and by Forced Outage Total Down-Time (Singh, 1993)

Outage Rate	*[%]*	*Outage Total Down-Time*	*[%]*
Control system	24.2	Generator	17.0
General gas turbine	19.2	Turbine	14.4
Fuel system	13.1	Compressor	12.0
Lubrication	11.1	General gas turbine	12.0
Combustion	8.3	Lubrication	11.5
Generator control	6.5	Power distribution	9.6
Power distribution	5.4	Combustion	7.9
Generator	4.8	Control system	7.5
Cooling system	3.8	Fuel system	4.7
Compressor	1.9	Generator control	2.2
Turbine	1.1	Cooling system	0.9
Exhaust system	0.6	Exhaust system	0.3

Table 1.1 illustrates that gas path components contribute substantially to down-time even though they exhibit a high reliability. GPA offers the possibility of identifying degradations at the module level, determining the trends of these degradations during the usage of the engine, and planning the maintenance action based on this information. Parts can be ordered ahead of engine removal for maintenance action, and the appropriate maintenance and repair work package and team can be assembled to ensure that the down-time is kept to a minimum, and hence a high level of availability becomes possible (Singh, 1995).

Even with mature reliable engines which offer long periods of time between maintenance actions, relatively poor availability results if the down-time associated with the maintenance action is large, as shown in Figure 1.2. For example, if the time between maintenance actions is 20,000 hours and because of procurement of complex parts the down-time is 3 months, the availability achieved would be only 90%. If, however, through use of appropriate diagnostic techniques, the parts could have been identified and ordered some months ahead and down-time reduced to 7 days, then the availability would be 99%. This clearly offers a very significant improvement in plant economics.

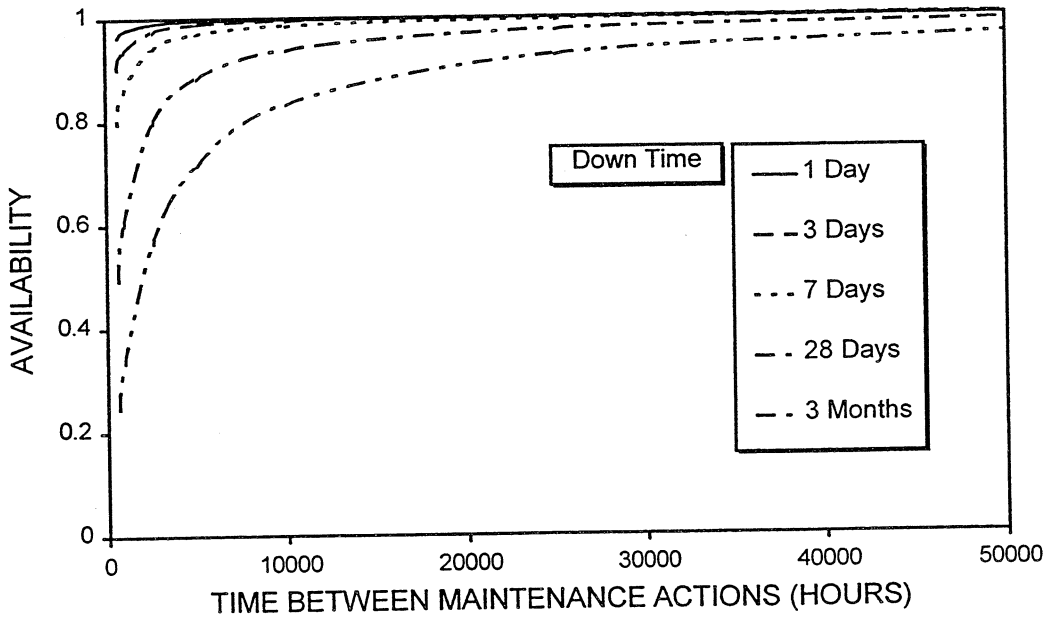


Figure 1.2: Availability vs Down-Time

Pioneering work in GPA was undertaken by Urban (1972), Boyce (1975) and Smetana (1974). But it is Urban's method which has proven to be the most effective for multiple

fault diagnostics (Urban, 1975). Urban's method relies on the implantation of known degraded components in order to generate analytically the fault coefficients. Once the coefficients are developed, a fault in a component is identified with the changes in measurements (Urban, 1969). A more flexible method for obtaining the coefficients is presented in Grewal (1988). The method uses the mathematical model of the engine to develop partial derivatives that accommodate small changes to the engine.

Urban's and Grewal's methods have been widely studied (Donaghy 1991, House 1992 etc.) and analysed (Lunderstaedt 1988, Mathioudakis 1994 etc.) and the methods have been incorporated in many diagnostic programs (Lazalier 1978, Grewal 1988, Doel 1994 etc.). However, both, the studies and the programs have been limited to the use of linear GPA. The limitation of linear GPA is that where the level of degradation that has to be detected is low, the residual error can be of the same order of magnitude as the degradation being sought. Non-linear GPA offers the possibility of accurately detecting almost any type of degradation (House, 1992). However, House' non-linear GPA method has only been applied to a simple shaft gas turbine for helicopters.

Thesis Objectives

At Cranfield University a performance diagnostics program called Detem (Grewal, 1988) has been developed. The program, whilst being applied, is still in the initial phase of use and as such certain 'bugs' and limitations exist. The most important of these limitations was that the program was limited to linear GPA. Therefore the aim of the present thesis is the development of a user-friendly software package that is based on recent experiences with the Detem program, with the important extension to non-linear GPA. The newly developed program, Pythia, is designed to a sufficient standard to make it suitable for conducting linear and non-linear GPA for any type of open cycle gas turbine engine under steady-state conditions. The successful implementation is followed by a test series in order to validate the program Pythia. Quality standards such as EN29000 are also introduced.

The thesis is divided into four main parts:

- Background and Overview
- Methods
- Case Studies
- Discussion and Conclusion

Chapters 1, 2 and 3 discuss the maintenance domain its problems and its limitations. Furthermore they provide a description of available engine health monitoring techniques. This is followed by a study of possible performance deterioration in a gas turbine.

Chapters 4 and 5 describe the methods that were used to develop the diagnostics program Pythia. Initially the linear and non-linear GPA methods are discussed. Then the program Detem is analysed so that the design specification of the new program Pythia can be specified. The analysis and design phase is presented within a commonly used structured methodology.

Chapter 6 discusses the application of Pythia to a variety of case studies. It is noteworthy that some of the cases were carried out by other students undertaking research in order to validate the program Pythia and to demonstrate its user-friendliness. However, the main results were obtained from two industrial gas turbine engines of similar configuration. The results show the significant improvement of the non-linear GPA technique over the linear.

Chapter 7 analyses the importance of the results and their implications. Although the present thesis concentrates on industrial gas turbines, most of the issues discussed are applicable to both types of engines: aero and industrial. The thesis concludes that the program Pythia can play a major role in planning maintenance strategies for the gas path components of a gas turbine engine. The user-friendly program is able to analyse deteriorated gas turbine performance with greater certainty and accuracy than the earlier linear GPA methods.

2 Engine Maintenance and Engine Health Monitoring

The last decades have seen an increasing number of developments in systems for operating and maintaining power plants. As a result of increasing competition, users of industrial plants try to run engines profitably and safely for the whole engine life cycle so as to keep the plant's cost of ownership as low as possible (Agrawal, 1979).

The required life of the plant will merely relate to the period of time that it is being used economically. For most industrial plants, this period of time will considerably exceed the average life of many of the plant components. An essential aspect of the plant specification and design is, therefore, to determine which of the critical components are the most likely to fail and then to ensure that these have easy and timely maintenance access for repair or replacement. It is also essential to make economical assessments of the relative costs of critical components, of installing parallel stand by units to take over when one unit fails, and of the profit or loss that will occur when the whole plant has to be stopped for repair.

The following sections are concerned with engine maintenance and engine health monitoring methods. An overview of some possible engine maintenance strategies is first discussed to demonstrate the importance of choosing an appropriate maintenance strategy. This is followed by a historical background of engine health monitoring systems. Then, the most common methods of engine health monitoring systems are discussed. It is important to note that engine health monitoring systems were initially established for aero industry applications.

2.1 MAINTENANCE METHODS

'The in-service costs of running a power plant, of whatever type, generally far outweigh the initial capital cost of design, construction and commissioning. In managing the engineering risk associated with the plant, therefore, maintenance becomes a key issue' (Kirwan, 1995). With more operators looking at plant optimisation, safety and environmental issues a strategy which makes the best use of maintenance methods can then be developed. The three maintenance strategies discussed are:

Run-to-failure Run-to-failure maintenance is a strategy that maintains high output levels for as long as possible until a breakdown occurs

Preventive Preventive maintenance is a strategy that operates a planned servicing routine to prevent failure

Predictive Predictive maintenance also known as condition based maintenance is a strategy that predicts component failure using data derived from some form of condition monitoring.

At first sight, the best method of running a plant would be to operate it until it breaks down and then repair it. However, this method is not always desirable because a failing component can cause damage to others and the consequential costs can be very high. In addition, the failure may occur at an inconvenient time at an inconvenient place, so that the necessary staff and replacement components are not available. Finally, the plant may not be allowed to be stopped at short notice because it provides an essential service such as power supply.

Therefore, preventive maintenance where a plant, an engine or an individual component is withdrawn from service at regular intervals for inspection and repair has been the traditional approach to run a power plant (Kirwan, 1995). However this is not the optimum method for achieving the maximum reliability and safety because engineering components do not fail at regular intervals. Consequently when regular preventive maintenance is adopted three problems may arise (Neale, 1987):

Unexpected Failure Some failures will continue to occur between the overhauls and these may be unexpected and causing inconvenience.

Unnecessary Overhaul During the overhaul many components in good condition will be stripped and inspected unnecessarily and in some cases their condition after reassembly can be worse than it was before the overhaul.

Long Overhaul The overhaul process, involving the examination of large numbers of components takes a considerable time and it can result in major profit losses especially for power generation plants.

The obvious answer to these problems is to move from preventive maintenance to predictive maintenance also known as condition based maintenance (Kirwan, 1995). With condition based maintenance, the health of the plant is determined by monitoring the engines condition, hence the appropriate maintenance action can be carried out before a fault occurs. However, this is an ideal situation and before such a maintenance strategy is adopted some of the following issues need to be addressed:

Economics Since it is not economical or practical to monitor every component the question is raised how are the critical components to be selected (Neale, 1987)?

Critical Components Since condition monitoring involves the measurement of trends or excessive variation from a desired level of some characteristics, it is only suitable for components where a failure occurs in a time-dependent fashion. This is fundamentally because it is the progression of the failure, which gives the lead time required to instigate the necessary corrective actions. What can therefore be done about critical components which do not fail in a relatively slow progressive manner (Neale, 1987)?

Fault When eventually a monitoring measurement indicates that there is a fault, the user of the plant needs to determine whether the fault is a real fault and a shut-down of the plant is justified or whether the fault is related to other problems such as measurement errors etc. (Neale, 1987).

Operator While one of the great advantages of condition monitoring is that it is well matched to the tendency of components to fail at random intervals, an inherent feature of this is that each component must be monitored regularly and reliably, possibly for years waiting for the random event to occur. This is not easy to manage with operators since they easily get bored. Could this task be (partially) allocated to systems or machines as opposed to operators?

A possible answer to these questions is the adaptation of a computerised engine health monitoring system (EHM). But before doing so, some basic maintenance concepts will be examined.

Regular preventive and condition based maintenance require both a reasonably uniform operating environment, to justify the regular overhaul interval and the concept of regular condition trending. Unfortunately, for many items of equipment, it often has random overload situations imposed upon it. Typically, gas turbines operate over a wide range of temperatures, speeds, power settings and changing environments. During their service, gas turbines ingest particles such as sand, dust, water droplet, ice, fly-ash, salt, etc. at a high rate. For example, a 7.5 MW unit will ingest 4.5 kg of foreign material in a single day assuming one part per million enters the compressor (Lakshminarasimha,

1994). With such a high rate of foreign particle ingestion, even with a good inlet filtration system, clean fuel or other ideal running conditions, the engine gas path components will deteriorate and therefore the engine performance will get progressively worse with increasing operating time. In other words, a change in performance occurs because of physical faults. In certain areas of machine application these random events may tend to occur at a greater frequency than the progressive failures. For example in certain parts of the world the ingestion of large birds into aircraft gas turbines may occur more frequently than bearing failures. In power station coal mills producing pulverised fuel, the occasional ingestion of random pieces of metal or very hard rocks can cause internal breakages more often than progressive failures in the transmission. The aforementioned phenomena are shown in Figure 2.1. Essentially, there are three types of failures common to all machines: time-dependent, delayed time dependent and instantaneous failures (Saravanamuttoo, 1983). These failure modes can also be described as bath-tub curves (see e.g. Sapsard 1994 and Day 1987).

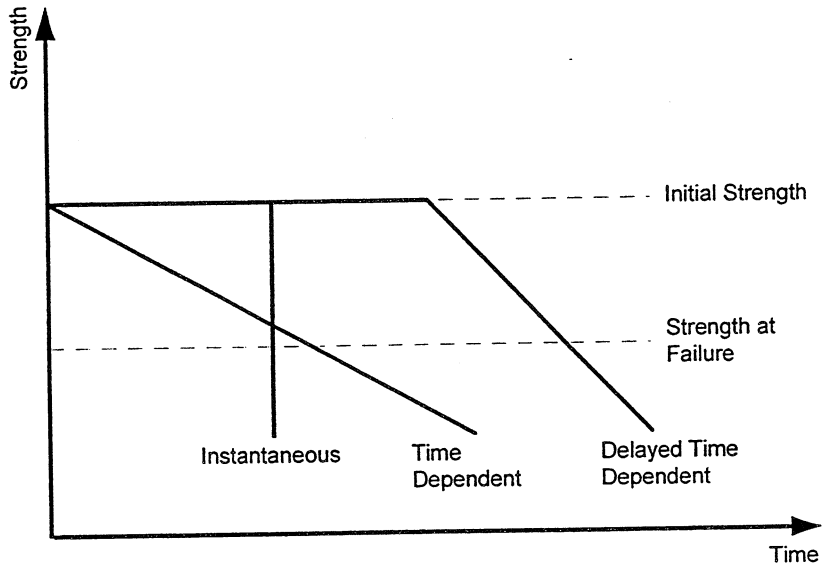


Figure 2.1: Types of Failures (Saravanamuttoo, 1983)

An instantaneous failure will occur without any warning. No monitoring system is capable of identifying such a failure. A typical instantaneous failure is bird ingestion into an engine. The delayed time-dependent failure is a failure where for a certain time no deterioration is detectable until at some point the deterioration is observable. Finally, the time-dependent failure will degrade an engine component at a well-defined rate. Both the dependent and delayed time dependent failures are ideal candidates for a

monitoring system. In some instances, the system design allows the safe continued use of the power system even after the instantaneous failure of a given engine. Examples are large civil passenger aircraft and systems providing power for safe shut-down of nuclear power plants..

In circumstances where very high availability is critical, some users of industrial power plants seek to achieve this by having a number of engines installed to cover the load, including some redundancy in the number of engines installed over the number of engines required. It is sometimes not appreciated that the availability of the plant can be very low even if the availability of individual engines is high, when a high number of engines are installed to achieve the plant requirement. This is illustrated in Figure 2.2. Hence the installation of multiple engines, far from guaranteeing a high availability, often requires a closer management of plant availability (Singh, 1994)

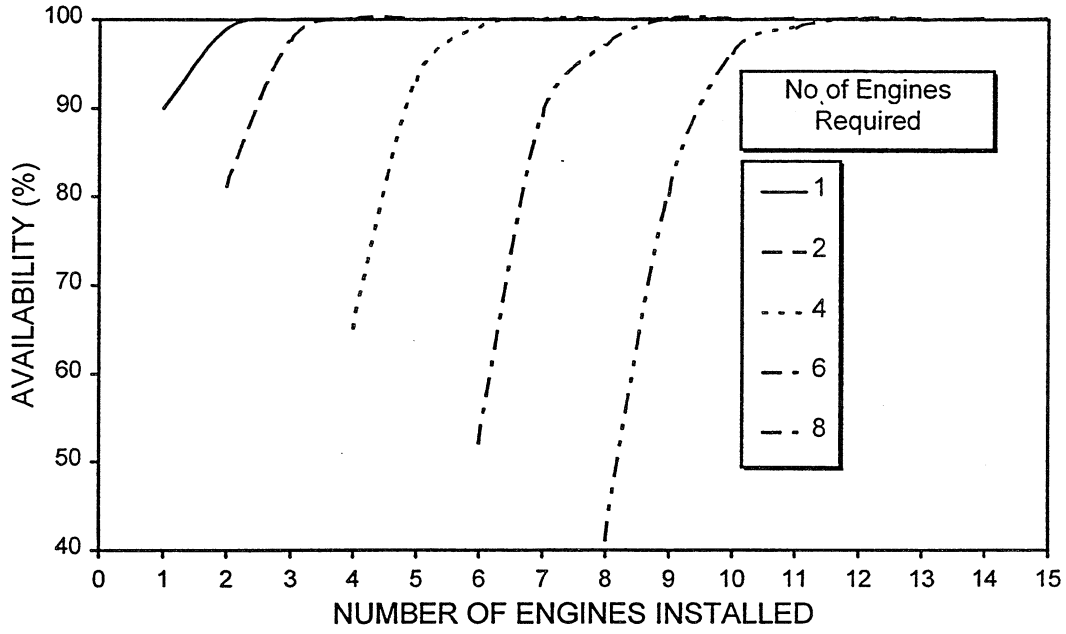


Figure 2.2: Plant Availability (Engine Availability = 90 %)

This leads to the conclusion that any overhaul system that is based on fixed time intervals is insufficient since overhauls should be determined by the random events.

There will be cases where regular preventive maintenance is the most suitable strategy for an operating environment that is fairly monotonous and the maintenance activity is associated with some kind of capacity absorption rather than a failure as such. Typical

examples are the need to change filter elements which become clogged with debris or where a sacrificial corrosion control device becomes consumed (Neale, 1987).

Therefore, in order to set up a maintenance system based on machine condition and its usage, the system should be able to identify a failure in a component and to state when the failure cannot be related to monitored parameters.

The first problem requires an appropriate choice of monitored parameters, since it is uneconomic to monitor all components. This requirement can be met by considering the likelihood of the failure, the effect of a failure and the time required for repairing the faulty component. This allows the identification of failures that are related to progressive wear or fatigue pitting of component surfaces. In addition, it detects progressive changes in the dynamic forces of moving components and progressive changes in the performance of components or systems (Neale, 1987).

The second problem refers to failures that cannot be directly monitored and are mostly related to structural components when they are subject to fatigue or creep. Components such as turbine discs or rotors can generate internal cracks which do not effect the external dynamic forces, and therefore are not detectable for example by vibration measurement. In practice these parts will come apart without a prior indication from current monitoring methods.

Problems of this kind must therefore be dealt with differently. Typically these failures are related to usage and operating conditions. If the relationships are understood then the monitoring of the relevant usage parameters may permit a reasonably accurate estimation of the likely remaining life at any time (Sapsard, 1995). However, the determination of the life of a component is not easy since it requires an assessment of the likely variation in material properties between one component and another. Also, if the operating conditions vary then the consumption of safe life of a component changes considerably (Sapsard, 1995). The aircraft engine industry is quite familiar with these problems and since imminent failures cannot be detected, indirect methods of measuring life consumption are used. These are based on the way in which engines are used and include measurements of speed, temperature and time which are related to the life that would be consumed in the weakest component (Wu, 1994).

In some situations the strategy of measuring life of components is more expensive than simply changing removable modules or subassemblies on a safe life basis. This strategy

has the advantage that while the original parts are overhauled the plant is in-service with a replaced part.

Another area which needs to be addressed is the way a plant is operated in case of a fault. In an aero gas turbine the crucial issue is safety whereas in a power generation plant the costs are important in case of long down-time. One way to solve the problem is to get the information from more than one monitoring system. This has the advantage that the operator has more confidence when two or more independently operating systems indicate the same failure.

But the main source of concern for a maintenance system is the operator himself. Some organisations, particularly the civil airlines have shown that a user-friendly and validated engine health monitoring system will find good acceptance by the operators of the engine (MacIsaac, 1992). An ideal system for performing routine data handling is the computer. This gives the operator time to concentrate on more important tasks.

2.2 THE DEVELOPMENT OF ENGINE HEALTH MONITORING SYSTEMS

Engine health monitoring (EHM) implies the ability to accurately assess the relative health and performance of an engine in a reliable, cost effective and technically sound way. The equipment, procedures and people necessary to provide EHM data is termed engine monitoring systems (EMS). An EMS collects, processes and displays data with the purpose of permitting meaningful conclusions on engine status to be drawn from measurable data in a cost effective fashion (Lynch, 1989). The following section briefly discusses the historical background of gas turbine engine maintenance philosophy and the reasons for EHM. The historical discussion then continues by reviewing typical approaches to EMS as applied to both aviation and industrial gas turbines. The review of EMS concludes with the description of engine health monitoring methods, their benefits and limitations.

Historical Development

In today's aero, marine, pipeline and power generation industries gas turbines play an increasingly critical role. The costs of fuel, engine price, spare parts and maintenance and overhaul expenditure over the gas turbine's life cycle have increased. Moreover, high penalty costs associated with non-availability and catastrophic failure have resulted in the need for health monitoring systems (MacIsaac 1992, Saravanamuttoo 1983).

The origins of EHM lie in the monitoring of aircraft engines and the comparison of engine performance data. This was an inherently inaccurate process at best, due to differences in production tolerances between individual engines, differences in total engine time, and human unreliability. Eventually this rough procedure was superseded by the use of engine cruise performance charts and engine performance calculations. The monitoring capabilities provided by cockpit instrumentation and ground testers formed the bases for more sophisticated techniques (LaGrandeur, 1986).

Furthermore, aircrafts have developed from simple piston engine powered airframes of tens of kilograms weight to complex multi spool bypass gas turbine powered airframes of tens of thousand kilograms weight. As the powerplants have evolved into more and more complex systems, so have their maintenance procedures grown from a semi skilled craft to a profession.

During the year of the DC-3 aeroplane, the maintenance philosophy was based very much on the time honoured system of a fixed time between overhauls. Overhaul was regarded as complete disassembly, inspection of component parts and replacement or repair of damaged parts prior to reassembly and testing.

The philosophy that regular complete overhauls were an effective strategy for preventing in-service failures prevailed for nearly 25 years. Little information was available to support or to challenge this belief. It was also assumed that each item had an appropriate overhaul time that could be identified. Thus, the basis for extending the overhaul time limits was a complete tear down inspection of a number of serviceable items that had reached their age limit. It was followed by an evaluation of each component part and by a judgement of whether the part could have continued to operate to a new limit.

As these tear down inspections provided no objective basis for extending the overhaul intervals, the continued viability of operating the new overhaul time limit was assessed by monitoring the failure rates and failure modes of the item at these new limits thereby ensuring that the reliability at the new limits was satisfactory. If the reliability was not adversely affected then the cycle was repeated in an attempt to ultimately identify the correct overhaul interval (trial-error).

Obviously, this process was both time consuming and costly. With the advent of gas turbine engines and modern jet aircraft to commercial aviation a better understanding of

the maintenance concept was required. This realisation lead to the establishment of the so-called reliability centred maintenance (RCM, Sapsard 1994).

RCM seeks to plan and manage maintenance activities. RCM tries to match the failure characteristics and the consequences of failure to a maintenance program that maintains an item to a desired service level most economically and effectively. RCM recognises failure as real and is directed at either preventing failure or reacting to the failure, depending upon the consequences of failure.

As overhaul intervals escalated, it became even more difficult to obtain sample engines that had reached their overhaul time limit, without having had at least one repair, during which many assemblies or parts were replaced. Emphasis thus had to move to evaluation of the condition of individual time expired parts.

By 1968, with the introduction of the scheduled maintenance program for the new wide body Boeing 747 powered by high by-pass ratio engines, three primary maintenance processes were put forward:

Hard Time — A maintenance scheme that carries out operations at specific time intervals even if the engine is still functioning well (LaGrandeur, 1986). In particular an engine or an engine part is periodically overhauled in accordance with the operator's maintenance manual or it is removed from service.

On Condition — A maintenance scheme where the maintenance is not scheduled on a strict time interval or running time but on the actual condition of the engine (Wright, 1990). In particular an engine or an engine part is periodically inspected or checked against some appropriate physical limits in order to determine whether it can continue in service. The purpose of the standard is to remove the engine from service before failure during normal operation occurs.

Condition Monitoring — The complete process of defining engine, engine component and subsystem health status through the use of sensor inputs, data collection, data analysis and human decision processes (LaGrandeur, 1986).

With the introduction and acceptance of three basic maintenance strategies, work now focused on more reliable and cost effective methods of monitoring gas turbine

performance. The methods employed included visual inspection procedures, monitoring techniques such as oil analysis, performance monitoring and trending. The objective was to identify incipient failures in a timely manner.

Engine Monitoring Systems (EMS)

Probably the first attempt at what could be called an engine monitoring system (EMS) was instigated in the mid 1950's, a system called the 'time-temperature recorder'. It was built for operational use in the J-47 engine for the USAF F86 aircraft. However, due to both mechanical problems and poor reliability, the system was never used operationally.

The development of automatic onboard recording systems lead to the application of more sophisticated trend analysis techniques by land based support computers. Thus, the first generation of Airborne Integrated Data systems (AIDS) was implemented. Since the first introduction of AIDS on wide body aircraft, the benefits of EMS have been identified by users and proclaimed by suppliers. Such benefits include improved flight safety, reduced aircraft turn around time, improved reliability, improved maintenance scheduling, reduced maintenance costs and improvements in engine diagnostics and prognostics have all been attributed to an EMS. Although the areas of replacement have been identified, the actual saving associated with the implementation, and hence the return on investment has not always been so easy to identify. Thus, in an attempt to clarify the area of EMS the Society of Automotive Engineers (SAE) published the first EMS guide ARP 1587 (SAE, 1981).

ARP 1587

ARP 1587 is intended as a guide for EMS definition and implementation, addressing benefits, capabilities and requirements of an EMS. Intended for all interested parties including users, manufacturers of aircraft and engines and equipment suppliers, the document broadly deals with an EMS.

A fundamental concept in ARP 1587 is that an EMS must address all aspects of the organisation, personnel, software and hardware to be at least successful in its goals of reducing maintenance intervention. This is clearly depicted in Figure 2.3.

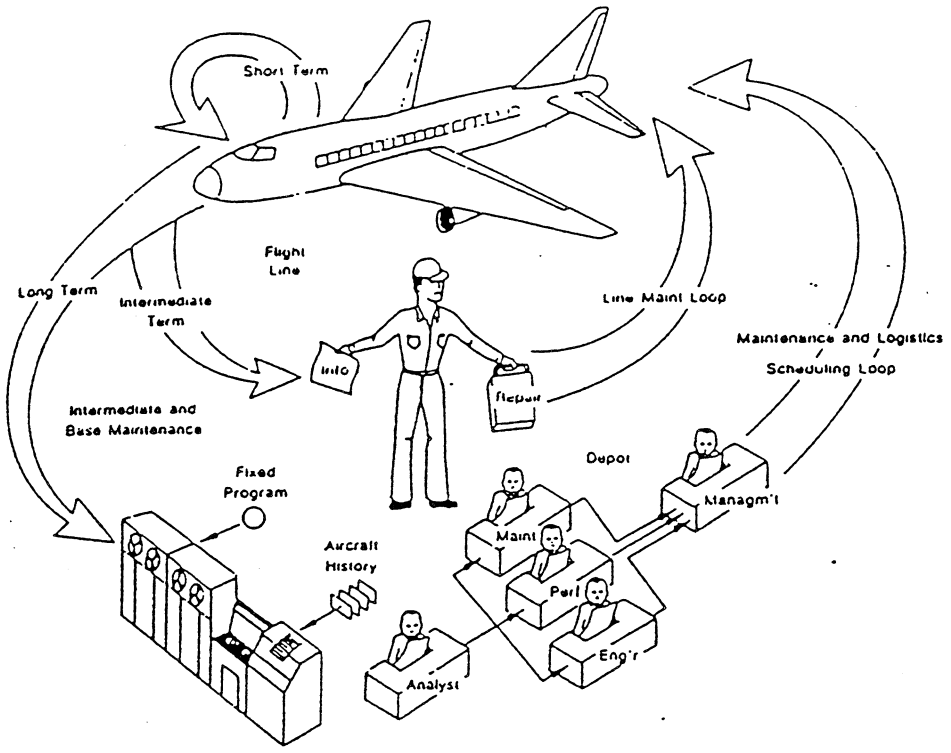


Figure 2.3: Engine Monitoring Systems General Description (Hess, 1983)

ARP 1587 identifies that, similar to other projects, "people are the key to success". The EMS designers following ARP 1587 are led to the understanding that no engine monitoring system meets the needs of all engines, aircrafts and organisations. Instead, the designers must tailor the systems available to match their requirements, and future growth requirements, as closely as possible (Lynch, 1986).

The remaining sections of this chapter are devoted to reviewing what aero and industrial engine monitoring systems have materialised in the past, and what type of systems are available at present.

Engine Monitoring Systems - Aero Applications

For aero applications, engine monitoring started with flight engineers comparing the levels of engine to engine performance of the aircraft. It was soon realised that this provided little useful information, due to the production tolerance variation inherent in the manufacture of the engines and due to the variation of engine performances. Different maintenance requirements of the individual engines further reduced any similarity between the supposedly identical engines (Lynch, 1989).

Consequently, the procedure was changed to one that reviewed individual engine performance to the manufacturer's cruise charts, with the follow up of engine performance calculations. This procedure allowed a comparison of engine performance without correction for altitude, forward flight speed or engine thrust settings. This procedure was based on manual recording and subsequent trending and further trend analysis of key engine parameters. The trend of the deviations of the real engine in comparison to the engine model or the cruise charts as a function of time, can indicate potential engine failures (Lynch 1989, Verte 1995 and Dehu 1995).

With the advent of cheaper and more powerful microprocessors, the ability to read, store and manipulate data was increased by orders of magnitude. Subsequently, the manual recording previously performed by the flight engineer was performed automatically. Thus the total number of feasible measurements and concurrently the objectives of the EMS were increased.

Commercial aviation EMS has focused on a cost effective approach to achieve lower operating costs (Burnell, 1995). Military use of EMS has, however tended to focus on better aircraft availability and mission success although reduced aircraft support costs have also been important considerations. The Air Forces reporting on EMS have different goals, but the philosophy of the systems is to enable a decision as to whether the engine/aircraft can undertake the next mission, or during missions to warn the pilot of any engine anomaly serious enough to justify aborting the mission (Goh 1989, Hess 1983 and Goodfellow 1988).

Engine Monitoring Systems - Industrial Applications

Condition monitoring techniques in industrial applications have largely concentrated on vibration monitoring and vibration analyses (Lynch, 1989). Gas turbine engines were predominantly of heavy-duty built industrial type engines. The engines were viewed as similar to any other piece of plant, and maintenance was performed at fixed intervals.

With the adoption of aero derivative engine technology, aero EMS experience accumulated and many users of aero derivative engines reviewed their maintenance philosophy. Consequently, many operators are now using on-condition criteria for maintenance activities (Agrawal 1979 and MacIsaac 1992). However, it must be observed that engine monitoring systems for industrial applications lag behind those initiated for the aero applications.

Consequently, the procedure was changed to one that reviewed individual engine performance to the manufacturer's cruise charts, with the follow up of engine performance calculations. This procedure allowed a comparison of engine performance without correction for altitude, forward flight speed or engine thrust settings. This procedure was based on manual recording and subsequent trending and further trend analysis of key engine parameters. The trend of the deviations of the real engine in comparison to the engine model or the cruise charts as a function of time, can indicate potential engine failures (Lynch 1989, Verte 1995 and Dehu 1995).

With the advent of cheaper and more powerful microprocessors, the ability to read, store and manipulate data was increased by orders of magnitude. Subsequently, the manual recording previously performed by the flight engineer was performed automatically. Thus the total number of feasible measurements and concurrently the objectives of the EMS were increased.

Commercial aviation EMS has focused on a cost effective approach to achieve lower operating costs (Burnell, 1995). Military use of EMS has, however tended to focus on better aircraft availability and mission success although reduced aircraft support costs have also been important considerations. The Air Forces reporting on EMS have different goals, but the philosophy of the systems is to enable a decision as to whether the engine/aircraft can undertake the next mission, or during missions to warn the pilot of any engine anomaly serious enough to justify aborting the mission (Goh 1989, Hess 1983 and Goodfellow 1988).

Engine Monitoring Systems - Industrial Applications

Condition monitoring techniques in industrial applications have largely concentrated on vibration monitoring and vibration analyses (Lynch, 1989). Gas turbine engines were predominantly of heavy-duty built industrial type engines. The engines were viewed as similar to any other piece of plant, and maintenance was performed at fixed intervals.

With the adoption of aero derivative engine technology, aero EMS experience accumulated and many users of aero derivative engines reviewed their maintenance philosophy. Consequently, many operators are now using on-condition criteria for maintenance activities (Agrawal 1979 and MacIsaac 1992). However, it must be observed that engine monitoring systems for industrial applications lag behind those initiated for the aero applications.

Industrial use of an EMS is not as extensive as aero use, this being due to the previously lower technology engines employed in industry. However, with the recent widespread use of aero derivative engines the situation is improving, as the aero experience in EMS is being adopted by industry. So far no equivalent guide to ARP 1587 has been elaborated for industrial applications. It must be questioned though, whether a guide is required since ARP 1587 introduces the concept of total system definition for an EMS rather than a set of rules and guidelines. Such an approach is equally suitable for industrial as well as aero use of an EMS. Instead, what must differ is the actual implementation because of the different needs and capabilities of industrial users (Lynch 1986 and Wright 1990).

2.3 ENGINE HEALTH MONITORING TECHNIQUES

Engine Health Monitoring (EHM), or Engine Condition Monitoring (ECM), are the terms usually applied to the many methods used for the surveillance of gas turbine power plants. In a general sense, any engine may be viewed as consisting of three general areas such as accessory equipment, rotational mechanical equipment, and the thermodynamic gas path elements. The accessory equipment includes elements such as the fuel control, fuel system, lubrication system, ignition system, and engine air bleed system. The rotational mechanical equipment includes the various main engine bearings, rotors, and gear trains. The gas path elements include the gas containment path, the compressors, the burners and the turbines. In view of the fact that physical problems may exist in any of these general areas a totally integrated EMS must have a proper balance of emphasis on all of them. The relative attention required by each of the three general areas depends on the nature and probability of occurrence of possible physical problems. The information gathered from the three areas may often be used in a cross complementary fashion to verify diagnoses and more precisely isolate faults (Urban, 1975).

Engine monitoring systems of almost limitless variety can be constructed from these general areas and the vast catalogue of available EHM tools and techniques depend only on the needs and capabilities of the individual organisation.

Diagnostic treatment of the accessory equipment is a subject in itself and has been expertly dealt with by others in prior published reports (e.g. Rolls-Royce 1986). Without detracting from its importance in any way, the following section concentrates on health monitoring techniques that are applied to monitor rotational equipment and gas path elements.

Applications for Engine Health Monitoring

With the increased interest in health monitoring in recent years there have been a number of developments in the techniques that are used. Most of these techniques fall into following monitoring categories:

- Mechanical integrity
- Wear debris
- Vibration
- Performance

In addition to these techniques, there have also been developments in the methods for monitoring the amount of life usage of components and systems which have expired during past service. This is based on recording the operating or environmental conditions to which a component has been subjected and is commonly applied when the failure mechanism is structural fatigue, creep or active corrosion (Wu 1994, Harrison 1995, Sapsard 1994 and Devereux 1992). These additional techniques, although important, will not be considered in the present thesis since they are a task in itself.

Mechanical Integrity Monitoring

Mechanical integrity monitoring technique deals with the mechanical health of the engine. Typically, the technique is subdivided into internal and external visual inspection. External visual inspection is the most effective health monitoring technique. It typically detects leaks of gas and fuel, security of pipes, accessories and control linkages. However, it is the internal inspection (endoscope) which provides a simple and effective technique for direct assessment of internal components. Both techniques, endoscopic and external inspection are equally applicable to both aero and industrial engines but they are not always applied to the same degree

Endoscope Inspection

The use of optical devices for inspection provides a simple and effective technique for direct assessment of internal components of gas turbines. It helps to detect cracks, nicks, tears, burns, distortions, deposits etc. and it provides an accurate basis for maintenance decision. However the technique relies on visible changes.

Typically the instruments that are used for internal inspection are termed 'endoscope' instruments. They encompass the following two probes (LaGrandeur 1986 and Lynch 1989):

Flexible Flexible probes are referred to as fibrescopes, flexible endoprobes etc. They have the advantage of enabling views of cavities and views of components which are not accessible in a straight line. However, these systems are expensive and the views is not always clear because of loss of light. The loss of light is due to the flexible fibre optic cables.

Rigid Rigid probes are referred to as borescopes, rigid endoscopes etc. They are more robust and cheaper than flexible probes. However they are restricted to views that are accessible in a straight line.

A typical endoscope system is shown in Figure 2.4.

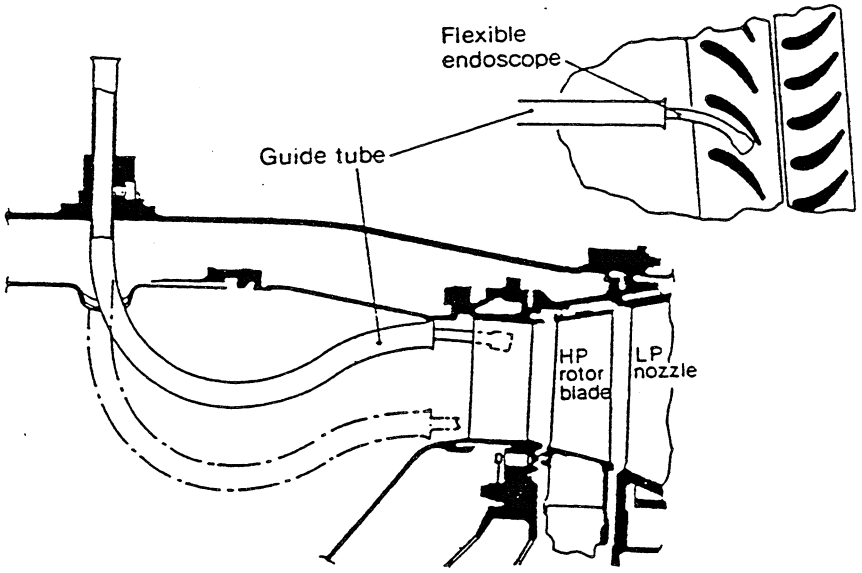


Figure 2.4: Flexible Endoscope System Capability (Lynch, 1989)

It consists of a solid or flexible fibre optic probe that is coupled with a light source to provide images for a video recorder or camera. The advantage of such a system is that pictorial trends can be maintained for monitoring the progressive growth of distress in a component. Things such as fluid leaks, blade defects in initial compressor stages,

cracks, corrosion, build-ups, insecure parts, jetpipe defects, and general observations are easily detectable and play an important role in a health monitoring system. A further advantage is the possibility for installing an endoscope video system which provides a real time image which can be viewed simultaneously by a number of people in comfortable surroundings. It is possible to monitor and record many different aspects and internal areas of the engine. This facilitates the training of personnel in inspection techniques (LaGrandeur, 1986). The main disadvantage of the endoscope system is its dependence on access ports being incorporated in the engine to accommodate the probe. Also, the diameter of the ports are invariably small and these limit the light gathering ability, resolution and general viewing capability of the probe. Moreover, many older engines were not designed for endoscope purpose therefore they lack access for ports. The fact that endoscope analysis can only be carried out on the ground prevents it from being useful as an airborne monitoring technique.

Although the effectiveness of this monitoring technique was originally limited to second generation engines with few access areas, more modern third generation engines allow easier access to more ports. This is demonstrated in Figure. 2.5 (Lynch, 1989) where endoscope ports are provided in the latter stages of the low pressure compressor as well as those in the combustor and turbines which are typical of previous generation engines.

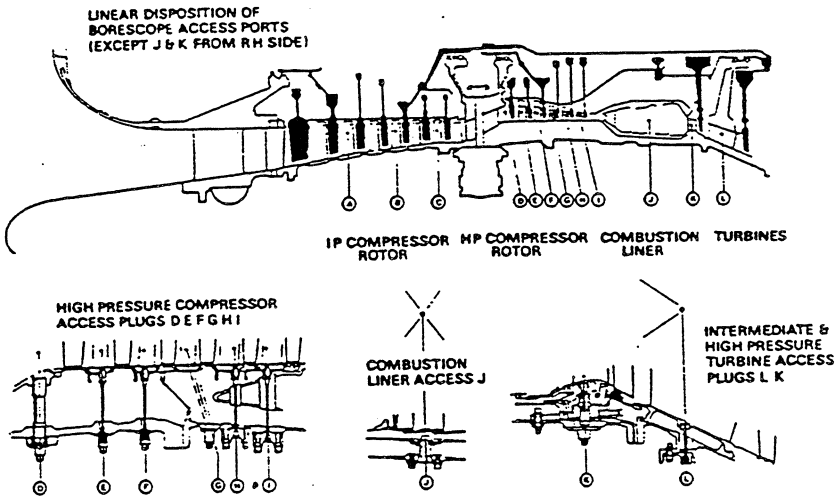


Figure 2.5: Borescope Access Points (Lynch, 1989)

Although visual inspections, both internal and external, are regarded as effective techniques in their own right, even with the most sophisticated data acquisition and diagnostic system it is advisable to carry out visual inspections before a diagnosis is confirmed or a maintenance action undertaken (Lynch, 1989).

Wear Debris Monitoring

Wear debris monitoring has been the oldest health monitoring technique. The term wear debris has become synonymous with oil analysis. The technique originally relied on specimen extraction. Either an oil sample was taken in order to assess its contained wear debris, while at the same time the oil condition was checked or a wear debris sample was taken from devices in the flow path such as a magnetic plug or a filter.

Typically, wear debris occurs in parts of the gas turbine engine such as bearings, gears, splines etc. These parts are in constant motion and hence removal of material by mechanical action occurs (friction). Oil is pumped to lubricate (decrease the friction) and cool these parts. The high temperature of the components changes the physical properties of the oil and some of this oil is consumed in the process. A sudden rise in, or increased level of oil usage can be an indication of internal deterioration of the lubricated mechanical parts or the whole engine. Usually the engine components do not fail suddenly. Before actual failure, there is a period in which wear particles are released at a greater rate than normal. By observing and analysing these particles, failures can be predicted. Basically, there are three methods which allow monitoring of the oil wear debris:

Capturing	Methods which capture debris for later quantitative and qualitative evaluation
Counting	Methods which count debris particles as they pass in the scavenge oil flow
Analysis	Analysis of oil samples - usually under laboratory conditions

The first two methods may be combined in a single sensor, such as a magnetic chip detector. As this detector is installed on the engine it can give a rapid response to an increase in wear rate. However, inspection of the particles is essential to permit interpretation of the wear mechanism involved in their generation. Once the wear mechanism is known the failure mode which the engine component is experiencing can be predicted. However, this latter more informative screening of the wear particles is

time consuming and requires some knowledge of the engine failure modes and wear mechanisms.

The ferrography method is a recent development in an attempt to provide a direct on-line method for separating, grading and assessing wear debris generation. Alternatively, ferrography is used with oil samples to speed up laboratory testing. The ferromagnetic wear debris is graded on a substrate by passing the oil through an unhomogeneous magnetic field. As its name implies ferrography is limited to the detection of ferromagnetic debris only.

Analysis of oil samples, such as off-line ferrography is a useful technique, but it is not the only technique available. A highly successful technique adopted by many engine operators is the Spectrometric Oil Analysis Program (SOAP). It involves burning an oil sample in such a way that the oil debris emit characteristic light spectra. The wave lengths and their respective relative intensities are used to identify the quantity and types of oil debris. This allows trends to be kept on engine health. Its disadvantages are the high cost of analysis equipment, the requirement for trained personnel and the inevitable delay in response time before results are available.

Oil analysis either on the engine or off the engine provides useful information. The amount of information, however, depends upon the engine in question and the type of deterioration which can be anticipated. Using the predicted failure mechanisms allows an estimate to be made of the wear particle size and distribution. This is important because each of the above mentioned methods has different ranges of particle size over which it is efficient in operation (see Figure 2.6). Thus, the method of oil monitoring/analysis is dependent on the engine and its application (Sapsard 1994, Lynch 1989 and Grewal 1988).

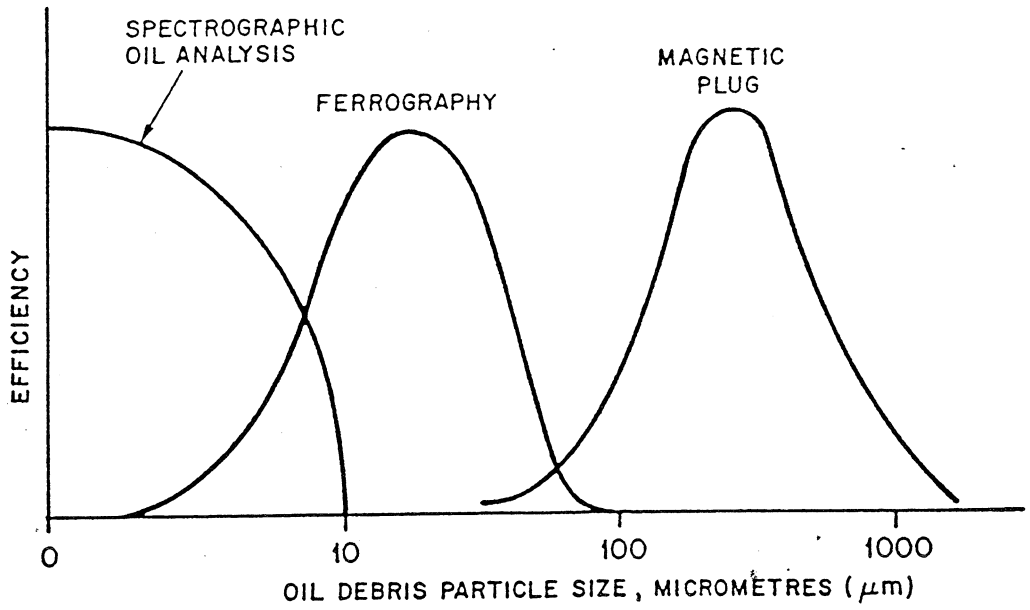


Figure 2.6: Efficiency of Debris analysis versus Wear Debris Particle Size (Lynch, 1989)

Gas Path Debris Monitoring

Gas Path Debris Monitoring is a recent development in engine health monitoring techniques. The technique allows to monitor electrostatically charged debris present in the engine gas path. This is achieved by mounting sensors in the gas path and then monitoring the electrostatic charge induced on these sensors. The sensors are suitable for the environment and requirements of the gas turbine engine. Electrostatically charged debris is produced by gas path faults such as blade rubs and combustor burns (Fisher, 1995).

Gas Path debris monitoring technique is able to detect actual deterioration to components rather than the secondary effects of deterioration. The technique can identify very small changes in the electrostatic signal characteristics, even in the early stage of fault development where only small amounts of debris are being produced. For this reason, the technique can provide an early indication of developing fault conditions, often before a deterioration in performance occurs. Typically, the monitoring system can detect debris from a few microns in size up to objects such as bolts or pieces of blades. The monitoring technique is suitable for airborne, marine and industrial applications. So far, the technique has been developed into two separate systems such

as the engine distress monitoring system (EDMS) and the ingested debris monitoring system (IDMS).

The EDMS has been developed to monitor debris present in the exhaust gas of the engine. The objective of the system is to provide an early warning of developing fault conditions, and to identify the fault and discriminate the source to module level. The system is able to detect debris-producing faults in the gas path components such as blade rubs, combustor degradation and ingested materials or objects.

The IDMS has been developed to monitor debris ingested into the engine intake. It can detect fine particulates like sand and dust as well as larger single objects like stones, bolts, leaves and insects.

Both the EDMS and IDMS can be installed as stand-alone systems or they can be integrated with each other or with existing compatible engine health monitoring systems. The strength of gas path debris monitoring is that it is able to provide an early warning of developing gas path component fault conditions, often before a deterioration in performance occurs. Unlike many of the other engine health monitoring techniques, it detects actual deterioration rather than the secondary effects. Gas path debris monitoring is able to detect both single and multiple faults and to easily discriminate between different fault conditions.

The limitation of gas path debris monitoring is that it is not able to quantify the amount of deterioration that has occurred. Furthermore, it diagnoses faults in terms of changes in parameters like activity level and event count rather than in terms of actual physical faults. Therefore the system user needs to have a good understanding of the system to be able to diagnose faults from changes in the system parameters (English 1995, Fisher 1995 and Sapsard 1994).

Vibration Monitoring

Vibration monitoring technique has been commonly used to determine the engine health status. The technique is able of measuring and trending absolute vibration levels or providing sophisticated real time spectral analysis data (Lynch, 1989).
Vibration in a rotating machinery occurs because of repetitions of motion at some multiple factor of the rotating frequency. They include gear teeth, the whirling of blades, bearings, pumps, turbulence and aerodynamic resonance. In normal operation these motions are transmitted throughout the engine structures as vibrations, and

through the gas path in the form of sound. When parts are worn or damaged their sonic and vibrational signature will change.

In order to apply vibration analysis techniques it is necessary to gather data in the form of vibrational or acoustic signatures. Early vibrational analysis methods used microphones placed near the engine to gather acoustic data, and accelerometers mounted on the engine to measure vibration. The data obtained were analysed using sophisticated pattern recognition techniques and narrow band tracking filters to improve the signal to noise ratio. The complexity of the analysis, however, rendered many vibration monitoring methods impractical.

The effectiveness of vibration and acoustic monitoring as an EHM technique is still limited by the interpretation of the vibration spectrograms obtained from the sonic/vibration data. The number of signals produced at different spool speeds for different combinations of failures is almost infinite. What is actually needed is a system that associates vibrational energies with the resonant frequencies of each engine component. This will help to establish the effect of wear of different components based on the resonant frequencies and the vibrational energies.

Recently, emphasis has been placed on the improvement of transducer instrumentation. Current accelerometers are integral, stainless steel units secured with bolts directly to the main engine hardware. The signal obtained provides an indication of the energy generated by the sum total of all the rotating components irrespective of frequency. By using filters, an increase in energy at any specific frequency will cause an increase in one band only. Mass imbalances brought on by bearing failure, foreign-object damage (FOD), or blade loss can be detected by properly mounted accelerometers. These new instruments have also found application in detecting certain types of gearbox anomalies (LaGrandeur, 1986).

In general, significant information concerning the mechanical condition of rotating equipment can be obtained by measuring the bearing vibrations and shaft relative vibrations. A large number of other locations are possible, however the choice of the measuring locations is important for the success of the monitoring process (LaGrandeur 1986, Sapsard 1994 and Grewal 1988).

Performance Monitoring and Analysis

Performance monitoring and analysis are now the areas of health monitoring which offer the greatest scope for further technical developments. In essence, the monitoring

method includes a consideration of what are the functions which a component or system is required to perform, followed by the application of methods to measure these functions. In some cases there will be two related functions associated with a component or system and a comparison of the relationship between the two can indicate its condition (Neale, 1987). As an example if the applied loads are compared against the resulting deflection on the component structure then this can give an indication whether the component's condition is changing with time. The analysis method involves a thermodynamic analysis of the condition of the engine down to the component level. This method is known as the Gas Path Analysis method. Both, the performance monitoring and the performance analysis are described in more detail in the Chapters three and four.

PART II

METHODS

3 Review of Traditional Gas Path Analysis Techniques

Little work has been published on gas path analysis (GPA) techniques. Scott (1979) introduced a low cost and simple device in order to assess compressor fouling by measuring the changes in the intake depression. Furthermore, Williams (1981) developed a simple mathematical model to analyse faults within the engine. Boyce (1974) presented a theoretical analysis of the gas turbine engine which allowed him to select appropriate dependent parameters. By trending these parameters, Boyce was able to evaluate component/module problems. A similar method was presented by Smetana (1974). As a result of these works, Saravanamuttoo (1974) developed a low cost effective GPA system using the minimum set of instrumentation. His system has been widely published (Saravanamuttoo 1974, 1983, etc.) and the system is based on Cohen's (1986) component-matching technique. With this technique it was possible to calculate mass flow and turbine inlet temperature by solving the flow and work compatibility through the gas generator with few numbers of additional measurements.

Urban (1969) developed a linear differential method where changes in dependent parameters allow to evaluate deteriorated components. The method relies on the implantation of known degraded components in order to generate analytically the fault coefficients. His method is implemented in a computer program Trends (Passalacqua, 1974) which is widely used in airline and marine industries. Urban's GPA technique formed a crucial base for further development and application. His method relies on the implantation of known degraded components in order to generate analytically the fault coefficients. Once the coefficients are developed, then a fault in a component is identified with the changes in measurements (Urban, 1969). A more flexible method of how to obtain the coefficients is presented in Grewal (1988). The method uses the aero-thermo relationship of the engine to develop partial derivatives that accommodate small changes to the engine.

Urban's and Grewal's methods have been widely studied (e.g. Donaghy 1991 and House 1992) and analysed (e.g. Lunderstaedt 1988 and Mathioudakis 1994) and the methods have been incorporated in many diagnostic programs (e.g. Lazalier 1978, Grewal 1988 and Doel 1994). However, both, the studies and the programs have been limited to the use of linear GPA. The limitation of linear GPA is that where the level of degradation that has to be detected is low, the residual error can be of the same order of magnitude as the degradation being sought. Non-linear GPA offers the possibility of

accurately detecting almost any type of degradation (House, 1992). House' non-linear GPA method has only been applied to a simple shaft gas turbine for helicopters.

More recently, GPA programs use statistical algorithms to estimate sensor error (Lunderstaedt 1988 and Doel 1994). This is a desirable feature since gas turbine engines have limited number of gas-path instruments and the obtained data from these instruments are prone to sensor errors. The former algorithms did not estimate sensor error. The measurement error was interpreted as component performance. This often produced bizarre results (e.g. high turbine efficiency coupled with a low compressor efficiency). Urban (1980) and Volponi (1982) showed that weighted least-square techniques could be used to reduce the sensitivity to measurement error (Doel, 1994). However, for the purpose of the present thesis, sensor errors and related problems will not be investigated. This should not distract from its importance, solving problems of instrumentation sensors being a task in itself.

The following sections critically review some of the GPA schemes which have been proposed earlier. In Chapter 4, Urban's GPA method and further developments will be presented.

3.1 PERFORMANCE ANALYSIS

The analytical performance model of a gas turbine engine is based on component characteristics and aero-thermo relationships such as the laws of conservation of energy and mass and special conditions such as choked nozzle and bleed. The calculation then proceeds to "match" all the components by satisfying the aero-thermo relationships. Assuming that the component characteristics are accurately defined, the model can provide the engine performance in terms of dependent measurable parameters such as pressure, temperature, etc. and in terms of independent non-measurable parameters such as efficiency, flow capacity, etc. (see Figure 3.1 and Singh 1989).

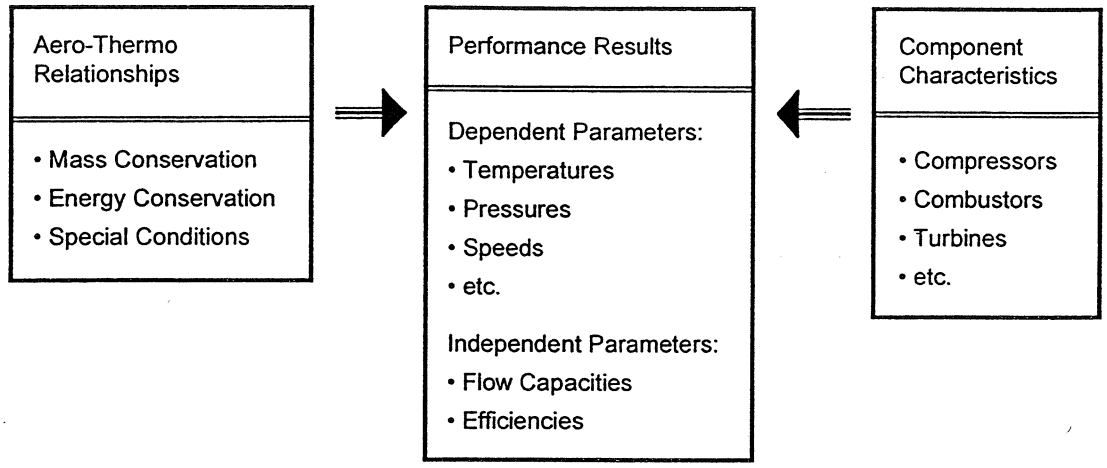


Figure 3.1: Performance Analysis

The aforementioned model is implanted in the Turbomatch program (Palmer 1967) which has been developed at Cranfield University. This is a modular program which allows the calculation of the performance of any open-cycle gas turbine engine under steady-state conditions. The program consists of various pre-programmed routines called 'bricks', which simulate the performance of different components/modules in a gas turbine engine. An engine model is created by assembling the appropriate component bricks such as compressors, turbines, combustors etc. Each brick requires inlet data and the component characteristic to generate outlet data for the component being modelled. Inlet and outlet data are termed 'station vector' data since they correspond to corrected total pressure, total temperature, air massflow and fuel flow at the engine stations. Compressor and turbine components use user-defined or built-in component characteristics in order to calculate the engine performance. Based on aero-thermo relationships the program will provide gas temperatures and pressures to various stations of the engine, individual component performance and the overall engine performance. The program enables to simulate variable or fixed geometry for compressors and/or turbines. However, the option of variable geometry was not investigated in the present thesis.

3.2 PERFORMANCE DETERIORATION

In-service performance deterioration in any mechanical device, such as a gas turbine, is inevitable. During the service of an engine, performance changes can occur because of physical faults such as erosion, fouling, corrosion etc. In order to develop a deterioration model, the faults have to be classified, quantified and identified (Diakunchak, 1992). The identification of faults is described in the following chapter.

Physical Fault Classification

Physical fault classification is based on reported operational problems of gas turbines such as:

- Fouling
- Erosion
- Corrosion
- Foreign Object Damage
- Faults Due to Thermal Distortion
- Faults Due to Engine Operating Conditions and Poor Maintenance Practices

Typically, a quantitative representation of a physical fault can be described as a change in one or more of the independent parameters which describe individual gas path component performances. In general, following independent parameters are used:

- Compressor flow capacity
- Compressor isentropic efficiency
- Turbine nozzle guide vane area
- Turbine isentropic efficiency
- Combustion efficiency
- Exhaust nozzle area

It is noteworthy that independent parameters are determined by the gas turbine engine's configuration and that they are usually not measured. In fact, some of these parameters such as efficiency cannot be measured at all. Once changes in the independent parameters have occurred, then these changes will alter the dependent (measurable) engine performance parameters. Similarly to the independent parameters the set of dependent parameters applicable to each engine depends on its configuration. A typical selection of dependent parameters could be:

- Component inlet and outlet temperatures and pressures
- Shaft power
- Fuel flow

- Spool speeds
- etc.

Finally, the relationship between physical faults and component faults needs to be defined. In the literature, the term 'fault' is widely used to define a change in one of the independent parameters. However, a physical fault is represented or simulated by a component fault set. This set is defined by one or more independent parameter changes. For example, compressor fouling is represented as a drop in flow capacity and as a drop in efficiency (Escher, 1995).

Physical Fault Quantification

Even under normal engine operating conditions, engine gas path components will experience physical faults such as fouling, corrosion, foreign object damage, etc. These physical faults will result in a deterioration in engine performance, which will get progressively worse with increased operating time. However, not all faults cause a noticeable change in the performance therefore the following sections identify the physical faults which do cause a change in the performance. For each physical fault, the parameters that change the performance are explained in the following section (Quantification). The quantification of physical faults allows to simulate faults in a computer program.

Fouling

Fouling can be defined as the 'degradation of flow capacity and efficiency caused by adherence of particulate contaminants to the gas turbine airfoil and annulus surfaces' (Diakunchak, 1992). Although fouling can occur in both compressor and turbine components it has been recognised that compressor fouling is one of the most common causes of engine performance deterioration (Aker, 1989). Gas turbines are particularly susceptible to fouling because of the large quantities of air they ingest. The incoming air consists of hard and soft particles. Hard particles such as dust, dirt, sand, rust, ash and carbon particles and soft particles such as oil, unburned hydrocarbons, soot, air-borne industrial chemicals, fertilisers, herbicides etc. can provide a source for fouling. The fouling problem is worsened if oil leaks. This oil can act as a glue and, especially at the later stages of the compressor, oil may bake on blades because of the higher temperatures. As a result of fouling there is an accumulation of material which changes the shape and inlet angle of the airfoil, increases surface roughness, and reduces the airfoil throat opening (Diakunchak, 1992). In some cases fouling does not cause

damage to the flow path surfaces. In this case, performance deterioration can often be recovered through washing or cleaning. Economic and safety benefits can be realised if fouling can be monitored (quantified) and hence a washing schedule can be determined. This enables the operator of the engine to move from conservative wash periods to on-condition maintenance strategies (Lakshminarasimha, 1994). For example, if washing is performed too often it can lead to unnecessary expense in terms of down-time, increased maintenance costs, and the premature erosion of blade surfaces. However, if the time between the washing periods is too long the performance may not be completely recovered. Typically, compressors are cleaned or washed when there is a 3% decrease in intake depression (Saravanamuttoo, 1985).

Effect of Fouling

Since gas turbines utilise large quantities of air, the incoming air incorporates in a variety of material which can deposit on the component blades resulting in the change of shape of the airfoil and therefore the component has fouled.

In the case of compressor fouling, the change in blade shape causes a reduction in compressor flow capacity and a reduction in compressor isentropic efficiency. The effect of fouling on compressor flow capacity is more significant than the effect on efficiency. Typically, the flow capacity is reduced by 3 - 8% and the efficiency by 1% depending on the severity of fouling (Saravanamuttoo 1985 and Diakunchak 1992). The reduction in mass flow capacity varies with operating speed, ambient temperature and altitudes (Saravanamuttoo 1985). Furthermore, compressor fouling not only reduces the flow capacity and efficiency, but also reduces the compressor surge margin and this may result in compressor surge (Diakunchak, 1992). Previous studies on compressor deterioration suggest that a reduction in flow capacity should be coupled with an equal reduction in compressor pressure ratio (Grewal 1988 and House 1992). As an example of compressor fouling, test data from a fouled large industrial gas turbine indicated that compressor fouling reduced the flow capacity by 5% and reduced efficiency by 1.8%. This amount of fouling resulted in a power loss of 7% and increased the heat rate by 2.5% (Diakunchak, 1992).

In the case of turbine nozzle guide vane fouling, the effective nozzle area (and therefore the turbine flow capacity) is reduced and so is the turbine isentropic efficiency. The effect on turbine nozzle area change is less significant than the flow capacity change in a fouled compressor. Typically the efficiency is reduced by 1% (Diakunchak 1992). No published figure is available for the flow capacity increase but it can be assumed that it will be of the order of 2%.

Erosion

Erosion can be defined as 'the abrasive removal of material from the flow path components by hard particles in the air or gas stream' (Diakunchak, 1992). A typical size of the particles is 20μm or more in diameter. The particulates which cause erosion are hard particulates such as dirt, dust, sand, carbon/soot (the carbon particles are produced as a result of inefficient combustion), ash, salt and industrial pollutants. As a result of erosion, the airfoil surface roughness is increased, inlet metal angle is changed (hence airfoil incidence), airfoil profile is changed, airfoil throat opening is changed, blade tip and seal clearances are increased. In some cases the eroded airfoil trailing edge thickness can be beneficial to performance, though it is unacceptable from mechanical integrity considerations (Diakunchak, 1992). As an example of turbine erosion, experimental data from a large industrial gas turbine showed that erosion resulted in a reduced turbine efficiency of 1%, in a power output loss of 3.7% and in a heat increase of 2.7% (Diakunchak, 1992).

Unlike fouling, erosion causes a permanent damage and a permanent loss of performance which can only be restored though the repair or replacement of components.

Effect of Erosion

The erosion of engine components results in blunting of airfoil leading edges, thinning of trailing edges and increased surface roughness. The eroded blades lose their ability to increase the total pressure of the stage efficiently. Pressure losses occur because of increased tip clearances and profile losses resulting from changes in the boundary layer. Erosion of blade profile and tip clearance increase occur more often in aircraft engines than in industrial gas turbines (Lakshminarasimha, 1994).

In both, the compressor and turbine, erosion causes a reduction in isentropic efficiency due to increased surface roughness, changes in the airfoil profile, and increased tip clearances. Turbine erosion causes an increase in nozzle guide vane area, which can be simulated as an increase in turbine flow capacity. However, compressor erosion results in a reduction in compressor flow capacity even though the inlet area has increased. This is because compressor erosion causes a drop in the engine pressure ratio, which lowers the mass flow capacity for a constant non-dimensional speed. As with compressor fouling, the operating line of an eroded compressor is raised, thus reducing the surge margin. Figure 3.2 illustrates the effect of compressor erosion on the compressor characteristic.

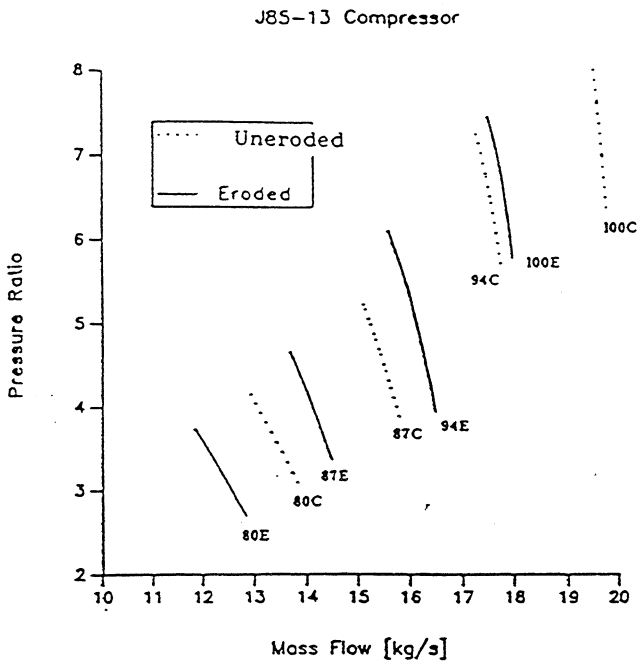


Figure 3.2: Effect of Compressor Erosion on the Compressor Characteristic (Tabakoff, 1990)

For both, compressor and turbine erosion, the effect on flow capacity is greater than the effect on efficiency. Typically, flow capacity for the turbine is increased by 2% (Zhu, 1992) and efficiency is decreased by 1% (Diakunchak, 1992). No figure is available for the compressor flow capacity decrease but it can be assumed that it will be of the order of 2%.

Corrosion

Corrosion can be defined as 'the loss of material form flow path components caused by the chemical reaction between these components and contaminants that enter the gas turbine with the inlet air, fuel, or injected water/steam. Salts, mineral acids, and reactive gases such as chlorine and sulphur oxides, in combination with water, can cause wet corrosion, especially of the compressor airfoils. Elements like sodium, vanadium and lead in metallic or compound form can also cause high temperature corrosion of the turbine airfoils. Hot end surface oxidation is another form of corrosion' (Diakunchak, 1992). Similarly to erosion, corrosion can result in the loss of material and an increase in surface roughness. In addition, corrosion results in a loss of performance and service life of the component effected.

Effect of Corrosion

The effect of corrosion on performance degradation is similar to the effect of erosion. Typically, compressor corrosion results in a reduction in compressor flow capacity and isentropic efficiency, whilst turbine erosion results in an increase in turbine effective area/flow capacity and a reduction in isentropic efficiency.

Foreign Object Damage

Foreign object damage (FOD) can be defined as the cause by which large objects strike the flow path components of the gas turbine engine. These objects enter the engine with the inlet air or are the result of pieces of the engine itself breaking off and being carried downstream. Foreign objects can be things like stones, birds, bolts, tools, etc. Excessive ice forming on the compressor inlet, carbon deposits forming on fuel nozzles, and engine subcomponents can break loose and result in damage to internal downstream components. Foreign object damage can vary from non-recoverable (with washing) engine performance deterioration to catastrophic engine failure (Diakunchak, 1992).

Effect of Foreign Object Damage

The effect of FOD on performance degradation varies significantly with the severity of the damage. FOD results in a large reduction of the component isentropic efficiency and in some cases can change the flow capacity of the damaged component. Typically, isentropic efficiency can be reduced by 5% (Zhu, 1992), However the value is very much dependent on the severity of the damage. The change in flow capacity depends on the type of FOD damage. In some cases the flow capacity may increase in other cases it may decrease. An increase of flow capacity can be the result of lost blades. A decrease of flow capacity can be the result of foreign particles blocked in the gas path. A blockage can be caused by desert sand that has been virtually glued to the turbine blades because of the heat.

Thermal Distortion

Thermal distortion is a fault that normally occurs at combustor exit/turbine entry where temperatures are highest. Distortion is caused by problems such as faulty fuel nozzle spray patterns and warped combustor components which cause changes in the radial and circumferential temperature traverse pattern at the combustor exit. This can result in temporary or permanent deformation of downstream components such as cracked, bowed, warped, burned, lost or damaged turbine nozzle guide vanes, area changes, increased leakage, and relative thermal growth between the static and rotating members (English, 1995). High temperature can cause first stage turbine blades to untwist.

These blades untwist as a result of creep damage during sustained high temperature operation (MacLeod, 1992).

It is noteworthy that although a change in the combustor outlet temperature profile is usually caused by a fault in the combustor, the combustor deterioration is not considered to directly affect the engine performance. Combustion efficiency normally remains constant with time (Diakunchak, 1992). In other words, combustor deterioration does not cause a change in the engine's performance parameters, therefore it cannot be detected by a gas path performance analysis method.

Effect of Thermal Distortion

Bowed, burned, warped, untwisted or damaged blades can cause a reduction in turbine isentropic efficiency due to increased air leakage and reduced airfoil performance. The damage of the blades can also result in changes to the effective flow area. However, the most significant effect will usually be on turbine isentropic efficiency (MacLeod, 1992).

Engine Operation and Maintenance Practices

The way in which the engine is operated will have a significant effect on the type of deterioration experienced and the rate it occurs. Under normal engine operation, the engines will experience wear and tear and consequent performance deterioration. However, engines which undergo a large number of start cycles or which undergo transient operation are more susceptible to performance degradation. This is because of the severe temperature gradients which are experienced during these cycles. Therefore, the hot end components such as combustor, turbine and exhaust diffuser are more susceptible to oxidation, corrosion and thermal distortion. Also, at start-up of the engine, blade tip and seal rubs may occur because of increased vibration, if the rotor is not balanced perfectly and due to transient differential expansion rates between the large mass stationary components and the lighter mass rotor assembly. This will result in increased clearances and in increased leakage and therefore in performance deterioration. Performance deterioration may also occur if the engine is operated outside its specified operating limits.

Poor engineering maintenance practices may have considerable impact on the rate of how the engine performance is deteriorated. The list of potential poor maintenance practices is extensive, however poorly maintained fuel nozzles is just one example. They can cause the temperature traverse pattern to change, causing accelerated turbine damage (Diakunchak 1992).

Component Deterioration

Deteriorated components can be simulated by modifying the appropriate component map characteristics. Since the maps, stored in Turbomatch, relate to a particular size of a component, they need to be scaled for the clean and for the deteriorated performance. In a clean performance simulation a chosen design point establishes the scaling factors for the maps i.e. the design point values such as efficiency and flow capacity correspond with the scaled maps. These factors are stored in Turbomatch and they are applied to all subsequent off-design point calculations. However, in a deteriorated performance simulation the stored scaling factors are adjusted by the appropriate change of the independent parameters. Examples of a fouled compressor map and an eroded turbine map can be seen in Figures 3.3 and 3.4. A detailed analysis of how component characteristics are scaled in Turbomatch is shown in Appendix A.

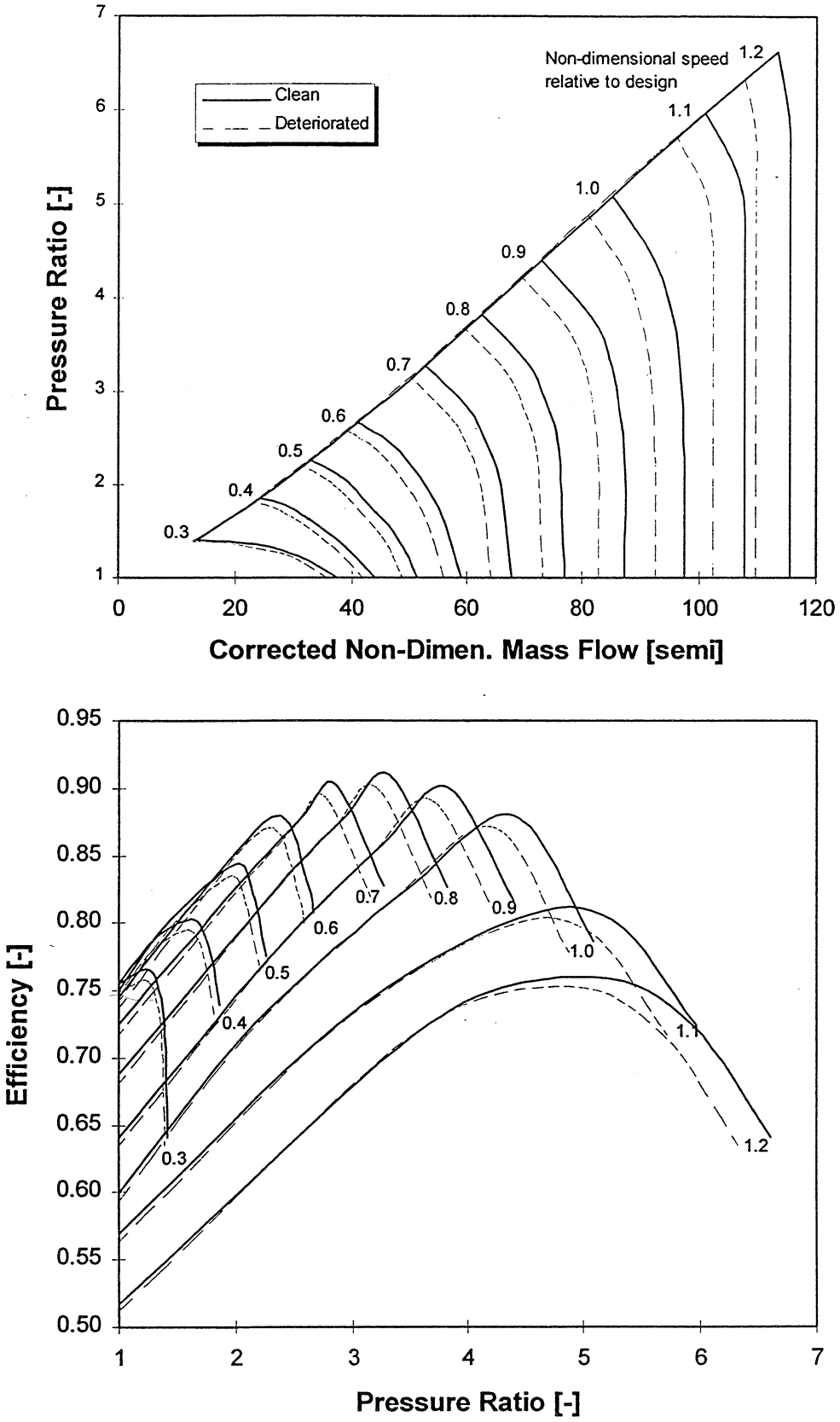


Figure 3.3: Compressor Fouling

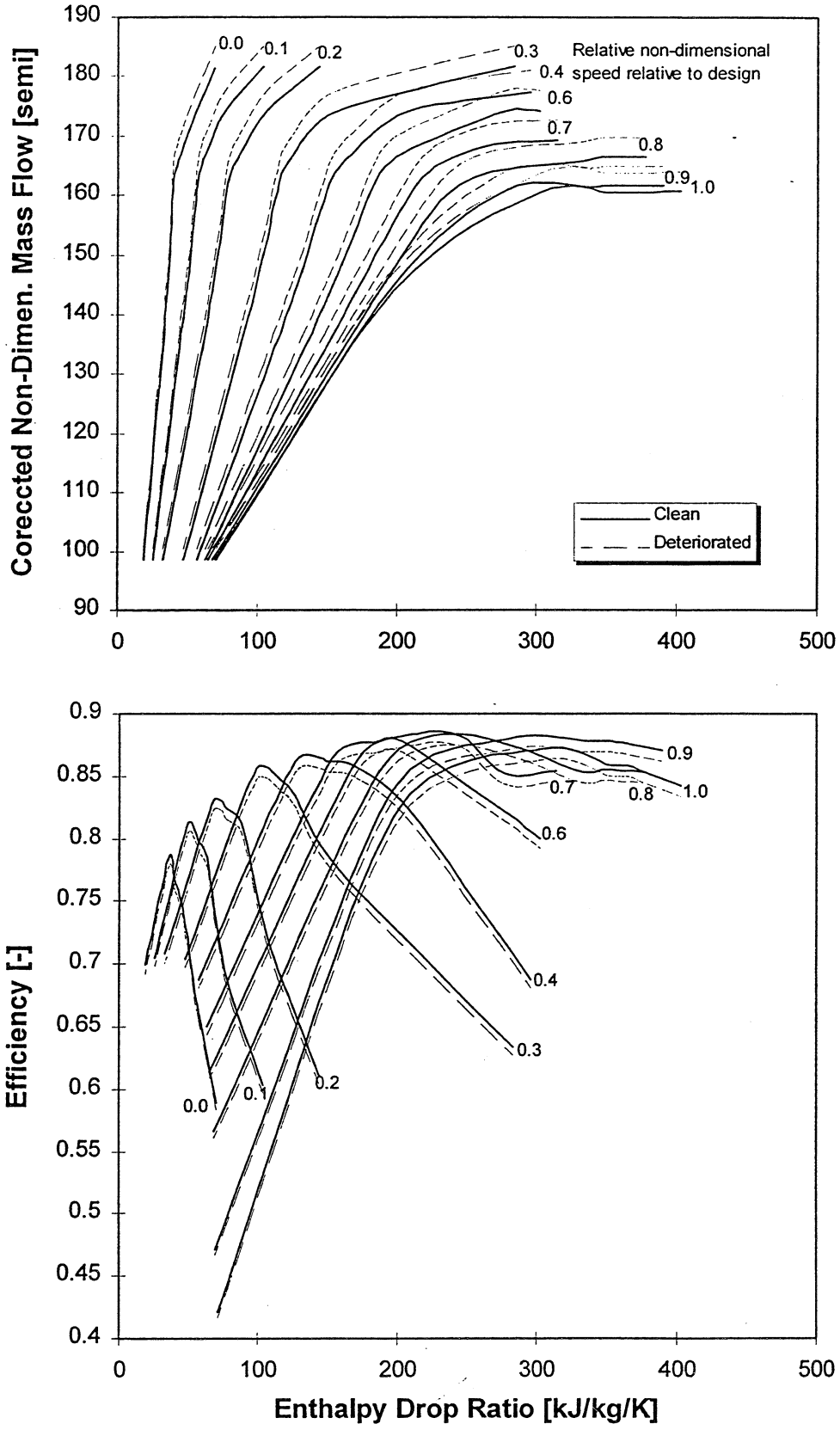


Figure 3.4: Turbine Erosion

Summary

Many operational problems can be diagnosed and resolved by correlating representative performance parameters. Performance degradation effects, caused by independent parameter changes, can be used in conjunction with condition monitoring to facilitate detection of faults and to aid understanding of operational problems such as compressor fouling and erosion (see Table 3.1).

Table 3.1: Physical Faults Expressed as Independent Parameter Changes

Physical Fault	*Non-dimensional mass-flow change A*	*Isentropic efficiency change B*	*Ratio A : B*
Compressor Fouling	$\Gamma_C \downarrow$	$\eta_C \downarrow$	~3-8 : 1
Turbine nozzle guide vane fouling	$\Gamma_T \downarrow$	$\eta_T \downarrow$	~2 : 1
Compressor erosion	$\Gamma_C \downarrow$	$\eta_C \downarrow$	~2 : 1
Turbine erosion	$\Gamma_T \uparrow$	$\eta_T \downarrow$	~2 : 1
Compressor Corrosion	$\Gamma_C \downarrow$	$\eta_C \downarrow$	~2 : 1
Turbine Corrosion	$\Gamma_T \uparrow$	$\eta_T \downarrow$	~2 : 1
Foreign object damage (non severe!)	$(\Gamma_M \updownarrow)$	$\eta_M \downarrow$	0.5 : 1
Thermal distortion	$(\Gamma_M \uparrow)$	$\eta_M \downarrow$	0.5 : 1
Blade rubbing (English, 1995)	$(\Gamma_M \uparrow)$	$\eta_M \downarrow$	0.5 : 1

Key to Table 3.1: C denotes compressor component/module, T denotes turbine module and M denotes compressor and or turbine module. However the list of physical faults in Table 3.1 is not complete and there are many other sources of deterioration which need to be addressed such as excessive drop in inlet filter differential pressure, excessive back pressure, increased mechanical losses (gearboxes, bearings, couplings etc.), internal losses etc. (Meher-Homji, 1993).

3.3 PERFORMANCE TREND MONITORING AND ANALYSIS

If dependent parameters are recorded and monitored as a function of time then performance trending allows the operator of the engine to keep track of the observed readings (Passalacqua, 1974). The performance monitoring is capable of determining delayed time dependent and time dependent failures but will not detect instantaneous failures (Saravanamuttoo, 1983). Since every new installed engine performs differently to an engine in a test cell, a baseline has to be established for each new engine in its installed position with the instrumentation that will normally be used for monitoring the engine.

Even when the performance of the degraded engine is compared with the established baseline, the measured change will not only result from component degradation but also from instrument non-repeatability. The instrument non-repeatability can be of the same order of magnitude as the component degradation being sought. Therefore the establishment of a baseline has to be made with considerable care. Any deviation from the baseline curve will determine the deterioration of the engine and hence the fault.

Critical Cycle Parameters

Critical cycle parameters are those which determine the health of a gas turbine engine. Unfortunately, they are generally not measured because of the difficulty of sensors position, accuracy, reliability in harsh environment and the associated costs (Connolly, 1985). Moreover some of the measurements can considerably disturb the gas path and therefore a loss in power output can occur. In particular, turbine entry temperature, massflow, efficiency are some of the non-measurable parameters. Turbine entry temperature measurement is possible but not practical even though the technique of optical pyrometry is well developed (Connolly, 1985). Mass flow measurement requires a bell-mouth intake installation on every engine. Therefore a thermodynamic analysis of the engine should allow the calculation of critical parameters based on a minimum set of monitored parameters (Staples, 1974).

Parameter Selection

A simple inferential diagnostic analysis (Boyce, 1975) can be used as an indicator of engine health. The analysis involves the careful selection of appropriate measurements for a particular problem. For example, compressor problem is best diagnosed by monitoring temperature, pressure and shaft vibration parameters. These measured parameters will in conjunction with calculated non-measurable parameters provide the information to investigate the following compressor problems:

Inspection	A compressor needs to be inspected when a predetermined level of decreasing compressor efficiency is reached
Surge	Surge can be detected by trending shaft vibration but only in the case where a critical level is reached.
Fouling	Fouling can be detected by exit pressure and temperature and shaft vibration. Indication of fouling is given by a decreasing trend of

pressure versus time and an increasing trend of the exit temperature. In addition, an increased level of vibration should be monitored.

Bearing Bearing problems in compressor can be detected by an increased difference between inlet and exit temperature in the lubrication system and by an increased trend in shaft vibrations.

Damaged Blades Damaged compressor blades can be detected by a sudden increase in shaft vibration and exit temperature.

Similarly to the compressor, the turbine problems can be investigated. It has to be pointed out that the inlet temperature and pressure of the turbine are estimated by the combination of the heat and work balance equation and exhaust temperature. This is because inlet temperature and pressure of a turbine are difficult to measure. The following turbine problems can be investigated:

Fouling Turbine fouling can be detected by turbine inlet and exit temperatures and pressures, and shaft vibration. An indication of fouling is given by an increased trend of calculated inlet temperature, exhaust temperature or fuel flow for the same power output. An increased trend of shaft vibration is another indicator for turbine fouling.

Inspection A turbine needs to be inspected when efficiency is decreased with time.

Damaged Blade Damaged turbine blades can be detected by a sudden increase in shaft vibration and exhaust temperature.

Bowed Nozzle Guide Vane Bowed Vanes can be detected by an increased trend in exhaust temperature at constant turbine inlet temperature setting. A decreased trend of output power at constant fuel flow indicates a change in angle of incidence of turbine blades.

Cooling Air Cooling air needs to be inspected when there is a pressure drop in the cooling air line (A drop will indicate a reduced cooling flow rate).

And finally, combustor problems can be investigated. An uneven circumferential distribution of temperature in the turbine exhaust will be indicated by a thermocouple reading higher than the others. By knowing the location of the thermocouple, the phase shift of the gas path can be evaluated from the turbine diameters, number of stages,

blades dimensions and gas velocity to calculate the position of the abnormal condition in the combustor.

Based on basic aero-thermo relationships and appropriate measurements, some basic diagnostics can be made. Typically, this information is stored in a fault matrix as shown in Table 3.2.

Table 3.2: Simple Turbine Diagnostics Matrix

Type	η	P_3/P_4	T_3/T_4	*Vibration*	*Bearing temp.*	*Cooling air press.*	*Wheel space temp.*	*Bearing press.*
Fouling	↓	-	↓	↑	-	-	↑	-
Damaged blade	↓	-	↓	↑	-	-	-	-
Bowed blade	↓	↓	↓	↑	-	-	↑	-
Bearing failure	-	-	-	↑	↑	-	-	↓
Cooling Air Failure	-	-	-	-	↑	↓	↑	-

Trend Analysis

Trend analysis is a technique that allows the diagnostics of the engine health by means of observation of measurement readings. The condition of the engine is routinely monitored and a change in performance caused by deterioration will change the readings. When using a baseline that test results can be compared with, monitoring change in performance from the baseline condition will aid the experienced user in trouble shooting faults. In addition, the readings ensure that the gas turbine is able to meet the expected power requirements. Trend analysis is a useful technique but somewhat limited because of the limited number of sensors. It detects deterioration but will not identify the affected component/module of a gas turbine. Furthermore, readings are often taken at different power settings resulting in considerable data scattering. Other areas of concern are the baseline type, data repeatability and fault determination.

Baseline Types

The establishment of a measured baseline is crucial for the success of health monitoring systems. Typically, two types of baselines are possible: generic or customised. A generic baseline is provided by the engine manufacturer or developed by the operator.

The baseline is a standard level of performance for all engines of a particular model installation. Since all engines perform in a slightly different manner due to build tolerances, the installed engine will have a certain offset from the generic baseline. Assuming the engine is well within operation limitations this should not cause a problem.

A customised baseline tries to establish a baseline for each engine installed. This requires a computer data base for each engine to manage the information. A drawback of this method is that once maintenance is conducted on the engine, a new baseline has to be established resulting in loss of previous trending information (House, 1992).

Data Repeatability

Trend analysis requires changes in performance from a baseline to actual engine performance. Therefore, monitored parameters should have good data repeatability. The coefficient, that defines good data repeatability is explained in Urban (1983). Since problems of sensors are not considered in the present thesis, a detailed analysis of the data repeatability coefficient is omitted. Good repeatability ensures that smooth performance trends are obtained. Rapid changes in performance trends can signal performance degradation but more often they will cause confusion since performance trends become obscure (House, 1992). By changing the engine testing environment without properly accounting for this change, the performance data will become highly non-repeatable. Figure 3.5 illustrates the effect of data non-repeatability caused by different installation, bleed and accessory loads, instrument calibration and instrument non-repeatability.

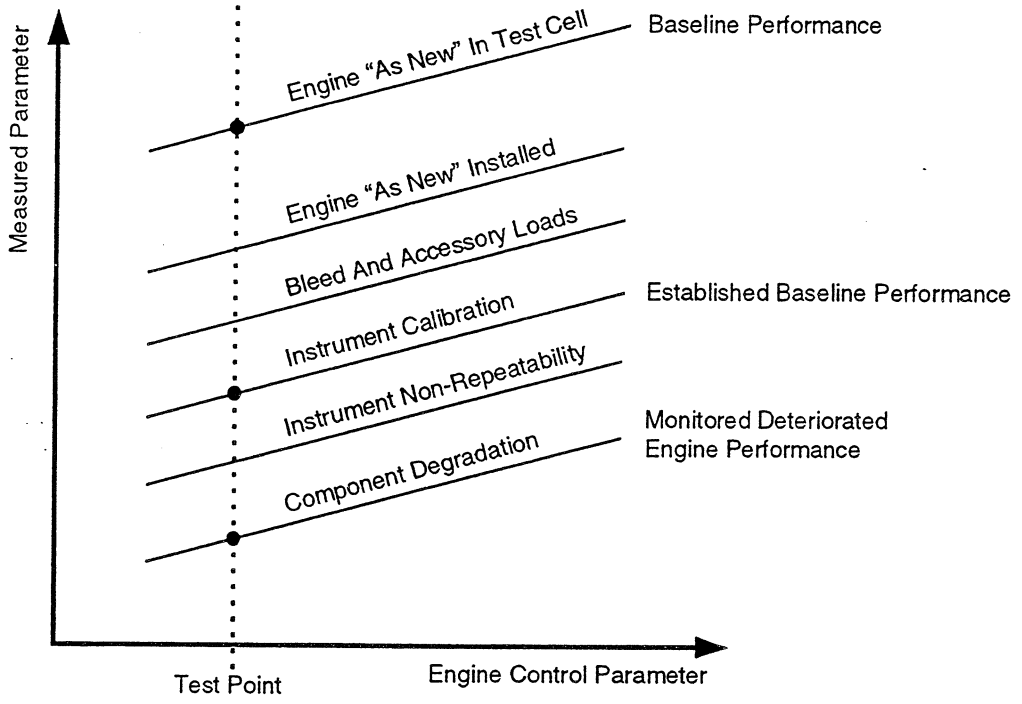


Figure 3.5: Data Non-Repeatability Effects

Whereas Figure 3.5 explains the reasons for measurement deviation of a particular instrumentation set, Figure 3.6 describes the baseline change of a fouled compressor versus service hours (Diakunchak, 1992).

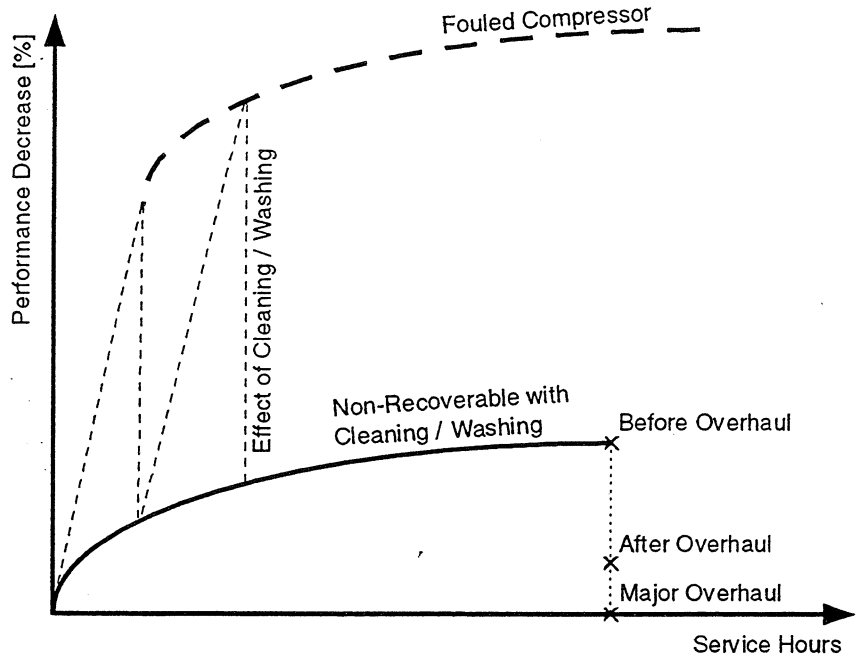


Figure 3.6: Typical Performance Deterioration of a Compressor

Different installation of engines will cause changes in performance. Typically, installation losses are higher in an aircraft than in an engine test cell. Airframe installation losses can relate to aerodynamic design of intake and exhaust ducts. Therefore, each installation type requires the establishment of an appropriate baseline performance.

Bleed and accessory loads will cause the engine to rematch at a different operating point. Bleed extraction will increase turbine entry temperature and it will move the compressor running line away from surge at a constant power setting. Changing loads from one performance test to another will cause data scatter since the performance baseline has changed.

Instrument errors are another source that affects data-repeatability. Pressure and temperature instrumentation are susceptible to calibration errors, recovery losses, electromagnetic radiation interference and drift caused by changes in ambient conditions.

Fault Determination

Once a deviation from a baseline performance to a deteriorated performance has been identified, then the next step is to diagnose the fault. The choice of fault determination can be undertaken by several methods such as the fault tree, the fault matrix method

developed by Saravanamuttoo (1974) or the fault matrix method developed by Urban (1969).

Fault Tree

The fault tree is a method that identifies the fault by working through a fault tree from a vague approximation of the fault to a better defined fault identification. Each step through the tree represents a decision which further refines the fault identification and establishes the path to the next level in the fault tree (see Figure 3.7 and House 1992).

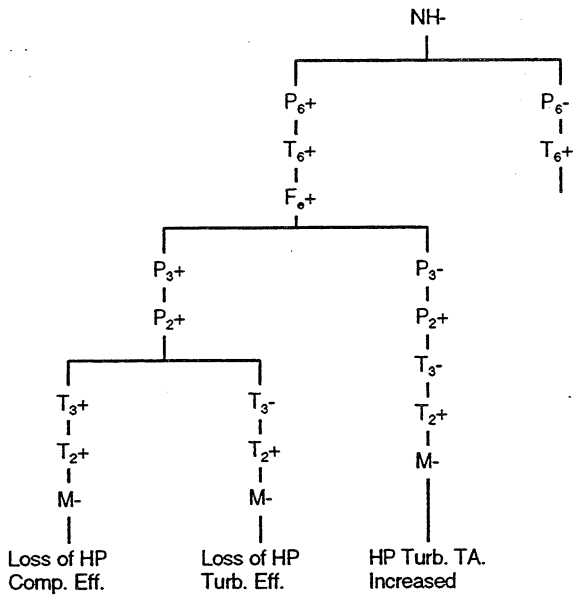


Figure 3.7: Example of a Fault Tree

Fault Matrix Method

The fault matrix method is a GPA method that can identify a single fault in an engine. The method was developed by Saravanamuttoo (1974) and it requires a computer component matching simulation of engine performance using thermodynamic equations and component characteristics. A fault is simulated by altering flow capacity and efficiency of the desired component characteristic and rematching the engine model at its new operating point. Once the clean and deteriorated engines states are defined, the change in engine speed, power output/thrust, temperature, pressure and fuel flow allows the creation of a fault matrix. The fault matrix then can be used to compare trends of actual measured parameters to the matrix hence an engine fault can be identified (see Figure 3.8).

Type	Fault	TIT	SHP	m_f	CPR	Vibration	Indication
Turbine (Generator)	Rotor Damage	↑	↑	↑	↑	Yes	η_t Low
	Nozzle Erosion	↑	↑	↑	↓	No	$m\sqrt{T_3}/P_3$ High
Turbine (Power)	Rotor Damage	0	↓	0	0	Yes	η_t Low, EGT High
	Nozzle Erosion	↓	↓	↓	↓	No	$m\sqrt{T_4}/P_4$ High
Compressor	F.O.D.	↑	↓	↑	↓	Yes	η_c Low, m_1 Low
	Dirty	↓	↓	↓	↓	No	η_c Low

Figure 3.8: Example of Engine Fault Matrix for a Single Spool Gas Generator with a Free Power Turbine (Saravanamuttoo, 1974)

Key to Figure 3.8: TIT denotes turbine entry temperature, SHP denotes shaft horse power, CPR denotes compressor pressure ratio, η denotes efficiency, m denotes massflow, T denotes temperature, P denotes pressure and EGT denotes exhaust gas temperature.

Summary

The aforementioned methods have a number of similarities since each of them has been developed from basic performance theory. They all appear to be ideally suited for low cost applications. Both techniques, the fault tree and the fault matrix method, share the limitation that only one fault can be detected within the engine and they are unable to offer a solution if two or more faults occur simultaneously within the engine. In addition, these techniques are unable to identify the magnitude of the degradation and therefore they only give qualitative rather than quantitative information. However, the accuracy of the method is greatly affected by data non-repeatabilities and the accuracy of sensor measurements. In Urban's method the accuracy of instruments is virtually levelled out because the method uses measurement differences instead of absolute values.

The limitation with Saravanamuttoo's fault matrix method is that different engine and sensor faults are often characterised by identical changes in the matrix. A possible solution would be to make the matrix more complex, but this requires a detailed understanding of failure modes and its effects for that particular engine. In addition, the method can only identify single faults. The method cannot detect multiple faults. Furthermore, for each control setting of the engine a different fault matrix is required. Therefore, in order to cover several of the possible fault types with different magnitude combinations, a set of fault matrices or a fault matrix library has to be established. Some

progress in detecting multiple faults without large libraries has been proposed by Dupuis (1986). Dupuis identified multiple faults by superposition of single faults. Predicting unrelated multiple faults was successful, achieved by adding the changes in performance with single faults. However, superposition applies only to linear systems and therefore the proposed method is unsuitable for multiple fault identification.

Another GPA method that can identify faults has been developed by Urban (1969). Unlike Saravanamuttoo's fault matrix method which provides qualitative results, Urban's method produces quantitative results which can be used for diagnostics and which can be used as a prognostic tool. The principle of Urban's GPA method are presented in Chapter four.

4 Gas Path Analysis Principles

Gas Path Analysis (GPA) is a quantitative technique for analysing the deteriorated performance of a gas turbine engine. Performance analysis typically involves the simulation and monitoring of clean (baseline) and deteriorated engine performance. Once the performance data are monitored then the degradation of an engine can be identified. So far, techniques of identifying degradation produce qualitative rather than quantitative results. However, GPA can detect degradation with greater certainty using an improved non-linear multiple fault scheme.

4.1 LINEAR GAS PATH ANALYSIS

Unlike the Fault Matrix method which provides qualitative results, the GPA method produces quantitative results which can be used for diagnostics and which can be used as a prognostic tool (Urban, 1972).

As stated earlier, during the operation of a gas turbine engine, the gas path may suffer several degradations such as erosion, fouling etc. (see Figure 4.1).

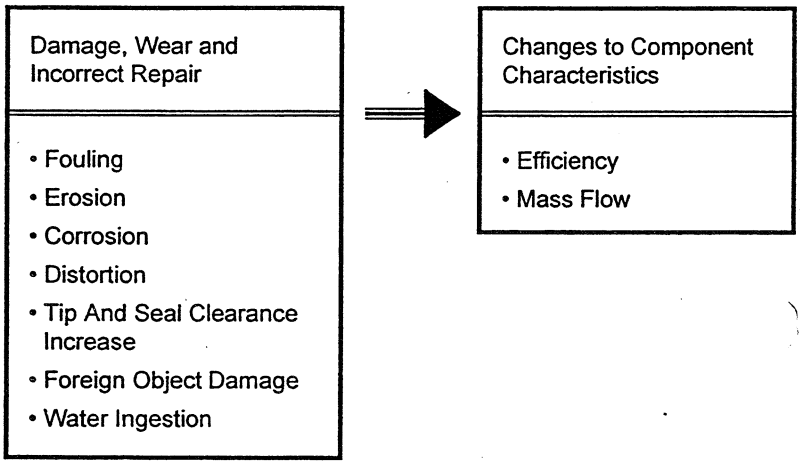


Figure 4.1: Performance Degradation

Each of these types of degradations potentially affects the gas path and therefore may alter flow capacity, component efficiency, etc. (i.e. change of independent parameters). Due to changes in independent parameters the engine will seek to operate at a new operating point which can be detected by shifts of measurable parameters (i.e. dependent parameters). If direct measurements of dependent gas path parameters are

related to independent parameters of the components then the aero-thermo relationship can isolate the degraded component (see Figure 4.2).

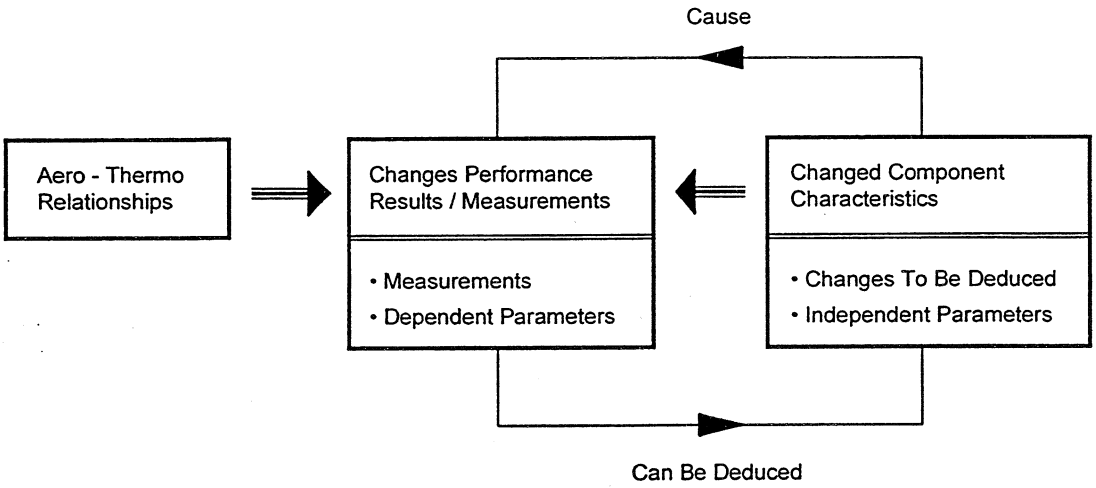


Figure 4.2: Performance Diagnostics

Assuming that changes in the independent parameters are relatively small the new set of equations can be linearized by a Taylor Series expansion. These linearized equations are expressed in matrix form. The matrix is usually given the name influence coefficient matrix (ICM).

Evaluation of Influence Coefficient Matrix

The influence coefficient matrix is derived by taking partial derivatives of the linearized thermodynamic expressions for a particular engine configuration. For that purpose Urban (1969) developed a generic working tool to allow rapid first-order approximation solutions to most performance questions about general cycles or specific engines. However when greater precision is required, more rigorous matrices need to be used, which are derived identically as explained in the previous section but extended to take into account second-order effects (Urban, 1983). A typical matrix for a single spool free turbine is shown in Figure 4.3.

	ΔT_{5C1}	ΔN_{1C1}	W_{aB1}	$\Delta\Gamma_1$	$\Delta\eta_C$	ΔA_5	$\Delta\eta_t$	$\Delta\eta_{PT}$
ΔT_{5C1}	1	0	0	0	0	0	0	0
ΔN_{1C1}	0	1	0	0	0	0	0	0
$\Delta\Gamma_1$ Inlet	-0.04	3.24	-0.10	0.90	0.01	0.09	0	0
ΔT_{3C1}	0.13	1.09	0.29	0.29	-0.57	-0.26	0	0
$\Delta P_3/P_1$	0.41	3.30	0.92	0.92	-0.03	-0.82	0	0
ΔW_{fC1}	2.01	2.10	0.60	0.60	0.60	0.36	0	0
ΔT_{7C1}	1.32	-0.72	0.18	-0.19	0.38	0.17	0	0
$\Delta P_7/P_1$	2.11	-0.26	1.79	-0.02	1.84	-0.08	1.81	0
ΔA_7	-1.53	3.15	-0.82	0.83	-1.67	0.26	-1.84	0
ΔSHP_{C1}	3.33	2.10	2.67	0.64	2.06	0.21	1.60	1.00
ΔT_{9C1}	0.91	-0.65	-0.16	-0.18	0.02	0.19	-0.36	-0.23

(All Δ's are in % of point values)

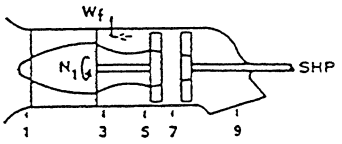


Figure 4.3: Influence Coefficient Matrix of a Single Spool Free Turbine Engine (Urban, 1983)

Each row of the matrix in the Figure 4.3 depicts a differential equation where the net change in the dependent parameter specified at the left of the row is the arithmetic sum of the coefficients multiplied by the change in the parameters specified at the head of each column (independent parameter). Each of the coefficients in any row accounts for thermodynamic expressions such as conservation of mass and energy, power balance between driving and driven members, specific shapes of component performance maps, variable specific heat effects, nozzle flow coefficient and unchoking effects etc. (Urban, 1983). The derivation of these coefficients is time consuming and prone to error due to interpretation of data from component maps. As a result of this, Grewal (1988) developed a method that obtains the coefficients from an off-design engine simulation program. The simulation involves changing each independent parameter variable by a perturbation magnitude and recording the change in the dependent parameters. Using a generic engine modelling program, Grewal could generate the ICM's for different engine configurations. The program has considerably simplified the process of the ICM creation and also the accuracy of fault diagnosis with GPA has been improved (House, 1992).

The ICM is then inverted to form the fault coefficient matrix (FCM). By multiplying measured changes in the dependent parameters by the FCM, the change in independent parameters can be found, hence the cause of engine deterioration. But if this method is used to identify multiple faults linear GPA becomes less reliable because the rates of degradation are seldom known and they are not likely to be linear (Saravanamuttoo 1983

and Grewal 1988). A further limitation is that linear GPA is, by definition, only suitable for small changes or degradations.

4.2 NON-LINEAR GAS PATH ANALYSIS

Gas Path Analysis, which has the ability to monitor multiple degradations, is clearly a very powerful tool for studying the health of a gas turbine. However, the linear GPA models that are available have the severe limitation that in many circumstances the level of error introduced by the assumption of the linear model can be of the same order of magnitude as the fault being sought. It has therefore been recognised that there is a powerful case for improving the accuracy of GPA systems. One way of improving the accuracy is to try to solve the non-linear relationship between dependent and independent parameters with an iterative method such as the Newton-Raphson method. In the present thesis, instrument problems and sources of data noise are not considered in the set of equations. In the following, the theory of non-linear GPA will be presented first. Then, additional information such as assumptions, requirements and economical aspects will be discussed.

Theory

Once changes in the dependent parameters have been detected then the changes in independent parameters can be mathematically identified by using aero-thermo relationships. The relationships are typically represented in matrix notation since each independent parameter can be expressed individually as a unique function of the dependent parameters. In other words, the matrix is a set of functions that describes how changes in component performance influence the dependent parameters. In mathematical terms a typical problem gives n functional relations F_i, i=1,2,...,n involving m independent parameters x_j, j=1,2,...,m and n dependent parameters y_i, i=1,2,...,n:

$$F(\underline{x}) = \underline{y} \tag{4.1}$$

where $\underline{x}$ denotes the entire vector of values x_i, $\underline{y}$ denotes the entire vector of values y_i and F denotes the entire vector of function F_i. In the region of $\underline{x}$, we assume for small changes $\delta\underline{x}$ that

$$\begin{aligned} F(\underline{x}+\delta\underline{x}) &= \underline{y}+\delta\underline{y} \qquad \text{or} \\ F(\underline{x}+\delta\underline{x}) &= F(\underline{x})+\delta\underline{y} \end{aligned} \tag{4.2}$$

For a single variable function g(a), the Taylor Series expansion about the variable a by a small change b would be:

$$g(a+b)=g(a)+b\cdot g'(a)+\frac{b^2}{2!}\cdot g''(a)+\frac{b^3}{3!}\cdot g'''(a)+\ldots \tag{4.3}$$

Therefore, Equation (4.2) can be rewritten as:

$$F(\underline{x}+\delta\underline{x})=F(\underline{x})+J\cdot\delta\underline{x}+HOT \tag{4.4}$$

where Jacobean matrix notation has been used to define the first derivative in the Taylor Series expansion of the matrix function $F(\underline{x}+\delta\underline{x})$ so that:

$$J=\begin{pmatrix} \frac{\partial f_1(\underline{x})}{\partial x_1} & \frac{\partial f_1(\underline{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\underline{x})}{\partial x_m} \\ \frac{\partial f_2(\underline{x})}{\partial x_1} & \frac{\partial f_2(\underline{x})}{\partial x_2} & & \frac{\partial f_2(\underline{x})}{\partial x_m} \\ \vdots & & & \\ \frac{\partial f_n(\underline{x})}{\partial x_m} & \frac{\partial f_n(\underline{x})}{\partial x_m} & & \frac{\partial f_n(\underline{x})}{\partial x_m} \end{pmatrix} \quad \text{and} \quad \delta\underline{x}=\begin{pmatrix} \delta x_1 \\ \delta x_2 \\ \vdots \\ \delta x_m \end{pmatrix}$$

$$\text{Given that } \underline{y}=\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \quad \text{where} \quad \begin{matrix} y_1=f_1(\underline{x}) \\ y_2=f_2(\underline{x}) \\ \vdots \\ y_n=f_n(\underline{x}) \end{matrix} \quad \text{and} \quad \underline{x}=\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

If the changes in $\underline{x}$ are small, we can neglect terms of order $\delta\underline{x}^2$ and higher (HOT). Therefore, Equation (4.4) can be rewritten as:

$$F(\underline{x}+\delta\underline{x})=F(\underline{x})+J\cdot\delta\underline{x} \tag{4.5}$$

Rearranging Equations (4.2) and (4.5) yields:

$$\delta\underline{y}=F(\underline{x}+\delta\underline{x})-F(\underline{x})=J\cdot\delta\underline{x} \tag{4.6}$$

Matrix Equation (4.6) can be solved by inverting J as described by Donaghy (1991):

$$J^{-1} \cdot \delta \underline{y} = J^{-1} \cdot J \cdot \delta \underline{x} \qquad \text{or}$$
$$J^{-1} \cdot \delta \underline{y} = \delta \underline{x} \tag{4.7}$$

The corrections $\delta \underline{x}$ are then added to the solution vector,

$$\underline{x}_{New} = \underline{x}_{Old} + \delta \underline{x} \tag{4.8}$$

and the process is iterated to convergence as illustrated in Figure 4.4. This iterative process has the advantage to overcome the problem that changes in $\delta \underline{x}$ have to be small. In other words, the process seeks to solve numerically the non-linear set of equation that is defined in Equation (4.1). Typically, the Jacobean matrix J is referred to as the Influence Coefficient Matrix (ICM) and the inverse of the ICM J^{-1} as the Fault Coefficient Matrix (FCM). A first order approximation to the IC matrix is created by perturbing in turn each independent parameter by the finite difference δx_j

$$\delta x_j = \frac{x_j}{100} \tag{4.9}$$

in equation (4.6). The change in the resulting dependent parameters δy_i is then determined from the aero-thermo relationships. The determination of the coefficients can in graphical terms be described as the slope of the functions that relates dependent to independent parameters (see Equation (4.1) and Figure 4.4). In order to avoid the risk of numerical problems when inverting the ICM, parameters are normalised using the following equation:

$$\Delta = \frac{\text{Observed Value - Baseline Value}}{\text{Baseline Value}} * 100\% \tag{4.10}$$

The implication of the aforementioned method is that linear GPA (Urban 1969, Grewal 1988) is used successively to obtain improved solutions via the Newton-Raphson technique until the exact solution is obtained. For each linear GPA calculation an appropriate baseline is required. In the first iteration the actual measured baseline is used. The second iteration uses a calculated baseline that is derived by implanting faults that are detected in the first iteration. The same applies to the third and following iterations where the implanted faults are taken from the previous iteration. The iterative process is terminated by the convergence criteria $\Delta \underline{y}_{sum}$, which is defined as:

$$\Delta \underline{y}_{sum} = \sum_{j=1}^{n} (\Delta \underline{ymeas}_j - \Delta \underline{yobs}_j) \tag{4.11}$$

where n is the number of dependent parameters, $\underline{y}_{meas}$ the actual measured deteriorated dependent parameter and $\underline{y}_{obs}$ the calculated deteriorated dependent parameter that is based on the detected independent parameter $\underline{x}$.

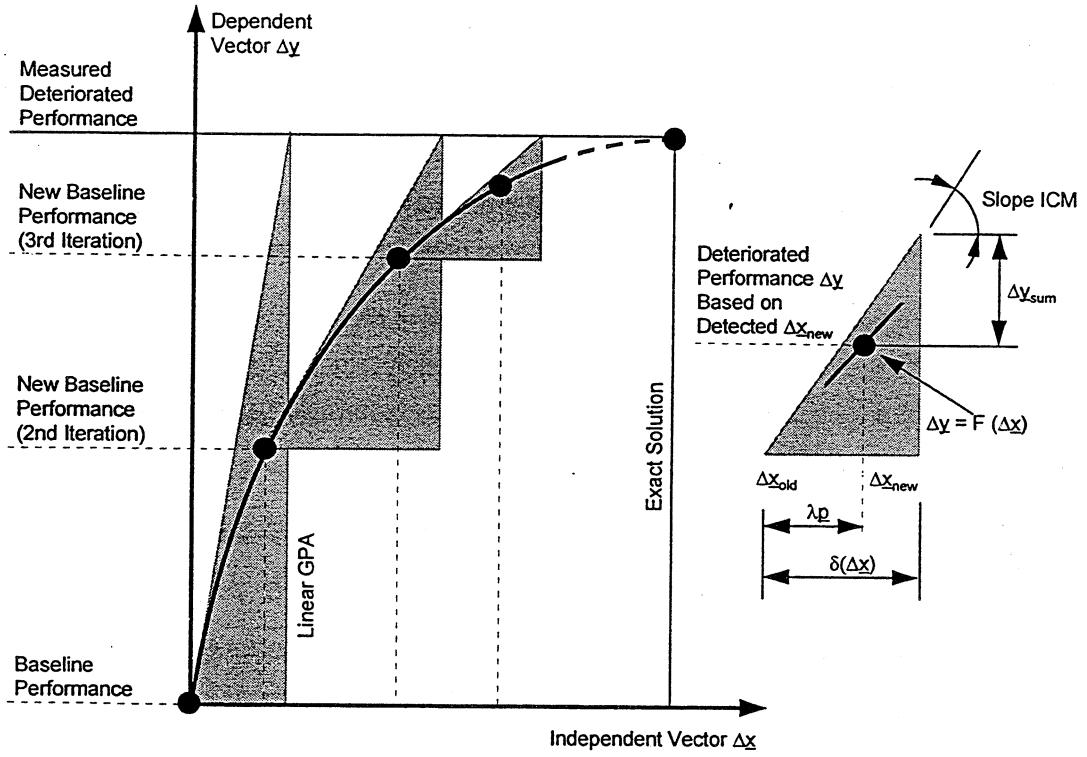


Figure 4.4: Simplified Illustration of Non-Linear Gas Path Analysis Method

It might occur that by taking the full step $\underline{p} = \delta(\Delta \underline{x})$ in the iterative process, the solving process has the unfortunate tendency to digress. One solution could be to move the new point $\Delta \underline{x}_{new}$ along the direction of the step $\underline{p}$, but not necessarily all the way:

$$\Delta \underline{x}_{new} = \Delta \underline{x}_{old} + \lambda \, \underline{p} \tag{4.12}$$

where λ is $0<\lambda\leq 1$. A typical value for λ is 0.66 (Read, 1993). However, a more sophisticated solution for λ is presented in Press (1994). The proposed strategy combines the rapid convergence of Newton's method with a convergent strategy that will guarantee some progress towards the solution of each iteration. A main feature of the strategy is that it always tries first the full Newton step (= linear GPA) and

'backtracks' along the Newton direction until an acceptable step is achieved. However, the aim of the present thesis is to investigate whether the non-linear GPA technique shows significant improvement when compared to the linear GPA technique. Therefore, the aforementioned strategy that combines the linear and the non-linear GPA technique will not be considered.

Assumptions

The presented GPA techniques such as the method of Saravanamuttoo (1974), Urban (1969), Grewal (1988) and House (1992) are all based on some or all of the following assumptions (Smetana, 1974):

One-dimensional The flow of air or air-combustion product is essentially one-dimensional, and therefore the pressures and temperatures are considered identical at every point in a plane normal to the gas path.

Steady-state The gas flow is to be considered steady-state and it is therefore possible to perform equilibrium thermodynamic analysis throughout the engine. The analysis requires accurate measurements of the gas state conditions.

Adiabatic The gas flow through the engine may be considered to be adiabatic. Cooling losses are neglected.

Specific Heat The ratio of specific heats of the gas before and after each of the major engine components is constant and previously known.

Thermal Efficiency By comparing measured changes in enthalpy across major engine components or even across the entire engine with those expected for the same air mass flow rate and/or fuel flow rate it is possible to compute thermal efficiencies of these components or the engine as a whole.

Exhaust Velocity The measurement of the exhaust gas velocity and mass flow can be used to determine engine thrust

Work Done The components of the engine operate in an inter-dependent fashion and hence the work done by the compressor, the accessory drive and the frictional work absorbed by the bearings cannot exceed the work done by the turbine.

Transducer	Computing the same quantity from different sets of measurements is an excellent means of checking the validity of the transducer indications.
Sensor	Sensor indications can be combined to form parameter values which are characteristic of engine performance.
Healthy engine	Healthy engines have a particular set of these parameter values for each operating condition. Each fault causes at least some of the parameter values in a set to change.

In addition, it was assumed that performance measurements are free of noise. Noise is scatter in data caused by data non-repeatability. Noise can hide engine faults and it can also introduce error into the analysis. Noise can be minimised by recognising and eliminating its sources as previously discussed. Urban (1983) expanded his model to include apparent sensor problems or sources of data noise. However, the problems of sensors are not investigated in the present thesis because it is beyond the scope of this thesis.

Dependent and Independent Parameter Selection

A GPA study requires the selection of dependent and independent parameters. During service not all independent parameters undergo changes. For this reason some of the independent parameters can be excluded from the differential set of equation (see Equation 4.1) since no benefit is gained by trying to identify faults that seldom or never occur. Similarly, every dependent parameter that is not monitored is excluded from the sets of equations. However, a smaller number of monitored dependent parameters reduces the probability of accurately detecting the engine faults. Therefore, a successful prediction of deteriorated performance requires an appropriate choice of dependent parameters that are related to the independent parameters in question (House, 1992).

It is noteworthy that the simulation of clean and deteriorated performance requires dependent parameters that specify the engine's operating point. For simplicity the term 'clean' is used to describe measured or calculated baseline performance. Although these parameters are measured they are not considered as dependent parameters and therefore they do not appear in the GPA set of equations (see Equation 4.1).

Requirement

The GPA fault isolation technique is only successful if the following requirements are met (Volponi, 1982):

Relationship Appropriate relationships between the dependent and independent parameters must be established. The fault diagnostic matrix coefficients are an accurate representation of the specific engine behaviour.

Sought Fault The engine gas path related faults occurring are among those being sought. An unsought fault occurrence affecting the measured parameters would result in a false diagnostic message.

Baseline Since the technique is based on differential changes, a representative baseline must be established.

Raw Data The raw data must be corrected for the prevailing ambient conditions. The data must also be adjusted for differences in operating conditions: bleed, forward speed, altitude or Reynolds number, specific heat, normalised power, turbine speed and humidity.

Repeatable Measurement The measurements are repeatable. It is important at this point to differentiate repeatability and accuracy. Since GPA is a differential method, it is changes in the parameters that are important not their absolute values.

Limitations

However GPA cannot diagnose any of the following failures (Smetana, 1974):

Catastrophic GPA cannot predict catastrophic failure of any component.

Oil GPA cannot predict failures due to oil system or leaks unless they result in excessive spool friction or unusual accessory drive power requirements.

Bearing GPA cannot predict impending bearing failures unless they also result in excessive spool friction.

Creep GPA cannot predict creep of turbine blades unless the creep is sufficient to modify the turbine's performance. The temperature in

this area, however, when integrated over time is a measure of the tendency to creep.

Fatigue — GPA cannot predict fatigue or other failures of any mechanical element unless such failures result in changes in the boundaries of the gas path.

Economic Aspect

An operator of a gas turbine must analyse the four following factors in order to select the proper size of GPA diagnostic system tailored to his own needs. These factors are: (1) importance and costs of interruption, (2) size of operation, (3) sophistication of operation, and (4) availability of facilities and resources (Connolly, 1985).

The introduction costs of GPA in an operation must not be higher than the corresponding maintenance savings incurred per operating hour. The same diagnostic system applied to a small size operation unit and then to a large size operation unit will result in a different degree of complexity and therefore a different impact on the life cycle cost (LCC) savings. Basically, there are two areas where an operator of GPA diagnostics system could benefit: short-term and long-term benefits.

Typically, the short-term benefits of a gas turbine engine monitoring system are those associated with the daily operations and maintenance. They include (Singh, 1985):

Safety — Safety can be improved

RAM — Reliability, availability and maintainability (RAM) can be improved

Operating cost — Operating costs can be reduced. This is achieved by highlighting excessive fuel consumption, by detection of out of trim condition, by monitoring variable stator vanes angle, by assessing engine performance levels and by preventing secondary damages.

Costs — Maintenance and overhaul costs can be reduced due to reduced inspections frequency, due to pin-point deteriorated component/modules and due to improved scheduling and planning of maintenance action

Logistics — Logistics can be improved due to prediction of spare parts requirement and due to reduced consumption of spares which occur because of unnecessary maintenance

Action Maintenance action can be validated

The long-term benefits concentrate on improving the knowledge of the engine operating environment, LCC and long term management through (Singh, 1985):

Usage Engine usage and life history data are recorded

Records Data are recorded and kept

Improvements Improvement of future engine design, testing and provision of data for component improvement program is possible

Assessment Management policies in operation and maintenance are assessed

The theory of linear (Grewal, 1988) and non-linear GPA is implemented in the computer program Pythia. As mentioned earlier, problems of sensor noises are not considered in the program. The development cycle of Pythia is described in the following Chapter 5.

5 The Development of GPA Diagnostics Program Pythia

A Gas Path Analysis (GPA) computer software program requires a reliable product that suits both user and developer. This is particularly the case for computer software used for the maintenance management of very high cost machinery. A reduction of risk can be achieved by introducing software quality assurance standards (SQA). SQA supports the development, the implementation and the application cycle of computer software. The development cycle typically involves analysing the currently used program if available and designing the specification for the new program using a structured methodology. The implementation cycle involves using an appropriate programming language that can cope with large and complex software products. And finally, the application cycle involves testing the software in terms of software faults and user-friendliness.

This chapter describes the main aspects of SQA, the basic development process of the new GPA program Pythia with a commonly used methodology and the implementation of Pythia on a personal computer (PC). Chapter 6 demonstrates the potential use and the user-friendliness of Pythia.

5.1 SOFTWARE QUALITY ASSURANCE

Software Quality Assurance is a term used to describe the design process of a product that gives the user maximum confidence at an acceptable level of quality (British Standards Institution). Usually, Software Quality Assurance can be presented as a three-task process: (1) defining clearly what is to be achieved and when (specification), (2) describing the activities and functions that need to be performed (quality standards) and (3) controlling and monitoring the performance of activities and functions (quality control).

Specification

It is a fact that with the increased development of computer hardware there has been a growing awareness of the cost of software faults (Vella, 1992). Software related faults are usually a result of faults in logic which cause errors in program statements and therefore produce incorrect results (Hancock, 1991). Faults can derive from:

- human errors in coding or entering data;

- data corruption due to hardware faults;
- electromagnetic interference;
- specification anomalies.

Whilst the first three sources of errors are 'easy' to diagnose the most frequent error arises as a result of omissions in the original specification. Bearing in mind that the cost of detecting and correcting failures increases throughout the stages of the design and implementation of software, the analysis and design phase is crucial to the development of computer software. It contributes to the production of clear design specifications.

Quality Standards

A reduction of risk in software development can be achieved by introducing quality standards such as ISO 9000 (TickIT, 1990). Quality standards in software engineering require guidelines, software tools and methodologies. Software tools such as Visual C++ (Microsoft, 1993) can considerably reduce the time spent with coding a computer program. Methodologies produce clear specification of the new program. The most commonly used methodology in the United Kingdom is Structured System Analysis and Design Methodology (SSADM, see Cutts 1991).

Quality Control

As the development of a software product progresses, it is important to have several check points so as to identify the quality achieved so far. A selection of quality control elements are: software testing, user training, support etc. (TickIT, 1990)

5.2 STRUCTURED SYSTEM ANALYSIS AND DESIGN METHODOLOGY

SSADM is a way of organising the analysis and design parts of projects which aim at delivering a computer-based information system. It is a diagramming technique that offers a number of views of a system. The different views allow checks for consistency to be made as a system is developed. The approach also combines the techniques of system analysis into a framework which defines when they should be used (SIMS, 1992).

Structured Methodology

The need for a structured methodology is evident from the ever increasing problems with current methods of developing computer systems (Cutts, 1991). The major problem is related to project costs and timescales. It is a fact that only a small proportion of projects are completed within their budget and on time. There are many reasons for this such as modifications of the user's requirements, developing systems that do not meet the user's requirements etc. In addition, the cost of rectification increases considerably if a system needs to be modified once it is implemented. The reduction of this rectification cost is another reason to follow a structured methodology.

Therefore a good methodology should provide the specification of requirements and the detailed design specification. In each case the methodology should lead to concise, unambiguous specifications. Typically, the product of a structured methodology is presented in a graphical and written manner that accurately reflects the user's requirements.

The SSADM Development Cycle

SSADM is a step by step approach which starts with an investigation of the current system (e.g. the program Detem) and which concludes with a detailed system design specification for the new system (e.g. the new program Pythia). It not only gives help to the design or development of a system, it also provides a structure to the different stages and the sequences in which they should be applied. There are many different representations of the systems development cycle. Figure 5.1 gives an outline of these stages (SIMS 1993, Cutts 1991 and Downs 1992).

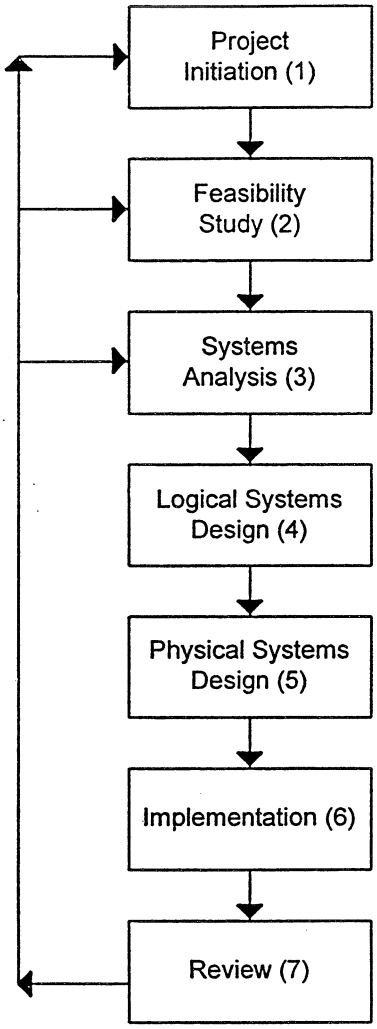


Figure 5.1: Stages in System Analysis and Design Projects

Project initiation (1) starts the project. A problem or a new requirement has been identified and a project is required to develop a solution. Typically, this stage includes the problem description, the scope and boundaries of the project and a project plan with the resources required.

A feasibility study (2) is required to determine whether the objectives are realistic, i.e. whether they are likely to be achieved and put into effect. The feasibility study has to address technical, economic and social areas of concern. Usually, at the end of a feasibility study, a stop/go decision must be made for the project. The feasibility study must therefore present a detailed specification of the objectives jointly with an analysis of the impact on the organisation of any project aimed at meeting them. For the present thesis, the stages project initiation and feasibility study have been tackled in Chapter one, where the problem of increased costs have resulted in better understanding of

maintenance strategies. One way to support maintenance decisions is to use performance analysis techniques such as the Gas Path Analysis technique.

The definition of system analysis (3) is the organisation of information gathered during the feasibility stage into a meaningful form. This generally means that the building of a model which represents what the current system accomplishes, not how it is accomplished. The model is logical, not physical, and should include both the functions carried out by the system and the data stored within the system. Furthermore, the model should be able to identify problems and possible improvements. This will give the specification of requirements for the new or final product.

The specification of requirements forms the input to the design stage (4 and 5). The objective of this phase is to specify the system in a way suitable for implementation on the chosen hardware with the chosen software. The design phase can be divided into logical design (4) and physical design (5). The logical design will transform the specification of requirements into detailed specifications of both data and processing requirements. In other words, what is required of the new system will be stated. The physical design will transform the logical specification into physical specifications, database and program specifications by including constraints imposed by the chosen hardware and software. During the physical design stage, details such as input and output formats will be designed.

Implementation (6) is the stage where the system design is translated into an operational system. It includes programming, program testing, hardware and software acquisition and installation, system testing, user training and change-over to the new system. However, programming and testing are not supported by the SSADM methodology.

Once the system is implemented the review stage should continually monitor the new system to ensure that it meets the user's requirements. If it fails to meet the user's requirements a new design project is to be initiated.

Benefits of SSADM

Many users of SSADM have demonstrated that the method does not increase the cost or timescale (Cutts, 1991). As a matter of fact, in some cases there exists a great potential to shorten the timescale and decrease the cost of development by increasing the chances of meeting the user's exact requirements the first time.

The reason why the method helps to meet the user's requirement lies in the fact that the user is involved in the development process. Also the format of the documentation

makes the system specification understandable for the user. Techniques such as data flow diagrams allow users to make their own decisions and contributions.

A structured methodology forces the development team to consider the detail early in the project to ensure that the specification of requirements and the design specification are correct. Another benefit is the detailed documentation as part of the methodology. This ensures that all documentation is completed as part of the project. In other words, SSADM enables the users to obtain the systems they require (Cutts, 1991).

The Techniques Used in SSADM

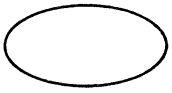
SSADM is not based on a single technique but comprises a framework of several stages that make use of several techniques. The crucial techniques are:

Data Flow Diagram	Data flow diagrams show the boundary of the system and its relationships to the external world. They also show the functions, data sources, input and output for the system.
Entity Model	Entity models show the data structures and data relationships for the system. Cluster of data structures or entities will eventually form the data stores on the final data flow diagrams.
Entity Life History	Entity life histories show how each entity is affected by system functions and provide a dynamic view of a system.
Normalisation	Normalisation transforms complex data structures into simple lists and is used to build entity models from the input and output data structures.
Process Outline	Process outlines specify the operations necessary to process a transaction in the system. These lead towards the production of program specifications.

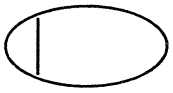
Data Flow Diagram

A key tool in SSADM are the data flow diagrams (DFD). They are used to show how the data flow through an information system and the activities that are carried out to create the outputs of the system. The diagrams provide a picture of the system that can be understood by all those involved in the study - not just the analysts. Essentially, DFD's are very simple and there are few symbols used in them. The basic symbols are denoted in Figure 5.2.

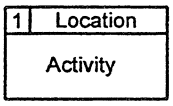
Context Boundary: defines the area to be studied



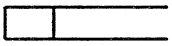
External Activity: defines activities outside the scope of the study; i.e. they are outside the context boundary



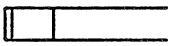
Duplicate External Activity: defines an activity that is drawn into more than one place within the DFD



Activities: defines function to the data or flow through the system



Data Store: defines stores of data in computer files



Duplicate Store: defines a store that is drawn into more than one place within the DFD

Figure 5.2: Basic Symbols Used in Data Flow Diagrams

The data flow diagram (DFD) of the programs Detem and Pythia adopt a top-down approach. Each program is defined at its highest level and then successively broken down until the lowest level is reached. In this way the relationship between activities and associated data can be shown (see Figure 5.3).

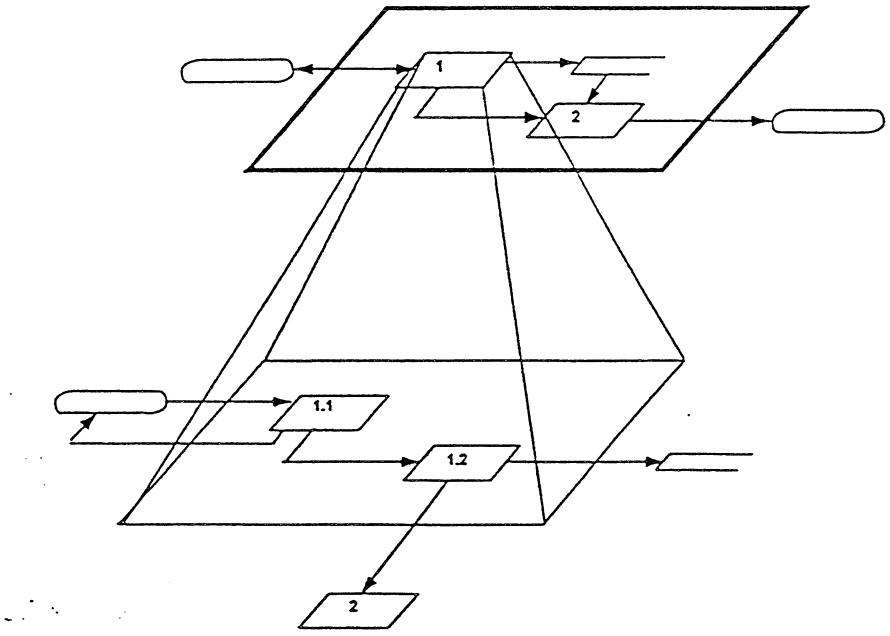


Figure 5.3: Data Flow Diagramming Process (SIMS, 1992)

The use of data flow diagrams for the programs Detem and Pythia are shown and explained in the System Analysis and System Design section at the end of this Chapter.

Entity Model

Data flow diagrams provide a clear representation of the flow of information and the activities are carried out. However, they do not describe the relationships between the entities with stored data in a system. The entity model diagrams can provide a system view of the data structures and data relationships within the system.

The use of entity modelling in the process of designing the new program Pythia has been undertaken but the relevant diagrams are not included in the present thesis. The reason is that an entity can be described as an object in the C++ environment. Objects in C++ will be explained in Chapter 5.3.

Entity Life History

So far, two diagram techniques have been introduced: Data Flow Diagrams and Entity Relationship Diagrams. They do not form a complete picture of the system. For an existing systems they are sufficient. When designing a new system it is important to know how entities change as they progress through the system. Entity life history diagrams (ELH) can achieve this. The ELH is a diagrammatic representation of the life

of an entity from its creation to its deletion. Its life is expressed as a sequence of events that cause an entity to change.

For the present Pythia program development this technique was not used since Pythia is not a complete new system.

Normalisation

Systems are mainly concerned with data. An essential part of the systems analysis is to establish what data is needed and how it is to be stored in e.g. a database. There is a procedure that can help to design a data base: Normalisation. The objective of Normalisation is to eliminate unnecessary duplication of data and the problems of updating the data base.

A detailed description of how the data are normalised for the Pythia program is not included in the present thesis since a detailed description of used data or variables is given in the Class Library (see Chapter 5.3).

Process Outline

At some stage, the diagrams and documentation that have been created for the system will have to be converted into a computer code. Process outlines are a means for gaining a picture of what the final program will look like. Process outlines specify in detail what the required processing is. However, this stage of SSADM is one of the weakest since no clear guidelines are given to what should be produced at this stage. One solution to this problem is the use of CASE tools such as Visual C++. CASE tools can considerably reduce time spent with coding a computer program as explained earlier.

System Analysis

Within the SSADM environment there are two parts to the system analysis phase. The first is to carry out a detailed study of the current program Detem and the second is to specify the requirements of the new program Pythia.

Current System Detem

The current program Detem (Grewal, 1988) allows to generate clean and deteriorated engine performance data and to carry out linear GPA studies. Detem obtains clean performance data by using the incorporated performance program Turbomatch (Palmer, 1967). Turbomatch's modular construction enables the modelling of any type of open-cycle gas turbine engine that runs under steady-state conditions. The steady-state gas

path parameters are established for relevant engine stations with the component matching technique (Cohen, 1986). It is also possible to obtain the overall performance indicators such as thrust, power or specific fuel consumption. The component-matching technique relies on built-in or user defined component characteristics. Detem obtains deteriorated performance data by changing the scaling factors of the appropriate component characteristics such as those for compressors and for turbines (see Chapter 5 for more details on scaling factors). As a special case, compressor component deterioration can be simulated by one of the three following methods:

DEFAULT Independent parameter changes such as flow capacity and efficiency are specified by the user. Detem calculates pressure ratio changes as a function of flow capacity changes.

Model 1 Flow capacity and efficiency changes are specified by the user. Detem changes pressure ratio equal to flow capacity changes.

Model 2 Flow Capacity, efficiency and pressure ratio changes are specified by the user.

Detem's interaction with the user is achieved via an interactive screen display. The display is based on an extended Fortran-77 graphics library and it runs on a VAX computer. Detem offers the user to choose between three modes of operation:

Diagnostics User carries out deteriorated performance studies other than at design point. Deteriorated performance is achieved by implanting independent parameter changes.

Coefficient Detem generates the influence coefficient matrix (ICM) at design point.

Analyser Detem performs linear GPA.

A toplevel DFD of the current program Detem is shown in Figure 5.4.

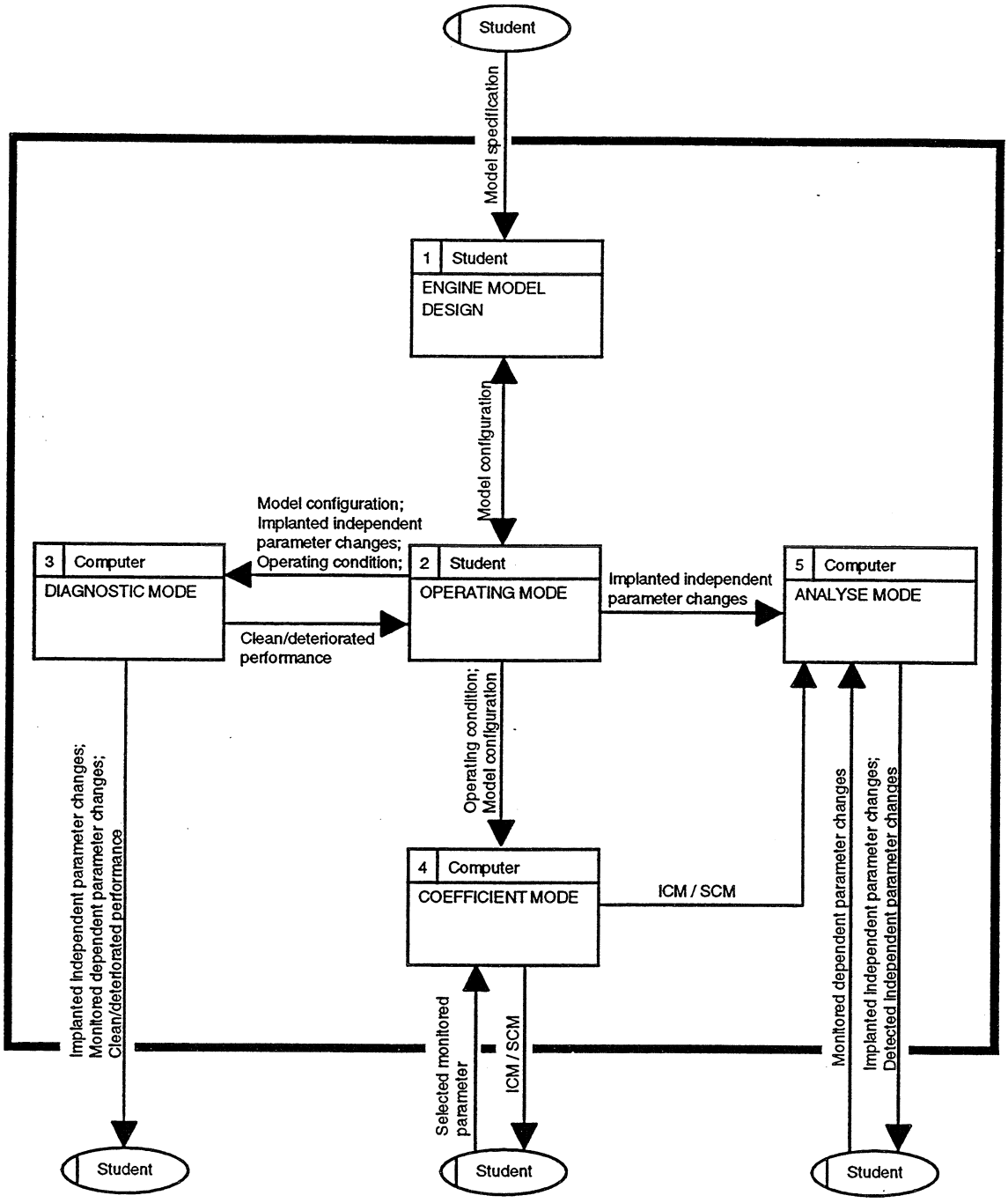


Figure 5.4: Current DFD of Program Detem (Level 1)

Detem allows the user to model an open-cycle gas turbine engine, to carry out design point and off-design point calculations, to look at deteriorated engine performance, to calculate the Influence Coefficient Matrix and to conduct linear Gas Path Analysis. The user decides on the specifications of the gas turbine engine. The engine is then modelled by assembling its components.

Once an engine is configured, the user can operate three modes: diagnostics, coefficient and analyse mode. If diagnostics mode is chosen the user can decide between clean or deteriorated engine performance modelling. The clean performance calculation requires the engine configuration and operating condition of the engine. The diagnostic mode then will provide clean performance data either at design point or off-design point. The deteriorated performance calculation requires implanted independent parameter changes. The results of clean and deteriorated performance data can be viewed or stored.

If coefficient mode is chosen an Influence Coefficient and a Sensor Coefficient matrix are calculated and stored. The sensor coefficient matrix (SCM) is concerned with measurement non-repeatability. In the present thesis sensor problems are not considered. If the analyse mode is chosen linear Gas Path Analysis is carried out. Changes in measured parameters and an inverted ICM will result in independent parameter changes which can be compared against implanted independent parameter changes. The measured changes are derived from the diagnostics mode.

System Requirements

The program Detem was investigated from the viewpoint of the persons who wrote it (Grewal, 1988) and those who used it (Burkhardt 1990, Donaghy 1991, Goh 1989, Lynch 1989 and House 1992). Burkhardt incorporated a number of revisions in the program Detem in order to rectify some of the discrepancies noted in earlier studies. However, the problems with Detem are too severe to make it capable of performing non-linear GPA studies. For this reason, instead of improving the program Detem a new program Pythia has been developed. Detem's knowledge base will be used to specify Pythia's requirements (see Table 5.1). In Table 5.1 the problem reference column refers to the activities in the DFD of the current program Detem. Similarly the solution reference refers to the activities in the DFD of the new program Pythia.

Table 5.1: Problems and Requirements of the System Detem/Pythia

Problem reference	*Problems and Requirements*	*Solution*	*Solution reference*
DFD activity 4, 5	Conducting GPA at off-design point.	Allow ICM generation at off-design point.	DFD activity 5
DFD activity 5	Analyse mode is not working.	Analyse mode redesigned and presented in GPA mode.	DFD activity 5
DFD activity 3, 5	Graphics mode is not working.	Use of already available graphics package on PC (e.g. MS-EXCEL). Thus graphics package is excluded from software development.	External entity Windows
DFD activity 3, 4, 5	Not all desired dependent parameter changes can be monitored.	Free choice of monitor parameters.	DFD activity 4
DFD activity 4,5	Different numbers of dependent variables to independent variables not possible (i.e. ICM can only be of square form).	Adapt Donaghy's (1991) technique of inverting non-square matrices.	DFD activity 4
Turbo-match	Off-design point setting choice is limited.	Introduction of new version of Turbomatch	Turbo-match
DFD activity 5	Linear GPA results are not accurate enough -> program Detem cannot detect multiple implanted faults (degradations).	Implementation of non-linear GPA (House, 1992).	DFD activity 4
DFD activity 4,5	Choice of independent parameter fixed.	Free choice of independent parameter for compressor, burner, turbine and nozzles.	DFD activity 4
DFD activity 5	Calculation of FCM required.	Inversion of ICM (linear- or non-linear method).	DFD activity 4

Detem	Program reliable and suitable for general applications required.	Program Turbomatch exchangeable with other performance simulation programs. Introduction of software quality assurance with SSADM methodology.	Pythia
Detem	Reusable source code.	Use of object-oriented programming language.	Pythia
Detem	User-friendly program required.	Program in Windows environment using programming aid tool Visual C++	Pythia

System Design

In order to meet the requirements that have been developed in the systems analysis section, the new program Pythia incorporates a number of modifications over the current program Detem such as the systems environment, the performance program Turbomatch and three separate modes of operation (Design engine, clean performance and GPA mode):

PC Environment Pythia is developed under the Microsoft Windows environment on a PC. This allows the exchange of data from Pythia to other Windows applications such as spreadsheet programs etc.

Turbomatch Turbomatch is excluded from the system boundary i.e. Pythia provides the interface for carrying out performance calculations with Turbomatch. This exclusion has the advantage that Turbomatch can be exchanged with other performance programs by simply adjusting the interface.

Engine Design Mode In the design engine mode, the user assembles the components such as intakes, compressors etc. in order to define the layout of the gas turbine engine. Pythia connects the components with programming objects and establishes the appropriate set of equations for design point and off-design point calculations.

Clean Performance In the clean performance mode, design point and off-design point calculations are carried out with the component matching method

Mode Turbomatch.

GPA Mode In the GPA mode, the user chooses between deteriorated performance studies and linear or non-linear GPA studies. Deteriorated performance studies require a clean set of engine data which can be obtained from an off-design point calculation. Once a clean engine is established the user can perform deterioration studies by implanting faults (changes in independent parameters). GPA studies require measurement changes (monitored dependent parameters) in order to identify deteriorated components. Changes in dependent parameters are obtained by taking the difference between clean and deteriorated engine data. Based on the dependent parameter changes, Pythia carries out GPA and calculates changes in independent parameters (Detected faults). A comparison of implanted and detected faults will determine the quality of the GPA study.

Data flow diagrams of the new program Pythia are shown and explained in the following section. As with the analysis of the program Detem, the Data Flow Diagrams (DFD) of the new program Pythia adopt a top-down approach. The DFD are based on the requirements that are described in Table 5.1.

Pythia

A toplevel DFD of the new program Pythia is shown in Figure 5.5.

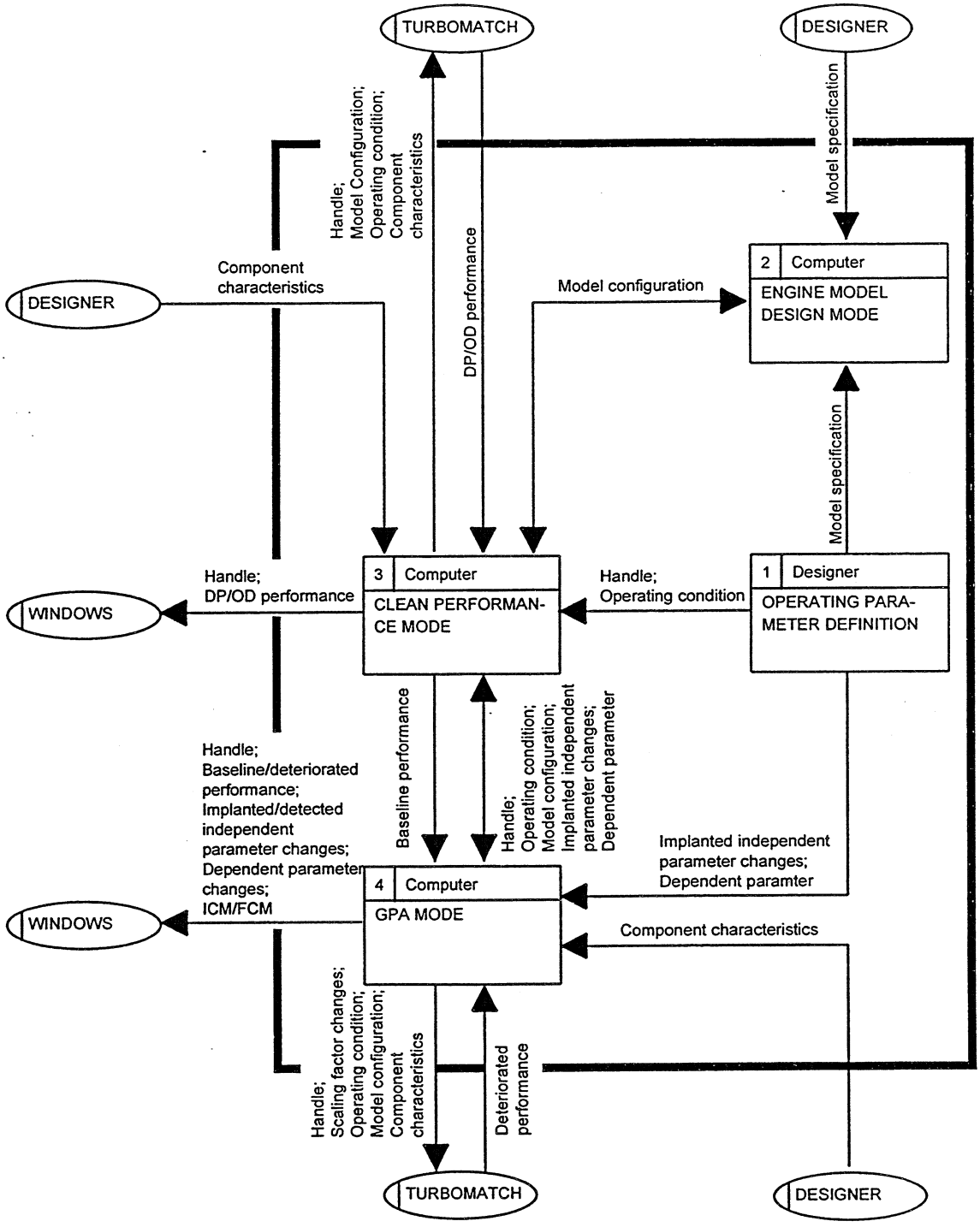


Figure 5.5: Required DFD of Program Pythia (Level 1)

Pythia allows the user to model any open-cycle gas turbine engine (Design engine model mode), to carry out design point and off-design point calculations (Clean performance mode), to look at deteriorated performance and to perform GPA studies (GPA mode).

In the engine model design, the user assembles components according to the user's specification and defines, if necessary, initial operating conditions for the engine. Once the engine model is completed (Model configuration), the model is transferred to the clean performance mode. In this mode, the user defines all necessary operating parameters for each component in order to carry out design point and off-design point calculations. For the off-design point calculations Pythia requires the definition of off-design point settings (Handles). In addition, the user can either supply user-defined or Turbomatch built-in component characteristics for compressors and/or turbines. The results of the performance calculations can either be viewed or exported to other Windows applications. If the engine needs to be changed then the set of data is exported to the engine model design where the user can rearrange the components. The new arrangement forms a new case for the clean performance mode.

In the GPA mode, the user can choose between two operating modes: (1) carrying out deteriorated performance studies and (2) conducting GPA studies. In both cases a suitable baseline performance is required. The baseline is obtained from an off-design point calculation that has been carried out in the clean performance mode. Deteriorated performance data are obtained by implanting independent parameter changes on the appropriate components. In other words, Pythia will call the Turbomatch program with changed scaling factors. Furthermore Pythia will use the handles that have defined the baseline performance. GPA studies require the selection of measurements (Monitored parameters) and independent parameters. Based on the monitored parameter changes from baseline to deteriorated performance, the linear and non-linear GPA will calculate independent parameter changes (Detected faults). These detected faults can be compared against implanted faults. Results of deteriorated performance and GPA studies can be exported to other Windows applications. If the handle setting needs to be changed, or a different baseline is required, the set of data can be exported to the clean performance mode where new handles and operating conditions can be defined. The new off-design point calculation forms a new baseline for deteriorated performance or GPA studies.

Engine Model Design

A level 2 DFD of the engine model design activity is shown in Figure 5.6.

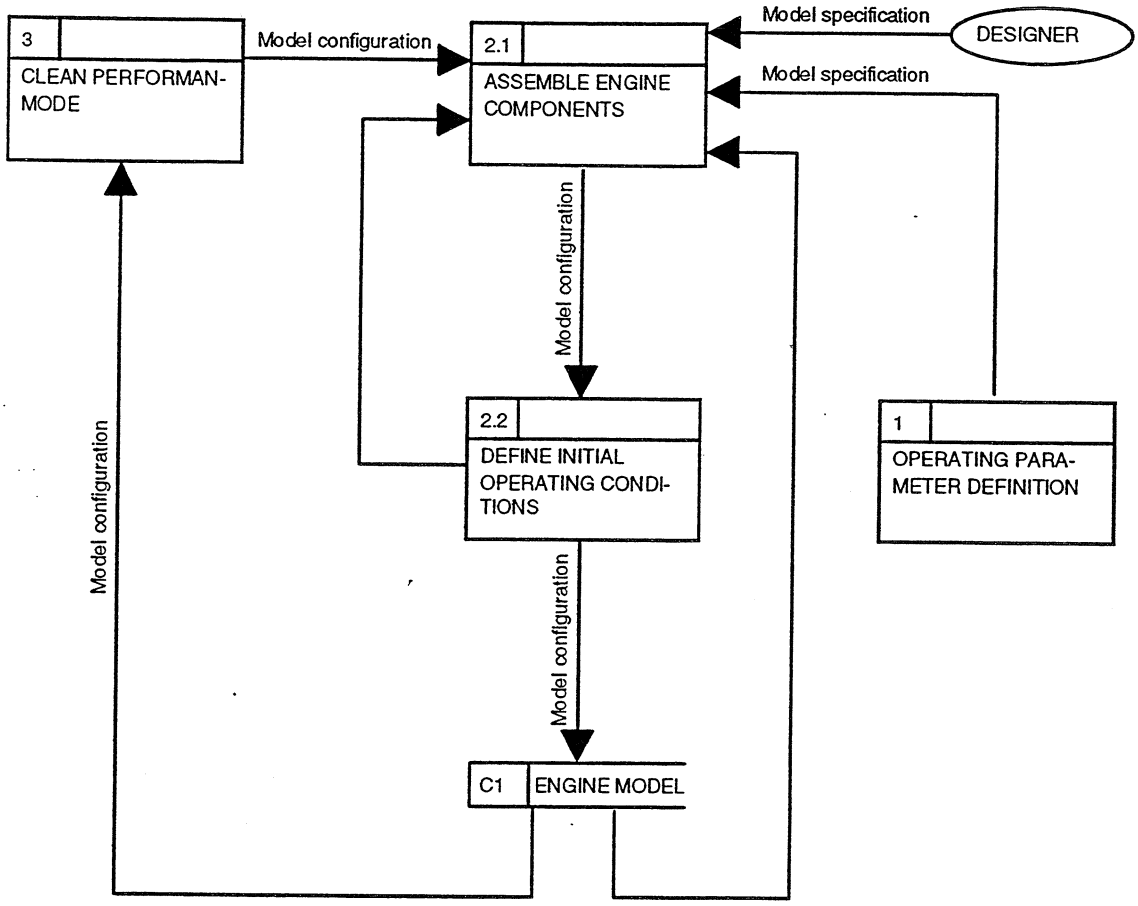


Figure 5.6: Required DFD of Program Pythia Level 2 (Activity 2: Engine Model Design Mode)

The activity enables the user to assemble the components of an engine by means of graphical symbols. Relevant operating conditions of each component can be initialised. It is also possible to rearrange the layout of the components.

Clean Performance Mode

A level 2 DFD of the clean performance mode is shown in Figure 5.7.

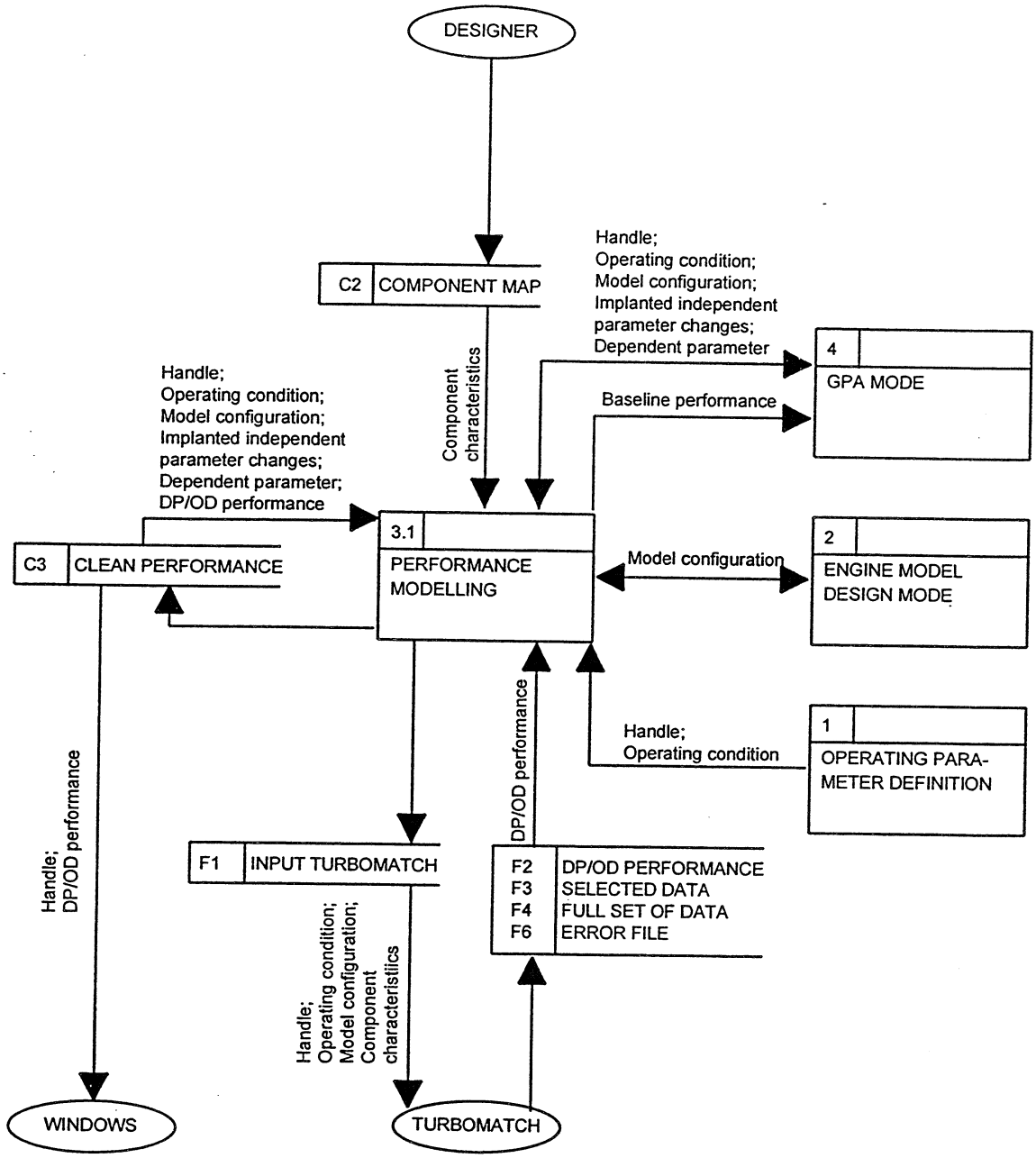


Figure 5.7: Required DFD of Program Pythia Level 2 (Activity 3: Clean Performance Mode)

This mode allows the user to carry out design point and off-design point calculations. For this purpose operating conditions in each component have to be defined. Operating conditions along with the model configuration and handles form the input file for Turbomatch. Once a performance calculation has been carried out Turbomatch will provide Pythia with four files that are read in.

GPA Mode

A level 2 DFD of the GPA mode is shown in Figure 5.8.

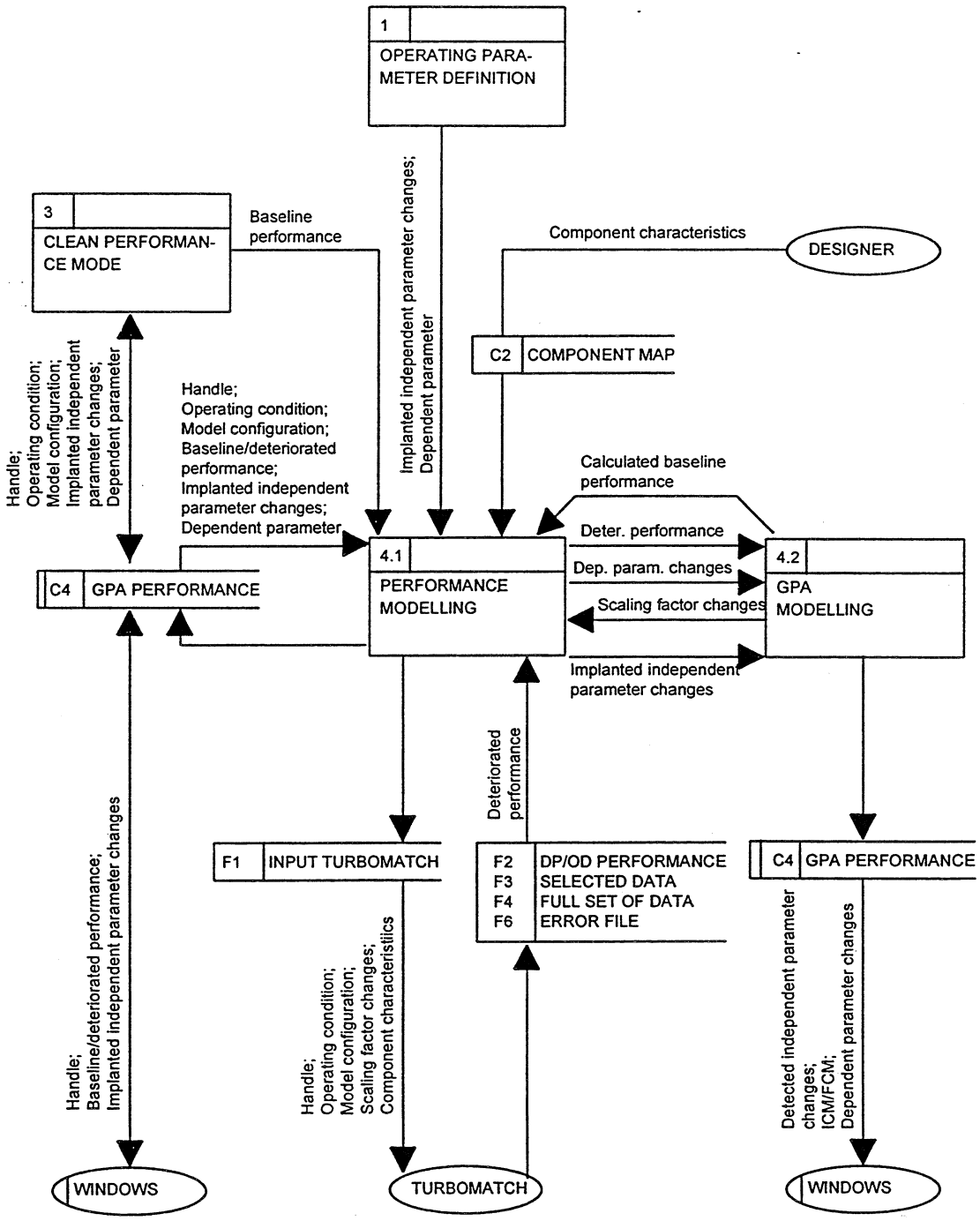


Figure 5.8: Required DFD of Program Pythia Level 2 (Activity 4: GPA Mode)

Two tasks can be carried out: (1) deteriorated performance studies and (2) GPA studies. Deteriorated performance studies are carried out similarly to the clean performance mode except scaling factors are changed according to changes in independent

parameters. GPA modelling uses either the linear or non-linear GPA technique in order to identify independent parameter changes.

GPA Modelling

A level 3 DFD of the GPA modelling activity is shown in Figure 5.9.

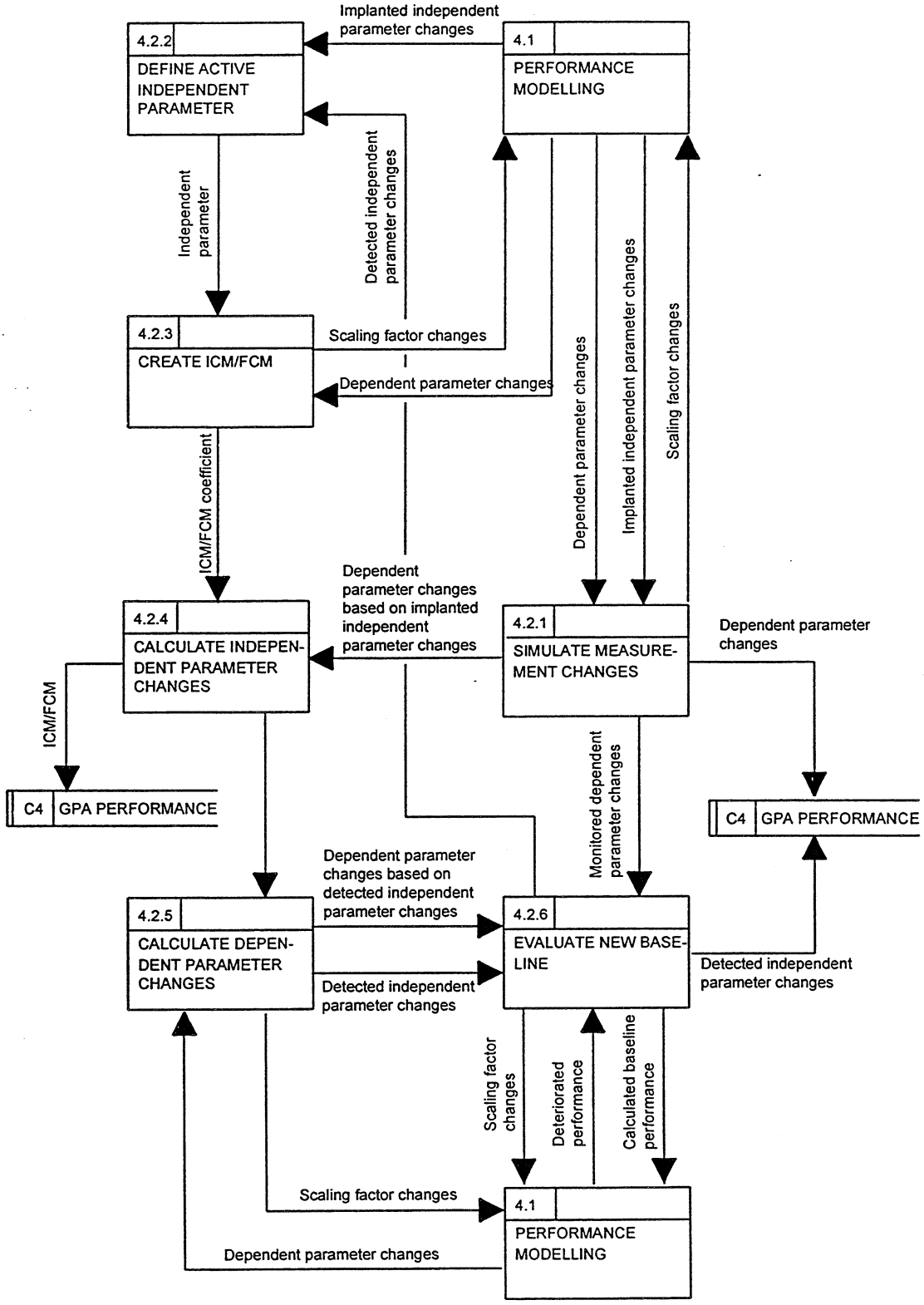


Figure 5.9: Required DFD of Program Pythia Level 3 (Activity 4.2: GPA Modelling)

The activity carries out linear and non-linear GPA calculations. A GPA calculation identifies changes in independent parameters based on changes in dependent parameters (simulated measurement changes). Therefore at the start of each GPA calculation, measurement changes are simulated by implanting independent parameter changes on component characteristics and 'rematching' the engine. Next the partial derivative of the ICM columns are defined by perturbing in turn each independent parameter in order to calculate the perturbed dependent parameter changes.

Once the ICM calculation is accomplished, the independent parameter changes are calculated by multiplying the inverted ICM (=FCM) by the simulated measurement changes. These calculated independent parameter changes define the solution of the linear GPA technique.

Next the non-linear GPA technique establishes a calculated deteriorated performance based on the calculated independent parameter changes. This calculated performance is compared against the simulated performance that was established by the implanted independent parameter changes. If the calculated performance is not within the user defined convergence margin (see $\Delta y_{\_sum}$ in Equation 4.11 in Chapter 4) then it will be used as a new baseline performance. The second iteration of non-linear GPA starts. Detected independent parameter changes are perturbed in order to create the ICM. Based on the measurement changes of the new baseline and simulated deteriorated performance new independent parameter changes are calculated. These parameters will define a new deteriorated performance that is checked against the simulated deteriorated performance. The non-linear GPA repeats the process until it meets the convergence criteria.

5.3 PROGRAMMING

The previously defined design specifications of Pythia are translated into an actual computer program. The process of programming requires an appropriate computer programming language and should provide a well-documented program with a manual for the user. The following section presents the reasons why the programming language C++ has been chosen, the description of the C++ program Pythia, the changes made to the Fortran program Turbomatch and the user manual for Pythia. It is assumed that the reader is familiar with the C++ programming language (Lafore, 1991), the terminology used in Turbomatch (Palmer, 1995) and the software tool Visual C++ (Microsoft, 1993). However, some basic concepts of each topic will be provided so that the unfamiliar reader can follow the main ideas.

The Programming Language C++

The main purpose of a programming language is to support the construction of a reliable software package. Therefore it is important to have a proper balance between powerful language features and their easy implementation into a computer program (Wilson, 1988). As programs have grown large and complex even structured programming approaches (e.g. Fortran-77) begin to show signs of strain (Lafore, 1991). Therefore object-oriented programming languages such as ADA, Smalltalk and C++ have been put forward as new software development tools.

Users of gas turbine engines require a user-friendly program since they do not have large teams of experienced engineers available during the operation phase (MacIsaac, 1992). With the development of Windows-based software, program users are no longer expected to spend long periods of time learning how to master a new program. Windows programs have a similar user-friendly frame that provides a multitasking, graphical based environment for program operation. The object-oriented programming language C++ is suitable for Windows programming. During the phase of programming the software aid tool Visual C++ assisted the coding (Microsoft, 1993).

Object-Oriented Programming

The fundamental idea behind object-oriented languages is to combine both data and the functions that operate on that data into a single unit. Such a unit is called an object. The functions of an object (Member functions) typically provide the only way to access its data. A C++ program consists of a number of objects, which communicate with each other by calling one another's member function. A collection of similar object is called a class. The concept of classes leads to the idea of inheritance. A class is divided into subclasses (base class). Other classes can then be defined that share the characteristics of the base class. These are called inherited or derived classes.

In the object-oriented view of programming, a program describes a system of objects interacting. Its major benefits are:

Data Abstraction Data abstraction makes writing large programs simpler.

Encapsulation Encapsulation makes it easier to change and to maintain a program.

Class Hierarchy Class hierarchy is a powerful classification tool that can make a program easily extensible (Microsoft, 1993).

'Abstraction is the process of ignoring details in order to concentrate on essential characteristics' (Microsoft, 1993). Typically a programming language is considered 'high-level' if it supports a high degree of abstraction. As an example, the Assembly programming language contains much more detailed description of what the computer has to perform for a particular task than the Fortran programming language. This process of abstraction in e.g. Fortran makes the program clearer and easier to understand.

Encapsulation is the process of hiding the internal processes of a class to support or enforce abstraction (Microsoft, 1993). In other words, an object's function (member function) provides the only way to access its data. If data need to be read from an object, then a member function is called. It will read the data and return a value but it cannot access the data directly. The data is hidden, so it is safe from accidental alteration. Data and its functions are said to be encapsulated.

Class hierarchy is the ability to define a hierarchy of types. In C++, a type could be a class, a derived class etc. It is also possible to define similarities between classes, or derive them from one base class etc.

The Pythia Class Library

As previously mentioned the concept of class hierarchy is used and a Pythia class library version 1.1 has been created. The Pythia Class Library is fully based on the Microsoft Foundation Class Library version 2.0 (Microsoft, 1993). The Pythia Class Library enables C++ programmers to write applications for Microsoft Windows. The class library gives the programmer a complete 'application framework'.

The Application Framework

'The Class Library is a group of C++ classes collectively known as an application framework. These classes provide the framework and essential components of an application for the Windows graphical environment. The purpose of the framework is to reduce the effort required to design and implement applications for Windows' (Microsoft, 1993). In other words, the framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton. The task of filling the skeleton (Things that are needed for the application) is left to the programmer.

In addition, the framework is easy to reuse because it is an object-oriented class library. This means that instead of directly editing the framework's source code, the programmer derives new, specialised classes from those in the library. The derived classes inherit all

of the behaviour and functionality of their base classes, but they can be extended by adding new member variables and functions and by modifying the existing behaviour by overriding inherited member functions. This has the advantage that large programs are easily extensible and maintainable.

For example, any programmer starting on a large project will develop some kind of structure for the code. The problem is that each programmer's structure is different, and for a new team member it is difficult to learn the structure and conform to it. Therefore the class library application framework incorporates its own application structure - one that has been proven in many software environments and in many projects (Microsoft, 1993). It is expected that the newly created Pythia class library will help C++ programmers to easily modify or enhance the program Pythia. In the same way the present version of Pythia has been developed with the Microsoft Foundation class library. A detailed description of Pythia's classes, its member functions and its member data is shown in Appendix B.

The framework is based on the four following main concepts:

Application Object	The core unit of an application for windows is an 'application object'. The application object manages a list of documents and dispatches commands to other objects in the program
Document	The unit of data that the user works with is a document. The document maintains, loads, and stores its data.
View	The user interacts with a document through a 'view' on the document. A view is a window embedded in the area of a frame window that borders the actual program. The view displays its document's data and takes mouse and keyboard input, which is translated into selection and editing actions.
Object	Objects in the user interface, such as menus and buttons, send commands to the documents, views, and other objects in the application. Those objects carry out the commands.

A possible relationship between a document and its view is shown in Figure 5.10.

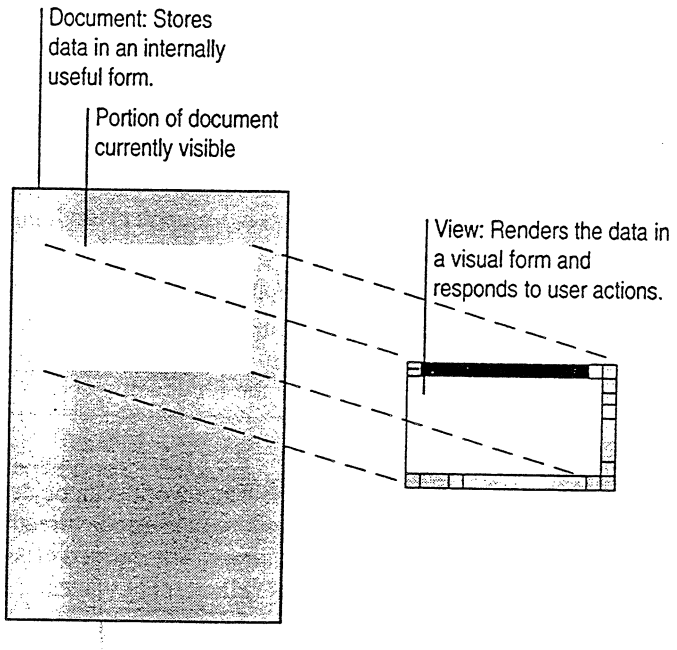


Figure 5.10: Document and View Relationship (Microsoft, 1993)

Based on this framework the main tasks of a programmer is to define the application's data in its document class(es), to define how the user views and interacts with the data inside a window and to connect menus, buttons, and other user-interface objects to commands and finally to define functions that can carry out the commands.

Benefits of the Class Library

The Pythia Class Library provides a thorough basis that allows the programmer to spend most of the programming effort with writing the code that handles the data rather than reinventing the graphical user interface. Figure 5.11 shows schematically how the programmed code can fit into the framework.

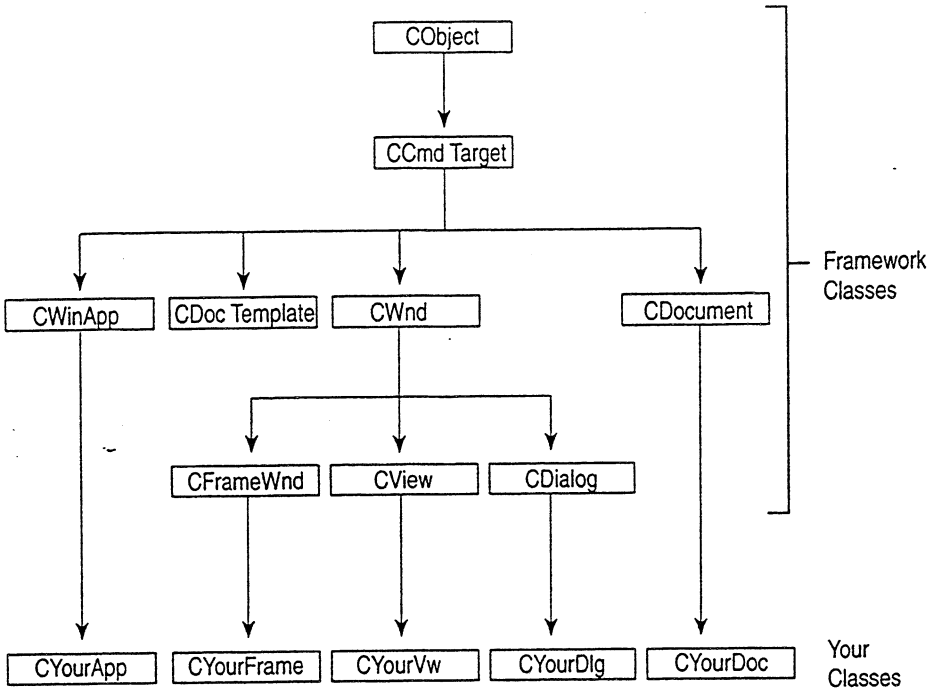


Figure 5.11: The Programmers Code in the Application Framework (Microsoft, 1993)

The Pythia class library is based on the Microsoft Foundation Class which is presented in Figure 5.11. The classes used in Pythia are derived from the classes in the Foundation class. The following classes are the ones that have been created for the Pythia application:

Collections — The class library contains a number of ready-to-use lists, arrays, etc. that are referred to as 'collection classes'. A collection is an extremely useful programming idiom for storing and processing groups of class objects or groups of standard types. A collection object appears as a single object. Class member functions can operate on all elements of the collection (see Figure 5.12).

Document Architecture — The document architecture includes document objects that manage the application's data for the engine design, the clean performance and the GPA mode (see Figure 5.14).

Maps — Objects in this category manage component characteristic data for compressor and turbine components (see Figure 5.14).

Modules Objects in this category assist the handling of components/modules of an engine (see Figure 5.13).

Dialog Boxes Applications for Windows frequently communicate with the user by means of dialog boxes. The dialog base class and its derived classes encapsulate dialog-box functionality (see Figure 5.12).

Control Bars Control bars greatly enhance a program's usability by providing quick, one-step command actions. This typically includes controls such as buttons, list boxes etc. (see Figure 5.14).

CCmdTarget The CCmdTarget class serves as the base class for all classes of objects that can receive and respond to messages (see Figure 5.14).

Frame Windows The framework creates, uses and destroys documents and views. This typically happens when the user opens or closes a file (see Figure 5.14).

Views Objects in the view category incorporate child windows that represent the client area of a frame window and that show and accept input for a document (see Figure 5.14).

Any relation to the Microsoft Foundation class in Figure 5.11 is shown in the Figures 5.12 to 5.14 with boxes that are surrounded by double lines.

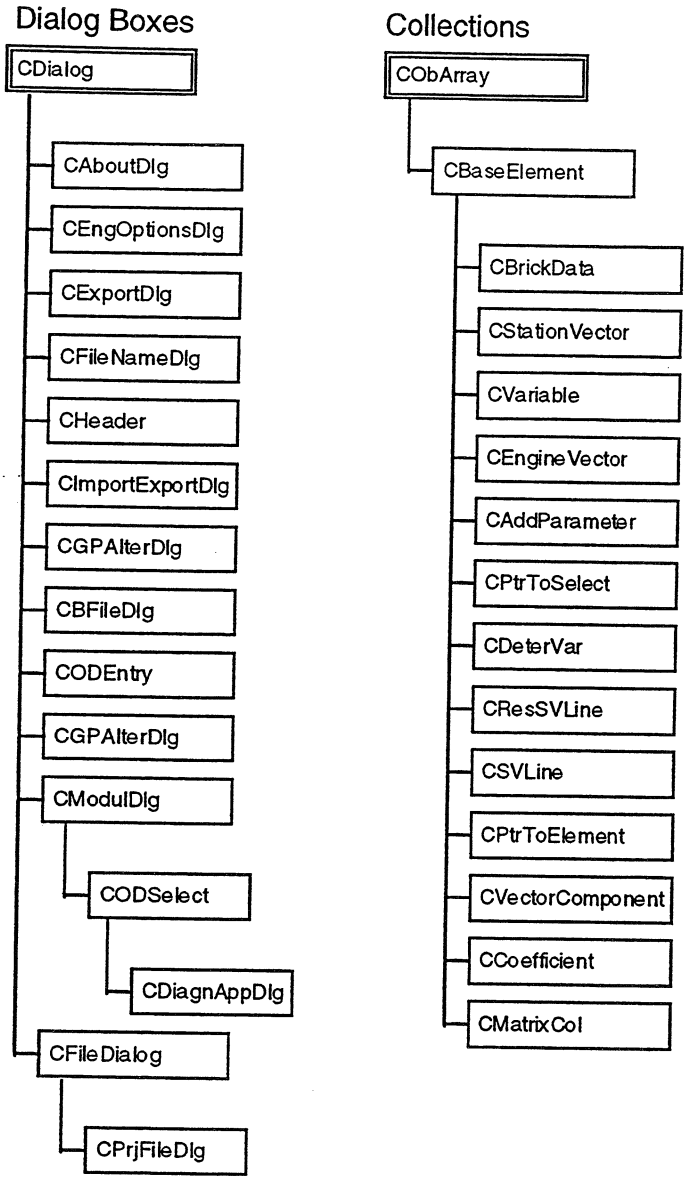


Figure 5.12: Dialog and Collection Class Hierarchy

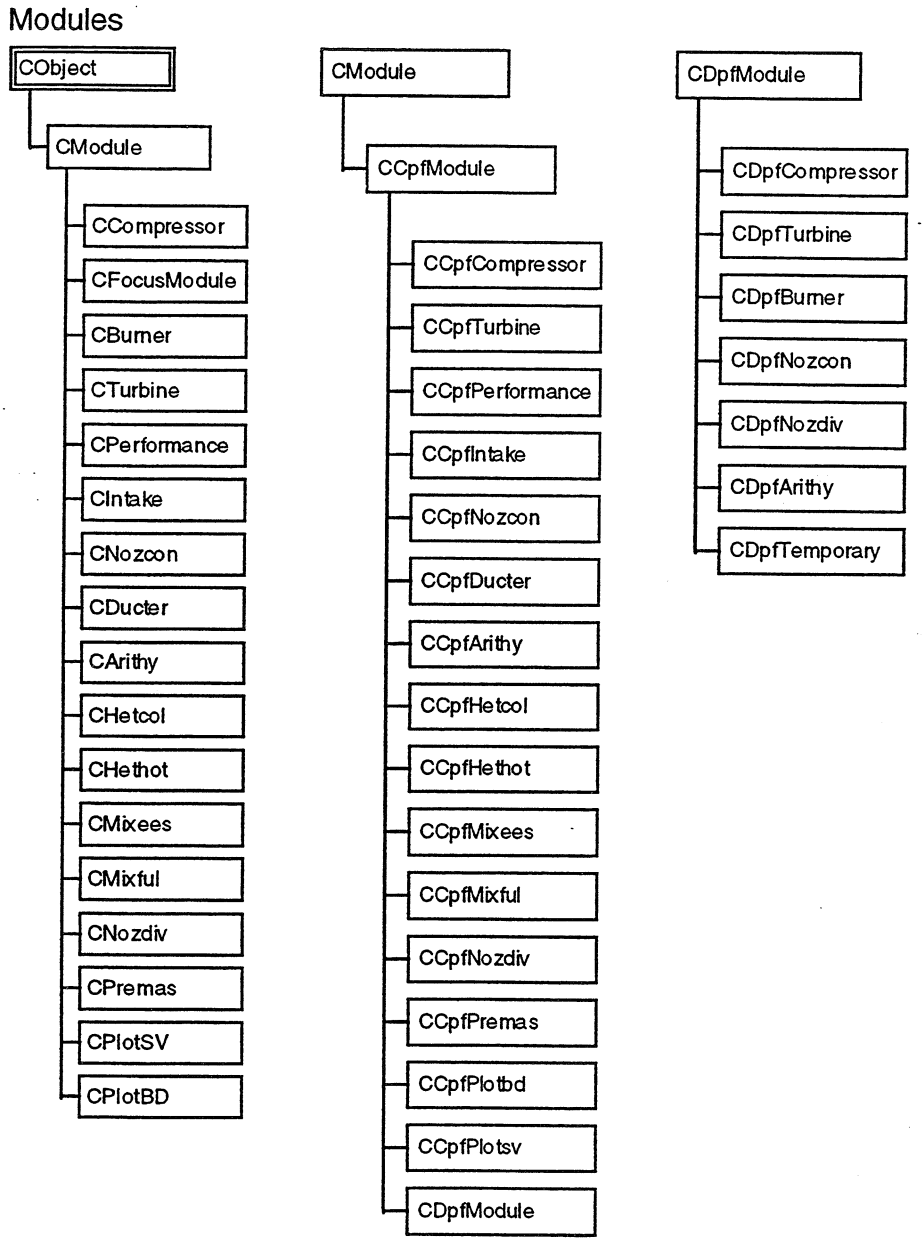


Figure 5.13: Module Class Hierarchy

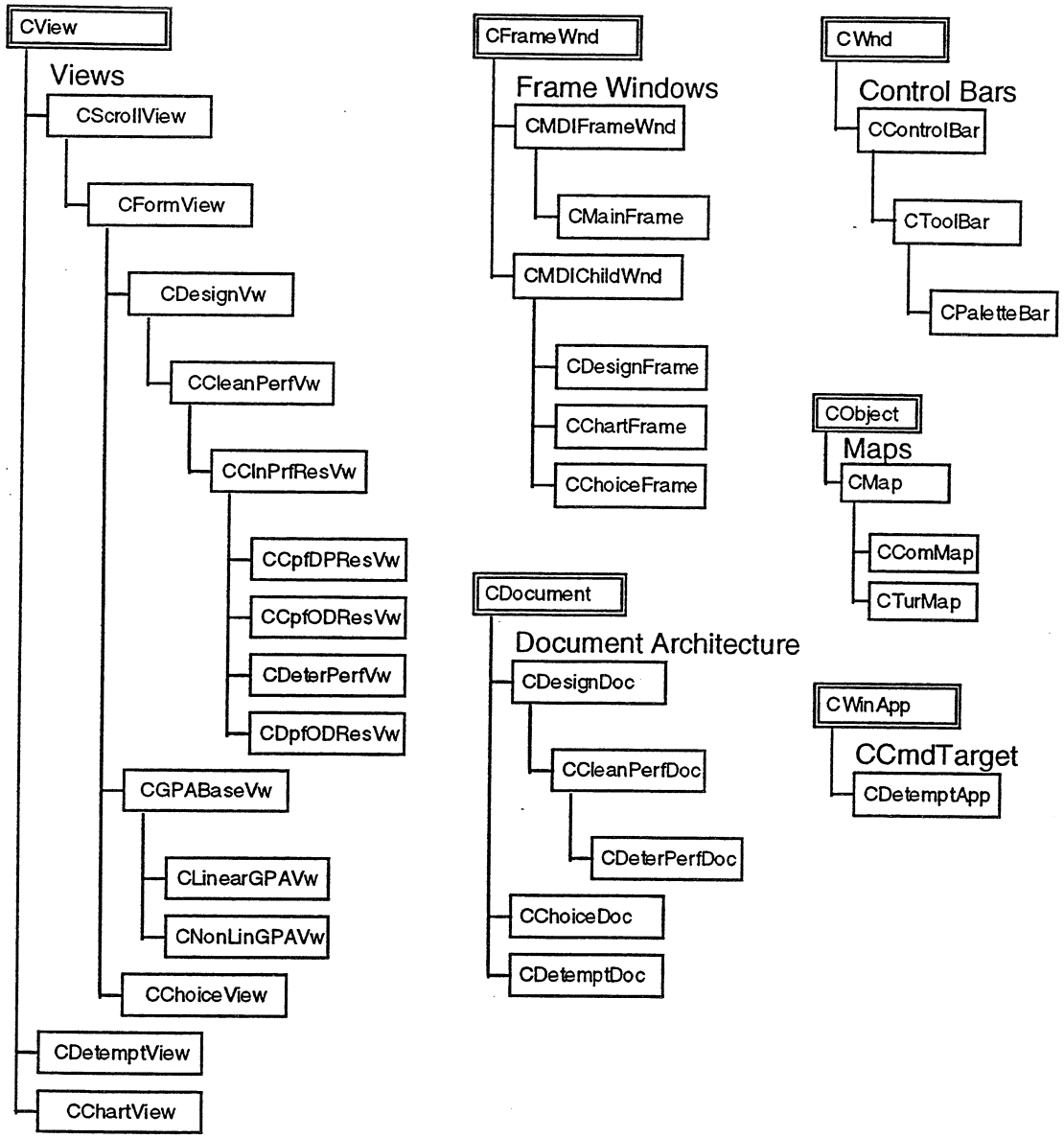


Figure 5.14: Views, Frame windows, Control Bars, Maps, Document Architecture and CcmdTarget Class Hierarchies

The Scaling Factors in Turbomatch

Scaling factors in Turbomatch allow the resizing of component maps in order to carry out clean and deteriorated engine performance studies. In a clean study, defining scaling factors is performed automatically (see Chapter 3). However, in a deteriorated study, scaling factors need to be adjusted according to the changes in independent parameters. Therefore the program Turbomatch has been modified and scaling factors for each compressor, turbine or burner component are stored in specially allocated pre-programmed routine data known as brick data (see Table 5.2 and Palmer 1995).

Table 5.2: Scaling Factors in Turbomatch

Brick	*Scaling Factors*	*Stored in Brick Number*
Compressor	Pressure ratio (PRCSF)	809 + # of Compressor*
	Isentropic efficiency (ETACSF)	819 + # of Compressor*
	Non-dimensional mass flow (WACSF)	829 + # of Compressor*
Turbine	Non-dimensional mass flow (TFTSF)	839 + # of Turbine*
	Isentropic efficiency (ETATSF)	849 + # of Turbine*
	Enthalpy drop (DHTSF)	859 + # of Turbine*
Burner	Combustion efficiency (BURSF)	870

* # means the number of the component

For example, the pressure ratio scaling factor of the compressor fan (Compressor number 1) would be stored in brick data 810 (809 + 1). This allows to change the scaling factors manually in the Turbomatch input file. The implementation of scaling factors in the program Turbomatch is shown in Appendix A. In Pythia, the process of modifying scaling factors is included in the interface, i.e. Pythia provides the necessary arithmetic operation in the Turbomatch input file that will modify the scaling factors.

The Pythia Manual

The user manual should provide the program user of Pythia with sufficient information of how to use and how to control the program in an efficient manner. Since Windows programs offer the possibility of help files it has been decided that the manual is part of the program i.e. the user obtains the information on how to use the program by means of standard Windows help menus. In other words, there is no written manual for Pythia. All the necessary information is stored on the computer. The information is designed for people who have a basic knowledge about gas turbine performance and who are familiar with the Windows 3.1 operating system. Appendix D shows and demonstrates how Pythia can be used to perform GPA studies.

PART III

CASE STUDIES

6 Pythia Applied to Industrial and Aero Gas Turbine Engines

The preceding chapters 4 and 5 have described the development of the GPA diagnostics program Pythia. In order to validate the program, Pythia is applied to several different types of engines: (1) a single and a twin spool gas generator engine with free power turbine, (2) a solid shaft single spool engine and (3) a jet engine. The engines referred to are thermodynamically similar to real engines. These engines are based on performance studies at Cranfield and are not based on data from the individual manufacturers. It should be noted that the GPA studies for the 2nd and 3rd type of the engines have been carried out by the authors Mustapha (1995), Waldock (1995) and English (1995). In addition, performance validation of Pythia has been carried out by DuBois (1995) and Pendedekas (1995). However, their results are not included in the thesis since they are only concerned with clean performance modelling. Both authors, DuBois and Pendedekas, compared the results of Pythia against the results of Turbomatch and data published earlier. In all cases, the comparison was satisfactory.

The purpose of this chapter is to show the GPA technique applied to a wide range of engines and to demonstrate the user-friendliness of the program Pythia. Chapter 7 discusses all the results of these studies in the light of potential uses of GPA.

6.1 SINGLE AND TWIN SPOOL GAS GENERATOR WITH FREE POWER TURBINE

For industrial gas turbines, little work has been done on analysing deteriorated engine performance. This is not true for aero gas turbines where the Gas Path Analysis (GPA) technique has been commonly used as part of maintenance practice. However, such studies have been limited to linear GPA. Therefore the GPA program Pythia has been applied to a small and a large industrial gas turbine engine. Both engines are of similar configuration (free power turbine) and they are investigated under the same operating conditions. The strength of linear and non-linear GPA is analysed by imposing typical physical faults such as fouling on compressors and erosion on turbines. As a result Pythia should identify the minimum set of instrumentation for monitoring the gas turbine engine's faults.

For both engines the same series of case studies is applied (see Table 6.1). Basically for each engine two sets of instrumentation are initially established: one can detect fouling

and one can detect erosion. Next the two sets are combined and the new instrumentation is used for identifying single and multiple physical faults. The study is finished with a sensitivity analysis of the instrumentation where changed magnitudes of faults are identified under different operating conditions of the engine.

Table 6.1: Case studies that establish the ideal set of instrumentation for each engine

Case	*Power*	*TET*	*Ambient*	*Instrum.*	*Fouling*	*Erosion*
A	X			var	X	
B		X		var	X	
C	X			var		X
D		X		var		X
E	X			X	X	X
F		X		X	X	X
G	X			X	var	var
H		X		X	var	var
I	X		var	X	X	X
J		X	var	X	X	X
K	var			X	X	X
L		var		X	X	X
M	X			max	X	X
N		X		max	X	X

where the column variables have the following meanings:

Case Specifies the particular case. Each case is subdivided into several sample studies (diagnostic approaches)

Power Specifies whether power output is used as an operating point that is held constant (= handle). It can be one of the following:

- *x* constant power turbine speed, constant power output (= design point value) and variable turbine entry
- *var* power output as specified, constant power turbine speed, and variable turbine entry temperature

TET Specifies whether turbine entry temperature (TET) is used as a handle. It can be one of the following:

- *x* constant power turbine speed, constant TET (= design point value) and variable power output

	var	TET as specified, constant power turbine speed and variable power output
Ambient		Specifies whether ambient temperature is used as a handle
Instrum.		Specifies the set of instrumentation. It can be one of the following:
	var	variable selection of instruments
	x	fixed set of instrumentation
	max	maximum possible instrumentation
Fouling		Specifies whether there is physical fault fouling in compressor
Erosion		Specifies whether there is physical fault erosion in turbine

Boundary Conditions

The series of case studies for each engine is based on the following assumptions and boundary conditions:

- Instrumentation
- Physical faults
- GPA sensitivity
- Non-linear GPA solver options

Instrumentation

The study concentrates on instruments (monitored dependent parameter) that are typical for industrial gas turbine engines (Agrawal 1979, Urban 1974 and Saravanamuttoo 1974). In order to keep the number of monitored parameter as low as possible, the initial study keeps the numbers of dependent and independent parameters equal. In other words, the matrix that solves the set of GPA equations (see Equation 4.7) is square.

Simulations of clean and deteriorated performance require parameters that specify the engine's operating point (handle). Although handles are measured they are not considered as dependent parameters and therefore they do not appear in the GPA set of equations. This study will make use of different handles such as power output, turbine

entry temperature (TET) and ambient temperature. Power output allows the simulation of an engine where a certain power output is required. TET enables to control the engine with fuel flow (Turbomatch does not allow to use fuel flow as a handle). Ambient temperature enables to simulate hot and cold weather conditions.

Physical Faults

Since the most common faults in industrial gas turbines are compressor fouling and turbine erosion (Williams 1981 and Seddigh 1991), the study is aimed at detecting fouling and erosion with an appropriate set of instrumentation. Therefore a fouled compressor is simulated with the following assumed changes in independent parameters:

- Non-dimensional mass flow Γ_C is decreased by 5%
- Isentropic efficiency η_C is decreased by 1% (Saravanamuttoo, 1985)

Similarly an eroded turbine is simulated with the following assumed changes in independent parameters:

- Non-dimensional mass flow Γ_T is increased by 2% (Zhu, 1992)
- Isentropic efficiency η_T is decreased by 1% (Diakunchak, 1992)

GPA Sensitivity

The study of both engines should determine whether the effort of using the non-linear GPA method should be made. Therefore each diagnostics approach in a particular case will carry out a linear and a non-linear GPA calculation. Each calculation will produce independent parameter changes that can be compared against implanted independent parameter changes. A simplified comparison of independent parameters is achieved by looking at their quadratic mean RMS (Root Mean Square, see also Spiegel 1972):

$$RMS = \sqrt{\frac{\sum_{j=1}^{N}(d_j)^2}{N}} \tag{6.1}$$

where N is the number of independent parameters and d the difference between implanted and detected independent parameter changes. In other words the closer the RMS moves to zero the better is the prediction of the considered linear or non-linear GPA method.

Since each case of the study involves a certain number of diagnostic approaches, each case consists of an equal number of RMS factors, that are related to the linear and to the non-linear GPA method. In statistics, designs in which the same subject (approach) is observed under two different conditions (linear or non-linear GPA) are called paired-sample (Norusis, 1993). In order to decide whether the linear is better than the non-linear GPA method, a hypothesis can be formulated that there is no difference between the means of the related samples. Such hypothesis are often called null-hypothesis. To test the null-hypothesis a paired t-test is used. For each pair, the difference between linear RMS and non-linear RMS is computed. If the distribution of the differences is not random (not normally distributed) another test must be used: the sign test. In both tests a probability number p (significance level) is calculated. If the significance level is relatively small ($p<0.1$), then there is enough evidence to reject the null-hypothesis or in statistical terms to state that the difference between the linear and the non-linear GPA methods is statistically significant (2-tail significance), thus the GPA technique showing an improvement with the non-linear solution (1-tail significance).

There are also statistical tests that can be used to test the null hypothesis that data stem from a normally distributed population (Shapiro-Wilks and Lilliefors test). If their observed significance level is small, then the hypothesis can be rejected that the sample is taken from a normal distribution. Consequently, instead of a paired t-test a sign test is used.

Non-linear GPA Solver Options

The iterative process of non-linear GPA uses following options:

- if not otherwise stated, the influence coefficient matrix ICM is created by degrading (perturbing) in turn each independent parameter by $\delta(\Delta x_j) = 1\%$ (see Equation 4.9)
- the damping factor $\lambda = 0.66$ (see Equation 4.12)
- the convergence criteria $\Delta y_{sum} = 0.5$ (see Equation 4.11)

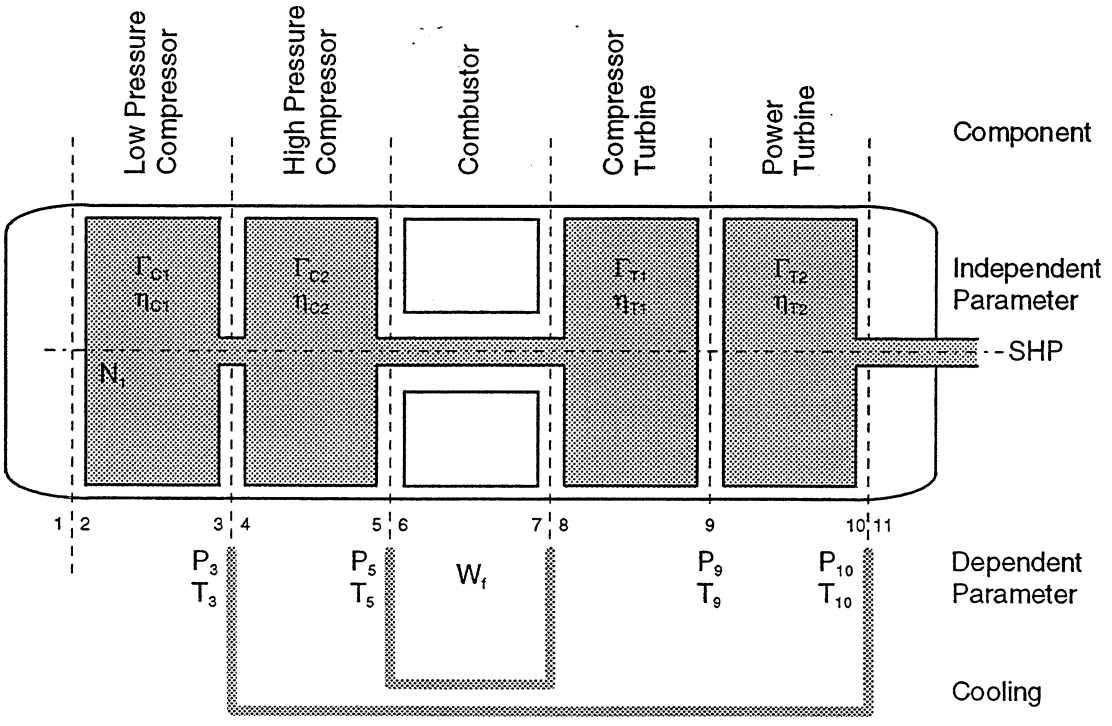
Single Spool Gas Generator Engine with Free Power Turbine

Industrial gas turbines were first introduced in the market in the early 1950's, and were largely an extension of steam turbine design. Restrictions of weight and space were not important factors for these ground based units, and so the design characteristics included heavy-wall casings split on horizontal centrelines, hydrodynamic or hydrostatic bearings, large diameter combustors, thick airfoil sections for blades and stators, and

large frontal areas. The overall pressure ratio of these units vary from 5:1 for the earlier units to 15:1 for the units in present-day service. The auxiliary modules used on most of these engines have gone through considerable hours of testing and are heavy duty pumps and motors, frequently duplicated to avoid unnecessary engine stoppages. One possible layout of such engines is the single spool gas generator with free power turbine.

Modelling

The single spool gas generator engine with free power turbine has a compressor, a combustor, a compressor turbine and a power turbine. The compressor turbine drives the compressor. An example is the STB5000S engine. This particular engine is thermodynamically similar to the Ruston TB5000 (see Figure 6.1). The engine generates 3.8 MW at design point and runs with a thermal efficiency of 26.5%. It passes 20.78 kg/s of air through the compressor with a pressure ratio of 7.3. The turbine entry temperature is about 900^{0}C. Such an engine is favoured in pipeline operations and peak power generator stations.



Key: Γ denotes non-dimensional massflow, η denotes efficiency, SHP denotes shaft horse power, N denotes spool speed, P denotes total pressure, T denotes total temperature and w_f denotes fuel flow.

Figure 6.1: Schematic Layout of STB5000S

The modelling of the STB5000S engine in Pythia is based on several assumptions. Overall performance data where obtained from Mohammed (1989). The components performance levels were evaluated by making reasonable assumptions and modifying them until close agreement with the overall performance data were achieved. The following list shows some of the assumptions:

Cooling — Turbine disc and stack structure cooling is modelled by extracting 3% of the compressor mass flow at intermediate pressure and 2% of the compressor exit mass flow is used to cool the turbine discs and turbine nozzle guide vanes.

Exhaust — Since Pythia is originally based on aero-type gas turbine engines the exhaust stack of the industrial gas turbine is simulated by assuming an imaginary nozzle with a fixed area.

Component Characteristics — Since component characteristics are proprietary to the manufacturer, general maps are used and scaled to meet the corresponding design point values (see Appendix A and Figures 3.3 and 3.4 in Chapter 3)

Other Specification — No mechanical losses or any power extraction for auxiliaries is taken into consideration. All bleed valves are closed and variable inlet guide vanes are fixed. A detailed description of the STB5000S can be found in Appendix C.

Results

A summary of the STB5000S's GPA study is shown in the Figures 6.2 to 6.8. The Figure 6.2 is explained in more detail but all the given information is valid for the Figures 6.3 to 6.8 and for the Figures 6.10 to 6.17.

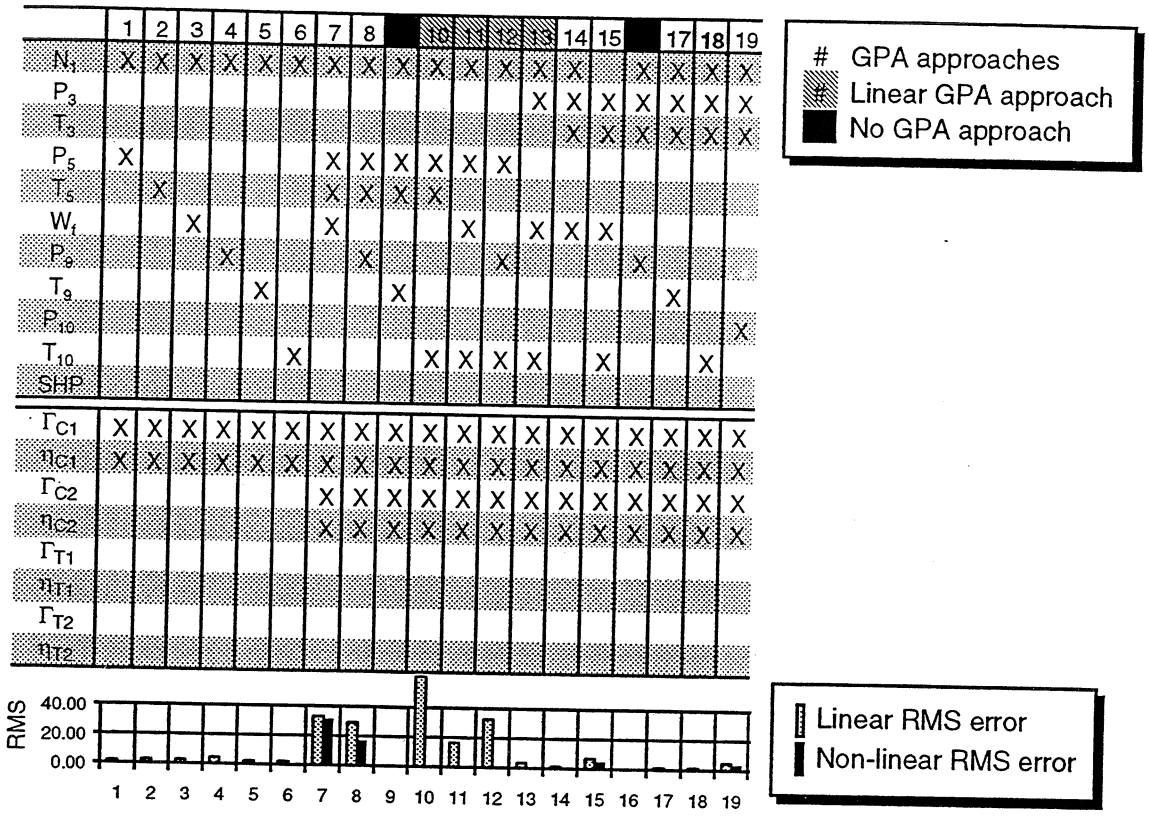


Figure 6.2: Diagnostics Approaches in the Case A (STB5000S)

Each of the figures 6.2 to 6.8 and 6.10 to 6.17 describes a set of diagnostic approaches for a particular case. A diagnostic approach is represented by a numbered column, where the (x)s mark the monitored dependent parameters and the single/multiple physical faults are diagnosed as described in the previous section. Furthermore, at the bottom of each figure the resulting RMS error for linear and non-linear GPA calculation is shown. Approaches with a column number on a white background indicates a solution using both, linear and non-linear GPA. Approaches, where the non-linear GPA solution did not converge are shown with a column number on a grey background. Approaches with no GPA solutions are shown with a black background (obviously in these latter approaches there are no linear or non-linear RMS errors shown). It should be noted that in most approaches the non-linear RMS error is so small that it does not appear in the figure. There are a few approaches, where the non-linear RMS error is higher than the linear RMS error. However, choosing a lower convergence criteria will reduce the non-linear RMS error to a value lower than the linear RMS error (at the expense of increased computing time). N.B.: In this study all approaches used the convergence criteria $\Delta y_{sum} = 0.5$. The statistical analysis is shown in Table 6.2. The study is separated into the effects of instrumentation, the effects of multiple faults, the effects of different operating point settings and the effects of having a full set of instrumentation.

Table 6.2: Statistical Analysis of RMS errors (STB5000S)

Case	*Saphiro*	*Lilliefors*	*t-test (2-tailed)*	*sign test (2-tailed)*	*Significance p (1-tailed)*
A	< 0.01	0.000		0.000	< 0.01
B	0.310	0.200	0.118		< 0.10
C	< 0.01	0.000		0.049	< 0.05
D	0.010	0.123		0.286	> 0.10
E	0.470	> 0.2	0.002		< 0.01
F	0.880	0.200	0.122		< 0.10
G	0.540	0.200	0.000		< 0.01
H	0.230	0.040	0.000		< 0.01
I	0.240	0.200	0.004		< 0.01
J	0.500	0.200	0.002		< 0.01
K	0.340	0.200	0.039		< 0.20
L*				0.125	< 0.10

*Sample size too small

Effects of Instrumentation

Cases A to D (Figures 6.2 to 6.5) deal with an arbitrary variation of the instrumentation set. Fouling is diagnosed in the cases A and B and erosion in the cases C and D. For the cases A and C, TET is used as a handle and for the cases B and D, power output served as the operating point (see Table 6.1).

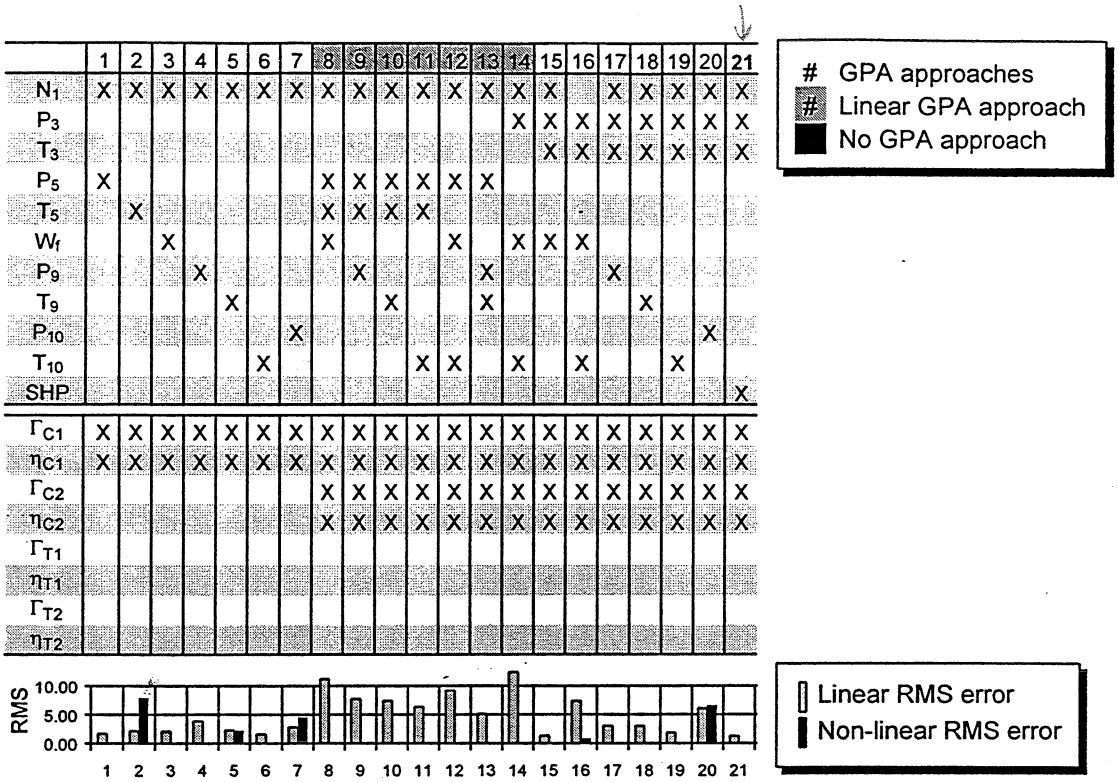


Figure 6.3: Diagnostics Approaches in the Case B (STB5000S)

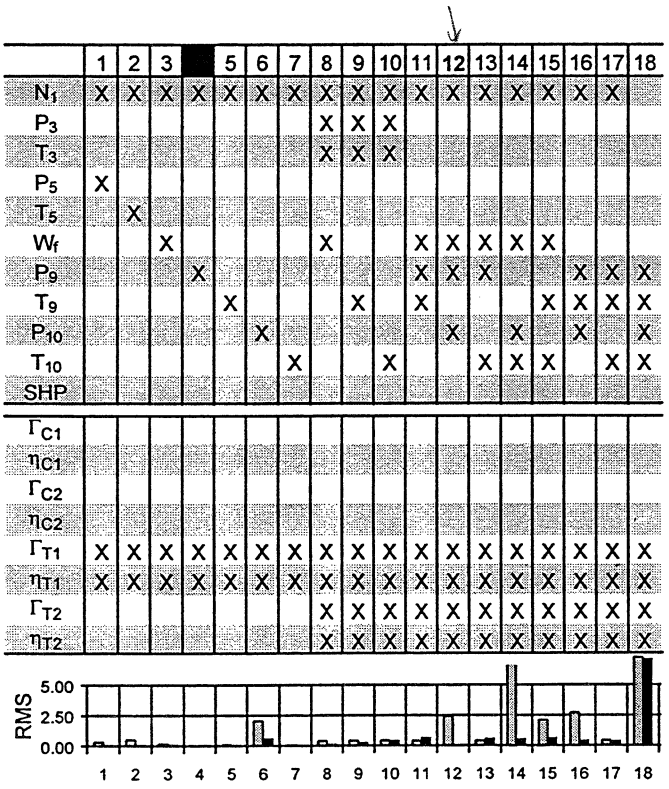


Figure 6.4: Diagnostics Approaches in the Case C (STB5000S)

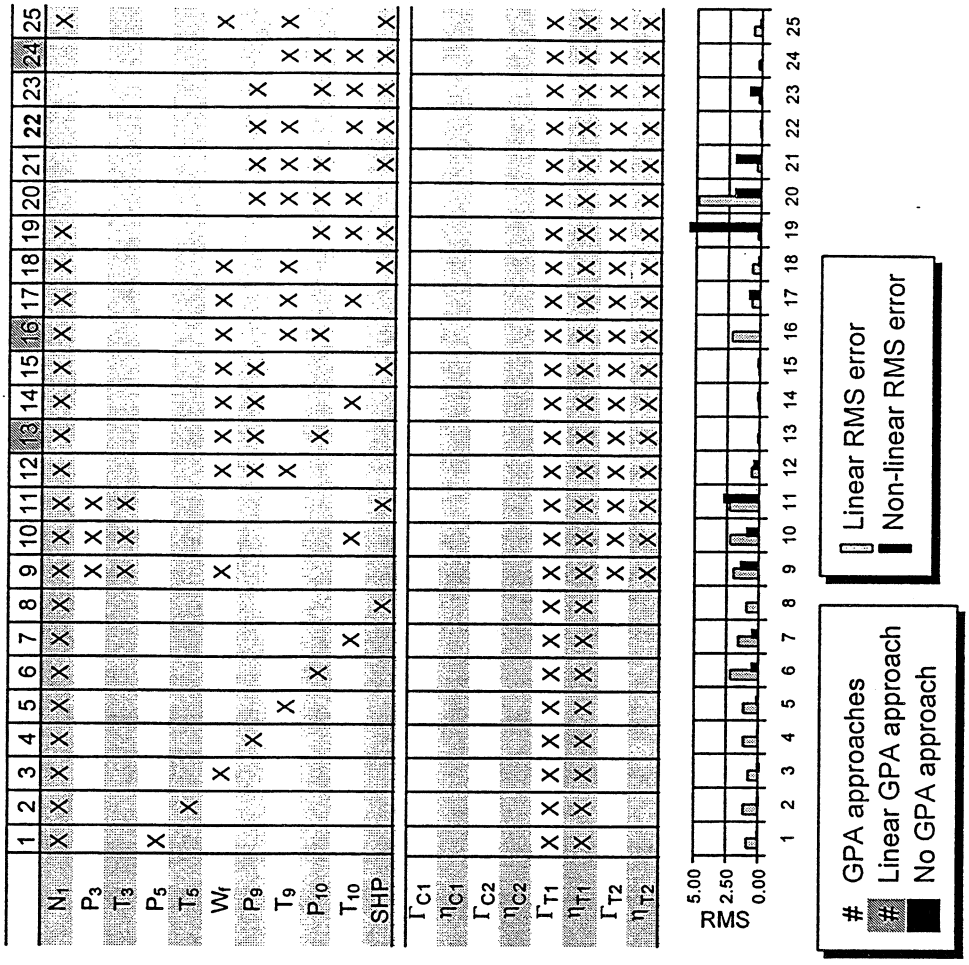


Figure 6.5: Diagnostics Approaches in the Case D (STB5000S)

The GPA technique showed statistically significant improvement with the non-linear solution for the cases A, B and C (see Table 6.2). Typically, the non-linear GPA solution was achieved after 5 iterations. In terms of computing time the non-linear solution was resolved after 30 seconds on a 486 PC running at 66 MHz clock speed. However, for some approaches the non-linear solution did not converge ($A_{10\text{-}13}$, $B_{8\text{-}14}$, $D_{13,16}$ and D_{24}). A closer look at their FCM reveals large coefficients for the linear GPA solution. Examples of FCM with 'normal' and large coefficients are given in Tables 6.3 and 6.4. The large coefficients indicate that a change in measured parameters would lead to a large change in independent parameters. In other words, the sought independent parameters are either weakly or not affected by the chosen measured parameters (Urban, 1975).

Table 6.3: A Non-Acceptable FCM

	ΔN_1	ΔP_5	ΔT_5	ΔT_{10}
$\Delta\Gamma_{C1}$	-3.38	112.05	-132.42	80.77
$\Delta\eta_{C1}$	-0.88	237.47	-303.13	177.48
$\Delta\Gamma_{C2}$	3.03	-322.37	391.31	-236.65
$\Delta\eta_{C2}$	-0.68	-256.28	336.53	-195.82

Table 6.4: An Acceptable FCM

	ΔN_1	ΔP_3	ΔT_3	ΔT_{10}
$\Delta\Gamma_{C1}$	-1.38	0.41	-0.07	-0.41
$\Delta\eta_{C1}$	0.05	1.46	-2.89	0.12
$\Delta\Gamma_{C2}$	-1.20	-1.46	1.46	-0.64
$\Delta\eta_{C2}$	-0.29	-1.81	4.26	-2.19

The reason why some of the approaches for a fouled compressor did not converge, can be seen quite clearly from the aero-thermo relationships. In order to distinguish between the independent parameters Γ_{C1} and η_{C1} the approach should have involved the measurements P_3 and T_3. The measurement P_3 is related to the non-dimensional massflow via the compressor characteristics and the combination of P_3 and T_3 is related to efficiency via the ratio of ideal and actual work changes. This effect is shown in the approaches $A_{14\text{-}19}$ and B_{15}-B_{21}. In most of these approaches the non-linear method resulted in a smaller RMS error than the linear method except for A_{16} where even the linear GPA did not calculate the independent parameter changes. A possible explanation could be found with the dependent parameter P_9. With power as a handle, Pythia 'matches' the engine for clean and deteriorated performance with constant power output in the free turbine. If we assume that the change in massflow entering the free turbine is negligible then a constant power output requires a constant temperature difference in the free turbine. If we further assume that the entering non-dimensional massflow of the free turbine is constant, then there is hardly any change in the pressure measurement P_9. In other words, the small change in P_9 did not lead to noticeable changes in Γ_{C1} and η_{C1} in the approach A_{16}.

Similarly to the fouled compressor, the sets of instrumentation that detect erosion in turbines can be analysed. In order to distinguish faults in the two turbines, the measurement P_9 should be used. The pressure measurement helps to split the two turbines (pressure ratio split turbines). This is true for some of the approaches such as $C_{11\text{-}13}$, $C_{16\text{-}18}$ and $D_{12,14,15,20,22\text{-}23}$ where the non-linear GPA solution converged and resulted in a low RMS error. Despite the fact that C_4 and D_4 had the same measurement arrangements (including P_9), C_4 did not converge. One possible reason could be the different handle settings. As explained earlier, a constant power output in a deteriorated engine will not change P_9 whereas a constant TET will have some effect on P_9. A missing P_9 measurement resulted in a high RMS error for the non-linear solution in D_{19}. For the approach C_{18}, the non-linear method predicted a high RMS as did the linear method despite the P_9 measurement. One explanation could be the missing rotational speed measurement N_1 since N_1' helps to relate P_9 to the non-dimensional turbine massflow via the turbine characteristics. A combination of P_9 and T_9 could identify independent parameter changes in the turbines (D_{20}, D_{22}, $C_{16\text{-}17}$) since P_9 and T_9 clearly relate to efficiency via the ratio of actual and ideal work and P_9 relates to non-dimensional massflow via the turbine characteristics.

The approaches D_{21} and D_{22} differ only in one dependent parameter, P_{10} and T_{10}. The non-linear solution for D_{22} resulted in a lower RMS than the linear whereas for D_{21} the opposite is true. An improved non-linear solution for D_{21} was achieved by lowering the convergence criteria $\Delta y'_{sum}$. This particular approach is not shown in the figures. However, with an RMS of approximately 2 the approach has sufficiently predicted the implanted independent parameter changes (House, 1992). The threshold where the RMS becomes an acceptable value depends on the maintenance requirements of the considered gas turbine engine (English, 1995).

To summarise, an appropriate set of instrumentation must bear a meaningful relationship to the sought independent parameter changes (Urban, 1975). In approaches where it is not clear whether the set is appropriate Pythia's non-linear GPA method will certainly help to discard unsatisfactory approaches. An unsatisfactory set can be identified by a non-convergence of the non-linear solution and/or large coefficients in the FCM. Appropriate sets of instrumentation for the cases A to D are A_{18}, B_{21}, C_{12} and D_{22}.

Effect of Multiple Physical Faults

Cases E to H (Figures 6.6 to 6.7) deal with different combinations and magnitudes of physical faults. The instrumentation used is a combination of the previously defined

sets that could identify fouling or erosion. In particular the instrumentation for the cases E and G is based on the sets from A_{18} and C_{12}. Similarly, the cases F and H are based on the sets from B_{21} and D_{22}.

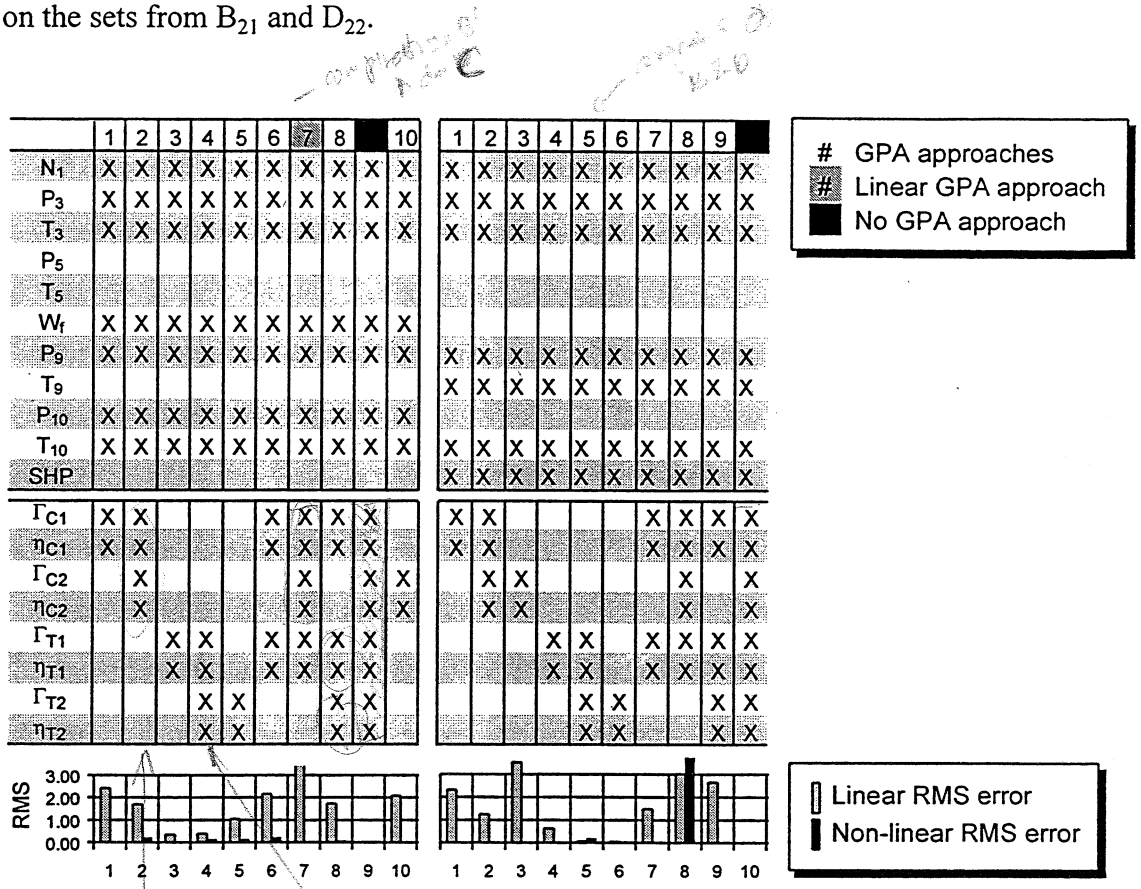


Figure 6.6: Diagnostics Approaches in the Cases E and F (STB5000S)

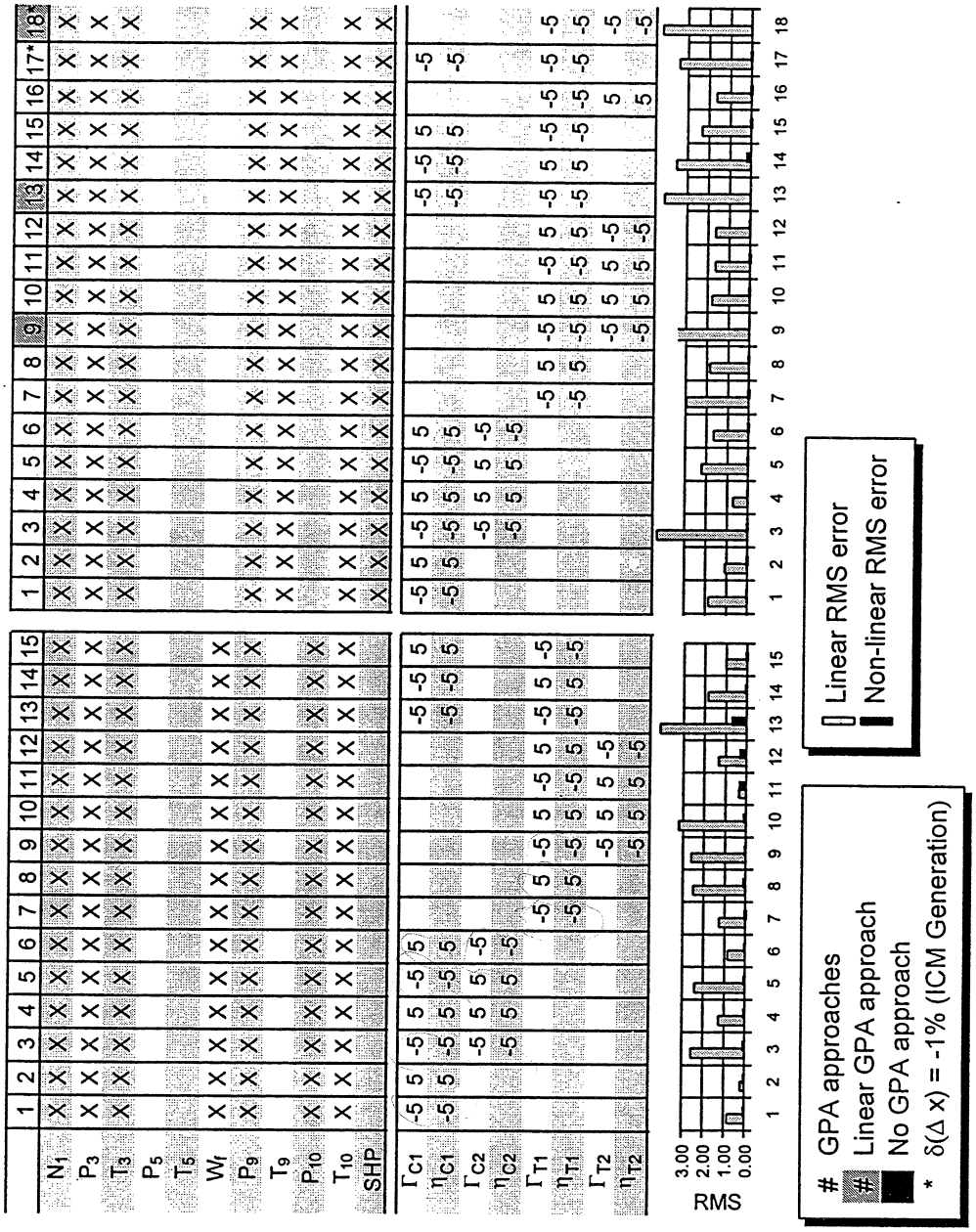


Figure 6.7: Diagnostics Approaches in the Cases G and H (STB5000S)

The GPA technique showed statistically significant improvements with the non-linear solution for all cases (see Table 6.1). The non-linear solution converged for almost all approaches in the cases E and F except in the approaches E_9 and F_{14} where an attempt was made to perform GPA with more independent parameter changes than dependent parameter changes. In other words, GPA had to solve a set of equations with more unknowns than knowns. As a result the non-linear GPA solution did not converge.

Three implanted faults were only identified by the approaches E_8 and F_9. The approaches E_7 and F_8 could not distinguish between fouling in both parts of the compressor and erosion in the compressor turbine.

The cases G and H analysed different magnitudes of changes in implanted independent parameters. In the majority of the approaches the non-linear GPA solution converged. The approaches H_9, H_{13} and H_{18} did not converge because the engine was forced to operate outside its limits. However, a convergence of H_{13} was achieved when the perturbation, that creates the ICM, was set to $\delta(\Delta x_j) = -1\%$ (see Equation 4.9 in Chapter 4).

In summary, if a fault set sought after is known in terms of mathematical signs, then the success of a convergence in GPA can be increased by adjusting the perturbation of the ICM creation.

Effect of Different Operating Point Settings

Cases I to L (Figure 6.8) deal with different handle settings. Cases I and J look at different ambient temperatures and cases K and L vary the handle settings for TET or power respectively. For all cases it was assumed that fouling occurs in the first part of the compressor and erosion in the compressor turbine. The instrumentation used for the cases I,K and J,L is the same as in case E and case F respectively.

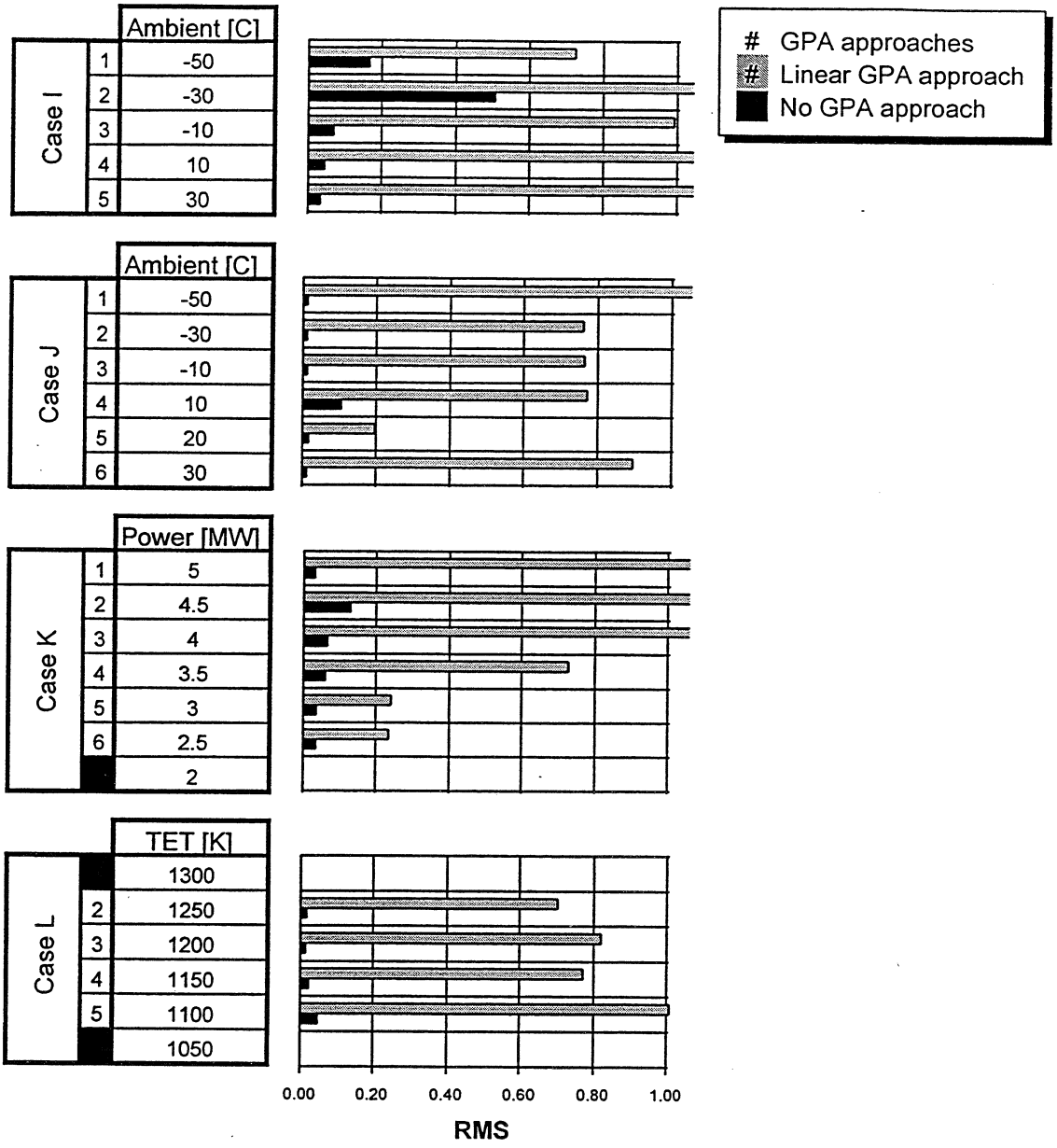


Figure 6.8: Diagnostics Approaches in the Cases I to L (STB5000S)

The GPA technique showed statistically significant improvements with the non-linear solution for all cases (see Table 6.1). No GPA could be carried out for the approaches K_7, L_1 and L_6. The operating settings forced the engine to operate outside its valid range.

Effect of Full Set of Instrumentation

Cases M and N (see Table 6.5) deal with a large number of monitored parameters that were used for the identification of physical faults in all compressor and turbine parts. In

both cases the non-linear GPA method succeeded in predicting the implanted independent parameter changes.

Table 6.5: Diagnostic Approaches in the Cases M and N

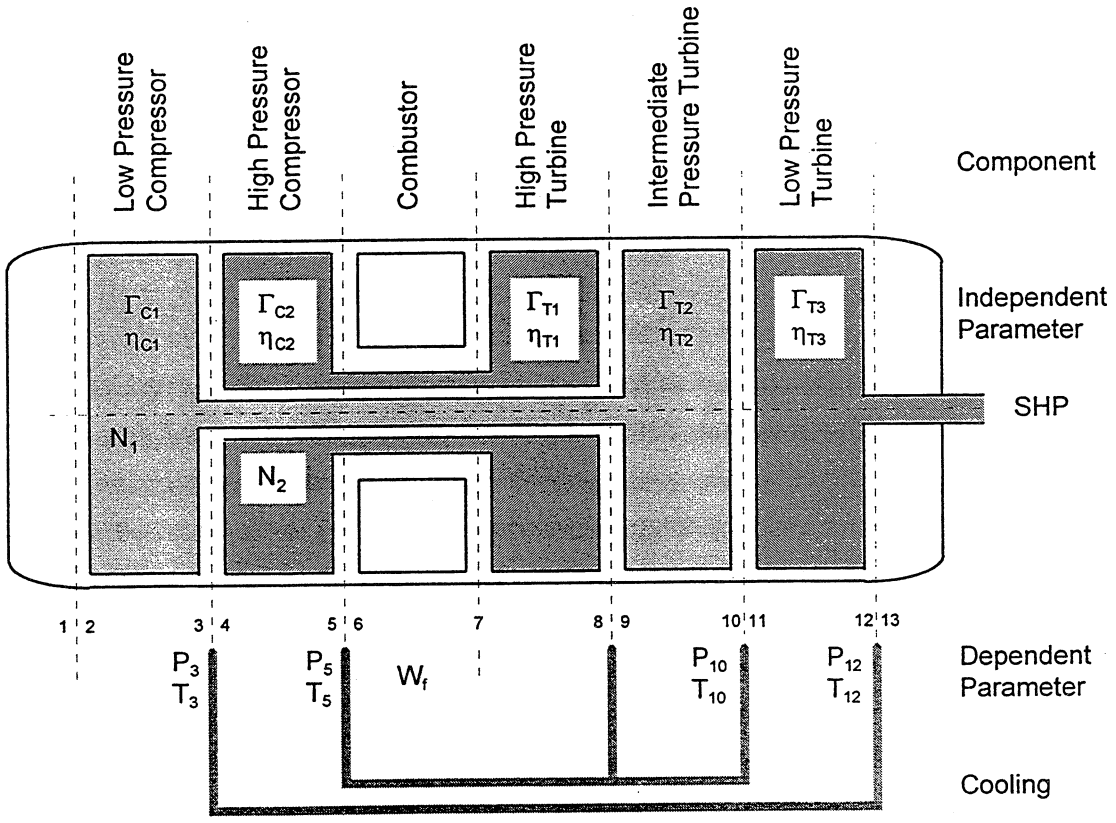
Case	*Dependent parameter*	*Independent parameter*	*Linear RMS error*	*Non-linear RMS error*
M	N_1, P_3, T_3, P_5, T_5, w_f, P_7, T_7, P_9, T_9, P_{10}, T_{10}, P_{12}, T_{12}	Γ_{C1}, η_{C1}, Γ_{C2}, η_{C2}, Γ_{T1}, η_{T1}, Γ_{T2}, η_{T2}	1.36	0.06
N	N_1, P_3, T_3, P_5, T_5, w_f, P_7, P_9, T_9, P_{10}, T_{10}, P_{12}, T_{12}, SHP	Γ_{C1}, η_{C1}, Γ_{C2}, η_{C2}, Γ_{T1}, η_{T1}, Γ_{T2}, η_{T2}	1.46	0.01

Twin Spool Gas Generator With Free Power Turbine

The aero derivative gas turbine engine consists of two basic components: an aircraft derived gas generator and a free power turbine. The gas generator is derived from an aircraft engine modified to burn industrial fuels at ground level for prolonged periods. The gas generator serves as a producer of gas energy or horsepower, which is converted to mechanical rotative energy by the free power turbine. The aircraft type turbine is mostly used by gas transmission companies, gas reinjection plants, and electricity peak topping generation. These account for about 80% of all sales of this type of unit. The benefit of aero derivative engines is a favourable installation cost. Generally, the development cost of the aero engine gas generation is passed on to the aero customers, so that the unit cost of the gas generator is lower than that of the industrial gas turbine engine designed to customer order. Also, as the gas generator was designed for use in aircraft propulsion, the specific work of the engine is high, being achieved using high pressure ratios (around 30:1) and high turbine entry temperatures (around 1200^0 C). The compact accessory equipment also developed for installation in the aircraft is coupled with the gas generator. Aero engines typically employ rolling element bearings that are small and require very little, low pressure oil in comparison to the industrial systems although offering a finite life. This results in considerably simplified oil accessory systems. Consequently, the aero derivative engine package is typically smaller, which facilitates factory testing and entails reduced commissioning costs. One possible layout of such engines is the twin spool gas generator with free power turbine.

Modelling

The twin spool gas generator engine with free power turbine has two compressors and 3 turbines. The high pressure (HP) turbine drives the high pressure compressor and the intermediate pressure turbine drives the low pressure (LP) compressor. An example is the SRB211S engine. This particular engine is thermodynamically similar to the aero-derivative Rolls-Royce RB211 (see Figure 6.9). The engine generates 24.45 MW at design point. It passes 89.9 kg/s of air through the LP compressor with a pressure ratio of 4.472. The HP compressor has the same pressure ratio as the LP compressor. TET is about 1160^{0}C. Such an engine is favoured in pipeline operations, peak power generator stations and marine applications.



Key: Γ denotes non-dimensional massflow, η denotes efficiency, SHP denotes shaft horse power, N denotes spool speed, P denotes total pressure, T denotes total temperature and w_f denotes fuel flow.

Figure 6.9: Schematic Layout of SRB211S

Key to Figure 6.9: Γ denotes non-dimensional massflow, η denotes efficiency, SHP denotes shaft horse power, N denotes spool speed, P denotes total pressure, T denotes total temperature and w_f denotes fuel flow.

The modelling of the SRB211S engine using Pythia is based on several assumptions (Mohammed 1989 and Connoly 1985):

Cooling — Turbine disc and stack structure cooling is modelled by extracting 3% of the compressor massflow at the exit to the low pressure compressor. 7% of the high pressure compressor exit massflow is used to cool the turbine discs and blades. However, the mixing of cooling air with hot gases is different to the STB5000S. This is because each turbine is single stage. This arrangement of stages allows the assumption that the air cooling the nozzle guide vanes is not doing any work at this stage therefore permitting its introduction after the stage with negligible error.

Other Specifications — Other assumptions concerning the exhaust and component characteristics correspond with the assumptions made for the Ruston STB5000S. All bleed valves are closed and variable inlet guide vanes are fixed. A detailed description of the SRB211S can be found in Appendix C.

Results

The GPA study for the SRB211S is carried out in the same way as for the Ruston STB5000S. Since the SRB211S is a more complex engine, more dependent and independent parameters are considered. A summary of the study is shown in the Figures 6.10 to 6.17. The statistical analysis is shown in Table 6.6.

Table 6.6: Statistical Analysis of RMS errors (SRB211S)

Case	*Saphiro*	*Lilliefors*	*t-test (2-tailed)*	*sign test (2-tailed)*	*Significance p (1-tailed)*
A	< 0.01	0.000		0.000	< 0.01
B		0.000		0.000	< 0.01
C	< 0.01	0.000		0.201	> 0.10
D	0.190	> 0.2	0.420		> 0.10
E	0.971	> 0.2	0.000		< 0.01
F	< 0.01	0.000		0.000	< 0.01
G	0.247	0.150	0.000		< 0.01
H	0.230	> 0.2	0.000		< 0.01
I	0.178	> 0.2	0.072		< 0.05
J	0.322	> 0.2	0.047		< 0.05
K*				0.125	< 0.10
L	0.062	> 0.2	0.023		< 0.05

*Sample size too small

Effects of Instrumentation

Cases A to D (Figures 6.10 to 6.13) deal with an arbitrary variation of the instrumentation set.

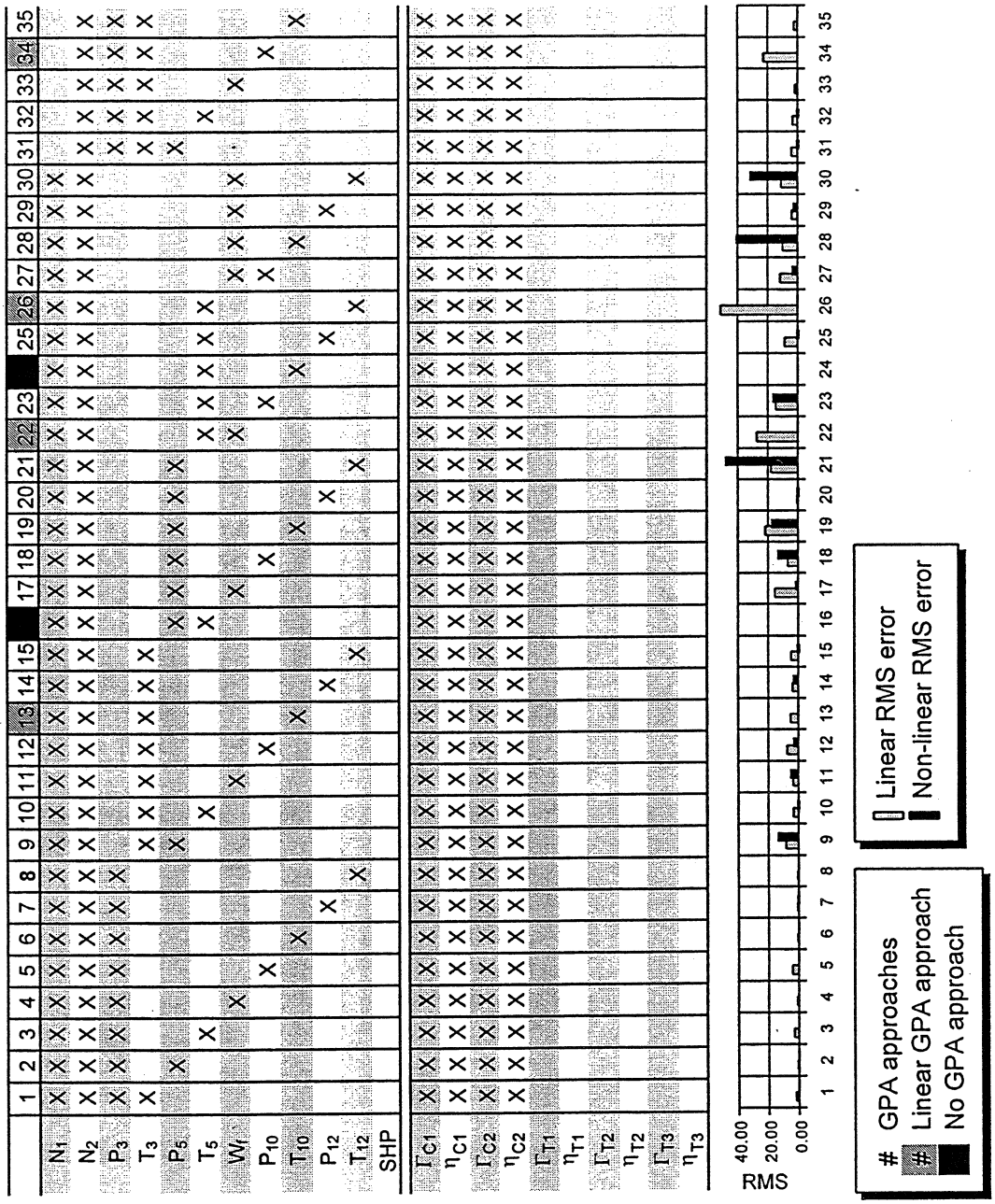


Figure 6.10: Diagnostics Approaches in the Case A (Part I, SRB211S)

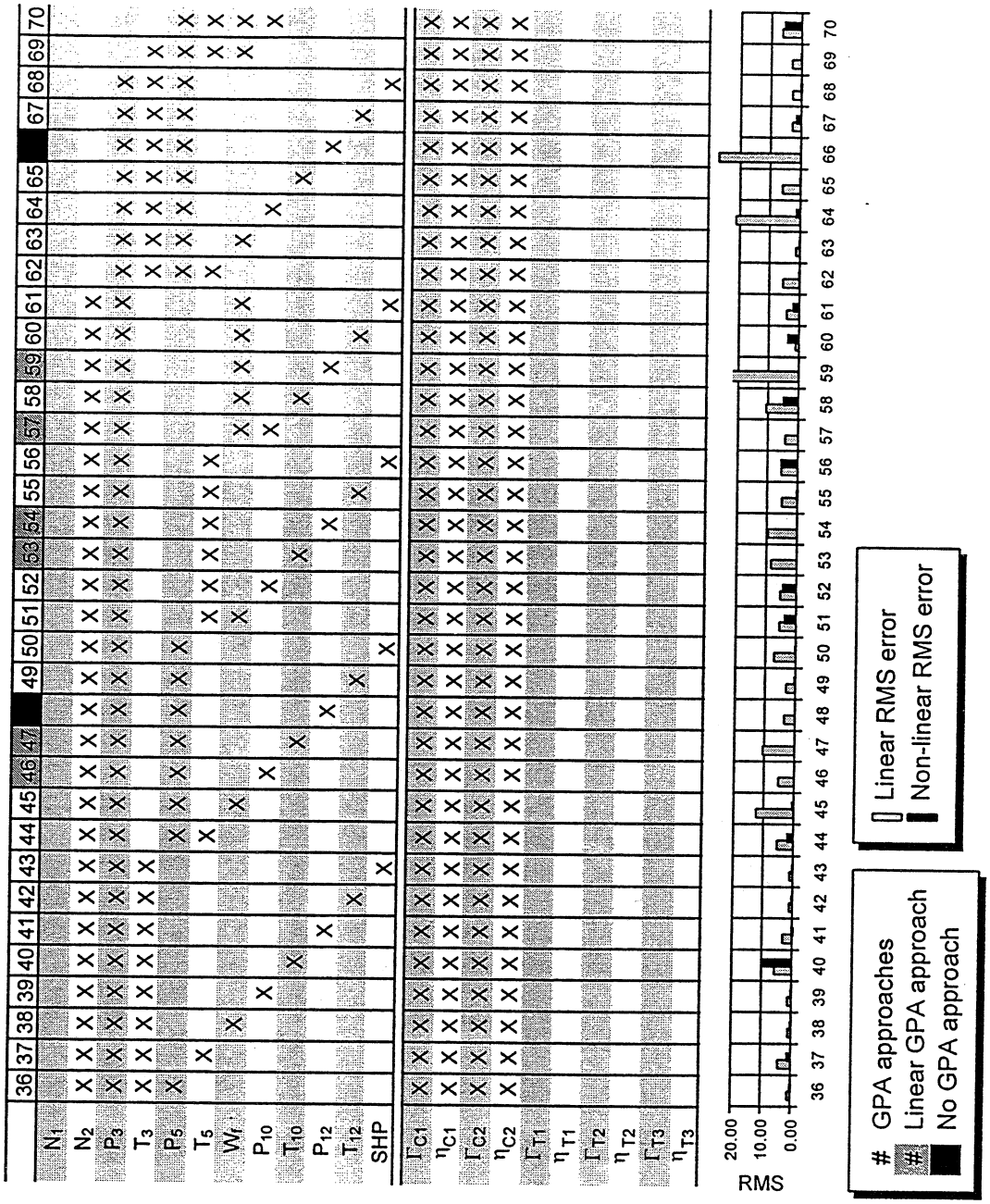


Figure 6.11: Diagnostics Approaches in the Case B (Part II, SRB211S)

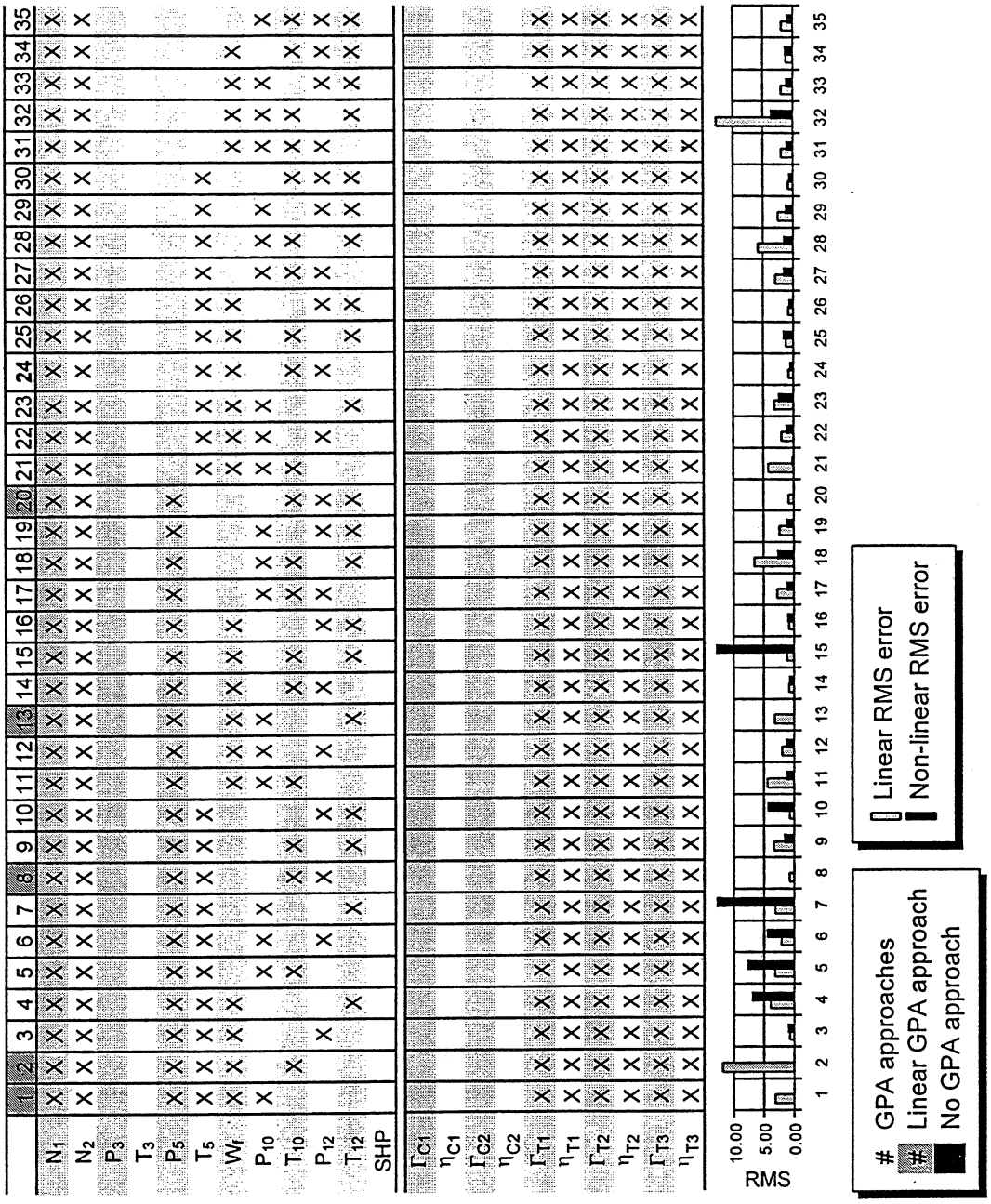


Figure 6.12: Diagnostics Approaches in the Case C (SRB211S)

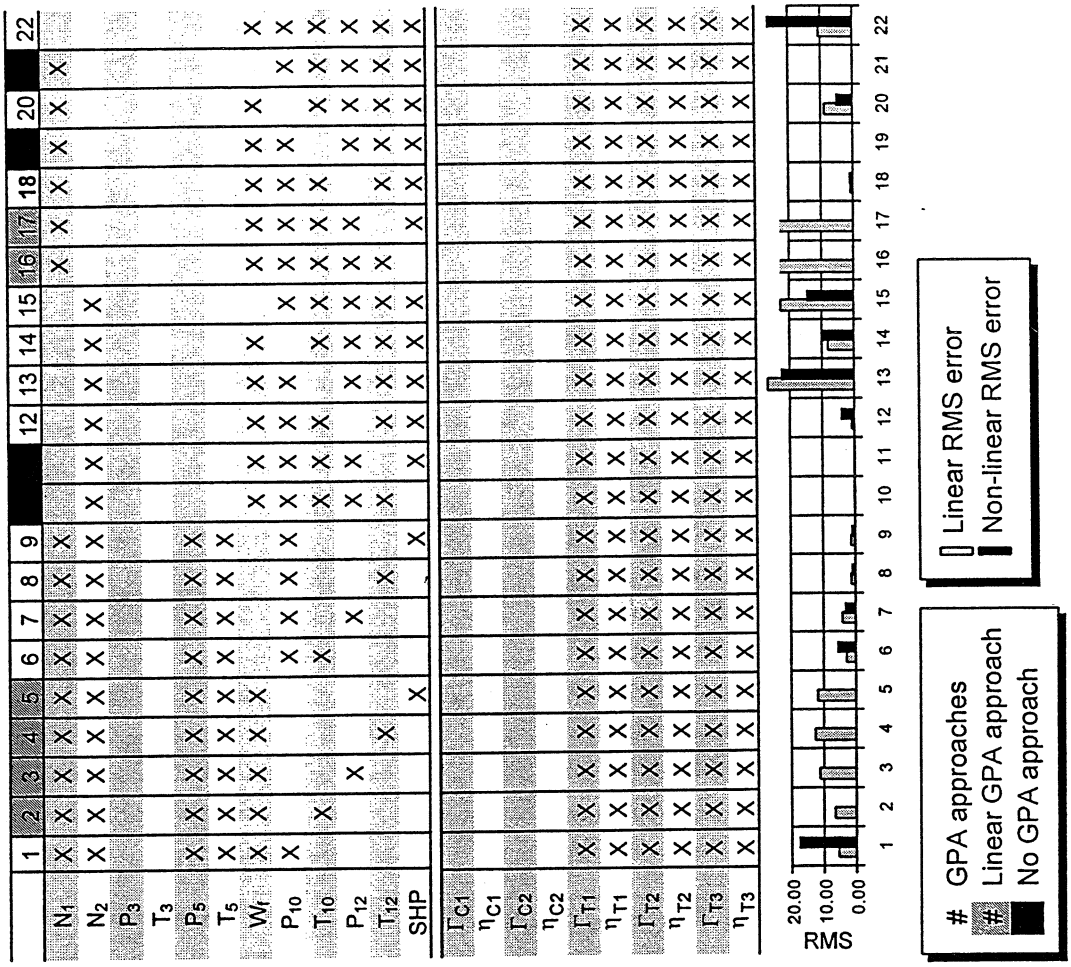


Figure 6.13: Diagnostics Approaches in the Case D (SRB211S)

Only the cases A and B showed statistically significant improvements with the non-linear GPA technique (see Table 6.6). Observations made in the STB5000S study are used to describe the phenomena in the SRB211S. Large coefficients in the FCM resulted in a non-convergence for some of the approaches such as $A_{13,22,26,34}$... etc. Fouled compressors were identified in most of the approaches where P_3 and T_3 was used (A_{11}, A_{31} etc.). An additional measurement P_{10} could only identify compressor problems in the approaches B_{39} and B_{64}. However, in the approaches A_{34}, A_{55} and A_{60}, where Power was used as a handle, GPA did not converge. Eroded turbines were diagnosed with the pressure measurement P_{10} ($C_{11,12}$ etc.). The combination of P_{10} and T_{10} improved the solution of the approaches $C_{11,17,18}$ etc.

In summary, cases A to D for the engines STB5000S and SRB211S analysed a selection of measurements in order to identify a chosen set of physical faults. Although not all possible approaches have been carried out and many other combinations of dependent

parameters could be studied, the present investigation showed that for the choice of instrumentation the measured parameters must bear a meaningful relationship to the independent parameter changes sought after (Urban, 1975). However, in a complex engine such as the SRB211S this is not always straightforward but the GPA technique in Pythia facilitates the identification of a suitable set of instrumentation (A_8, B_5, C_{24}, D_{18}).

Effect of Multiple Physical Faults

Cases E to H (Figures 6.14 to 6.16) deal with different combinations and magnitudes of physical faults. The instrumentation for the cases E and G is based on the sets from A_8 and C_{24}. Similarly, the cases F and H are based on the sets from B_5 and D_{18}.

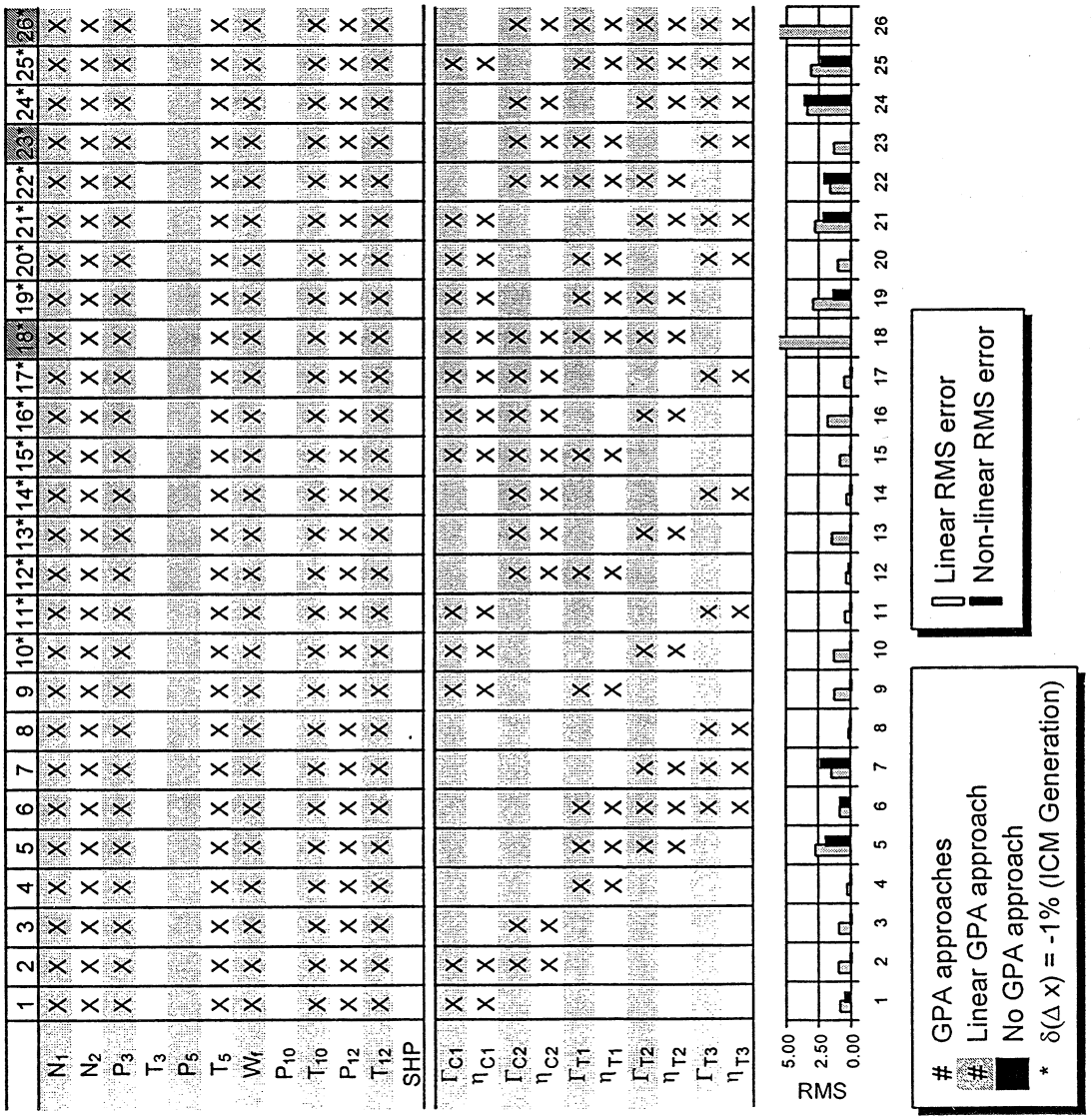


Figure 6.14: Diagnostics Approaches in the Case E (SRB211S)

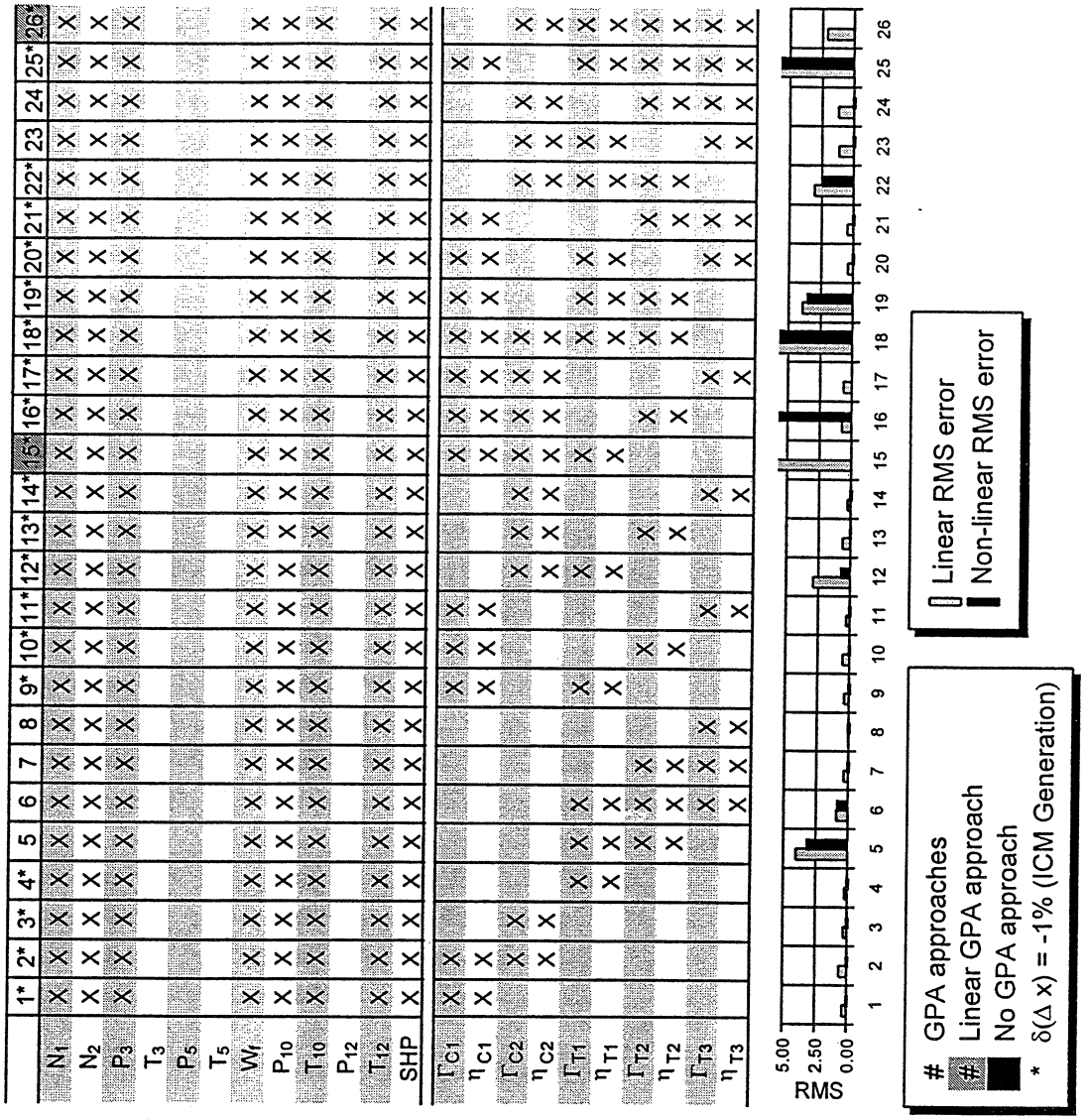


Figure 6.15: Diagnostics Approaches in the Case F (SRB211S)

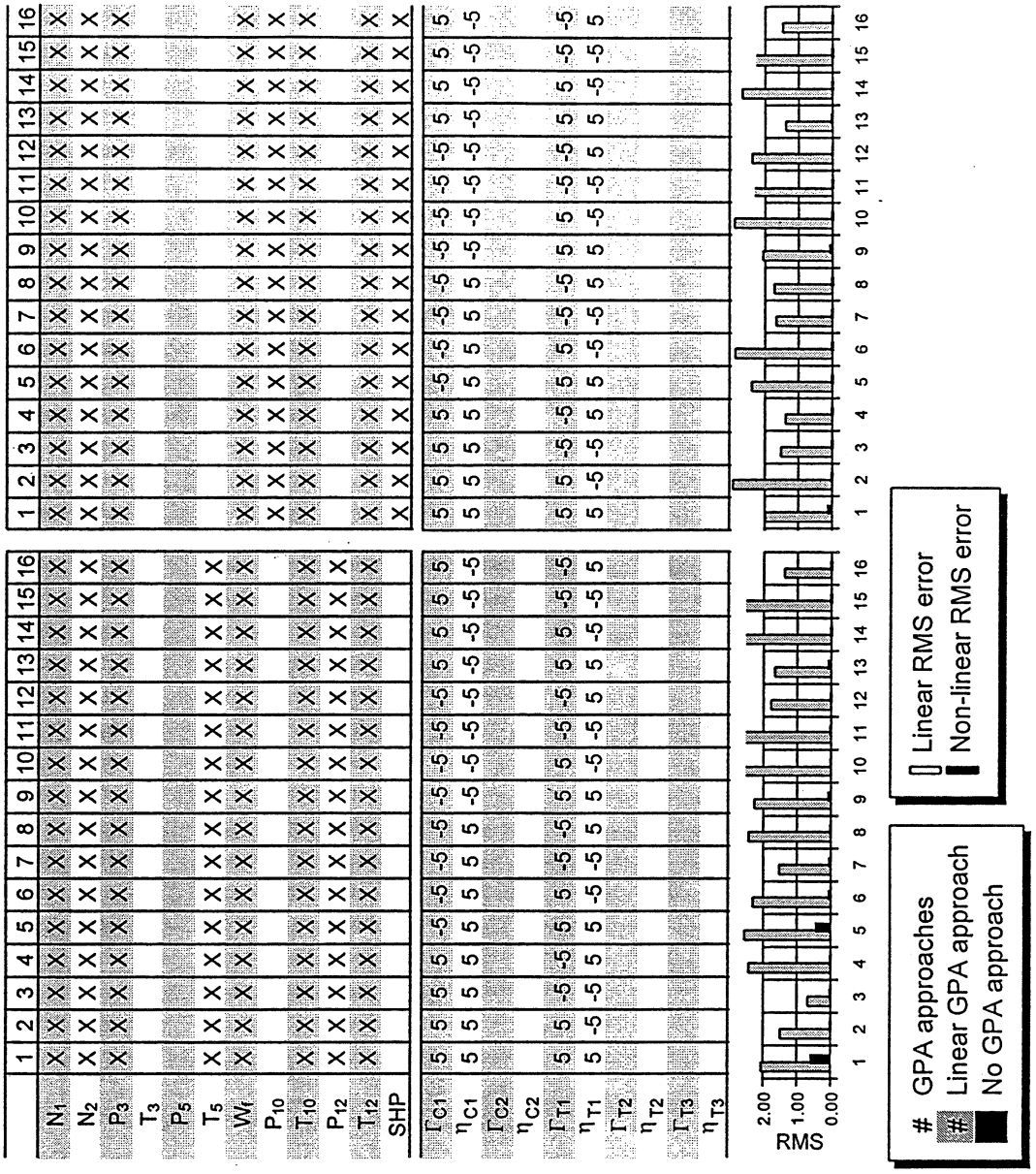


Figure 6.16: Diagnostics Approaches in the Cases G and H (SRB211S)

The GPA technique showed statistically significant improvements with the non-linear solution for all four cases (see Table 6.6).

However, for some approaches the non-linear solution was only achieved by setting the perturbation that creates the ICM, to $\delta(\Delta x_j) = -1\%$ (see Equation 2.4). Typically, with 4 physical faults the non-linear GPA was not successful (see approaches E_{18}, E_{26}, F_{18}, F_{25} and F_{26}). Different magnitudes of implanted faults were diagnosed in all of the approaches in the cases G and H.

Effect of Different Operating Point Settings

Cases I to L (Figure 6.17) show the effects of different handle settings. For all the cases it was assumed that fouling occurs in the low pressure compressor and erosion in the high pressure turbine. The instrumentation used for the cases I,K and J,L is the same as in case E and case F respectively.

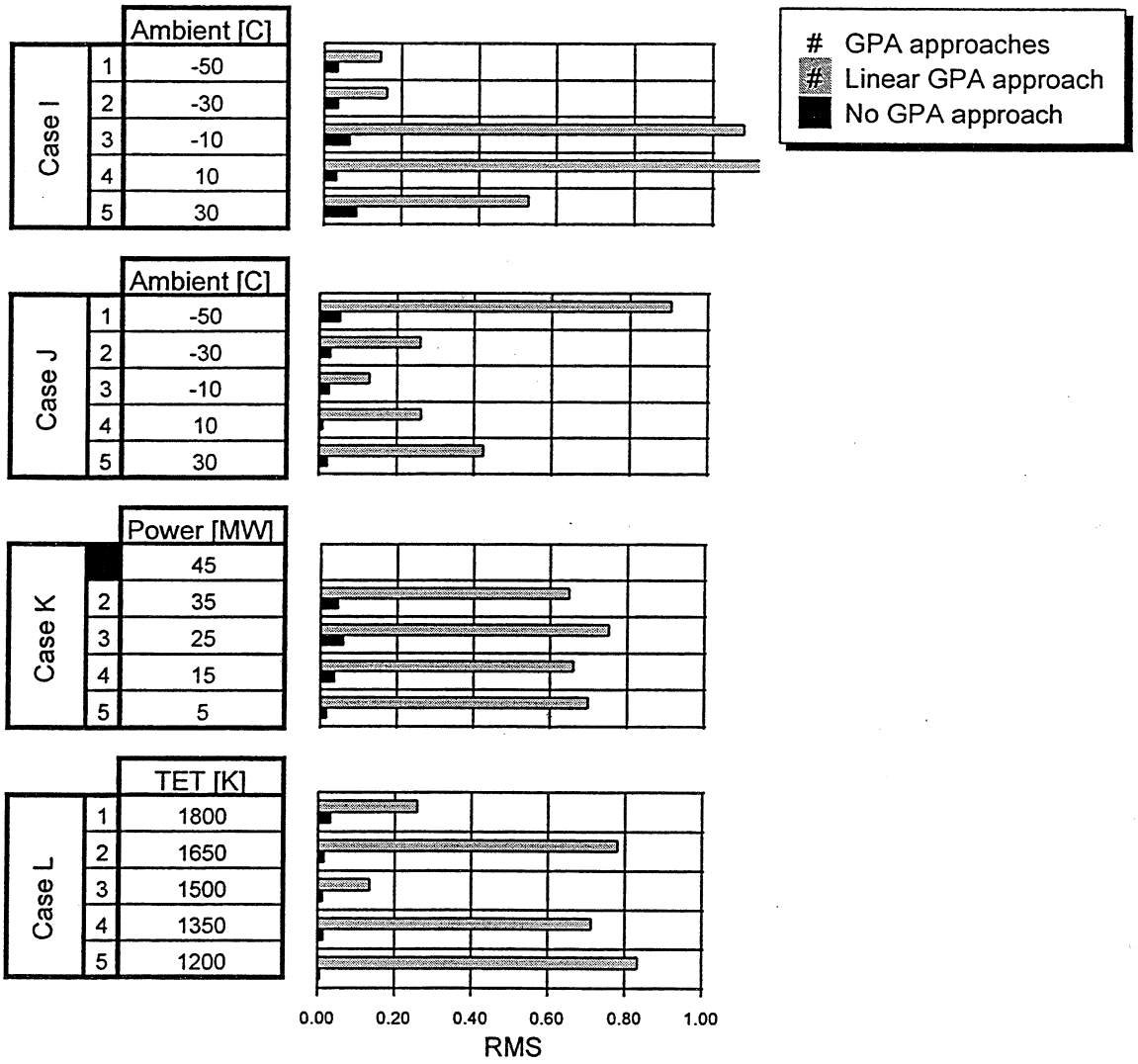


Figure 6.17: Diagnostics Approaches in the Cases I to L (SRB211S)

The GPA technique showed statistically significant improvement with the non-linear solution for all cases (see Table 6.6). For the approach K_1, there is no GPA solution because the operating settings forced the engine to operate outside its valid range.

Effect of Full Set of Instrumentation

Cases M and N show the effect of GPA where 17 dependent parameters were used for the identification of 10 independent parameters (see Table 6.7). In both cases the non-linear GPA succeeded in predicting the implanted independent parameter changes.

Table 6.7: Diagnostic Approaches in the Cases M and N (SRB211S)

Case	*Dependent parameter*	*Independent parameter*	*Linear RMS error*	*Non-linear RMS error*
M	N_1, N_2, P_3, T_3, P_5, T_5, w_f, P_7, T_7, P_8, T_8, P_{10}, T_{10}, P_{12}, T_{12}, P_{13}, T_{13}	Γ_{C1}, η_{C1}, Γ_{C2}, η_{C2}, Γ_{T1}, η_{T1}, Γ_{T2}, η_{T2}, Γ_{T3}, η_{T3}	0.71	0.01
N	N_1, N_2, P_3, T_3, P_5, T_5, w_f, P_7, T_7, P_8, T_8, P_{10}, T_{10}, P_{12}, T_{12}, P_{13}, T_{13}, SHP	Γ_{C1}, η_{C1}, Γ_{C2}, η_{C2}, Γ_{T1}, η_{T1}, Γ_{T2}, η_{T2}, Γ_{T3}, η_{T3}	0.78	0.02

6.2 SOLID SHAFT SINGLE SPOOL ENGINE

Heavy industrial gas turbines are designed for electricity generation and are capable of yielding up to 240 MW per unit. Many heavy-duty units have run well in excess of 100 000 hours and some have exceeded 200 000 hours. It is worth noting that aero-derivative units have exceeded 50 000 hours before overhaul, in pipeline applications. The main driving parameters are low emission, long service life, high reliability and high efficiency. Another significant parameter is the high turbine exhaust temperature. This high temperature allows to use this type of engines for combined cycle power plants i.e. the hot exhaust gas is used to raise steam in a heat recovery steam generator which is then used to drive a steam turbine generator. As a result, more power can be produced for the same heat input and hence high thermal efficiency up to 60%. One possible layout of such engines is the solid shaft single spool engine turbine (see Figure 6.18).

Modelling

The solid shaft single spool engine has a compressor, a combustor and a single turbine which drives both the compressor and the load (see Figure 6.18). An example is the SGE MS9001ES engine. This particular engine is thermodynamically similar to the General Electric MS9001E. The engine generates 117 MW at design point and runs with a thermal efficiency of 33 %. It passes 408 kg/s of exhaust at 529^{0}C. The turbine

entry temperature (TET) is about 1104°C and the compressor pressure ratio is 12.1. Such an engine can be used for a combined cycle power plant.

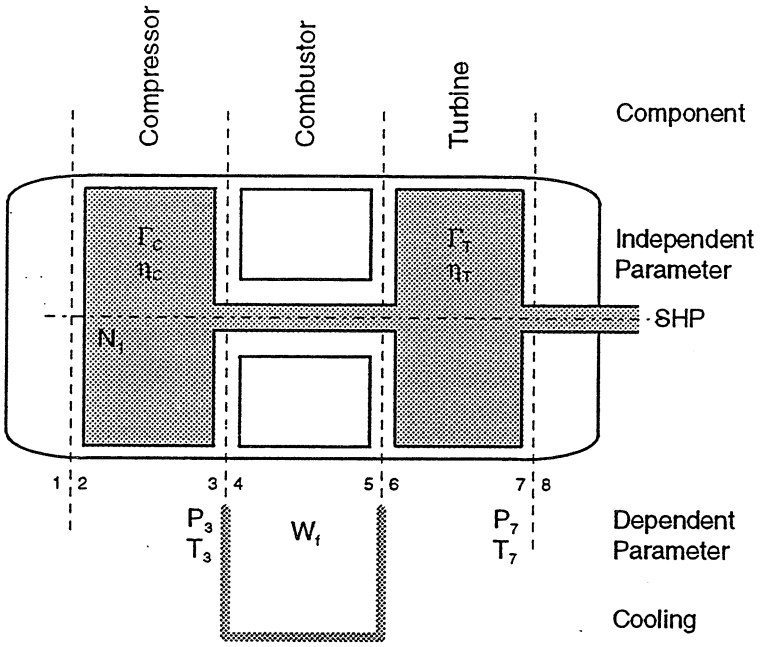


Figure 6.18: Schematic Layout of SGE MS9001ES

Results

The GPA study of the SGE MS9001ES has been carried out by Mustapha (1995). The purpose of the study was to analyse various types of diagnostics approaches to the SGE engine with the program Pythia. In particular, the diagnostics studied the effects of changed ambient temperature (+30°C) and the effects of intake filter deterioration (5%) on different sets of instrumentation. In both cases the set of instrumentation were used for the identification of fouling in the compressor and/or erosion in the turbine with either TET or Power as a handle. Fouling was simulated by a 5% decrease in compressor flow capacity (independent parameter change) and by a 2% decrease in compressor efficiency. Erosion was simulated by a 2% increase in turbine flow capacity and by a 1% decrease in turbine efficiency. It was observed that in most approaches the resulting non-linear RMS error was lower than the linear RMS error. In some of the approaches where either compressor exit temperature or pressure measurement was used, the non-linear solution did not converge. Similarly, the non-linear solution was not achieved when the set of measurements did not include turbine exit temperature measurements. An increase in magnitude of implanted independent parameter

changes increased the linear RMS error but not the non-linear RMS error. The non-linear RMS error was kept comfortably below the value of 0.5. The study concludes that fouling in the compressor was best identified with compressor exit pressure and temperature measurements. Fouling in the compressor and erosion in the turbine should require fuel flow and compressor exit pressure measurements. In addition, the GPA calculation showed lower RMS values for the approaches where TET was used as a handle.

6.3 SINGLE SHAFT TURBOPROP ENGINE

The greatest impact of the gas turbine has been in the field of aircraft propulsion. The gas turbine has been used for subsonic and supersonic speeds. For low-speed aircraft a combination of propeller and exhaust jet provides the best propulsive efficiency. One possible layout of such an engine is the single shaft turboprop engine (see Figure 6.19).

Modelling

The single shaft turboprop engine has a compressor, a combustor and a turbine which drives both the compressor and the propeller via a gearbox (see Figure 6.19). An example is the SAllison T56S engine. This particular engine is thermodynamically similar to the Allison T56. The engine produces about 3 MW of shaft power at the design point. It passes 15 kg/s of air through the compressor with a pressure ratio of about 9. TET is about 1010^0C. The propeller speed is controlled via the fuel system. Maximum power settings allow the propeller to rotate at 13 820 RPM. Since Turbomatch does not offer to use fuel flow as a handle, TET is used instead. This restriction limits the number of existing measurements to two: fuel flow (w_f) and shaft horse power (SHP).

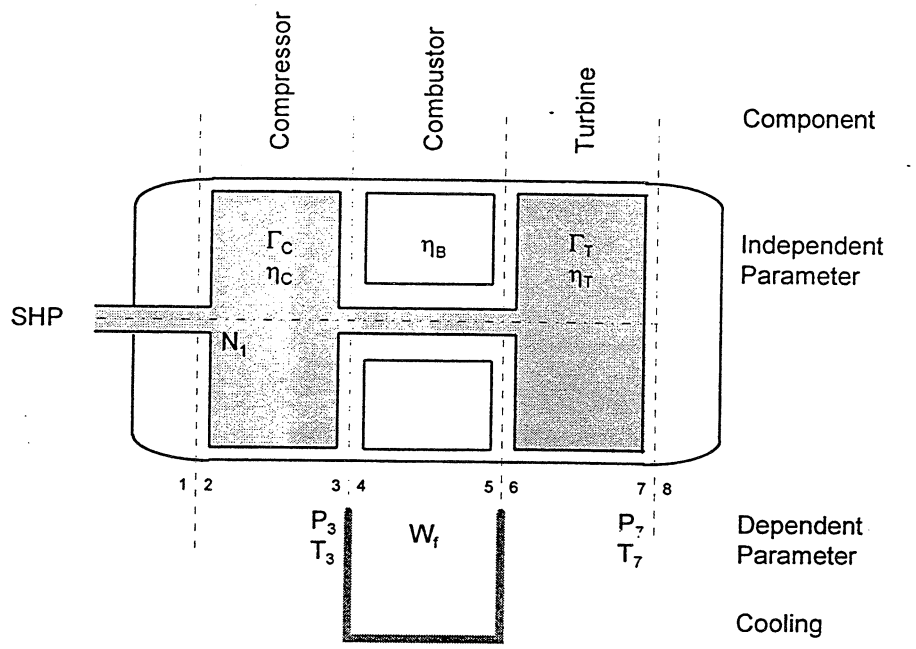


Figure 6.19: Schematic Layout of the SAllison T56S

Results

The GPA study of the ST56 has been carried out by Waldock (1995) and English (1995). The purpose of Waldock's study was to understand the GPA technique used in Pythia and to validate deteriorated performance calculations against published engine data. The purpose of English's study was to improve engine health monitoring systems for the ST56. To achieve this, English used Pythia's GPA technique in order to identify the ideal set of instrumentation for different fault sets. Furthermore, the analysis should determine whether linear or non-linear GPA gives better results.

Validation Process

The validation process of Pythia involved a comparison between simulated performance results and published data. In a first attempt, Waldock compared design point and off-design point results calculated by hand for a simple jet engine against Pythia's simulated results. Both results were similar but not identical. One possible reason for the discrepancy could be that the hand calculation did not use component characteristics. In a next step Waldock matched successfully the measured baseline of the ST56 with the computer generated baseline. And finally Waldock simulated deterioration by implanting physical faults such as eroded nozzle vanes and compressor blade tip wear.

A successful comparison of computed and measured deteriorated performance is shown in Figures 6.20 and 6.21.

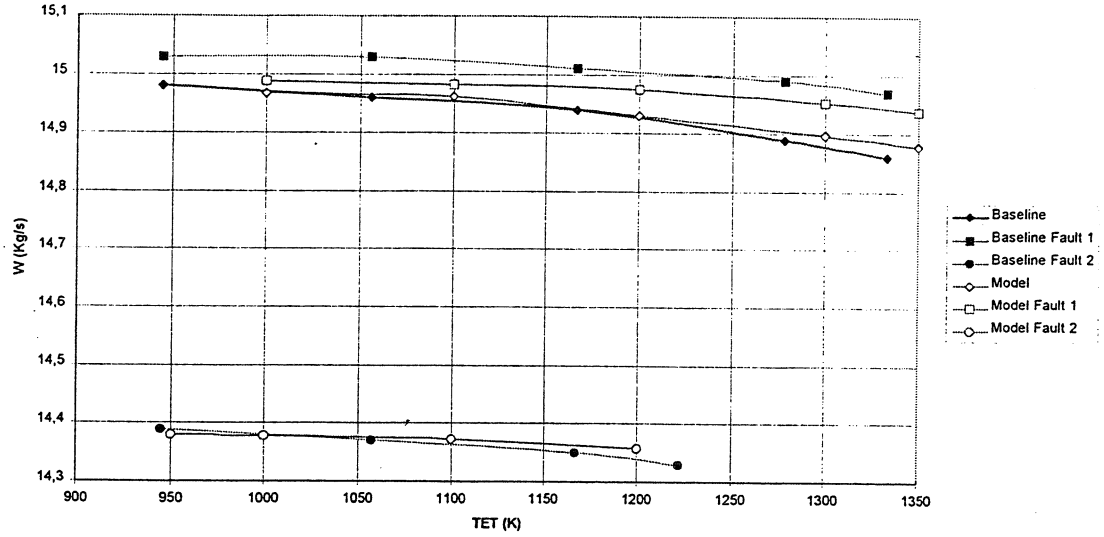


Figure 6.20: Validation of Deteriorated Performance: Air Mass Flow versus TET (Waldock, 1995)

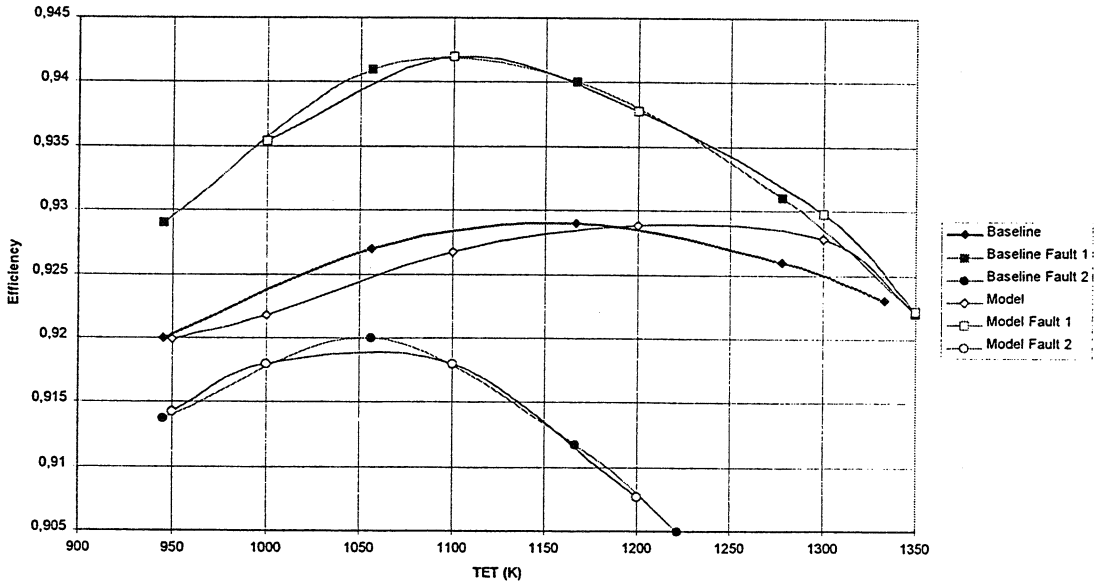


Figure 6.21: Validation of Deteriorated Performance: Turbine Efficiency versus TET (Waldock, 1995)

Physical Fault Simulation

Physical faults in a deteriorated engine simulation can be represented by changes in independent parameter changes. English tried to extend the library which describes the effect of physical faults on independent parameter (see Table 3.1 in Chapter 3)

In a computer simulation with Pythia, English implanted several different single and multiple physical faults and analysed the deteriorated performance. As a result the ST56 engine demonstrated in most of the cases a linear relationship between dependent and independent parameter changes. Interestingly, an implanted reduction in combustion efficiency (poor combustion) had hardly any effect on the performance of the engine. The only dependent parameter which showed a noticeable change was fuel flow.

GPA Studies

English's studies of Pythia were concerned with comparing linear and non-linear GPA and investigating the existing and the ideal instrumentation set for the ST56 engine. The result of the comparison showed that small changes in one implanted independent parameter was often better identified by the linear method than the non-linear method. However, the non-linear GPA demonstrated superior capability in detecting multiple independent parameter changes (see Figure 6.22). Typically English's implanted independent parameter changes represented physical faults with changed magnitudes. The result of the instrumentation analysis revealed that the existing instrumentation could adequately detect fault sets containing two or less independent parameters. The ideal instrumentation set for the ST56 should also include, apart from the existing, compressor exit temperature and pressure measurements.

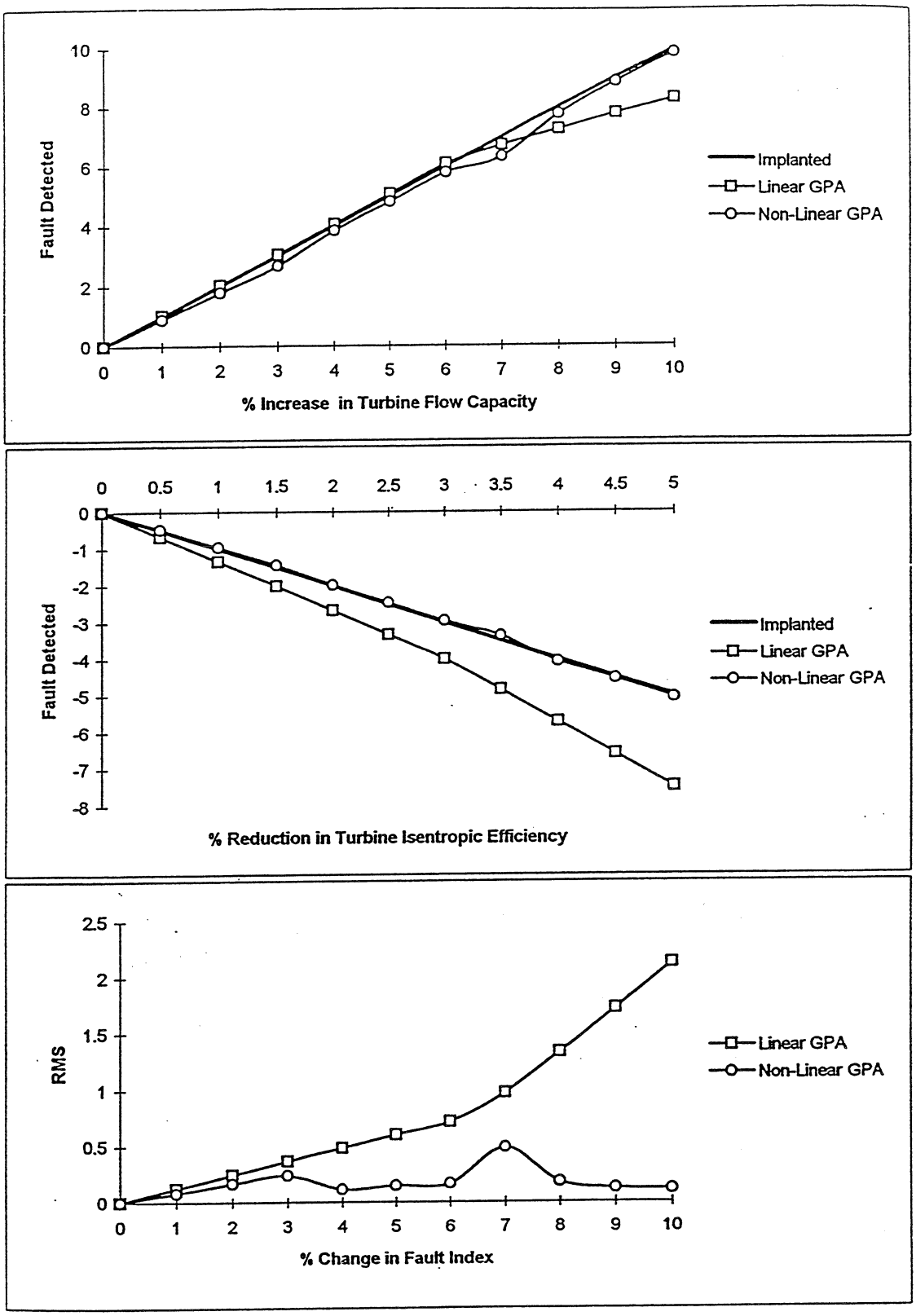


Figure 6.22: Effect of Independent Parameter Changes: Comparison of Linear and Non-linear GPA (English, 1995)

Summary

The non-linear GPA technique developed for the program Pythia is a powerful tool for identifying the appropriate set of instrumentation for a given fault set. Pythia was able to successfully identify sets of instrumentation for all five investigated gas turbine engines. In particular, it has emerged from the studies that some measurements are more suitable for the identification of implanted faults. Fouling in compressors was typically identified where compressor exit pressure and temperature measurements were used. Additional spool speed measurements helped to detect non-dimensional massflow changes. Erosion in turbines was best identified by using pressure and temperature measurements between the turbines. However, there are many other viable combinations of measurements that could have been used. As long as the measurements have some relation to the fault sought after, the non-linear GPA will result in a statistically significant improved solution, hence a low RMS error. An unsuitable set of instrumentation is identified by the presence of large coefficients in the FCM and by the non-convergence of the non-linear GPA method.

PART IV

DISCUSSION & CONCLUSION

7 Discussion

Once a model exists for performing GPA techniques then the model can be used to explore the strength and limitations of performance diagnostics and any unexpected side effects.

Potential Use of GPA Diagnostics

GPA is able to identify degradation at module level. The trends of these degradation can be determined and therefore appropriate maintenance action can be planned before secondary damages occur. As a result, engine down-time is kept to a minimum and hence a high level of availability becomes possible. For military gas turbines an improvement in availability of 1 % would potentially provide 5 percent more flying hours per year across a squadron of 15 aircraft (Sapsard, 1994). For a baseload industrial 370 MW powerplant a 1 % improvement in plant availability could result in higher net revenues and profits of up to $ 19 million over a period of 20 years (Peterson, 1991). GPA in conjunction with cost optimisation codes can enhance the planning of maintenance activities and the competitiveness of plants, particularly for industrial gas turbine users that compete to sell electricity (Meher-Howig, 1993).

The non-linear GPA technique is able to diagnose single and multiple physical faults. It predicts different faults from various combinations of measured parameters with greater accuracy than the linear GPA method. The reason for this is because the non-linear method imitates an exact solution whereas the linear presupposes the solution is linear. So far, in the process of GPA, physical faults were described as changes in independent parameters. However, in some cases, a change in an independent parameter can not be related to a physical fault. For example, a change in turbine flow can be either due to surface roughness or due to blade bowing. Whereas the cost of rectifying the surface roughness is relatively low, the cost of replacing of all affected stator segments is fairly high. One possible solution to this redundant relationship could be the combination of GPA with another engine health monitoring technique such as turbine borescope inspection (House, 1992). The borescope inspection could clearly identify changes in blade surfaces and hence the change in flow can be related to surface roughness. A combination of GPA with vibration measurements can clearly identify compressor fouling (Boyce, 1975). Another common problem is the loss of thermal barrier coating from turbine blades. If blades loose some of their protective coating, then the GPA technique will not be able to identify a fault. However, with increased time, the hot gas will penetrate the blade and therefore the blade will untwist and as a result the

performance will change. This change in performance can then be identified by the GPA technique. A combination of the GPA technique with debris monitoring technique would allow detection of loss of coating at an earlier stage of failure development. If the blade coating is used for corrosive protection, then a similar effect to the performance will occur as explained for the thermal barrier. The choice of fault sets used for GPA depends on the strategy adopted and the way the engine deteriorates over time. One strategy might be to choose a fault set indicative of the most probable faults or one that can identify less frequently occurring faults. The choice is arbitrary and depends on what goals the user wants to achieve by means of GPA.

GPA is able to accurately identify the appropriate set of instrumentation for a given fault set. Improvements in engine diagnostics can in theory be achieved simply by adding more and more reliable instrumentation to monitor the engine's health. However, the instrumentation itself has its own mean-time to failure. Additionally, inappropriate or badly maintained instrumentation can lead to the detection of spurious faults, leading to unnecessary expensive maintenance actions. This will occur particularly if the diagnostic technique employed is not robust. The instrumentation has substantial costs associated with its installation and use. These costs rise as the amount of instrumentation included is increased. Simple measurement devices such as thermocouples and pressure probes are cheaper to install than more sophisticated devices such as pyrometry etc. However, cost savings are limited by the capability of the technique employed, the design and the maturity of the engine in use, the fuel utilised, the operating environment, and the operating profile. Figure 7.1 illustrates how an excessive use of instrumentation results in a loss rather than a gain in terms of return on investment. A computer simulation of the engine, the instrumentation and the operating profile allows the user to optimise the instrumentation set for a particular task (Singh 1994 and Provost 1995).

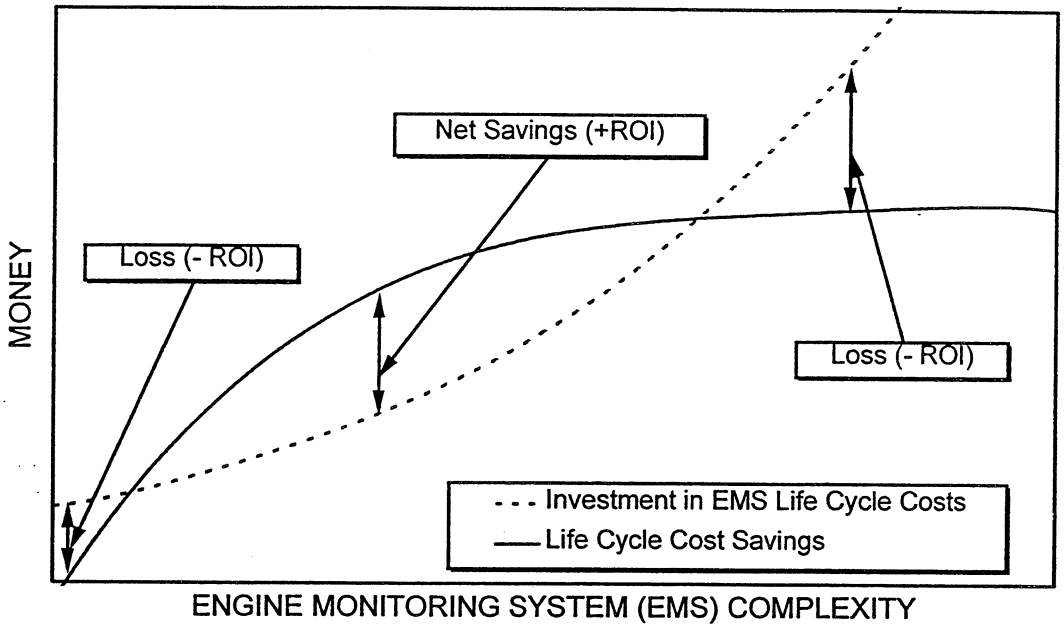


Figure 7.1: Return on Investment (ROI) of Engine Monitoring System (EMS)

The development of an appropriate set of instrumentation depends on the gas turbine user's acceptance level. Typically, a major change in an existing set of instrumentation is not likely to receive serious consideration (Saravanamuttoo, 1983). Any suggestions to use new instrumentation must be fully justified to the user and in most cases it is only considered for new engines.

It has emerged from the case studies that some instruments are more suitable than others for identifying component problems. In particular, problems in components such as compressors or turbines were usually identified where the component exit pressure and temperature were measured. A compressor mass flow fault was identified in most cases with compressor speed. In each case the dependent parameters relate to the fault specified as expected. This has also been observed for a small aero engine (Virgili 1989) and a helicopter engine (House, 1992). Similar observations are confirmed by Boyce (1975) who obtained appropriate instrumentation sets by studying the aero-thermo relationships of a gas turbine engine. The given information of suitable instrumentation is useful for the development of diagnostics tool or for expanding a diagnostic by modifying engines with additional instrumentation.

Limitations

GPA is limited to detecting only faults that are associated with noticeable changes in specified measurements. However, the measurements themselves depend on the

accuracy and reliability of the chosen instruments. Whereas accuracy errors are virtually eliminated in the process of measuring the change in performance, reliable measurements are dependent on the random error that derives from repeated measurements (precision error). Since Pythia's GPA technique is based on the assumption that dependent parameter changes are derived from perfect (repeatable) instruments, it will restrict practical applications of the program Pythia. Another limitation in the program is that changes in measured parameters cannot be entered into the system. In Pythia, measured parameter changes are generated by implanting independent parameter changes into the performance model. GPA identifies a deteriorated component by changes in independent parameters rather than physical faults. It should also be recognised that the faults must be associated with the thermodynamic measurements. There are many possible failure modes such as loss of oil pressure etc. which will not be found by GPA techniques (Saravanamuttoo, 1974). It is therefore important that the user/operator should have a good understanding of the relationship between physical faults and changes in independent parameters. The proposed GPA is based on the assumption that the number of measured dependent parameter is equal or greater than the number of independent parameters. A lower number of dependent parameters than independent parameters will result in an erroneous diagnostics approach.

Pythia divides GPA diagnostics into a linear and a non-linear GPA calculation. This separation is justified for the present thesis in order to show the significant improvement of the non-linear over the linear technique. However, for practical reasons a combination of the two techniques is advisable (see Chapter 4 and Lee 1995). In particular, the first iteration in the GPA process should always carry out the full Newton step. A reduced step should follow the process if the first solution is not satisfactory.

Further Applications

Creep Life

Creep life of turbine blades for industrial gas turbines is often quoted at base load power and ISO conditions. However, a user may operate in an environment where ambient temperatures are high. The creep life of the high pressure turbine blade for a range of potential faults at various power levels has been calculated (Ortiz, 1987) to study the effect of engine deterioration on component life. An example of the results of this study is shown in Figure 7.2.

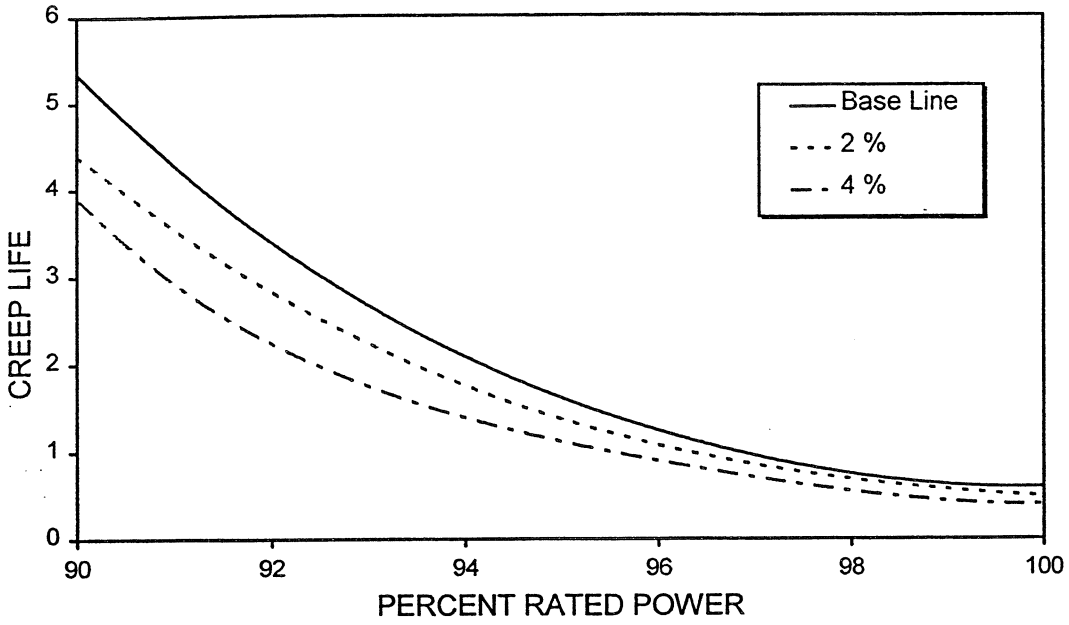


Figure 7.2: Effect of Compressor Degradation on 1st Stage Turbine Rotor Blade Creep Life

The calculations used actual temperature variations by considering the change in ambient temperature over the twenty four hour cycle, and over 365 days, along with the power required. The gas generator speed and temperature were calculated for each of the imposed deteriorations, and the creep life was calculated as a function of engine power and operating conditions. Such assessments can be of considerable value to engine operators. The use of such information allows to judge the most suitable maintenance or operation strategy, covering such items as the type of air filtration to be used, the frequency of filter changing, the compressor wash frequency etc. The operator should trade such maintenance actions against reduced fuel consumption and/or increased component life. Figure 7.2 is illustrative, but the original study (Ortiz, 1987) includes a number of imposed faults and engine cycles, and considers the implications in detail.

Low Cycle Fatigue Life (LCF)

The low cycle fatigue life of high pressure turbine blades is influenced by a range of parameters. As an example, Figure 7.3 shows how the LCF life of a high pressure turbine blade is reduced when the low pressure compressor efficiency is progressively deteriorated for a range of ambient (inlet) temperatures.

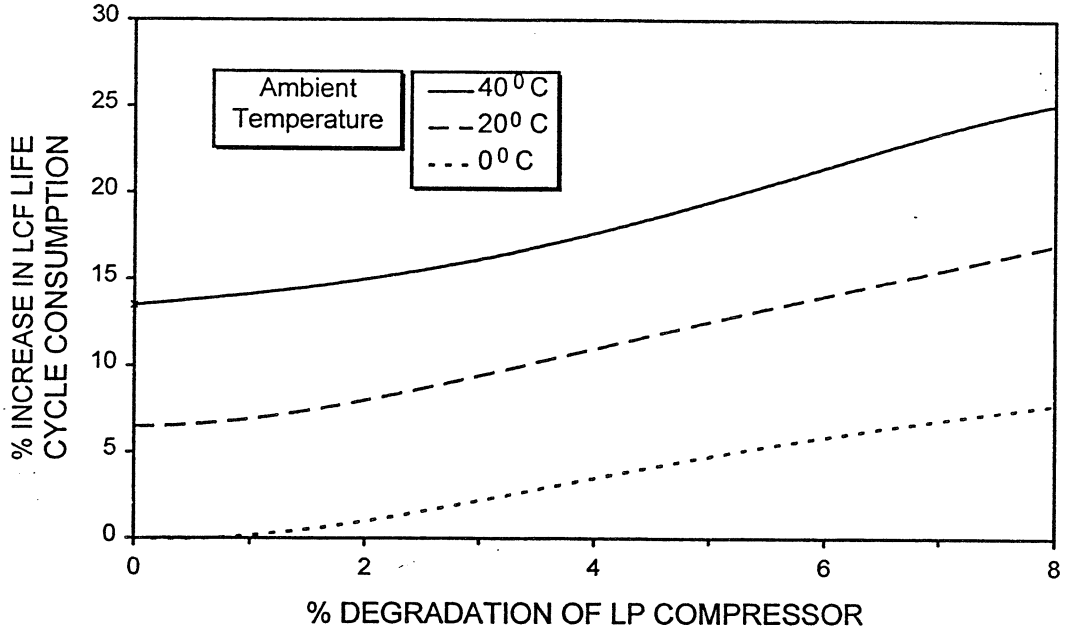


Figure 7.3: Low Cycle Fatigue Life. Effect of Degradation of LP Compressor & Variation of Inlet Temperature

The LCF calculation assumptions and method, along with the actual cycles to which the engine is subjected, will influence any LCF assessment. Again, the calculation presented is for illustration purposes to show how operators can benefit by using such techniques. A recent study shows how such techniques can be used for reducing engine life-cycle costs (Devereux, 1992). It is not the intention of this thesis to discuss LCF calculation methods (these are reviewed in Wu, 1994), but rather to illustrate how engine operators can benefit by using such techniques.

Instrumentation Sensitivity

Pythia's GPA technique can be enhanced by considering two areas of measurement sensitivity: (1) a faulty instrument and (2) estimation of instrument (sensor) errors. With a perfect (unchanging) engine and perfect (repeatable) sensors, the measured dependent parameter changes and therefore the calculated independent parameter changes will be zero (Urban, 1975). However, in an unchanging engine with imperfect instrumentation, the resulting change in the independent parameter is the error induced by the instrumentation . Agrawal (1979) suggested a procedure that can distinguish between faulty engines and faulty instrumentation. The procedure estimates the operating point based on one or a few of the measurements and checks the remainder of the measurements against the new operating point. If the comparison shows that all the measured parameters are within an acceptable limit then the measurements can be assumed to be accurate and hence a fault has occurred in the engine.

A further improvement of the GPA technique is suggested by Lunderstaedt (1988). Lunderstaedt enhances the GPA equations by considering measurement noise and systematic sensor errors which have been caused by a drift of the zero point. This enhancement introduces two unknown vectors in the set of equations and therefore the changes in independent parameters cannot be determined directly. Lunderstaedt solves the equation with a filter algorithm and a statistical approach. A similar enhancement of the set of equations is presented in Urban (1980).

House (1992) offers an alternative solution as to how sources of measurement errors can be reduced. House argues that errors in data can be minimised by automating and increasing the frequency of data acquisition. Automated data acquisition eliminates unsought problems that are introduced by manual readings. An increased frequency of data acquisition gives the engine operator the possibility to check discrepancies in the data against maintenance and operation records for classification. Additionally, failures of components that occur rapidly can be detected at an earlier stage.

Artificial Intelligence

Artificial intelligence is an area of computer science which attempts to produce programs or structures which simulate the way in which a human displays intelligent behaviour (Vivian, 1994). This area of science is typically separated into 5 categories such as (1) pattern recognition, (2) knowledge based systems (KBS), (3) natural language understanding, (4) robotics and (5) machine learning. However, for GPA techniques the most promising area of further development is for pattern recognition and knowledge based systems.

Pattern recognition is a method that discriminates engine faults by matching a set of measurement changes (Pattern) within a fault library. The fault library contains simulated deteriorated performance data that are obtained by imposing degradation on engine components. Typically, the pattern matching method is able to identify measurement uncertainty and faulty instrumentation (Lee, 1995). One drawback of pattern recognition is the creation of the extensive fault library for each engine type. However, a combination of the non-linear GPA and the pattern recognition method can form a powerful tool for identifying the appropriate instrumentation and for providing the user/operator with additional information for maintenance actions.

Knowledge based system are a suitable application for gas turbine fault diagnostics where there is no exact algorithmic solution (many problems have many solutions) or where an algorithmic solution is not feasible (too few instrumentation). KBS are a form

of computer program that have the ability to reproduce some of the elements associated with intelligent human behaviour. KBS are able to solve problems which cannot be solved with conventional algorithmic programs. They encapsulate the knowledge of an human expert within their knowledge base and use this expertise to guide the computer to solve a problem or arrive at a decision. KBS are designed to support or even replace an expert taking decisions and are therefore constructed to give advice rather than answers. In addition, a KBS can be used to integrate information from all three major parts of the gas turbine (gas path components, rotational mechanical equipment and accessory systems). Therefore KBS offer the possibility of integrating qualitative and quantitative elements of the maintenance decision process (Doel 1994 and Vivian 1995). In the case of GPA, several KBS have been developed that are able to select the most appropriate fault which has been caused by changes in the measurements (Vivian 1994, English 1995). With the development of the non-linear GPA technique it is expected that the stored knowledge base within the system is more accurate and therefore the identification of a fault more precise. The limitation with KBS is that it is restricted to the amount of expert knowledge stored in the knowledge base

A promising area is the application of neural networks. Neural networks are a good example of recent innovation for applications in many areas such as speech recognition etc. The advantage of the neural network approach is that it finds the patterns (if any) in the data by itself (Junk 1995, Meher-Howig 1985 and Sapsard 1994). This requires a training period and after the network has been trained it can classify new data or predict new values on the basis of its earlier learning. In the case of GPA, neural networks could be used to establish the relation of changes in dependent parameter to changes in independent parameters. The neural network finds its solution based on an extensive fault library that can be created by the non-linear GPA technique.

Stage-Stacking Technique

So far, GPA is used to identify a change in the performance of a particular component. But in reality common engine faults relate only to one or two blade rows. Therefore it would be more useful to identify a faulty stage especially for the compressor since it is the most critical component of a gas turbine. Mathioudakis (1994) proposes a method that employs a stage-stacking technique as a basic tool (Robbins, 1965). It is coupled with suitably formulated fault models and numerical optimisation schemes. Existing empirical background data are applied in order to give the technique as general a feature as possible (Mathioudakis, 1994). The stage-stacking procedure is a well established technique in the industry (Robbins 1965, Stones 1958 and Doyle 1962). The procedure

calculates the overall compressor performance characteristics by stacking estimated or generalised stage characteristics (Howell, 1978). Although the predicted characteristics conform to test results, the prediction of the surge line is less satisfactory (Bloch, 1992). The proposed method has some potential features that makes it suitable for an advanced GPA diagnostics program.

8 Conclusions

Over the years high cost of ownership in gas turbines have resulted in considerable interest in advanced maintenance strategies. One way to tackle the high cost is to employ GPA techniques. The potential of GPA can contribute significantly to an improved management of availability and hence the reduction of costs. For this reason a generalised GPA computer program Pythia has been developed that incorporates new techniques such as a non-linear multiple fault diagnostics scheme. The program can be used for analysing clean (design point and off-design point) and deteriorated gas turbine performances. As part of the GPA technology, Pythia simulates clean and deteriorated performance data by using the component-matching scheme Turbomatch.

In order to develop a reliable program Pythia, a structured methodology conforming to quality standards, has been implemented and used. The program is based on an object-oriented programming method that can be run with a modern PC. The validation process of the program covered a wide range of case studies on various types of engines. It is noteworthy that some of the case studies were carried out by five other authors. The features of the program Pythia are:

- modelling of deteriorated engines and GPA, although complex, is now sufficiently suitable for consideration by industrial gas turbine users and manufacturers.
- the use of such techniques allows the detection of component degradations at the module level with greater certainty and accuracy than the earlier linear GPA method.
- the recent development and validation of non-linear GPA offers a considerable advance in gas path diagnostics.
- the technique can assist in assessing consequences of imposed deterioration on components for operation of gas turbines, and hence for the identification of appropriate instrumentation sets for monitoring.
- GPA can be used to identify faults to create appropriate information for the generation of rules and inferential algorithms for knowledge-based systems. This is particularly of interest when the complete or desired set of instrumentation is not available.
- performance simulation of a degraded engine can form a useful basis for component life usage studies.

- a user-friendly frame makes the program easy to handle even for inexperienced users and operators.
- the reliable program can easily be expanded due to object-oriented design and software quality assurance. This will allow the use of the program for a variety of engine cycles.
- so far, engine diagnostics have been concerned with what set of measurements are required to identify a specific pattern of faults. But the GPA technique can be used to identify faults with a given set of measurements. It allows the generation of rules and inferential algorithms for knowledge-based systems.

However, the proposed method is complex and needs to be applied with considerable care. This is especially true for:

- the choice of measurements
- the establishment of a baseline
- the awareness of measurement non-repeatability
- the representation of physical faults by the changes in the independent parameters

Recommendations

The Pythia program was able to resolve the objectives of this thesis, however the program can be enhanced in the following areas:

GPA	The linear GPA and the iterative non-linear GPA process should be combined as a single process.
Sensor	The GPA model should include sensor problems and/or sources of data noise.
Measurement	The program Pythia should be modified to allow the user to enter rather than simulate actual deteriorated performance changes.
Fault	The representation of physical faults by independent parameter changes should be further investigated.

Intelligence The application of intelligent systems such as expert systems and neural networks could provide a platform for improved diagnostics in the gas path analysis technique.

References

ABB POWER GENERATION 1993 *GT24: Advanced cycle system, the innovative answer to lower the cost of electricity.* Brochure, ABB Power Generation

AGRAWAL R K, MACISAAC B D, SARAVANAMUTTOO H I H 1979 An analysis procedure for the validation of on-site performance measurements of gas turbines. *Trans. ASME Journal of Engineering for Power* **101**: 405

AKER G F, SARAVANAMUTTOO H I H 1989 Predicting gas turbine performance degradation due to compressor fouling using computer simulation techniques. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **111**

BLOCH G S 1992 *A wide-range axial-flow compressor stage performance model.* ASME Paper 92-GT-58

BOURASSA G 1984 *Technical and economic aspects of engine health monitoring.* MSc Thesis, Cranfield University

BOYCE M P 1975 *Parametric study of a gas turbine.* ASME Paper 75-GT-46

BOYCE M P, HANAWA D A 1974 *Development of techniques for monitoring turbomachinery.* Proceeding of Gas Turbine Operation and Maintenance symposium, Canadian Arctic Gas Study Ltd., 1974

BRITISH STANDARDS INSTITUTION 1990 *Quality assurance.* BSI, London, 4th Revised Edition

BURKHARDT R 1990 *DETEM User's Guide.* Manual, SME, Cranfield University

BURNELL D 1995 *Engine condition monitoring and its benefits to airlines.* 5th European Propulsion Forum, Pisa, Italy April 1995

COHEN H, ROGERS G F C, SARAVANAMUTTOO H I H 1986 *Gas turbine theory.* Longman Scientific & Technical, 3rd Edition

CUTTS G 1991 *Structured Systems Analysis and Design Methodology* Blackwell Scientific Publication, 2nd Edition

DAY M J, WAY N R, THOMPSON K 1987 *The use of particle counting techniques in the condition monitoring of fluid power systems.* Proceedings of an International Conference on Condition Monitoring, Swansea, UK

DEHU M 1995 *Health monitoring optimisation based on 60 million hours of CFM56 engine field experience.* 5th European Propulsion Forum, Pisa, Italy April 1995

DEVEREUX B 1992 *Improving life usage of the F404 engine through thrust rating.* MSc Thesis, Cranfield University

DEVEREUX B, SINGH R 1994 *Use of computer simulation techniques to assess thrust rating as a means of reducing turbo-jet life cycle costs.* ASME Turbo Expo '94, The Hague, Netherlands

DIAKUNCHAK I S 1992 Performance deterioration in industrial gas turbines. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **114**: 161-168

DOEL D L 1994 TEMPER - A gas-path analysis tool for commercial jet engines. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **116**: 82-89

DONAGHY M J 1991 *Gas path analysis - fault diagnosis using Detem.* MSc Thesis, SME, Cranfield University

DOWNS E, CLARE P, COE I 1992 *Structured systems analysis and design method: Application and context.* Prentice-Hall, New York, 2nd Edition

DOYLE M D 1962 The Stacking of Compressor Stage Characteristics to Give an Overall Compressor Performance Map. *Aeronautical Quarterly* Nov. 1962

DUBOIS G 1995 *Comparison of high bypass ratio turbofan with separate and mixed exhausts.* MSc Thesis, SME, Cranfield University

ENGLISH L 1995 *Application of gas path analysis, gas path debris monitoring and expert system technology to the Allison T56 turboprop engine.* MSc Thesis, SME, Cranfield University

ESCHER P C, SINGH R 1995 *An object-oriented diagnostics computer program suitable for industrial gas turbines.* United 21st International Congress of Combustion Engines (CIMAC), Interlaken, Switzerland May 1995

FISHER C 1995 *Aero engine gas path monitoring.* 5th European Propulsion Forum, Pisa, Italy April 1995

GOH J W 1989 *Gas path analysis of a two spool low bypass ratio turbofan engine based on the dry version of the ge f404.* MSc Thesis, SME, Cranfield University

GOODFELLOW J V 1988 *Operational requirements for engine condition monitoring from the EFA viewpoint.* AGARD-CP-165

GREWAL M S 1988 *Gas turbine engine performance deterioration modelling and analysis.* PhD Dissertation, SME, Cranfield University

HANCOCK P 1991 *Software aids to design.* 2nd Edition, SIMS, Cranfield University

HARRISON G F, SMITH M E F, NURSE J 1995 *Procedure for aero engines component life usage prediction.* 5th European Propulsion Forum, Pisa, Italy April 1995

HESS A J 1983 *An overview of US navy engine monitoring system: Program and users experience. Gas path analysis principles I.* Engine and Systems Performance Monitoring Seminar/Workshop, ADI Transportation Systems, Woodbury, New York, USA

HESS A J 1983 *Engine monitoring system requirements - military aircraft.* Engine and Systems Performance Monitoring Seminar/Workshop, ADI Transportation Systems, Woodbury, New York, USA

HOUSE P 1992 *Gas path analysis techniques applied to a turboshaft engine.* MSc Thesis, SME, Cranfield University

HOWELL A R 1978 *A new stage-stacking technique for axial compressor performance prediction.* ASME Paper 78-GT-139

JUNK R, LUNDERSTAEDT R 1995 *Model optimization of a gas turbine with fault affected reference measurements by neuronal networks for state and sensor diagnosis.* AIDAA 5th European Propulsion Forum, Pisa, Italy

KIRWAN A 1995 *A reliable approach to maintenance.* Power Plant Technology: Economics & Maintenance, AIP Publication, Sept. 1995

LAFORE R 1991 *Object-oriented programming in turbo C++.* Waite Group

LAGRANDEUR R D 1986 *Instrumentation for aero gas turbine engine condition monitoring systems.* MSc Thesis, SME, Cranfield University

LAKSHMINARASIMHA A N, BOYCE M P, MEHER-HOWJI C B 1994 Modelling and analysis of gas turbine performance deterioration. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **116**: 46-52

LAZALIER G R 1987 A gas path performance diagnostic system to reduce engine overhaul costs. *Trans. ASME Journal of Engineering for Power.* Oct. 1987

LUNDERSTAEDT R, FIEDLER K 1988 *Gas path modelling, diagnosis and sensor fault detection.* AGARD-CP-448

LYNCH S L 1989 *Reasoned guideline to the use of engine health monitoring for gas turbines in industrial applications.* MSc Thesis, Cranfield University

MACISAAC B D 1992 *Engine performance and health monitoring models using steady state and transient prediction methods.* AGARD-LS-183

MACLEOD J D, TAYLOR V, LAFLAMME J C G 1992 Implanted component faults and their effects on gas turbine engine performance. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **114**: 174-179

MACMILLAN W L 1974 *Development of a modular type computer program the calculation of gas turbine off design performance.* PhD Dissertation, SME, Cranfield University

MARKWELL T R 1985 *Condition monitoring of gas turbines through wear debris analysis.* MSc Thesis, SME, Cranfield University

MATHIOUDAKIS K 1994 Compressor fault identification from overall performance data based on adaptive stage stacking. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **116**: 156-164

MEHER-HOWIG C B 1985 *A feasibility study of the application of artificial intelligence techniques for turbomachinery.* ASME Paper 85-GT-102

MEHER-HOWIG C B, BOYCE M P et al 1993 *Condition monitoring and diagnostic approaches for advanced gas turbines.* Proceedings of the 7th Congress & Exposition on Gas Turbines Cogeneration and Utility, New York, USA

MICROSOFT 1993 *Microsoft visual C++: Development system for Windows version 1.0.* Microsoft Corporation

MOHAMMED R A 1987 *Engine condition monitoring of industrial gas turbine compressors.* MSc Thesis, SME, Cranfield University

MUSTAPHA N A K 1995 *The application of gas path analysis to combined cycle gas turbines.* MSc Thesis, SME, Cranfield University

NEALE H J 1987 *Trends in maintenance and condition monitoring.* Proceedings of an International Conference on Condition Monitoring, Swansea, UK

NORUSIS M J 1993 *SPSS for Windows: Base system user's guide.* Chicago, Release 6

ORTIZ T G 1987 *Gas turbine hot section life usage considerations.* MSc Thesis, SME, Cranfield University

PALMER J R 1967 *The "turbocode" scheme for the programming of thermodynamic cycle calculations on an electronic digital computer.* Report CoA/Aero-198, CoA, Cranfield University

PALMER J R 1995 *The turbomatch scheme for aero/industrial gas turbine engine design point/off-design performance calculation.* Unpublished guide, SME, Cranfield University

PASSALACQUA J 1974 *Description of automatic gas turbine engine trends diagnostic system.* First Symposium on Gas Turbine Operation and Maintenance National Research Council of Canada

PENDEDEKAS S L 1995 *Snecma M53-P2 turbofan engine health monitoring.* MSc Thesis, SME, Cranfield University

PETERSON J 1991 *Enhancing steam turbine-generator reliability-availability.* 4th International Power Generation Exhibition and Conference

PRESS W H 1994 *Numerical recipes in C: The art of scientific computing.* Cambridge University Press, 2nd Edition

PROVOST M J, SINGH R 1995 *Gas path analysis: Preparing for success.* AIDAA 5th European Propulsion Forum, Pisa, Italy April 1995

READ S 1993 *Algorithms.* Private Communication, SME, Cranfield University

ROBBINS W H, DUGAN J F 1965 *Predictions of off-design performance of multi-stage compressors.* NASA SP-36

ROLLS-ROYCE 1986 *The jet engine.* The Technical Publication Department, Rolls-Royce plc., Derby, England, 4th Edition

SAE 1981 *Aircraft gas turbine engine monitoring system guide.* Aerospace Recommended Practice (ARP) 1567, SAE-87/1734

SAPSARD M 1994 *Eucams: The story.* The Eucams Brochure, Futures Systems 42, MoD, UK

SARAVANAMUTTOO H I H 1974 *Gas path analysis for pipeline gas turbines.* Proceedings of the 1st Canadian NRC Symposium on Gas Turbines Operation and Maintenance

SARAVANAMUTTOO H I H 1985 *A preliminary assessment of compressor fouling.* ASME Paper 85-GT-153

SARAVANAMUTTOO H I H 1992 *Overview on Basis and Use of Performance prediction Methods.* AGARD-LS-183

SARAVANAMUTTOO H I H, MACISAAC B D 1983 Thermodynamic models for pipeline gas turbine diagnostics. *Trans. ASME Journal of Engineering for Power* **105**: 875-884

SCOTT J N 1979 *Axial compressor monitoring by measuring for intake depression.* Third Symposium on Gas Turbine Operation and Maintenance National Research Council of Canada

SEDDIGH F 1991 A proposed method for assessing the susceptibility of axial compressors to fouling. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **113**

SIMS 1992 *Systems Analysis.* Lecture Notes, Unpublished, SIMS, Cranfield University

SINGH R 1985 *Advanced condition monitoring.* Briefing Notes ME104, Unpublished, SME, Cranfield University

SINGH R 1993 *Availability and reliability.* Lecture Notes SME /2107/RS/MEI, Unpublished, SME, Cranfield University

SINGH R 1994 *Some considerations in the application of gas path analysis to stationary industrial gas turbines.* The International Gas Turbine Users Association 39th Annual Conference Caracas Venezuela 19 July 1994

SINGH R, ESCHER P C 1995 *Gas turbine diagnostics and availability.* Industrial & Power Gas Turbine Operations & Maintenance Conference, London, UK Sep. 1995

SINGH R, GREWAL M S 1989 *The emergence of gas path analysis as a maintenance aid for industrial gas turbines.* Proceedings 18th International Congress on Combustion Engines, Tianjin, China June 1989

SMETANA F O 1974 *Turbojet engine gas path analysis - a review.* AGARD-CP-165

SPIEGEL M R 1972 *Schaum's outline of theory and problems of statistics in SI units.* Schaum's Outline Series, McGraw-Hill, New York

STAPLES L J, SARAVANAMUTTOO H I H 1974 *An engine analyzer program for helicopter turboshaft powerplants.* AGARD-CP-165

STONES A 1958 Effects of stage characteristics and matching an axial flow compressor performance. *Trans. ASME* **80**: 1273-1293

TABAKOFF W 1990 Simulation of compressor performance deterioration due to erosion. *Trans. ASME Journal of Engineering for Power* **112**: 78-83

TICKIT 1990 *Guide to software quality management system construction and certification using EN29001.* DTI

URBAN L A 1969 *Gas turbine engine parameter interrelationships.* Hamilton Standard Division of United Aircraft Windsor Locks

URBAN L A 1972 *A gas path analysis applied to turbine engine condition monitoring.* AIAA-72/1082

URBAN L A 1974 *Condition monitoring of turbine engines and gas compressors for gas line pumping*. First Symposium on Gas Turbine Operation and Maintenance National Research Council of Canada

URBAN L A 1975 Parameter selection for multiple fault diagnostics of gas turbine engines. *Trans. ASME Journal of Engineering for Power* **96**: 225-230

URBAN L A 1980 *Gas path analysis - a tool for engine condition monitoring.* Safe & Efficient Management Energy Proceeding at the 33rd Annual International Air Safety Seminar, Christchurch, New Zealand

URBAN L A 1983 *Gas path analysis principles I.* Engine and Systems Performance Monitoring Seminar/Workshop, ADI Transportation Systems, Woodbury, New York, USA

VELLA A 1992 *Software quality assurance.* Lecture Notes, Unpublished, SIMS, SME, Cranfield University

VERTE D 1995 *Engine health monitoring: The point of view of an operator who happens to have an engine shop.* 5th European Propulsion Forum, Pisa, Italy April 1995

VIRGILI R 1989 *Gas path analysis for small aero gas turbine engines* . MSc Thesis, SME, Cranfield University

VIVIAN B 1993 *Gas path analysis diagnostics applied to a turboprop engine.* MSc Thesis, SME, Cranfield University

VIVIAN B, SINGH R *Application of expert system technology to gas path analysis of a single shaft turboprop engine.* 5th European Propulsion Forum, Pisa, Italy April 1995

VOLPONI A J 1982 *Gas path analysis: An approach to engine diagnostics.* 35th Symposium, Mechanical Failures Prevention Group, Gaithersburg, MD

WALDOCK D 1995 *An introduction to gas path analysis.* Preliminary Year Report, SME, Cranfield University

WILLIAMS L J 1981 *The use of mathematical modelling in the analysis of gas turbine compressor unit test data.* ASME Paper 81-GT-217

WILSON L B 1988 *Comparative programming languages.* Addison-Wesley Publishing

WRIGHT M R 1990 *Combined gas/steam cycle power generation from a user's viewpoint.* MPhil Dissertation, SME, Cranfield University

WU F E 1994 *Aero-engine life evaluated for combined creep and fatigue and extended by trading off excess thrust.* PhD Dissertation, SME, Cranfield University

ZHU P, SARAVANAMUTTOO H I H 1992 Simulation of an advanced twin-spool industrial gas turbine. *Trans. ASME Journal of Engineering for Gas Turbines and Power* **114**

Appendix A

COMPONENT MAP SCALING

The program Turbomatch simulates clean and deteriorated performance by modifying component characteristics. A performance simulation is accomplished when the engine is 'rematched' with the modified i.e. scaled component characteristics (Palmer, 1967). The following sections assume that the reader is familiar with the terminology used in Turbomatch (see Palmer 1995 and MacMillan 1974).

Theory of Scaling Factors

Compressor characteristics can be modified by the following scaling factors:

- Pressure ratio PR
- Non-dimensional mass flow Γ_C
- Isentropic efficiency η_C

Similarly, turbine characteristics can be modified by the following scaling factors:

- Non-dimensional mass flow Γ_T
- Isentropic efficiency η_T
- Enthalpy drop DH
- Rotational speed CN

In a clean performance simulation, each of the scaling factors PR, η_C, Γ_C, η_T, DH and CN is defined as follows:

$$SF_{Clean} = \frac{y_{Clean}}{y_{TM}} \qquad (A1)$$

where SF_{Clean} represents the clean scaling factor, y_{Clean} represents the design point value and y_{TM} represents the value that corresponds to the map value stored in Turbomatch. However, due to an inconsistency in the Turbomatch program the scaling factor Γ_T is defined as follows:

$$SF_{Clean} = \frac{y_{TM}}{y_{Clean}} \tag{A2}$$

In a deteriorated performance simulation, not all scaling factors are used. Scaling factors such as PR, η_C, Γ_C and η_T are defined as:

$$SF_{Deter} = SF_{Clean}(1 + \Delta y) \tag{A3}$$

where SF_{Deter} represents the deteriorated scaling factor and Δy represents the relative change in independent parameter. The scaling factor Γ_T is defined as:

$$SF_{Deter} = \frac{SF_{Clean}}{1 + \Delta y} \tag{A4}$$

In conclusion, a deteriorated component can be seen as a shift in the component characteristics (see Figures 3.3 and 3.4 in Chapter 3).

Scaling Factors in Turbomatch

Scaling factors in Turbomatch have been limited to the simulation of clean performance studies. Therefore, in order to carry out deteriorated performance studies, the program Turbomatch has been modified in the pre-programmed routines compressor and turbine for fixed and variable geometry (Bricks, see Palmer 1995). For each brick three new functions have been incorporated which can manipulate the scaling factors:

Updating — Updating component scaling factors (UPDATECOMSF for compressors and UPDATETURSF for turbines, see Figures A1 and A2). These functions operate only in the design point calculations. The purpose of the functions is to enable the user to specify its own scaling factors in the allocated brick data (see Table 5.2 in Chapter 5). If no data is available, then Turbomatch calculates automatically the scaling factors as described in the clean performance simulation and copies the scaling factors to the brick data.

Obtaining — Obtaining component scaling factors (GETCOMSF for compressors and GETTURSF for turbines, see Figures A1 and A2). These functions operate in design point and off-design point calculations. In design point, the functions obtain the scaling factors from the allocated brick data (see Table 5.2 in Chapter 5). Since the off-design point calculation is an iterative process, the scaling factors are

obtained in the first loop. In the following loops the function is inactive.

Resetting Updating all scaling factors (UPDATEALLSF, see Figures A1 and A2). The function operates in design point and off-design point calculations. The purpose of the function is to update all brick data scaling factors with the calculated scaling factors such as PRCSF, ETACSF and WACSF for compressors and TFTSF and ETATSF for turbines (see Table 3.1 of Chapter 3).

The modification to Turbomatch enables the access of scaling factors via the Turbomatch input file. Either the user/Pythia specifies directly the scaling factors in a design point calculation or the user/Pythia performs an arithmetic operation (see brick ARITHY, Palmer 1995) on the appropriate scaling factors in an off-design point calculation. Hence deteriorated performance studies can be carried out.

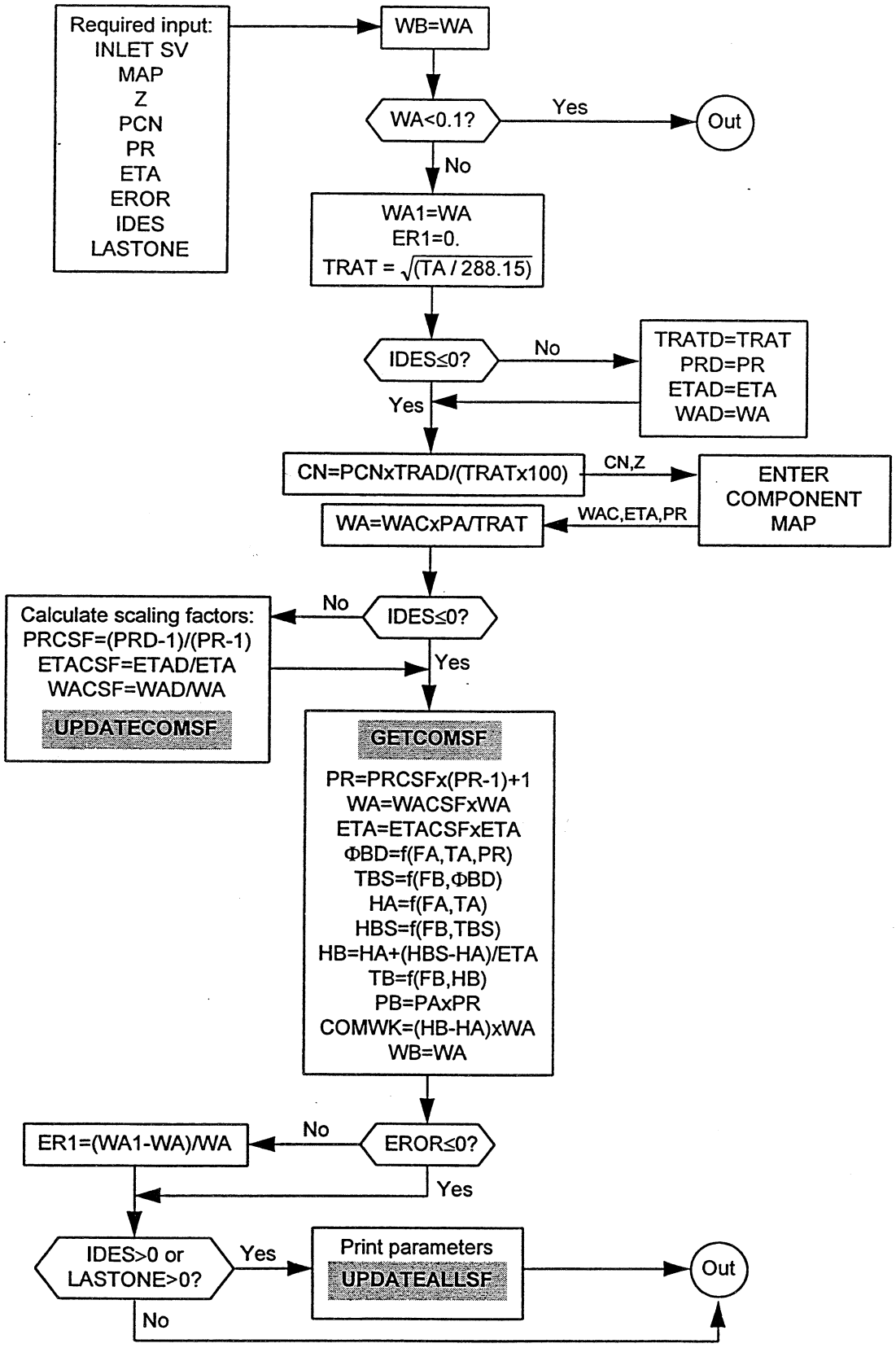


Figure A1: Compressor Scaling in Turbomatch (MacMillan, 1974)

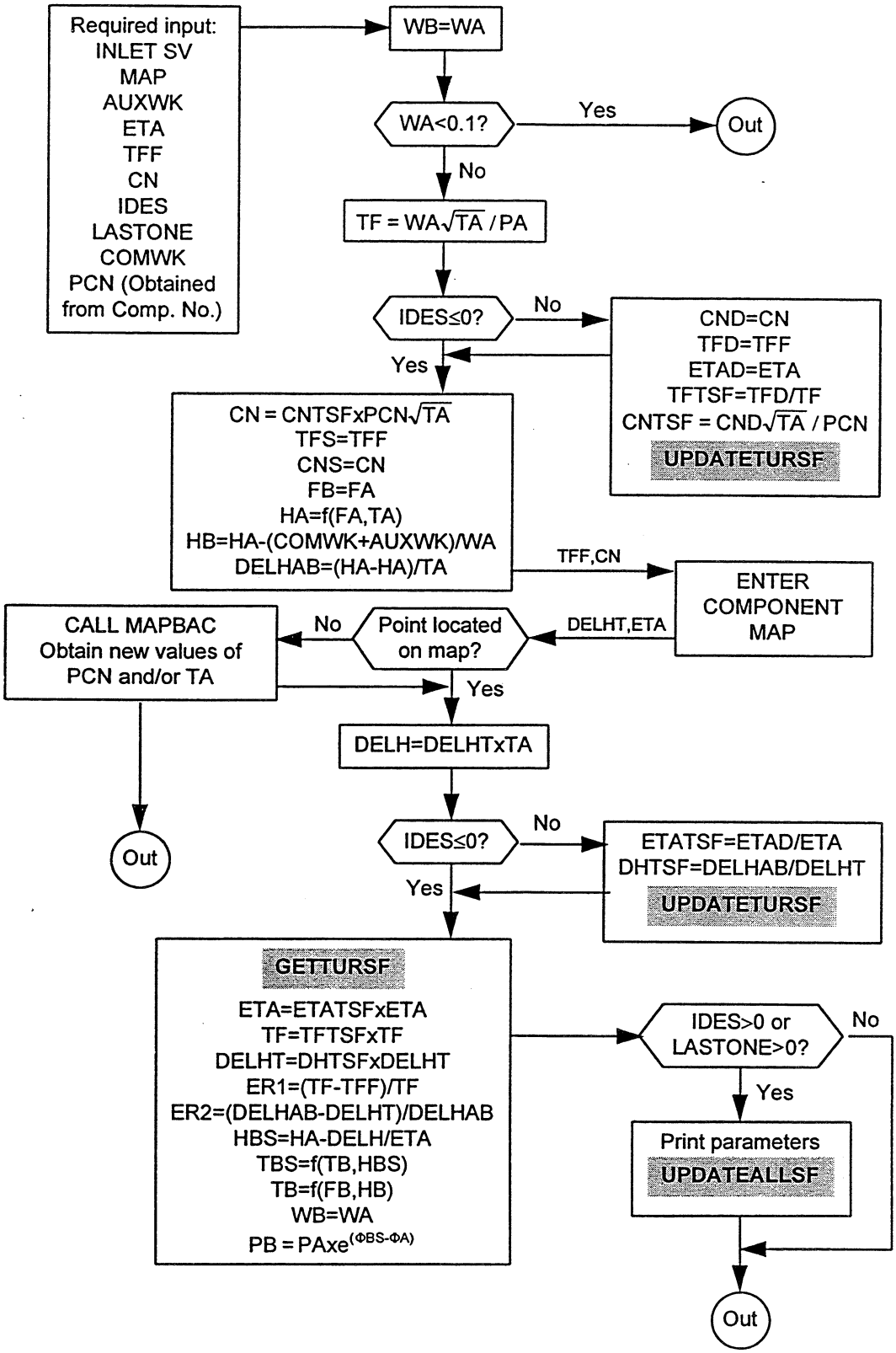


Figure A2: Turbine Scaling in Turbomatch (MacMillan, 1974)

Appendix B

PYTHIA CLASS LIBRARY VERSION 1.1

A detailed description of the Pythia class library 1.1 is given below. The library describes classes, its member functions and its member data. The classes are sorted in alphabetical order.

PYTHIA CLASS LIBRARY

VERSION 1.1

class CAddParameter: public CBaseElement

The **CAddParameter** class deals with additional parameters that are displayed in modules. Usually these parameters are calculated from other brick data or station vector data.

#include <designdo.h>

See Also **CBrickData.**

Construction/Destruction-Public Members

CAddParameter	Constructs **CAddParameter** in various ways.

Overridables-Public Members

GetData	Gets data for particular case.
GetStatus	Gets status.
UpdateData	Updates data for particular case.
UpdateStatus	Updates status.

Data Members-Protected Members

m_DPData	A **double** data number 1.
m_nStatus	An integer that specifies status of additional parameter.
m_ODData	A **double** data number 2.

class CBaseElement : public CObArray

The **CBaseElement** class serves as a base class for several classes that are needed to specify data in modules.

#include <designdo.h>

Construction/Destruction-Public Members

CBaseElement	Constructs **CBaseElement** in various ways.

Overridables-Public Members

GetData	Gets data for particular case.
GetItem	Gets item for particular case.
GetNumber	Gets number for particular case.
GetPointer	Gets pointer.
GetStatus	Gets status.
Serialize	Serializes data member.
UpdateData	Updates data for particular case.
UpdateItem	Updates item for particular case.
UpdateNumber	Updates number for particular case.
UpdatePointer	Updates pointer.
UpdateStatus	Updates status.

Operations-Public Members

CheckDoubleOK	Checks if **double** data is okay.
CheckIntOK	Checks if **int** data is okay.
ShowDoubleError	Shows **double** error message.
ShowIntegerError	Shows **int** error message.

Member Functions

CBaseElement:: CheckDoubleOK

BOOL CheckDoubleOK(double *Data*, **UINT** *nCase*);

Data Data to be checked.

nCase Data that is one of the following cases:

- **BRICK_DATA** Range of data for brick data.
- **STATION_VECTOR** Range of data for station vector data.
- **DETER_VAR** Range of data for faults.

Remarks Checks **double** data if it is in the valid range for a particular case.

Return Value **TRUE** if no error in data.

CBaseElement:: CheckIntOK

BOOL CheckIntOK(int *nNumber*, **UINT** *nCase*);

nNumber Number to be checked.

nCase Number that is one of the following cases:

- **BRICK_DATA** Range of numbers from minimum brick data number to maximum engine vector result number.
- **STATION_VECTOR** Range of numbers that are valid for station vector numbers.
- **ENGINE_VECTOR** Range of numbers that are valid for engine vector result numbers.
- **SERIAL_NO** Range of numbers that are valid for serial numbers.

Remarks Checks integer numbers if they are in the valid range for a particular case.

Return Value **TRUE** if no error in number.

class CBFileDlg : public CDialog

The **CBFileDlg** class defines base file that is needed to use in clean or in deteriorated performance modelling.
#include <bfiledlg.h>

Construction/Destruction-Public Members

CBFileDlg	Constructs **CBFileDlg** in various ways.

Overridables-Protected Members

DoDataExchange	Dialog data exchange.

Message Handlers-Protected Members

OnDblclkCpfBaseFile	Closes dialog.
OnInitDialog	Initializes dialog.
OnOK	Checks valid pathname before closing dialog.

Data Members-Public Members

m_ListBoxEngFile	A **CListbox** object that displays pathnames.
m_nViewStatus	An integer that specifies active view.
m_strPathName	A **CString** that specifies base file name.

Member Functions

CBFileDlg::CBFileDlg

CBFileDlg(CWnd* *pParent*);
CBFileDlg(int *nViewStatus*);
pParent The parent window.
nViewStatus Active view status.

Remarks Constructs **CBFileDlg** object.

CBFileDlg::OnDblclkCpfBaseFile

void OnDblclkCpfBaseFile();

Remarks Call **CBFileDlg::OnOK** member function.

CBFileDlg::OnOK

void OnOK();

Remarks Closes dialog and checks if pathname is valid.

---

class CBrickData : public CBaseElement

The **CBrickData** class that deals with brick data from modules/components.
#include <designdo.h>

Construction/Destruction-Public Members

CBrickData	Constructs **CBrickData** in various ways.

Overridables-Public Members

GetData	Gets brick data for particular case.
GetNumber	Gets brick number for particular case.
GetStatus	Gets status.
Serialize	Serializes **CBrickData** member data.
UpdateData	Updates data for particular case.
UpdateNumber	Updates number for particular case.
UpdateStatus	Updates status.

Data Members-Protected Members

m_Data	A **double** data number 1.
m_DPData	A **double** data number 2.
m_nNumber	An integer that specifies the brick data number.
m_nStatus	An integer that specifies status of brick data.
m_ODData	A **double** data number 3.

Member Functions

CBrickData::CBrickData

CBrickData()
CBrickData(int *nNumber*, **double** *Data*)
nNumber Specifies brick data number.
Data Specifies brick data.

Remarks Constructs **CBrickData** object.

CBrickData::GetData

virtual double GetData(UINT *nCase*);
nCase Defines one of the following cases:

- **DESIGN_ENGINE** Design engine mode.
- **DESIGN_POINT** Design point.
- **OFFDESIGN_POINT** Off-design point.
- **CPF_DESIGN** Clean performance design mode.

- **CPF_DP** Clean performance design point.
- **CPF_OD** Clean performance off-design point.
- **GPA_DESIGN** Gas path analysis design mode.
- **DPF_BASE** Baseline performance mode.
- **DPF_DETER** Deteriorated performance mode.
- **GPA_LINEAR** Linear gas path analysis mode.
- **GPA_NONLINEAR** Non-linear gas path analysis mode.

Remarks Obtains data for particular case.

Return Value The **double** value for particular case.

---

class CChartFrame : public CMDIChildWnd

The **CChartFrame** is the frame window for displaying charts.

#include <chartvw.h>

Construction/Destruction-Public Members

CChartFrame	Constructs **CChartFrame** object.
~CChartFrame	Destructs **CChartFrame** object.

Overridables-Protected Members

OnCreateClient	Creates a client window for the frame.

Data Members-Public Members

m_Splitter	A **CSplitterWnd** object.

---

class CChartView : public CView

The **CChartView** handles the graphical presentation of data created in clean and/or deteriorated performance calculations.

#include <chartvw.h>

Construction/Destruction-Protected Members

CChartView	Constructs **CChartView** object.
~ CChartView	Destructs **CChartView** object.

Operations-Public Members

GetChartMetrics	Gets number of distances between ticks.
GetDataFromGrid	Gets data from station vector line
GetDisplayRect	Gets **CRect** dimensions of display.
GetDocument	Attaches appropriate document to view.
GetMaxValue	Gets maximum value of plotted data.
GetTickValue	Gets value for ticks on axis.
PlotAxes	Plots axes.
PlotBarChart	Plots bar chart graph.
PlotCaptions	Plots captions for axis.
PlotLineChart	Plots line chart graph.

Overridables-Protected Members

OnDraw	Updates data in graph charts.
OnInitialUpdate	Initializes before view is called.
OnUpdate	Updates view.

Data Members-Public Members

m_pData	Points to a **double** variable where chart data is stored.
m_nDataRows	An integer that specifies number of rows.
m_nDataCols	An integer that specifies number of columns.
m_nStartRow	An integer that specifies number of starting row.
m_nStartCol	An integer that specifies number of starting column.
m_nChartType	An integer that specifies chart type (Bar chart or line chart).
m_nViewStatus	An integer that specifies active view status.

---

class CChoiceDoc : public CDocument

The **CChoiceDoc** controls parts of pull-down menus.

#include <choicedo.h>

Construction/Destruction-Protected Members

CChoiceDoc	Constructs **CChoiceDoc** object.

Message Handlers-Protected Members

OnUpdateFileOpen	Controls open file menu for project file.
OnUpdateFileNew	Controls new file menu for project file.

---

class CChoiceFrame : public CMDIChildWnd

The **CChoiceFrame** is responsible for choosing operation modes such as clean performance modelling etc.

#include <choicefr.h>

Construction/Destruction-Protected Members

CChoiceFrame	Constructs **CChoiceFrame** object.
~CChoiceFrame	Destructs **CChoiceFrame** object.

Message Handlers-Protected Members

OnFileNewEng	Creates new file appropriate to user's choice of mode.
OnFileOpenEng	Opens file appropriate to user's choice of mode.
OnRunCleanPerformance	Shows available clean performance files.
OnRunDeterioratedPerformance	Shows available deteriorated/GPA performance files.
OnRunDesign	Shows available design engine files.

---

class CChoiceView : public CFormView

The **CChoiceView** class provides the user with listboxes, that distinguish between design engine mode, clean performance mode and deteriorated performance mode. Latter mode also includes Gas Path Analysis.

#include <choicevw.h>

Construction/Destruction-Public Members

CChoiceView	Constructs **CChoiceView** object.

Operations-Protected Members

OpenDocumentFile	Opens document file appropriate to user's choice of mode.

Operations-Public Members

ActivateEngFile	Calls new or open file function.
ChoiceMode	Activates user's choice of mode.
UpdateListbox	Updates listbox that displays available files for particular mode.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.
OnActivateView	Updates status bar and listbox.
OnDraw	Called when view is redrawn.
OnInitialUpdate	Initializes listbox.
OnUpdate	Updates listbox.

Message Handlers-Protected Members

OnClickedChcCleanperf	Activates clean performance mode.
OnClickedChcDelete	Calls delete file.
OnClickedChcDesign	Activates design engine mode.
OnClickedChcDeterperf	Activates deteriorated performance mode (=Gas Path Analysis mode).
OnClickedChcNew	Calls new file.
OnClickedChcOpen	Opens selected file.
OnDblclkChcList	Opens selected file from listbox.

Data Members-Protected Members

m_cleanperfButton	A **CButton** that controls clean performance mode.
m_DesFileStatNew	A **BOOL** variable that specifies status of new file (**TRUE** if new, **FALSE** if already exist).
m_designButton	A **CButton** that controls design engine mode.
m_deterperfButton	A **CButton** that controls deteriorated/GPA performance mode.
m_nChoice	An integer variable that specifies active mode.
m_strPathName	A **CString** that specifies path name of file to be opened.

Data Members-Public Members

m_EnableButtonNew	A **CBitmapButton** that creates new file.
m_EnableButtonOpen	A **CBitmapButton** that opens an existing file.
m_ListBox	A **CListBox** object.

Member Functions

CChoiceView::ChoiceMode

void ChoiceMode(UINT *nID*);

nID Defines active mode that user wants to work with.

- **DESIGN** Design engine mode.
- **CLEAN_PERFORMANCE** Clean performance modelling mode.
- **DETERIORATED_PERFORMANCE** Deteriorated performance modelling mode including Gas Path Analysis mode.

Remarks Activates mode.

---

class CCleanPerfDoc : public CDesignDoc

The **CCleanPerfDoc** class deals with the objects in clean performance modelling. It is also responsible for carrying out performance calculation using Turbomatch (Palmer, 1995).

#include <clnprfdo.h>

See also **CDesignDoc, CDeterPerfDoc**

Construction/Destruction-Protected Members

CCleanPerfDoc	Constructs **CCleanPerfDoc** object.
~CCleanPerfDoc	Destructs **CCleanPerfDoc** object.

Overridables-Protected/Public Members

Serialize	Serializes data member.
DelModule	Deletes all objects of particular module.

Performance-Public Members

AddODSelect	Adds handle.
AddTempSelect	Adds temporary handle.
DelAllModules	Deletes all modules/components.
DelTempSelects	Deletes temporary handles.
GetModule	Gets the current module/component pointer.
GetNumODSelects	Gets number of handles.
GetNumTempSelects	Gets number of temporary handles.
GetODSelect	Gets the current handle pointer.
GetSVLine	Gets the current station vector pointer.
GetTempSelect	Gets the current temporary handle pointer.
InitializeModules	Initialize modules with data from design engine modules.
LoopNumberOK	Checks if station vector number is valid.
ReadCpfTurbomatch	Reads in design point and/or off-design point data from Turbomatch run.
ReadTMOutputFile	Reads in data from Turbomatch output file (FOR004.DAT).
RunTurbomatch	Invokes Turbomatch.
WriteModules	Writes out modules to logfile.
WriteODSelect	Writes out handle selection to logfile.
WritePlotSVBrickData	Writes out plotsv brick data.
WritePlotSVCodeWord	Writes out plotsv codeword.
WriteStationVector	Writes out station vector data.
WriteSVLine	Writes out all station vector data to logfile.
WriteTMInputFile	Writes out input file (FOR001.DAT) that is used for running Turbomatch.

Export-Public Members

ExportExcel	Exports data to spreadsheet file.
ExportFile	Exports clean performance data to design engine data file.
WriteExcelHandles	Writes out handles to spreadsheet file.
WriteExcelModules	Writes out modules to spreadsheet file.
WriteExcelSVLine	Writes out station vectors to spreadsheet file.
WriteExcelTitle	Writes out user defined text to spreadsheet file.

Chart-Public Members

GetSelRowCol	Gets row and column selection.
ResetSelRowCol	Resets rows and columns selection.
UpdateSelRowCol	Updates row and column selection.

Status-Public Members

GetStatusLoop	Gets loop number for status bar.
GetTime	Gets time.
UpdateLoopNumber	Updates loop number.
UpdateStatusLoop	Updates loop number in status bar.
UpdateTime	Updates time in status bar.

Diagnostics-Public Members

AddDiapp	Adds monitored parameter (=dependent parameter).
DelAllDiapp	Deletes all monitored parameter.
GetDiapp	Gets pointer to current monitored parameter.
GetNumDiapp	Gets number of monitored parameters.

Message Handlers-Public Members

OnBuildMapsRead	Called when maps are read in from external file.
OnBuildMapsReset	Called when status of maps is reset to default.
OnBuildOffdesign	Called when user wants to change handle selection for off-design point calculation.
OnCpfExport	Called when user wants to export data to spreadsheet file.
OnCpfImport	Called when user wants to import clean performance data from other locations.
OnPlotArea	Called when station vector area is to be plotted.

OnPlotFueltoair	Called when station vector fuel-to-air ratio is to be plotted.
OnPlotMassflow	Called when station vector mass flow is to be plotted.
OnPlotPressStat	Called when station vector static pressure is to be plotted.
OnPlotPressTot	Called when station vector total pressure is to be plotted.
OnPlotTempStat	Called when station vector static temperature is to be plotted.
OnPlotTempTot	Called when station vector total temperature is to be plotted.
OnPlotVelocity	Called when station vector velocity is to be plotted.
OnRunCpfTurbomatch	Called when clean performance modelling is to be carried out.
OnToolsRestoreMouse	Called when mouse has become invisible.
OnUpdatePlotArea	Called when area is updated.
OnUpdatePlotFueltoair	Called when fuel-to-air ratio is updated.
OnUpdatePlotMassflow	Called when mass flow is updated.
OnUpdatePlotPressStat	Called when static pressure is updated.
OnUpdatePlotPressTot	Called when total pressure is updated.
OnUpdatePlotTempStat	Called when static temperature is updated.
OnUpdatePlotTempTot	Called when total temperature is updated.
OnUpdatePlotVelocity	Called when velocity is updated.

Operations-Public Members

GetFuel	Gets fuel selector.
GetGeometry	Gets geometry selector.
GetPrint	Gets print selector.
GetSerialNo	Gets current serial number.
GetUnits	Gets unit selector.
ResetSerialNo	Resets serial number to minimum value.
SerialNoPlus1	Adds 1 to current serial number.
ShowRunError	Shows run error.
UpdateCase	Updates case selector.
UpdateSelector	Updates all selectors.
UpdateSerialNo	Updates serial number with new value.

Data Members-Protected Members

m_arrPtrDiapp	A **CObArray** object that contains monitored parameters.
m_bIsExport	A **BOOL** variable that specifies status of exported file (**TRUE** if file is exported, **FALSE** if file is imported).
m_bIsLogFile	A **BOOL** variable that specifies status of log file (**TRUE** if log file is present, **FALSE** if no log file).
m_LogFile	An **ofstream** object that specifies log file.
m_nDPSelEndCol	An integer that specifies end of column.
m_nDPSelEndRow	An integer that specifies end of row.
m_nDPStartCol	An integer that specifies start of column.
m_nDPStartRow	An integer that specifies start of row.
m_nNumCpfOdLoops	An integer that specifies number of loops in off-design point calculation.
m_nODSelEndCol	An integer that specifies end of column.
m_nODSelEndRow	An integer that specifies end of row.
m_nODStartCol	An integer that specifies start of column.
m_nODStartRow	An integer that specifies start of row.
m_nSerialNo	An integer that specifies serial number.
m_strExpText	A **CString** that is exported to spreadsheet file.
m_TimeCpfDesign	A **CTime** that specifies time for clean performance design data.
m_TimeCpfDp	A **CTime** that specifies time for design point result data.
m_TimeCpfOd	A **CTime** that specifies time for off-design point result data.

Member Functions

CCleanPerfDoc::AddDiapp

void AddDiapp(CPtrToSelect* *PDiapp*)**;**

PDiapp Pointer to current monitored parameter.

Remarks Adds monitored parameter pointer to **CCleanPerfDoc::m_arrPtrDiapp.**

CCleanPerfDoc::AddODSelect

void AddODSelect(CPtrToSelect* *PSelect*)**;**

PSelect Pointer to current handle selection.

Remarks Adds handle pointer to **CDesignDoc::m_arrODSelect.**

CCleanPerfDoc::AddTempSelect

void AddTempSelect(CObArray* *arrSelect*, **CPtrToSelect*** *PSelect*);

arrSelect Pointer to current temporary handle array.

PSelect Pointer to current temporary handle selection.

Remarks Adds temporary handle to temporary handle array. This array is used to store temporarily changed handle selection(s). If user cancels dialog box for managing handles (see **CCleanPerfDoc::OnBuildOffdesign**) then old selection of handles is restored.

CCleanPerfDoc::GetFuel

int GetFuel();

Remarks Gets fuel selector variable.

Return Value One of the following values, with the meaning as given:

- **0** Kerosin.
- **1** Hydrogen.

See also Palmer (1995).

CCleanPerfDoc::GetGeometry

int GetGeometry();

Remarks Gets the geometry selector variable.

Return Value One of the following values, with the meaning as given:

- **0** Fixed geometry in compressor.
- **1** Variable geometry in compressor.

See also Palmer (1995).

CCleanPerfDoc::GetModule

CCpfModule* GetModule(int *nIndex*);

CModule* GetModule(char *strCheckModuleName[]*, **int&** *nIndex*, **BOOL&** *bRead*);

strCheckModuleName Module (=brick) name.

nIndex Index to current selection of module.

bRead Check if module is in current **CDesignDoc::m_arrModule**.

Remarks The first member function obtains the module that describes the brick. The second member function checks if the given module name is available in the current module list.

Return Value The **CCpfModule** or the **CModule** pointer element currently at this index.

CCleanPerfDoc::GetPrint

int GetPrint();

Remarks Gets the print selector variable. Selector defines the output file of Turbomatch.

Return Value One of the following values, with the meaning as given:

- **0** Full print.
- **1** Short print.
- **2** No print
- **3** Extra print.

See also Palmer (1995).

CCleanPerfDoc::GetSelRowCol

void GetSelRowCol(int& *nStartRow*, **int&** *nStartCol*,**int&** *nSelEndRow*, **int&** *nSelEndCol*, **UINT** *nCase*);

nStartRow Start row index.

nStartCol Start column index.

nSelEndRow End row index.

nSelEndCol End column index.

nCase Current view status of PYTHIA program.

- **CPF_DP** Clean performance design point mode.
- **CPF_OD** Clean performance off-design point mode.
- **DPF_BASE** Baseline performance mode.
- **DPF_DETER** Deteriorated performance mode.

Remarks Gets selection of data for plotting on charts. The data are chosen from station vectors.

CCleanPerfDoc::GetSerialNo

int GetSerialNo();

Remarks Obtains the temporary serial number that is usually used in creating output file for clean performance run.

Return Value The serial number.

CCleanPerfDoc::GetStatusLoop

virtual CString GetStatusLoop(UINT *nViewStatus*);

nViewStatus Current view status of PYTHIA program.

- **CPF_DESIGN** Clean performance edit mode.
- **CPF_DP** Clean performance design point mode.

- **CPF_OD** Clean performance off-design point mode.

Remarks Gets loop number of clean performance off-design point run. For CPF_DESIGN and CPF_DP status bar shows no loop number.

Return Value The **CString** for status bar.

CCleanPerfDoc::GetSVLine

CSVLine* GetSVLine(char *strFileSVLine[]*, **BOOL&** *bRead*);
CSVLine* GetSVLine(int *nIndex*);

strFileSVLine Station vector number..

bRead Check if station vector is in current **CDesignDoc::m_arrSVLine**.

nIndex Index to current selection of station vector.

Remarks The first member function checks if the chosen station vector is available in the current station vector line (**CDesignDoc::m_arrSVLine**). The second member function obtains selected station vector from the station vector line.

Return Value The **CSVLine** pointer element currently at this index.

CCleanPerfDoc::GetTime

virtual CString GetTime(UINT *nViewStatus*);

nViewStatus See **CCleanPerfDoc::GetStatusLoop**.

Remarks Obtains time from currently shown data in PYTHIA.

Return Value Time as a **CString** variable.

CCleanPerfDoc::GetUnits

int GetUnits();

Remarks Gets the unit selector variable.

Return Value One of the following values, with the meaning as given:

- **IMPERIAL** Imperial units.
- **SI** SI - units.

See also Palmer (1995).

CCleanPerfDoc::InitializeModules

void InitializeModules();

Remarks Initialize modules with data from design engine file. Usually this member function is called after a new clean performance file has been created and a design engine file has to be loaded in order to initialize the data.

CCleanPerfDoc::LoopNumberOK

BOOL LoopNumberOK(char *strName[]*);

strName String that needs to be checked.

Remarks Turbomatch output file provides the loop numbers for an off-design point calculation after a specified label.

Return Value **TRUE** if loop number can be read.

CCleanPerfDoc::ReadCpfTurbomatch

BOOL ReadCpfTurbomatch(ifstream(*TMOutputFile***), UINT** *nCase*);

TMOutputFile Specifies file.

nCase Current view status of PYTHIA program.

- **CPF_DP** Clean performance design point mode.
- **CPF_OD** Clean performance off-design point mode.

Remarks Gets the specified performance data.

Return Value **TRUE** if no error while reading in Turbomatch output file (FOR004.DAT).

CCleanPerfDoc::Serialize

virtual void Serialize(CArchive& *ar*);

ar A **CArchive** object to serialize to or from.

Remarks Reads or writes this object from or to an archive. The process of serialization distinguishes between following options:

- **ENG_VERSION_11** Design engine PYTHIA version 1.1.
- **CPF_VERSION_11** Clean performance PYTHIA version 1.1.
- **VERSION_10** PYTHIA version 1.0.
- **VERSION_11** PYTHIA version 1.1.

---

class CCleanPerfVw : public CDesignVw

The **CCleanPerfVw** class deals with the display of modules that describe a gas turbine engine. It is also responsible for calling module dialogs in order to specify module data such as brick data, station vector data, engine vector result data etc.

#include <clnprfvw.h>

See also **CDesignVw**, Palmer (1995).

Construction/Destruction-Protected Members

CCleanPerfVw	Constructs **CCleanPerfVw** object.
~ CCleanPerfVw	Destructs **CCleanPerfVw** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.
OnActivateView	Called when view is active.
OnDraw	Called when view is redrawn.
OnInitialUpdate	Initializes view.
OnUpdate	Updates view.

Operations-Public Members

GetDocument	Gets document that is attached to view.
Initialize	Initializes module and station vector data.
InitModuleMember	Initializes module data.
UpdateSelector	Updates selectors.
ViewSelectors	Displays selectors.

Message Handlers-Public Members

OnClickedCpfSelCt	Called when fixed geometry selector is to be shown.
OnClickedCpfSelFprint	Called when full print selector is to be shown.
OnClickedCpfSelHyd	Called when hydrogen fuel selector is to be shown.
OnClickedCpfSelIm	Called when imperial units selector is to be shown.
OnClickedCpfSelKer	Called when kerosen fuel selector is to be shown.
OnClickedCpfSelNprint	Called when no print selector is to be shown.
OnClickedCpfSelSi	Called when si units selector is to be shown.
OnClickedCpfSelSprint	Called when short print selector is to be shown.
OnClickedCpfSelVa	Called when vairable geometry selector is to be shown.
OnClickedCpfSelXprint	Called when extra print selector is to be shown.
OnCreate	Called when view is created.
OnModule	Called when user invokes dialog for particular module.
OnSetFocus	Called when view gets focus.
OnVScroll	Called when view is scrolled.

Data Members-Public Members

m_nFuel	An integer that specifies fuel selector.
m_nGeometry	An integer that specifies geometry selector.
m_nPrint	An integer that specifies print selector.
m_nUnits	An integer that specifies unit selector.

Member Functions

CCleanPerfVw::CCleanPerfVw

CCleanPerfVw();

CCleanPerfVw(UINT *nID*);

nID Contains the ID number of a dialog-template resource.

Remarks Constructs a **CCleanPerfVw** object.

CCleanPerfVw::Initialize

void Initialize();

Remarks Initializes module members and station vectors that belong to a particular module. After initialization the modules are displayed as a symbol that is representative for the module. Following modules are only displayed in CPF_DESIGN mode:

- **IDS_ENG_ARITHY** Arithy brick.
- **IDS_ENG_PLOTBD** PLOTBD brick.
- **IDS_ENG_PLOTSV** PLOTSV brick.

See also Palmer (1995).

CCleanPerfVw::OnActivateView

virtual void OnActivateView(BOOL *bActivate*, **CView*** *pActivateView*, **CView*** *pDeactiveView*);

bActive Indicates wheter a view is being activated or deactivated..

pActivateView Points to the view object that is being activated.

pDeactiveView Points to the view object that is being deactivated.

Remarks Updates status bar with information that correspond to displayed data.

CCleanPerfVw::OnModule

BOOL OnModule(UINT *nID*);

nID ID that specifies drawn module.

Remarks Calls dialog for particular module. Before calling dialog additional paramters and station vectors are updated in **CCpfModule**.

Return Value **TRUE** if no error while calling dialog.

See also **CCpfModule, CAddParameter**

CCleanPerfVw::OnUpdate

virtual void OnUpdate(CView* *pSender*, **LPARAM** *lHint*, **CObject*** *pHint*);

pSender Points to the view that modified the document, or **NULL** if all views are to be updated.

lHint Contains information about the modifications.

pHint Points to an object storing information about the modifications.

Remarks In case that user imported new data from other file, module data are updated with newly imported data.

CCleanPerfVw::ViewSelectors

void ViewSelectors(BOOL *bView*);

bView **TRUE** if selector is to be shown; **FALSE** if selector is not to be shown.

Remarks Displays selectors.

Data Members

CCleanPerfVw::m_nFuel

Remarks Fuel selector which contains one of the following values:

- **0** Kerosin.
- **1** Hydrogen.

CCleanPerfVw::m_nGeometry

Remarks Geometry selector which contains one of the following values:

- **0** Fixed geometry in compressor.
- **1** Variable geometry in compressor.

CCleanPerfVw::m_nPrint

Remarks Print selector which contains one of the following values:

- **0** Full print.
- **1** Short print.
- **2** No print
- **3** Extra print.

CCleanPerfVw::m_nUnits

Remarks Unit selector which contains one of the following values:

- **IMPERIAL** Imperial units.
- **SI** SI - units.

---

class CClnPrfResVw : public CCleanPerfVw

The **CClnPrfResVw** class serves as a base class for **CCpfDPResVw** and **CCpfODResVw.** It is responsible for displaying station vector data in listboxes and for displaying modules. Each module calls a dialog box which enables to modify module data.

#include <cpfresvw.h>

See also **CCpfDPResVw, CCpfODResVw,** Palmer (1995).

Construction/Destruction-Protected Members

CClnPrfResVw	Constructs **CClnPrfResVw** object.
~ CClnPrfResVw	Destructs **CClnPrfResVw** object.

Overridables-Public Members

DoDataExchange	DDX/DDV support.
OnInitialUpdate	Called when view is initialized.
OnUpdate	Called when view data need to be updated.

Operations-Protected Members

GetDocument	Gets document that is attached to view.

Operations-Public Members

AddScrollBar	Adds scroll bar to listbox.
GetScrollPos	Gets scroll position.
SetCurSel	Sets cursor selection at specified section.
SetListboxSel	Sets selection to listbox.
SetScrollPos	Sets scroll position.
SetScrollRanges	Sets scroll ranges.
UpdateFocusListBox	Updates listbox that currently has focus. Also the chart view is updated.
UpdateListBoxes	Updates data in listboxes that display station vector data.

Message Handlers-Public Members

OnSelchangeCpfResList0	Called when station vector number listbox is selected.

OnSelchangeCpfResList1	Called when fuel-to-air listbox is selected.
OnSelchangeCpfResList2	Called when mass flow listbox is selected.
OnSelchangeCpfResList3	Called when static pressure listbox is selected.
OnSelchangeCpfResList4	Called when total pressure listbox is selected.
OnSelchangeCpfResList5	Called when static temperature listbox is selected.
OnSelchangeCpfResList6	Called when total temperature listbox is selected.
OnSelchangeCpfResList7	Called when velocity listbox is selected.
OnSelchangeCpfResList8	Called when area listbox is selected.

Data Members-Protected Members

m_listboxScrollPos	An integer that specifies listbox selection.
m_nListBoxID	An integer that specifies ID of each listbox.
m_scrSV	A **CScrollBar** object that specifies scroll bar for station vector numbers

---

class CCoefficient : public CBaseElement

The **CCoefficient** class handles the individual coefficients in matrices such as influence coefficient matrix (ICM) and fault coefficient matrix (FCM).

#include <dtrprfdo.h>

Construction/Destruction-Public Members

CCoefficient	Constructs a **CCoefficient** object.

Overridables-Public Members

GetData	Obtains data that specifies vector component.
Serialize	Serializes **CCoefficient** data member.
UpdateData	Updates data with new value.

Data Members-Protected Members

m_Data	A **double** variable that specifies coefficient within matrix.

---

class CCompressor : CModule

The **CCompressor** objects (= compressor brick) are used in design engine modelling. Since other modules such as turbine etc. have the same layout as the **CCompressor** class, following classes are not explained in detail: **CArithy** (= arithy brick), **CDucter** (= ducter brick), **CHetcol** (= heat exchanger cold side brick), **CHethot** (= heat exchanger hot side brick), **CIntake** (= intake brick), **CCpfMixees** (= mixees brick), **CMixful** (= mixful brick), **CNozcon** (= convergent nozzle brick), **CNozdiv** (= divergent nozzle brick), **CPerformance** (= perfor brick), **CCpfPlotbd** (= plotbd brick), **CPlotsv** (= plotsv brick), **CPremas** (= premas brick) and **CTurbine** (= turbine brick).

#include <designdo.h>

See also **CModule**, **CCpfModule**, (Palmer, 1995).

Construction/Destruction-Protected Members

CCompressor	Constructs **CCompressor** object in various ways.

Overridables-Public Members

Draw	Draws module picture.
DrawDialog	Calls module dialog.
Serialize	Serializes **CCompressor** data member.

Member Functions

CCompressor::CCompressor

virtual CCompressor();

Public **virtual Ccompressor(int** *nModuleID*, **int** *nLastBD*, **int** *nLastSV*, **int** *nLastEV*);

nModuleID	Unique module ID.
nLastBD	Variable that specifies last brick data number.
nLastSV	Variable that specifies last station vector number.
nLastEV	Variable that specifies last engine vector result number

Remarks Constructs **CCompressor** object and initializes data member.

---

class CCpfCompressor : CCpfModule

The **CCpfCompressor** objects (= compressor brick) are used in clean performance modelling. Since other modules such as turbine etc. have the same layout as the **CCpfCompressor** class, following classes are not explained in detail: **CCpfArithy** (= arithy brick), **CCpfDucter** (= ducter brick), **CCpfHetcol** (= heat exchanger cold side brick), **CCpfHethot** (= heat exchanger hot side brick), **CCpfIntake** (= intake brick), **CCpfMixees** (= mixees brick), **CCpfMixful** (= mixful brick), **CCpfNozcon** (= convergent nozzle brick), **CCpfNozdiv** (= divergent nozzle brick), **CCpfPerformance** (= perfor brick), **CCpfPlotbd** (= plotbd brick), **CCpfPlotsv** (= plotsv brick), **CCpfPremas** (= premas brick) and **CCpfTurbine** (= turbine brick).

#include <clnprfdo.h>

See also **CModule**, **CCpfModule**, Palmer (1995).

Construction/Destruction-Protected Members

CCpfCompressor	Constructs **CCpfCompressor** object in various ways.

Overridables-Public Members

Draw	Draws module picture.
DrawDialog	Calls module dialog.
Serialize	Serializes data member.
WriteBDCodeWord	Writes out brick data codeword.
WritePlotBDCodeWord	Writes out plotbd codeword.

Member Functions

CCpfCompressor::CCpfCompressor

virtual CCpfCompressor();

Public **virtual CCpfCompressor(CModule*** *pModule*);

pModule Pointer to **CModule** object.

Remarks Constructs **CCpfCompressor** object and initializes data member if **CModule** object is available.

class CCpfDPResVw : public CClnPrfResVw

The **CCpfDPResVw** class deals with the display of design point data from clean performance calculations. A similar class to **CCpfDPResVw** is the class **CCpfODResVw**. The latter one is reponsible for displaying off-desing point data from clean performance calculations. For a detailed description of the class **CCpfODResVw** see class **CCpfDPResVw.**

#include <cpfdpvw.h>

See also **CClnPrfResVw**

Construction/Destruction-Protected Members

CCpfDPResVw	Constructs **CCleanPerfVw** object.
~ CCpfDPResVw	Destructs **CCleanPerfVw** object.

Overridables-Protected Members

OnDraw	Called when view is redrawn.
OnActivateView	Called when view is active.

Overridables-Public Members

OnInitialUpdate	Called when view is initialized.
OnUpdate	Called when view data need to be updated.

class CCpfModule : public CModule

The **CCpfModule** class is a base class for objects used in clean performance modelling.

#include <clnprfdo.h>

Construction/Destruction-Protected Members

~CCpfModule	Destructs **CCpfModule** object.

Operations-Public Members

DeleteBrickParam	Deletes all implanted and observed faults.
DelImpDeterVar	Deletes **CDeterVar** object from **CCpfModule::m_arrImpDeterVar** array.
DelObsDeterVar	Deletes **CDeterVar** object from **CCpfModule::m_arrObsDeterVar** array.
GetImpDeterVar	Gets pointer to implanted **CDeterVar** object.
GetNumImpDeterVar	Gets number of implanted faults.
GetNumObsDeterVar	Gets number of observed (or detected) faults.
GetObsDeterVar	Gets pointer to observed **CDeterVar** object.
InitAddParameter	Initializes additional parameters.
InitMembers	Initializes members for clean performance modelling.
UpdateAddParameter	Updates additional parameter.

Overridables-Public Members

Draw	Draws module picture (= button).
DrawDialog	Calls **CModulDlg** in order to update member data.
Serialize	Serializes data member.
Write	Writes out data member to logfile.
WriteBDCodeWord	Writes out brick data codeword.
WriteBrickData	Writes out brick data.
WriteExcelFaults	Writes out excel faults.
WritePlotBDCodeWord	Writes out plotbd codeword.

Data Members-Protected Members

m_arrImpDeterVar	A **CObArray** object that specifies implanted fault (independent parameter).
m_arrObsDeterVar	A **CObArray** object that specifies observed/detected fault.

Member Functions

CCpfModule::Draw

virtual void Draw(CWnd* *pParentWnd*, **CPoint** *ptModOrigin*, **Cpoint** *ptOffsetScroll*);

virtual void Draw(CWnd* *pParentWnd*, **CRect** *rectModOrigin*, **CPoint** *ptOffsetScroll*);
pParentWnd pointer to parent window.
ptModOrigin **CPoint** coordinates where module has to be drawn.
rectModOrigin **CRect** coordinates where module has to be drawn.
ptOffsetScroll Offset scroll position of window.
Remarks Loads module picture and draws module on specified location.

CCpfModule::DrawDialog

virtual void DrawDialog(int *nViewStatus*);
nViewStatus Status specifying active view.
- **CPF_DESIGN** User specifies design point and off-design point parameter.
- **CPF_DP** Design point results are shown.
- **CPF_OD** Off-design point resutls are shown.

Remarks **CModulDlg** dialog is called and data are shown according to view status.

CCpfModule::GetImpDeterVar

virtual CDeterVar* GetNumImpDeterVar(int *nIndex*);
nIndex Element index in array.
Remarks Returns the **CDpfModule::m_arrImpDeterVar** element at the specified index.
Return Value The **CDeterVar** pointer element currently at this index.

CCpfModule::GetNumImpDeterVar

virtual int GetNumImpDeterVar();
Remarks Obtains the number of implanted faults in array **CDpfModule::m_arrImpDeterVar.**
Return Value Number of faults.

CCpfModule::GetNumObsDeterVar

virtual int GetNumObsDeterVar();
Remarks Obtains the number of observed or detected faults in array **CDpfModule::m_arrObsDeterVar.**
Return Value Number of faults.

CCpfModule::GetObsDeterVar

virtual CDeterVar* GetNumObsDeterVar(int *nIndex*);
nIndex Element index in array.
Remarks Returns the **CDpfModule::m_arrObsDeterVar** element at the specified index.
Return Value The **CDeterVar** pointer element currently at this index.

CCpfModule::InitAddParameter

void InitAddParameter(int *NumAddPara*);
NumAddPara Number of additional parameters are calculated and displayed in **CModulDlg** dialog.

CCpfModule::InitMembers

void InitMembers(CModule* *PModule*);
PModule Pointer to **CModule** object.
Remarks Data member of **CDesignDoc** are taken in order to create new objects in **CCleanPerfDoc** class.

---

class CDesignDoc : public CDocument

The **CDesignDoc** class serves as a base class for clean and deteriorated performance modelling.. It is also responsible for designing engines.
#include <designdo.h>
See also Palmer (1995), **CCleanPerfDoc, CDeterPerfDoc**

Construction/Destruction-Public Members

CDesignDoc	Constructs **CDesignDoc** object.
~CDesignDoc	Destructs CDesignDoc object.

Overridables-Protected Members

GetStatusLoop	Obtains status of loop (not available in **CDesignDoc**).
GetTime	Obtains time of current data.
OnSaveDocument	While saving, temporarily created modules are removed.
Serialize	Serializes **CDesignDoc** member data.
UpdateStatusLoop	Updates loop status in status bar (In **CDesignDoc** loop status is not defined).
UpdateTime	Updates time of current displayed data.

Module-Public Members

AddModule	Adds **CModule** object at specified index in **CDesignDoc::m_arrModule** array.
DelAllModules	Deletes all **CModule** objects from **CDesignDoc::m_arrModule** array.
DelModule	Deletes **CModule** object at specified index from **CDesignDoc::m_arrModule** array.
DelODSelects	Deletes all **CPtrToSelect** objects from **CDesignDoc::m_arrODSelect** array.

GetModule	Obtains pointer to **CModule** object at specified index from **CDesignDoc::m_arrModule** array.
GetNumMods	Obtains number of modules in **CDesignDoc::m_arrModule** array.
InsertModule	Inserts pointer to **CModule** object at specified index in **CDesignDoc::m_arrModule** array.
ResetModuleNumbering	Resets module numbering.

SV Line-Public Members

AddSVLine	Adds pointer to **CSVLine** object in **CDesignDoc::m_arrSVLine** array.
DelSVLine	Deletes **CSVLine** object at specified index from **CDesignDoc::m_arrSVLine** array.
GetNumSVLine	Gets number of station vectors in the **CModule::m_arrSVLine** array.
GetSVLine	Obtains pointer of **CSVLine** object at specified index from **CDesignDoc::m_arrSVLine** array.
UpdateSVLine	Updates pointer to **CStationVector** objects in station vector line.

Operations-Public Members

GetViewStatus	Obtains current view status.
RemoveFile	Removes file from hard disc.
UpdateBaseFileName	Updates base file name.
UpdateViewStatus	Updates current view status.

Message Handlers-Protected Members

OnToolsDataSwitch	Called when data fields in **CModulDlg** are shown or hidden.
OnToolsRenumber	Called when brick data numbers, station vector numbers and engine vector results need new numbering.
OnToolsRenumberSwitch	Called when renumber switch menu is updated.
OnUpdateFileNew	Called when new file menu is updated.
OnUpdateFileOpen	Called when open file menu is updated.
OnUpdateToolsDataSwitch	Called when data fields switch menu is updated.
OnUpdateToolsRenumberSwitch	Called when renumber switch menu is updated.

Data Members-Protected Members

m_arrModule	A **CObArray** that contains all **CModule** objects.
m_arrODSelect	A **CObArray** that contains all **CPtrToSelect** objects.
m_arrSVLine	A **CObArray** that contains all **CSVLine** objects.
m_nCase	An integer that defines case of performance calculation.
m_nFuel	An integer that defines fuel.
m_nGeometry	An integer that defines geometry.
m_nPrint	An integer that defines print selector.
m_nUnits	An integer that defines unit selector.
m_nViewStatus	An integer that specifies view status of current data.
m_strBaseFileName	A **CString** that specifies base file name.
m_TimeDesign	A **CTime** that specifies time of current data.

Data Members-Public Members

m_bIsFieldHidden	A **BOOL** that specifies data field switch.
m_bIsRenumber	A **BOOL** that specifies renumber switch.
m_nExport	An integer that specifies export status.
m_nImport	An integer that specifies import status.

Member Functions

CDesignDoc::GetStatusLoop

virtual CString GetStatusLoop(UINT *nViewStatus*);

nViewStatus Defines one of the following case:

- **DESIGN_ENGINE** Design engine modus.

Remarks Obtains loop number for particular view. In **CDesignDoc** loop number is not defined.

Return Value A **CString** that specifies loop status.

CDesignDoc::GetViewStatus

int GetViewStatus();

Remarks Obtains view status of current data.

Return Value An integer that specifies view status.

See Also **CDesignDoc::UpdateViewStatus**

CDesignDoc::Serialize

virtual void Serialize(CArchive& *ar*);

ar A **CArchive** object to serialize to or from:

Remarks Reads or writes this object from or to an archive. The process of serialization distinguishes between following options:

- **VERSION_10** PYTHIA version 1.0.
- **VERSION_11** PYTHIA version 1.1.

CDesignDoc::UpdateSVLine

void UpdateSVLine();
void UpdateSVLine(CModule* *PModule***);**

PModule Pointer to **CModule** object

Remarks The first member function updates pointers in the **CDesignDoc::m_arrSVLine** array. Several modules can have the same station vector. To avoid that the **CDesignDoc::m_arrSVLine** contains the same station vector more than once, only the first occurrence of a station vector is stored in the array. The first occurrence receives a positive pointer. Following occurences receive negative pointers. The pointer defines the station vector number (eg. pointer = -2 in compressor module means that a former module (e.g. intake) already has station vector 2). The latter member functions updates station vector line reference in **CModule**.

---

class CDesignFrame : public CMDIChildWnd

The **CDesignFrame** class provides the functionality of a Windows multiple document interface child window, along with members for managing the window.

#include <designfr.h>
#include <palette.h>

Construction/Destruction-Protected Members

CDesignFrame	Constructs **CDesignFrame** object.
~CDesignFrame	Destructs **CDesignFrame** object.

Message Handlers-Protected Members

OnMDIActivate	Maximize child window.

---

class CDesignVw : public CFormView

The **CDesignVw** class serves as a base class for displaying clean and deteriorated performance data. It is also responsible for dispalying and handling module pictures in the design engine mode.

#include <designvw.h>

Construction/Destruction-Public Members

CDesignVw	Constructs **CDesignVw** object in several ways.
~CDesignVw	Destructs **CDesignVw** object.

Overridables-Protected Members

DoDataExchange	DDV/DDX support.
OnActivateView	Called when view is activated.
OnDraw	Called when view is redrawn.
OnInitialUpdate	Called when data are initialized.
OnUpdate	Called when data are updated.
Serialize	Serializes **CDesignVw** member data.

Operations-Public Members

ActivateSpool	Activates specified spool.
DeleteButtons	Deletes all pictures for specified module in **CModule::m_arrButton** array.
DelModule	Deletes module picture at specified index in **CModule::m_arrButton** array.
FocusModule	Sets focus on specified module.
GetDocument	Attaches appropriate document to view.
GetFocusID	Obtains module ID which currently has focus.
GetFocusSpool	Obtains spool number that currently has focus.
Initialize	Initializes spools.
InitModuleMember	Initializes locations for drawing module pictures on spools.
LoadEngineModules	Loads and redraws all modules.
LoadFocusModule	Loads focus module.
RedrawModules	Redraws all modules.
ResetSpoolNumbers	Reset number of modules on each spool to zero.
UpdateModuleMember	Defines location of module picture.
UpdateSpoolNumber	Updates spool number.

Message Handlers-Protected Members

OnBuildDelAllModules	Called when all modules need to be deleted.
OnBuildDelModule	Called when module at specified index is deleted from **CDesignDoc::m_arrModule** array.
OnClickedEngSpool1	Called when spool 1 is activated.
OnClickedEngSpool2	Called when spool 2 is activated.

OnClickedEngSpool3	Called when spool 3 is activated.
OnModule	Called when a module dialog needs to be displayed.
OnPalette	Called when a module object is added to **CDesignDoc::m_arrModule** array.
OnUpdateBuildDelAllModules	Called when delete all menu is updated.
OnUpdateBuildDelModule	Called when delete menu is updated.
OnUpdatePalette	Called when building module menu is updated.
OnUpdatePerfor	Called when performance build menu is updated.
OnUpdateSpool1	Called when spool 1 menu is updated.
OnUpdateSpool2	Called when spool 2 menu is updated.
OnUpdateSpool3	Called when spool 3 menu is updated.
OnVScroll	Called when view is scrolled.

Data Members-Protected Members

m_Button	A **CBitmapButton** that defines picture of module.
m_MaxNumModules	An integer that specifies maximal possible number of modules.
m_nBrickDataEnd	An integer that specifies last brick data number.
m_nEngineVectorEnd	An integer that specifies last engine vector result number.
m_nFocusModule	An integer ID that specifies the module which currently has the focus.
m_nModuleID	An integer that specifies unique ID for module.
m_nNumModSpool	An array of integers that specifies number of possible modules on each spool.
m_nNumPerforMod	An integer ID that specifies ID for performance module.
m_nSpoolNumber	An integer that specifies spool number.
m_nStationVectorEnd	An integer that specifies last station vector number.
m_ptModOrigin	A **CPoint** that specifies location of module picture.
m_rectPerforMod	A **CRect** that specifies dimensions of **CPerformance** module picture.
m_sizeBM	A **CSize** that specifies dimensions of module picture.

Member Functions

CDesignVw::ActivateSpool

void ActivateSpool(int *nSpoolNumber*, **UINT** *nCase*);

nSpoolNumber Specifies spool number number.

nCase Specifies one of the following cases:

- **DESIGN_ENGINE** Design engine mode.
- **CPF_DESIGN** Edit mode for clean performance run.

Remarks Activates one of the possible three spools. While focusing on new spool, the **CFocusModule** object is deleted and added to the new spool.

See Also **CFocusModule**

CDesignVw::CDesignVw

CDesignVw();

CDesignVw(UINT *nID*);

nID Contains the ID number of a dialog-template resource.

Remarks Constructs a **CDesignVw** object.

CDesignVw::FocusModule

void FocusModule(int *nIndex*);

nIndex Specifies ID number.

Remarks Sets focus on a module picture at specified index.

CDesignVw::GetFocusID

UINT GetFocusID();

Remarks Obtains ID number of module picture that has currently the focus.

Return Value An unsigned integer that contains module ID.

CDesignVw::GetFocusSpool

UINT GetFocusSpool();

Remarks Obtains spool number that has currently the focus:

- **SPOOL1** Spool 1.
- **SPOOL2** Spool 2.
- **SPOOL3** Spool 3.

Return Value An unsigned integer that contains spool number.

CDesignVw::Initialize

void Initialize(UINT *nCase*);

nCase Specifies following case:

- **DESIGN_ENGINE** Design engine mode.

Remarks Initializes spools and loads focus module on **SPOOL1**.

CDesignVw::LoadFocusModule

void LoadFocusModule(int *nSpoolNumber*);

nSpoolNumber Specifies spool number.

Remarks Creates new **CFocusModule** object at specified spool number.

CDesignVw::OnUpdate

virtual void OnUpdate(CView* *pSender*, **LPARAM** *lHint*, **CObject*** *pHint*);

nSender Points to the view that modified the document.

lHint Contains information about the modifications.

pHint Points to an object storing information about the modifications.

Remarks Called after the view's document has been modified. Module pictures are only redrawn if view is in **DESIGN_ENGINE** mode.

CDesignVw::UpdateModuleMember

BOOL UpdateModuleMember(CModule* *PModule*);

PModule Pointer to **CModule** object.

Remarks Updates member data that define location of last drawn module.

Return Value **TRUE** if no error while updating data members.

class CDetemptApp : public CWinApp

The **CDetemptApp** class defines the behaviour of the application PYTHIA 1.1. The name Detempt originates from the old program DETEM (Grewal, 1988). The name PYTHIA was given at a later stage of the development of the program.

#include <detempt.h>

Construction/Destruction-Public Members

CDetemptApp	Constructs **CDetemptApp** object.
~CDetemptApp	Destructs **CDetemptApp** object.

Overridables-Protected Members

InitInstance	Standard initialization.
OpenDocumentFile	Opens document file.

Message Handlers-Protected Members

OnAppAbout	Called when **CAboutDlg** is activated.
OnFileNew	Called when new file is created.
OnFileOpen	Called when existing file is opened.
OnUpdateFileNew	Called when new file menu is updated.
OnUpdateFileOpen	Called when open file menu is updated.

Data Members-Protected Members

m_HeaderDlg	A **CHeader** object that specifies PYTHIA 1.1 title picture.
m_ProjFileStatNew	A **BOOL** variable that specifies status of new file.
m_strFileName	A **CString** that specifies file name.

Data Members-Public Members

m_pChoiceTemplate	A **CMultiDocTemplate** object that handles the mode selection.
m_pCleanPerfTemplate	A **CMultiDocTemplate** object that handles clean performance mode.
m_pClnPrfResTemplate	A **CMultiDocTemplate** object that handles clean performance results.
m_pCpfDPResTemplate	A **CMultiDocTemplate** object that handles clean performance design point results.
m_pCpfODResTemplate	A **CMultiDocTemplate** object that handles clean performance off-design point results.
m_pDesignTemplate	A **CMultiDocTemplate** object that handles design engine mode.
m_pDeterPerfTemplate	A **CMultiDocTemplate** object that handles baseline performance results.
m_pDpfODResTemplate	A **CMultiDocTemplate** object that handles deteriorated performance results.
m_pLinGPATemplate	A **CMultiDocTemplate** object that linear GPA results.
m_pNonLinGPATemplate	A **CMultiDocTemplate** object that handles non-linear GPA results.

class CDetemptDoc : public CDocument

The **CDetemptDoc** class deals with application document of PYTHIA 1.1. Since no member functions are changed, a detailed explanation is omitted.

#include <detemdoc.h>

class CDetemptView : public CView

The **CDetemptView** class deals with application view of PYTHIA 1.1. Since no member functions are changed, a detailed explanation is omitted.

#include <detemvw.h>

class CDeterPerfDoc : public CCleanPerfDoc

The **CDeterPerfDoc** class deals with the objects in deteriorated performance modelling. It is also responsible for carrying out linear and non-linear Gas Path Analysis.
In order to simplify the terminology used in Gas Path Analysis (GPA), the changes in non-measurable independent parameters are represented by the x-axis and the changes in measurable parameters are represented by the y-axis. The whole process of non-linear GPA can be presented in a x-y chart (see also Chapter 2).
#include <dtrprfdo.h>

See also **CDesignDoc, CCleanPerfDoc**

Construction-Protected Members

CDeterPerfDoc	Constructs **CDeterPerfDoc** object.
~CDeterPerfDoc	Destructs **CDeterPerfDoc** object.

Operations-Public Members

AddDeterModules	Adds **CDpfArithy** objects that controls deterioration of modules such as compressor, turbine, burner, convergent and divergent nozzle by adding arithy bricks in the **TURBOMATCH** input file **FOR001.DAT**. These temporarily created objects handle the scaling factors in the **TURBOMATCH** program.
BreakIteration	Checks if user wants to interrupt non-linear GPA iteration process.
CheckODOK	Checks if off-design point data come from clean performance mode.
DelDeterModules	Deletes all temporarily stored modules in **CModule::m_arrModule**.
InitializeModules	Initializes modules for deteriorated performance modelling that can deteriorate independent parameters.
ReadDpfTurbomatch	Reads in performance data from TURBOMATCH output file **FOR004.DAT**.
RunReadTurbomatch	Creates input file **FOR001.DAT** and invokes TURBOMATCH program. After program has finished performance data are read in from file **FOR004.DAT.**
UpdateODSelect	Updates pointer to off-design point selection because if temporary modules are added then the pointer to off-design points have to be relocated.

Overridables-Protected Members

ExportExcel	Exports performance data to spreadsheet file.
ExportFile	Exports deteriorated performance data file (*.dtr) to clean performance data file (*.cln).
GetStatusLoop	Gets the current status of displayed data.
GetTime	Gets the current time of displayed data.
Serialize	Serializes **CDeterPerfDoc** member data.
UpdateLoopNumber	Updates number that specifies number of loops used for performance calculation.
UpdateStatusLoop	Updates the status bar.
UpdateTime	Updates time of displayed data.

Matrix-Public Members

AddFCMCol	Adds **CMatrixCol** object to fault coefficient matrix **CDeterPerfDoc::m_arrFCMMatrix.**
AddICMCol	Adds **CMatrixCol** object to influence coefficient matrix **CDeterPerfDoc::m_arrICMMatrix.**
CreateFCM	Creates fault coefficient matrix FCM.
CreateICM	Creates influence coefficient matrix ICM.
DefineObsFaults	Calculates observed faults.
DelAllFCM	Deletes all columns in fault coefficient matrix array **CDeterPerfDoc::m_arrFCMMatrix.**
DelAllICM	Deletes all columns in fault coefficient matrix array **CDeterPerfDoc::m_arrICMMatrix.**
GetFCMCol	Obtains pointer to **CMatrixCol** object at specified index within **CDeterPerfDoc::m_arrFCMMatrix.**
GetICMCol	Obtains pointer to **CMatrixCol** object at specified index within **CDeterPerfDoc::m_arIFCMMatrix.**
GetNumFCMCols	Gets number of FCM columns.
GetNumICMCols	Gets number of ICM columns.
GetObsFault	Gets the value of observed fault for the specified row.
ResetFaults	Resets faults to zero.
ResetImpFaults	Resets implanted faults to zero.
WriteFCM	Writes out fault coefficient matrix to log file.
WriteICM	Writes out influence coefficient matrix to log file.
WriteImpFaults	Writes out implanted faults to log file.
WriteICMxFCM	Writes out product ICM x FCM

Gas Path Analysis-Public Members

AddDxComponent	Adds **CVectorComponent** object to vector **CDeterPerfDoc::m_arrDxVector** that defines the independent changes.
AddTempModule	Adds **CDpfTemporary** object to temporary module array **CDeterPerfDoc::m_arrTempModule**.
AddTempSVLine	Adds **CSVLine** object to temporary station vector line array **CDeterPerfDoc::m_arrTempSVLine**.
DefineDx	Calculates changes of independent parameters (=.observed faults).
DefineDyDx	Calculates influence coefficient matrix.
DefineYNew	Calculates deteriorated performance using implanted faults.
DelAllTempModules	Deletes all temporarily stored **CDpfTemporary** modules from **CDeterPerfDoc::m_arrTempModule** array.
DelDxVector	Deletes all **CVectorComponent** objects from **CDeterPerfDoc::m_arrDxVector** array.
DelTempSVLine	Deletes all temporarily stored **CSVLine** objects from **CDeterPerfDoc::m_arrTempSVLine** array.
GetBaseAndDeterFromTempData	Gets baseline and deteriorated performance data from temporarily stored modules (**CDeterPerfDoc::m_arrTempModule**) and station vector line (**CDeterPerfDoc::m_arrTempSVLine**).
GetDeltaSum	Gets the sum of errors that are derived from monitored parameters and calculated dependent parameters.
GetDeterFromTempData	Gets deteriorated performance data from temporarily stored modules (**CDeterPerfDoc::m_arrTempModule**) and station vector line (**CDeterPerfDoc::m_arrTempSVLine**)..
GetDxComponent	Gets pointer to **CVectorComponent** object from **CDeterPerfDoc::m_arrDxVector** array.
GetIsIteration	Obtains status of iteration process.
GetNumDxComponents	Gets number of independent changes components in **CDeterPerfDoc::m_arrDxVector** array.
GetNumOuterIterations	Obtains number of iteration used for non-linear GPA.
GetNumTempMods	Gets number of temporarily stored **CDpfTemporary** objects in **CDeterPerfDoc::m_arrTempModule** array.
GetNumTempSVLine	Gets number of temporarily stored station vectors in **CDeterPerfDoc::m_arrTempSVLine**.
GetTempModule	Gets pointer to **CDpfTemporary** object from **CDeterPerfDoc::m_arrTempModule** array.
GetTempSVLine	Gets pointer to **CSVLine** object from **CDeterPerfDoc::m_arrTempSVLine** array.
Initialize	Initializes performance diagnostic calculation.
InitializeDxVector	Initializes changes in independent parameter components in **CDeterPerfDoc::m_arrDxVector** array.
PutBaseAndDeterToTempData	Stores temporarily baseline and deteriorated performance data.
PutDeterToBaseData	Copies deteriorated performance data to baseline data.
PutTempObsToImpFaults	Copies temporarily observed faults to implanted faults.
ResetDpfDeter	Resets deteriorated performance data to zero.
Restore	Restores performance diagnostic calculation.
UpdateTempObsFaults	Updates temporarily observed faults with new calculated observed faults.
WriteExcelFaults	Writes out implanted and observed faults to spreadsheet file.
WriteExcelFCM	Writes out fault coefficient matrix to spreadsheet file.
WriteExcelICM	Writes out influence coefficient matrix to spreadsheet file.
WriteExcelParameters	Writes out several parameters to spreadsheet file.

Message Handlers-Protected Members

OnBuildDiagApproach	Called when diagnostics approach is defined by user (= monitored parameters).
OnDpfExport	Called when data are exported.
OnDpfImport	Called when data are imported.
OnEditSolverOptions	Called when options for Gas Path Analysis need to be changed.
OnRunDpfLinGpa	Called when linear Gas Path Analysis is invoked.
OnRunDpfNonlinGpa	Called when non-linear Gas Path Analysis is invoked.
OnRunDpfTurbomatch	Called when deteriorated performance modelling is invoked.

Data Members-Protected Members

m_arrDxVector	A **CObArray** object that contains changes in independent parameters.
m_arrFCMatrix	A **CObArray** object that contains fault coefficient matrix.
m_arrICMatrix	A **CObArray** object that contains influence coefficient matrix.
m_arrTempModule	A **CObArray** object that contains temporarily stored modules.
m_arrTempSVLine	A **CObArray** object that contains temporarily stored station vectors.
m_bIsIteration	A **BOOL** that specifies status of iteration process in Gas Path Analysis.
m_bIsLinearGpaConverged	A **BOOL** that specifies convergence status of linear Gas Path Analysis.
m_bIsNonlinearGpaConverged	A **BOOL** that specifies convergence status of non-linear Gas Path Analysis.
m_DampingFactor	A **double** that specifies damping factor for non-linear Gas Path Analysis iteration process.
m_DeltaSum	A **double** that specifies sum of **DELTA** (= difference between monitored and calculated value).
m_HOldCursor	A **HCURSOR** handle for old cursor.
m_ICMFault	A **double** that is used for creating influence coefficient matrix
m_nMaxNumOuterIter	An integer that specifies maximal possible number of iterations in non-linear GPA.
m_nNumDpfDeterLoops	An integer that specifies number loops in **TURBOMATCH** run in order to calculate deteriorated performance.
m_nNumOuterIter	An integer that specifies current number of iterations in non-linear GPA run.
m_TimeDpfBase	A **CTime** that specifies time for baseline data.
m_TimeDpfDeter	A **CTime** that specifies time for deteriorated performance data.
m_TimeLinearGpa	A **CTime** that specifies time linear GPA data.
m_TimeNonlinearGpa	A **CTime** that specifies time non-linear GPA data.
m_Tolerance	A **double** that specifies convergence criteria for non-linear GPA.

Member Functions

CDeterPerfDoc::CreateFCM

BOOL CreateFCM();

Remarks Inverts influence coefficient matrix ICM by using a **FORTRAN** subroutine **invnonsq.obj**. The subroutine can invert a non-square matrix. Since the original routine was written in FORTRAN, the original FORTRAN routine was compiled and the object file linked to the PYTHIA 1.1 program using mixed programming languages.

Return Value **TRUE** if no error while creating matrix.

See Also Microsoft (1993) and Donaghy (1991)

CDeterPerfDoc::CreateICM

BOOL CreateICM(UINT *nRunModus*);

nRunModus Defines one of the following diagnostic modes:

- **GPA_LINEAR** Linear Gas Path Analysis.
- **GPA_NONLINEAR** Non-linear Gas Path Analysis.

Remarks Creates influence coefficient matrix by alternatively adding small perturbation to independent parameters.

Return Value **TRUE** if no error while creating matrix.

See Also **CDeterPerfDoc::DefineDxDy** and Grewal (1988)

CDeterPerfDoc::DefineDx

BOOL DefineDx(UINT *nRunModus*);

nRunModus Defines diagnostic modus

Remarks Calculates changes in independent parameters (observed faults).

Return Value **TRUE** if no error.

See Also **CDeterPerfDoc::CreateICM, CDeterPerfDoc::DefineObsFault**

CDeterPerfDoc::DefineDyDx

BOOL DefineDyDx(UINT *nRunModus*);

nRunModus Defines diagnostic modus

Remarks The Gas Path Analysis can be represented as a x-y chart. The ICM is the derivative of the error function y = f(x) at the specified change of independent paramter (see Chapter 2).

Return Value **TRUE** if no error.

See Also **CDeterPerfDoc::CreateICM**

CDeterPerfDoc::DefineObsFaults

BOOL DefineObsFaults(UINT *nRunModus*);

nRunModus Defines diagnostic modes

Remarks Calculates changes in independent parameters (observed faults).

Return Value **TRUE** if no error.
See Also **CDeterPerfDoc::DefineDx**

CDeterPerfDoc::DefineYNew

BOOL DefineYNew(UINT *nRunModus***);**
nRunModus Defines diagnostic modus
Remarks Calculates the deteriorated dependent performance by using implanted faults.
Return Value **TRUE** if no error.
See Also **CDeterPerfDoc::CreateICM**

CDeterPerfDoc::GetDeltaSum

double GetDeltaSum();
Remarks In Gas Path Analysis, a parameter that identifies the success of the diagnostic is the sum of differences in monitored and calculated parameters.
Return Value A **double** value that specifies sum.

CDeterPerfDoc::GetStatusLoop

virtual CString GetStatusLoop(UINT *nViewStatus***);**
nViewStatus Defines one of the following cases:

- **DPF_BASE** Baseline performance.
- **DPF_DETER** Deteriorated performance.
- **GPA_LINEAR** Linear Gas Path Analysis.
- **GPA_NONLINEAR** Non-linear Gas Path

Remarks Obtains the status of the currently shown data regarding convergence criteria.
Return Value A **CString** that specifies status bar loop text.

CDeterPerfDoc::GetTime

virtual CString GetTime(UINT *nViewStatus***);**
nViewStatus Defines currently view status.
Remarks Obtains the time of the currently shown data.
Return Value A **CString** that specifies status bar time text.
See Also **CDeterPerfDoc::GetStatusLoop**

CDeterPerfDoc::Initialize

BOOL Initialize(UINT *nRunModus***);**
nRunModus Defines one of the following cases:

- **DPF_DETER** Deteriorated performance.
- **GPA_LINEAR** Linear Gas Path Analysis.
- **GPA_NONLINEAR** Non-linear Gas Path

Remarks Initializes diagnostics case such as deterioration modelling, linear or non-linear GPA.
Return Value **TRUE** if no error.

CDeterPerfDoc::InitializeDxVector

BOOL InitializeDxVector();
Remarks Initializes vector that represents changes in independent variables.
Return Value **TRUE** if no error.

CDeterPerfDoc::InitializeModules

void InitializeModules();
Remarks In the deteriorated performance mode, some of the modules have additional parameters that are used for deterioration modelling. The following modules can deteriorate independent parameters: compressor, burner, turbine, convergent and divergent nozzle.

CDeterPerfDoc::ReadDpfTurbomatch

BOOL ReadDpfTurbomatch(ifstream(*TMOutputFile***), UINT** *nRunModus***);**
TMOutputFile Specifies file.
nRunModus Defines diagnostic mode.
Remarks Reads in TURBOMATCH output data from file **FOR004.DAT.**
Return Value **TRUE** if no error.
See Also **CDeterPerfDoc::Initialize**

CDeterPerfDoc::Restore

void Restore(UINT *nRunModus***);**
nRunModus Defines diagnostic case
Remarks Restores views, removes temporary modules etc.
See Also **CDeterPerfDoc::Initialize**

RunReadTurbomatch

BOOL RunReadTurbomatch(UINT *nRunModus***);**
nRunModus Defines diagnostic case

Remarks Creates input file **FOR001.DAT** for **TURBOMATCH** run, invokes **TURBOMATCH** and reads in data from **TURBOMATCH** output file **FOR004.DAT**.

Return Value **TRUE** if no error.

CDeterPerfDoc::Serialize

virtual void Serialize(CArchive& *ar***);**

ar A **CArchive** object to serialize to or from.

Remarks Reads or writes this object from or to an archive. The process of serialization distinguishes between following options:

- **CPF_VERSION_11** Clean performance PYTHIA version 1.1.
- **VERSION_10** PYTHIA version 1.0.
- **VERSION_11** PYTHIA version 1.1.

CDeterPerfDoc::UpdateODSelect

void UpdateODSelect(BOOL *bIsDeletingModule***);**

bIsDeletingModule **TRUE** if modules are added; **FALSE** if modules are removed.

Remarks Off-design handles are defined by pointer to modules. Since in a deteriorated performance calculation **CDpfArithy** modules are added in order to deteriorate scaling factors in TURBOMATCH, the off-design point selection (handle) of a particular modules changes the location and therefore the pointer of the handle has to be updated.

---

class CDeterPerfVw : public CClnPrfResVw

The **CDeterPerfVw** class deals with the display of modules that show baseline data.

#include <dtrprfvw.h>

Construction/Destruction-Protected Members

CDeterPerfVw	Constructs **CDeterPerfVw** object.
~CDeterPerfVw	Destructs **CDeterPerfVw** object.

Overridables-Protected Members

OnActivateView	Called when view is active.
OnDraw	Called when view is redrawn.
OnInitialUpdate	Initializes view.
OnUpdate	Updates view.

Operations-Public Members

UpdateListBoxes	Updates all listboxes in view.

---

class CDeterVar : public CBaseElement

The **CDeterVar** class that deals with deterioration in particular modules such as **CCDpfCompressor**, **CDpfTurbine**, **CDpfBurner**, **CDpfNozcon** and **CDpfNozdiv**.

#include <designdo.h>

Construction/Destruction-Public Members

CDeterVar	Constructs **CDeterVar**.

Overridables-Public Members

GetData	Gets data.
GetStatus	Gets status.
Serialize	Serializes **CDeterVar** member data.
UpdateData	Updates data.
UpdateStatus	Updates status.

Data Members-Protected Members

m_Data	A **double** data number 1.
m_NonlinData	A **double** data number 2.
m_nStatus	An integer that specifies status of deteriorated variable.

Member Functions

CDeterVar::CDeterVar

CDeterVar()

Remarks Constructs **CDeterVar** object.

CDeterVar::GetData

virtual double GetData(UINT *nCase***);**

nCase Defines one of the following cases:

- **NOT_DEFINED** Variable is not used.
- **GPA_DESIGN** Variable is used for deteriorated performance calculation.

Remarks Obtains the value from a deteriorated variable.

Return Value A **double** value that specifies deterioration.

CDeterVar::GetStatus

virtual int GetStatus();

Remarks Obtains status of deteriorated variable.
Return Value An integer that defines status.
See Also **CDeterVar::GetData**

---

class CDiagnAppDlg : public CODSelect

The **CDiagnAppDlg** class defines the choice of monitored parameters for Gas Path Analysis.
#include <diappdlg.h>

See also **CODSelect**

Construction/Destruction-Public Members

CDiagnAppDlg	Constructs **CDiagnAppDlg** object.
~CDiagnAppDlg	Destructs **CDiagnAppDlg** object.

Operations-Protected Members

UpdateControls	Updates controls that handle choice of monitored parameters.
UpdateViews	Updates listbox **IDC_CPF_OD_DISPLAY** that displays diagnostics approach.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.
OnInitDialog	Called when diagnostics approach dialog is initialized.

Message Handlers-Protected Members

OnClickedCpfOdAdd	Called when monitored parameter is added to diagnostics approach array **CCleanPerfDoc::m_arrPtrDiapp**.
OnClickedCpfOdClear	Called when all monitored parameter are removed from diagnostics approach array **CCleanPerfDoc::m_arrPtrDiapp**.
OnClickedCpfOdDel	Called when monitored parameter at specified index is added to diagnostics approach array **CCleanPerfDoc::m_arrPtrDiapp**.
OnDblclkCpfOdDisplay	Called when monitored parameter is modified.

---

class CDpfArithy : public CDpfModule

The **CDpfArithy** class creates module objects for deteriorated performance modelling. The objects are used for deteriorating the scaling factors in performance modelling.. The objects are usually created before a **TURBOMATCH** run and destroyed afterwards. The layout of the **CDpfArithy** class is similar to the **CDpfCompressor** class.
#include <dtrprfdo.h>

See also **CDpfModule, CDpfArithy, CDpfCompressor,** Palmer (1995)

---

class CDpfCompressor : CDpfModule

The **CDpfCompressor** objects (= compressor brick) are used in deteriorated performance modelling. Since other modules such as turbine etc. have the same layout as the **CDpfCompressor** class, following classes are not explained in detail: **CDpfTurbine**(= turbine brick), **CDpfNozcon** (= convergent nozzle brick), **CDpfNozdiv** (= divergent nozzle brick and **CDpfBurner** (= burner brick).
#include <dtrprfdo.h>

See also **CModule. CCpfModule, CDpfModule,** Turbomatch manual (Palmer, 1995).

Construction/Destruction-Protected Members

CDpfCompressor	Constructs **CDpfCompressor** object in various ways.

Overridables-Public Members

Draw	Draws module picture.
DrawDialog	Calls module dialog.
Serialize	Serializes data member.
WriteBDCodeWord	Writes out brick data codeword.
WritePlotBDCodeWord	Writes out plotbd codeword.

---

class CDpfModule : public CCpfModule

The **CDpfModule** class creates module objects for deteriorated performance modelling. The objects then are used for linear and non-linear Gas Path Analysis. In a Gas Path Analysis calculation, observed faults are calculated by computer and implanted faults are either entered by user or taken from previous calculation that defined observed faults.
#include <dtrprfdo.h>

See also **CCpfModule, CDpfTemporary**

Destruction-Protected Members

~CDpfModule	Destructs **CDpfModule** object.

Operations-Public Members

InitDeterMembers	Initializes member data of **CDeterVar** object.
InitMembers	Initializes module member data for performance modelling.
InitTempMembers	Initializes temporary **CDpfTemporary** module members.

Overridables-Public Members

Draw	Draws module picture (= button).
DrawDialog	Calls **CModulDlg** in order to update member data.
Serialize	Serializes **CDpfModule** data member.
WriteBDCodeWord	Writes out brick data codeword.
WriteBrickData	Writes out brick data.
WriteExcelFaults	Writes out faults to spreadsheet file.
WritePlotBDCodeWord	Writes out plotbd codeword.

class CDpfODResVw : public CClnPrfResVw

The **CDpfODResVw** class deals with the display of deteriorated performance results.

#include <dpfodvw.h>

See also **CClnPrfResVw**

Construction/Destruction-Protected Members

CDpfODResVw	Constructs **CDpfODResVw** object.
~ CDpfODResVw	Destructs **CDpfODResVw** object.

Operations-Protected Members

UpdateListBoxes	Updates listboxes that display deteriorated performance results.

Overridables-Protected Members

OnActivateView	Called when view is activated. Time and convergence information regarding displayed data is updated in status bar.
OnInitialUpdate	Called when view is initialized.
OnUpdate	Called when displayed data are updated.

class CDpfTemporary : public CDpfModule

The **CDpfTemporary** class creates a temporary module object that assist the Gas Path Analysis calculation. At the end of the calculation the temporary objects are destroyed.

#include <dtrprfdo.h>

See also **CDpfModule**

Construction/Destruction-Public Members

CDpfTemporary	Constructs **CDpfTemporary** in various ways.

Member Functions

CDpfTemporary::CDpfTemporary

Protected **CDpfTemporary()**

CDpfTemporary(CCpfModule *PModule*)

pModule Specifies pointer of module object.

Remarks Constructs **CDpfTemporary** object.

class CEngineVector : public CBaseElement

The **CEngineVector** class that deals with engine vector results from modules.

#include <designdo.h>

Construction/Destruction-Public Members

CEngineVector	Constructs **CEngineVector** in various ways.

Overridables-Public Members

GetNumber	Gets number.
GetStatus	Gets status.
Serialize	Serializes **CEngineVector** member data.
UpdateNumber	Updates number.
UpdateStatus	Updates status.

Data Members-Protected Members

m_nNumber	An integer that specifies the engine vector result number.
m_nStatus	An integer that specifies status of engine vector result.

Member Functions

CEngineVector::CEngineVector

CEngineVector()

CEngineVector(int *nNumber*)

nNumber Specifies engine vector result number.

Remarks Constructs **CEngineVector** object.

CEngineVector::GetNumber

virtual int GetNumber();

Remarks Obtains engine vector result number.

Return Value An integer that defines engine vector result number.

CEngineVector::GetStatus

virtual int GetStatus();

Remarks Obtains status of engine vector result.
Return Value An integer that defines status of engine vector result.

---

class CEngOptionsDlg : public CDialog

The **CEngOptionsDlg** class provides a dialog box that enables the user to define the way of building an engine. In order to build an engine, the user assembles modules/components (=bricks) first. Then by clicking on the module picture, the dialog box for editing module members is invoked. In the **CModulDlg** dialog box, all numbers such as station vector etc. are defined. If data input is desired, then the dialog box has to be left and in the pull-menu options the show fields option has to be activated. Returning to the module dialog, the data fields are shown and the station vector numbers are read only.

#include <enoptdlg.h>

Construction-Public Members

CEngOptionsDlg	Constructs **CEngOptionsDlg** object.

Overridables-Public Members

DoDataExchange	DDX/DDV support.

Message Handlers-Public Members

OnClickedEngHideFields	Called when data fields need to be shown or hidden.
OnClickedEngRenumber	Called when renumbering is automatic.
OnInitDialog	Called when dialog is initialized.

Data Members-Public Members

m_bIsFieldHidden	A **BOOL** that specifies if data fields in **CModulDlg** dialog boxes are shown or hidden.
m_bIsRenumber	A **BOOL** that specifies if numbering of station vector, brick data and engine vector results is automatic.

---

class CExportTextDlg : public CDialog

The **CExportTextDlg** class provides a dialog box that gives the opportunity to add text to the spreadsheet file.

#include <extxtdlg.h>

Construction-Public Members

CExportTextDlg	Constructs **CExportTextDlg** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.

Message Handlers-Public Members

OnInitDialog	Called when dialog is initialized.

Data Members-Public Members

m_strText	A **CString** that contains text that is exported to spreadsheet file.

---

class CFileNameDlg : public CDialog

The **CFileNameDlg** class provides the definition of new file names.

#include <extxtdlg.h>

Construction-Public Members

CFileNameDlg	Constructs **CFileNameDlg** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.

Message Handlers-Public Members

OnInitDialog	Called when dialog is initialized.
OnOK	Called when dialog is closed.

Data Members-Public Members

m_strExt	A **CString** that extension of file.
m_strFileNameNew	A **CString** that file name.
m_strPathName	A **CString** that contains path, file and extension names.
m_strTitle	A **CString** that contains title of dialog box.

---

class CGPABaseVw : public CFormView

The **CGPABaseVw** class serves as a base class for displaying Gas Path Analysis results.

#include <gpabasvw.h>

See also **CLinearGPAVw, CNonLinGPAVw**

Construction/Destruction-Protected Members

CGPABaseVw	Constructs **CGPABaseVw** object.
~CGPABaseVw	Destructs **CGPABaseVw** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.
OnActivateView	Called when view is active.
OnDraw	Called when view is redrawn.

OnInitialUpdate Initializes view.
OnUpdate Updates view.

Operations-Public Members

AddDepenText Adds dependent (measurable) text to listbox.
AddHandleText Adds handle (off-design point setting) text.
AddIndepenText Adds independent (non-measurable) text to listbox.
GetDocument Gets document that is attached to view.
ResetAllListBoxes Deletes contents of all listboxes.
ResetListBox Deletes content of specified listbox.
SetDepenTextID Sets base ID for dependent text.
SetIndepenTextID Sets base ID for independent text.
UpdateAllBoxes Updates all boxes that display data.
UpdateDeltaBox Updates data in **DELTA** listbox.

Message Handlers-Protected Members

OnVScroll Called when scroll bar is changed.
OnSelChangeGpaDelta Called when delta window is scrolled.
OnSelChangeGpaDepen Called when dependent parameter window is scrolled.
OnSelChangeGpaImpFault Called when implanted fault window is scrolled.
OnSelChangeGpaIndepen Called when independent parameter window is scrolled.
OnSelChangeGpaObsFault Called when observed fault window is scrolled.

Data Members-Protected Members

m_DeltaSum A **float** that specifies **DELTA** sum.
m_nBDTextBase An unsigned integer that specifies brick data text base.
m_nDeterVarTextBase An unsigned integer that specifies deteriorated text base.
m_RMSError A **float** that specifies RMS error.

Data Members-Public Members

m_scrDepen A **CScrollBar** object that specifies independent parameter scrollbar.
m_scrIndepen A **CScrollBar** object that specifies dependent parameter scrollbar.

Member Functions

CGPABaseVw::AddDepenText

void AddDepenText();

Remarks Constructs the text string for measurable changes (dependent parameter).

CGPABaseVw::AddHandleText

void AddHandleText();

Remarks Constructs handle text string (off-design point setting).

CGPABaseVw::AddIndepenText

void AddIndepenText(UINT *nModuleNameID*, **int** *nNumber*, **int** *nIndex*);

nModuleNameID ID that defines the module name.
nNumber Number that specifies the occurrence of the module (e.g. compressor 2)
nIndex Index that specifies text string ID

Remarks Constructs the text string for non-measureable changes (= faults).

CGPABaseVw::SetDepenTextID

void SetDepenTextID(UINT *nButtonModule*);

nButtonModule ID that specifies module (e.g. **IDS_ENG_COMPRESSOR** for compressor module etc.).

Remarks Sets module name ID by calling **friend** function in **CModulDlg**.

See Also **CModulDlg**

CGPABaseVw::SetIndepenTextID

void SetIndepenTextID(UINT *nButtonModule*);

nButtonModule ID that specifies module (e.g. **IDS_ENG_COMPRESSOR** for compressor module etc.).

Remarks Sets module name ID by calling **friend** function in **CModulDlg**.

See Also **CModulDlg**

---

class CGPAIterDlg : public CDialog

The **CGPAIterDlg** class provides the settings for iteration process in Gas Path Analysis calculation.

#include <iterdlg.h>

Construction/Destruction-Protected Members

CGPAIterDlg Constructs **CGPAIterDlg** object.

Overridables-Protected Members

DoDataExchange DDX/DDV support.

Data Members-Public Members

m_DampingFactor A **double** that specifies damping factor of iteration process.

m_Tolerance	A **double** that specifies convergence criteria.
m_nMaxNumOuterIter	An integer that specifies maximal number of iterations.
m_ICMFault	A **double** that specifies perturbations for creating influence coefficient matrices (ICM).

class CHeader : public CDialog

The **CHeader** class provides the title picture of PYTHIA 1.

#include <headdlg.h>

Construction/Destruction-Protected Members

CHeader	Constructs **CHeader** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.

Message Handlers-Protected Members

OnInitDialog	Called when dialog is initialized.
OnButton1	Called when button is clicked.

Operations-Protected Members

Create	Called when dialog is created.
CenterWindow	Centers dialog on screen.

Data Members-Public Members

m_BMModule	A **CBitmapButton** member that contains header picture.
m_pOwnerWnd	A **CWnd** member data that points to owner window.

class CImportExportDlg : public CDialog

The **CImportExportDlg** class provides the dialog box that deals with importing or exporting data.

#include <impexdlg.h>

Construction/Destruction-Protected Members

CImportExportDlg	Constructs **CImportExportDlg** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.

Message Handlers-Protected Members

OnInitDialog	Called when dialog is initialized.
OnOK	Called when dialog is closed.

Data Members-Public Members

m_bMode	A **BOOL** that specifies mode (import = **IMPORT_DATA** and export = **EXPORT_DATA**).
m_nViewStatus	An unsigned integer that specifies view status of displayed data.
m_nDataSource	An unsigned integer that specifies source of data (**VERSION_10**, **VERSION_11**, **EXCEL** etc).

class CLinearGPAVw : public CGPABaseVw

The **CLinearGPAVw** is responsible for displaying linear Gas Path Analysis results.

#include <gpalinvw.h>

See also **CGPABaseVw**, **CNonLinGPAVw**

Construction/Destruction-Protected Members

CLinearGPAVw	Constructs **CLinearGPAVw** object.
~CLinearGPAVw	Destructs **CLinearGPAVw** object.

Overridables-Protected Members

OnActivateView	Called when view is active.
OnDraw	Called when view is redrawn.
OnInitialUpdate	Initializes view.
OnUpdate	Updates view.

class CMainFrame : public CMDIFrameWnd

The **CCleanPerfVw** class deals with the display of modules that describe a gas turbine engine. It is also responsible for calling module dialogs in order to specify module data such as brick data, station vector data, engine vector result data etc.

#include <mainfrm.h>

Construction/Destruction-Protected Members

CMainFrame	Constructs **CMainFrame** object.
~ CMainFrame	Destructs **CMainFrame** object.

Operations-Public Members

CreateOrActivateFrame	Called when frame is created or activated.
ViewFile	Displays contents of file by using editor.

Message Handlers-Public Members

OnActivate	Calls base class.
OnCalculator	Called when **WINDOWS** calculator is needed.
OnCreate	Called when frame is created.
OnPaint	Calls base class.
OnUpdateViewCpfDesign	Called when design menu is updated.
OnUpdateViewCpfDp	Called when design point view menu is updated.
OnUpdateViewCpfOd	Called when off-design point view menu is updated.
OnUpdateViewDpfDesign	Called when baseline view menu is updated.
OnUpdateViewDpfLinearGpa	Called when linear GPA view menu is updated.
OnUpdateViewDpfNonlinearGpa	Called when non-linear GPA view menu is updated.
OnUpdateViewDpfOd	Called when deteriorated view menu is updated.
OnViewCpfDesign	Called when design view.
OnViewCpfDp	Called when design point view.
OnViewCpfOd	Called when off-design point view.
OnViewDpfDesign	Called when baseline view.
OnViewDpfLinearGpa	Called when linear GPA view.
OnViewDpfNonlinearGpa	Called when non-linear GPA view.
OnViewDpfOd	Called when deteriorated performance view.
OnViewLogFile	Displays contents of log file using text editor.
OnViewMatrixFile	Displays contents of matrix file using text editor.
OnViewTmErrorFile	Displays contents of error file using text editor.
OnViewTmLogFile	Displays contents of **TURBOMATCH** log file using text editor.
OnViewTmResultFile	Displays contents of **TURBOMATCH** result file using text editor.

Data Members-Public Members

m_wndStatusBar	A **CStatusBar** data member.
m_wndToolBar	A **CToolBar** data member.

---

class CMatrixCol : public CBaseElement

The **CMatrixCol** class deals with the creation of indivual columns of matrices with its rows.

#include <dtrprfdo.h>

Construction-Protected Members

CMatrixCol	Constructs **CMatrixCol** object.

Overridables-Protected Members

Serialize	Serializes **CMatrixCol** member data.

Operations-Public Members

GetNumRows	Gets number of rows in specified column of the **CMatrixCol::m_arrMatrixCol** array.
AddRow	Adds **CCoefficient** object to the **CMatrixCol::m_arrMatrixCol** array.
GetRow	Gets pointer of row within the **CMatrixCol::m_arrMatrixCol** array.
DelRow	Deletes **CCoefficient** object at specified index from **CMatrixCol::m_arrMatrixCol** array.
DelAllRows	Deletes all **CCoefficient** objects from **CMatrixCol::m_arrMatrixCol** array.

Data Members-Protected Members

m_arrMatrixRow	A **CObArray** that contains **CCoefficient** row objects.

---

class CModulDlg : public CDialog

The **CModulDlg** class provides the dialog facility of modules. The class CModulDlg is so designed that it can view data from engine design, clean performance and deteriorated performance modelling in the same dialog boxes. In the **CModulDlg** class following data can be shown: brick data, station vector, engine vector results, possible variables, additional parameters and implanted faults.

#include <moduldlg.h>

Friend Classes **CGPABaseVw, CCleanPerfDoc, CDeterPerfDoc**

See also **CDesignDoc, CCleanPerfDoc, CDeterPerfDoc**

Construction/Destruction-Protected Members

CModulDlg	Constructs **CModulDlg** object in various ways.
~CModulDlg	Destructs **CModulDlg** object.

Overridables-Public Members

DoDataExchange	DDX/DDV support.
OnInitDialog	Called when dialog is initialized.
OnOK	Called when dialog is closed and data are saved.

Operations-Public Members

AddAddParameter	Adds additional parameter dialog boxes.

AddBrickData	Adds brick data dialog boxes.
AddDeterVar	Adds fault dialog boxes.
AddEngineVector	Adds engine vector result dialog boxes.
AddStationVector	Adds station vector dialog boxes.
AddSVNumber	Adds station vector number dialog boxes.
AddVariable	Adds possible variable dialog boxes.
DisplayDblMessage	Displays error message when **double** data is wrong.
DisplayIntMessage	Displays error message when **int** data is wrong.
EditDisplay	Called when station vector data needs to be updated.
FillStationVector	Fills station vector dialog boxes.
InitAddParameter	Initializes additional parameter data member.
InitBrickData	Initializes brick data member.
InitDeterVar	Initializes fault member.
InitEngineVector	Initializes engine vector result member.
InitStationVector	Initializes station vector member.
InitSVNumber	Initializes station vector number members.
InitVariable	Initializes possible variable members.
SetTextID	Sets text IDs.
ShowDoubleError	Displays **double** data error.
ShowIntError	Displays **int** data error.
UpdateBrickData	Updates brick data member.
UpdateControls	Updates station vector controls.
UpdateDeterVar	Updates fault member.
UpdateEngineVector	Updates engine vector result member.
UpdateSectionFrame	Updates frames of specified section that contains member data.
UpdateStationVector	Updates station vector member.
UpdateSVLine	Updates station vector line with user defined station vectors.
UpdateSVNumber	Updates station vector number member.
UpdateVariable	Updates possible variable member.
VariableOK	Checks if variable selection is okay.
ViewAddParameter	Views additional parameter member.
ViewBrickData	Views brick data member.
ViewDeterVar	Views fault member.
ViewSVNumber	Views station vector number member.

Array handlers-Public Members

AddCpfSVLine	Deletes **CSTationVector** object to **CModulDlg::m_arrCpfSVLine** array.
CreateArrCpfSVLine	Creates **CModulDlg::m_arrCpfSVLine** that temporarily contains user-defined station vectors.
DelAllCpfSVLine	Deletes all **CStationVector** objects from **CModulDlg::m_arrCpfSVLine** array.
DelCpfSVLine	Deletes **CStationVector** objects at specified index from **CModulDlg::m_arrCpfSVLine** array.
GetAddPara	Obtains pointer to **CAddParameter** object at specified index from **CModulDlg::m_arrAddPara** array.
GetBD	Obtains pointer to **CBrickData** object at specified index from **CModulDlg::m_arrBD** array.
GetCpfSVLine	Obtains pointer to **CStationVector** object at specified index from **CModulDlg::m_arrCpfSVLine** array..
GetDeterVar	Obtains pointer to **CDeterVar** object at specified index from **CModulDlg::m_arrDeterVar** array.
GetEV	Obtains pointer to **CEngineVector** object at specified index from **CModulDlg::m_arrEV** array..
GetNumAddPara	Gets number of additional parameter in the **CModulDlg::m_arrAddPara** array.
GetNumBD	Gets number of brick data in the **CModulDlg::m_arrBD** array.
GetNumCpfSVLine	Gets number of user defined station vector in the **CModulDlg::m_arrCpfSVLine** array.
GetNumDeterVar	Gets number of faults in the **CModulDlg::m_arrDeterVar** array.
GetNumEV	Gets number of engine vector results in the **CModulDlg::m_arrEV** array.
GetNumSV	Gets number of station vector in the **CModulDlg::m_arrSV** array.
GetNumSVLine	Gets number of station vector in the **CModulDlg::m_arrSVLine** array.
GetNumVar	Gets number of possible variables in the **CModulDlg::m_arrVar** array.

GetSV	Obtains pointer to **CStationVector** object at specified index from **CModulDlg::m_arrSV** array.
GetSVLine	Obtains pointer to **CSVLine** object at specified index from **CModulDlg::m_arrSVLine** array.
GetVar	Obtains pointer to **CVariable** object at specified index from **CModulDlg::m_arrVar** array.

Message Handlers-Protected Members

OnChangeEngBdNo1..13	Called when 1.,2.,... or 13. brick data number is changed.
OnChangeEngEvNo1	Called when 1. engine vector result number is changed.
OnChangeEngEvNo2	Called when 2. engine vector result number is changed.
OnChangeEngSvNo1	Called when 1. station vector number is changed.
OnChangeEngSvNo2	Called when 2. station vector number is changed.
OnChangeEngSvNo3	Called when 3. station vector number is changed.
OnClickedCpfSvAdd	Called when user-defined station vector is added to the **CModulDlg::m_arrCpfSVLine** array.
OnClickedCpfSvClearAll	Called when all user-defined station vectors in the **CModulDlg::m_arrCpfSVLine** array are removed.
OnClickedCpfSvDelete	Called when user-defined station vector at specified index is removed from the **CModulDlg::m_arrCpfSVLine** array.
OnClickedCpfSvEdit	Called when user-defined station vector is edited.
OnClickedDpfDetervarStatus1	Called when 1. fault (independent change) is changed.
OnClickedDpfDetervarStatus2	Called when 2. fault (independent change) is changed.
OnClickedDpfDetervarStatus3	Called when 3. fault (independent change) is changed.
OnClickedEngHelp	Called when help file is needed.
OnClickedEngVisible	Called when invisible dialog boxes are made visible.
OnDblclkCpfSvList	Called when user-defined station vector is edited.

Data Members-Public Members

m_AddParaData1..6	A **double** that specifies 1.,2.,... or 6. additional parameter.
m_arrAddPara	A **CObArray** that contains **CAddParameter** objects.
m_arrBD	A **CObArray** that contains **CBrickData** objects.
m_arrCpfSVLine	A **CObArray** that contains temporarily user-defined **CStationVector** objects.
m_arrDeterVar	A **CObArray** that contains **CDeterVar** objects (= faults or independent changes).
m_arrEV	A **CObArray** that contains **CEngineVector** objects.
m_arrSV	A **CObArray** that contains **CStationVector** objects.
m_arrSVLine	A **CObArray** that contains **CSVLine** objects.
m_arrVar	A **CObArray** that contains **CVariable** objects.
m_BD1..13	A **double** that specifies 1.,2.,... or 13. brick data.
m_bDeterVarStatus1	A **BOOL** that specifies usage of 1. fault (TRUE = fault is used).
m_bDeterVarStatus2	A **BOOL** that specifies usage of 2. fault (TRUE = fault is used).
m_bDeterVarStatus3	A **BOOL** that specifies usage of 3. fault (TRUE = fault is used).
m_bIsShaftEngine	A **BOOL** that specifies shaft engine (TRUE = shaft engine).
m_bIsTimeNew	A **BOOL** that specifies new time (TRUE = data are saved).
m_bIsVisible	A **BOOL** that specifies visibility (TRUE = specified dialog boxes are shown).
m_bVar1	A **BOOL** that specifies 1. possible variable (TRUE = specified variable is used).
m_bVar2	A **BOOL** that specifies 2. possible variable (TRUE = specified variable is used).
m_bVar3	A **BOOL** that specifies 3. possible variable (TRUE = specified variable is used).
m_DeterVar1	A **double** that specifies 1. fault (= independent change).
m_DeterVar2	A **double** that specifies 1. fault (= independent change).
m_DeterVar3	A **double** that specifies 1. fault (= independent change).
m_nAddParaTextBase	An integer that specifies base text ID for additional parameter.
m_nBD1..13	An integer that specifies 1.,2.,... or 13. brick data number.
m_nBDTextBase	An unsigned integer that specifies base text ID for brick data.
m_nDeterVarTextBase	An unsigned integer that specifies base text ID for faults.
m_nEV1	An integer that specifies 1. engine vector result number.
m_nEV2	An integer that specifies 2. engine vector result number.
m_nEVTextBase	An unsigned integer that specifies base text ID for engine vector results.
m_nMaxBDNumber	An integer that specifies maximal number of brick data.
m_nMaxSVNumber	An integer that specifies maximal number of station vector.
m_nSV1	An integer that specifies 1. station vector number.

m_nSV2	An integer that specifies 2. station vector number.
m_nSV3	An integer that specifies 3. station vector number.
m_nSVTextBase	An unsigned integer that specifies base text ID for station vectors.
m_nSVTextItemBase	An unsigned integer that specifies base text ID for station vector items.
m_nVarTextBase	An unsigned integer that specifies base text ID for possible variables.
m_nViewStatus	An integer that specifies view status of displayed data.
m_StaticAddPara	A **CStatic** array that specifies additional parameter texts.
m_StaticBD	A **CStatic** array that specifies brick data texts.
m_StaticDeter	A **CStatic** array that specifies fault texts.
m_StaticEV	A **CStatic** array that specifies engine vector result texts.
m_StaticSV	A **CStatic** array that specifies station vector texts.

Member Functions

CModulDlg::AddAddParameter

void AddAddParameter(UINT *nData*, **UINT** *nText*, **int** *nIndex*, **BOOL** *bReadOnly*);

nData Dialog data ID.

nText Text ID.

nIndex Index of dialog box.

bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds addtional parameter boxes to **CModulDlg** dialog.

CModulDlg::AddBrickData

void AddBrickData(UINT *nIndex*,**UINT** *nBDNumber*,**UINT** *nBDData*,**UINT** *nBDText*, **BOOL** *bReadOnly*);

nIndex Index of dialog box.

nBDNumber Brick data number dialog box index.

nBDData Brick data dialog box index.

nBDText Brick data text index.

bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds brick data boxes to **CModulDlg** dialog.

CModulDlg::AddDeterVar

void AddDeterVar(UINT *nDeterVarStatus*,**UINT** *nDeterVar*,**UINT** *nDeterVarText*, **BOOL** *bReadOnly*);

nDeterVarStatus Fault status dialog box index.

nDeterVar Fault dialog box index.

nDeterVarText Fault text index.

bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds fault boxes to **CModulDlg** dialog. A fault is a change in independent parameters (= change in non-measurable parameters).

CModulDlg::AddEngineVector

void AddEngineVector(UINT *nIndex*,**UINT** *nEVNumber*, **UINT** *nTextEV*, **BOOL** *bReadOnly*);

nIndex Index of dialog box.

nEVNumber Engine vector result number dialog box index.

nTextEV Engine vector result text index.

bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds engine vector result boxes to **CModulDlg** dialog.

CModulDlg::AddStationVector

void AddStationVector(BOOL *bReadOnly*);

bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds user-defined station vectors to **CModulDlg** dialog only if the view status is **CPF_DESIGN**. For all other cases the tool boxes for adding/deleting station vectors is not displayed.

CModulDlg::AddSVNumber

void AddSVNumber(UINT *nIndex*, **UINT** *nSVNumber*, **UINT** *nSVText*, **BOOL** *bReadOnly*);

nIndex Index of dialog box.

nSVNumber Station vector number dialog box index.

nSVText Station vector text index.

bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds user-defined station vector numbers to **CModulDlg** dialog. If view status is **ENGINE_DESIGN** user can change station numbers when data fields are hidden. In case the data fields are shown, station numbers can not be changed.

See Also **CEngOptionsDlg**

CModulDlg::AddVariable

void AddVariable(UINT *nVar*,**UINT** *nTextVar*, **BOOL** *bReadOnly*);

nIndex Index of dialog box.

nTextVar Possible variable dialog box index.
bReadOnly **TRUE** if data dialog box is read only.

Remarks Adds possible variable box to **CModulDlg** dialog.

CModulDlg::CModulDlg

CModulDlg(UINT *nIDD*, **CWnd*** *pParent*);
CModulDlg(CObArray* *arrBD*, **CObArray*** *arrSV*, **CObArray*** *arrSVLine*, **CObArray*** *arrEV*, **CObArray*** *arrVar*, **CObArray*** *arrAddPara*, **UINT** *nButtonModule*, **int** *nViewStatus*);
CModulDlg(CObArray* *arrBD*, **CObArray*** *arrSV*, **CObArray*** *arrSVLine*, **CObArray*** *arrEV*, **CObArray*** *arrVar*, **CObArray*** *arrAddPara*, **CObArray*** *arrDeterVar*, **UINT** *nButtonModule*, **int** *nViewStatus*);

nIDD Contains the ID number of a dialog-box template resource..
pParent Points to the parent window object.
arrBD Contains reference to brick data array.
arrSV Contains reference to station vector array.
arrSVLine Contains reference to station vector line array.
arrEV Contains reference to engine vector result array.
arrVar Contains reference to possible variable array.
arrAddPara Contains reference to additional parameter array.
nButtonModule Contains ID of module.
nViewStatus Contains one of the following view status:

- **DESIGN_ENGINE** Design engine mode.
- **CPF_DESIGN** Clean performance edit mode.
- **CPF_DP** Clean performance design point mode.
- **CPF_OD** Clean performance off-design point mode.
- **DPF_BASE** Deteriorated performance baseline mode.
- **DPF_DETER** Deteriorated performance deteriorated mode.

arrDeterVar Contains reference to fault array.

Remarks Constructs a **CModulDlg** object.

CModulDlg::CreateArrCpfSVLine

void CreateArrCpfSVLine();

Remarks A temporarily stored copy of the **CModulDlg::m_arrSVLine** array is created (**CModulDlg::m_arrCpfSVLine**). In case the user wants to save the newly defined station vectors, the **CModulDlg::m_arrSVLine** is replaced by the temporarily stored copy **CModulDlg::m_arrCpfSVLine**.

CModulDlg::FillStationVector

void FillStationVector();

Remarks Displays the user-defined station vectors.

CModulDlg::OnOK

virtual void OnOK();

Remarks Depending on the view status, displayed data are updated in the corresponding arrays.

See Also **CModulDlg::CModulDlg**

CModulDlg::SetTextID

void SetTextID(UINT *nButtonModule*);
nButtonModule Contains ID of module.

Remarks Since each module has different numbers of brickdata etc, each displayed data has different text. Therefore for each module a set of text string is stored. Each set starts with a base number..

CModulDlg::ViewBrickData

void ViewBrickData(UINT *nIDNumber*, **UINT** *nIDData*, **BOOL** *bCheckRemove*);
nIDNumber Brick data number dialog index.
nIDData Brick data dialog index.
bCheckRemove **TRUE** if brick data can be removed from display.

Remarks Displays brick data. In some view cases, it is possible to hide unused brick data permanently.

CModulDlg::ViewDeterVar

void ViewDeterVar(UINT *nIDDeterVarStatus*, **UINT** *nIDDeterVar*);
nIDDeterVarStatus Fault status dialog index.
nIDDeterVar Fault dialog index.

Remarks Displays fault dialog boxes.

CModulDlg::ViewSVNumber

void ViewSVNumber(UINT *nIDSVNumber*, **BOOL** *bCheckRemove*);
nIDSVNumber Station vector number dialog index.
bCheckRemove **TRUE** if brick data can be removed from display.

Remarks Displays station vector numbers. In some view cases, it is possible to hide unused station vector numbers.

---

class CModule : public CObject

The **CModule** class serves as a base class for defining modules of an engine. In Turbomatch terminology module is represented by brick. A **CModule** object contains information on brick data, station vectors, engine vector results, possible variables and additional parameters. In some modes of the application PYTHIA, the user can edit data, in other modes, the data are read only.

#include <designdo.h>

See also Palmer (1995), **CBrickData, CStationVector, CEngineVector, CAddParameter**

Construction/Destruction-Public Members

~CModule	Destructs **CModule** object.

Drawing-Public Members

DelButton	Deletes picture associated with module from **CModule::m_arrButton** array.
Draw	Prepares for drawing module picture.
DrawDialog	Calls dialog for editing module data.
DrawModule	Draws module picture.
GetButton	Gets pointer of module picture within the **CModule::m_arrButton** array.
GetNumButton	Gets number of picture attached to module of the **CModule::m_arrButton** array.
GetsizeBMEnd	Obtains **CSize** of module picture.
GetSpoolNumber	Gets spool number where module is located.
UpdateSpoolNumber	Updates spool number of module.

Map-Public Members

GetMapNumber	Gets map number that is used in **CCompressor** or **CTurbine** module.

SV Line-Public Members

GetNumSV	Gets number of station vectors for particular module of the **CModule::m_arrSV** array.
GetNumSVLine	Gets number of station vectors of the station vector line array **CModule::m_arrSVLine**.
GetSVLine	Obtains pointer of **CSVLine** object within the **CModule::m_arrSVLine** array.
InitializeSVLine	Initializes **CObArray::m_arrSVLine** array.
UpdateSVLine	Updates reference to the module station vector line **CModule::m_arrSVLine** with reference to the **CDesignDoc::m_arrSVLine**.

Operations-Public Members

DelAddParameter	Deletes **CAddParameter** object at specified index from **CModule::m_arrAddPara** array.
DelBD	Deletes **CBrickData** object at specified index from **CModule::m_arrBD** array.
DeleteBrickParam	Deletes all **CBrickData, CStationVector, CAddParameter, CVariable** and **CEngineVector** objects from their arrays.
DelEV	Deletes **CEngineVector** object at specified index from **CModule::m_arrEV** array.
DelSV	Deletes **CStationVector** object at specified index from **CModule::m_arrSV** array.
DelVar	Deletes **CVariable** object at specified index from **CModule::m_arrVar** array.
GetAddPara	Obtains pointer to **CAddParamter** object at specified index within the **CModule::m_arrAddPara** array.
GetBD	Obtains pointer to **CBrickData** object at specified index from **CModule::m_arrBD** array.
GetBrickDataEnd	Gets last brick data number.
GetEngineVectorEnd	Gets last engine vector result number.
GetEV	Obtains pointer to **CEngineVector** object at specified index from **CModule::m_arrEV** array.
GetModuleID	Obtains module ID.
GetModuleName	Obtains module name.
GetModuleNameID	Obtains module name ID.
GetNumAddPara	Gets number of additional parameters for particular module of the **CModule::m_arrAddPara** array.
GetNumBD	Gets number of brick data for particular module of the **CModule::m_arrBD** array.

GetNumEV	Gets number of engine vector results for particular module of the **CModule::m_arrEV** array.
GetNumVar	Gets number of possible variables for particular module of the **CModule::m_arrVar** array.
GetStationVectorEnd	Gets last station vector number.
GetStatus	Obtains status of module.
GetSV	Obtains pointer to **CStationVector** object at specified index from **CModule::m_arrSV** array.
GetVar	Obtains pointer to **CVariable** object at specified index from **CModule::m_arrVar** array.
InitMembers	Initializes all **CBrickData**, **CStationVector**, **CAddParameter**, **CVariable** and **CEngineVector** objects.
IsTimeNew	Checks if time is version of current data has changed.
Serialize	Serializes **CModule** member data.
UpdateAddParameter	Updates additional parameter for particular module.
UpdateBrickData	Updates brick data number with specified number.
UpdateBrickDataEnd	Updates variable **CModule::m_nMaxBD** that specifies last brick data number.
UpdateEngineVector	Updates engine vector result number with specified number.
UpdateEngineVectorEnd	Updates variable **CModule::m_nMaxEV** that specifies last engine vector result number.
UpdateModuleID	Updates module ID.
UpdateModuleNameID	Updates module name ID.
UpdateStationVector	Updates station vector number with specified number.
UpdateStationVectorEnd	Updates variable **CModule::m_nMaxSV** that specified last station vector number.
UpdateStatus	Updates status.
WriteExcelFaults	Writes out faults that are defined in **CDeterPerfDoc**.

Data Members-Protected Members

m_arrAddPara	A **CObArray** that contains additional parameter objects.
m_arrBD	A **CObArray** that contains brick data objects.
m_arrButton	A **CObArray** that contains picture objects.
m_arrEV	A **CObArray** that contains engine vector result objects.
m_arrSV	A **CObArray** that contains station vector objects.
m_arrSVLine	A **CObArray** that contains all result station vector objects from performance calculations.
m_arrVar	A **CObArray** that contains possible variable objects.
m_bIsTimeNew	A **BOOL** variable that specifies time of current data.
m_nMaxBD	An integer that specifies last brick data number.
m_nMaxEV	An integer that specifies last engine vector result number.
m_nMaxSV	An integer that specifies last station vector number.
m_nModuleID	An integer that specifies module ID.
m_nModuleNameID	An integer that specifies module name ID.
m_nSpoolNumber	An integer that specifies spool number.
m_nStatus	An integer that specifies status of module.
m_sizeBM	A **CSize** that specifies size of picture.
m_strModuleName	A **CString** that specifies module name.

Member Functions

CModule::DrawDialog

virtual void DrawDialog(int *nViewStatus***)**

void DrawDialog(CObArray* *arrBD*, **CObArray*** *arrSV*, **CObArray*** *arrSVLine*, **CObArray*** *arrEV*, **CObArray*** *arrVar*, **CObArray*** *arrAddVar*, **int** *nViewStatus*);

nViewStatus One of the following cases define dialog status:

- **DESIGN_ENGINE** Design engine mode.
- **CPF_DESIGN** Edit mode for clean performance run.
- **CPF_DP** Clean performance design point mode.
- **CPF_OD** Clean performance off-design point mode.
- **DPF_BASE** Deteriorated performance baseline mode.
- **DPF_DETER** Deteriorated performance deteriorated mode.

arrBD Reference to brick data array.

arrSV Reference to station vector array.

arrSVLine Reference to station vector line array.
arrEV Reference to engine vector result array.
arrVar Reference to possible variable array.
arrAddVar Reference to additional parameter array.
Remarks Calls dialog for particular module so that user can edit module data.
See Also **CCpfModule**

CModule::DrawModule

BOOL DrawModule(CWnd* *pParentWnd*, **LPCSTR** *lpszCaption*, **CPoint** *ptModOrigin*, **CPoint** *ptOffsetScroll*);
BOOL DrawModule(CWnd* *pParentWnd*, **LPCSTR** *lpszCaption*, **CRect** *rectModOrigin*, **CPoint** *ptOffsetScroll*);
pParentWnd Pointer to parent window.
lpszCaption Caption of picture.
ptModOrigin Location where picture is drawn.
ptOffsetScroll Offset scroll position of
rectModOrigin Size of picture (Used for **CPerformance** module).
Remarks Draws module picture.

CModule::GetModuleID

int GetModuleID();
Remarks In order to identify pictures, each pictures gets unique number.
Return Value An integer that specifies module ID.

CModule::GetModuleName

CString GetModuleName();
Remarks Obtains module name.
Return Value A **CString** that specifies module name.

CModule::GetModuleNameID

int GetModuleNameID();
Remarks Obtains module name ID such as **IDS_ENG_COMPRESSOR** etc..
Return Value An integer that specifies module name ID.

CModule::GetStatus

UINT GetStatus();
Remarks Obtains status of module.
Return Value An unsigned integer that specifies status of module.
See Also **CModule::UpdateStatus**

CModule::InitMembers

void InitMembers(int *nModuleID*, **int** *nLastBD*, **int** *nLastSV*, **int** *nLastEV*, **int** *nMaxBD*, **int** *nMaxSV*, **int** *nMaxEV*, **int** *nMaxVar*);
nModuleID Module ID.
nLastBD Last brick data number.
nLastSV Last station vector number.
nLastEV Last engine vector result number.
nMaxBD Max numbers of brick data.
nMaxSV Max numbers of station vectors.
nMaxEV Max numbers of engine vector results
nMaxVar Max numbers of possible variables.
Remarks Initializes module members such as brick data, station vectors, engine vector results and possible variables.

---

class CNonLinGPAVw : public CGPABaseVw

The **CNonLinGPAVw** is responsible for displaying non-linear Gas Path Analysis results.
#include <gpanlnvw.h>
See also **CGPABaseVw, CLinearGPAVw**

Construction/Destruction-Protected Members

CNonLinGPAVw	Constructs **CNonLinGPAVw** object.
~CNonLinGPAVw	Destructs **CNonLinGPAVw** object.

Overridables-Protected Members

OnActivateView	Called when view is active.
OnDraw	Called when view is redrawn.
OnInitialUpdate	Initializes view.
OnUpdate	Updates view.

Operations-Public Members

UpdateIterBoxes	Updates faults and RMS errors during iteration process.
UpdateIterSection	Updates number of iterations and **DELTA** sum.

class CODEntryDlg : public CDialog

The **CODEntryDlg** class is used to enter data.

#include <oddatdlg.h>

Construction/Destruction-Public Members

CODEntryDlg	Constructs **CODEntryDlg** object.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.

Message Handlers-Protected Members

OnInitDialog	Called when dialog is initialized.

Data Members-Public Members

m_Data	A **double** that contains the data entry.

class CODSelect : public CModulDlg

The **CODSelect** class is responsible for choosing handles that are used for off-design point calculation.

#include <odseldlg.h>

Construction/Destruction-Public Members

CODSelect	Constructs **CODSelect** object in various ways.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.
OnInitDialog	Called when dialog is initialized.

Operations-Public Members

ActivateCatSettings	Activates specified category listboxes.
AddSelect	Adds **CPtrToSelect** object to **CODSelect::m_arrSelect** array.
DelAllSelect	Deletes all **CPtrToSelect** objects from **CODSelect::m_arrSelect** array.
DelSelect	Deletes **CPtrToSelect** objects at specified index from **CODSelect::m_arrSelect** array.
EditDisplay	Updates selected handle.
GetModule	Obtains pointer to **CModule** object at specified index from **CODSelect::m_arrModule** array.
GetNumMods	Gets number of modules in the **CODSelect::m_arrModule** array.
GetNumSelect	Gets number of handles in the **CODSelect::m_arrSelect** array.
GetSelect	Obtains pointer to **CPtrToSelect** object at specified index from **CODSelect::m_arrSelect** array.
GetSVLine	Obtains pointer to **CSVLine** object at specified index from **CODSelect::m_arrSVLine** array.
InitCatSelect	Fills listbox with category settings such as **BRICK_DATA** and **STATION_VECTOR**.
InitModSelect	Fills combobox with modules of engine.
UpdateCatSettings	Fills listboxes according to category setting.
UpdateControls	Updates control buttons.
UpdateViews	Updates all views.

Message Handlers-Protected Members

OnClickedCpfOdAdd	Called when handle is added to listbox.
OnClickedCpfOdClear	Called when all handles are deleted.
OnClickedCpfOdDel	Called when specified handle is deleted.
OnDblclkCpfOdEdit	Called when specified handle in listbox is edited.
OnDblclkCpfOdDisplay	Called when specified handle in listbox is edited.
OnSelchangeCpfOdCatSelect	Called when category is selected.
OnSelchangeCpfOdModuleSelect	Called when module is selected.

Data Members-Public Members

m_arrModule	A **CObArray** that contains **CModule** objects.
m_arrSelect	A **CObArray** that contains **CPtrToSelect** objects.
m_arrSVLine	A **CObArray** that contains **CSVLine** objects.
m_Data	A **double** that specifies data of handle.

Member Functions

CODSelect::CODSelect

CODSelect(UINT *nIDD*, **CWnd*** *pParent*);

CODSelect(CObArray* *arrModule*, **CObArray*** *arrSVLine*, **CObArray*** *arrODSelect*);

nIDD Contains the ID number of a dialog-box template resource.

pParent Points to the parent window object to which the dialog object belongs.
arrModule Contains reference to module array.
arrSVLine Contains reference to station vector line array.
arrODSelect Contains reference to off-design point selection (=handles) array.
Remarks Constructs **CODSelect** object.

---

class CPaletteBar : public CToolBar

The **CPaletteBar** class creates a tool bar that can be used in the **DESIGN_ENGINE** mode. It simplifies the process of choosing the modules.
#include <palette.h>
See Also Microsoft (1993)

---

class CPrjFileDlg : public CFileDialog

The **CPrjFileDlg** class is used to define new project name and to verify that the name is unique.
#include <prjfildlg.h>

Construction/Destruction-Public Members

CPrjFileDlg	Constructs **CPtrToElement**.

Overridables-Protected Members

DoDataExchange	DDX/DDV support.
OnInitDialog	Called when dialog is initialized.
OnOK	Called when project name is defined and needs to be checked if it is unique.

---

class CPtrToElement : public CBaseElement

The **CPtrToElement** class is used to store pointers.
#include <designdo.h>

Construction/Destruction-Public Members

CPtrToElement	Constructs **CPtrToElement**.

Overridables-Public Members

GetPointer	Gets pointer.
Serialize	Serializes **CPtrToElement** member data.
UpdatePointer	Updates pointer.

Data Members-Protected Members

m_nPointer	An integer that specifies pointer.

---

class CPtrToSelect : public CBaseElement

The **CPtrToSelect** class that deals with storing selection i.e. in clean performance modelling off-design point and in deteriorated performance modelling diagnostics approach (= monitored parameters).
#include <designdo.h>

Construction/Destruction-Public Members

CPtrToSelect	Constructs **CPtrToSelect** in various ways.

Overridables-Public Members

GetCategory	Gets category.
GetData	Gets data for particular selection.
GetItem	Gets item if selection is station vector.
GetModule	Gets pointer to module where data is from.
GetNumber	Gets number of either brick data or station vector.
Serialize	Serializes **CPtrToSelect** member data.
UpdateData	Updates data.
UpdateModule	Updates module pointer.

Data Members-Protected Members

m_Data	A **double** data.
m_nCategory	An integer that specifies category.
m_nItem	An integer that specifies item.
m_nModule	An integer that specifies module pointer.
m_nNumber	An integer that specifies number.

Member Functions

CPtrToSelect::CPtrToSelect

CPtrToSelect()
CPtrToSelect(int *nModule*, **int** *nCategory*, **int** *nNumber*, **double** *Data*)
CPtrToSelect(int *nModule*, **int** *nCategory*, **int** *nNumber*, **int** *nItem*, **double** *Data*)
nModule Pointer to module.
nCategory Defines one of the following cases:

- **BRICK_DATA** Selection is brick data.
- **STATION_VECTOR** Selection is station vector.

nNumber Specifies either brick data number or station vector number.
nItem In case of **STATION_VECTOR** item is specified.
Data Specifies data.
Remarks Constructs **CPtrToSelect** object.
See Also **CPtrToSelect::CPtrToSelect, CStationVector::CStationVector**

CPtrToSelect::GetCategory
int GetCategory();
Remarks Obtains category from selection.
Return Value An integer value.
See Also **CPtrToSelect::CPtrToSelect**

CPtrToSelect::GetData
virtual double GetData();
Remarks Obtains data.
Return Value A **double** value.

CPtrToSelect::GetItem
virtual int GetItem();
Remarks Obtains item in case selection is a **STATION_VECTOR.**
Return Value An integer that defines item.
See Also **CStationVector::CStationVector**

CPtrToSelect::GetModule
int GetModule();
Remarks Obtains pointer to **CDesignDoc::m_arrModule** for selected module.
Return Value An integer value.

CPtrToSelect::GetNumber
virtual int GetNumber();
Remarks In case of **BRICK_DATA,** number is brick data number; in case of **STATION_VECTOR,** number is station vector number.
Return Value An integer that specifies number.

---

class CResSVLine : public CBaseElement
The **CResSVLine** class deals with data that come from clean or deteriorated performance calculation. **CResSVLine** contains all possible data for particular station vector. **CResSVLine** is referenced by **CSVLine.**
#include <designdo.h>
See Also **CSVLine**

Construction/Destruction-Public Members

CResSVLine	Constructs **CResSVLine.**

Overridables-Public Members

GetData	Gets data for particular case.
GetItem	Gets item for particular data.
GetStatus	Gets status.
Serialize	Serializes **CResSVLine** member data.
UpdateData	Updates data for particular case.
UpdateItem	Gets item for particular case.
UpdateStatus	Updates status.

Data Members-Protected Members

m_DPData	A double data number 1.
m_nItem	An integer that specifies an item.
m_nStatus	An integer that sepcifies status.
m_ODData	A double data number 2.

Member Functions
CResSVLine::GetData
virtual double GetData(UINT *nCase*);
nCase Specifies case of data.
Remarks Updates data for particular case.
See Also **CStationVector::GetData**

CResSVLine::GetItem
virtual int GetItem();
Remarks Obtains item for particular data.
See Also **CStationVector::CStationVector**

---

class CStationVector : public CBaseElement
The **CStationVector** class that deals with station vector data from modules.

#include <designdo.h>

Construction/Destruction-Public Members

CStationVector	Constructs **CStationVector** in various ways.

Overridables-Public Members

GetData	Gets data.
GetItem	Gets item.
GetNumber	Gets number.
GetPointer	Gets pointer to station vector line.
GetStatus	Gets status.
Serialize	Serializes **CStationVector** member data.
UpdateData	Updates data.
UpdateItem	Updates item.
UpdateNumber	Updates number.
UpdatePointer	Updates pointer for station vector line.
UpdateStatus	Updates status.

Data Members-Protected Members

m_Data	A **double** data.
m_nItem	An integer that specifies items.
m_nNumber	An integer that specifies the station vector number.
m_nPointer	An integer that specifies pointer to station vector line.
m_nStatus	An integer that specifies status of station vector data.

Member Functions

CStationVector::CStationVector

CStationVector()

CStationVector(int *nNumber*, **int** *nItem*, **double** *Data*)

nNumber Specifies station vector number.

nItem One of the following items are possible:

- **0** Fuel to air ratio.
- **1** Mass flow.
- **2** Static pressure.
- **3** Total pressure.
- **4** Static temperature.
- **5** Total temperature.
- **6** Velocity.
- **7** Area.

Data Specifies station vector data.

Remarks Constructs **CStationVector** object.

See Also Palmer (1995).

CStationVector::GetData

virtual double GetData();

nCase Defines one of the following cases:

- **DESIGN_ENGINE** Design engine mode.
- **DESIGN_POINT** Design point.
- **OFFDESIGN_POINT** Off-design point.
- **CPF_DESIGN** Clean performance design mode.
- **CPF_DP** Clean performance design point.
- **CPF_OD** Clean performance off-design point.
- **GPA_DESIGN** Gas path analysis design mode.
- **DPF_BASE** Baseline performance.
- **DPF_DETER** Deteriorated performance.
- **GPA_LINEAR** Linear gas path analysis.
- **GPA_NONLINEAR** Non-linear gas path analysis.

Remarks Obtains data for particular case.

Return Value The **double** value.

CStationVector::GetItem

virtual int GetItem();

Remarks Obtains item number.

Return Value An integer that defines item number.

CStationVector::GetNumber

virtual int GetNumber();

Remarks Obtains station vector number.
Return Value An integer that defines station vector number.

CStationVector::GetPointer

virtual int GetPointer();

Remarks Obtains pointer to station vector line. Station vector line is an array, that contains all station vectors. Since each module consists only of a selection of station vectors, data for these module station vectors are stored in station vector line (**CDesignDoc::m_arrSVLine**). In order to obtain these data from station vector line, a pointer is being used. A station vector can occur in more than one module. Therefore the first occurrence of a station vector in a brick series gets a positive number, following same station vector with the same number have negativ pointer. A pointer = 0 indicates that there is no station vector e.g. a station vector in a compressor module with pointer = -2 means that there is a station vector number 2 in a previous module (e.g. intake) and data for that particular station vector are taken from station vector line array element 1 (array is zero based).

Return Value An integer that is the pointer to the station vector line.

CStationVector::GetStatus

virtual int GetStatus();

Remarks Obtains status of station vector. This function is used in off-design point calculation in order to define if the station vector is used as a brick data or station vector

Return Value An integer that defines status of station vector.

class CSVLine : public CBaseElement

The **CSVLine** class contains all data for a particular station vector. Each **CSVLine** object is stored in **CDesignDoc::m_arrSVLine** that contains all station vectors. In clean performance or deteriorated performance calculation, the result of the performance calculation is stored in **CSVLine::m_arrResSVLine**. In case the user defines a station vector, its data is stored in **CSVLine::m_arrCpfSVLine**.

#include <designdo.h>

See Also **CDesignDoc::m_arrSVLine**

Construction/Destruction-Public Members

CSVLine	Constructs **CSVLine** object.

Overridables-Public Members

GetNumber	Gets number of station vector.
Serialize	Serializes **CSVLine** member data.
UpdateNumber	Updates number of station vector.

Operations-Public Members

AddCpfSVLine	Add user defined **CStationVector** object.
AddResSVLine	Add result **CStationVector** object.
DelAllCpfSVLine	Deletes all element in array.
DelAllResSVLine	Deletes all element in array.
DelCpfSVLine	Deletes specified element from array.
DelResSVLine	Deletes specified element from array.
GetCpfSVLine	Gets pointer to user defined station vector at specified index.
GetData	Gets station vector data.
GetItem	Gets station vector item.
GetNumCpfSVLine	Gets number of user defined station vectors.
GetNumResSVLine	Gets number of result station vectors.
GetResSVLine	Gets pointer to result station vector.
UpdateData	Updates station vector data.
UpdateItem	Updates station vector item.

Data Members-Protected Members

m_arrCpfSVLine	A **CObArray** that contains user defined station vectors.
m_arrResSVLine	A **CObArray** that contains result station vectors..
m_Data	A **double** that specifies station vector data.
m_nItem	An integer that specifies station vector item.
m_nNumber	An integer that specifies station vector number.

Member Functions

CSVLine::AddCpfSVLine

void AddCpfSVLine(CBaseElement* *PCpfSVLine*);

PCpfSVLine Pointer to station vector object.

Remarks Adds pointer to station vector to **CSVLine::m_arrCpfSVLine**. The **PCpfSVLine** pointer refers to a **CStationVector** object that contains the station number, the item and the actual value for that particular station vector.

CSVLine::AddResSVLine

void AddResSVLine(CResSVLine* *PResSVLine*);

PResSVLine Pointer to station vector object.

Remarks Adds pointer to station vector to **CSVLine::m_arrResSVLine**. The PResSVLine pointer refers to a CstationVector object that contains the result from a clean or a deteriorated performance calculation.

CSVLine::GetCpfSVLine

CStationVector* GetCpfSVLine(int *nIndex*);

nIndex Specifies index.

Remarks Obtains the **CStationVector** object at the specified index that contains information of a user defined station vector.

Return Value A **CStationVector** object.

CSVLine::GetNumber

virtual int GetNumber();

Remarks Obtains number of the station vector.

Return Value A station vector number.

CSVLine::GetNumCpfSVLine

int GetNumCpfSVLine();

Remarks Obtains number of user defined station vectors.

Return Value Number of user-defined station vectors.

CSVLine::GetNumResSVLine

int GetNumResSVLine();

Remarks Obtains number of result station vectors.

Return Value Number of result station vectors

CSVLine::GetResSVLine

CResSVLine* GetResSVLine(int *nIndex*);

nIndex Specifies element in array.

Remarks Obtains the **CStationVector** object at the specified index that contains information of the result station vector.

Return Value A **CStationVector** object.

class CVariable : public CBaseElement

The **CVariable** class that deals with variables that are used in off-design calculations.

#include <designdo.h>

See Also Palmer (1995).

Construction/Destruction-Public Members

CVariable	Constructs a **CVariable** in various ways.

Overridables-Public Members

GetStatus	Gets status of variable.
GetVarBD	Gets case of variable.
GetVarPosition	Gets position.
GetVarSVItem	Gets item if variable is station vector.
GetVarVariable	Gets status if variable is activated.
Serialize	Serializes **CVariable** member data.
UpdateStatus	Updates status of variable.
UpdateVarBD	Updates case of variable.
UpdateVarPosition	Updates position.
UpdateVarSVItem	Updates item.
UpdateVarVariable	Updates status if variable is activated.

Data Members-Public Members

m_bBD	A **BOOL** variable that specifies the case.
m_bVariable	A **BOOL** variable that specifies its use.
m_nPosition	An integer that specifies brick data or station vector as variable
m_nStatus	An integer that specifies status of variable.
m_nSVItem	An integer that specifies item of variable.

Member Functions

CVariable::CVariable

CVariable()

CVariable(BOOL *bVariable*)

bVariable One of the following cases are possible:

- **TRUE** Variable is used in off-design point calculation.
- **FALSE** Variable is not used in off-design point calculation.

Remarks In an off-design point calculation, variables have to be defined that allow the engine to operate at a new handle setting.

CVariable::GetVarBD

BOOL GetVarBD();

Remarks Gets case that specifies if variable is brick data or station vector.

Return Value **TRUE** if brick data, **FALSE** if station vector data.

CVariable::GetVarPosition

int GetVarPosition();

Remarks Obtains the position. If the variable is used as a brick data, then the position defines the brick data of that particular module (e.g. position 0 in compressor module is the surge margin, see Palmer 1995). If the variable is used as a station vector, then the position defines the station vector number.

Return Value An integer that defines position.

CVariable::GetVarSVItem

int GetVarSVItem();

Remarks Obtains item that defines station vector as a variable.

Return Value An integer that specifies the item.

See Also **CVariable::UpdateVarSVItem**

CVariable::GetVarVariable

BOOL GetVarVariable();

Remarks Defines whether the variable is used in an off-design point calculation or whether it is not used.

Return Value **TRUE** if variable is used, **FALSE** if variable is not used.

---

class CVectorComponent : public CBaseElement

The **CVectorComponent** class handles the individual components in vectors.

#include <dtrprfdo.h>

Construction/Destruction-Public Members

CVectorComponent	Constructs a **CVectorComponent**.

Overridables-Public Members

GetData	Obtains data that specifies vector component.
UpdateData	Updates data with new value.

Operations-Public Members

GetMaxData	Obtains data that specifies absolute value of vector.
UpdateMaxData	Updates absolute value of vector.

Data Members-Protected Members

m_Data	A **double** variable that specifies component within vector.
m_MaxData	A **double** variable that specifies absolute value of vector.

Appendix C

TURBOMATCH SIMULATION

Performance simulation with Turbomatch requires an input file (see Table C1), that is designed according to the Turbomatch manual (Palmer, 1995). In Pythia, the process of creating an input file is automatic.

Table C1: Compared Input Files for Turbomatch Simulation of RB211 and TB5000

Brick	*RB211*	*TB5000*
	TURBOMATCH SCHEME - PC VERSION (11/11/94) Turbomatch simulation//// OD SI KE CT FP -1 -1	TURBOMATCH SCHEME - PC VERSION (11/11/94) Turbomatch simulation//// OD SI KE CT FP -1 -1
INTAKE	S1,2 D1,2,3,4 R300	S1,2 D1,2,3,4 R300
COMPRE	S2,3 D5,6,7,8,9,10,0 R301 V5 V6	S2,3 D5,6,7,8,9,10,0 R301 V5 V6
PREMAS	S3,16,4 D12,13,14,15	S3,14,4 D12,13,14,15
ARITHY		D16,17,18,19,20,0,0,0,0
COMPRE	S4,5 D16,17,18,19,20,21,0 R302 V16 V1	S4,5 D25,26,27,28,29,30,0 R302 V25
ARITHY		D32,33,34,35,36,37,38,0,0
PREMAS	S5,17,6 D23,24,25,26	S5,15,6 D41,42,43,44
PREMAS	S17,18,19 D27,28,29,30	
BURNER	S6,7 D31,32,33 R303 W7,6	S6,7 D45,46,47 R303
TURBINE	S7,8 D34,35,36,37,38,39,40,41,302,0 V3	
MIXEES	S8,18,9	S7,15,8
TURBIN	S9,10 D44,45,46,47,48,49,50,51,301,0 V45	S8,9 D48,49,50,51,52,53,54,55,56,0 V49
MIXEES	S10,19,11	
TURBIN	S11,12 D54,55,56,57,58,59,60,61,62,0 V55	S9,10 D58,59,60,61,62,63,64,65,66,0 V59 V58
ARITHY	D63,64,65,66,67,0,0,0,0	D68,69,70,71,72,0,0,0,0
MIXEES	S12,16,13	S10,14,11
DUCTER	S13,14 D72,73,74,75 R304	S11,12 D77,78,79,80 R304
NOZCON	S14,15,1 D76 R305	S12,13,1 D81 R305
PERFOR	S1,0,0	S1,0,0
	D306,78,79,80,305,300,303,0,0,304	D306,83,84,85,305,300,303,0,0,304
	CODEND	CODEND

Brick Data & Station Vector Description	*RB211*	*TB5000*
INTAKE		
Altitude	1 0.000000	1 0.000000
ISA Deviation	2 0.000000	2 0.000000

Mach Number	3 0.000000	3 0.000000
Pressure Recovery	4 0.980000	4 0.980000
LP COMPRESSOR		
Surge Margin (Default = 0.85)	5 -1.000000	5 -1.000000
Spool Speed (Default = 1.0)	6 -1.000000	6 -1.000000
Pressure Ratio	7 4.472000	7 3.659000
Efficiency	8 0.870000	8 0.850000
Error Selector	9 0.000000	9 0.000000
Compressor Map Number	10 1.000000	10 1.000000
SPLITTER		
Bypass Ratio	12 0.030000	12 0.030000
Massflow Loss	13 0.000000	13 0.000000
Pressure Factor	14 1.000000	14 1.000000
Pressure Loss	15 0.000000	15 0.072600
ARITHMETIC		
Copy		16 5.000000
		17 -1.000000
		18 26.000000
		19 -1.000000
		20 6.000000
HP COMPRESSOR		
Surge Margin (Default = 0.85)	16 -1.000000	25 -1.000000
Spool Speed (Default = 1.0)	17 -1.000000	26 -1.000000
Pressure Ratio	18 4.472000	27 1.969400
Efficiency	19 0.870000	28 0.850000
Error Selector	20 1.000000	29 1.000000
Compressor Map Number	21 1.000000	30 1.000000
ARITHMETIC		
Add		32 1.000000
		33 -1.000000
		34 56.000000
		35 -1.000000
		36 301.000000
		37 -1.000000
		38 302.000000
SPLITTER		
Bypass Ratio	23 0.070000	41 0.020000
Mass Flow Loss	24 1.500000	42 0.000000
Pressure Factor	25 1.000000	43 1.000000
Pressure Loss	26 0.000000	44 0.000000
SPLITTER		
Bypass Ratio	27 0.750000	
Mass Flow Loss	28 0.000000	
Pressure Factor	29 1.000000	
Pressure Loss	30 0.000000	
COMBUSTOR		
Pressure Loss	31 0.050000	45 0.060000
Efficiency	32 1.000000	46 1.000000
Fuel Flow (-1 = TET Specified)	33 -1.000000	47 -1.000000
HP TURBINE		
Auxiliary Power Required	34 0.000000	
Non-dimensional Massflow (Default = 0.8)	35 -1.000000	
Non-dimensional Speed (Default = 0.6)	36 -1.000000	

Efficiency	37 0.880000	
Compressor Turbine	38 -1.000000	
Compressor Number	39 2.000000	
Turbine Map Number	40 3.000000	
Auxiliary Work Constant	41 -1.000000	
IP TURBINE		
Auxiliary Power Required	44 0.000000	48 0.000000
Non-dimensional Mass Flow (Default = 0.8)	45 -1.000000	49 -1.000000
Non-dimensional Speed (Default = 0.6)	46 -1.000000	50 -1.000000
Efficiency	47 0.880000	51 0.880000
Compressor Turbine	48 -1.000000	52 -1.000000
Compressor Number	49 1.000000	53 2.000000
Turbine Map Number	50 3.000000	54 3.000000
Power Law Index (-1 = Const. Auxiliary Work)	51 -1.000000	55 -1.000000
Compressor Work (see Arithmetic)		56 0.000000
POWER TURBINE		
Power Required	54 24450000.00	58 3845000.00
Non-dimensional Mass Flow (Default = 0.8)	55 -1.000000	59 -1.000000
Non-dimensional Speed (Default = 0.6)	56 -1.000000	60 -1.000000
Efficiency	57 0.880000	61 0.913500
Relative Rotational	58 1.000000	62 1.000000
Compressor Number	59 0.000000	63 0.000000
Turbine Map Number	60 3.000000	64 3.000000
Auxiliary Work Constant	61 -1.000000	65 -1.000000
Power Turbine	62 -1.000000	66 -1.000000
ARITHMETIC		
Copy	63 5.000000	68 5.000000
	64 -1.000000	69 -1.000000
	65 306.000000	70 306.000000
	66 -1.000000	71 -1.000000
	67 54.000000	72 58.000000
DUCTER		
Reheat Selector Off	72 0.000000	77 0.000000
Pressure Loss	73 0.030000	78 0.030000
Reheat Combustion Efficiency	74 0.000000	79 1.000000
Max Reheat Fuel Flow	75 1000000.00	80 1000000.00
CONVERGENT NOZZLE		
Area Fixed	76 -1.000000	81 -1.000000
PERFORMANCE		
Power for Power Turbine	78 1.000000	83 1.000000
Propeller Efficiency	79 0.000000	84 0.000000
Scaling Index (0 = No Scaling)	80 0.000000	85 0.000000
	-1	-1
Mass Flow at Station 1	1 2 89.800000	1 2 20.750000
TET at Station 7	7 6 1434.0000	7 6 1173.0000
Velocity at Station 15	15 7 50.000000	
Area at Station 12		12 8 1.334000
	-1	-1

Appendix D

PYTHIA APPLICATION

The following Figures (D1 to D7) show a possible application of Pythia to a 3 shaft gas turbine engine.

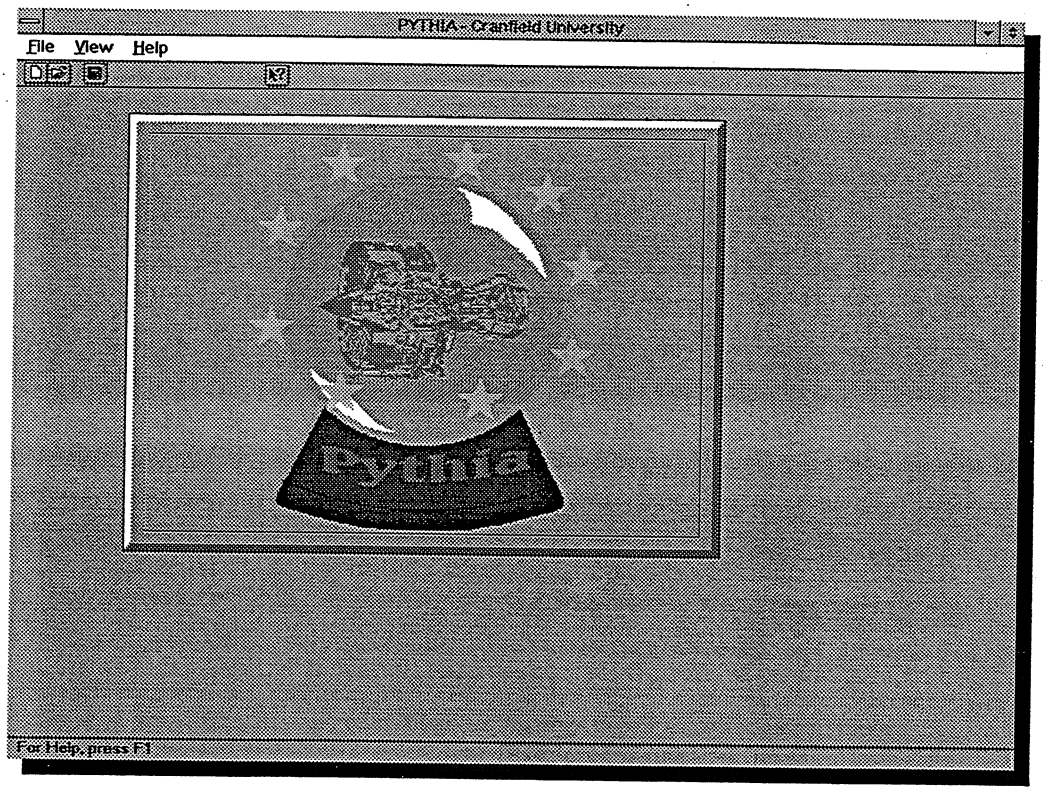
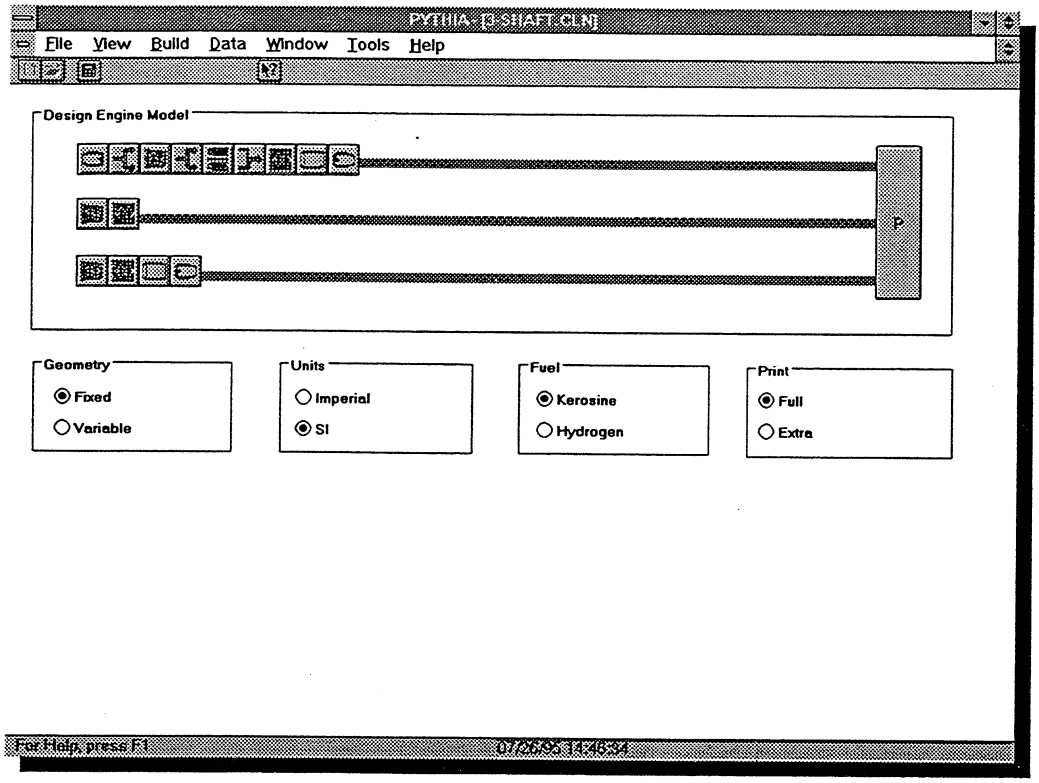


Figure D1: Start of Pythia Application



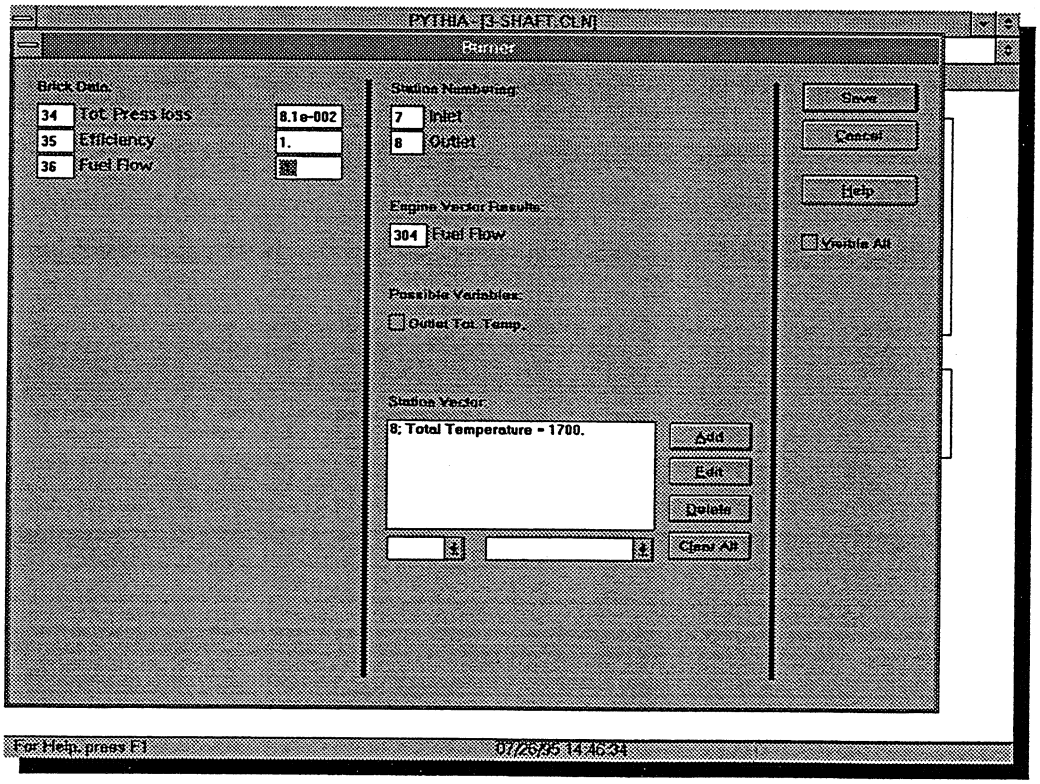


Figure D4: Prepare Clean Performance by Modifying Component Data

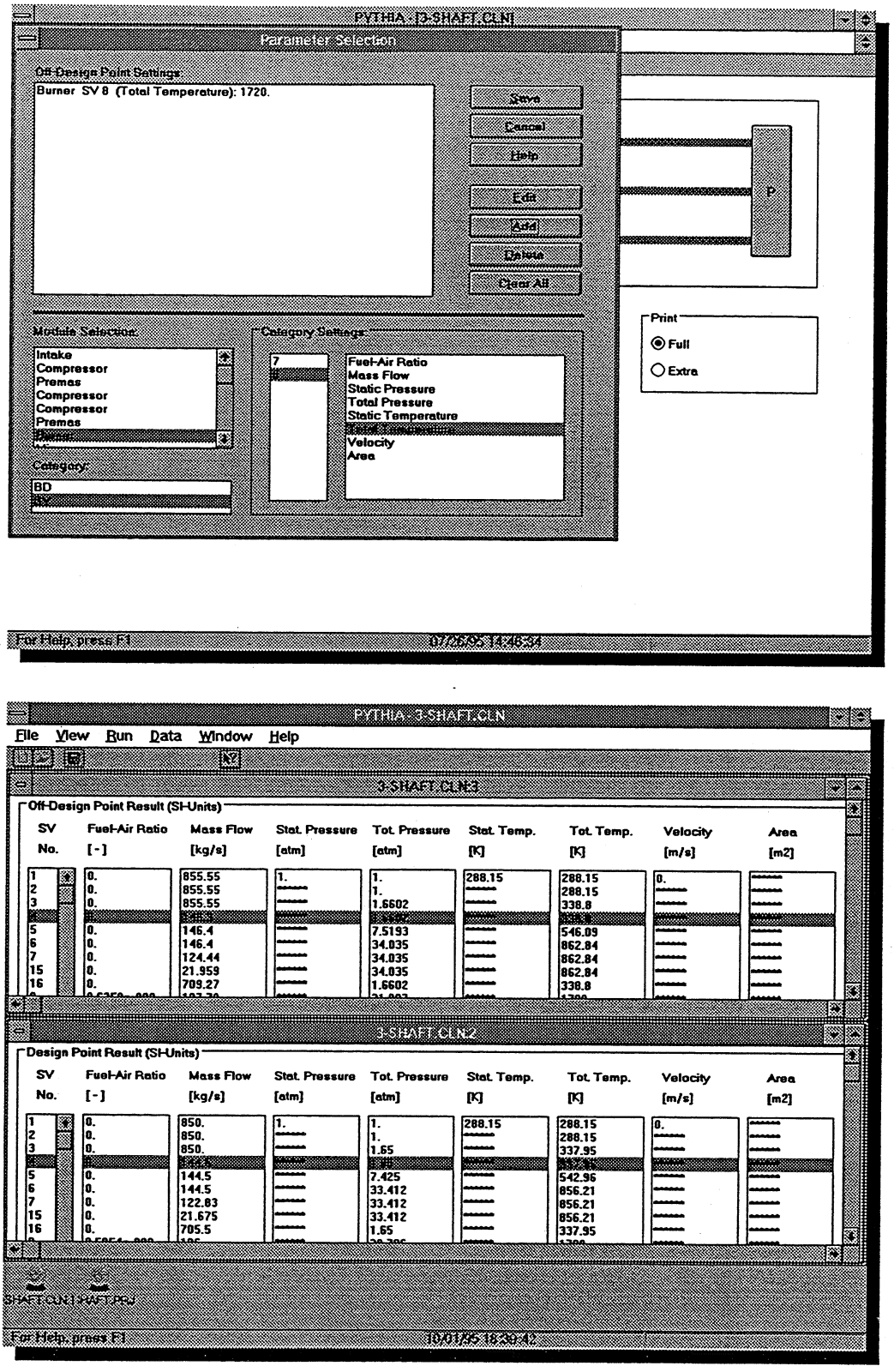


Figure D5: Define Off-design Point Settings and Compare Performance Results (DP-OD)

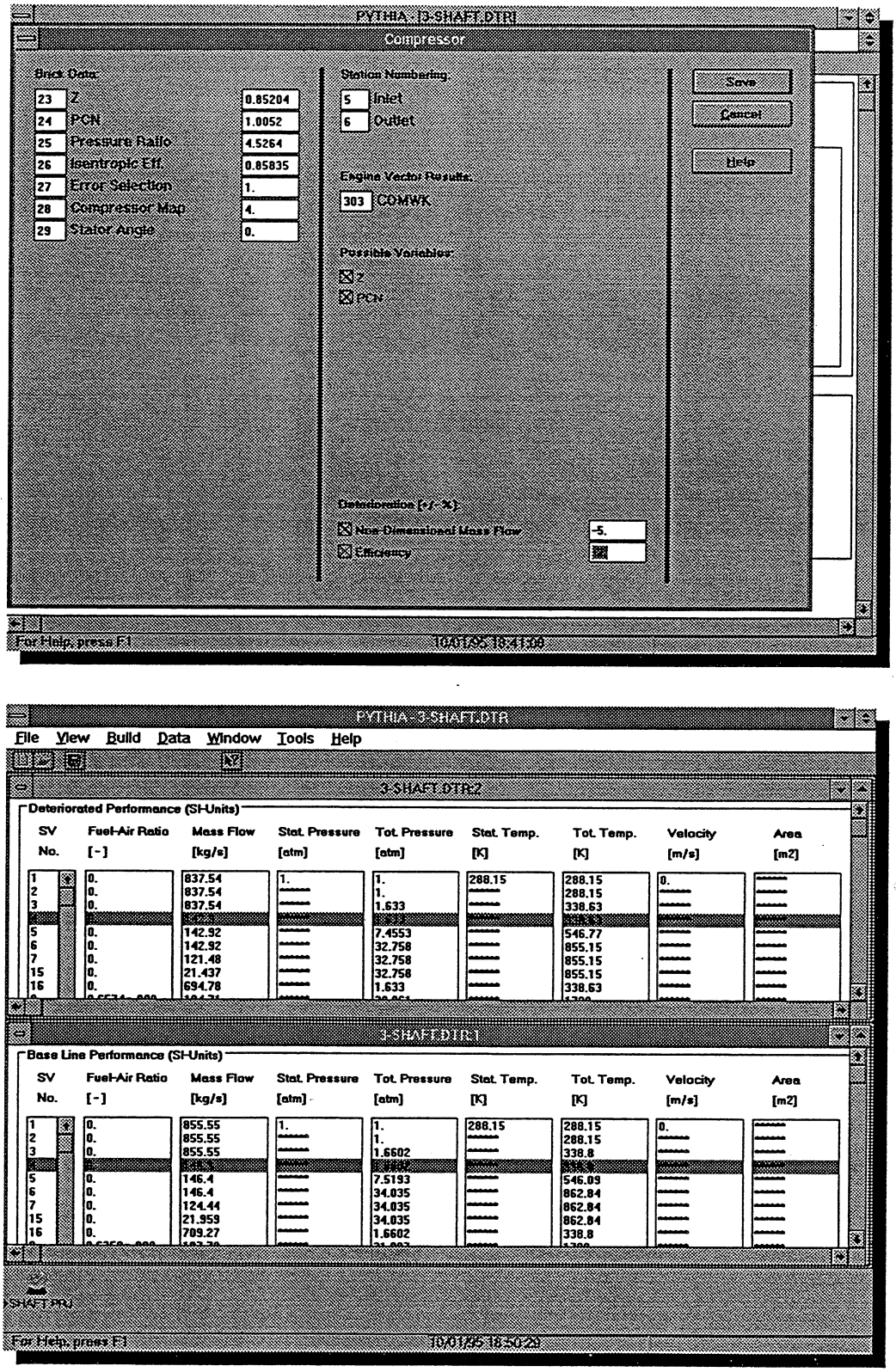


Figure D6: Implant Deterioration and Compare Baseline to Deteriorated Performance

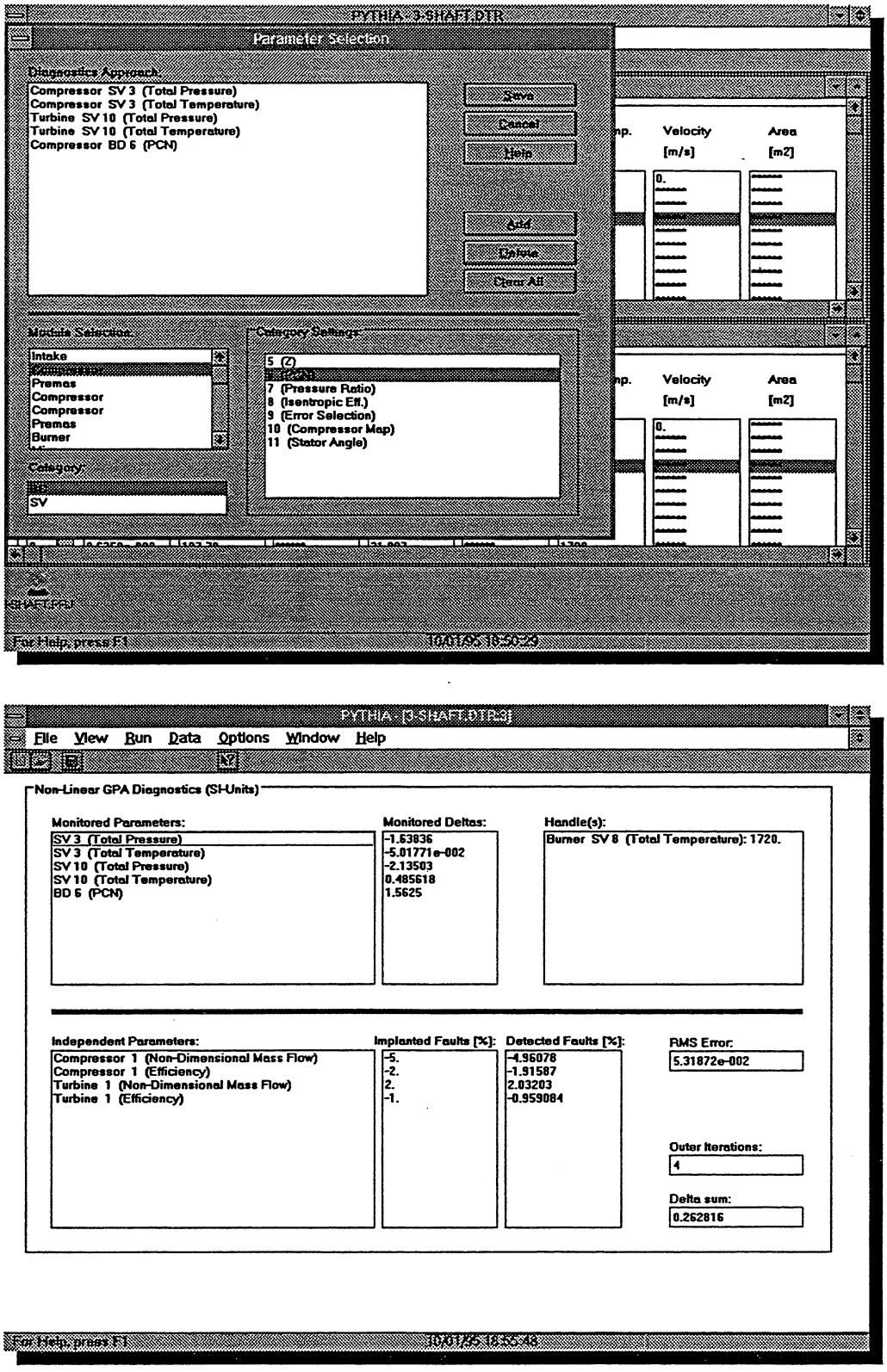


Figure D7: Define Sets of Monitored Parameter and Conduct Non-linear GPA