Parallel Methods for Systems of Nonlinear Equations Applied to Load Flow Analysis

Adriaan Joubert

March 1996
PARALLEL METHODS FOR SYSTEMS OF
NONLINEAR EQUATIONS APPLIED TO LOAD
FLOW ANALYSIS

ADRIAAN JOUBERT

DEPARTMENT OF COMPUTER SCIENCE
QUEEN MARY AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

MARCH 1996

A THESIS SUBMITTED TO THE UNIVERSITY OF
LONDON IN FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
Abstract

Large systems of nonlinear, algebraic equations arise in many transportation problems such as circuit simulation, and gas, water and power distribution. The structure of the system of equations mirrors that of the underlying distribution network. This leads to a large, very sparse system of equations with a very irregular structure. As distribution systems become larger and are interconnected, their detailed, real-time monitoring is hampered by the computational requirements. When modern optimisation techniques are applied to, for example, power systems, the solution of hundreds of such nonlinear systems may be required and the solution speed on a single processor becomes a constraint. Parallel processing promises large reductions in computation time, provided that suitable algorithms can be found.

Developing parallel algorithms for these problems has not been very successful due to their awkward structure. In this thesis novel techniques are investigated which exploit the available parallelism in the solution process more effectively. As a vehicle for this research the modelling of load flow in power systems is used. It is important to apply algorithms to real-world situations, as algorithms developed for theoretical problems often do not perform well in practice. In real-life load flow simulations, constraints are imposed on some variables, and this has significant impact on the efficiency of parallel algorithms.

The purpose of this thesis is thus twofold: firstly, to develop efficient parallel algorithms for linear and nonlinear equations that exploit the structure of the underlying network, and secondly, to develop efficient algorithms for load flow simulation.

A complete overview of the problem area and the algorithms used for load flow problems is given. Approaches to the parallel solution of sparse linear systems, using both iterative and direct methods, are presented. The concept of system reduction is introduced and utilised to provide an algorithm with improved performance on parallel computers. When combining this algorithm with the Newton-Raphson method a novel and significantly faster parallel solver for load flow analysis emerges.

In an attempt to avoid the synchronisation problems encountered when using a global method, such as the Newton-Raphson method with parallel linear systems solution, algorithms that partition the nonlinear system are investigated. Traditional algorithms in this class, such as nonlinear block-SOR, proved unsuccessful. However, it proved possible to transfer some of the system reduction ideas to nonlinear systems, and to construct a competitive algorithm.

A complete load flow analysis program and all algorithms were implemented on the Intel iPSC/860 multiprocessor in Daresbury. The parallel programs are compared to a very efficient sequential implementation.
Acknowledgements

I would like to thank my supervisors, Heather Liddell and Neil Adams for their help and guidance. Special thanks to Gerhard Joubert, Dennis Parkinson, Carlos Falcó Korn, Jonathan Hill and Mike Dracopoulos for reading various drafts of this thesis.

It has been a pleasure to complete this thesis at the Department of Computer Science and the London Parallel Applications Centre at Queen Mary & Westfield College. I would particularly like to thank Jonathan Hill, Andrew Smallbone and Carlos Falcó Korn for many hours of lively debate on parallel computing.

Intel provided me with a simulator for the iPSC/860, which greatly aided program development and for which I am very grateful. Daresbury computer centre provided access to their machines and have been very helpful over the years. The first two years of this research were funded under the EEC SCIENCE grant B/900160.

Last, but not least, I would like to thank Jacque for her unwavering support over the years, and Paul, Jacques and Christian, for putting up with my absences and providing welcome distraction.

This version of the thesis includes changes recommended by my examiners, Dr Nick Gould, Prof Berc Rustem and Prof Donald McDonald.
# Contents

1 Introduction

1.1 Load Flow Analysis and Parallelisation ........................................ 2
1.2 Approaches to parallel solution methods ...................................... 4
1.3 Overview of the thesis ................................................................. 5
1.4 Contribution of this thesis ........................................................... 6

I Load Flow Analysis ........................................................................... 9

2 Load Flow Analysis ........................................................................ 11

2.1 Alternating currents ....................................................................... 12
  2.1.1 Current, voltage and power ....................................................... 12
  2.1.2 Phase angles and reactance ...................................................... 12
  2.1.3 Real and reactive power ............................................................ 13
  2.1.4 Complex representation of voltages and currents ................. 14
  2.1.5 Complex power .................................................................. 15
  2.1.6 Impedance and admittance ....................................................... 15

2.2 Electric system models ................................................................. 16

2.3 Per unit calculations .................................................................... 19

2.4 Power system components .......................................................... 20
  2.4.1 Load flow in a branch .............................................................. 20
  2.4.2 Modelling transmission lines .................................................. 23
  2.4.3 Modelling transformers .......................................................... 25
  2.4.4 Capacitor banks and synchronous compensators ............... 28

2.5 Modelling a power system ........................................................... 30
  2.5.1 Self-admittance of a node ......................................................... 31
  2.5.2 The nodal load flow model ....................................................... 31
  2.5.3 Variables and Parameters ....................................................... 33
  2.5.4 Regulated (PV) busses ............................................................. 34
### CONTENTS

2.6 Data requirements ........................................ 35  
2.6.1 Impedance and line charging data ..................... 36  
2.6.2 Operating conditions .................................. 37  
2.6.3 Regulated bus data .................................... 38  
2.6.4 Transformer data ....................................... 39  
2.6.5 Static capacitor data .................................. 39  
2.7 Test problems ............................................ 39

3 Load Flow Methods ........................................ 42  
3.1 DC load flow analysis .................................... 43  
3.2 The Gauss-Seidel method .................................. 43  
3.3 Newton methods .......................................... 44  
3.4 Decoupled Newton methods ................................ 47  
3.5 The fast decoupled load flow (FDLF) method ............ 49  
3.6 Second order load flow .................................... 54  
3.6.1 Handling PV busses .................................... 57  
3.6.2 Interpretation of the second order load flow method . 58  
3.6.3 Modifications of the SOLF method .................... 60  
3.7 Choice of method .......................................... 61  
3.8 Implementation of the Newton-Raphson method ........... 64  
3.8.1 Sparse data structures for load flow analysis .......... 65  
3.8.2 The admittance matrix and sparse data structures .... 66  
3.8.3 Calculation of power mismatches and the Jacobian .... 68  
3.8.4 Sparse LU-decomposition ............................... 69  
3.8.5 Regulated busses ...................................... 72  
3.8.6 Termination criterion .................................. 73  
3.8.7 Results ................................................ 73  
3.9 Overview of the parallelisation of load flow analysis .... 74  
3.9.1 Parallelisation of the linear system solution ........... 75  
3.9.2 Other nonlinear methods ............................... 78  
3.9.3 Diakoptics ............................................. 78  
3.9.4 Lines of investigation .................................. 79

II Parallel methods for linear systems ................. 83

4 Parallel direct methods ................................ 87  
4.1 Introduction ............................................. 87  
4.2 Reordering a matrix ........................................ 89
4.3 Solving double-bordered block structured systems ........................................... 91

5 Iterative methods ......................................................................................... 98
  5.1 Krylov subspace methods ........................................................................ 98
  5.2 Preconditioning ....................................................................................... 101
  5.3 Block preconditioning ............................................................................. 106
  5.4 Error estimates ....................................................................................... 110
  5.5 Summary ............................................................................................... 112

6 Linear System Reduction ........................................................................... 113
  6.1 Constructing a reduced system ............................................................... 114
  6.2 Reduced systems and iterative methods ................................................. 117
  6.3 A new parallel direct method .................................................................. 120
  6.4 Theoretical comparison of the parallel direct methods ......................... 120
    6.4.1 Complexity analysis for NODE ..................................................... 121
    6.4.2 Complexity analysis for BRANCH .............................................. 123
    6.4.3 Comparison of NODE and BRANCH .......................................... 124

7 Reordering systems ..................................................................................... 126
  7.1 Nested dissection .................................................................................... 127
  7.2 The Kernighan & Lin algorithm .............................................................. 129
  7.3 Elimination tree rotations ....................................................................... 130
  7.4 Spectral bisection ................................................................................... 130
  7.5 Clustering methods ............................................................................... 132
    7.5.1 Branch tearing ............................................................................... 132
    7.5.2 Node tearing ................................................................................... 133
    7.5.3 The clustering algorithm ............................................................... 133
    7.5.4 PABLO ......................................................................................... 137
  7.6 Reordering a power system ..................................................................... 137

8 Implementation and Comparison ............................................................... 139
  8.1 Routines implemented ............................................................................ 139
  8.2 Design of parallel programs .................................................................... 139
  8.3 BLAS-routines and vectorisation ............................................................. 141
  8.4 Duplication of code ............................................................................... 142
  8.5 Implementation of LU decomposition .................................................... 142
  8.6 Implementation of iterative methods ....................................................... 144
  8.7 Testing and debugging ........................................................................... 146
  8.8 Data ....................................................................................................... 147
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.9</td>
<td>The iPSC/860</td>
<td>148</td>
</tr>
<tr>
<td>8.10</td>
<td>Timing</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>8.10.1 Discussion of the results</td>
<td>151</td>
</tr>
<tr>
<td>8.11</td>
<td>Event graphs for the direct algorithms</td>
<td>158</td>
</tr>
<tr>
<td>8.12</td>
<td>Conclusions</td>
<td>161</td>
</tr>
</tbody>
</table>

**III Parallel methods for nonlinear systems**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>The Newton-Raphson method</td>
<td>169</td>
</tr>
<tr>
<td>9.1</td>
<td>Notation and assumptions</td>
<td>169</td>
</tr>
<tr>
<td>9.2</td>
<td>Implementation of the parallel Newton-Raphson method</td>
<td>171</td>
</tr>
<tr>
<td>9.3</td>
<td>Results for the parallel Newton-Raphson method</td>
<td>174</td>
</tr>
<tr>
<td>9.4</td>
<td>Results with PVM</td>
<td>175</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Block methods and descent methods</td>
<td>178</td>
</tr>
<tr>
<td>10.1</td>
<td>Jacobi-Newton and SOR-Newton methods</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td>10.1.1 Convergence of generalised linear methods</td>
<td>180</td>
</tr>
<tr>
<td>10.2</td>
<td>Generalised linear methods and load flow</td>
<td>181</td>
</tr>
<tr>
<td>10.3</td>
<td>Descent directions</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>10.3.1 Another view of this derivation</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td>10.3.2 Descent in multiple directions simultaneously</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>10.3.3 Line searches</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>10.3.4 Convergence</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>10.3.5 Results</td>
<td>190</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Reduced methods</td>
<td>193</td>
</tr>
<tr>
<td>11.1</td>
<td>Reduced nonlinear systems</td>
<td>194</td>
</tr>
<tr>
<td>11.2</td>
<td>Definition of the reduced algorithm</td>
<td>197</td>
</tr>
<tr>
<td>11.3</td>
<td>Convergence of the reduced method</td>
<td>198</td>
</tr>
<tr>
<td>11.4</td>
<td>Initial results</td>
<td>201</td>
</tr>
<tr>
<td>11.5</td>
<td>Corrections</td>
<td>203</td>
</tr>
<tr>
<td>11.6</td>
<td>The multilevel Newton method</td>
<td>205</td>
</tr>
<tr>
<td>11.7</td>
<td>Convergence of the multilevel Newton method</td>
<td>208</td>
</tr>
<tr>
<td>11.8</td>
<td>Results with the multilevel method</td>
<td>209</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Conclusions and further work</td>
<td>214</td>
</tr>
<tr>
<td>12.1</td>
<td>Applicability to other areas</td>
<td>216</td>
</tr>
<tr>
<td>12.2</td>
<td>Further work</td>
<td>216</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.9</td>
<td>The iPSC/860</td>
<td>148</td>
</tr>
<tr>
<td>8.10</td>
<td>Timing</td>
<td>149</td>
</tr>
<tr>
<td>8.10.1</td>
<td>Discussion of the results</td>
<td>151</td>
</tr>
<tr>
<td>8.11</td>
<td>Event graphs for the direct algorithms</td>
<td>158</td>
</tr>
<tr>
<td>8.12</td>
<td>Conclusions</td>
<td>161</td>
</tr>
<tr>
<td>III</td>
<td>Parallel methods for nonlinear systems</td>
<td>165</td>
</tr>
<tr>
<td>9</td>
<td>The Newton-Raphson method</td>
<td>169</td>
</tr>
<tr>
<td>9.1</td>
<td>Notation and assumptions</td>
<td>169</td>
</tr>
<tr>
<td>9.2</td>
<td>Implementation of the parallel Newton-Raphson method</td>
<td>171</td>
</tr>
<tr>
<td>9.3</td>
<td>Results for the parallel Newton-Raphson method</td>
<td>174</td>
</tr>
<tr>
<td>9.4</td>
<td>Results with PVM</td>
<td>175</td>
</tr>
<tr>
<td>10</td>
<td>Block methods and descent methods</td>
<td>178</td>
</tr>
<tr>
<td>10.1</td>
<td>Jacobi-Newton and SOR-Newton methods</td>
<td>178</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Convergence of generalised linear methods</td>
<td>180</td>
</tr>
<tr>
<td>10.2</td>
<td>Generalised linear methods and load flow</td>
<td>181</td>
</tr>
<tr>
<td>10.3</td>
<td>Descent directions</td>
<td>183</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Another view of this derivation</td>
<td>186</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Descent in multiple directions simultaneously</td>
<td>187</td>
</tr>
<tr>
<td>10.3.3</td>
<td>Line searches</td>
<td>187</td>
</tr>
<tr>
<td>10.3.4</td>
<td>Convergence</td>
<td>189</td>
</tr>
<tr>
<td>10.3.5</td>
<td>Results</td>
<td>190</td>
</tr>
<tr>
<td>11</td>
<td>Reduced methods</td>
<td>193</td>
</tr>
<tr>
<td>11.1</td>
<td>Reduced nonlinear systems</td>
<td>194</td>
</tr>
<tr>
<td>11.2</td>
<td>Definition of the reduced algorithm</td>
<td>197</td>
</tr>
<tr>
<td>11.3</td>
<td>Convergence of the reduced method</td>
<td>198</td>
</tr>
<tr>
<td>11.4</td>
<td>Initial results</td>
<td>201</td>
</tr>
<tr>
<td>11.5</td>
<td>Corrections</td>
<td>203</td>
</tr>
<tr>
<td>11.6</td>
<td>The multilevel Newton method</td>
<td>205</td>
</tr>
<tr>
<td>11.7</td>
<td>Convergence of the multilevel Newton method</td>
<td>208</td>
</tr>
<tr>
<td>11.8</td>
<td>Results with the multilevel Newton method</td>
<td>209</td>
</tr>
<tr>
<td>12</td>
<td>Conclusions and further work</td>
<td>214</td>
</tr>
<tr>
<td>12.1</td>
<td>Applicability to other areas</td>
<td>216</td>
</tr>
<tr>
<td>12.2</td>
<td>Further work</td>
<td>216</td>
</tr>
</tbody>
</table>
CONTENTS

A The operators $\otimes$ and $\ominus$ 218
B LU-decomposition of the Jacobian 220
C Matrix vector multiplications 222
D Estimate of the SOR parameter $\omega$ 226
E Matrix structures 230
F Convergence tests with the multilevel method 236
References 248
Index 258
List of Figures

1-1 The structure of a linear system ........................................ 3
2-1 Current, voltage and power plotted versus time .................. 14
2-2 Relation between the polar and rectangular coordinates for the voltage ................................................................. 15
2-3 One line diagram of the IEEE 14 bus power system [119] ............ 17
2-4 The graph associated with the IEEE 14 bus power system in figure 2-3 ................................................................. 17
2-5 Nominal-π model of a transmission line ................................ 24
2-6 Transformer equivalent circuit using an ideal transformer ........... 25
2-7 The transformer model used in power systems ...................... 26
2-8 Elementary three-phase ac generator ................................... 29
3-1 The second order load flow method as defined by [71] .......... 59
3-2 A sparse matrix and its compressed storage ........................... 65
3-3 The graph associated with the IEEE-14 power system and the structure of the admittance matrix .............................................. 67
3-4 An outline of the sparse LU factorisation algorithm .............. 71
4-1 A sparse matrix and its associated graph .............................. 88
4-2 The matrix from figure 4-1 after reordering with node tearing ...... 89
4-3 Elimination trees ............................................................ 90
4-4 Structure of factors $L$ and $U$ ............................................. 91
4-5 Parallel decomposition of the linear system shown in figure 4-2 .... 92
4-6 The structure of the Jacobian for the ieee57 test system after node tearing .............................................................. 94
4-7 The hierarchical decomposition of a linear system .................. 95
5-1 Branch tearing in a graph and the resulting matrix structure ........ 105
5-2 The structure of the Jacobian for the ieee57 test system after branch-tearing ......................................................... 106
5-3 Eigenvalue distribution for various basic iterative methods ........ 107
6-1 A typical matrix structure after branch tearing ..................... 115
6-2 The structure of $L$ and $\hat{U}$ .............................................. 115
LIST OF FIGURES

6-3 The reduced system ............................................. 115
6-4 The Jacobian of the ieee57 test system after reordering. ........ 116
6-5 The example graph from figure 4-1 with I/O excluded and included. ... 117
6-6 Parallel decomposition of the linear system of figure 5-1 .......... 119

7-1 Constructing level sets for the example graph from figure 4-1. .... 128
7-2 Building the first cluster in the branch tearing algorithm. .......... 135
7-3 Building the first cluster in the node tearing algorithm. ............ 135

8-1 Elimination of a column in a dense block row in the reduced system. ... 144
8-2 CGS running on one and several processors. ........................... 146
8-3 Series on different numbers of processors ............................... 157
8-4 Events during the solution with the algorithm BRANCH ................ 159
8-5 Events during the solution with the algorithm NODE .................... 160
8-6 Events during the solution with the algorithm BRANCH ................ 162
8-7 Events during the solution with the algorithm NODE .................... 163

11-1 Reordering of the variables in the nonlinear system to simplify the study of the convergence. ........................................ 198
11-2 Run-time comparison for the system pow236. ............................ 211
11-3 Run-time comparison for the system pow529. ............................ 212
11-4 Run-time comparison for the system pow795. ............................ 212
11-5 Run-time comparison for the system pow1412. ............................ 213

D-1 Number of iterations needed for the Block-SOR method for different ω’s... 228

E-1 Structure of the matrix m795 after reordering with the minimum degree algorithm. ........................................ 231
E-2 Structure of the matrix m795 partitioned for three processors with the branch tearing algorithm. ........................ 232
E-3 Structure of the matrix m795 partitioned for three processors with the node tearing algorithm. ............................... 233
E-4 Structure of the matrix m795 partitioned for seven processors with the branch tearing algorithm. .......................... 234
E-5 Structure of the matrix m795 partitioned for seven processors with the node tearing algorithm. ............................... 235
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Notations used in the load flow problem.</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>Self-admittance due to connections to node i.</td>
<td>31</td>
</tr>
<tr>
<td>2-3</td>
<td>The different nodes in a power system and their associated variables.</td>
<td>35</td>
</tr>
<tr>
<td>2-4</td>
<td>Impedance and line charging data for the IEEE 14 bus test system.</td>
<td>36</td>
</tr>
<tr>
<td>2-5</td>
<td>Operating conditions for the IEEE 14 bus test system.</td>
<td>37</td>
</tr>
<tr>
<td>2-6</td>
<td>Regulated bus data for the IEEE 14 bus test system.</td>
<td>38</td>
</tr>
<tr>
<td>2-7</td>
<td>Transformer data for the IEEE 14 bus test system.</td>
<td>39</td>
</tr>
<tr>
<td>2-8</td>
<td>Static capacitor data for the IEEE 14 bus test system.</td>
<td>39</td>
</tr>
<tr>
<td>2-9</td>
<td>The power systems used as test systems.</td>
<td>40</td>
</tr>
<tr>
<td>3-1</td>
<td>The elements of the Jacobian for the polar formulation.</td>
<td>46</td>
</tr>
<tr>
<td>3-2</td>
<td>Elements to form the matrices $B'$ and $B''$.</td>
<td>51</td>
</tr>
<tr>
<td>3-3</td>
<td>Formulation of the matrices $B'$ and $B''$.</td>
<td>51</td>
</tr>
<tr>
<td>3-4</td>
<td>Timings for the sequential Newton-Raphson method.</td>
<td>73</td>
</tr>
<tr>
<td>6-1</td>
<td>Comparison of the number of critical nodes for node and branch tearing.</td>
<td>122</td>
</tr>
<tr>
<td>7-1</td>
<td>Statistics for level sets generated from real-life power systems.</td>
<td>129</td>
</tr>
<tr>
<td>7-2</td>
<td>Sizes of partitionings for various test problems.</td>
<td>136</td>
</tr>
<tr>
<td>8-1</td>
<td>List of the routines that were implemented.</td>
<td>140</td>
</tr>
<tr>
<td>8-2</td>
<td>Scalar products computed sequentially and in parallel.</td>
<td>140</td>
</tr>
<tr>
<td>8-3</td>
<td>Abbreviations for methods and preconditioners.</td>
<td>151</td>
</tr>
<tr>
<td>8-4</td>
<td>Results for linear methods for the test system m236.</td>
<td>152</td>
</tr>
<tr>
<td>8-5</td>
<td>Results for linear methods for the test system m529.</td>
<td>153</td>
</tr>
<tr>
<td>8-6</td>
<td>Results for linear methods for the test system m795.</td>
<td>154</td>
</tr>
<tr>
<td>8-7</td>
<td>Results for linear methods for the test system m1419.</td>
<td>155</td>
</tr>
<tr>
<td>8-8</td>
<td>Theoretical measures and real performance of the direct methods.</td>
<td>158</td>
</tr>
<tr>
<td>8-9</td>
<td>Correlation between the theoretical measures and the real performance.</td>
<td>158</td>
</tr>
<tr>
<td>9-1</td>
<td>Results on an iPSC/860 with the parallel Newton-Raphson method.</td>
<td>175</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-2</td>
<td>Results under PVM over two DECstations (5000/240 and 5000/200).</td>
<td>176</td>
</tr>
<tr>
<td>9-3</td>
<td>Results under PVM for 1 to 3 DEC Alpha's connected by GigaSwitch.</td>
<td>177</td>
</tr>
<tr>
<td>10-1</td>
<td>Block methods for the ieeel4 test system.</td>
<td>181</td>
</tr>
<tr>
<td>10-2</td>
<td>Block methods for the ieeel30 test system.</td>
<td>181</td>
</tr>
<tr>
<td>10-3</td>
<td>Block methods for the ieeel57 test system.</td>
<td>183</td>
</tr>
<tr>
<td>10-4</td>
<td>Block methods for the ieeel18 test system.</td>
<td>183</td>
</tr>
<tr>
<td>10-5</td>
<td>Spectral radius at the solution $x^<em>$ for $G'(x^</em>)$.</td>
<td>189</td>
</tr>
<tr>
<td>10-6</td>
<td>Block methods for the ieeel4 test system.</td>
<td>191</td>
</tr>
<tr>
<td>10-7</td>
<td>Block methods for the ieeel30 test system.</td>
<td>191</td>
</tr>
<tr>
<td>10-8</td>
<td>Block methods for the ieeel57 test system.</td>
<td>191</td>
</tr>
<tr>
<td>10-9</td>
<td>Block methods for the ieeel18 test system.</td>
<td>191</td>
</tr>
<tr>
<td>10-10</td>
<td>State of regulated busses during the descent method.</td>
<td>192</td>
</tr>
<tr>
<td>11-1</td>
<td>Results with the reduced algorithm.</td>
<td>202</td>
</tr>
<tr>
<td>11-2</td>
<td>The reduced method with initial iterations to find a good starting value.</td>
<td>204</td>
</tr>
<tr>
<td>11-3</td>
<td>Results for the multilevel algorithm.</td>
<td>210</td>
</tr>
<tr>
<td>11-4</td>
<td>A comparison between the Newton method and the multilevel method.</td>
<td>210</td>
</tr>
<tr>
<td>B-1</td>
<td>The Jacobian for the sample system.</td>
<td>221</td>
</tr>
<tr>
<td>B-2</td>
<td>The first elimination step with a $2 \times 2$ pivot.</td>
<td>221</td>
</tr>
<tr>
<td>C-1</td>
<td>Timings for various versions of the sparse matrix-vector multiplication gained on a single node of the Intel iPSC/860.</td>
<td>224</td>
</tr>
<tr>
<td>D-1</td>
<td>Number of iterations with the block SOR method.</td>
<td>227</td>
</tr>
<tr>
<td>D-2</td>
<td>Number of iterations for Bi-CGStab with block-SOR preconditioning</td>
<td>229</td>
</tr>
<tr>
<td>F-1</td>
<td>Tests carried out with the multilevel Newton method.</td>
<td>236</td>
</tr>
<tr>
<td>F-2</td>
<td>In the first iteration 2 inner iterations are done and subsequently only 1. If $|r|_\infty &lt; 0.1$ no inner iterations are done. Tests on 3 processors.</td>
<td>238</td>
</tr>
<tr>
<td>F-3</td>
<td>In the first iteration 2 inner iterations are done and subsequently only 1. If $|r|_\infty &lt; 0.1$ no inner iterations are done. A correction step is done before every full iteration. Tests on 3 processors.</td>
<td>239</td>
</tr>
<tr>
<td>F-4</td>
<td>In the first iteration 2 inner iterations are done and subsequently only 1. If $|r|_\infty &lt; 0.1$ no inner iterations are done. Tests on 7 processors.</td>
<td>240</td>
</tr>
<tr>
<td>F-5</td>
<td>In the first two full iterations 2 inner iterations are done and subsequently only 1. If $|r|_\infty &lt; 0.1$ no inner iterations are done. Tests on 7 processors.</td>
<td>241</td>
</tr>
</tbody>
</table>
F-6 In the first iteration 2 inner iterations are done and subsequently only 1. If
$\|r\|_{\infty} < 0.1$ no inner iterations are done. A correction step is done before
every full iteration. Tests on 7 processors. .......................... 242

F-7 In the first iteration 2 inner iterations are done and subsequently only 1. If
$\|r\|_{\infty} < 0.1$ no inner iterations are done. Tests on 12 processors. .... 243

F-8 In the first two full iterations 2 inner iterations are done and subsequently
only 1. If $\|r\|_{\infty} < 0.1$ no inner iterations are done. Tests on 12 processors. 244

F-9 In the first iteration 2 inner iterations are done and subsequently only 1. If
$\|r\|_{\infty} < 0.1$ no inner iterations are done. A correction step is done before
every full iteration. Tests on 12 processors. .............................. 245

F-10 Inner iterations are terminated if $\|r^{\text{inner}}\|_{\infty} < \|r\|_{\infty}/10$. In the first two
full iterations 2 inner iterations are allowed and subsequently only 1. If
$\|r\|_{\infty} < 0.1$ no inner iterations are done. Tests on 7 processors. .... 246

F-11 Inner iterations are terminated if $\|r^{\text{inner}}\|_{\infty} < \|r\|_{\infty}/10$. In the first two full
iterations 3 and 2 inner iterations are allowed respectively and subsequently
only 1. If $\|r\|_{\infty} < 0.1$ no inner iterations are done. Tests on 7 processors. ... 247
Chapter 1

Introduction

"Experience shows that the solution of a 4-mesh network by the usual method takes from three to ten days, depending on whether or not any errors are made in the 7,680 calculations and 9,600 determinations of signs. The [business] machines will do the same work in four to six hours, requiring only the attention of the regular machine operator." — P.D. Jennings and G.E. Quinan 1946 [73]

"...the science of calculation also is indispensable as far as the extraction of the square and cube roots: Algebra as far as the quadratic equation and the use of logarithms are often of value in ordinary cases: but all beyond these is but a luxury; a delicious luxury indeed; but not to be in indulged in by one who is to have a profession to follow for his subsistence." — Thomas Jefferson [103]

This thesis is concerned with the problem of developing efficient parallel algorithms for load flow analysis. Load flow analysis is used to determine the state of a power system under real or contemplated conditions. This includes simulations to guarantee the reliability of power systems, and answering questions such as 'what happens when a power line goes down at a time of peak usage', to 'what is the effect on the London grid if there is a concert in Wembley?'. The importance of this problem is illustrated by the early stage in which 'business machines' were considered for load flow simulations. Today computers are essential for the planning and running of power systems — and calculation has become much more than 'a delicious luxury'.

The size of the load flow simulations has increased dramatically over the years, and nowadays problems which are hundreds of times larger than the example Jennings and Quinan cited above can be solved in seconds. One of the reasons for the increase in size is that separate grids have been interconnected to form national grids, and that these are being connected in turn (e.g. Britain buys nuclear power from France). A second reason is that modern computers have made it possible to simulate in much more detail,
incorporating more aspects of the local distribution networks. Especially in the US power systems nowadays have up to 10000 nodes.

Load flow analysis is also important for the planning of new power systems. Power systems have largely grown organically over the years, and adding new lines or interconnections between grids can have far-reaching consequences. Load flow analysis enables engineers to study the effects of different design solutions under a large set of scenarios [124]. Load flow analysis is also important as part of many other applications, such as security analysis of power systems [128] or automated system planning [86]. Similar problems have to be solved for the optimisation of power generation [104]. Engineers are also starting to study the dynamics of power systems, rather than static cases. This increases the computational requirements dramatically, and load flow analysis is a key element in these calculations [86].

Not only is load flow analysis a fascinating problem in its own right, combining the modelling of electrical components with graph theory, linear algebra, nonlinear function solution and parallel computing methodologies, but it also provides an opportunity to apply mathematical theory in practice. Similar problems are also found in gas- and water distribution, as well as circuit simulation [76], and many of the algorithms developed for load flow, can be transferred to these problem areas.

1.1 Load Flow Analysis and Parallelisation

The reason to concentrate only on load flow is not only that it is a key component of many calculations required in power system analysis, but that it is also one of the hardest to parallelise. Load flow analysis requires the solution of a nonlinear, complex system of algebraic equations with constraints on some quantities. Due to the irregular structure of these systems (caused by the close relation to the graph structure of the power system), as well as their sparsity, it is very difficult to devise efficient parallel solution methods. The structure of the linear system that has to be solved during the Newton-Raphson solution of a standard test case, the ieee118 test system, is given in figure 1-1. It demonstrates the irregular structure of such systems and their extreme sparsity; as the structure depends on the underlying network, the irregularity is independent of the ordering. In a report of an IEEE task force on power systems and parallel computing, the authors says the following about parallel processing and load flow [131]:

"The application of parallel processing to power systems analysis is motivated by the desire for faster computation and not because of the structure of the problem. ... there are no obvious parallelisms inherent in the mathematical structure of power systems problems. Thus, for a particular problem a parallel (or near-parallel) formulation has to be found that is amenable to formulation as
1.1. LOAD FLOW ANALYSIS AND PARALLELISATION

Figure 1-1: The structure of the linear system that has to be solved during the load-flow analysis of the ieee118 test system after reordering with the minimum degree algorithm.

a parallel algorithm. This solution has then to be implemented on a particular parallel machine keeping in mind that computational efficiency is dependent on the suitability of the parallel architecture to the parallel algorithm.

The interconnected generation and transmission system is inherently large and any problem formulation tends to have thousands of equations. The most common analysis, the power flow, requires the solution of a large set of nonlinear algebraic equations, approximately two for each node. ... Parallel algorithms for handling dense matrices are not competitive with sequential sparse matrix methods, and since the pattern of sparsity is irregular, parallel sparse matrix methods have been difficult to find. The power flow describes the steady state condition of the power network and thus, the formulation (or some variation) is a subset of several other important problems like the optimal power flow or transient stability. An effective parallelisation of the power flow problem would also help speed up these other solutions."
CHAPTER 1. INTRODUCTION

As it is very hard to obtain any speed-up for this problem through parallel computation, the question must be asked whether it is worth expending the effort. The same IEEE task force cited earlier state that '... the industrial need [for faster simulation] is more in the area of on-line applications in the control centre where faster computation gives the power system operator more automatic analysis to help in decision making. Obviously, any breakthroughs will be beneficial for off-line use because those analyses that are now too slow for interactive use can be made so, and more of those that are already quite fast can be run in the same time period. That is, faster programs increase engineering efficiency.'

1.2 Approaches to parallel solution methods

A summary of parallel algorithms used for load flow analysis is given in section 3.9. The most widely used parallel solution methods reorder the matrices to obtain a double-bordered block system, and then use a multi-frontal approach to solve these in parallel (see section 4). Unfortunately this procedure results in a severe bottle-neck when the border equations are solved. The two best algorithms developed here both have advantages over this approach, especially with respect to the ability to scale to more processors.

The parallel algorithms developed in this thesis are based on the conviction that the physical structure of the problem must be taken into account to obtain efficient methods. It is also desirable to have algorithms that are as generally applicable as possible. At the same time characteristics of the problem must be taken into account to make the algorithms both realistic and competitive.

To achieve this all algorithms take as starting point the partitioning of the graph representing a power system. This partitioning determines how the load flow equations are distributed over the processors and thus determines the amount of parallelism that can be exploited. Both branch tearing and node tearing, that is the removal of branches or nodes respectively from the graph in order to partition it, are considered.

Two classes of methods are investigated for the solution of the nonlinear systems of equations:

1. Parallelising the solution of the systems of linear equations arising during a standard Newton-Raphson iteration.

2. Partitioning the system of nonlinear equations and solving several systems of nonlinear equations in parallel.

For the first class several algorithms for the solution of linear systems of equations are considered. Among these are preconditioned CG-type solvers, as well as direct methods. Since the linear systems are very badly conditioned, one of the latter emerges as the fastest
algorithm. Combining it with the Newton-Raphson algorithm results in a parallel solver that provides real speed-ups with respect to optimised sequential implementations.

Devising parallel nonlinear algorithms, which is the second class of methods investigated, is a very difficult problem. Relatively little theory exists, and even less is applicable due to the difficult nature of this topic. Nevertheless it proved possible to develop algorithms which require fewer global information exchanges, but sacrifice the convergence properties of the Newton method.

Throughout the research the algorithms were tested on realistic problems. Constraints are imposed on some variables in practical load flow problems, and these are taken into account in all implementations. This has far-reaching consequences for the efficiency of the parallel algorithms, as additional synchronisation and data exchange is needed.

The goal was to develop algorithms that are competitive in real-life, and all parallel algorithms were compared to a very efficient single processor implementation. When timing the algorithms the complete solution time, including the distribution of data and the collection of results, is considered.

All algorithms (some 20000 lines of code) were implemented on an Intel iPSC/860 in Fortran 77 using Intel’s native message passing constructs and the parallel Newton method was also implemented on a workstation cluster.

1.3 Overview of the thesis

The thesis separates naturally into three parts: part I provides an overview of the load flow problem and a survey of solution methods; part II considers the parallelisation of the solution of linear systems arising during load flow simulation; and part III considers the solution of the nonlinear load flow equations. In the latter part methods in which the algorithms of part II are used, as well as algorithms in which the solution of the nonlinear systems themselves is parallelised, are considered. A short overview of every chapter follows.

In chapter 2 a complete overview of the load flow problem and details of the modelling of system components is given. No complete description was found in the literature, and a good understanding of the load flow problem is essential for the development of realistic solution methods. The test problems used throughout the thesis to compare algorithms are also discussed.

Chapter 3 surveys different methods that are currently used for the solution of load flow problems. These include an overview of the convergence of specialised methods, and the motivation for the choice of the Newton-Raphson method for subsequent work. It is explained why the industry standard, the fast decoupled load flow method, does not lend itself to parallelisation, even though it initially looks very promising. Finally a very
efficient sequential implementation of the Newton-Raphson method, as well as the sparse matrix techniques that are used throughout, are described.

This concludes the first part of the thesis. The second part starts with chapter 4 which gives an introduction to node-tearing and parallel direct methods. Due to the problems with scaling to more processors, iterative methods are considered in chapter 5. Special emphasis is placed on various preconditioners, as these are essential to obtain convergence for the very badly conditioned systems considered.

Ideas developed for block-preconditioning lead naturally to the concept of system reduction in chapter 6, which is essentially the calculation of a distributed Schur complement. This turns out to be a key idea, and an efficient direct parallel algorithm is derived from this methodology.

None of these algorithms are any use in practice, unless automated partitioning is available. Unfortunately it is not an easy problem to develop algorithms which are both fast, and give a well-balanced partitioning. After considering several approaches in chapter 7, a methodology developed by Sangiovanni-Vincentelli et al. [118] is generalised and used for both node and branch tearing.

The concluding chapter of part II, chapter 8, discusses the implementation of the algorithms for linear systems and compares results on a set of real-life problems.

Part III commences with a description of the implementation of the full Newton-Raphson iteration, employing the best linear solution method developed in part II. General implementation considerations and results are presented in chapter 9.

Subsequently other traditional approaches to the solution of nonlinear equations, such as nonlinear block-SOR, are considered in chapter 10. It turns out that these are not capable of solving the badly conditioned load flow problems in a reasonable amount of time. A related algorithm that always provides a descent direction, and is thus globally convergent, is also developed. This algorithm converges very slowly, and its disappointing behaviour is explained.

A final approach to solving the load flow problem is presented in chapter 11. Here an attempt is made to transfer the ideas developed in part II for the reduction of linear systems to nonlinear systems. Several approaches are investigated and a new method is developed, which proves to be competitive with the Newton-Raphson algorithm combined with the fastest linear solution method.

1.4 Contribution of this thesis

The main contributions of this thesis are

- the introduction of the idea of reduced systems,
1.4. CONTRIBUTION OF THIS THESIS

- the application of this idea to provide an effective parallel solution method for linear systems (chapter 6), and,

- an investigation into novel parallel algorithms for the solution of nonlinear systems (chapter 11).

Further contributions are the following.

- A unified view of the models for power system components that are required for load flow analysis (chapter 2).

- A survey of load flow analysis methods, investigating their convergence characteristics and explaining the limits to some obvious parallelisation approaches (chapter 3).

- A detailed comparison of direct and iterative methods for the solution of linear systems of equations arising in load flow in part II.

- A generic, fast partitioning approach, that is useful for both node and branch tearing (chapter 7).

- An investigation into generalised linear methods for the solution of nonlinear systems of equations (chapter 10).

- An implementation and realistic comparison of all these algorithms.
Part I

Load Flow Analysis
Chapter 2

Load Flow Analysis

In this chapter the formulation of load flow equations is described. ‘A load study is the determination of the voltage, current, power and power factor or reactive power at various points in an electric network under existing or contemplated conditions of normal operation.’ [124, p.5]

The motivation for load flow studies is that, apart from the day-to-day operation of a power system, ‘load studies are essential in planning the future development of the system because satisfactory operation of the system depends on knowing the effects of interconnections with other systems, of new loads, new generating stations, and new transmission lines.’ [ibid]

The main contribution of this chapter is to give a simple, unified and complete overview of the modelling of a power system for the purpose of load flow analysis. No complete description was found in the literature.

A prerequisite for discussing the modelling of a power system is to understand how alternating currents are modelled. A summary is given in section 2.1.

For a load flow analysis a description of a power network is required. The nodal model is described in section 2.2. Some remarks are made on per unit (p.u.) calculations in section 2.3. Doing calculations in p.u. not only simplifies load flow programs, but scales the problem as well.

Section 2.4 is devoted to modelling the power system components relevant to a load flow study. These are transmission lines, transformers, capacitor banks and synchronous compensators. In section 2.5 these models are combined to result in a mathematical model of a power system. The data required for a load flow analysis and the necessary data conversions are discussed in section 2.6. Finally the test systems used throughout this thesis are described in section 2.7. Most of the work in this chapter appeared as a technical report [77].
2.1 Alternating currents

2.1.1 Current, voltage and power

In alternating current (ac) analysis, current and voltage are sinusoidal waves with a frequency \( f \). With \( \omega = 2\pi f \) the current \( i \) and the voltage \( v \) are expressed as functions of the time \( t \) as

\[
\begin{align*}
    i & = I_p \sin \omega t \\
    v & = V_p \sin \omega t,
\end{align*}
\]

with \( I_p \) and \( V_p \) the amplitudes of the sine waves.

In the case of direct current the power is given by \( P = VI \). For alternating current the average power \( p \) over the time period \( T = 1/f \) is given by [10]

\[
p = vi = \frac{1}{T} \int_0^T vi \, dt = \frac{V_p I_p}{T} \int_0^T \sin^2 \omega t \, dt = \frac{V_p I_p}{2}.
\]

So the effective current or voltage is given by the root-mean-square (rms) values

\[
I_e = \frac{I_p}{\sqrt{2}} \quad \text{and} \quad V_e = \frac{V_p}{\sqrt{2}}. \quad (2-1)
\]

These are the values shown by voltimeters or ammeters.

2.1.2 Phase angles and reactance

Important for power system analysis is that capacitance and inductance give rise to a phase angle between current and the voltage. If the two sine waves are shifted with respect to each other, they are described by

\[
\begin{align*}
    i & = I_p \sin \omega t \\
    v & = V_p \sin(\omega t + \theta),
\end{align*}
\]

where the phase angle is \( \theta \) by definition.

The occurrence of phase angles is intuitively clear when considering a capacitor consisting of two parallel metal plates separated by a thin air gap. If the potential difference between the two plates is constant, one will be positively charged and the other negatively. As soon as the potential difference over the plates changes, a current starts flowing to adapt the charge on the plates to the new situation. In an ac circuit the voltage over the capacitor changes with the time, which gives rise to a time-varying charge and thus causes a current to flow.

Expressed in formulas (\( C \) is the capacitance of the capacitor, and \( q \) the charge):

\[
\begin{align*}
    i & = \frac{dq}{dt} = \frac{dCv}{dt} = C \frac{d}{dt}(V_p \sin \omega t) \\
    & = \omega C V_p \cos \omega t = \omega C V_p \sin(\omega t + \frac{\pi}{2}). \quad (2-3)
\end{align*}
\]
2.1. ALTERNATING CURRENTS

Thus the current, which is a result of the reactance of the capacitor to the alternating current, is sinusoidal and leads the voltage by $\frac{\pi}{2}$ rad. In terms of rms values (2-3) can be expressed as

$$V_e = \frac{1}{\omega C} \cdot I_e.$$  

(2-4)

This is similar to Ohm's Law ($V = IR$) and the proportionality factor $\frac{1}{\omega C}$ between the voltage and the current is called the capacitive reactance.

Inductors act analogously, except that the current lags the voltage. The relation between the effective voltage and current is given by ($L$ is the inductance)

$$V_e = \omega L \cdot I_e.$$  

(2-5)

The quantity $\omega L$ is called the inductive reactance. In combination with other circuit elements, the phase shift between voltage and current can take on other values than $\pi/2$ (see for example RL-filters and RC-filters [10]).

2.1.3 Real and reactive power

As the power $P = I \cdot V$ it is affected by the phase angles between current and voltage. If the phase angle is $\theta$, as in equation (2-2), the average power is given by [10]

$$p = \frac{1}{T} \int_0^T v \cdot i dt = \frac{V_p I_p}{T} \int_0^T \sin \omega t \cdot \sin (\omega t + \theta) dt$$

$$= \frac{V_p I_p}{2} \cos \theta$$  

(2-6)

From this equation it follows that no useful electrical power is delivered for a phase angle $\theta = \pi/2$, although the current and the voltage may be very large. This becomes clearer when plotting the power with the voltage and the current in one diagram as seen in figure 2-1 [10, p.39]. Note that the power in a part of the circuit may be negative at times.

As inductors and capacitors induce a phase angle, most electrical appliances, notably those containing electro-motors, will induce a phase difference. Equation (2-6) implies that the peak power that has to be delivered to such an appliance is higher than the effective power and that reactive power (the ‘imaginary’ power) has to be absorbed by the system. Put differently this means that large currents may be flowing though a power system while little effective power is transmitted.

As reactive power can be very large it cannot be ignored in power system analysis. In power systems reactive power is supplied by generators to compensate for the reactive power required at load busses. Load bus bars, which are not connected to a generator, are often equipped with a capacitor bank or synchronous compensator, which can supply all
or part of the reactive power required by the load on that bus bar. Capacitor banks thus reduce the line current necessary to supply the power required by the load and, due to this improvement in the power factor, the voltage drop in the line is reduced [124, p.212].

2.1.4 Complex representation of voltages and currents

In order to measure differences in phase a reference is needed. Consider a voltage and a current described as a function of time, with the current leading the voltage by $\theta$ as in (2-2). In this case the current is used as reference. Usually the voltage in one node of the system is defined as having a phase angle of 0 degrees.

$I_p$ and $V_p$ are the maximum values of the current and the voltage, but the values of interest are the rms or effective values, denoted by $|I|$ ($I_e$) and $|V|$ ($V_e$) respectively [124, p.11]. Current and voltage are often written as phasors, which give their rms values and phase angle:

\[
I = |I|e^{j\theta} \\
V = |V|e^{j\theta}.
\]  

(2-7)

In figure 2-2 the voltage is shown as a phasor. It turns out that flow analysis is greatly facilitated if sinusoidal currents and voltages are represented by complex numbers such that

\[
i = I_pe^{j\omega t} = I_p(\cos\omega t + j\sin\omega t) \\
v = V_pe^{j(\omega t + \theta)} = V_p(\cos(\omega t + \theta) + j\sin(\omega t + \theta)).
\]  

(2-8)

Note that $j$ is used instead of $i$ to represent $\sqrt{-1}$ as $i$ is used as symbol for current.

Sometimes it is convenient to express these values in rectangular coordinates. Then the voltage and the current are expressed as

\[
V = e + jf \\
I = a + jb.
\]
In figure 2-2 the voltage $|V|\angle \theta = e + jf$ is shown.

### 2.1.5 Complex power

If the phasor expressions for voltage and current are known the calculation of real and reactive power in complex form is convenient [124, p.18]. If voltage and current are expressed by $V = |V|\angle \theta$ and $I = |I|\angle \phi$ the complex power $S$ is defined as

$$S = VI^* = \frac{V_0 I_0 e^{j\theta} e^{-j\phi}}{2} = |V||I|e^{j(\theta - \phi)}.$$  \hspace{1cm} (2-9)

The real part of the complex power $S$ is the same as the power in equation (2-6). Complex power is denoted by

$$\text{power} = \text{real power} + j \cdot \text{reactive power}
\quad
S = P + jQ.$$  

Complex power written in rectangular coordinates is

$$VI^* = (e + jf)(a - jb) = (ea + fb) + j(fa - eb).$$

**Notation:**

Usually $V_0$ or $|V|$ is simply denoted by $V$, as is $I_0$ or $|I|$ by $I$. Although the notation for complex voltage and current is identical, no confusion should arise as the context makes clear which value is meant.

### 2.1.6 Impedance and admittance

Consider a current with a resistor and a conductor connected in series. Then the voltage over these two elements is given by

$$v = R\dot{i} + L\frac{di}{dt},$$  \hspace{1cm} (2-10)
and with (2-8) this gives

\[ V_p e^{j\omega t} = RL_p e^{j\omega t} + j\omega LI_p e^{j\omega t} = (R + j\omega L)I_p e^{j\omega t} \]  \hspace{1cm} (2-11)

The quantity \((R + j\omega L)\) is called the complex impedance \(Z\) of the circuit. The imaginary part of the impedance specifies the reactance of the circuit while the real part stands for the resistance. In short

\[ \text{impedance} = \text{resistance} + j\cdot\text{reactance} \]
\[ Z = R + jX \]

The term impedance comes from the fact that both reactance and resistance impede the current in a circuit. The unit of impedance is Ohm (\(\Omega\)).

The inverse of impedance, \(Y = Z^{-1}\), is the complex admittance.

\[ \text{admittance} = \text{conductance} + j\cdot\text{susceptance} \]
\[ Y = G + jB \]

Polar as well as rectangular representations of the admittance and impedance are used in modelling connections between nodes. Customarily the polar version of the admittance is denoted by \(|Y|e^{j\xi}\).

This concludes the introductory section on alternating currents. The concepts which are of particular importance to power system analysis are active and reactive power, as well as impedance and admittance.

### 2.2 Electric system models

In order to do a load flow analysis for a power system a mathematical model of the network is required. Although there are different types of network models the nodal model is almost universally preferred [42, p.39]. Only the nodal model will be considered here.

In circuit simulation a node is defined as a junction where two or more elements are connected to each other. In power systems nodes usually correspond to the busses. The interconnections between nodes are called branches. The collection of nodes and branches define a connected graph which is used in the efficient solution of a load flow problem.

See figure 2-3 for a one-line diagram of a fourteen bus bar power system and its associated graph. Real-life power systems are 3-phase systems, but it is sufficient to model one of the phases. Furthermore, multiple lines between busses are modelled as single lines. For a mathematical description of such a graph the following notation is needed.
Figure 2-3: One line diagram of the IEEE 14 bus power system [119].

Figure 2-4: The graph associated with the IEEE 14 bus power system in figure 2-3.
CHAPTER 2. LOAD FLOW ANALYSIS

Notation:
A graph is represented by a set $\mathcal{V}$ of nodes (vertices) and a set $\mathcal{E}$ of edges (branches). An edge is an unordered pair $(v, w)$, $v \neq w$, and $v, w \in \mathcal{V}$. An edge $(v, w) \in \mathcal{E}$ if, and only if, there is a connection between nodes $v$ and $w$.

Consider the graph depicted by the diagram in figure 2-4. For this graph $\mathcal{V} = \{1, \ldots, 14\}$ and $\mathcal{E} = \{(1, 2), (1, 5), (2, 3), (2, 4), \ldots, (12, 13), (13, 14)\}$.

Load flow models are the result of applying Kirchhoff's circuit laws in conjunction with descriptions of the components in the power system. Kirchhoff's laws are [10, 76]:

**Kirchhoff's Current Law** The algebraic sum of the currents at any node is zero. Currents flowing into the node are considered positive and currents flowing out of the node negative.

**Kirchhoff's Voltage Law** The algebraic sum of the potential differences around any complete loop in the network is zero.

Kirchhoff's first law simply states that current flowing into a node must flow out of that node again. The second law states that the potential difference between two nodes is the same as the sum of potential differences between all intermediate nodes. Together these laws state that the power that flows into a node has to flow out of the node again. For $n$ nodes this can be summarised in the following $n$ complex power equations [89]

$$ S_{Gi} - S_{Di} + S_{Hi} = 0 \quad i = 1, \ldots, n \quad (2-12) $$

Here

- $S_{Gi} = P_{Gi} + jQ_{Gi}$ - Power generated at bus bar $i$
- $S_{Di} = P_{Di} + jQ_{Di}$ - Load demand at bus bar $i$
- $S_{Hi} = P_{Hi} + jQ_{Hi}$ - Net power injected into bus bar $i$ from the network

Equation (2-12) is equivalent to the $2n$ nodal equations for the real and reactive power

$$ P_{Gi} - P_{Di} + P_{Hi} = 0 \quad i = 1, \ldots, n \quad (2-13) $$
$$ Q_{Gi} - Q_{Di} + Q_{Hi} = 0 \quad i = 1, \ldots, n $$

This is called the nodal model as the power equations (2-12) are defined as functions of the power flowing in or out of the nodes in the network.

In the following discussion, based on [124, p. 193], three types of busses are distinguished.

**Load bus (PQ bus)** A bus to which only a load is connected. This load is considered to be fixed and may be zero.
2.3. **PER UNIT CALCULATIONS**

**Generator bus (PV bus)** A bus to which generators are connected, in other words $S_{Gi} \neq 0$. A load may be connected to a generating bus.

**Swing bus** A single generator bus in a system for which the power generation is not fixed.

Real and reactive power demand ($S_{Di}$) is always fixed and has to be found through measurements or forecasts.

Real power generation ($P_{Gi}$) is specified at all the busses except the swing bus. Power generation cannot be defined at the swing bus as the total power losses in the system are not known in advance.

Reactive power generation is kept variable within specified limits at generator busses. As mentioned when discussing reactive power in section 2.1, capacitor banks or synchronous compensators may be employed as sources of reactive power at load busses. These are further discussed in section 2.4.4. For technical reasons busses equipped with synchronous compensators are classified as generator busses with no real power generation.

The diagram of the power system depicted in figure 2-3 shows pure load busses (for example nodes 12, 13) and generator busses (nodes 1 and 2). Either of the generator busses can be designated the swing bus. Because of the synchronous compensators at nodes 3, 6, and 8, these nodes are designated generator busses.

As will be seen in section 2.5, there are four variable quantities at every bus: the generated real power ($P_G$), the generated reactive power ($Q_G$), the voltage magnitude ($|V|$), and the voltage angle ($\theta$). The relations between these quantities in connected busses are governed by the branch equations discussed in section 2.4.1.

### 2.3 Per unit calculations

In power systems the unit for voltage is kV, the unit for power MW and so forth. To avoid adding scaling factors it is often more convenient to work with values expressed as a fraction of a base value. For instance if 120 kV is chosen as a base value, 108, 120 and 126 kV become 0.9, 1.0 and 1.05 p.u. (per unit) respectively.

Usually base values are given for the voltage and for the power. One might choose a base value of 120 kV for the voltage and 100 MW for the power, which specifies a base value of $100 \cdot 10^6/120 \cdot 10^3 \, \text{A} = 5/6 \cdot 10^3 \, \text{A}$ for the current, as $p = v \cdot i$. A base value for impedance is also needed and, as there is a fixed relation between voltage and power, a base value for the impedance cannot be chosen independently. The base value for the impedance must be

$$
\text{Base impedance} = \frac{(\text{Base voltage, kV})^2}{\text{Base power, MW}}.
$$

as can be seen from the formula $P = V I = V^2/Z$. For the values quoted above the base impedance would thus be 144 $\Omega$. 
Although the base value for the power is the same throughout a power system the base values for the voltage and the current may change as a result of transformers in the system. So the voltages in one part of the system could be around 138 kV, while voltages in another part are around 34.5 kV. The transformation ratio from the primary to the secondary side (see section 2.4.3) would then be
\[
\frac{V_p}{V_s} = \frac{138}{34.5} = 4 = \frac{I_s}{I_p}.
\]
If the base values in both parts of the circuit have been chosen consistent with this transformation ratio, in other words \((\text{Base voltage}, kV)_p = 4(\text{Base voltage}, kV)_s\), one would get
\[
(\text{Base power}, MW)_p = (\text{Base voltage}, kV)_p \cdot (\text{Base current}, kA)_p
\]
\[
= 4(\text{Base voltage}, kV)_s \cdot \frac{1}{4}(\text{Base current}, kA)_s
\]
\[
= (\text{Base power}, MW)_s.
\]
So although the base currents and voltages change, the base power remains constant throughout the system.

Using p.u. values has the big advantage that all the variables in the system can be assumed to be well scaled as they can be expected to be within an order of magnitude of their base value. Thus voltage values will always be around 1 p.u. even though the real voltages in a system may vary by several orders of magnitude.

2.4 Power system components

In this section mathematical models for the power system components relevant to load flow analysis are discussed. In section 2.4.1 a simple model for a connection between two nodes with nonzero impedance is described, before introducing the nominal-\(\pi\) model for transmission lines in section 2.4.2. Finally transformers are discussed in section 2.4.3 and capacitor banks and synchronous compensators in section 2.4.4. The modelling of a complete power system is discussed in section 2.5.

2.4.1 Load flow in a branch

During load flow analysis the effect of the impedance in branches of the network must be taken into account. In this section the mathematical model of a connection between two nodes \(i\) and \(k\), with \(Z\) the complex impedance of the connection between them, is examined.
2.4. POWER SYSTEM COMPONENTS

In analogy to Ohm’s law $V = IZ$. The (complex) voltage in node $i$ is defined as $V_i = |V_i|e^{j\theta_i}$, with $|V_i|$ the potential difference between node $i$ and a reference node, and $\theta_i$ the phase angle relative to that same node. Rewriting the equation $V = IZ$ with the admittance $Y = Z^{-1}$ gives

$$I = VY = (V_k - V_i)Y.$$ 

The power injected into node $i$ from node $k$ is

$$S_i = V_i^*I^* = V_i^*(V_k - V_i)^*Y^* = V_i^*V_k^*Y^* - V_i^*V_i^*Y^*.$$ \hspace{1cm} (2-14)

The power flow in the line is modelled by two sets of equations, one for the real and one for the reactive power. This necessitates expressions for

$$P_i = \text{re}\{V_i^*V_k^*Y^* - |V_i|^2Y^*\}$$ \hspace{1cm} (2-15a)

and

$$Q_i = \text{im}\{V_i^*V_k^*Y^* - |V_i|^2Y^*\}.$$ \hspace{1cm} (2-15b)

Three different formulations for these equations are discerned, depending on whether the complex numbers in these formulas are expressed in polar or rectangular coordinates.

All Polar Formulation

The right hand sides of the equations (2-15b) involves a complex expression in the phasor voltages and admittances. All these quantities can be expressed in polar form. The phasor notation for voltage and admittance is

$$V_i = |V_i|e^{j\theta_i} = |V_i|e^{j\theta_i}$$
$$Y = |Y|e^{j\xi} = |Y|e^{j\xi}$$

Notation:

If $\theta_i$ and $\theta_k$ are the voltage angles in node $i$ and $k$ respectively, then the difference between these angles is denoted by

$$\theta_{ik} := \theta_i - \theta_k.$$ 

From (2-14) it follows that (as mentioned on page 15, $|V_i|$ is commonly denoted by $V_i$ in real valued equations)

$$S_i = V_iV_k|Y|e^{j(\theta_i - \theta_k - \xi)} - V_i^2|Y|e^{-j\xi}$$
$$= V_iV_k|Y|\{\cos(\theta_{ik} - \xi) + j \sin(\theta_{ik} - \xi)\} - V_i^2|Y|\{\cos \xi - j \sin \xi\}$$ \hspace{1cm} (2-16)
so that

\begin{align*}
P_i &= V_i V_k |Y| \cos(\theta_{ik} - \xi) - V_i^2 |Y| \cos \xi \\
Q_i &= V_i V_k |Y| \sin(\theta_{ik} - \xi) + V_i^2 |Y| \sin \xi.
\end{align*}

(2-17)

**Polar formulation**

In rectangular coordinates the admittance \( Y \) is conventionally written as \( Y = G + jB \). Using the polar representation for the voltages the following expression is obtained for (2-14)

\begin{align*}
S_i &= V_i V_k e^{j\theta_{ik}} (G - jB) - V_i^2 (G - jB) \\
&= V_i V_k (\cos \theta_{ik} + j \sin \theta_{ik})(G - jB) - V_i^2 (G - jB) \\
&= V_i V_k \{(G \cos \theta_{ik} + B \sin \theta_{ik}) + j(G \sin \theta_{ik} - B \cos \theta_{ik})\} + V_i^2 (G - jB).
\end{align*}

Equations (2-15b) are then written as \([129, 42]\)

\begin{align*}
P_i &= V_i V_k (G \cos \theta_{ik} + B \sin \theta_{ik}) - V_i^2 G \\
Q_i &= V_i V_k (G \sin \theta_{ik} - B \cos \theta_{ik}) - V_i^2 B
\end{align*}

(2-19)

The computation of the trigonometric functions in these formulas seems to be a drawback of the polar formulation. However the difference in voltage angle over a connection is rarely more than 20° so that simple approximations of the sine and cosine are often adequate. In most cases no large error is made by setting \( \cos \theta_{ik} \approx 1 \) and \( \sin \theta_{ik} \approx \theta_{ik} \).

**Rectangular formulation**

For the rectangular formulation voltages are expressed in rectangular coordinates too, so that

\[ V_i = e_i + jf_i. \]

Then (2-14) becomes

\begin{align*}
S_i &= (e_i + jf_i)(e_k - jf_k)(G - jB) - (e_i + jf_i)(e_i - jf_i)(G - jB) \\
&= (e_i e_k G - e_i f_k B + f_i e_k G + f_i e_k B) + \\
&\quad j(f_i e_k G - f_i f_k B - e_i f_k G - e_i e_k B) - (e_i^2 + f_i^2)(G - jB),
\end{align*}

(2-20)

and equations (2-15b) are formulated as

\begin{align*}
P_i &= e_i (e_k G - f_k B) + f_i (f_k G + e_k B) - V_i^2 G \\
Q_i &= f_i (e_k G - f_k B) - e_i (f_k G + e_k B) + V_i^2 B.
\end{align*}

(2-21)

The rectangular description has the advantage that no sines or cosines have to be computed.
2.4. POWER SYSTEM COMPONENTS

In this way all the connections within a power system can be described. The remaining problem is to describe the admittance $Y$ of a branch. As the admittance is a measure of the power loss in a connection, models for all the power system components are required.

2.4.2 Modelling transmission lines

Four parameters have to be taken into account when modelling a transmission line. These are resistance, inductance, capacitance and conductance.

Resistance is the most important cause of power loss and depends on the temperature, the diameter and the length of the transmission line, as well as the material the line is made of (usually aluminium or an aluminium alloy). The so-called skin effect [124, p.42], caused by changes in the magnetic flux in the transmission line, also influences the resistance. In normal load flow analysis these effects do not have to be taken into account separately, as manufacturers provide tables of the electrical characteristics of their conductors.

To explain the concept of inductance in the case of transmission lines consider two lines running in parallel. Variation of the current in the lines causes a change in the number of lines of magnetic flux linking them, which induces a voltage in the lines. Inductance is the property of the circuit that relates the voltage induced by changing flux to the rate of changing current [124, p.52].

The combined effects of resistance and inductance, uniformly distributed along the line, are responsible for the series impedance of a transmission line.

In the case of two-or-more-wire lines capacitance exists as result of the potential difference between the different wires. Capacitance between parallel conductors is a constant depending on the size and spacing of the conductors. For power lines less than 80 km long the effect of capacitance is slight, but for longer lines the capacitance becomes increasingly important [124, p.63].

Conductance between conductors or between conductors and the ground accounts for the leakage of current at the insulators of overhead lines and through the insulation of cables. The effect of conductance is small and very variable, so that conductance is neglected.

The conductance and capacitance, uniformly distributed along the line, make up the shunt admittance. Although the resistance, inductance and capacitance are distributed along the line, models of transmission lines make use of lumped parameters.

In load flow problems transmission lines are characterised by three parameters, these being the resistance ($R$), the reactance ($X$) and the capacitance ($2BC$).

Consider a transmission line between two nodes $i$ and $k$. Let $Z = R + jX$ be the series impedance of the line $i - k$, and $I$ the current flowing through this line. Then

$$V_k = V_i + IZ.$$
There are several ways in which the shunt admittance can be included in the model. In the case of the nominal-π model of transmission lines shown in figure 2-5, half the shunt admittance $2B_C$ is lumped at either end of the transmission line [124, p. 93].

The current in the capacitance is $jB_C V_i$ and the current $I$ flowing through the line is $I_i - jB_C V_i$. Thus the expression

$$I_i = \frac{V_k - V_i}{Z} - jB_C V_i$$

(2-22)

is obtained. This formula is not quite correct for a long line, but an equivalent circuit can be constructed, from which adjusted factors $B_C$ and $Z$ can be determined [124, p.100].

Using the admittance of the line $Y = 1/Z$ gives the equation

$$S_i = V_i I_i^* = V_i (V_k - V_i)^* Y^* - V_i V_i^* (-jB_C)$$

$$= V_i V_k^* Y - V_i V_i^* (Y^* - jB_C)$$

(2-23)

for the power flowing in at node $i$.

**Polar formulation**

As in section 2.4.1 the formulation of (2-23) with $Y = G + jB$ results in

$$S_i = V_i V_k (\cos \theta_{ik} + j \sin \theta_{ik}) (G - jB) - V_i^2 \{G - j(B + B_C)\}.$$  

(2-24)

**Rectangular formulation**

In rectangular coordinates (2-23) is written as

$$S_i = (e_i + jf_i)(e_k - jf_k)(G - jB) - V_i^2 \{G - j(B + B_C)\}.$$  

(2-25)

This concludes the section on transmission lines. In the next section transformers are discussed.
2.4. POWER SYSTEM COMPONENTS

2.4.3 Modelling transformers

Transformers provide the links between generators, transmission lines and the local distribution systems. Higher transmission voltages have the advantage that the transmission capacity of a line is increased. Already long distance lines operate at voltages as high as 765 kV [124].

The description given here is largely taken from Stevenson [124, chap. 6.5-6.7 & 8.10]. For an ideal transformer with \( N_1 \) primary windings and \( N_2 \) secondary windings the following relations hold

\[
\frac{V_1}{V_2} = \frac{N_1}{N_2} \quad \text{and} \quad \frac{I_1}{I_2} = \frac{N_2}{N_1}.
\]

(2-26)

Such a transformer is called ideal, as the power flowing into the transformer is equal to the power flowing out of the transformer

\[
S_1 = V_1 \cdot I_1^* = \frac{N_1}{N_2} V_2 \cdot \frac{N_2}{N_1} I_2^* = V_2 \cdot I_2^* = S_2.
\]

A model for a real transformer is given in figure 2-6. Losses that occur in a transformer due to the impedance of the coils, the cyclic changing of the direction of flux in the iron core etc. are taken into consideration. In a real life power system it is difficult to measure all these parameters, so that a much simpler model is used in which the circuit-magnetising current \( I_E \) is neglected and the impedance is lumped at the low voltage side of the transformer. A diagram of this transformer model is given in figure 2-7.

In this diagram the transformation ratio (tap setting) is given as a complex number as transformers may introduce a phase shift. In fact so called phase shifters do not change the voltage magnitude at all, but only introduce a phase shift. If the voltage ratio is \( t \) and the phase shift \( \phi \), the complex ratio becomes

\[
te^{j\phi} = a + jb
\]

Figure 2-6: Transformer equivalent circuit using an ideal transformer. [124, p.143]
and the following relations hold

\[ \frac{V_p}{V_s} = a + jb \quad \text{and} \quad \frac{I_p^*}{I_s^*} = \frac{1}{a + jb}. \quad (2-27) \]

The power flow in the line \( i - k \) is determined as follows. Use the admittance \( Y' = 1/Z \), so that the current flowing in the line from node \( s \) to \( k \) is

\[ I_k = (V_s - V_k)Y' = \left\{ \frac{V_i}{a + jb} - V_k \right\} Y'. \]

The power flowing into node \( k \) thus is

\[ S_k = V_k I_k^* = V_k V_i^* \frac{Y^*}{a - jb} - V_k V_k^* Y^*. \quad (2-28) \]

To compute the power flowing in or out of node \( i \), all the values have to be referred to the side on which the taps are. The admittance then is

\[ Y' = \frac{I_p}{V_p} = \frac{I_s}{V_s(a + jb)} = \frac{1}{a^2 + b^2} = Y' \]

\( Z' = 1/Y' \) is now considered to be an impedance inserted between nodes \( p \) and \( k \), and replaces the impedance \( Z \). The current flowing in this connection is

\[ I_i = (V_p - V_i)Y' = (a + jb)V_k - V_i \frac{Y'}{a^2}, \]

and the power flowing into node \( i \) is

\[ S_i = V_i I_i^* = V_i V_k^* (a - jb) \frac{Y^*}{a^2} - V_k V_k^* \frac{Y^*}{a^2}. \quad (2-29) \]

The admittances for the computation of the power flow into/out of the nodes \( k \) and \( i \) now differ. So the symmetry present in the transmission line equations — in equation 2-23 the power injected into node \( k \) instead of node \( i \) is found by simply swapping the indices \( i \) and \( k \) — is lost. Still, the similarity between these equations is big enough that connections with transformers in them are described in the same way as transmission lines. If a transformer
is present, the impedances of transmission line and transformer are lumped, and the tap setting \((a + jb)\) is given in addition to the impedance and capacitance.

Although the formulas above have all been deduced for a transformer with a phase shift, phase shifters are in fact seldom found in power systems due to their prohibitive cost. Thus tap settings in most load flow analyses are real.

Again the equations can be formulated in polar or rectangular coordinates.

**Polar formulation**

With the admittance written as \(Y = G + jB\) the formulation of equation (2-28) is

\[
S_k = V_k V_i (\cos \theta_{ki} + j \sin \theta_{ki}) \frac{G - jB}{a - jb} - V_k^2 (G - jB) \quad (2-30)
\]

and equation (2-29) is

\[
S_l = V_i V_k (\cos \theta_{lk} + j \sin \theta_{lk}) \frac{(a - jb)(G - jB)}{t^2} - V_i^2 \frac{G - jB}{t^2} \quad (2-31)
\]

**Rectangular coordinate formulation**

Equation (2-28) is written as

\[
S_k = (e_k + jf_k)(e_i - jf_i) \frac{G - jB}{a - jb} - (e_k^2 + f_k^2)(G - jB) \quad (2-32)
\]

while equation (2-29) is written as

\[
S_l = (e_i + jf_i)(e_k - jf_k) \frac{(a - jb)(G - jB)}{t^2} - (e_i^2 + f_i^2) \frac{(G - jB)}{t^2} \quad (2-33)
\]

For the implementation of a load flow program the similarity of the equations describing the power flow through a transformer with those describing a transmission line (equations (2-24) and (2-25)) can be exploited to good effect.

**Per unit calculations and transformers.**

As remarked in section 2.3, the base values of voltage and current are generally different in the parts of the circuit connected through a transformer, while the base power remains the same. It was also stated that, if the voltage bases have the same ratio as the turns ratio of the transformer windings, impedance will be the same when it is expressed in p.u. on the tap changing side of the transformer as when it is expressed in p.u. on the other side of the transformer.

Usually the transformation ratio is not exactly equal to the ratio of the voltage bases, but nearly so. The reason that the transformation ratio is not exactly one, is that the
voltage bases would then have to be changed every time the taps are moved. In the UK many calculations do not use per unit values for this reason. Thus the transformation ratio for the voltages expressed in p.u. is usually not exactly one, but close to one.

Consider e.g. the power system shown in figure 2-3 on page 17. For the transformers the data is

<table>
<thead>
<tr>
<th>Transformer designation</th>
<th>Tap setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>4–7</td>
<td>0.978</td>
</tr>
<tr>
<td>4–9</td>
<td>0.969</td>
</tr>
<tr>
<td>5–6</td>
<td>0.932</td>
</tr>
</tbody>
</table>

Both transformers do not introduce a phase shift but only a slight per unit change in the voltage. The ordering of the buses indicates on which side of the transformer the taps are. In the first case the taps are at node 4, so that

\[
\frac{V_4}{V_7} = 0.978.
\]

### 2.4.4 Capacitor banks and synchronous compensators

As mentioned at the end of section 2.2 capacitor banks and synchronous compensators are sometimes available in a power system. These can supply all or part of the reactive power required by the loads (electrical motors act as inductors), which causes a reduction in line currents.

Busses may also be equipped with inductors. If there is little load on the system, the whole system acts like a huge capacitor due to the capacitance in the lines. This effect can be neutralised by inductors.

In the power system shown in figure 2-3 on page 17 synchronous compensators are attached to buses 3, 6 and 8, and a capacitor bank is attached to bus 9.

**Capacitor banks**

Capacitor banks are used infrequently in power systems, as huge banks are required to have any real effect. If the susceptance of a capacitor is \(jB_S\), then the factor that has to be added to the reactive power injected into a node \(i\) is

\[
\Delta Q_H = |V_i|^2 jB_S. \tag{2-34}
\]

Note the similarity with the equations (2-23) for transmission lines: the capacitance of the transmission line allocated to node \(i\) is added to the reactive power as a factor \(|V_i|^2 B_C\). The sum of the capacitances of the connected transmission lines and the capacitance of a
capacitor bank can be taken as a lump factor called the *shunt reactance*. Inductance is negative capacitance, so that inductors and capacitors can be modelled in the same way, but for inductors a negative value is assigned to $B_S$.

**Synchronous compensators**

Synchronous compensators are more generally usable than capacitor banks, as they can deliver a variable amount of leading or lagging reactive power. To explain what a synchronous compensator is, it is necessary to discuss how generators control the amount of reactive power drawn from or supplied to the system.

A diagram of a generator is shown in figure 2-8. The field winding on the rotor generates a strong magnetomotive force (mmf), which combines with the mmf produced by current in the armature winding. The resultant flux across the air gap generates voltage in the coils of the armature winding. Changing the excitation of the generator, that is the size of the mmf generated by the field winding, is an important factor in controlling the flow of reactive power [124, p.137].

An overexcited generator supplies lagging current and an under-excited generator supplies leading current. Supplying leading current is equivalent to drawing lagging current, which means that reactive power is drawn from the system.

A synchronous compensator is an electrical motor, with its excitation adapted to the reactive power requirements. Although no real power is produced, reactive power is ‘supplied’ by such a device.

A bus equipped with a synchronous compensator is interpreted as a generator bus which produces only reactive power, in other words $P_G = 0$ and $Q_G$ is variable with limits imposed


<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{Gi} = P_{Gi} + jQ_{Gi}$</td>
<td>The power generated at bus $i$</td>
</tr>
<tr>
<td>$S_{Di} = P_{Di} + jQ_{Di}$</td>
<td>The power demand at bus $i$</td>
</tr>
<tr>
<td>$S_{ii} = P_{ii} + jQ_{ii}$</td>
<td>The power injected into bus $i$ from the network</td>
</tr>
<tr>
<td>$V_i = V_i + j\theta_i$</td>
<td>The complex voltage at node $i$</td>
</tr>
<tr>
<td>$Y_{ik} = G_{ik} + jB_{ik}$</td>
<td>The admittance of a connection $i - k$</td>
</tr>
<tr>
<td>$B_{Gik}$</td>
<td>Half the susceptance of transmission line $i - k$</td>
</tr>
<tr>
<td>$t_{ik} = \phi_{ik}$</td>
<td>The tap setting of a transformer in line $i - k$</td>
</tr>
<tr>
<td>$Y_{Si}$</td>
<td>The susceptance of a capacitor bank.</td>
</tr>
</tbody>
</table>

Table 2-1: Notations used in the load flow problem.

by the characteristics of the synchronous compensator.

To summarise: In order to reduce line currents capacitor banks and synchronous compensators are used to generate some or all of the required reactive power at a bus. Capacitor banks are handled similarly to the capacitance of transmission lines, but do not occur very often. Synchronous compensators are more common, and are viewed as generators which do not produce any real power.

2.5 Modelling a power system

Models for all the components of a power system relevant to load flow analysis were described in the last few sections. The remaining problem is the assimilation of these equations to gain a complete model of a power system. First of all the concept of self-admittance is introduced, before the complete nodal representation of a power system is given in section 2.5.2. Comments are made on the interaction between real and reactive power and the variables associated with different types of busses, as this turns out to be an important issue for the development of algorithms. Lastly issues connected with regulated busses are addressed.

All notations used are listed in table 2-1. With the notation $(\mathcal{V}, \mathcal{E})$ for a graph introduced on page 18, define the sets

$$\mathcal{G}(i) := \{ k \in \mathcal{V} \mid (i, k) \in \mathcal{E} \} \quad \text{and} \quad \bar{\mathcal{G}}(i) := \mathcal{G}(i) \cup \{i\}.$$  

Thus $k \in \mathcal{G}(i)$ if a transmission line exists that connects $i$ to $k$.  

2.5. MODELLING A POWER SYSTEM

<table>
<thead>
<tr>
<th>Transmission Line</th>
<th>$Y_{ik} := Y$</th>
<th>$Y^{(k)}_{ii} := Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>$Y_{ik} := \frac{Y}{a + jb}$</td>
<td>$Y^{(k)}_{ii} := Y$ if tap on side $k$</td>
</tr>
<tr>
<td></td>
<td>$Y_{ik} := \frac{(a + jb)Y}{i^2}$</td>
<td>$Y^{(k)}_{ii} := \frac{Y}{i^2}$ if tap on side $i$.</td>
</tr>
</tbody>
</table>

Table 2-2: Self-admittance due to connections to node $i$.

2.5.1 Self-admittance of a node

Equations (2-12) specify the power balance that has to be satisfied in a power system

$$S_{Gi} - S_{Di} + S_{fi} = 0 \quad i = 1, \ldots, n. \quad (2-35)$$

The demand $S_{Di}$ remains constant during a load flow analysis, the power generation $S_{Gi}$ is adapted after every iteration of the solution method, and the injected power $S_{fi}$ is determined by the power flow in the system. To simplify the definition of $S_{fi}$, define the admittances $Y_{ik}$ as shown in table 2-2.

Equations (2-23), (2-28) and (2-29) then imply that the total power flowing into node $i$ is

$$S_{fi} = \sum_{k \in G(i)} \left\{ V_i V_k^* Y_{ik} - V_i^2 Y^{(k)*}_{ii} + j B_{Cik} V_i^2 \right\} + j B_{Si} V_i^2.$$  

with $B_{Cik}$ the capacitance of a transmission line and $B_{Si}$ any additional shunt admittance introduced through for example capacitor banks.

Define the self admittance

$$Y_{ii} := - \sum_{k \in G(i)} (Y_{ii}^{(k)} + j B_{Cik}) - j B_{Si}$$

and then it follows that

$$S_{fi} = \sum_{k \in G(i)} V_i V_k^* Y_{ik}.$$  

(2-36)

2.5.2 The nodal load flow model

The load flow problem is now specified as a function for the mismatch in the equations (2-35). The complex power mismatch $S_i$ at node $i$ is

$$\Delta S_i = S_{Gi} - S_{Di} + \sum_{k \in G(i)} V_i V_k^* Y_{ik}^*.$$  

(2-37)
A solution to the load flow problem is a set of values for all the variables in the system such that the equations (2-35) are satisfied, i.e. such that the power mismatches \( \Delta S_i = 0 \) for all \( i \in \mathcal{V} \). This simple representation is the big advantage of the nodal model for load flow; it is not necessary to determine loops in networks etc. as with other models.

Define the set of nonlinear equations

\[
F \overset{\text{def}}{=} (\Delta S_1, \Delta S_2, \ldots, \Delta S_n).
\]  

(2-38)

The load flow problem is solved if a zero of the function \( F \) is found for a given set of parameters. The question of which values are parameters and which are variables at different types of busses is discussed shortly.

**Polar formulation.**

The polar formulation of the equations (2-37) for \( i = 1, \ldots, n \) is

\[
\begin{align*}
\Delta P_i &= P_{Gi} - P_{Di} + \sum_{k \in \mathcal{G}(i)} V_i V_k \{ G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik} \} \\
\Delta Q_i &= Q_{Gi} - Q_{Di} + \sum_{k \in \mathcal{G}(i)} V_i V_k \{ G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik} \}.
\end{align*}
\]  

(2-39a)

(2-39b)

In this formulation the power mismatch function is ordered either as

\[
F_1 = (\Delta P_1, \Delta P_2, \ldots, \Delta P_n, \Delta Q_1, \ldots, \Delta Q_n)
\]

or as

\[
F_2 = (\Delta P_1, \Delta Q_1, \Delta P_2, \Delta Q_2, \ldots, \Delta P_n, \Delta Q_n).
\]

(2-40a)

(2-40b)

Algorithms using both orderings are described in the next chapter.

The number of nodes in a network can vary between tens and several thousand, and every node has only a few other nodes connected to it. Typically \( |\mathcal{G}(i)| \) is 3 or 4 [47]. This means that the system of nonlinear equations defined by the above equations is extremely sparse.

The formulation for the mismatch equations in rectangular coordinates is analogous to (2-39), but it is rarely used in practice as it is often easier to find a starting value for load flow algorithms for the polar than for the rectangular representation (this is discussed below). The only algorithm for which the rectangular formulation dominates is the second order load flow method discussed in section 3.6, where more will be said about the rectangular formulation.
2.5. MODELLING A POWER SYSTEM

2.5.3 Variables and Parameters

As the power demand is fixed, it follows from the equations (2-39) that there are four variables per node. These are: the real power generation \( P_{Gi} \), the reactive power generation \( Q_{Gi} \), the phase angle \( \theta_i \) and the voltage magnitude \( V_i \). Two variables have to be specified, and then the other two are computed when solving the nonlinear system (2-40a).

Refer to the classification of busses in section 2.2. At a load bus no power is generated, so that \( S_{Gi} = 0 \), and the voltage magnitude and angle remain variable. As the power generation is fixed (to zero), such a bus is called a PQ bus. At a generating bus the real power generation and the voltage magnitude are fixed, while the reactive power generation is left variable. Such a bus is called a regulated or PV bus. At the swing bus the power generation must be variable as the losses in the system are not known beforehand, so that the voltage magnitude and angle are preset. To summarise:

<table>
<thead>
<tr>
<th>Bus</th>
<th>Specified</th>
<th>Variable</th>
<th>( P_G ) – Real power generation</th>
<th>( Q_G ) – Reactive power generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>(-)</td>
<td>(</td>
<td>V</td>
<td>, \theta)</td>
</tr>
<tr>
<td>Generator</td>
<td>(</td>
<td>V</td>
<td>, P_G)</td>
<td>(Q_G, \theta)</td>
</tr>
<tr>
<td>Swing</td>
<td>(</td>
<td>V</td>
<td>, \theta)</td>
<td>(P_G, Q_G)</td>
</tr>
</tbody>
</table>

**Interaction of variables**

For many solution methods for the system of \( 2n \) nonlinear equations (2-40a) it is essential to know with which variables the real and complex part of the power mismatch equations interact strongly.

It is worthwhile to follow the train of thought presented in Talukdar and Wu [129]. The line conductances \( G_{ik} \) are usually small \((\approx \frac{1}{3} B_{ik})\), the difference between \( \theta_i \) and \( \theta_k \), where \( i \) and \( k \) are directly connected nodes, is usually small (less than 20°) and the magnitude of the voltage \( V_i \) rarely deviates from its nominal value by more than 10%. Approximate \( G_{ik} \sin(\theta_i - \theta_k) \approx 0 \) and \( \cos(\theta_i - \theta_k) \approx 1 \). Then the reactive power equation is (see (2-13))

\[
Q_{Gi} - Q_{Di} + \sum_{k \in G(i)} V_i V_k B_{ik} \approx 0 . \tag{2-41a}
\]

If the assumption is added that \( V_i \approx 1 \) and \( \sin \theta_{ik} \approx \theta_{ik} \), the real power equation becomes

\[
P_{Gi} - P_{Di} + \sum_{k \in G(i)} B_{ik} \theta_{ik} \approx 0 . \tag{2-41b}
\]

According to Talukdar and Wu [129] this just states a well known fact — real power flows are strongly affected by the angles of node voltages and weakly affected by magnitudes, while reactive power flows are strongly affected by the magnitudes of node voltages and only weakly affected by their angles. This is also the reason for fixing \( P \) and \( V \) at PV busses and not \( Q \) and \( \theta \).
From the foregoing discussion it is clear that, if the voltage angle is variable at a specific node the real power balance equation for that node should be included in the system of equations, and if the voltage magnitude is a variable the reactive power equation should be included.

This decoupling is not always as strong as it may seem here. In the case of underground lines for example \(|G_{ik}| \approx |B_{ik}|\). Although the decoupled equations (2-41b) usually give a good starting point for other algorithms, most problems do not warrant ignoring the coupling terms altogether.

If the equations are formulated in rectangular coordinates a similar train of thought can be followed. The voltage angle and magnitude are not available any more, but only the complex number specifying the voltage

\[ e_i + jf_i = V_ie^{j\theta_i}. \]

If \(\theta_i\) is small, it is seen from figure 2-2 on page 15 that \(f_i\) has to be small, i.e. for small \(\theta_i\), \(f_i\) and \(\theta_i\) are correlated. A similar argument holds for the relation between \(e_i\) and \(V_i\): if \(\theta_i\) remains small throughout the computation, a change in angle will not influence the size of \(e_i\) strongly, but a change in \(V_i\) will have a significant effect on \(e_i\). From this it follows that a coupling between \(Q_i\) and \(e_i\) and between \(P_i\) and \(f_i\) can be stipulated.

Unfortunately the angle of the voltage relative to a reference bus is not always small. Even if the differences between voltage angles in connected busses is small, the difference in voltage angle with respect to the reference bus can become large. This suspicion is confirmed by a comment made by Peterson and Meyer in their discussion of Stott [125]. They state that ‘in this country [US] it is generally the case for large power flow studies that phase angles on the periphery are 60 or 70 degrees out of phase with the slack bus’. This causes additional difficulties for most algorithms which is why the polar representation of load flow problems is almost universally preferred.

### 2.5.4 Regulated (PV) busses

A **regulated bus** is a bus connected to a reactive power source, with the voltage magnitude kept constant. That the voltage magnitude rather than the voltage angle is kept constant is unsurprising after the discussion of the previous section. Generator busses, except for the special case of the swing bus, and those load busses that have synchronous compensators connected to them, are regulated busses.

As the reactive power generation is only a linear variable in the equations (2-39), the size of the system is usually reduced through elimination of the reactive power equation. Incidentally the same is true for the real and reactive power at the swing bus. With every iteration these values are computed from the newest estimates for the voltage magnitudes.
2.6. DATA REQUIREMENTS

<table>
<thead>
<tr>
<th>Bus</th>
<th>Variables</th>
<th>Parameters</th>
<th>Corresponding Equation</th>
<th># Equation(s) in System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load (PQ)</td>
<td>$\theta_i$</td>
<td>$P_i = 0$</td>
<td>$\Delta P = 0$</td>
<td>2</td>
</tr>
<tr>
<td>Bus</td>
<td>$V_i$</td>
<td>$Q_i = 0$</td>
<td>$\Delta Q = 0$</td>
<td></td>
</tr>
<tr>
<td>Generator (PV)</td>
<td>$\theta_i$</td>
<td>$P_i = P_{Gi}$</td>
<td>$\Delta P = 0$</td>
<td>1</td>
</tr>
<tr>
<td>Bus</td>
<td>$Q_{Gi}$</td>
<td>$V_i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swing</td>
<td>$P_{Gi}$</td>
<td>$\theta_i$</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Bus</td>
<td>$Q_{Gi}$</td>
<td>$V_i$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-3: The different nodes in a power system and their associated variables.

and angles in all the connected busses, and they are then seen as fixed parameters for the next solution of the system of equations.

Regulated busses thus only contribute to the first equation of (2-39) of the system of nonlinear equations, while load busses contribute to both equations and the swing bus to none. The variables associated with different types of nodes in a power system are listed in table 2-3.

As was indicated in section 2.4.4 the amount of reactive power that can be generated at regulated busses underlies some restrictions. These generally take the form

$$Q_{Gi}^{Min} \leq Q_{Gi} \leq Q_{Gi}^{Max}$$

When the upper or lower bound is reached the reactive power generation is fixed at this bound and the voltage is left variable, i.e. the PV bus becomes a PQ bus. This implies that an additional equation must be added to the system of nonlinear equations. Should the voltage become higher (lower) than the specified value, the voltage magnitude is again fixed at its previous, constant value and the reactive power is left variable.

This completes the description of the load flow problem and the associated variables and parameters. The last obstacle to the practical solution of load flow problems is the interpretation of the available data.

2.6 Data requirements

In this section the data that is required for a load flow study is itemised. Data for a load flow analysis of the power system shown in figure 2-3 is given in tables 2-4 to 2-8. The data is discussed table by table, mentioning any necessary conversion of the input data.

The accuracy of the data is relevant to the evaluation of solution methods. Due to the high currents and voltages involved, the accuracy of measurements in power systems is quite low. An error of up to 5% is not uncommon for high voltages. It follows that load flow
problems do not have to be solved to a very high accuracy, as they can only give estimates of a situation. Consequently an error

\[ \max \{ \| \Delta P \|_\infty, \| \Delta Q \|_\infty \} < \epsilon \]

with \( \epsilon = 10^{-4} \) is acceptable.

2.6.1 Impedance and line charging data

Modelling a transmission line requires the following data:

1. indices of the nodes connected by the line;
2. impedance \( Z = R + jX \) of the line;
3. capacitance \( B_C \) of the line.

The impedance and line charging data for the IEEE 14 test system [119] is given in table 2-4. The capacitance (line charging) is neglected for all lines, except those connecting.
2.6. DATA REQUIREMENTS

Operating conditions

<table>
<thead>
<tr>
<th>Bus</th>
<th>Starting bus voltage</th>
<th>Generation</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Magnitude Phase angle</td>
<td>MW MVar</td>
<td>MW MVar</td>
</tr>
<tr>
<td>1(^\d)</td>
<td>1.060 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>1.000 0.0</td>
<td>40.0 0.0</td>
<td>21.7</td>
</tr>
<tr>
<td>3</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>9</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>10</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>11</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>13</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>14</td>
<td>1.000 0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 2-5: Operating conditions for the IEEE 14 bus test system.

nodes 1–5. These are the longer, high voltage lines. The rest of the lines are shorter, low voltage lines for which the capacitance can readily be ignored. The reason for this is that, in p.u., the factor introduced by the capacitance is much bigger for high voltage lines than for low voltage lines.

Connections through transformers are also included in this table. As discussed in section 2.4.3 such connections require some data similar to that for transmission lines.

It is possible to construct a complete graph of the power system from this data table (see figure 2-4 on page 17). This graph is necessary to determine the structure of the admittance matrix and the Jacobian of the system of nonlinear equations.

A last remark: the impedance of the connections is given here, while the admittance is used throughout. The impedances are converted to admittances at the start of the computation.

2.6.2 Operating conditions

The information required about a bus is the

1. number of the bus;
2. voltage magnitude at swing bus;
3. starting values for voltage magnitude and angle;


**Table 2-6**: Regulated bus data for the IEEE 14 bus test system.

<table>
<thead>
<tr>
<th>Bus number</th>
<th>Voltage magnitude</th>
<th>Minimum MVAR capacity</th>
<th>Maximum MVAR capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.045</td>
<td>-40.0</td>
<td>50.0</td>
</tr>
<tr>
<td>3</td>
<td>1.010</td>
<td>0.0</td>
<td>40.0</td>
</tr>
<tr>
<td>6</td>
<td>1.070</td>
<td>-6.0</td>
<td>24.0</td>
</tr>
<tr>
<td>8</td>
<td>1.090</td>
<td>-6.0</td>
<td>24.0</td>
</tr>
</tbody>
</table>

4. real power generation (except for swing bus);

5. real and reactive power demand;

6. the index of the swing bus.

The operating conditions data for the IEEE 14 test system is given in table 2-5. Regulated busses are treated separately, and the voltage magnitude values given here should be ignored.

The real power generation is given for bus 2, which, except for the swing bus, is the only generator bus in the system. The starting values for the reactive power generation are set to 0. The real and reactive power demand is given for the load busses, or is set to zero for the busses which have no load attached to them. The power is usually given in MW, while all computations are done in p.u.. Usually the base value for the power is 100 MW, so all values have to be divided by 100 to convert them to p.u..

Only the swing bus can be identified from this table. To determine which of the remaining busses are load, and which are generator busses, the information from the Regulated Bus Data table is needed.

### 2.6.3 Regulated bus data

Additional data required for regulated busses is the

1. voltage magnitude;

2. maximum and minimum capacity for reactive power generation.

For the IEEE 14 test system this data is given in the Regulated Bus Data table 2-6. With this information the problem of the classification of busses can be resolved: all busses given in this table are generator busses, while all remaining busses are load busses. In this case busses 3, 6 and 8 are not generator busses but load busses with synchronous compensators attached to them.
### Transformer data

<table>
<thead>
<tr>
<th>Transformer</th>
<th>Tap setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 - 7</td>
<td>0.978</td>
</tr>
<tr>
<td>4 - 9</td>
<td>0.969</td>
</tr>
<tr>
<td>5 - 6</td>
<td>0.932</td>
</tr>
</tbody>
</table>

Table 2-7: Transformer data for the IEEE 14 bus test system.

### Static capacitor data

<table>
<thead>
<tr>
<th>Bus Number</th>
<th>Susceptance (p.u.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Table 2-8: Static capacitor data for the IEEE 14 bus test system.

### 2.6.4 Transformer data

Most of the data required for connections containing transformers is given in the impedance and line charging table (table 2-4), but the

1. indices of the nodes connected by the transformer;
2. tap setting

are still needed. These are given in table 2-7.

Most transformers encountered in practice do not contain phase shifters as these are too expensive. So the tap settings in these cases are real instead of complex values. The first node given in the connection indicates the side on which the taps are. The ratio of the primary and secondary voltages can then be determined as shown in section 2.4.3.

### 2.6.5 Static capacitor data

Lastly the data for the static capacitors is given in table 2-8 namely the

1. number of the bus where a capacitor bank is available;
2. susceptance of the capacitor bank.

The susceptance is given in p.u. and causes no further difficulties.

This concludes the summary of the data needed for a load flow problem.

### 2.7 Test problems

It is very difficult to obtain realistic data for testing as most of the needed information is confidential. Fortunately the IEEE published a series of power systems, which were used in this project. They are known as the IEEE test systems: these are systems with 14, 30, 57 [119] and 118 [3] busses.
<table>
<thead>
<tr>
<th>Name</th>
<th>Number of nodes</th>
<th>Constructed from systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieeel4</td>
<td>14</td>
<td>IEEE test system</td>
</tr>
<tr>
<td>ieeel30</td>
<td>30</td>
<td>IEEE test system</td>
</tr>
<tr>
<td>ieeel57</td>
<td>57</td>
<td>IEEE test system</td>
</tr>
<tr>
<td>ieeel118</td>
<td>118</td>
<td>IEEE test system</td>
</tr>
<tr>
<td>pow236</td>
<td>236</td>
<td>ieeel118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ieeel118</td>
</tr>
<tr>
<td>pow529</td>
<td>529</td>
<td>pow236</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ieeel57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow236</td>
</tr>
<tr>
<td>pow795</td>
<td>795</td>
<td>pow529</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ieeel30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ieeel118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ieeel118</td>
</tr>
<tr>
<td>pow1419</td>
<td>1419</td>
<td>pow236</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow236</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow236</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow236</td>
</tr>
</tbody>
</table>

Table 2-9: The power systems used as test systems.

On modern computers the solution time for these systems is so short that they do not benefit from parallel computation. To obtain data for larger systems an attempt was made to construct larger power systems using the IEEE systems as building blocks. This is not a straightforward process as the resulting systems have to be realistic enough to have both a solution and some semblance to reality\(^1\).

In most cases IEEE test systems were connected by only a few transmission lines and the first question that arises is whether this approach is realistic. The obvious objection is that partitioning of the graph, which is important for all parallel algorithms, may be unduly simplified and that the resulting cut-sets are smaller than can realistically be expected for large power systems.

However, if the different blocks are seen as subnetworks belonging to different power companies, or networks corresponding to separate urban areas, a small number of connections is likely. A second argument is that loosely coupled sections already exist in most power systems: as transformers are expensive, there are usually only very few connections between the high and low voltage parts of a network.

The second concern is in how far the artificial networks are representative of real-life power systems. Especially with the larger power systems it was quite difficult to find a choice of swing bus, and a choice of connections that led to a system for which a load flow

---
\(^1\)The assistance of Dr. Adams, Department of Electrical Engineering at Queen Mary & Westfield College, with the construction of these systems is gratefully acknowledged.
solution could be found. In all these cases the convergence process showed widely varying voltage angles and negative voltage magnitudes, which indicates a highly unstable system. Such systems do not occur in practice. The systems that were finally chosen all converge easily and show, as expected, an even increase of voltage angles to the edges of the power system. All voltage magnitudes are positive and within reasonable limits.

The largest system created in this way has 1419 busses, which is still not very large. In the literature there are many references to the solution of systems with 4000 – 10000 busses. Artificial systems larger than 1419 busses all exhibited unrealistic behaviour, such as widely different voltage angles in adjoining nodes and negative voltages. A summary of the power systems is given in table 2-9.
Chapter 3

Load Flow Methods

The first description of load flow analysis on computers was published in 1946 by Jennings and Quinan [73] and innovatively named ‘The use of Business Machines in Determining the Distribution of Load and Reactive Components in Power Line Networks’.

Since then load flow analysis has progressed rapidly. The development of algorithms for digital computers started in the fifties with papers by Ward and Hale [138] and Brown and Tinney [11]. These were all enhancements of the Gauß-Seidel method. The next big advance was the use of the Newton-Raphson method for the solution of the load-flow problem. Many derivatives of the Newton-Raphson method have been developed since, most of which concentrate on simplifying the Jacobian and using a constant iteration matrix. An overview of these early developments is given in Stott [126].

The objective of this chapter is to survey popular load flow methods and to compare these theoretically where possible. The motivation for this is that load flow problems are very complex and timings are too implementation dependent to provide a reliable comparison. Another reason for looking at the theory in such detail is that it provides some clues as to which approaches to parallelisation will and which will not work.

Several classical methods for load flow analysis are discussed. These are the

3.2 Gauß-Seidel method

3.3 Newton-Raphson method

3.4 Decoupled Newton method

3.5 Fast Decoupled Load Flow (FDLF) method

3.6 Second-Order Load Flow (SOLF) method

The FDLF method is the current industry standard for load flow computations.
3.1 DC load flow analysis

Direct Current (DC) load flow analysis was used extensively when computers were too weak to cope with full load flow analysis. DC load flow analysis greatly simplifies the model by only calculating voltage angles. The results are consequently quite inaccurate. Nowadays DC load flow is still used to find good starting values for iterative solution methods.

DC load flow analysis only takes the real power mismatch into account [46, p.192]. The real power flowing through a single connection in the all-polar formulation (the first equation in (2-17)) with $Y_{ij}^* = |Y_{ij}|e^{-j(\pi/2 - \xi_{ij})}$ is

$$P_i = V_i V_j |Y_{ij}| \cos(\theta_{ij} + \pi/2 - \xi_{ij}) + V_i^2 |Y_{ij}| \cos(\pi/2 - \xi_{ij})$$

$$= V_i V_j |Y_{ij}| \sin(\theta_{ij} - \xi_{ij}) + V_i^2 |Y_{ij}| \sin \xi_{ij}$$

(3-1)

The second term is the line’s contribution to the self-admittance at node $i$. In section 2.5.3 it was mentioned that usually $G_{ij} \ll B_{ij}$ (or $R_{ij} \ll X_{ij}$) so that the angle $\xi_{ij}$ is very small and is set to zero. It was also mentioned that the coupling between the real power mismatch and the voltage magnitude is weak. As the voltage magnitude is usually close to 1 p.u., it is set to 1. Furthermore $\theta_{ij}$ is small so that $\sin \theta_{ij} \approx \theta_{ij}$. Equation (3-1) then becomes

$$P_i = |Y_{ij}| \theta_{ij}$$

(3-2)

and the solution of this linear system is the DC load flow method. The DC load flow method does provide good starting values for the voltage angles $\theta_i$ which can be large at the periphery (up to $60^\circ - 70^\circ$ in large load flow studies) [106]. This can improve the convergence of iterative load flow methods.

3.2 The Gauß-Seidel method

Equation (2-37) gives an expression for the power mismatch $\Delta S_i$ in every bus $i$. It follows that the current mismatch in bus $i$ is

$$\Delta I_i = \frac{\Delta S_i^*}{V_i^*} = \frac{S_{G_i}^* - S_{D_i}^*}{V_i^*} - \sum_{j \in G(i)} V_j Y_{ij}$$

so that the voltage mismatch is

$$\Delta V_i = \frac{\Delta I_i}{Y_{ii}} = \frac{\Delta S_i^*}{V_i^* Y_{ii}} = \frac{\Delta P_i - j \Delta Q_i}{V_i^* Y_{ii}}$$

(3-3)

The Gauß-Seidel method successively solves (3-3). In step $i$ the newest estimates for $V_1$ to $V_{i-1}$ are used to calculate the power mismatch $\Delta S_i$. A full Gauß-Seidel iteration consists of calculating new estimates for all voltages.
CHAPTER 3. LOAD FLOW METHODS

At PV busses the voltage magnitude \(|V_i|_{sp}\) is specified, while the reactive power generation \(Q_{Gi}\) is not available. This problem is resolved by setting \(\Delta Q_i = 0\) in (3-3), \(Q'_{Gi} := Q_{Gi} - \Delta Q_i\) and re-evaluating \(\Delta V_i\). The magnitude of the computed voltage \(V'_i = V_i + \Delta V_i\) is then reset to \(|V_i|_{sp}\), while maintaining its new angle.

The method is assumed to have converged when \(\Delta S_i \leq c\) in all busses, where \(c\) is a preset constant. Evaluating busses one at a time makes it very easy to take bounds on reactive power generation at PV busses into account.

Very modest storage requirements are the main advantage of the Gauß-Seidel method. As there are usually no more than three off-diagonal elements per row in the admittance matrix, the computational requirements per iteration are equally modest (proportional to the number of busses \(n\)). For well-conditioned systems the number of iterations required is of order \(n\) — for large problems the Gauß-Seidel method is therefore not competitive with higher order methods like Newton-Raphson.

A good starting value for this and most other methods is a \textit{flat start}. A flat start is defined as a a choice of starting values such that all angles \(\theta_i = 0\), except at the swing bus where it is the prescribed angle (usually also 0). Furthermore all voltages are chosen as \(V_i = 1\), except at regulated busses where they are the prescribed voltage. With this choice of angles \(\theta_{ik} = 0\), which is reasonable as angle differences between adjacent nodes are usually small, and \(V_i = 1\) p.u. is usually very close to the final value for the voltage.

As mentioned in the previous section the DC method provides an even better starting value. As most comparisons between load flow methods in the literature use a flat start, all computational experiments reported in this thesis will use a flat start.

3.3 Newton methods

The Newton-Raphson method for the solution of a set of nonlinear equations \(f(x) = 0\), with \(J(x^{(k)})\) the Jacobian of \(f\) evaluated at \(x^{(k)}\), is:

The Newton-Raphson method.

1. Choose a starting vector \(x^{(0)}\)

2. While not converged
   
   \((a)\) Solve \(J(x^{(k)})s^{(k)} = f(x^{(k)})\)
   
   \((b)\) Set \(x^{(k+1)} = x^{(k)} - s^{(k)}\)

\(J\) is very sparse, and this sparsity must be taken into account when solving the linear system in step 2(a). For a discussion of the sparse solution of such systems see section 3.8.1 of this thesis.
If the starting point is 'close enough' to the solution, the convergence of the Newton method is quadratic. If the function \( f \) is linear, a solution is obtained in a single iteration, and if \( f \) is not 'too nonlinear', convergence is very fast. For details on the convergence analysis see for example Rheinboldt and Ortega [111] or Dennis and Schnabel [26].

In load flow analysis the Newton method is used with great success. A good starting value is usually \( V_i = 1 \), due the use of p.u. values, and \( \theta_i = 0 \).

Recall the function \( F_1 \) as defined in (2.40a)

\[
F_1 = (\Delta P_1, \Delta P_2, \ldots, \Delta P_n, \Delta Q_1, \ldots, \Delta Q_n).
\]

In order to solve the load flow problem, the Newton method is used to find a zero of \( F_1 \).

For PV busses only, a real equation must be included to compute the new voltage angle, i.e. if \( i \) is a regulated bus \( \Delta Q_i \) is discarded from \( F_1 \). After every iteration the reactive power mismatch in PV buses is computed and \( Q'_G_i \) is set to \( Q'_G_i := Q_Gi - \Delta Q_i \). As mentioned in section 2.5 the reactive power equation for a PV bus must be included in the system when the reactive power generation hits its limits.

Instead of the polar, the rectangular formulation of the power mismatch functions may be used. By doing so the evaluation of trigonometric functions is avoided. But, as the voltage is not split into magnitude and angle, the voltage magnitude has to be rescaled to \( |V_i|_{sp} \) after every iteration, while maintaining the new angle.

Stott [126] reports that the rectangular version was found to be slightly less reliable and slower then the polar version. Here only the latter is considered.

For the development of the following methods it is convenient to write the linear system (3-4) in the partitioned form

\[
\begin{bmatrix}
\Delta P \\
\Delta Q
\end{bmatrix} =
\begin{bmatrix}
H & N \\
M & L
\end{bmatrix}
\begin{bmatrix}
\Delta \theta \\
\Delta V
\end{bmatrix}
\]

(3-5)

where

\[
H_{ik} = -\frac{\partial \Delta P_i}{\partial \theta_k} \quad N_{ik} = -\frac{\partial \Delta P_i}{\partial V_k}
\]

\[
M_{ik} = -\frac{\partial \Delta Q_i}{\partial \theta_k} \quad L_{ik} = -\frac{\partial \Delta Q_i}{\partial V_k}
\]

Equation (3-5) is a representation of the linear system \( J(x^{(k)})s^{(k)} = f(x^{(k)}) \) that is solved in every iteration of the Newton method (3-4).

In (3-5) the equations for the swing bus and the reactive equations for the PV busses are omitted, as are the corresponding voltage magnitude and angle corrections. For future reference the analytical formulation of the elements of the Jacobian are given in table 3-1.

Usually \( \Delta P_i \) is divided by \( V_i \) to make the terms in \( N \) and \( L \) more similar to the terms in \( H \) and \( M \). Computationally there is some gain, while the result is unaffected.
\[-H_{ii} = \frac{\partial \Delta P_i}{\partial \theta_i} = -V_i \sum_{k \in \mathcal{G}(i)} V_k \{G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik}\}\]
\[-H_{ik} = \frac{\partial \Delta P_i}{\partial \theta_k} = V_i V_k \{G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik}\}\]
\[-N_{ii} = \frac{\partial \Delta P_i}{\partial V_i} = \sum_{k \in \mathcal{G}(i)} V_k \{G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik}\} - 2V_i G_{ii}\]
\[-N_{ik} = \frac{\partial \Delta P_i}{\partial V_k} = V_i \{G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik}\}\]
\[-M_{ii} = \frac{\partial \Delta Q_i}{\partial \theta_i} = V_i \sum_{k \in \mathcal{G}(i)} V_k \{G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik}\}\]
\[-M_{ik} = \frac{\partial \Delta Q_i}{\partial \theta_k} = -V_i V_k \{G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik}\}\]
\[-L_{ii} = \frac{\partial \Delta Q_i}{\partial V_i} = \sum_{k \in \mathcal{G}(i)} V_k \{G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik}\} - 2V_i B_{ii}\]
\[-L_{ik} = \frac{\partial \Delta Q_i}{\partial V_k} = V_i \{G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik}\}.\]

Table 3-1: The elements of the Jacobian for the polar formulation.

The linear system (3-5) is usually solved by computing an LU-decomposition of the Jacobian at every iteration. Skillful sparse matrix manipulation and an initial fill-reducing ordering (see [32, 76]) is essential for the efficiency of the Newton method. It is also possible to take advantage of similarities in the computation of the power mismatches and the elements of the Jacobian.

Stott [126] states that the computation time of a Newton iteration is roughly 7 times that of a Gauß-Seidel iteration for the systems (up to 500 busses) he was considering — this ratio is worse for larger systems. Yet Stott cites an example with 500 busses which takes roughly 500 Gauß-Seidel iterations and only 4 Newton iterations — which demonstrates the big speed advantage of the Newton method when solving large problems.

Irving and Sterling [70] show how the Newton-Raphson calculation can be reorganised to simplify the implementation. Their idea is to use the reordered system of equations (2-40)

\[F_{pol} = (\Delta P_1, \Delta Q_1, \Delta P_2, \Delta Q_2, \ldots, \Delta P_n, \Delta Q_n).\]  

Then the Jacobian consists of $2 \times 2$ blocks of the form
3.4 Decoupled Newton methods

Usually numerical methods are most efficient if they take advantage of the physical properties of the underlying problem. As noted in section 2.5.3, the interaction between voltage angles and real power is strong as is that between voltage magnitudes and reactive powers. The coupling between these "P–Q" and "Q–V" components of the problem is relatively weak [126, 129].

The Newton method can be decoupled by neglecting the sub-matrices $M$ and $N$ in the Jacobian in (3-5). Any such approximation of the Jacobian sacrifices the quadratic convergence — in fact, convergence is not guaranteed at all — but may still be faster as less computation is required for every iteration.

To simplify the formulas the notation $\otimes$ and $\oslash$ is introduced. These are the component-wise multiplication and division of two vectors or the multiplication and division of a row in a matrix with the components of a vector. For a complete definition and a list of properties refer to appendix A.

It is advantageous to divide equations $i$ by $V_i$ as this reduces the nonlinearity in the "Q–V" problem. The decoupled version of equation (3-5) is

\[
\begin{align*}
\Delta P \otimes V &= H \otimes V \cdot \Delta \theta \\
\Delta Q \otimes V &= L \otimes V \cdot \Delta V
\end{align*}
\]

Two updating schemes can be used for the decoupled Newton method. In the first both block equations are updated simultaneously; this is a *Jacobi-Newton* method [111, p.220].
CHAPTER 3. LOAD FLOW METHODS

Jacobi-Newton method For \( f(x_1, \ldots, x_q) \in C^2(R^n, R^n) \) with \( n = n_1 + \cdots + n_q, \) \( x_i \in R^{n_i}, \) and \( f \) partitioned so that \( f_i \in C^2(R^n, R^{n_i}), \) define the iteration (for \( k = 0, 1, \ldots \))

\[
x_i^{(k+1)} = x_i^{(k)} - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} f_i(x^{(k)})
\]

for \( i = 1, \ldots, q. \)

The second updating scheme updates the \( q \) equations successively. Define

\[
x^{(k,i)} := (x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k)}, x_{i+1}^{(k)}, \ldots, x_q^{(k)}).
\]

Gauß-Seidel-Newton method For \( f(x_1, \ldots, x_q) \in C^2(R^n, R^n) \) with \( n = n_1 + \cdots + n_q, \) \( x_i \in R^{n_i}, \) and \( f \) partitioned so that \( f_i \in C^2(R^n, R^{n_i}), \) define the iteration (for \( k = 0, 1, \ldots \))

\[
x_i^{(k+1)} = x_i^{(k)} - \left( \frac{\partial f_i}{\partial x_i}(x^{(k,i)}) \right)^{-1} f_i(x^{(k,i)})
\]

for \( i = 1, \ldots, q. \)

The difference between the Gauß-Seidel-Newton and Jacobi-Newton methods is that the Gauß-Seidel-Newton method updates \( \Delta V \) and \( \Delta \theta \) successively, always using the newest available estimates, while the Jacobi-Newton method updates \( \Delta V \) and \( \Delta \theta \) simultaneously.

The Jacobi-type update is well suited for parallel computation, but requires significantly more iterations than the Gauß-Seidel type update. Why this is the case becomes clear when considering the effect of ‘decoupling’ the equations as described by Monticelli et al [100].

In the following discussion reference will be made to the Schur complement of a matrix. When decomposing a matrix into block factors it can be written as [32, p 59ff]

\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
= \begin{pmatrix}
I & A_{11} \\
L_{21} & I
\end{pmatrix}
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
I & U_{12} \\
0 & I
\end{pmatrix}
\]

In the two decompositions, the matrices \( \tilde{A}_{22} \) and \( \tilde{A}_{22} \) are identical and are known as the Schur complement. In general the Schur complement can be written as

\[
\tilde{A}_{22} = A_{22} - A_{21} A_{11}^{-1} A_{12}
\]

For an overview of characterisations of Schur complements and how they are related see Carlson [15].

Compute the Schur complements of the Jacobian in equation (3-5) to obtain

\[
\begin{pmatrix}
\Delta P \\
\Delta Q - M H^{-1} \Delta P
\end{pmatrix}
= \begin{pmatrix}
H & N \\
0 & L - M H^{-1} N
\end{pmatrix}
\begin{pmatrix}
\Delta \theta \\
\Delta V
\end{pmatrix}
\]

(3-11)
and, compute the Schur complement of the Jacobian 'backwards' to obtain

\[
\begin{pmatrix}
\Delta P - NL^{-1} \Delta Q \\
\Delta Q
\end{pmatrix} = \begin{pmatrix}
H - NL^{-1}M & 0 \\
M & L
\end{pmatrix}
\begin{pmatrix}
\Delta \theta \\
\Delta V
\end{pmatrix}.
\] (3-12)

Combining the first and last equations of these two systems leads to the following 'decoupled' system (the matrix in this system of equations is block-diagonal)

\[
\begin{pmatrix}
\Delta P - NL^{-1} \Delta Q \\
\Delta Q - MH^{-1} \Delta P
\end{pmatrix} = \begin{pmatrix}
H - NL^{-1}M & 0 \\
0 & L - MH^{-1}N
\end{pmatrix}
\begin{pmatrix}
\Delta \theta \\
\Delta V
\end{pmatrix}.
\] (3-13)

Consider the first equation in the system (3-11). If \( \Delta V = 0 \) it implies that \( \Delta \theta = H^{-1} \Delta P \), and with \( M = -\partial \theta \Delta Q \) the first two terms of the Taylor expansion for \( \Delta Q \) gives

\[
\Delta Q(\theta + \Delta \theta, V) \approx \Delta Q - MH^{-1} \Delta P.
\] (3-14)

Similarly it follows from the second equation in (3-13) and \( \Delta \theta = 0 \) that

\[
\Delta P(\theta, V + \Delta V) \approx \Delta P - NL^{-1} \Delta Q.
\] (3-15)

If the decoupled Newton method is applied in a Gauß-Seidel fashion, in other words the appropriate power mismatch is recalculated every half iteration, the coupling terms are automatically taken into account. As the coupling between the reactive power mismatch \( \Delta Q \) and \( V \) and between \( \Delta P \) and \( \theta \) is strong, the approximations in the above two equations are good, so that a near exact replication of the full Newton method is obtained without having to calculate the matrices \( N \) and \( M \).

In the decoupled method the factors \( NL^{-1}M \) and \( MH^{-1}N \) are neglected as the assumption is that both \( N \approx 0 \) and \( M \approx 0 \), and these factors are thus very small.

That this approximation to the full Newton-Raphson method is good is borne out by reports that the general convergence of the decoupled method is not much slower than the standard Newton method [126]. The Newton method usually solves the system to a much higher degree of accuracy than required practically, while a similar number of iterations is needed for both methods to reach practical accuracies. This is of course only the case because a good starting values are available in the form of a flat start, or the solution of a DC-load flow.

In section 3.8 the cost of computing the power mismatches and the efficient implementation of the Newton-Raphson method is discussed. It follows from that discussion that not much is gained computationally by using the decoupled Newton method.

### 3.5 The fast decoupled load flow (FDLF) method

The FDLF method first proposed by Stott and Alsaç [127] is currently the most popular method for the solution of load flow problems. A lot of experimentation with decoupled
methods has led to the very efficient FDLF method, but its mathematical properties are only partially understood.

In this section the traditional method, as developed by Stott and Alsaç, is introduced first, and recent modifications to this method, which seems to be able to solve a wider range of problems, are described. For reasons which will become clear later, the original method is called the XB version, while the newer method is called the BX version of the FDLF method. Finally some comments on the derivation and convergence of these methods are made.

Consider the equations for the decoupled Newton method (3-9) again, and especially the analytical form of the elements of $H$ and $L$ as given in table 3-1. When first considering the coupling in power flow equations in section 2.5, it was noted that the assumptions $\cos \theta_{ij} \approx 1$ and $G_{ij} \sin \theta_{ij} \approx 0$ are nearly always valid. The imaginary part $Q_{li}$ of the injected power $S_{li}$ (2-36) is

$$Q_{li} = \sum_{j \in G(i)} V_i V_j \{ G_{ij} \sin \theta_{ij} - B_{ij} \cos \theta_{ij} \}.$$  

If $\theta_{ij} \approx 0$, then $Q_{li} \approx V_i \sum V_j B_{ij}$. If it is furthermore assumed that $V_j = 1$ and any shunt admittances are ignored, then $B_{ii} = \sum B_{ij}$. This can be done safely [127] as the shunt admittances tend to be small compared to $B_{ii}$. Then the main diagonal elements of $H$ and $L$ in equations (3-6) in table 3-1 can be approximated by

$$H_{ii} = \frac{\partial \Delta P_i}{\partial \theta_i} \approx -V_i B_{ii}$$

$$L_{ii} = \frac{\partial \Delta Q_i}{\partial V_i} \approx -V_i B_{ii}.$$  

With the same assumptions it follows that

$$H_{ij} = \frac{\partial \Delta P_i}{\partial \theta_j} \approx -V_i B_{ij}$$

$$L_{ij} = \frac{\partial \Delta Q_i}{\partial V_j} \approx -V_i B_{ij}.$$  

Now (3-9) is transformed to the new system (with $B' = B'' = B$)

$$\Delta P = [V \otimes (B'V)] \cdot \Delta \theta$$

$$\Delta Q = [V \otimes (B''V)] \cdot \Delta V \otimes V$$  

where $[V \otimes (B'V)]$ is the approximation to $-H$ and $[B'' \otimes V]$ the approximation to $-L$. In this form the algorithm produces disappointing results, but by adapting the matrices $B'$ and $B''$ performance can be improved considerably [127]. The changes are

a. Omit all elements which predominantly affect reactive power flows from $B'$, i.e. shunt reactances and off-nominal transformer taps.
3.5. THE FAST DECOUPLED LOAD FLOW (FDLF) METHOD

<table>
<thead>
<tr>
<th>$B^t$</th>
<th>$B^u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line: $b^t_{ik} = \frac{-X_{ik}}{R_{ik}^2 + X_{ik}^2}$</td>
<td>$b^u_{ik} = \frac{-X_{ik}}{R_{ik}^2 + X_{ik}^2}$</td>
</tr>
<tr>
<td>Transformer: $b^t_{ik} = \frac{-X_{ik}}{R_{ik}^2 + X_{ik}^2}$</td>
<td>$b^u_{ik} = \frac{-X_{ik}}{R_{ik}^2 + X_{ik}^2} \cdot \frac{1}{i}$</td>
</tr>
<tr>
<td>Line: $b^{t(k)}<em>{ii} = \frac{X</em>{ik}}{R_{ik}^2 + X_{ik}^2}$</td>
<td>$b^{u(k)}<em>{ii} = \frac{X</em>{ik}}{R_{ik}^2 + X_{ik}^2} + Y_C$</td>
</tr>
<tr>
<td>Transformer: $b^{t(k)}<em>{ii} = \frac{X</em>{ik}}{R_{ik}^2 + X_{ik}^2}$</td>
<td>$b^{u(k)}<em>{ii} = \left{ \begin{array}{ll} \frac{X</em>{ik}}{R_{ik}^2 + X_{ik}^2} + Y_C \cdot \frac{1}{i^2} \ \frac{X_{ik}}{R_{ik}^2 + X_{ik}^2} + Y_C \end{array} \right.$</td>
</tr>
</tbody>
</table>

*If tap on side i.
†If tap on side k.

Table 3-2: Elements to form the matrices $B^t$ and $B^u$. $R_{ik}$ is the resistance and $X_{ik}$ the reactance of the connection $i - k$.

<table>
<thead>
<tr>
<th>$B^t$</th>
<th>$B^u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B^t_{ik} = b^t_{ik}$</td>
<td>$B^u_{ik} = b^u_{ik}$</td>
</tr>
<tr>
<td>$B^t_{ii} = \sum_{k \in G(i)} b^{t(k)}_{ii}$</td>
<td>$B^u_{ii} = \sum_{k \in G(i)} b^{u(k)}<em>{ii} + Y</em>{Gi}$</td>
</tr>
</tbody>
</table>

*If $i$ is swing bus.
†If $i$ is swing or PV bus.

Table 3-3: Formulation of the matrices $B^t$ and $B^u$.

b. Omit the effects of phase shifters from $B^u$.

c. Take the left hand $V$ terms in (3-16a) and (3-16b) to the left hand side.

See tables 3-2 and 3-3 for details. These changes result in the system

$$\Delta F \odot V = B^t \cdot \Delta \theta$$
$$\Delta Q \odot V = B^u \cdot \Delta V$$

(3-17)

A further change is necessary to make this prescription for the FDLF method successful. First consider the elements of $B^t$ and $B^u$ presented in tables 3-2 and 3-3, taken from Bacher's remarks on Amerongen's paper [132]. Impedance is $Z = R + jX$ so that the admittance is $Y = 1/(R + jX) = (R - jX)/(R^2 + X^2) = G + jB$, which implies that $B = -X/(R^2 + X^2)$. In the original method as presented in [127] the resistances are neglected in the $B^t$ matrix, so that $b^t_{ik} = -1/X_{ik}$ for a line contribution, while they are retained in the $B^u$ matrix.
Therefore this scheme is called the XB version of the FDLF. Nanda et al. [101] discuss the
effects of these assumptions.

This method is very efficient, as the matrices $B'$ and $B''$ only have to be constructed
once at the beginning of the load flow analysis. At the same time a $LL^T$-(Cholesky)-
decomposition can be done, so that every iteration consists solely of back substitutions and
re-computations of the power mismatches.

Stott and Alsac [127] propose as iteration procedure the Gauss-Seidel type update, also
used for the decoupled Newton method (see section 3.4). They report experiments with
multiple iterations on either of the equations, but these proved to be slower than the Gauss-
Seidel update. Tests done during this research showed for example that, using a Jacobi-type
update on the IEEE-30 system, with the reactances multiplied by 0.25 to make the system
more ill-conditioned, the Gauss-Seidel type update requires 8 iterations to reduce the error
below $10^{-3}$, while after 50 iterations with a Jacobi-type update the error is still $7 \cdot 10^{-3}$.
After the discussion on the decoupled Newton method in the previous section this is not
surprising.

The advantage of using this formulation of the matrices $B'$ and $B''$, with additional
equations included for the PV busses and the swing bus, is that both matrices have the
same structure as defined by the adjacency matrix of the network; the relationship between
the graph representing a power system and the structure of the matrices is discussed in
detail in section 3.8. Note that this was not true for the matrices $H$ and $L$ introduced in
section 3.3, as these omitted the rows corresponding to the swing bus and in $L$ the those
corresponding to the regulated busses. For sparse matrix computation this means that the
structure of the matrices as well as those of the LU-factors have to be computed once only.
The biggest advantage of this formulation is that both matrices are symmetric, so that the
cheaper Cholesky decomposition can be used instead of LU-decomposition, and that only
half the matrix has to be stored.

For the IEEE 118 bus test system, this method only requires $4\frac{1}{2}$ iterations to reduce the
maximum norm of the power mismatches ($\max_i\{\|\Delta P_i\|_\infty, \|\Delta Q_i\|_\infty\}$) below $10^{-2}$ [127].

Guidelines for handling changes in the system due to e.g. reactive power generation
reaching limits at PV busses are given by Chang and Brandwajn [18]. Re-triangulisation of
$B''$ seems to be favoured if PV busses turn into PQ busses.

All in all, three conditions have to be met for the successful application of FDLF to
power systems [132]:

1. the $R/X$ ratios have to be small ($\sim \frac{1}{3}$) for all branches
2. the voltages must be around their nominal values
3. the angle differences across lines must be small.
3.5. **THE FAST DECOUPLED LOAD FLOW (FDLF) METHOD**

Only in a very small number of cases are the last two conditions a serious factor. Difficulties can be expected when solving systems where \( R/X \) is large; in normal systems \( |R| \approx \frac{1}{3}|X| \), but in some systems, e.g. local distribution networks with underground transmission cables, \( |R| \approx |X| \). The performance of the FDLF method on this type of system is indeed worse.

Rajić and Bose [109] found that the FDLF method usually performs quite well as long as \( X \) does not exceed \( R \) significantly. Experimentally they found some factors that can be added to the elements of \( B' \), and which produce faster convergence in most difficult cases.

Van Amerongen [132] came up with a new scheme for the FDLF algorithm which is more robust for some problems. Instead of neglecting the line resistances in \( B' \), the line resistances in \( B'' \) are neglected. This version is called the BX version. In reference to table 3-3 the two versions of the FDLF algorithm are obtained according to the table below:

<table>
<thead>
<tr>
<th></th>
<th>( B' )</th>
<th>( B'' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>XB</td>
<td>( R_{ij} = 0 )</td>
<td>( R_{ij} \neq 0 )</td>
</tr>
<tr>
<td>BX</td>
<td>( R_{ij} \neq 0 )</td>
<td>( R_{ij} = 0 )</td>
</tr>
</tbody>
</table>

Wu [140] presented the first theoretical discussion of the convergence characteristics of the FDLF. He made plausible that the ratio \( R/X \) in the branches of the system influences convergence strongly. The discussion of the convergence for the Gauß-Seidel update in [140] already hints that the coupling matrices \( N \) and \( M \) are relevant for the discussion of the convergence. A more general discussion of the origin and convergence of the two versions of the FDLF method is given in Monticelli et al. [100] and Hubbi [68].

Consider the system (3-5) and recall the approach used in section 3.4 to motivate why the Gauß-Seidel update must be used for the decoupled Newton method. Applying the same approximations to the matrices \( N \) and \( M \) as to \( H \) and \( L \) gives

\[
\begin{align*}
M_{ii} &\approx -V_i G_{ii} \\
M_{ij} &\approx -V_i G_{ij} \\
N_{ii} &\approx V_i G_{ii} \\
N_{ij} &\approx V_i G_{ij}
\end{align*}
\]

This gives an approximate version of equation (3-5) of the form

\[
\begin{bmatrix}
\Delta P \odot V \\
\Delta Q \odot V
\end{bmatrix}
\approx
\begin{bmatrix}
-B & G \\
-G & -B
\end{bmatrix}
\begin{bmatrix}
\Delta \theta \\
\Delta V
\end{bmatrix}
\tag{3-18}
\]

Equation (3-11) now becomes

\[
\begin{bmatrix}
\Delta P \odot V \\
\Delta Q \odot V - G B^{-1} \Delta P
\end{bmatrix}
= \begin{bmatrix}
-B & G \\
0 & B - G B^{-1} G
\end{bmatrix}
\begin{bmatrix}
\Delta \theta \\
\Delta V
\end{bmatrix}
\tag{3-19}
\]
Monticelli et al. [100] show that this is equivalent to the BX FDLF method described by Van Amerongen [132]. For a 2-bus system the matrices $B$ and $G$ consist of exactly 1 element and

$$b = \frac{-x}{r^2 + x^2} \quad \text{and} \quad g = \frac{r}{r^2 + x^2}.$$ 

For this system it follows immediately that

$$B - GB^{-1}G = \frac{1}{x}$$

which gives the BX version of the FDLF method. Hubbi [68] shows that this is also true for larger systems, i.e. that $(B - GB^{-1}G)_{ij} = 1/x_{ij}$ for $i \neq j$ and $(B - GB^{-1}G)_{ii} = \sum_{j \neq i} 1/x_{ij}$.

The XB version of the FDLF method now follows from (3-12) analogously. For details see Monticelli et al. [100]. They state that the active and reactive parts of the system converge at different speeds and that this is the main reason that the BX method tends to be more successful.

Hubbi [68] builds on the work of [100] and shows theoretically that the BX method is better for high $R/X$ ratios, while the XB method is better if the voltage angles are large ($\theta_i \geq 20^\circ$).

A new method from the theoretical work described above is a FDLF method developed by Haque [60], which attempts to take the coupling matrices into account explicitly. This results in a better approximation of the Jacobian and thus a performance closely related to that of the Newton-Raphson method. Only results for a few systems and no timings are presented though and it is not clear that this method is any faster in practice.

Many versions of the FDLF method have been developed, which perform slightly better or worse under certain conditions, but they do not add much to the discussion above. Keyhani [82] presents a method with reduced memory requirements.

The FDLF method is thus very fast and is cheap to calculate for well-behaved systems. For badly conditioned systems the method may not converge at all in which case the Newton method is used to find a solution.

Another application of the FDLF method would be to use it as a preconditioner for iterative solvers of the linear system in say the Newton-Raphson method. As the FDLF method often solves the system to sufficient accuracy, it seems likely that it would be a very good preconditioner. Although several preconditioners are discussed in chapter 5, the FDLF method is not considered for this purpose. The reason for this is that a different ordering of the equations is considered there, which avoid pivoting.

### 3.6 Second order load flow

The second order load flow (SOLF) method was first introduced by Iwamoto and Tamura [71]. The SOLF method attempts to take the second order terms of the Taylor expansion
3.6. SECOND ORDER LOAD FLOW

of the power mismatch equations (2-39a) and (2-39b) into account. This method merits consideration as comparisons have shown it to be up to five times faster than the Newton-Raphson method for some problems [38, 40]. The second order load flow method has been formulated using polar coordinates [114] as well as rectangular coordinates [71, 40, 39]. The rectangular version of the SOLF method is generally preferred as the second order Taylor expansion of the power mismatch equations is exact (in rectangular coordinates the power mismatch equations are quadratic). The evaluation of the many trigonometric functions is also avoided in rectangular coordinates. In this section only the rectangular version of the second order load flow method is described.

The load flow equations are written as

\[ \Delta S = S_{sp} + V \otimes Y^* \cdot V^* , \]

with \( V \) the complex voltage, and \( Y \) the admittance matrix.

As mentioned previously this complex system cannot be solved with a Newton-type method, as the term \( V \otimes Y \cdot V \) is not differentiable (\( V_i \cdot V_i^* = |V_i|^2 \) is only differentiable at the point \( V_i = 0 \)). Complex differentiability is a very strong assumption though and to find a solution to this system, it is split into its real and imaginary part. With

\[ \Delta S = \Delta P + j \Delta Q \]
\[ Y = G + jB \]
\[ V = e + jf \]

this leads to

\[
\begin{align*}
\Delta P &= P_{sp} + e \otimes (Ge - Bf) + f \otimes (Be + Gf) \\
\Delta Q &= Q_{sp} + e \otimes (Ge - Bf) - e \otimes (Be + Gf)
\end{align*}
\]

(3-20)

For the development of the second order load flow method, the Taylor expansion of the functions (3-20) is needed. As these are quadratic functions in rectangular coordinates, the first three terms of the Taylor series give an exact representation. To simplify the calculation set \( P_{sp} = Q_{sp} = 0 \) initially. Let \((e_0, f_0)\) be an initial value for the voltages and \((e, f)\) the values at which \( \Delta P \) and \( \Delta Q \) are evaluated. Define \( \Delta e := e - e_0 \) and \( \Delta f := f - f_0 \). Then

\[
\begin{pmatrix} \Delta P \\ \Delta Q \end{pmatrix} (e, f) = \begin{pmatrix} \Delta P \\ \Delta Q \end{pmatrix} (e_0, f_0) + \begin{pmatrix} \partial_e \Delta P & \partial_f \Delta P \\ \partial_e \Delta Q & \partial_f \Delta Q \end{pmatrix} (e_0, f_0) \begin{pmatrix} \Delta e \\ \Delta f \end{pmatrix} + \frac{1}{2} D^2 \begin{pmatrix} \Delta P \\ \Delta Q \end{pmatrix} (\Delta e, \Delta f)
\]

(3-21)

With the differentiation rules given in lemma A.3 in appendix A, the components of the Jacobian are easily computed as

\[
\begin{align*}
\partial_e \Delta P &= e \otimes G + Ge \otimes I - Bf \otimes I + f \otimes B \\
\partial_f \Delta P &= -e \otimes B + Be \otimes I + Gf \otimes I + f \otimes G \\
\partial_e \Delta Q &= -e \otimes B - Be \otimes I - Gf \otimes I + f \otimes G \\
\partial_f \Delta Q &= -e \otimes G + Ge \otimes I - Bf \otimes I - f \otimes B
\end{align*}
\]

(3-22)
At this stage a small computation is required. Substitute \((e, f) = (e_0 + \Delta e, f_0 + \Delta f)\) into equation (3-20) to obtain

\[
\Delta P(e_0 + \Delta e, f_0 + \Delta f) = (e_0 + \Delta e) \otimes (G(e_0 + \Delta e) - B(f_0 + \Delta f)) + (f_0 + \Delta f) \otimes (B(e_0 + \Delta e) + G(f_0 + \Delta f))
\]

\[
e_0 \otimes G e_0 - e_0 \otimes B f_0 + f_0 \otimes B e_0 + f_0 \otimes G f_0 +
\]

\[
e_0 \otimes G \Delta e + \Delta e \otimes G e_0 - \Delta e \otimes B f_0 + f_0 \otimes B \Delta e -
\]

\[
e_0 \otimes B \Delta f + \Delta f \otimes B e_0 + \Delta f \otimes G f_0 + f_0 \otimes G \Delta f +
\]

\[
\Delta e \otimes G \Delta e - \Delta e \otimes B \Delta f + \Delta f \otimes B \Delta e + \Delta f \otimes G \Delta f
\]

\[
= \Delta P(e_0, f_0) + \partial_e \Delta P(e_0, f_0) \cdot \Delta e + \partial_f \Delta P(e_0, f_0) \cdot \Delta f +
\]

\[
\Delta P(\Delta e, \Delta f)
\]

Similarly

\[
\Delta Q(e_0 + \Delta e, f_0 + \Delta f) =
\]

\[
\Delta Q(e_0, f_0) + \partial_e \Delta Q(e_0, f_0) \cdot \Delta e + \partial_f \Delta Q(e_0, f_0) \cdot \Delta f + \Delta Q(\Delta e, \Delta f)
\]

(3-24)

Comparing equations (3-23) and (3-24) with (3-21) and taking account of the fact that the Taylor series expansion is an exact representation of (3-20), it is clear that

\[
\frac{1}{2} P^2 \left( \frac{\Delta P}{\Delta Q} \right)(\Delta e, \Delta f) = \left( \frac{\Delta P}{\Delta Q} \right)(\Delta e, \Delta f).
\]

(3-25)

The same result can be obtained by calculating the second derivative explicitly. This is a special case of a mismatch theorem introduced by Hubbi [67].

Now the specified load and generation is reintroduced into these equations. A factor \((P_{sp}, Q_{sp})\) is thus present in the first and the last terms of equations (3-23) and (3-24). The equations must be corrected for this constant term, and an additional factor \(-(P_{sp}, Q_{sp})\) is introduced.

At the solution \(\Delta P(e, f) = \Delta Q(e, f) = 0\) holds. Setting equation (3-21) to 0 results in

\[
\begin{pmatrix}
0 \\
0
\end{pmatrix}
= \begin{pmatrix}
\Delta P \\
\Delta Q
\end{pmatrix}(e, f)
\]

\[
= \begin{pmatrix}
\Delta P \\
\Delta Q
\end{pmatrix}(e_0, f_0) - \begin{pmatrix}
P_{sp} \\
Q_{sp}
\end{pmatrix} + \begin{pmatrix}
\partial_e \Delta P & \partial_f \Delta P \\
\partial_e \Delta Q & \partial_f \Delta Q
\end{pmatrix} \begin{pmatrix}
\Delta e \\
\Delta f
\end{pmatrix}
\]

(3-26)

\[
+ \begin{pmatrix}
\Delta P \\
\Delta Q
\end{pmatrix}(\Delta e, \Delta f).
\]
3.6. SECOND ORDER LOAD FLOW

The Jacobian in this equation is evaluated at \((e_0, f_0)\). This equation is still quadratic in \((\Delta e, \Delta f)\), so that these values cannot be computed directly. Iwamoto & Tamura [71] rewrite (3-26) as the iterative method

\[
\begin{pmatrix}
\Delta e^{(k+1)} \\
\Delta f^{(k+1)}
\end{pmatrix} = 
- \left( \begin{array}{cc}
\partial_e \Delta P & \partial_f \Delta P \\
\partial_e \Delta Q & \partial_f \Delta Q
\end{array} \right)^{-1} \left\{ \left( \begin{array}{c}
\Delta P_0 - P_{sp} \\
\Delta Q_0 - Q_{sp}
\end{array} \right) + \left( \begin{array}{c}
\Delta Q \\
\Delta Q
\end{array} \right) (\Delta e^{(k)}, \Delta f^{(k)}) \right\}. \tag{3-27}
\]

In the first step \((\Delta e^{(0)}, \Delta f^{(0)}) = (0, 0)\), so that this is simply the first step in the Newton-Raphson method.

Iwamoto and Tamura [71] interpret the last term in (3-26) as a correction to the Newton-Raphson method, and see (3-27) as successive applications of a corrected Newton-Raphson method. As (3-26) is exact no new evaluation of the Jacobian is needed, and (3-27) is an algorithm to solve the systems of nonlinear equations (3-26) for \((\Delta e, \Delta f)\). Any other method for the solution of this system of nonlinear equations could also be used. A more detailed discussion of this iterative process is given at the end of this section.

3.6.1 Handling PV busses

If rectangular coordinates are used it is more difficult to include the special case of PV-busses in the calculations. To include a PV bus in the calculations the equation for \(\Delta Q\) for that bus is dropped from (3-21). The question now is which of the variables \(e\) or \(f\) should be dropped. In the paragraph on interaction between different variables on page 33 it was noted that, as long as the voltage angle was small, there is quite a large correlation between \(e\) and \(|V|\). Yet the voltage angle is not always small! Thus an additional equation is added to ensure that the voltage magnitude remains constant [71, 39],

\[\Delta E = e^2 + f^2 - |E_{sp}|^2\]

To fit this equation into the framework of equation (3-26) note that

\[
\Delta E(e, f) = e^2 + f^2 - |E_{sp}|^2 \\
= (e_0 + \Delta e)^2 + (f_0 + \Delta f)^2 - |E_{sp}|^2 \\
= \Delta E(e_0, f_0) + |E_{sp}|^2 + (2e_0, 2f_0) \cdot \begin{pmatrix} \Delta e \\ \Delta f \end{pmatrix} + \Delta E(\Delta e, \Delta f). \tag{3-28}
\]

Incorporating (3-28) into (3-27) for the \(N_G\) generation busses leads to the standard formulation of the SOLF method

\[
\begin{pmatrix}
\Delta e^{(k+1)} \\
\Delta f^{(k+1)}
\end{pmatrix} = 
- \left( \begin{array}{cc}
\partial_e \Delta P & \partial_f \Delta P \\
\partial_e \Delta Q & \partial_f \Delta Q
\end{array} \right)^{-1} \left\{ \left( \begin{array}{c}
\Delta P_0 - P_{sp} \\
\Delta Q_0 - Q_{sp}
\end{array} \right) + \left( \begin{array}{c}
\Delta Q \\
\Delta Q
\end{array} \right) (\Delta e^{(k)}, \Delta f^{(k)}) \right\}. \tag{3-27}
\]
\[
\begin{pmatrix}
\partial_e \Delta P & \partial_f \Delta P \\
\partial_e \Delta Q & \partial_f \Delta Q \\
\partial_e \Delta E & \partial_f \Delta E
\end{pmatrix}
\begin{pmatrix}
\Delta P_0 - P_{sp} \\
\Delta Q_0 - Q_{sp} \\
\Delta E_0 + |E_{sp}|^2
\end{pmatrix}
+ \begin{pmatrix}
\Delta P \\
\Delta Q \\
\Delta E
\end{pmatrix}
\begin{pmatrix}
\Delta e^{(k)}_e \\
\Delta f^{(k)}_f
\end{pmatrix} = 0.
\]

The Jacobian in this system is a square matrix as there is exactly one power generation bus equation in the last block row for every reactive power mismatch equation dropped from the second block row. Often the solution process is started with a flat start, so that \(e_i = 1\) and \(f_i = 0\). From this it follows immediately that
\[
\partial_e \Delta E = 2I \quad \text{and} \quad \partial_f \Delta E = 0.
\]

At PV busses the voltage magnitude is specified as \(|E_{sp,i}|\) while \(|E_{sp,j}| = 0\) for all other nodes. For the \(N_G\) generating nodes it follows that \([41]\)
\[
\Delta e^{(k+1)}_e = \frac{1}{2} \left\{ |E_{sp}|^2 - |E_{q}|^2 - |\Delta e^{(k)}_e|^2 \right\}.
\]

By eliminating the columns corresponding to the generating busses from \(\partial_e \Delta P\) and \(\partial_e \Delta Q\) to form the reduced matrices \(\partial_e \Delta P\) and \(\partial_e \Delta Q\) respectively, the SOLF method becomes \([39]\)
\[
\begin{pmatrix}
\Delta e^{(k+1)} \\
\Delta f^{(k+1)}
\end{pmatrix} = -\begin{pmatrix}
\partial_e \Delta P & \partial_f \Delta P \\
\partial_e \Delta Q & \partial_f \Delta Q
\end{pmatrix}^{-1}
\begin{pmatrix}
\Delta P_0 - P_{sp} \\
\Delta Q_0 - Q_{sp}
\end{pmatrix}
+ \begin{pmatrix}
\Delta P \\
\Delta Q
\end{pmatrix}
\begin{pmatrix}
\Delta e^{(k)}_e \\
\Delta f^{(k)}_f
\end{pmatrix}
+ \begin{pmatrix}
\sum_{N_G} \partial_{e_{gj}} \Delta P_j \Delta e^{(k+1)}_g \\
\sum_{N_G} \partial_{e_{gj}} \Delta Q_j \Delta e^{(k+1)}_g
\end{pmatrix}.
\]

The values \(\Delta e_{gj}\) have been dropped from \(\Delta e\) to form \(\Delta e\). This is the formulation usually used in practice.

Ekwele and Adams \([40]\) report that an improved version of this algorithm is obtained if the starting conditions are obtained from a DC load flow analysis.

### 3.6.2 Interpretation of the second order load flow method.

Hubbi \([67]\) has shown that the second order load flow method, as defined by Iwamoto and Tamura \([71]\), is really a fixed Jacobi (FJM) method or simplified Newton method. Simplified Newton methods only converge under restricted conditions, see e.g. Rheinboldt and Ortega \([111, \text{p.421}]\). To simplify the discussion define the function \(F(x) = (\Delta P, \Delta Q)^T(\epsilon, \delta)\).

In order to show the correspondence between the simplified Newton method and the second order load flow method Hubbi examines the algorithm proposed in (3-27) for a quadratic function \(F\) with Jacobian \(J\) and a second order term \(H(x_0, \Delta x)\). \(x_0\) is the initial value and \(x_n\) the solution. Then the Taylor-expansion around a point \(x_0\) gives
\[
F(x) = F(x_0) + J_0 \cdot \Delta x + H(x_0, \Delta x)
\]  

(3-31)
and the iteration (3-27) is written as

\[ \Delta x_{k+1} = -J_0^{-1}(F(x_0) + H(x_0, \Delta x_k)) \]  

(3-32)

where \( \Delta x_k = x_k - x_0 \). In order to show that this is indeed a simplified Newton method, the following result is needed.

**Lemma 3.1** For the iterative sequence (3-32) it holds that

\[ H(x_0, \Delta x_k) = \sum_{i=1}^{k} F(x_i) \]

**Proof:** \( k = 1 \): The first step in the process is a Newton-Raphson step so that

\[ 0 = F(x_0) + J_0 \cdot \Delta x_1 \]

Consequently it follows from (3-31) that

\[ F(x_1) = F(x_0 + \Delta x_1) = H(x_0, \Delta x_1) \]

\( k - 1 \to k \): In the kth step of the SOLF method the equation

\[ 0 = F(x_0) + J_0 \cdot \Delta x_k + H(x_0, \Delta x_{k-1}) \]

is solved. Subtracting this equation from the expansion (3-31) for \( x_k \)

\[ F(x_k) = F(x_0) + J_0 \cdot \Delta x_k + H(x_0, \Delta x_k) \]

gives

\[ F(x_k) = H(x_0, \Delta x_k) - H(x_0, \Delta x_{k-1}) \].

Figure 3-1: The second order load flow method as defined by [71].
With the induction supposition that $\sum_{i=1}^{k-1} F(x_i) = H(x_0, \Delta x_{k-1})$ it then follows immediately that

$$\sum_{i=1}^{k} F(x_i) = H(x_0, \Delta x_k).$$

This is exactly the result obtained in equation (3-25). Step $k$ of the SOLF method (3-32) is thus equivalent to solving

$$0 = \sum_{i=0}^{k} F(x_i) + J_0 \cdot \Delta x_{k+1}$$

(3-33)

and is shown in figure 3-1. To see that this is indeed the fixed Jacobi method set $\delta x_k = x_k - x_{k-1}$ so that

$$\begin{align*}
\delta x_k &= \Delta x_k - \Delta x_{k-1} \\
&= J_0^{-1} \left( \sum_{i=0}^{k-1} F_i - \sum_{i=0}^{k-2} F_i \right) \\
&= J_0^{-1} F(x_{k-1}).
\end{align*}$$

This is just the simplified Newton method and its convergence characteristics are described in detail in Rheinboldt and Ortega [111, p. 421]. The criteria for the convergence of the simplified Newton method are sufficient for the Newton method, i.e. the Newton method will always converge if the simplified Newton method converges. The convergence of the simplified Newton method depends on how well $J_0$ approximates all Jacobians in the iteration sequence, but it normally does not converge faster than the Newton-Raphson method.

### 3.6.3 Modifications of the SOLF method

Many modifications of the basic SOLF method have been developed. A summary and comparison of seven methods can be found in [20]. Most of the methods centre on finding simpler ways of solving the equation (3-30). Rao et al. [110] and Roy [112] replace the Jacobian with a symmetric matrix. In both cases the symmetric matrices are derived from the assumption that the process is started with a flat start, which simplifies the Jacobian (3-22) significantly. Hubbi [67] and Cory et al. [20] report that these methods are not competitive in practice.

Other methods attempt to speed up the solution of (3-30) by making use of the techniques employed in the decoupled Newton and FDLF methods. To this end define

$$\begin{pmatrix}
\Delta P' \\
\Delta Q'
\end{pmatrix} = \begin{pmatrix}
\Delta P_0 - P_{sp} + \Delta P \Delta e + \sum_{G}^{N_G} \delta_{e_{G}} \Delta P_{G} \Delta e_{g}^{(k+1)} \\
\Delta Q_0 - Q_{sp} + \Delta Q \Delta f + \sum_{G}^{N_G} \delta_{f_{G}} \Delta Q_{G} \Delta e_{g}^{(k+1)}
\end{pmatrix}$$
3.7. CHOICE OF METHOD

The decoupled Newton method decouples the real and imaginary mismatch equations to obtain the iteration sequence

\[ \Delta P' - \partial_e \Delta P \Delta e^{(k)} = \partial_f \Delta P \Delta f^{(k+\frac{1}{2})} \]
\[ \Delta Q' - \partial_f \Delta Q \Delta f^{(k+\frac{1}{2})} = \partial_e \Delta Q \Delta e^{(k+1)} \]

The FDLoF-SOF method has the same computational scheme but simplifies the block-matrices in the Jacobian as discussed in section 3.5. Several other variants are also possible.

All these methods rely on a constant matrix iteration. As the second-order term is always calculated at the previous iterate these are not true second-order methods but are variants of parallel-chord methods [111] with different choices for the iteration matrices. For their speed they thus rely on having a starting value well inside the domain of attraction of the matrix operator.

3.7 Choice of method

In the previous sections the development of a whole range of methods was discussed. Other methods are for example the minimum mismatch method, which uses a hybrid method developed by Powell for the solution of the nonlinear systems of equations [87]. Optimisation methods have also been used for the solution of these systems [75, 66].

Other possibilities for the solution of nonlinear systems, are methods that do not compute the Jacobian explicitly, but use some approximation to it, which is refined in successive iterations. Curtis, Powell and Reid [22] developed a method that calculates an estimate to a sparse Jacobian with a single function evaluation, by only updating a few columns at a time. This is possible by selecting a set of columns in every iteration that do not, or hardly, have nonzero elements in the same rows. The problem is then to find a suitable partitioning of all the columns.

A different idea is to use quasi-Newton method for sparse matrices. Schubert [121] developed such a method. Li [92] combines elements of these two ideas to define an algorithm that successively updates columns of the Jacobian in every iteration, and shows that his algorithm is competitive with the algorithm developed by Curtis, Powell and Reid, as well as Schubert’s algorithm. Li’s algorithm could be used to refine an initial estimate.

The reason these ideas are not further considered are two-fold: firstly, load flow algorithms tend to converge in very few iterations as good starting values are available, which would allow these methods very little time to adapt the estimate of the Jacobian if they are to be competitive; and secondly, re-using subexpressions from the calculation of the power mismatches in the calculation of the Jacobian means that the Jacobian can be calculated very cheaply.
The industry standard for load flow analysis is currently the FDLF method or one of its variants. Some comparisons have shown the SOLF method to be more robust than the FDLF method, while not being much slower. There is general agreement that the most robust method is the Newton-Raphson method, but that this is also the most expensive.

Comparisons of load flow methods can be found in many articles [20, 99, 72]. The only conclusion that can be drawn from these comparisons is that the performance of the different methods is highly problem and implementation dependent. Many comparisons are done on the IEEE test systems, which are much smaller than the systems that are simulated today. This way a benchmark is provided, but nowadays this benchmark has little to do with the problems that are solved in practice.

That the performance of these methods is inconsistent on different problems is not surprising: Firstly, different power systems have very different characteristics and, secondly, the nonlinear systems of equations are usually solved in very few iterations, so that the choice of starting values is crucial. Classical convergence analysis is of limited use in the study of such problems. For example: on many problems a fixed-Jacobi type method, such as SOLF, will perform better than the Newton-Raphson method itself, as the initial Jacobian is good enough for the method to progress to a solution while the calculations of the Jacobians are saved.

Most methods attempt to find a fixed approximation to the Jacobian which allows the resulting parallel chord method to converge in a large number of cases. One interesting development in this area are so-called distributed slack bus methods [58], which distribute the slack bus so that there are no large changes in the power injected into the slack bus. The result is that the changes in the Jacobian are smaller, so that a fixed approximation to the Jacobian is more likely to solve a wider range of problems.

If a problem is badly conditioned, in other words if the R/X ratios are high, the system is heavily loaded, the system comprises high and low voltage parts and so forth, all these methods will fail for some problems. If the systems are large it is also less likely that a fixed approximation to the Jacobian is sufficient as the initial values, for which the fixed Jacobian is calculated, may be quite far from the solution in some parts of the system. Although some methods (e.g. SOLF) seem to approximate the Newton-Raphson method very well, closer scrutiny reveals that this is not the case. One always pays a price when using a cheaper method.

The reasons for developing all these specialised methods are two-fold: speed and memory requirements. The question is whether these reasons are still valid with modern computer technology and modern sparse matrix algorithms. After some experiments with a carefully implemented Newton-Raphson method it was concluded that this is not the case. The reasons are the following:
3.7. CHOICE OF METHOD

- Processors are getting faster: the ieee118 test system solved with the Newton Raphson method is solved in 309 ms on a DEC 5000/240, 142 ms on an Intel i860 and 101 ms on a (by now out of date) Alpha 3000/300L. A test system with 1419 busses (see section 2.7) is solved in around a second on both the i860 and an Alpha. The largest chunk of time in a calculation is often spent on input and output, not on the calculation.

- Processor memory is not an issue: high-end work stations are delivered with up to a GByte of memory.

- Sparse matrix technology has improved and storing a Jacobian in sparse data structures, and doing a sparse LU decomposition is not a problem.

- Load flow analysis is more frequently applied to lower voltage networks and other systems in which relatively high branch resistance to reactance ratios occur. The decoupled algorithms may then fail to converge [70], which would cause most programs to switch to a Newton method.

- Larger systems are simulated which do contain low and high voltage parts. Such simulations require general-purpose algorithms.

- Using $2 \times 2$ pivots (see below) makes the LU-decomposition of the Jacobian in the Newton method stable even if combinations of capacitive and inductive impedances create relatively small diagonal elements.

- A carefully implemented Newton-Raphson method which exploits sparsity is much more efficient than previously thought. The calculation of the Jacobian does not turn out to be very expensive. It should also be kept in mind that methods such as FDLP require two function evaluations per full iteration.

- By using a $2 \times 2$ block representation for the Jacobian, it is easy to accommodate changes of PV busses into PQ busses and vice versa. In methods using constant iteration matrices, such changes require the recalculation of the iteration matrices.

- Parallel computers provide ways of increasing the speed of some methods. As the discussions in the previous sections show, the decoupling of the real and reactive equations requires the two resulting systems to be solved successively: this limits the scope for parallelisation as the solution of either subsystem does not contain much calculation.

Consider the following profile on a DECstation 5000/240. The operations were executed 200 times and the average time was taken. The sparse systems were reordered with the minimum degree algorithm to reduce fill.
The LU-decomposition is the most expensive, but the function evaluations add up to more work than the forward and back substitutions, and the calculation of the Jacobian takes roughly as long as a forward and back substitution. After the initial decomposition the FDFF method requires two function evaluations and two system solutions per iteration, i.e. roughly 6 and 59 milliseconds for each of the two examples, while the Newton-Raphson method requires 5.73 and 60 milliseconds. As the Newton-Raphson method usually requires fewer iterations than other methods these figures seem competitive. Considering that the minimum degree reordering of the admittance matrix of the whole system requires 0.648 seconds for pow795, the discussion of different methods is somewhat irrelevant.

The only one of the classical methods that is considered for parallel implementation is the Newton-Raphson method. It is fast enough on modern sequential computers for many applications and can be parallelised, as will be shown in this thesis, for those applications requiring even greater speed. The virtue of using the Newton-Raphson method is its generality and its quadratic convergence.

### 3.8 Implementation of the Newton-Raphson method

An efficient implementation of a Newton-Raphson method for the solution of the load flow problem is discussed in this section. Only a sequential implementation is described, but this method is used as benchmark for all parallel methods developed later. As parallel algorithms only benefit the user if they are faster than the very best sequential algorithm, a lot of effort was invested in making this implementation as efficient as possible. Throughout the calculation sparse data structures are used, which are discussed in section 3.8.1 and calculations are always reordered with vectorisation in mind. The parallel architecture on which most of the work in this thesis was carried out is an Intel iPSC/860, and on the pipelined i860 processors good vectorisation has a significant impact on the execution time.

An iteration of the standard Newton method is formulated as follows for $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $F$ a $C^1$ function, and a starting value $x_0 \in \mathbb{R}^n$ [26, 111]:

$$x_{k+1} = x_k - J(x_k)^{-1}F(x_k).$$  \hspace{1cm} (3-34)

Here $J(x_k)$ is the Jacobian of $F$ evaluated at $x_k$.

When solving (3-34) three steps are needed:
3.8. IMPLEMENTATION OF THE NEWTON-RAPHSON METHOD

\[
A = \begin{pmatrix}
  \vdots \\
  \vdots \\
  \vdots \\
\end{pmatrix}
\]

\[
\begin{array}{cccccc}
\text{Row} & 1 & 4 & 7 & 9 & 11 \\
\text{Col} & 1 & 2 & 4 & 1 & 2 & 3 & 2 & 3 & 1 & 4 \\
A & a_{11} & a_{12} & a_{14} & a_{21} & a_{22} & a_{23} & a_{32} & a_{33} & a_{41} & a_{44} \\
\end{array}
\]

Figure 3-2: A sparse matrix and its compressed storage.

1. \( F(x_k) \) is computed
2. \( J(x_k) \) is computed
3. The linear system \( J(x_k) \cdot \Delta x_k = -F(x_k) \) is solved

To ensure that the Newton method is globally convergent, a line-search should be done as a fourth step; line searches are discussed in more detail in chapter 10. As a good starting value is available for load flow methods, line-searches were never required for any of the data-sets considered.

The computation of \( F(x_k) \) and \( J(x_k) \) is the subject of section 3.8.3. As the calculation of the function value can require up to a third of the computation time per iteration the efficiency of this calculation is an important issue. The computation of the Jacobian is usually even more computationally intensive, but judicious re-use of terms from the calculation of the power mismatches reduces the number of required calculations significantly.

The linear systems are solved with LU-decomposition and forward/back substitution. Iterative methods for the solution of these systems of linear equations are considered in part II of this thesis, which also considers parallel implementations of both direct and iterative methods.

Finally the methodology for handling regulated buses is discussed in section 3.8.5 and results for the test systems are presented in section 3.8.7.

3.8.1 Sparse data structures for load flow analysis

Throughout the calculation sparse data structures are used, as explained in for example Duff et al. [32] and George and Liu [52]. As sparse data structures are essential for efficiency, but are quite complex (especially in parallel algorithms), they are discussed in some detail.

The storage mechanism used here is sparse row storage, in other words nonzero elements in the same row of a matrix are stored consecutively in the compressed data structures. To explain how this storage mechanism works, consider the matrix \( A \) in figure 3-2. The array Col contains the column numbers for every nonzero element in the matrix, while the array Row contains the indices in Col where new rows start. For example, Row(2)=4 states that
the first nonzero element in row 2 of A is in column Col(4)=1. For administrative reasons
the (n+1)th element of Row contains the first index after the end of Col, i.e. the position
where the (n+1)th row would start if there was one. The values of the nonzero elements
of A are stored in an array A with the same length as Col.

As an example, calculating a matrix-vector multiplication \( b = Ax \) is very simple with
this storage scheme:

\[
\text{DO } \text{irow} = 1, N \\
\quad b(\text{irow}) = 0 \\
\text{DO } \text{icol} = \text{Row(irow)}, \text{Row(irow+1)} - 1 \\
\quad b(\text{irow}) = b(\text{irow}) + A(\text{icol}) \times (\text{Col(icol)}) \\
\quad \text{END DO} \\
\text{END DO}
\]

Another attractive feature is that, if the same operation must be carried out on every
element of a matrix, or if two matrices with the same storage scheme are added to each
other, the structural information is not needed. For example \( C = A + B \) is executed by

\[
\text{DO } i = 1, \text{Row(N+1)} - 1 \\
\quad C(i) = A(i) + B(i) \\
\text{END DO}
\]

These are \textit{fully vectorisable} operations, which makes them very fast indeed.

3.8.2 The admittance matrix and sparse data structures

An explanation of how sparsity occurs in load flow analysis is still required. Consider the
IEEE-14 power system in figure 2-3 on page 17. A graph detailing the structure of the
power system is shown in figure 2-4 and repeated in figure 3-3. The graph and the structure
of the sparse matrix are isomorphic [61], and this is widely exploited in sparse matrix
computations.

Consider the equation defining the power mismatches in node \( i \) of the system as given
in equation (2-37):

\[
\Delta S_i = S_{Gi} - S_{Di} + V_i \sum_{j \in \partial \langle i \rangle} Y_{ij}^* V_j^* . \quad (3-35)
\]

For node 7 in the IEEE-14 test system the calculation of the power mismatch is

\[
\Delta S_7 = S_{G7} - S_{D7} + V_7 (Y_{74}^* V_4^* + Y_{77}^* V_7^* + Y_{78}^* V_8^* + Y_{79}^* V_9^*)
\]
with \( Y_{77} \) the self-admittance of bus 7. Every nonzero admittance \( Y_{7k}, k \neq 7 \), is represented by an edge from node 7 to node \( k \) in the graph of the power system. The structure of the admittance matrix for the IEEE-14 test system and the associated graph are shown in figure 3-3. This demonstrates how the graph structure is reflected in the nonzero structure of the matrix: nonzero off-diagonal elements correspond to edges in the graph. The matrix has symmetric structure, as a power input for one node is an output for another.

The mismatch for all the nodes in the system is calculated as

\[
\Delta \tilde{S} = \Delta \tilde{S}_G + \Delta \tilde{S}_D + \tilde{V} \otimes Y^* \tilde{V}^* \tag{3-36}
\]

where \( Y^* \tilde{V}^* \) is a sparse matrix-vector multiplication. For the test system pow1419 a dense representation would require storage for \( 1419^2 = 2013561 \) elements, while the sparse representation only requires \( 1420 + 5763 = 7183 \) elements for the storage of the structural information and 5763 elements for the actual values — a significant saving.

During the Newton-Raphson solution of the load flow equations the Jacobian of the function \( \Delta \tilde{S} = \Delta \tilde{F} + i \Delta \tilde{Q} \), split into real and imaginary parts, must be calculated. There are four nonzero elements in the (real) Jacobian for every element in the (complex) admittance matrix \( Y \), as defined in (3-8) on page 47.

This structure can be calculated efficiently from the structural information of the admittance matrix. When calculating the Jacobian’s row information note that the length of both rows \( 2i - 1 \) and \( 2i \) is \( 2 \times (\text{Row}(i+1) - \text{Row}(i)) \). The new column indices are calculated by noting that an entry in column \( k \) in the admittance matrix gives rise to entries in columns \( 2k - 1 \) and \( 2k \) in the Jacobian.
CHAPTER 3. LOAD FLOW METHODS

If LU-decomposition is used during the solution of the system of linear equations, it is necessary to predetermine the space that is needed for new nonzero elements (fill-in) that appear in the matrix during the elimination process.

Irving & Sterling [70] report that, provided that only $2 \times 2$ blocks are used during the decomposition, the decomposition is always stable. If single pivots are used pivoting is required in practically all cases. Fill-in can thus be determined from the admittance matrix and then expanded to $2 \times 2$ blocks, before doing the decomposition.

In the implementation discussed here the structure of the Jacobian (including fill-in) is computed at the start of the calculation. Additionally an array with the indices of the diagonal elements in the compressed array is computed, as this simplifies the LU-decomposition. The structure of the calculation is only computed once as all the Jacobians have the same structure before and after elimination.

The amount of fill-in changes with the ordering of the matrix. The ordering of an admittance matrix reflects the numbering of the mismatch equations, in other words the numbering of the nodes in the power system. It is important to minimise fill-in to reduce storage, computation time and the precision of the calculation (fewer rounding errors). The reordering algorithm used in this sequential implementation is the minimum degree algorithm. For a description of this well-known algorithm see for example Duff et al. [32], or for a variant developed for circuit simulation see Berry [6]. The admittance matrix is reordered before the structure of the Jacobian is computed, so that the numbering of equations in both cases is the same. It is also much cheaper to reorder the admittance matrix instead of the Jacobian, as it has only half the dimension of the Jacobian.

3.8.3 Calculation of power mismatches and the Jacobian

In load flow analysis the computation of the function $F$ is expensive, taking up to a third of the computation time in every iteration. For this reason unnecessary function evaluations must be avoided. Fortunately the mismatch functions are well-behaved enough that line searches can be avoided and a full Newton step can be taken in every iteration.

Often one of the most expensive parts of a Newton method is the evaluation of the Jacobian. Fortunately the most expensive factors in both the function evaluation (equations (2-39)) and the computation of the Jacobian (table 3-1) are identical, these being the calculation of

$$G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik} \quad \text{and} \quad G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik}. \tag{3.37}$$

The function evaluation proceeds as follows:

1. Compute the angle differences $\theta_{ik} = \theta_i - \theta_k$ for all nonzero admittances
2. Compute \( \sin \theta_{ik} \) and \( \cos \theta_{ik} \) in two vector statements

3. Compute the expressions \( (G_{ik} \cos \theta_{ik} + B_{ik} \sin \theta_{ik}) \) and \( (G_{ik} \sin \theta_{ik} - B_{ik} \cos \theta_{ik}) \)—again with vector statements

4. Compute the function values \( \Delta P_i \) and \( \Delta Q_i \)

The angle differences are computed in a double loop similar to the matrix-vector multiplication discussed earlier, the outer loop determining the index of \( i \) and the inner loop the index \( k \). The angle differences are then stored in an array with the same size as the compressed admittance matrix. This is followed by the computation of the sines and cosines of the angle differences which are stored in a complex array \( \text{Tmp} \) as \( \sin \theta_{ik} + j \cos \theta_{ik} \). As this array and the compressed admittance matrix have the same size a complex vector-vector multiplication with the conjugate of the admittance matrix produces both terms in equation (3-37).

The function values can then be computed with a complex/real matrix vector multiplication, where every row is simultaneously multiplied with the voltage. If the temporary values are stored in an array \( \text{Tmp} \), the calculation proceeds as follows (\( PQ \) is the array with the complex power mismatches).

\[
\text{DO } \text{irow} = 1, N \\
\quad \text{DO } \text{icol} = \text{Row(irow)}, \text{Row(irow+1)}-1 \\
\quad \quad \text{PQ(irow)} = \text{PQ(irow)} + \text{Tmp(icol)}*V(\text{Col(icol)}) \\
\quad \text{END DO} \\
\quad \text{PQ(irow)} = \text{PQ(irow)}*V(\text{irow}) \\
\text{END DO}
\]

Different methodologies for the computation of sparse matrix-vector multiplications are investigated in appendix C, as these techniques are fundamental for several of these calculations. The tests show that at least the Portland Group compiler on the Intel iPSC/860 does a good job of vectorising the above double-loop and that other techniques do not offer any advantage.

The values stored in the temporary array \( \text{Tmp} \) can be re-used when calculating the Jacobian. Typically the calculation of the Jacobian requires less than half the time needed for the calculation of the function values.

### 3.8.4 Sparse LU-decomposition

Parallel sparse LU decomposition is discussed in more detail in the second part of this thesis. Nevertheless a few remarks on the sequential implementation are required here, as the LU-decomposition is one of the most compute intensive tasks during load flow analysis.
As mentioned earlier, the LU decomposition is always stable if it is done with $2 \times 2$ block-pivots instead of single pivots. Recall the definition of the elements of the Jacobian in table 3-1 on page 46. Usually $\cos \theta_{ik} \approx 1$, $\sin \theta_{ik} \approx 0$ and $V_i \approx 1$, so that, provided that shunt admittances are ignored, the diagonal elements of the Jacobian can be approximated by

$$H_{ii} \approx L_{ii} \approx -B_{ii} \quad \text{and} \quad -N_{ii} \approx M_{ii} \approx G_{ii}$$

as $Y_{ii} = -\sum_{k \in G(i)} Y_{ik}$. It follows that the diagonal $2 \times 2$ block-pivots can be approximated by

$$\begin{pmatrix} B_{ii} & G_{ii} \\ G_{ii} & -B_{ii} \end{pmatrix}$$

Under normal circumstances $|B_{ii}| > |B_{ik}|$ for $k \neq i$, but these are only half the off-diagonal elements. Usually the factors $B_i > G_i$ and the diagonal blocks, but the matrix is usually not diagonally dominant. However, even if the reactive terms $B_i$ become small as a result of the cancellation between inductance and capacitance, the inverse of the $2 \times 2$ blocks is still well defined. In practice pivoting was required in all cases if $1 \times 1$ pivots were used, while the use of $2 \times 2$ pivots always led to a stable decomposition without pivoting.

That $2 \times 2$ blocks provide a stable pivot throughout has to be justified. As the calculation of the LU-decomposition proceeds the diagonal blocks are updated, but they retain essentially the same structure, just with some of the contributions removed. To see this consider a system with three busses and two transmission lines. Then the Jacobian for this system (assuming no swing bus and ignoring any line and shunt admittances) is

$$J = \begin{pmatrix} B_1 & -G_1 & -B_1 & G_1 & 0 & 0 \\ G_1 & B_1 & -G_1 & -B_1 & 0 & 0 \\ -B_1 & G_1 & B_1 + B_2 & -G_1 - G_2 & -B_2 & G_2 \\ -G_1 - B_1 & G_1 + G_2 & B_1 + B_2 & -G_2 & -B_2 & 0 \\ 0 & 0 & -B_2 & G_2 & B_2 & -G_2 \\ 0 & 0 & -G_2 & -B_2 & G_2 & B_2 \end{pmatrix}$$

Doing one step of the LU-decomposition results in the minor matrix

$$J^{(1)} = \begin{pmatrix} B_2 & -G_2 & -B_2 & G_2 \\ G_2 & B_2 & -G_2 & -B_2 \\ -B_2 & G_2 & B_2 & -G_2 \\ -G_2 & -B_2 & G_2 & B_2 \end{pmatrix}$$

and the diagonal blocks still have the same form. As the matrices are quite well scaled due to the use of p.u. values, it is only necessary to check the size of the determinant of the $2 \times 2$ pivot to check for stability. In cases where the pivotal matrix is nearly singular, Irving
Set IWrk(I) to Row(I), i.e. the position of the first nonzero element in every row.

DO I = 1, N, 2
   I0ffs = Row(i+1) - Row(i) Offset of element in column I in row i+1
   Pd = A(Dia(I)) First element of diagonal block
   Pb = A(Dia(I) + 1)
   Pc = A(Dia(I) + I0ffs)
   Pa = A(Dia(I) + I0ffs + 1)
   Compute determinant and check that large enough.
   Compute inverse of 2 × 2 diagonal element.
   The structure of the pivotal row determines which rows have nonzero elements in column I and I+1.
   DO K = Dia(I) + 2, Row(I+1) - 1, 2
      IK = Col(K) Row IK has an element in column I
      JK = IWrk(IK) This is the index of the nonzero element in column I
      IWrk(IK) = IWrk(IK) + 2 Update pointer to next nonzero column
      I0ffs2 = Row(IK+1) - Row(IK) Offset to next row
      Multiply inverse of pivotal block with 2 × 2 block in this column and store in A
      DO J = Dia(I) + 2, Row(I+1) - 1
         Increment JK until Col(JK) = Col(J)
         Update A(JK) and A(JK+I0ffs2)
         JK = JK + 1
      END DO
   END DO
END DO

Figure 3-4: An outline of the sparse LU-factorisation algorithm as used in the implementation of load-flow analysis. Note the use of a temporary array to minimise the searching through target rows and the way in which pairs of rows are handled.

and Sterling [70] perturbate the matrix to keep the decomposition stable. This was never necessary in any of the problems considered in this thesis.

The LU-decomposition algorithm is thus a non-pivoting algorithm which uses 2 × 2 pivots throughout. As discussed in the next section the diagonal elements for regulated busses are still 2 × 2 pivots, as an additional equation is added to give a block of the form

\[
\begin{pmatrix}
  B_{ii} & 0 \\
  0 & 1
\end{pmatrix}
\]

The position of fill-in is pre-computed and space is left in the data structures of the Jacobian for these additional elements. The decomposed matrix overwrites the original Jacobian, which saves memory. The decomposition algorithm also makes use of a temporary
array \text{Irk} to avoid searching through the data structures for the elements that have to be updated, as illustrated in the outline of the sparse matrix factorisation algorithm in figure 3-4. This avoids any searches through the sparse data structures for the elements in the pivotal column. A second array, \text{Dia}, is used to access the position of the main diagonal in every row. Rows are always eliminated in pairs and this is greatly simplified by noting that both rows in a pair have identical structure. This means that two elements in the same column in a row-pair are separated by a fixed offset in the sparse data structures.

It is important that the fill in a matrix with symmetric structure is symmetric as well. This simplifies the update of the matrix during decomposition significantly, as the structure of column \(i\) is the same as the structure of row \(i\). All searches for rows which have nonzero elements in the pivotal column can thus be avoided.

These ideas transfer immediately to the forward and back substitution routines. These algorithms are very efficient, which is reflected in the very short execution times given in table 3-4.

### 3.8.5 Regulated busses

A major issue in the load flow problem is the handling of regulated busses. These impose constraints on the range of some variables and, worse, modify the problem that must be solved, if variables are at their limits. At a PV-bus only an equation for the real power mismatch is required. As discussed in section 2.5.4, the reactive power mismatch instigates a change in the reactive power generation and not in the voltage magnitude.

After every calculation of the power mismatches the reactive power in regulated busses is updated, so that \(Q^{\text{new}}_G = Q^{\text{old}}_G - \Delta Q\), and \(\Delta Q\) is set to zero. But \(Q_G\) is bounded by

\[
Q_G^{\text{min}} \leq Q_G \leq Q_G^{\text{max}}
\]

Presume, for example, that \(Q_G > Q_G^{\text{max}}\) after an update; then \(\Delta Q\) is set to \(Q_G - Q_G^{\text{max}}\) and \(Q_G = Q_G^{\text{max}}\). This bus is then changed from a PV-bus into a PQ-bus, so that the voltage magnitude \(V\) is allowed to vary.

The effect is an increase in the voltage magnitude. It often happens that the voltage magnitude decreases again in the next iterations, and once it is below the voltage prescribed for the regulated bus, it is fixed again and the bus is converted back into a PV bus. If a bus is converted back into a PV bus, the voltage is modified, which affects the power mismatches in all other busses as well. If this is the case all the power mismatches must be recalculated, which makes this an expensive modification.

Every change of a PV-bus into a PQ-bus adds an equation to the system of nonlinear equations, while every change of a PQ-bus into a PV-bus removes one. From the discussion on sparse data structures it follows that this is a very unattractive feature, as the sparsity
3.8. IMPLEMENTATION OF THE NEWTON-RAPHSON METHOD

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Iterations</th>
<th>Time</th>
<th>#Reg. Busses at bounds</th>
<th>#Reg. bus changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee14</td>
<td>2</td>
<td>38 ms</td>
<td>3/4</td>
<td>3</td>
</tr>
<tr>
<td>ieee30</td>
<td>3</td>
<td>53 ms</td>
<td>4/5</td>
<td>4</td>
</tr>
<tr>
<td>ieee57</td>
<td>3</td>
<td>69 ms</td>
<td>1/6</td>
<td>1</td>
</tr>
<tr>
<td>ieee118</td>
<td>4</td>
<td>142 ms</td>
<td>15/53</td>
<td>15</td>
</tr>
<tr>
<td>pow236</td>
<td>4</td>
<td>252 ms</td>
<td>31/107</td>
<td>31</td>
</tr>
<tr>
<td>pow529</td>
<td>4</td>
<td>565 ms</td>
<td>58/222</td>
<td>60</td>
</tr>
<tr>
<td>pow795</td>
<td>6</td>
<td>1156 ms</td>
<td>88/336</td>
<td>98</td>
</tr>
<tr>
<td>pow1419</td>
<td>4</td>
<td>1323 ms</td>
<td>188/650</td>
<td>196</td>
</tr>
</tbody>
</table>

Table 3-4: Timings for the sequential Newton-Raphson method on a single processor of an Intel iPSC/860 hypercube. The number of regulated busses that had converted to PQ busses is also given, as well as the total number of regulated busses that changed state.

structure changes and must be recalculated. It also means that there are no simple $2 \times 2$ blocks in the Jacobian. The solution to this problem is to leave space in the data structures as if all the PV busses were PQ busses. If a bus is a PV bus, the column and row corresponding to the reactive power for that bus are set to zero and the main diagonal element to 1. This adds an equation

$$(\Delta Q = \cdots) 0 = 1 \cdot \Delta V$$

so that $\Delta V = 0$. Consequently the voltage magnitude for that bus does not change, as desired.

3.8.6 Termination criterion

The difficulty in determining the correct termination criterion is that the error, that is the size of the power mismatch $\Delta S$, in every single bus must be within specified limits. Using the Euclidean norm averages errors out and the infinity norm $\| \cdot \|_\infty$ is thus used, guaranteeing that the error in every bus is then smaller than a specified amount.

As mentioned earlier the error in measuring some of the quantities in a power system are up to 5% and a high precision in the solution of the nonlinear system of equations does not add any information. Here the termination criterion was chosen as

$$\max\{\|\Delta P\|_\infty, \|\Delta Q\|_\infty\} < 10^{-4}$$

throughout.

3.8.7 Results

The results in table 3-4 for the sequential Newton-Raphson method applied to the load flow problem were obtained on a single processor of an Intel iPSC/860 hypercube. The names
of the runs correspond to the names of the data sets in section 2.6. In all cases the starting values were a so-called flat start, in other words \( V_i = 1.0 \) and \( \theta_i = 0.0 \) for all \( i \). The times do not include input/output or the reordering time, but purely the time spent on the solution of the nonlinear system.

The results show that the Newton method always converges in very few iterations and that this is a very efficient implementation, as indicated by comparisons with timings provided in the literature (e.g. [80]). Table 3-4 also gives the number of regulated busses that have converted to PQ busses. In all cases a substantial proportion of the PV-busses have changed state, demonstrating that this is by no means an exceptional occurrence. The total number of state changes is also given, and there are not many more state changes than regulated busses at their limits. In fact, most regulated busses have reached their final state after a single Newton iteration. For the larger systems there are, however, a few reversions of PQ busses back to PV busses, which necessitates a recalculation of the power mismatches.

3.9 Overview of the parallelisation of load flow analysis

In this section various approaches to the parallelisation of load flow analysis are discussed. Initially some general observations are made and then published approaches are summarised.

First of all it is important to note that there are two main costs in a load flow ‘iteration’, where iteration is a recurrent set of operations in some iterative method for the solution of the nonlinear system of equations. These are

1. The computation of the power mismatches and (in some algorithms) the computation of the Jacobian, and

2. The solution of a sparse, linear system of equations.

Another cost that cannot be ignored, is the handling of regulated busses, but this will nearly always depend on how the operations above are parallelised.

The computation of the power mismatches is trivially parallelisable, as is the computation of the Jacobian. These can both be computed independently for all the busses in the system. On a parallel machine the only difficulty is to have the required data available for this calculation.

The second large cost is the solution of the linear system. In the case of parallelising any of the sequential methods described earlier in this section, the linear system will be distributed over several processors. Previous research has shown that it is hard to solve such systems efficiently over a large number of processors [76, 16].
3.9. OVERVIEW OF THE PARALLELISATION OF LOAD FLOW ANALYSIS

Of the methods discussed earlier, the decoupled methods initially seem ideal candidates for parallelisation as the iterations on the real and the imaginary part of the system could be done in parallel. But the convergence analysis has shown that it is essential for rapid convergence to solve these systems alternately. This makes methods such as FDLF hard to parallelise, as now two linear systems, each half the size of the linear systems encountered in the Newton-Raphson method, have to be solved successively.

An IEEE task force on parallel processing in the power systems industry [131] states bluntly ‘since the pattern of sparsity is irregular, parallel sparse matrix methods have been difficult to find’. This is the crux of the problem of developing parallel methods for load flow. They also emphasise a difficulty with most parallel implementations, namely that many parallel algorithms are many times slower than the best serial implementation, and even with a good speed-up on many processors they never manage to beat the best serial implementations — rendering the parallelisation effort useless. It is, however, a general rule that options for parallelisation increase with the size of the system, and Koester at al. [83] state that systems with 10 000 equations (i.e. linear systems with \( \approx 20000 \) variables) have to be solved in real-life problems.

Before summarising the options that will be investigated in this thesis, approaches reported in the literature are reviewed. Special emphasis will be placed on the granularity of the problem. Only articles exposing the different trains of thought on parallelisation are cited here. Many of these discuss circuit simulation, but this is a very similar problem to load flow analysis, and the development of parallel algorithms started much earlier.

The approaches can be divided into three classes:

- The first class are methods which attempt to parallelise the solution of the linear systems of equations. These can be divided into fine-grain and medium-grain algorithms, which are quite successful on shared memory processors, and coarse-grained methods that attempt to exploit the structure of the underlying network in the parallel method.

- The second class are methods which attempt to use parallel versions of nonlinear methods, especially parallel versions of the nonlinear Jacobi and SOR methods.

- The third class of methods makes the splitting of the underlying network more explicit and is known as diakoptics.

Most of the methods for which parallel implementations are reported are parallel versions of the Newton-Raphson method.

3.9.1 Parallelisation of the linear system solution

A typical article on the parallelisation of the solution of the linear systems arising in the Newton method is that by Sadayappan and Visvanathan [115]. They discuss circuit simulation
on a shared memory multi-processor, employing a fine-grained as well as a medium-grained algorithm. A task graph for every single elemental operation is constructed and then levelised. Precedences in such task graphs for sparse LU-decomposition have been considered by many authors, see e.g. [105, 74, 31]. Levelisation implies identifying tasks which are on the same level in the task graph and which can thus be executed independently. This greatly reduces the cost of scheduling the operations. This is a technique analogous to that used in the reordering of sparse matrices as discussed later in this thesis. Having this many fine-grained tasks gives good load-balancing, and the authors claim that the scheduling cost is limited. The real cost is the irregularity of data-access: memory-access is pipe-lined in most modern computers and a large penalty is paid if the data accessed is not in consecutive positions in memory. This is of course much worse on a distributed memory machine, where 'irregular data access' implies communication.

Another medium-grained approach is used by Kaiser et al. [80] for load flow analysis. They pair the real and imaginary parts of a power mismatch equation and apply the elimination steps to two equations at a time. The full matrix is simply split into blocks of rows and these are distributed in a round-robin fashion over the (4 to 8) transputers they use. Then the eliminations are done with two rows at a time (although they do not seem to use $2 \times 2$ pivots). Due to the round-robin fashion of distributing block rows they get good load-balancing and they report competitive results. No ordering to minimise computation is done, just reordering to minimise fill. A similar approach is used by Lee et al. [91] on an iPSC Hypercube.

Berry et al. [7] also use an approach where the matrix is not explicitly reordered to enhance parallelism. But they do try to detect blocks on the diagonal and assign these to separate processors. They do however change the order in which the different parts of the LU-decomposition are done to improve performance.

Sadayappan et al. [115] consider a medium-grained algorithm, in which complete row updates are scheduled separately. They see as one of the main problems the cost of gather/scatter operations to match nonzero elements in the pivotal row with those in the row that is updated. But the sequential implementation of a Newton method in section 3.8 shows that these costs can be largely avoided. Their main reason for using row-updates is to employ vectorisation — for which they claim a speed-up of up to 70%. This figure is unrealistic for the very sparse systems arising in load flow analysis (on average 6–8 elements per row, compared with nearly 18 in their case). Going from a single-processor, unvectorised version to a fully-vectorised parallel version on 6 processors, they claim a speed-up of up to 7.35. Based on personal experience of the author with this type of problem, this is probably more due to the efficiency of the sequential method used for comparison than the efficiency of the parallelisation (the increase in the overall cache size only has limited effect,
as there is very little re-use of data).

These approaches are not viable on distributed memory machines due to the high communication costs. The obvious idea is to cluster the tasks from the task graph in such a way that it is known at the start of the calculation which updates will be executed on which processor. This idea is investigated by many authors, e.g. Lau et al. [88]. They map subtrees of the task graph onto different processors and show that this is much more efficient than to map single tasks onto different processors.

Mapping subtrees of the task graph onto different processors is equivalent to employing node tearing. Node tearing means that a circuit is split into independent subcircuits by removing a (small) set of nodes (busses) from it. The operations involved in the LU-decomposition of a sub-circuit are then independent of those required for the LU-decomposition of any other sub-circuit, and communication is only needed for the LU-decomposition of the block containing the torn nodes. This algorithm is discussed in detail in the next part of this thesis. Node tearing as basis for a parallel algorithm has proved to be by far the most popular approach and many authors report on it. One of the earliest references to node tearing is by Sangiovanni-Vincentelli et al. [118] and goes back to 1977.

The basis for the parallelisation of node torn systems rests on the realisation that some columns can be eliminated independently from one another during an LU-decomposition. One example of a node torn algorithm is that reported by Cox et al. [21]. They consider various granularities and find that the most effective method is to split the circuit into more sub-circuits than processors and use the surplus of tasks created in this way to improve load-balancing. It should be emphasised that they were working on an Alliant system with shared memory, and that the cost of distributing tasks on a distributed memory machine is much higher. They report speed-ups of up to 3.85 going from 1 to 4 processors. Parallel implementations for the solution of node torn systems are also reported by Koester et al. [84, 83]. For circuit simulation Braham and Hamblen [9] also used parallel LU-decomposition on a node torn system. Node tearing can also be done in a hierarchical manner, i.e. the subsystems are split with node tearing again. This is the approach used for example by Schmitz and Glavitsch [120]. In previous work [76] this approach is also discussed, and further work has shown that the hierarchical approach creates additional overheads which make it only worthwhile in exceptional cases. This issue is further discussed in part II of this thesis.

Other methods for tearing power systems are also discussed in the literature. Branch tearing, i.e. splitting the system by removing tie-lines, is discussed by Wu [141]. This approach is also considered later in this thesis.

Another interesting approach is the solution of the linear systems with iterative algorithms. Mayaram et al [97], Decker et al. [25] and the author [76] investigated the use
of preconditioned conjugate gradient algorithms for the solution of similar systems of equations. The first two references used a block-preconditioned method applied to a node torn system, while traditionally branch torn systems are used with block-preconditioning. In [76] the effectiveness of the CGS method was compared to that of direct methods and it was concluded that iterative methods were not competitive on sequential computers. In part II of this thesis a detailed comparison of direct with new iterative methods on a parallel computer is presented.

3.9.2 Other nonlinear methods

Many attempts have also been made to parallelise the nonlinear methods themselves. The oldest load flow methods, such as Gauß-Seidel, are fairly easy to parallelise if the independence of nodes in different sub-circuits is taken into account. In part III of this thesis the reasons why these methods are not competitive are discussed.

Chen and Berry [19] use a parallel Gauß-Seidel method after an initial reordering with branch tearing for the simulation of the space shuttle’s power system. They report good speed-ups when moving from 1 to 4 processors, but their comparison is not with the best sequential algorithms. Taoka et al. [130] use essentially a parallel Jacobi method on a hypercube for real-time simulations. Chai et al. [17] compare an SOR-Newton method with a Newton-Raphson method and conclude that the Newton-Raphson method is more efficient on distributed memory computers. Braham and Hamblen [9] also used nonlinear SOR and Jacobi methods for circuit simulation. A very fine-grained version of the nonlinear Gauß-Seidel method was implemented on the Connection Machine for Circuit Simulation [139].

Zhang et al. [142] developed an interesting method for node torn systems in which several Newton iterations are done on block rows of the system, before including the connecting nodes in a full iteration. He reports applications of his method to circuit simulation. This approach looks promising and similar ideas are considered in part III.

3.9.3 Diakoptics

An approach that seems very attractive for parallelisation is *diakoptics*. This methodology was developed by Kron [85] and extended by Happ [59]. The way they approached the subject was very complicated as they worked with explicit Kirchhoff laws, and spend much of their time examining loops in graphs. The fundamental idea is very simple though: separate a complete network into subsystems and examine every subsystem on its own, only adjusting the inputs and outputs as required. Unfortunately it is not straight-forward to transfer these ideas to modern simulation systems, as nodal analysis causes the Jacobians for subsystems without their own swing bus to be singular. For an explicit calculation see
appendix B. Rafian et al. [108] use a method derived from the diakoptic method and the FDLF method. They add additional nodes to make the solution process stable.

A similar idea is developed later in this thesis, which leads to reduced systems. The main difference is that reduced systems do not require additional nodes, but the solution of a smaller equivalent system.

3.9.4 Lines of investigation

Only a few of the exponents of the very large literature on parallel methods have been cited here, but they do exemplify the main research areas. Underlying most of these algorithms is the problem of partitioning a network, as is emphasised by Chai and Bose [16] in their discussion of bottle-necks in parallel implementations. This problem is investigated in some detail in chapter 7.

Assuming that it is known how to partition a system, the question is still how to exploit this partitioning in a parallel algorithm. This thesis concentrates on coarse-grained approaches for distributed memory machines, where one of the main objectives for a good parallel algorithm is to minimise communication costs and the proportion of sequential code. In part II the emphasis is on parallel methods to solve the sparse systems of linear equations arising in the Newton method. In part III the best of these linear solvers is incorporated into a full load flow method. This means taking into account problems specific to load flow analysis, such as the handling of regulated busses. The reason for investing this effort in speeding up the linear solution methods is that the Newton method is well-established, and it is preferable to use a method which is so well understood to a new parallel method with uncertain convergence characteristics.

In part III parallel nonlinear algorithms are also investigated, including the generalised linear methods (nonlinear Jacobi and SOR) mentioned above.
Summary of Part I

In this section of the thesis the modelling of power systems was analysed in detail and a variety of methods for solving the load flow problem were described.

Many of the characteristics of the load flow problem are important for the choice and the behaviour of solution methods, and these are summarised here.

The load flow problem is formulated as a system of complex, non-analytic equations. To solve this system, it is split into a real and imaginary part. Here only the nodal formulation of the load flow problems is considered. The advantage of this approach is the ease with which the system of equations can be constructed and the simple one-to-one mapping between the graph representing the power system and the structure of the admittance matrix. Throughout per unit values are used, which ensure that the problem is always well scaled. The polar formulation is preferred for describing component behaviour, although it necessitates the evaluation of trigonometric functions. One reason is that the polar formulation ensures that a good starting value is always known, namely a flat start: the voltage magnitudes are chosen as $V_i = 1$ p.u. in PQ busses and as $V_i = V_i^{\text{ref}}$ in PV busses, while the voltage angles are chosen as $\theta_i = 0$, except in the swing bus where $\theta_i = \theta_i^{\text{ref}}$. In most systems the voltage magnitudes at the solution are close to 1 p.u., while the angle differences $\theta_{ij}$ are close to zero. A second reason is that the polar formulation gives engineers the information they require, that is the voltage magnitudes and angles at the busses, without any further post-processing. A third reason is that the interaction between the different variables in each node is usually clear-cut: the real power $P_i$ and the voltage angle $\theta_i$ interact strongly, as do the reactive power $Q_i$ and the voltage magnitude $V_i$.

A difficulty in load flow problems is the presence of regulated (PV) busses. Constraints are imposed on the reactive power generation in such busses, and when the constraints are breached, the PV bus reverts to a PQ bus, in other words the problem changes. This has a significant effect on the solution time, as it leads to a change in the structure of the system for many methods, triggers the recalculation of the iteration matrix in methods such as FDLF, and increases the error in the power mismatch in the regulated bus.

For most solution methods the Jacobian, or some approximation of it, is required. As was discussed in section 3.3, the structure of the Jacobian is obtained by expanding every
nonzero element in the admittance matrix to a $2 \times 2$ block. The Jacobian used here is only the derivative with respect to the variables $V_i$ and $\theta_i$ — if the reactive power generation is variable, as in PV busses, it is handled separately as it is just a linear variable.

At PV busses the voltages $V_i$ can be constant in some iterations of any solution method. In this case the corresponding rows and columns of the Jacobian are set to zero, with a one on the main diagonal. This has the advantage that the structure of the Jacobian remains the same throughout the solution process.

Several methods for the solution of load flow systems were discussed. Some of the classical methods, such as the SOR method, require very little storage, but converge very slowly. For fast convergence it is essential to use a higher-order method, in other words the Newton method or one of its derivatives. The main thrust of load flow research has been aimed at finding ways of using constant approximations to the Jacobian in the Newton method, as this avoids the recalculation of the Jacobian and, above all, the LU decomposition.

In section 3.7 this issue was discussed and the conclusion was drawn that, contrary to popular perception, no other method has significant advantage over the Newton-Raphson method, while the Newton-Raphson method is much more generally applicable than other methods. The decision was thus taken to concentrate on the Newton-Raphson method as a solution method.

Approaches to parallelisation of the load flow solvers were surveyed in section 3.9. These can be classified as: i) methods that parallelise the solution of linear systems, ii) methods that parallelise the solution of the nonlinear systems; and iii) diakoptic methods. Methods in the first two classes are examined in parts II and III respectively, while ideas from diakoptics are used in the development of reduced systems.
Part II

Parallel methods for linear systems
The subject of this part of the thesis is the development of algorithms for the parallel solution of the linear systems arising during load flow analysis. Here the emphasis is on the linear systems that have to be solved when the Newton-Raphson method is applied to the load flow problem.

It was seen in section 3.8 that a feature of network problems in general is that they lead to linear and nonlinear systems whose structure is determined by the graph of the network. These systems are usually large, very sparse, and symmetrically structured; otherwise the structure is quite irregular. This structure is taken into account in all the algorithms discussed here. However, in most applications the linear systems are not numerically symmetric and they are usually indefinite and badly conditioned (large spectral radius and condition number). Due to the bad conditioning the algorithms have to be tested on real-life problems, to determine whether solutions can be found in an acceptable time or at all.

It should be emphasised that general solution methods are not considered here. By this general sparse solvers, of which Harwell's MA48 routine is an example, are meant. As the systems have a complicated structure experience shows that special purpose methods, that take the structure into account, perform much better on a parallel machine. It is also not sufficient to simply attempt to parallelise a sequential method, as this offers scant chance of achieving any speed-up; for a discussion on the problems with parallelising sparse solvers see for example Daydé and Duff [23]. Instead all characteristics of network problems in general and the load flow problem in particular are exploited to give fast parallel solution methods.

When developing parallel algorithms the type of target architecture cannot be ignored. Here MIMD (Multiple Instruction – Multiple Data) distributed memory machines are considered. An attractive feature of such machines is that they are cheap, as they use of-the-shelf processors, and large machines can be built that do not suffer from bus-contention problems in the same way that most shared memory machines do. The problem with distributed memory machines is that communication between processors is prohibitively expensive compared to computation. All algorithms attempt to minimise communication. Although it may seem that such algorithms are only useful for distributed memory machines, this is not strictly true: algorithms that minimise communication will also have low memory contention on shared memory machines.

In chapter 4 a classical direct method for the parallel solution of such systems of linear equations is discussed. This method proves to be limited in respect of scalability and speed-up, which is why iterative methods for the same problem are examined in chapter 5. The discussion of block-preconditioners for iterative methods leads to the idea of system reduction and a novel solution method in chapter 6.
The development of methods in these chapters is as follows:

- In chapter 4 direct methods for double-bordered block structured matrices are considered. These methods do not scale well to many processors.

- The iterative methods in chapter 5 require expensive block-preconditioning and are partitioned into a block structure. The systems are reordered to make the off-diagonal blocks as sparse as possible. Unfortunately these methods turn out to be uncompetitive.

- It turns out that by reordering the equations in the diagonal blocks a large part of the work required for block-preconditioning can be saved. This can be understood as generating a reduced system.

- Finally this reduced system leads to a new direct method, which scales better than any of the other methods.

As the partitioning of networks is necessary in order to apply any of these parallel algorithms, heuristic methods for graph partitioning are discussed in chapter 7. A generalised version of a classical clustering algorithm is developed, which performs well on a wide range of power problems.

Finally the effectiveness of these methods is compared for practical problems. Chapter 8 discusses various implementation details as well as the data used and results. The results were obtained on an Intel iPSC/860.

The contribution of this part of the thesis is a detailed and fair comparison of a whole range of parallel solution methods for real-life problems; the concept of reduced systems in the context of the parallel solution of linear systems derived from networks; as well as many details in the parallel algorithms.

Most of the work in this part of the thesis appeared earlier as a technical report [78] and in conference proceedings [79].
Chapter 4

Parallel direct methods

4.1 Introduction

Gaussian elimination is often the most efficient solution method for sparse systems of linear equations. Even on sequential computers several difficulties arise from the use of Gaussian elimination:

- the elimination process causes fill-in, that is new nonzero elements, and,
- to guarantee a stable elimination process, it is usually necessary to pivot.

All the matrices dealt with in this report have symmetric structure, and consequently the fill-in caused by Gaussian elimination is symmetric as well, provided that diagonal pivots are used. In figure 4-1 a sparse matrix $A$ is given. The elements $\bullet$ are the original nonzero elements in the matrix, while fill-in is denoted by $\circ$. In this example there is substantial fill-in. Fill-in can cause serious problems as it increases memory usage and the number of arithmetic operations. The latter can in turn result in an increase in numerical errors, as more floating point operations imply more rounding errors.

In many instances the amount of fill-in can be reduced by reordering the linear systems [32, 52]. For most of the reordering algorithms discussed in the following sections it will be necessary to work with the graph of a sparse matrix. Such a graph is a representation of the zero/nonzero structure of a matrix, as illustrated for power systems in section 3.8.2. Every diagonal element corresponds to a node in the graph, and off-diagonal nonzero elements correspond to edges. For general sparse matrices directed graphs must be used, but for symmetrically structured matrices ordinary graphs are sufficient. In figure 4-1 the graph of the matrix $A$ is given. A reordering of the matrix corresponds to renumbering the nodes in the graph.

This does mean that the sparsity structure after fill-in has to be determined before or during the computation. It is more efficient to compute the final structure before computing
the decomposition, but this is not always possible if pivoting is required. Pivoting alters the sparsity structure, as the order of rows and columns is changed. For such problems it is usually better to use a Markowitz-type algorithm [96], which determines the structure of the decomposed matrix during the elimination process.

Bunch and Parlett [14] took up an idea of Kahan and proposed to use $2 \times 2$ pivots as well as $1 \times 1$- pivots. This allows more flexibility in the way pivots are chosen, and Duff et al. [36], who extended Markowitz’s algorithm to $2 \times 2$ pivots and provide a complete error analysis for the symmetric indefinite case, found that it reduces fill-in in many cases.

As discussed in section 3.3.4 a stable decomposition of the linear systems arising in load flow analysis can be obtained when using $2 \times 2$ pivots throughout. Consequently pivoting is not required during the LU-decomposition and here the reordering of the linear system does not have stability as its goal, but purely reducing the amount of fill-in.

Sparse codes for sequential computers are already very complex: linear systems must be reordered to reduce fill, the systems must be stored in sparse data structures — as discussed in section 3.3.1 — and storage space for fill-in must be provided. For a description of a modern sparse routine see the report by Duff and Reid [35] on the MA48 routine from the Harwell subroutine library. On parallel computers this situation becomes more complicated, as the sparse data structures are spread over several processors. This adds the complication of global versus local indices, in other words the position in the distributed matrix of a nonzero element in the local sparse data structures. The complex communication patterns during a sparse elimination process also make an efficient implementation very difficult. Pivoting on parallel machines causes difficulties if no suitable pivot can be found on the local processor: pivoting then requires either communication or some very complicated book-keeping. This is a major problem with parallel direct methods which is fortunately avoided here through the use of $2 \times 2$ pivots.

Many of the ideas for parallel direct algorithms originated in the solution of finite element matrices on computers with limited core memory. This led to the development of first the
4.2. REORDERING A MATRIX

Figure 4-2: The matrix from figure 4-1 after reordering with node tearing.

 frontal and then the multi-frontal methods [34]. It was realised that the sparsity of the
desired made it possible to change the order of some operations. Refer to the matrix shown
in figure 4-1: the pivot b can be used to eliminate column 2, before a is used to eliminate
column 1. This is due to the fact that $A_{12} = A_{21} = 0$. If the order of operations does
not matter, they can be done in parallel, provided that updates caused by these operations
do not ‘interfere with each other’. Some care must be taken: the updates caused by the
elimination of columns 1 and 2 can be calculated in parallel, but updating the rest of
the matrix may lead to conflicts. For example, the elimination of both columns requires
updating row 4, and, although the updates to row 4 can be calculated in parallel (only the
elements $a_{41}$ and $a_{42}$ are required), row 4 must be updated sequentially. How this is done
in practice is considered in section 4.3.

4.2 Reordering a matrix

A traditional method to reorder matrices for parallelism is node tearing. Node tearing
implies that a graph is separated by removing sets of nodes, called cut sets, from it. The
remaining nodes then fall into sets that are not connected to each other. See figure 4-2 for
an example. This type of structure has obvious advantages for parallelisation, as columns
in different diagonal blocks can be eliminated independently. The reordered matrix is said
to have a double-bordered block structure.

This type of reordering first became popular for finite difference matrices, where a cut-set
is simply a row or column in the grid. This process can than be repeated with the remaining
parts of the grid, which leads to the nested dissection algorithm [32, 52]. The success of
this algorithm relies heavily on the simple structure of the underlying graph (a rectangular
grid) — for general graphs it is much more difficult to find suitable partitionings. Finding
partitionings which are in some sense optimal, is believed to be an NP complete problem
[117]. Heuristics are thus used to find partitionings. Algorithms for partitioning graphs are
discussed in chapter 7.
A so-called elimination tree can be generated easily from the matrix structure, which specifies the order in which columns can be eliminated (see figure 4-3). Such a tree can be found by taking the triangular matrix $L$ from the LU-decomposition of $A$, and removing all nonzeros in column $i$ except the one immediately below the diagonal. Call the resulting matrix $L_i$, and the elimination tree of $A$ is the graph of the matrix $L_i + L_i^T$ [94]. Liu [93] also gives a fast algorithm to determine elimination trees from the structure of the original matrix. The leaves of such a tree represent columns that can be eliminated completely independently, while internal nodes of the tree represent columns that can be eliminated as soon as the columns corresponding to their child nodes have been eliminated.

The elimination tree depends on the ordering of the matrix. Evidently there is more scope for parallel elimination if the elimination tree is low and broad. As the examples in figure 4-3 demonstrate, the original ordering of the linear system offered very little scope for parallelism, while the node-torn system allows the parallel elimination of three columns in the first step. Liu [95] proposed a method, called elimination tree rotations, which reorders the matrix in such a way that the elimination tree is changed to a lower, broader tree, without changing the amount of fill. The problem with this approach is that the bookkeeping overhead to keep track of single, parallel column eliminations is considerable and a simplified approach is desirable.

If the elimination of several nodes is to be done sequentially (either because of the structure of the matrix, or because they are on the same processor, it is convenient to group them in a super node. In figure 4-3 the nodes $b, e$ and $d$ are in a super node, as they have to be eliminated sequentially.

Comparing this elimination tree with the corresponding matrix structure in figure 4-2, it
Figure 4-4: After the LU-decomposition of the diagonal blocks and updating the Schur complement the factors $L$ and $U$ have the structure shown above.

is seen that the diagonal blocks correspond to subtrees in the elimination tree. Parallelising this algorithm is thus mainly a matter of mapping subtrees of the elimination tree to processors.

As there are only two diagonal blocks only two processors can be used efficiently in this example. In larger matrices it is often possible to have many more blocks on the diagonal, but this usually comes at a cost: the cut sets become larger and the border consequently becomes wider. As will be seen in the next section, the width of the border is very important for the efficiency of the parallel LU-decomposition of such a system.

4.3 Solving double-bordered block structured systems

The LU-decomposition of a double-bordered block structured matrix can be interpreted as the computation of Schur complements. After the decomposition of the diagonal blocks into $L$ and $U$ factors the structure shown in figure 4-4 is obtained. Here it is assumed that the diagonal blocks were both non-singular, so that all pivoting took place in the diagonal blocks themselves. $S$ is known as the Schur complement of $A$ [32, p.60]. When using this approach the diagonal blocks may be small and dense enough to use dense data structures, which simplifies the decomposition significantly. Pivoting can be done within blocks, but this process breaks down if one of the diagonal blocks is singular. It is possible that a diagonal block is singular, while the matrix as a whole is regular, as for example in the matrix

$$
\begin{pmatrix}
1 & 1 & 0 \\
1 & 1 & 1 \\
0 & 1 & 1
\end{pmatrix}
$$

However in most practical applications the diagonal blocks are nonsingular (but see the problems with fully decoupled load flow systems, where singular diagonal blocks appear as
Figure 4-5: Parallel decomposition of the linear system shown in figure 4-2.

For the matrix $A$ from figure 4-2 the LU-decomposition proceeds as shown in figure 4-5. To aid the discussion the naming of matrix blocks indicated in figure 4-4 is used. The matrix is distributed over two processors, as the corresponding elimination tree (figure 4-3) has two subtrees. Every processor receives all elements corresponding to the nodes in the subtree assigned to it, as well as all the edges connected to any ancestor nodes. This corresponds to an L-shaped part of the matrix, consisting of a diagonal block and sections of the borders (these are the matrix blocks $A_\alpha$ and $B_\alpha$ respectively). The corner block itself is not copied. This results in the two matrices shown after the decomposition step in figure 4-5. Subsequently the LU-decomposition of the diagonal blocks — that is $A_{11}$
and \( B_{11} \) is computed, while the complete sub-matrices are updated. This fills in the corner blocks with the Schur complements of the two sub-matrices: \( S^A = -A_{21}A_{11}^{-1}A_{12} \) and \( S^B = -B_{21}B_{11}^{-1}B_{12} \). That these are indeed the Schur complements follows from the equality
\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & 0
\end{pmatrix} =
\begin{pmatrix}
L_{11}^A & I \\
L_{21}^A & I
\end{pmatrix}
\begin{pmatrix}
U_{11}^A & U_{12}^A \\
0 & S^A
\end{pmatrix}
\]
so that consequently
\[
L_{21}^A U_{12}^A + S^A = 0
\]
\[
S^A = -L_{21}^A U_{12}^A
\]
\[
= -L_{21}^A U_{11}^A (U_{11}^A)^{-1} (L_{11}^A)^{-1} L_{11}^A U_{12}^A
\]
\[
= -A_{21}A_{11}^{-1}A_{12}.
\]

The Schur complements can be stored in either a sparse or a dense matrix, but in most cases they are small, so that dense data structures are preferable.

In the next step of the decomposition process the Schur complements obtained from these independent calculations are sent to a process containing the original corner block \( C \). The Schur complement of the matrix as a whole is
\[
\hat{S} = C - \begin{pmatrix} A_{21} \\ B_{21} \end{pmatrix}^T \begin{pmatrix} A_{11} & A_{12} \\ B_{11} & B_{12} \end{pmatrix}^{-1} \begin{pmatrix} A_{12} \\ B_{12} \end{pmatrix}
\]
and as
\[
\begin{pmatrix} A_{11} \\ B_{11} \end{pmatrix}^{-1} = \begin{pmatrix} A_{11}^{-1} \\ B_{11}^{-1} \end{pmatrix}
\]
it can be computed as
\[
\hat{S} = C - A_{21}A_{11}^{-1}A_{12} - B_{21}B_{11}^{-1}B_{12}
\]
\[
= C + S^A + S^B.
\]

Consequently the Schur complement of the whole matrix is obtained through a simple addition of the Schur complements of the sub-matrices to the corner block. Lastly the LU-decomposition of \( \hat{S} \) is computed to obtain the distributed LU-decomposition of the whole matrix.

It is possible to do Gaussian elimination instead of LU-decomposition in all these steps. Use the following notation for the segments of the right hand side and the solution \( x \):
\[
\begin{pmatrix}
A_{11} & A_{12} \\
B_{11} & B_{12} \\
A_{21} & B_{21}
\end{pmatrix}
\begin{pmatrix}
x_A \\
x_B \\
x_C
\end{pmatrix} =
\begin{pmatrix}
b_A \\
b_B \\
b_C
\end{pmatrix}.
\]
The subproblem for the matrix section $A_s$ then is

$$
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & 0
\end{pmatrix}
\begin{pmatrix}
x_A \\
x_C
\end{pmatrix}
=
\begin{pmatrix}
b_A \\
0
\end{pmatrix}.
$$

After the computation of the Schur complement of this matrix, the system

$$
\begin{pmatrix}
L_{11}^A & U_{12}^A \\
L_{21}^A & I
\end{pmatrix}
\begin{pmatrix}
U_{11}^A & U_{12}^A \\
S^A
\end{pmatrix}
\begin{pmatrix}
x_A \\
x_C
\end{pmatrix}
=
\begin{pmatrix}
b'_A \\
b'_C
\end{pmatrix}
$$

is obtained. Both the Schur complement $S^A$ and the update to the right hand side $b'_A$ are now sent to the process with the corner block $C$. The right hand side $b_C$ is updated with $b'_C$ and $b'_C$ to give

$$
b_C = b_C + b'_C + b'_C .$$
The Gaussian elimination process terminates after calculating the elimination of $\tilde{S}x_C = b'_C$.

The back substitution proceeds in a similar fashion. The corner system is solved for $x_C$, which is distributed to all the subsystems. The solution $x_A$ is then obtained from

$$L_{11}^A U_{11}^A x_A = b'_A - U_{12}^A x_C.$$ 

$x_B$ is obtained analogously.

This algorithm will be referred to as NODE in the remainder of this thesis. Not all the potential parallelism is exploited, but the administration of this process is straightforward and consequently the overheads are low. An efficient, standard, sequential sparse LU-decomposition method can be used on every node, which also simplifies matters considerably.

The forward and back substitution process is done in parallel. On a message passing architecture this step can be very expensive, as the overhead in sending messages is generally large while not that much work is being done on the different processors. The alternative would be to collect the decomposed matrix on a single processor and compute the solution of the system sequentially. In many cases this would speed up the solution process relative to the parallel implementation, but it requires the transmission of the decomposed matrix, as well as significant administrative overhead. Overall it is thus better to do the back substitution in parallel — even if the back substitution as such is slower in parallel than sequentially.

For a real-life example see the structure of the node-torn Jacobian for the ieee57 test system in figure 4-6.

If a diagonal block has a double-bordered block structure, it is possible to continue distributing work over processors. This corresponds to mapping subtrees of a subtree of the
elimination tree on to different processors. This process can be continued until every leaf of the elimination tree is mapped onto its own processor. For the matrix $A$ from figure 4-2 the decomposition shown in figure 4-7 could be used. It is theoretically possible to use 3 processors for the solution of this system instead of two, but the Schur complements that must be computed at the lower levels are substantially larger. As the Schur complements become relatively larger as sub-matrices are split up, the computation to communication ratio worsens.

When simulating very large integrated circuits, for example, it may well be worthwhile to take this hierarchical approach, but communication tends to dominate the parallel execution time for most load flow problems, rendering the hierarchical approach unattractive. In fact, the diagonal blocks must be large (how large depends on the ratio between communication and computation speed) to make this type of parallelisation viable at all. To obtain speed-up on a parallel machine it is thus better to have several large diagonal blocks, rather than to pursue this fine-grained approach.

The corner block in figure 4-6 has a distinct block structure. If a graph is torn into three or more subgraphs this structure is quite common, as the tearing is achieved with distinct cut-sets. This cannot usually be exploited during the computation, as the Schur complement is usually nearly dense after fill-in is taken into account. As the Schur complements are usually small, it is best to use dense data structures and optimised library routines (such as the BLAS-3 [29] or LAPACK routines) for the factorisation.

Another aspect which needs consideration is the possible speed-up that can be achieved with the NODE algorithm. From the elimination tree it is seen that, although there is a lot of parallelism at the start of the factorisation, this becomes less as the nodes closer to the root are eliminated. Although not very relevant in this specific example, this poses a serious restriction to the amount of speed-up that can be gained by eliminating columns in parallel. In many problems an elimination tree with many leaves, but tapering sharply to the root, is obtained. As the nodes closer to the root of the tree have to be eliminated (nearly) sequentially, the algorithm’s performance is governed by Amdahl’s law: the (nearly) sequential part of the algorithm dominates the computation time. Amdahl’s law states that the speed-up can be modelled as follows:

$$S_p = \frac{1}{\alpha + (1 - \alpha)/p}$$

Here $\alpha$ is the fraction of the code that must be executed sequentially, $1 - \alpha$ is the fraction that is fully parallel and $p$ is the number of processors employed. The number of processors that can be used effectively may be limited by the structure of the problem. This equation simply states that the sequential part of the code cannot be speeded up, while the parallel part can be speeded up $p$ times. As an example, say that the elimination tree has 8 distinct
4.3. SOLVING DOUBLE-BORDERED BLOCK STRUCTURED SYSTEMS

subtrees, which are all of the same size, but that 1/10 of the nodes are in the root super-
node. Then the maximum speedup obtainable is $S_p = 1/(.1 + .9/8) = 4.7$, even if 8 or more
processors are used. Note that this speed-up is only obtainable if

1. communication costs are nil,
2. there is no administrative overhead, and
3. the load is balanced perfectly.

This is not the case in practice, and the speed-up will consequently be much lower. For a
full discussion of Amdahl’s law and speed-up see for example Golub and Ortega [54].

From this discussion it follows that the speed-up that can be achieved with the algorithm
NODE is limited by the structure of the problem, and that the size of the Schur complement
(which corresponds to a super node at the root of the elimination tree) is crucial for the
efficiency of the parallel method.

This indicates that a successful parallel method based on eliminating columns in parallel
must endeavour to reduce the amount of sequential processing as much as possible. This
is governed by the structure of the linear system, and in most cases only limited speed-ups
are obtainable, and only few processors can be used efficiently.

To summarise

- The graph of the matrix must be decomposed into two or more sections, with the
  smallest possible cut-set. This is an NP complete problem, but heuristic algorithms,
  producing varying results, exist.

- The resulting matrix must be strongly nonsingular, i.e. all the diagonal blocks must
  be nonsingular. This is usually the case for sufficiently large subgraphs in practical
  problems.

- The diagonal blocks are reordered with a fill-reducing algorithm, such as the minimum
  degree algorithm.

- The amount of parallelism is restricted by the structure of the problem.

- There is a bottle-neck when the Schur complement is updated and subsequently de-
  composed. This part of the operation often dominates the whole calculation.

Results for the method NODE are presented in section 8.10.1. Implementation issues
and comparisons with other methods are also presented there.
Chapter 5

Iterative methods

Due to the constraints on speed-ups with parallel direct methods, iterative methods are considered. The motivation is that iterative methods seem easier to parallelise, as the main costs usually result from the matrix-vector multiplications. These can be parallelised trivially, promising better load balancing and a greater degree of parallelism.

Tests with circuit simulation matrices [76] showed that the matrices arising from network simulation tend to be badly conditioned, so that preconditioning is required to speed-up or even achieve convergence. When developing parallel iterative algorithms, there is a second dilemma: the matrices tend to be very sparse, and thus the matrix-vector multiplications do not dominate the cost of the iterative methods to the same extent as they do for many other applications. Thus it is more difficult to get speed-up purely through parallelisation of the matrix-vector multiplications.

On some specially structured matrices, e.g. banded matrices, vector-processors can give a large speed-up. Due to the irregular structure of the matrices considered here, vector processors can only provide a limited speed-up. See appendix C for experiments with the vectorisation of matrix-vector multiplication with irregular sparse matrices on a single i860 node. The potential for parallel computation however, is not governed by the structure of the problem, but purely by the number of operations that can be done in parallel. As many operations occurring in iterative methods, such as dot-products and matrix-vector multiplications, are easy to parallelise, the investigation of parallel iterative methods for load flow is worthwhile.

5.1 Krylov subspace methods

A very popular class of iterative methods are Krylov subspace methods. Suppose the linear system

\[ Ax = b \]  

(5-1)

98
with $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$ must be solved. The $k$th Krylov subspace generated by a vector $x^{(0)}$ and the matrix $A$ is then defined as

$$\mathcal{K}_k(x^{(0)}, A) := \text{span}\{x^{(0)}, Ax^{(0)}, \ldots, A^{k-1}x^{(0)}\}.$$  \hfill (5.2)

A Krylov subspace method produces a set of vectors of the form $x^{(k)} = x^{(0)} + v^{(k)}$ with $v^{(k)} \in \mathcal{K}_k(x^{(0)}, A)$. Usually an attempt is made to choose $v^{(k)}$ in such a way that some measure for the error of $x^{(k)}$ is minimised over $\mathcal{K}_k(x^{(0)}, A)$.

For symmetric positive definite (SPD) systems, the Conjugate Gradient (CG) method [57, 55] is very popular. In the $k$th step it minimises the $A$-norm (defined as $\|x\|_A = \sqrt{x^TAx}$ with $A$ SPD) of the error along a direction vector $p^{(k)}$. More important, this one-dimensional minimisation is in fact a minimisation over a whole translated Krylov subspace

$$\|x - x^{(k+1)}\|_A = \min_{v \in x^{(0)} + \mathcal{K}_k(x^{(0)}, A)} \|x - v\|_A.$$  

Here $r^{(0)}$ is the residual, defined as $r^{(0)} := b - Ax^{(0)}$.

This minimisation is achieved by choosing the direction vectors $p^{(k)}$ to be $A$-conjugate, i.e. in such a way that $(p^{(k)}, Ap^{(j)}) = 0, \forall j \neq k$. Provided that $A$ is positive definite and symmetric, the $A$-conjugate vectors $p^{(k)}, k = 1, \ldots, n$ span the whole space $\mathbb{R}^n$ [55]. This gives the algorithm a finite termination property: the solution must have been found by the $n$th step, as the Krylov space and the search space are then the same. Unfortunately rounding errors prevent the CG method from achieving this goal in practice, as they cause loss of $A$-conjugacy in the search directions $p^{(k)}$. Nevertheless, the CG method is still a very successful method, and often finds an acceptable solution in much fewer than $n$ steps.

The CG method breaks down if $A$ is indefinite, as $(p^{(k)}, Ap^{(k)})$ can become very small or zero, which makes the minimisation in the $k$th step impossible. To avoid this problem, methods have been developed to compute orthonormal bases for the Krylov subspaces. One possibility is to use a Lanczos algorithm, which generates orthonormal vectors. SYMMLQ by Paige and Saunders [43] is an example of such a method.

Other methods use strategies analogous to the Gram-Schmidt procedure for orthogonalisation of the bases. The drawback of this approach is that all the previous vectors are needed in every step to produce the next iterate. To avoid this, the space can either be truncated or the method restarted. The most popular of these algorithms is GMRES [113], which uses the Arnoldi-algorithm (a modified Gram-Schmidt process) to compute an orthonormal basis for the Krylov subspaces.

The restarted version of GMRES, GMRES(k), is a powerful method. With GMRES the orthogonalisation is expensive if the number of vectors in the iteration sequence becomes large, so GMRES(k) never does more than $k$ orthogonalisations. Every GMRES(k) iteration uses the Arnoldi method to calculate an orthonormal basis for the Krylov subspace $\mathcal{K}_k(r^{(0)}/\|r^{(0)}\|_2, A)$ and minimises the residual with respect to this subspace [43].
GMRES(k) has a good norm minimisation property for the Euclidean norm of the error. Convergence can be proven if \( \frac{1}{2}(A + A^T) \) is positive definite [37]. Unfortunately the Arnoldi method that is required in every iteration is expensive. In a parallel implementation information from all processors is required, and usually the orthogonalisation is duplicated on all processors [137]. It appears that, in view of the speed of the direct methods, this is too expensive.

The Lanczos algorithm was originally defined to reduce an arbitrary non-symmetric matrix to tridiagonal form, using two sets of bi-orthogonal vectors. This algorithm generates two Krylov subspaces, using two short term recurrences [43]: one with the matrix \( A \) and one with its transpose \( A^T \). This makes it possible to develop a non-symmetric method, the bi-conjugate gradient method (Bi-CG) . Unfortunately, there is no guarantee that Bi-CG won’t break down, and its convergence behaviour tends to be highly erratic. Recently there has been a revival of interest in this class of methods.

New developments include, among others, the Conjugate Gradient Squared (CGS) [123] and Bi-CGSTab [135] methods. The drawback of these methods is that they do not have a norm minimising property, so that wild fluctuations in the residual are possible. This is certainly the case for CGS, but for the problems considered here Bi-CGSTab generated a smoothly converging residual. Both methods used are discussed in detail in [135]. Throughout the Bi-CGSTab-P [ibid] version of Bi-CGSTab is used. Recently newer versions of Bi-CGSTab, which combine Bi-CG steps with minimum residual steps, were developed by Sleijpen et al. [122].

The Quasi-Minimal Residual Algorithm (QMR), which was recently published by Freund and Nachtigal [51], and its derivatives (see [137]) are a promising class of methods. Basermann [5] and Basermann and Buecker [13] report on parallel implementations of the Conjugate Gradient Squared (CGS) method, QMR and the Transpose Free QMR (TFQMR) [50] methods on the iPSC/860 for finite element problems. They do not use pre-conditioned methods and report good speed-ups, especially for QMR.

As the linear systems that have to be solved during load-flow analysis are unsymmetric, they must either be transformed to

\[ A^T A = A^T b, \]

which would make them symmetric, or, alternatively, an unsymmetric solution method must be used. For one of the test systems the spectral radius and condition number were calculated (see figure 5-3 on page 107) and these were \( \rho(A) = \kappa(A) = 584.051 \). The convergence of the CG-method depends on the condition number, \( \kappa(A) \) of the matrix; to be precise, for the CG-method for symmetric positive-definite problems the number of iterations required for convergence is approximately proportional to the square root of the condition number of the coefficient matrix [43]. Using \( A^T A \) approximately doubles the spectral radius (and
5.2. PRECONDITIONING

with the smallest eigenvalue equal to 1, it would also approximately double the condition number in this example), and therefore this approach was not further considered.

For unsymmetric problems the choice of methods is restricted to either GMRES-type or Bi-CG-type methods. In most parallel implementations the expensive orthogonalisation required for GMRES is replicated on all processors. Because of this it is thought that GMRES-type methods will not be competitive with the direct methods discussed in the previous section. From the class of Bi-CG methods CGS and Bi-CGSTab were chosen for this project because of their speed and simplicity. QMR had not been published when this project started, and there was very little experience with the method.

5.2 Preconditioning

To obtain satisfactory convergence behaviour with iterative methods preconditioners must be used. As experiments reported in chapter 8 will show, cheap preconditioners, such as Jacobi or SOR preconditioning, do not lead to satisfactory convergence behaviour, and expensive block-preconditioners are necessary.

Before discussing preconditioners in more detail the basic iterative solution of a system is considered. A basic iterative method, or stationary method, is constructed as follows [57]:

\[
Ax = b \\
A = M - N \\
Mx = Nx + b \\
\Rightarrow \quad x = M^{-1}Nx + M^{-1}b \\
x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b
\] (5-4)

The matrix \(M\) should be easily invertible and the equation (5-4) must be consistent. A basic iterative method \(x^{(k+1)} = Gx^{(k)} + c\) is called consistent if its solution is the unique solution of the system \(Ax = b\). Fixed point theory says that the speed of convergence of such a system is determined by the spectral radius of the linear operator \(\rho(M^{-1}N)\). Intuitively it can be argued that \(M \approx A\) implies that \(N \approx 0\), so that \(\rho(M^{-1}N) \approx 0\) and convergence is very fast.

Define the error \(e^{(k)} := x^* - x^{(k)}\) with \(x^* = A^{-1}b\), and rewrite (5-4) as

\[
x^{(k+1)} = x^{(k)} + M^{-1}b - (I - M^{-1}N)x^{(k)} \\
= x^{(k)} + M^{-1}(Ax^* - Ax^{(k)}) \\
= x^{(k)} + M^{-1}Ae^{(k)} \\
\Rightarrow \quad e^{(k+1)} = e^{(k)} - M^{-1}Ae^{(k)}
\]
\[ (I - M^{-1}A)e^{(k)} \]  \hspace{1cm} (5-5)

Consider the equation (5-5) with a parameter \( \alpha_k \), so that

\[ e^{(k+1)} = (I - \alpha_k M^{-1}A)e^{(k)} \]  \hspace{1cm} (5-6)

If \( \alpha_k \) can be chosen in such a way that \( \rho(I - \alpha_k M^{-1}A) \) is smaller than \( \rho(I - M^{-1}A) \), then convergence of the stationary method will be accelerated. Krylov subspace methods choose such parameters automatically [43, 57] by solving an optimisation problem in every step.

Often the acceleration of a basic iterative method is referred to as the *preconditioning* of the matrix \( A \). Recalling that \( A = M - N \) is a splitting of the matrix, the multiplication of (5-3) from the left with the matrix \( M^{-1} \) results in

\[ M^{-1}Ax = M^{-1}(M - N)x = (I - M^{-1}N)x = M^{-1}b. \]

As the multiplication with the matrix \( M^{-1} \) changes the condition of the iteration matrix, this is called preconditioning from the left.

It is also possible to solve the system

\[ (AM_2^{-1})M_2x = AM_2^{-1}y = b \]  \hspace{1cm} (5-7)

instead of (5-4), where \( M_2 \) is a suitable, easily invertible matrix. This is called preconditioning from the right.

Of course both preconditioning approaches can be combined. The goal for basic iterative methods is to make the spectral radius \( \rho((I - M^{-1}A)M_2^{-1}) = \rho(M^{-1}NM_2^{-1}) \) as small as possible; for Krylov-subspace methods the goal is to make the condition number as small as possible. Often it is advantageous not to calculate the iteration matrix \( M^{-1}NM_2^{-1} \) explicitly. This is generally the case if the inverses of \( M \) and \( M_2 \) are expensive to compute or \( M^{-1}NM_2^{-1} \) is much more dense than \( M, N \) or \( M_2 \).

The basic iterative methods that are used for preconditioning in this work are the Jacobi, Gauss-Seidel and SOR methods. Partition the matrix as

\[ A = L + D + U \]
with
\[ L = \begin{pmatrix} 0 & A_{21} & & \\ A_{21} & 0 & \ddots & \\ & \ddots & \ddots & A_{n,n-1} \\ A_{n,1} & & \cdots & 0 \end{pmatrix} \]
\[ D = \begin{pmatrix} A_{11} & & & \\ & \ddots & \ddots & \\ & & \ddots & A_{n,n-1} \\ & & A_{1,n} & 0 \end{pmatrix} \]
\[ U = \begin{pmatrix} 0 & A_{12} & & \\ & \ddots & \ddots & \\ & & \ddots & A_{1,n} \\ & & A_{1-n,n} & 0 \end{pmatrix} . \]

Several iterative methods can be defined with this partitioning [37]. Two of the best-known methods are the following.

**Jacobi Method:** \( M = D \) and \( N = -(L + U) \), so that
\[ x^{(k+1)} = -D^{-1}(L + U)x^{(k)} + D^{-1}b \]  
(5.8)

**SOR Method:** \( M = (D + \omega L) \) and \( N = (1 - \omega)D - \omega U \), with \( \omega \in (0, 2) \), so that
\[ x^{(k+1)} = (D + \omega L)^{-1}((1 - \omega)D - \omega U)x^{(k)} + \omega(D + \omega L)^{-1}b \]  
(5.9)

The Gauß-Seidel method is a special case of the SOR method with \( \omega = 1 \). For both methods point and block versions exist. For point methods the blocks \( A_{ij} \) in the partitioning are all \( 1 \times 1 \) while at least some are larger for block methods. In the case of block methods, the inverse of the main diagonal blocks \( A_{ii} \) is needed. If these blocks are small and dense, the inverse may be computed explicitly. But if they are large and sparse, it is usually better to compute an LU decomposition \( A_{ii} = L_{ii}U_{ii} \) of every diagonal block and replace a multiplication \( w = A_{ii}^{-1}v \) with the solution of the system \( L_{ii}U_{ii}w = v \).

The block-Jacobi method is trivially parallelisable as the calculations for every block row can be done on a separate processor. It is also possible to solve the diagonal subsystem with a second iterative method instead of LU-decomposition. The resulting method is then called a two-stage multi-splitting method [12]. Convergence results for both synchronous and asynchronous parallel versions of these methods are available for certain classes of matrices. The required condition is that the splittings are regular splittings, which is usually not the case for load flow matrices. For a definition of regular splittings, see page 109.

The SOR method has not been popular for preconditioning as it destroys the symmetry of the iteration matrices in the case of symmetric problems. This is not the case with the Symmetric Successive Over-relaxation method (SSOR) which retains symmetry. Nevertheless Elman and Golub [44] report very good results with the SOR method. The main
problem with the SOR method is the choice of the parameter $\omega$. For some problems — matrices with Property $A$ as arise in the discretisation of elliptic and parabolic pde's [57, 43] — it is possible to determine the optimal value for $\omega$ analytically. For the type of problem considered here $\omega$ cannot be determined analytically, and it is necessary to experiment to find a value which will give good performance in a large number of cases.

Other preconditioning methods that were used in this work are

**Symmetric Jacobi Preconditioning:** If the matrix $A$ is (nearly) symmetric, it is advisable to do a symmetric scaling of the matrix $A$. Then the equations (5-4) become

$$S A S A^{-1} x = S b$$
$$S y = x$$
$$y^{(k+1)} = S A S y^{(k)} + S b$$

with $S$ the diagonal matrix defined by $S_{ii} = 1/\sqrt{A_{ii}}$. Now elements $A_{ij}$ and $A_{ji}$ are both scaled by a factor $1/\sqrt{A_{ii} A_{jj}}$, so that symmetry is retained.

Note that the convergence properties of a symmetrically scaled system are exactly the same as for the system scaled with the standard Jacobi method, as $D^{-1} = SS$ so that $D^{-1} A = S (S A S) S^{-1}$, and as this is a similarity transformation these have the same eigenvalues.

**$2 \times 2$ Jacobi:** In matrices resulting from a load flow problem, it is useful to use $2 \times 2$ diagonal blocks for scaling, as every pair of equations corresponds to a single complex equation. As the equations $i$ and $i+1$, $i = 1, 3, 5, \ldots$ are the real and imaginary part of the same complex equation, two rows in the matrix have exactly the same sparsity structure.

**Incomplete LU-factorisation:** Another popular preconditioning method is *Incomplete LU factorisation* (ILU). This method was introduced by Van der Vorst and Meijerink in 1977 [98]. An LU-decomposition of the matrix $A$ is computed, which is incomplete insofar that all fill-in is ignored. If $\bar{L}$ and $\bar{U}$ are the incomplete LU-factors, this means that $\bar{L}\bar{U} \approx A$. In practice there can still be quite a large error in this approximation, but ILU tends to be one of the most successful preconditioners. In the notation of the basic iterative methods, this would mean that $M \approx \bar{L}\bar{U}$ and $N \approx \bar{L}\bar{U} - A$. The method used combines pre- and post-conditioning, that is

$$(\bar{L}^{-1} \bar{A}\bar{U}^{-1})(\bar{U} x) = \bar{L}^{-1} b$$

(5-11)

The matrix $\bar{L}^{-1} \bar{A}\bar{U}^{-1}$ is not computed explicitly, as this would cause large amounts of fill-in, and may give a worse solution due to rounding errors.
5.2. - PRECONDITIONING

![Graph Diagram]

Figure 5.1: Branch tearing in a graph and the resulting matrix structure.

It is hard to implement ILU (or Incomplete Cholesky Decomposition as it is known in the symmetric case) efficiently on high-performance computers. In the case that the underlying problem has a simple structure, it is possible to compute the ILU-decomposition efficiently, as discussed by Van der Vorst [134]. Duff and Meurant [33] investigated the effect of different orderings on the effectiveness of ICCG (CG method with incomplete Cholesky preconditioning) and demonstrated that the ordering of the system can have significant impact on the effectiveness of the preconditioning. Interestingly enough the number of iterations required by the CG method is not proportional to the fill that would have occurred during the full LU-decomposition of a system — in other words, an ordering minimising fill does not necessarily result in a better preconditioner. When computing the ILU decomposition for irregular problems many of the same difficulties encountered when using direct methods reappear.

ILU preconditioning is however very effective for a wide range of problems. Heiser, Pommerell, Weis and Fichtner [63] report that ILU preconditioning of CGS and BiCGStab were the most effective algorithms on their ill-conditioned device simulation problems.

The method described here is ILU(0). Different levels of ILU are defined, by allowing limited fill-in. In ILU(1) fill-in by elements from the original structure is allowed, in ILU(2) by elements from the original structure, as well as by elements included in ILU(1), etc.

A modified version of ILU, called MILU (Modified ILU) [56], also exists. With this method the errors in the incomplete LU-decomposition are lumped to the main diagonal, so that the sum of errors per row of the matrix is equal to 0. Often these errors are weighted with a parameter $\alpha \in [0, 1]$. This improves convergence for some model problems.

Another method that could be of interest in some applications, is to compute a Cholesky factorisation of the symmetric part of the matrix, i.e. $(A + A^T)/2$ [45]. If the non-symmetric part $(A - A^T)/2$ of the matrix is small in some sense, this could be a good preconditioner.
5.3 Block preconditioning

When using one of the block methods described above, it is intuitive — details are discussed in the special case of M-matrices below — that it is advantageous to have as many of the nonzero elements in the main diagonal blocks as possible. If considering a parallel implementation with every block row assigned to a separate processor, a good load-balance decrees that the diagonal blocks have approximately the same size and the same number of off-diagonal elements. With node-tearing a matrix structure with distinct diagonal blocks is obtained, but the corner block is usually much smaller than the other diagonal blocks, and the associated block row contains as many off-diagonal elements as all the other processors combined.

It is possible to use \textit{branch tearing} instead of node tearing, i.e. removing branches from the underlying graph instead of nodes, to partition the graph. If the nodes in each cluster...
5.3. BLOCK PRECONDITIONING

(a) Eigenvalues of the unpreconditioned matrix \textbf{m118.3} ($\rho = 584.051$, $\kappa = 584.051$).

(b) Eigenvalues of the matrix $D^{-1}(L + U)$ with $D$ consisting of $2 \times 2$ diagonal blocks ($\rho = 0.997$, $\kappa = 498.5$).

(c) Eigenvalues of the block-Jacobi matrix $D^{-1}(L + U)$ with $D$ consisting of three diagonal blocks ($\rho = 0.915$, $\kappa = 19.468$).

(d) Eigenvalues of the block-SOR matrix $(D + \omega L)^{-1}((1 - \omega)D - \omega U)$ with $\omega = 1.45$, with $D$ consisting of three diagonal blocks ($\rho = 0.451$, $\kappa = 1.005$).

Figure 5-3: Eigenvalue distribution for iteration matrices for various basic iterative methods obtained from the \texttt{ieee118} test system. The spectral radius $\rho$, and condition number $\kappa$ are given in brackets, but see text.
are again numbered successively, a structure with diagonal blocks, but without a distinct 'double border', is obtained. The torn branches are found back in elements lying outside these diagonal blocks. An example of the matrix $A$ from figure 4-1, reordered with branch tearing is given in figure 5-1. Ideally the number of torn branches is kept as small as possible. For a real-life example see the structure of the Jacobian of the ieee57 test system after branch tearing in figure 5-2. Compare this to the node-torn structure of the same system in figure 4-6 on page 94.

Networks can often be disconnected in this way. In circuit simulation blocks can correspond to sub-circuits and in power systems blocks can correspond to the grids of different power companies or to splits between the high and low voltage parts of the system — transformers are expensive, so there are usually few of them. To find a minimal set of branches that will split the matrix into a set number of components is thought to be an NP-complete problem [117].

If a 'good' partitioning is found, the block diagonal matrix $D$ will contain most of the elements in $A$, so that $A \approx D$ and $L + U \approx 0$. If a block-Jacobi preconditioning is now used, most of the matrix is inverted, so that $D^{-1}A \approx I$ and the spectral radius $\rho(I - D^{-1}A) \ll 1$, which would imply very fast convergence of the preconditioned method. Computing the inverses (in practice the LU-decomposition) of $D$ is expensive, but can be done in parallel for every block. Hegland and Saylor [62] describe an implementation of the block-Jacobi method on parallel vector processors (Cray Y-MP) and show that it can be made efficient for a (regular) model problem.

To demonstrate the effect which block preconditioning has on the condition of a matrix, the eigenvalue distributions for a matrix arising in the solution of the load flow problem ieee118 (see 2.7 for details on the data sets) are shown in figure 5-3. The eigenvalue distribution for the unpreconditioned system is shown in figure 5-3(a) — with a spectral radius of $\rho = 584.051$ it is clear why it is hard to achieve convergence with a basic iterative methods for these systems of equations. The spectral radius for a $2 \times 2$ Jacobi method is $\rho(M^{-1}N) = 0.997$, hence convergence is very slow, as expected. The $2 \times 2$ Jacobi method does not converge in 1000 iterations. If the system is partitioned into three blocks, then for the block-Jacobi method there are still a few eigenvalues which are substantially larger than 0, so that the assumption that $D^{-1}A \approx I$ is much too optimistic. What is encouraging, is that there are only six eigenvalues with $|\lambda| > 0.5$ and that these are well separated. For the SOR method with a good choice for $\omega$ the spectral radius is substantially smaller. Only a few eigenvalues are nonzero, and $\rho \ll 1$. The method converges in only 16 iterations.

For Krylov-subspace methods the condition number is decisive for the speed of convergence. With block-preconditioning many of the eigenvalues are zero, but this does not mean that the system that is solve is singular. The block-preconditioning step eliminates
equations from the linear system, and when these are ignored the system is still nonsingular. This 'elimination' is explained and made explicit in the next chapter. It is encouraging to see that the condition number is reduced significantly through block-preconditioning — for block-SOR preconditioning the condition number \( \kappa \approx 1 \).

These results demonstrate that, for the type of system considered here, it is necessary to use block methods in order to achieve convergence in a reasonable number of iterations. Fast convergence can be expected for matrices with an eigenvalue distribution as in figures 5-3 (c) and (d): not only is the spectral radius small, but the extremal eigenvalues are well separated. The better the extremal eigenvalues are separated, the faster the Ritz-values converge towards them. As soon as a Ritz-value comes close to an active eigenvalue, the CG-process continues with an accelerated reduction factor, depending on the next largest eigenvalue (see [133] for details).

Motivation for block methods in the case of M-matrices.

Intuitively it is clear that, as \( M \to A, \rho(M^{-1}N) \to 0 \). If \( M \) is defined as the diagonal blocks of \( A \) after branch tearing, \( M \) is in some way 'close' to \( A \). But the question remains whether this is 'close' enough to obtain fast convergence. Some explanation for the relationship between \( A \) and a matrix \( M \), which is equal to \( A \) except that a few off-diagonal elements have been set to zero, is given by Varga [136] for M-matrices.

A nonsingular \( n \times n \) matrix \( A \) is an M-matrix if \( a_{ij} \leq 0 \) for all \( i \neq j \) and if \( A^{-1} \geq 0 \) [136, definition 3.3]. It should be said that the linear systems occurring during load flow analysis are usually nonsingular, but \( a_{ij} \leq 0 \) for all \( i \neq j \). The inverses of some small load flow matrices were computed, and these were almost completely negative, i.e. \( -A^{-1} \geq 0 \).

Varga defines regular splittings, that is a splitting \( A = M - N \) with \( M \) nonsingular, \( M^{-1} \geq 0 \), and \( N \geq 0 \). The motivation for using block methods with most of the nonzero elements in the diagonal blocks is given by the following theorem

**Theorem 5.1** ([136]) Let \( A = M_1 - N_1 = M_2 - N_2 \) be two regular splittings of \( A \), where \( A^{-1} > 0 \). If \( N_2 \geq N_1 \geq 0 \), equality excluded, then

\[
1 > \rho(M_2^{-1}N_2) > \rho(M_1^{-1}N_1) > 0.
\]

For M-matrices \( M_1 \) and \( M_2 \) can be obtained by setting off-diagonal elements to 0, so that \( N_2 > N_1 \) if more off-diagonal elements were set to zero to obtain \( M_2 \) than to obtain \( M_1 \). The reasoning now goes as follows: if more nonzero elements are in the diagonal blocks, fewer have to be set to zero to obtain a block diagonal matrix, and the above theorem states that the spectral radius is smaller if fewer off-diagonal elements are set to zero to obtain \( M \).
In this form this argument is not valid for the linear systems considered here, as the matrices are neither M-matrices, nor is the theorem valid for systems with different orderings (usually $N_2 \nleq N_1$ for differently ordered $A$, even if there are fewer nonzero elements in the off-diagonal blocks). Nevertheless this theorem does provide motivation to seek diagonal blocks which are as full as possible.

5.4 Error estimates

One of the problems with all iterative methods is to decide when to stop iterating. If the tolerance (i.e. the bound under which the error $e^{(k)}$ must fall) is chosen too small, it may not be possible to attain the precision required in an acceptable number of iterations. If it is chosen too large on the other hand the result may be far from the real solution. In the load flow problems considered here the measurement of some of the parameters (high voltages) is very difficult, and measurements may have errors of up to 5% [77]. It is senseless to solve the linear systems to an accuracy of say 10 magnitudes smaller than the voltage, as this does not add any information.

A more immediate problem is to decide how big the error is. Ideally the error would be computed from $e^{(k)} = x^* - x^{(k)}$, but the solution $x^*$ is not known. Another measure for the nearness of the solution, is the residual $r^{(k)}$ as

$$ r^{(k)} = b - Ax^{(k)} = Ax^* - Ax^{(k)} = Ae^{(k)}. $$

This is not a very good measure in general, as it may seriously under or overestimate $e^{(k)}$. If $A$ only contains very small elements, $r^{(k)}$ may be small, even though $e^{(k)}$ is large, on the other hand $r^{(k)}$ could remain large if $b$ and $A$ contain large elements.

To avoid these problems Arioli, Duff and Ruiz [2] recommend using either

$$ \epsilon_2 = \frac{\|r^{(k)}\|_\infty}{\|A\|_\infty \|x^{(k)}\|_1 + \|b\|_\infty} $$

or the classical

$$ \epsilon_3 = \frac{\|r^{(k)}\|_\infty}{\|b\|_\infty}. $$

They remark that $\epsilon_3$ may seriously overestimate the error if $\|b\|_\infty \ll \|A\|_\infty \|x^{(k)}\|_1$. This is not a serious problem, as the systems considered in this work are all fairly well scaled [77]. The measure $\epsilon_3$ was used throughout.

The next problem is how $r^{(k)}$ is to be found. In the case of an iterative process defined by $x^{(k+1)} = Gx^{(k)} + f$, a calculation shows that

$$ r^{(k)} = f - (I - G)x^{(k)} $$
$$ = f + Gx^{(k)} - x^{(k)} $$
$$ = x^{(k+1)} - x^{(k)}. $$
So an estimate for the error in step $k$ can be gained from the $(k + 1)$st and the $k$th iterates. This means that the error estimate always lags one step behind, but provided that the process is smoothly convergent, this means that $x^{(k+1)}$ is at least as good a solution as $x^{(k)}$.

If preconditioning is used the error in the solution of the preconditioned system is not the same as the error in the solution of the original system. It is straightforward to find the correct weighting of the error estimate for the stationary method (5-4) from

$$ r^{(k)} = M \cdot M^{-1}(b - Ax^{(k)}) $$
$$ = M(M^{-1}b - M^{-1}(M - N)x^{(k)}) $$
$$ = M(M^{-1}b + M^{-1}Nx^{(k)} - x^{(k)}) $$
$$ = M(x^{(k+1)} - x^{(k)}). $$

It follows that

$$ ||r^{(k)}||_\infty \leq ||M||_\infty ||x^{(k+1)} - x^{(k)}||_\infty. $$

If the system is preconditioned from the right the residual remains unchanged, as is seen from

$$ r^{(k)} = b - A(M_2^{-1}M_2)x^{(k)} $$
$$ = b - AM_2^{-1}y^{(k)} $$
$$ \overset{(5-14)}{=} y^{(k+1)} - y^{(k)} $$

with $x^{(k)} = M_2y^{(k)}$. These results are summarised in the following lemma.

**Lemma 5.2** A bound for the residual $r^{(k)}$ of a system $Ax^{(k)} = b$, $A \in \mathbb{R}^{n \times n}$, $x^{(k)}, b \in \mathbb{R}^{n}$, that is preconditioned from the left by $M_1^{-1}$, and from the right by $M_2^{-1}$, is obtained from

$$ ||r^{(k)}||_\infty \leq ||M_1||_\infty ||y^{(k+1)} - y^{(k)}||_\infty, $$

where $y^{(k)}$ are the iterates generated by the iterative method

$$ y^{(k+1)} = (I - M_1^{-1}AM_2^{-1})y^{(k)} + M_1^{-1}b. $$

Krylov subspace methods usually update an estimate for the residual in every step, for a preconditioned method, call this residual $\tilde{r}^{(k)}$. As the system that is being solved is $M^{-1}Ax = M^{-1}b$, $\tilde{r}^{(k)} \approx M^{-1}b - M^{-1}Ax^{(k)} = M^{-1}r^{(k)}$, so that $r^{(k)} \approx M\tilde{r}^{(k)}$. An estimate for the residual in the preconditioned method thus is

$$ ||r^{(k)}||_\infty \leq ||M||_\infty ||\tilde{r}^{(k)}||_\infty, $$

and this estimate for the error was used throughout. To test whether these estimates are reliable, the estimated error was compared to the real error. See the results in section 8.10.1. For Block Jacobi and Block-SOR preconditioning, the estimates were good, but for ILU preconditioning the estimates severely overestimated the error.
CHAPTER 5. ITERATIVE METHODS

5.5 Summary

In the first part of this chapter Krylov-subspace methods for unsymmetric linear systems were listed. There has been a lot of activity in this area in the last few years, and a survey to do this field justice would be longer than this thesis. For more details refer for example to the excellent overview given by Elman [43]. Two representatives of this class, CGS and Bi-CGStab, were chosen for their simplicity and the favourable reports on their practical application.

Earlier work on circuit simulation [76] demonstrated the difficulties that are encountered when applying iterative methods to the difficult class of problems considered here: the matrices tend to be very ill-conditioned so that it is difficult to achieve (fast) convergence. For this reason the preconditioning of Krylov-subspace methods with basic iterative methods is discussed in some detail.

The use of block methods can improve the conditioning of the iteration matrix, as the plots of the eigenvalue spectrum of iteration matrices for the ieee118 test problem demonstrate. Varga [136] shows that it is beneficial to have as many nonzero elements as possible in the diagonal blocks in the case of M-matrices. Some attention is thus given to obtaining diagonally blocked matrices through reordering of the systems. The key to this reordering is branch tearing.

Lastly the problem of finding a reasonable termination criterion is touched upon.

Results for the block-Jacobi and block-SOR, as well as CGS and Bi-CGStab preconditioned with block-Jacobi, block-SOR and ILU(0) are given in chapter 8 for a range of real-life problems. There the methods are compared sequentially and in parallel with direct methods.
Chapter 6

Linear System Reduction

Iterative methods using block preconditioners were described in the previous section. A strategy to make block methods more efficient is described in this section. These developments lead to a new parallel direct method and a theoretical comparison with the direct methods described in chapter 4 shows that this method may be more efficient.

The approach proposed here reorders a branch-torn linear system by renumbering the nodes within each partition. Some notation is needed. Recall the graph notation introduced on page 18.

Definition 6.1 A partitioning \( Q \) defines a reduction ordering of a graph \( (\mathcal{V}, \mathcal{E}) \), if the nodes in \( \mathcal{V} \) are partitioned as \( Q = \{Q_{11}, Q_{12}, Q_{21}, \ldots, Q_{p1}, Q_{p2}\} \) with the following condition for \( i = 1, \ldots, p \)

1. if \( v \in Q_{i1} \), then for all edges \( (v, u) \in \mathcal{E} \), \( u \in Q_{i1} \cup Q_{i2} \), and,

2. if \( v \in Q_{i2} \), then for all edges \( (v, u) \in \mathcal{E} \), \( u \in \bigcup_{i=1}^{p} Q_{i2} \cup Q_{i1} \).

A matrix is said to have reduction ordering, if the nodes in \( Q_{11} \) are numbered first, the nodes in \( Q_{12} \) next, etc.

Numbering the nodes of a graph with reduction ordering, defines a partitioning \( P = \{P_{11}, P_{12}, P_{21}, \ldots, P_{p1}, P_{p2}\} \) of the indices \( 1, \ldots, n \) by assigning the numbers of nodes in \( Q_{i1} \) to the set \( P_{ik} \).

Define a vector \( x_{[i]} \) as the vector consisting only of values \( x_r \), such that \( r \in P_{ik} \), and \( x_{[i,k]} \) as a vector with elements in \( \bigcup_i P_{ik} \).

A reduction ordering for a graph separates a graph into connected subgraphs \( i \) so that the internal nodes of a subgraph are in \( Q_{i1} \), while nodes with edges connecting them to other partitions are in \( Q_{i2} \). By numbering the nodes in the partitions monotonically, \( Q_{ik} \) are the nodes associated with the \( i \)th diagonal block in the matrix. This results in all the
rows with connections to other blocks, in other words containing elements which fall outside
the main diagonal block, being numbered last within the block-row. Such an ordering can
always be obtained by branch-tearing the graph and numbering the nodes in every partition
appropriately.

The typical matrix structure resulting from this reordering is shown schematically in
figure 6-1 and in figure 6-4 for the ise57 test system. Compare figure 6-4 to the branch
torn system depicted in figure 5-2 on page 106.

If pivoting is required, this reordering imposes a further restriction on the choice of
pivots if a reduction ordering is to be maintained. It follows that a pivot for the elimination
of a column with index in \( P_{ik} \) may only be chosen from the set \( P_{ik} \). It is desirable to
use diagonal pivoting in order to retain a symmetrical structure, which further restricts
the choice of pivots. As discussed earlier it is possible to obtain a stable decomposition in
many practical applications without pivoting, or to use system perturbations to make the
decomposition stable. The last option is frequently used in circuit simulation codes. So in
practice such a constraint on pivots is a satisfiable condition.

6.1 Constructing a reduced system

What is meant by system reduction will be illustrated with the help of the example system
from figure 6-1. First of all the graph is partitioned to have reduction ordering. In the
associated matrix the diagonal blocks are partially decomposed — that is, columns with
indices in \( P_{+1} \) are eliminated — so that the system shown in figure 6-2 is obtained.

After the partial decomposition the Schur complements of the diagonal blocks — \( S_A \), \( S_B \)
and \( S_C \) respectively — are found in rows with indices in \( P_{-2} \). Refer to the left hand matrix
as \( \hat{L} \) and to the right hand matrix as \( \hat{U} \). In the next phase of the LU decomposition, columns
in \( S_A \) are eliminated. This causes updates in the shaded region of figure 6-2 only. When
factorising the updated blocks \( S_B \) and \( S_C \) subsequently, again only elements in the shaded
area are involved. These operations are thus equivalent to computing an LU decomposition
for the reduced system \( S \) in figure 6-3.

Consider the solution of the original system \( Ax = b \) again. This is now equivalent to
solving

\[
\hat{U}x = \hat{L}^{-1}b = \hat{b}.
\]

As was indicated above, the system \( \hat{U}x = \hat{b} \), is solved in two steps: factorisation of
the system \( S \), followed by a forward and back substitution on \( \hat{b}_{[+2]} \) to solve for \( x_{[+2]} \). Subsequently
a back substitution with the triangular part of \( \hat{U} \) — in other words all rows \( s \), such that
\( s \in P_{+1} \) — will produce the whole solution \( x \).
6.1. CONSTRUCTING A REDUCED SYSTEM

Figure 6-1: A typical matrix structure after branch tearing and reordering of the variables within each block.

\[ A = \hat{L} \hat{U} = \]

Figure 6-2: The structure of \( \hat{L} \) and \( \hat{U} \) after eliminating all columns \( j \), with \( j \in P_1 \).

Figure 6-3: The reduced system.
The reduced system

\[ S\hat{x}_{[\nu_2]} = \hat{b}_{[\nu_2]} \] (6-1)

was solved via an LU decomposition in this exposition, but this system can be solved with any other method, such as an iterative method, as well.

Why (6-1) is called a reduced system still requires an explanation. Consider the two systems in figure 6-5. In the first case the system is reduced to a new system including only the two nodes that have edges to the other set in the partitioning. In the second case 'nodes with input', that is the nodes with corresponding nonzero elements in the right hand side, are included in the set \( P_{\nu_2} \), so that all the nonzero elements in \( b \) are also in \( \hat{b}_{[\nu_2]} \). Then \( \hat{L}^{-1}b = \hat{b} = \hat{b} \). Now the original system has been reduced to a much smaller system which behaves 'in the same way'. By this is meant that, given the same input — that is, the
same right hand side — the solution \( x_{[2]} \) will be identical for the original and the reduced system. In other words, to an external observer the original and reduced systems will be indistinguishable.

The construction of such a reduced system is only possible because these are linear systems. To construct a reduced system as described above the right hand side must be sparse. For the purposes of solving load flow equations such fully reduced systems are not important — after all, load flow equations are neither linear, nor is the right hand side sparse — but when solving the linear systems arising in load flow analysis, a reduced system can still be constructed. Why this may be of interest is discussed next.

### 6.2 Reduced systems and iterative methods

One could consider solving the reduced system \( Sx_{[2]} = \hat{b}_{[2]} \) with an iterative method, e.g. a block-SOR method. First consider the block form of the SOR method for block row \( i \) of a non-reduced system \( Ax = b \):

\[
x_i^{(k+1)} = \omega A_{ii}^{-1} \left\{ b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} \right\} + (1 - \omega)x_i^{(k)}. \tag{6-2}
\]

Refer to the nonzero parts of the off-diagonal blocks as \( A'_{ij} \), (these are the blocks \( A_B, A_C, B_A, \ldots \) in figure 6-1). As \( \hat{L}^{-1} A'_{ij} = A'_{ij} \) (see figure 6-2), equation (6-2) can be
rewritten as
\[
\begin{pmatrix}
    x_{[1]}^{(k+1)} \\
    x_{[2]}^{(k+1)}
\end{pmatrix}
= \omega \begin{pmatrix}
    \hat{U}_i \\
    0
\end{pmatrix}^{-1}
\begin{pmatrix}
    L_i^{-1} 
    \begin{pmatrix}
        b_{[1]}^{(k)} \\
        b_{[2]}^{(k)}
    \end{pmatrix} \\
    0
\end{pmatrix}
- \left( \sum_{i=1}^{i-1} A_{ij}^{(k)} x_{[j2]}^{(k+1)} \right)
- \left( \sum_{j=1}^{n+1} A_{ij}^{(k)} x_{[j2]}^{(k)} \right)
+ (1 - \omega) \begin{pmatrix}
    x_{[1]}^{(k)} \\
    x_{[2]}^{(k)}
\end{pmatrix}.
\]

For \( x_{[2]}^{(k+1)} \) this results in the equation
\[
x_{[2]}^{(k+1)} = \omega S_i^{-1}
\begin{pmatrix}
    \dot{b}_{[2]}^{(k)} \\
    - \sum_{j=1}^{i-1} A_{ij}^{(k)} x_{[j2]}^{(k+1)} - \sum_{j=i+1}^{n} A_{ij}^{(k)} x_{[j2]}^{(k)}
\end{pmatrix}
+ (1 - \omega) x_{[2]}^{(k)},
\]
which is exactly the block-SOR method for the reduced system (6.1). It follows from this equivalence that

- the block-SOR method only requires \( x_{[2]}^{(k)} \) to compute \( x_{[2]}^{(k+1)} \).

- the vectors \( x_{[2]}^{(k)} \) generated by the full block-SOR method are identical (except for rounding errors) to those generated by the block-SOR method for the reduced system.

It is therefore possible to use a block iterative method for the solution of the linear system without doing all the forward and back substitutions that are usually required for diagonal preconditioning. This is evidently also true for the block-Jacobi and block-Gauß-Seidel methods. Furthermore it says something about the eigenvalues of the reduced system. As exactly the same iterates are generated during the solution of the reduced system with the block-Jacobi or block-SOR method as for the complete system, the spectral radius of the reduced and the whole system, preconditioned with a block-Jacobi or block-SOR preconditioner, must be identical.

Reduced systems are not only useful for basic iterative methods, but can be used for Krylov subspace methods, pre-conditioned with basic iterative methods, as well. The result is a Krylov-subspace method applied to the reduced system only — but the iteration sequence is identical to a method using the full system (in the absence of rounding errors).

For the SOR method, as well as other methods using reduced systems, the blocks in the matrix are always ordered in an attempt to find a structure such that there is a maximum amount of parallelism. This translates to a requirement that the elimination tree, with the nodes in the sets \( Q_{\alpha_i} \) as super nodes, is as broad and low as possible. As an example, consider the following network:

```
A -- C -- B
```
If the blocks in the matrix in figure 6-1 correspond to the blocks in the system as indicated by the letters, both the blocks $A_B$ and $B_A$ would be empty, as there are no direct links between these blocks. As the SOR method always requires a forward substitution with the lower triangular matrix, this can be exploited to improve performance. The reasoning is the same as that used for multi-frontal methods in chapter 4: the iterate $x_B^{(k+1)}$ can now be computed without having to wait for the computation of $x_A^{(k+1)}$. 

Figure 6-6: Parallel decomposition of the linear system of figure 5-1 with the direct method BRANCH.
6.3 A new parallel direct method

The reordering described in section 6.1 makes it possible to reorganise the parallel LU decomposition. As mentioned there, the reduced system can be solved with a direct method. Take the system with the structure shown in figure 6-5(a) and its reduced equivalent as an example. Supposing that every processor contains a block row of the matrix $A$ the solution of the system proceeds as shown in figure 6-6. First of all the reduced system is computed by eliminating all internal nodes in every block row. Subsequently the reduced system is solved, and the result is used to calculate the solution for the whole system. Refer to this algorithm as BRANCH.

This is not obviously a good way to go about constructing a parallel algorithm. After all, the reduced system is both distributed and dense, so that the elimination process is sequential and requires expensive communication. When using branch tearing, both nodes on either side of a torn branch are in the set $Q_2$, i.e. fall into the reduced system. Only one of these nodes must be removed to cut the graph if node tearing is used. There are thus roughly twice as many nodes involved in the reduced system as there are in the Schur complement for the whole system.

A very simple optimisation which has a significant effect, is to compute the zero/nonzero block structure of the reduced block system $S$ before computing the LU-decomposition. As was mentioned in section 6.2, the blocks in $S$ corresponding to subsystems which are not connected to each other are often empty. If $S$ is reordered it is often possible to prevent some of these blocks to ‘fill in’. It is then possible to use a parallel block-multifrontal method to obtain additional parallelism. This is a very important optimisation, as the decomposition of the block system requires communication, which makes this part of the computation very time-critical. To emphasise this point, consider the example shown in figure 6-2. If the blocks $A_B$ and $B_A$ are zero, there is no need for the LU decomposition of the block $S_B$ to wait for updates from the processor containing $S_A$, and so there is some parallelism in this otherwise sequential process.

6.4 Theoretical comparison of the parallel direct methods

In this part of the thesis two direct methods were discussed. The first is the method named NODE, introduced in chapter 4; the second is the method BRANCH, introduced in the previous section. The question is which of these two methods is faster, and an attempt to answer this question is made here.

The execution pattern of parallel algorithms for sparse matrices is very complex, as it depends on their structure. It is thus difficult to decide which of the two proposed direct algorithms is the more favourable. In order to prove one algorithm better than the other,
it must be shown theoretically as well as experimentally which is the better. Therefore an analysis of the time-critical parts of both algorithms is given. The results are quite surprising, and show how flawed intuition can be in the case of parallel algorithms.

The analysis for the parallel direct method for node-torn systems (NODE) is given first, followed by an analysis of the method for branch torn systems (BRANCH). It is assumed that the graph was broken into \( p \) parts in both cases. Only the critical parts of the algorithm are discussed, i.e. updating and factorising the Schur complement in the case of NODE, and solving the reduced system on \( p \) processors in the case of BRANCH. The motivation is that the calculation of the reduced system in the case of BRANCH and the updates to the Schur complements in the case of NODE requires no communication and roughly the same amount of computation. In a comparison of these two methods this work can therefore be ignored.

Define nodes in the system graph to be critical nodes under the following conditions: in the case of NODE if the nodes are in a cut-set; in the case of BRANCH if a torn branch originates on the node.

It will be assumed that there are \( m \) nodes in the nodal cut-set. This implies that the Schur complement in the NODE algorithm is an \( m \times m \) matrix. Furthermore the assumption will be made that there are twice as many, i.e. \( 2m \), critical nodes in the corresponding BRANCH algorithm. This is based on the assumption that in the case of NODE only one node has to be removed to break a connection, while in the case of BRANCH a branch is removed. Both nodes that are connected by a branch are then critical nodes. To test whether this is a realistic assumption, the number of critical nodes was determined for different power systems which were partitioned by the algorithms described in chapter 7. The results are given in table 6-1 and, as the last column shows, the assumption that there are \( 2m \) critical nodes for branch tearing is pessimistic for the larger systems. It does not seem useful to distribute the ieees57 system (with only 57 nodes) over 4 or more processors, but solving the pow795 system (with 795 nodes) on 3 or 4 processors is a realistic assumption.

In practice all calculations on the Schur complement or reduced system are done in dense data structures, as the matrices involved are fairly small, and usually quite full. Thus only the dense case is considered in the following analysis.

6.4.1 Complexity analysis for NODE

*Factorisation of the Schur complement.* The factorisation of the Schur complement has the same complexity as the LU decomposition of an \( m \times m \) matrix. It requires \((m - i)\) divisions in the \( i \)th step, as well as \((m - i)^2\) multiplications in updating the remainder
<table>
<thead>
<tr>
<th>Power System</th>
<th>Number of Partitions</th>
<th>Number of critical nodes NODE</th>
<th>Number of critical nodes BRANCH</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee67</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td>1:2.75</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>12</td>
<td>1:2.40</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>20</td>
<td>1:2.86</td>
</tr>
<tr>
<td>ieee118</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>1:2.25</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>1:1.71</td>
</tr>
<tr>
<td>pow236</td>
<td>3</td>
<td>12</td>
<td>20</td>
<td>1:1.67</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>13</td>
<td>24</td>
<td>1:1.85</td>
</tr>
<tr>
<td>pow529</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>1:1.13</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>15</td>
<td>14</td>
<td>1:0.93</td>
</tr>
<tr>
<td>pow795</td>
<td>3</td>
<td>16</td>
<td>13</td>
<td>1:0.81</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>18</td>
<td>22</td>
<td>1:1.22</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>17</td>
<td>35</td>
<td>1:2.06</td>
</tr>
</tbody>
</table>

Table 6-1: Comparison of the number of critical nodes for node and branch tearing.

of the matrix. In total that gives

\[
\sum_{i=1}^{m} (m-i) + \sum_{i=1}^{m} (m-i)^2 = \sum_{i=0}^{m-1} i + \sum_{i=0}^{m-1} i^2 \\
= \frac{m(m-1)}{2} + \frac{(m-1)m(2m-1)}{6} \\
= \frac{1}{3}(m^3 - m)
\]  

(6-3)

(or \(O(\frac{1}{3}m^3)\)) multiplications and divisions.

Forward and Back substitution During forward substitution \((m - i)\) multiplications are required to update the right hand side in every step, and during back substitution \((m - i)\) multiplications and 1 division are required in the \(i\)th step. In total that gives

\[
2 \sum_{i=1}^{m} (m-i) + m = 2m(m-1)/2 + m = m^2
\]  

(6-4)

multiplications and divisions.

Critical communication If every processor has the same amount of work, all of them will start communicating their updates to the Schur complement at the same time. This involves sending

\[p \cdot m^2\]

double precision floating point numbers. It should also be mentioned that subsequently \(p \cdot m^2\) additions are needed to update the Schur complement.
To summarise: the total number of multiplications and divisions is
\[ \frac{1}{3}m^3 + m^2 - \frac{1}{3}m. \]
and the number of elements that have to be communicated is
\[ pm^2. \]

6.4.2 Complexity analysis for BRANCH

The complexity analysis for the BRANCH algorithm is far more difficult. In order to simplify the analysis, it is assumed that every block has exactly \( s \) critical nodes, i.e. there are \( s \) rows in every block row in the reduced system. Here it is assumed that \( s = 2m/p \) where \( m \) is the number of critical nodes in the equivalent node-torn system and \( p \) is the number of blocks. The index \( i \) always refers to the block row on which the operations are performed.

Factorisation of a block row. The factorisation of a block row involves the LU-decomposition of an \( s \times s \) matrix, which requires
\[ \frac{1}{3}(s^3 - s) \]  
(6-5)
multiplications and divisions according to (6-3). For the back substitution, the off-diagonal blocks to the right of the diagonal, each of size \( s \times s \), must be multiplied by \( L_{i1}^{-1} \). In order to update the rest of the reduced system, a further back substitution on these blocks is required. A forward and back substitution requires \( s^2 \) operations per column (see (6-4)), so for \( p - i \) blocks of \( s \) columns
\[ (p - i) \cdot s \cdot s^2 = (p - i)s^3 \]
(6-6)
multiplications and divisions are required.

Update of block row \( j \) in the \( i \)th step. Once a block row is factorised, all the following block rows must be updated. The blocks \( A_{i1}^{-1} \cdot (A_{i,i+1}, \ldots, A_{i,p}) \) are received from the processor containing the \( i \)th block row, and this must then be multiplied with the block matrix \( A_{ji} \), and added to the block row \( j \). This requires
\[ (p - i)s^3 \]
(6-7)
multiplications.

Critical communication in step \( i \). In the \( i \)th step the \( p - i \) updated blocks have to be sent from the processor containing row \( i \) to every processor containing rows with a higher number. This message contains
\[ (p - i)s^2 \]
double precision floating point numbers.
CHAPTER 6. LINEAR SYSTEM REDUCTION

It is important to note which operations are critical for the execution time of the algorithm. Here it is assumed that there are no zero blocks in the reduced system, so that there is no parallelism in the LU-decomposition of the whole reduced system. For simplicity, assume that processor \( i \) contains block row \( i \). In step \( i \) the factorisation of block row \( i \) is always critical, as is the update of the next row \( i + 1 \). Updating block rows \( j > i + 1 \) is not critical, as these rows subsequently have to wait for the update from the factorisation of the block row \( i + 1 \). From equations (6-5), (6-6) and (6-7) it follows that there are

\[
\sum_{i=1}^{p} \left\{ \frac{1}{3} (s^3 - s) + (p-i)s^3 + (p-i)s^3 \right\} \\
= p \frac{1}{3} (s^3 - s) + 2 \frac{p(p-1)}{2} s^3 \\
= (p^2 - 2p/3)s^3 + O(s)
\]

critical (i.e. sequential) operations. In the same way only the communication of the updated block row from processor \( i \) to processor \( i + 1 \) is critical. The total critical communication thus is

\[
\sum_{i=1}^{p} (p-i)s^2 = \frac{p(p-1)}{2} s^2
\]

double precision floating point numbers.

6.4.3 Comparison of NODE and BRANCH

Now it is possible to make a comparison between these two methods for the hypothetical conditions detailed above. The decomposition of diagonal blocks in the original system to compute either Schur complements or a reduced matrix, are ignored. These operations are fully parallel. The blocks will be slightly smaller for BRANCH than for NODE (as a result of which the reduced system in BRANCH is larger than the Schur complement in NODE), but usually the diagonal blocks are huge, and a few extra rows in some blocks do not make a significant difference to the completion time. Differences in block sizes are also ignored: these are a result of the algorithm used to compute the partitioning of the system, and will depend heavily on the problem concerned. Provided that the partitioning algorithms are equally ‘good’ in the sense that they provide partitions of approximately the same size and small sets of critical nodes, this will not give one algorithm a consistent advantage over the other.

To judge which algorithm is the best, the operations detailed in the previous two sections are the crux. These are only a small part of the total algorithm, but they determine the success of an algorithm, as they involve all the communication, as well as the critical section of sequential code. First of all a comparison of the number of critical operations is made.
As mentioned above, the block row size \( s \) is set to \( s = 2m/p \). Because of the use of \( O \)-notation, no exact number of processors can be given for which BRANCH requires fewer operations than NODE. To obtain an approximate number for \( p \) one must determine when

\[
(p^2 - 2p/3)(2m/p)^3 < m^3/3 + m^2 - m/3
\]

\[
(1 + 3/m - 1/m^2)p^2 - 24p - 16 > p^2 - 24p - 16 > 0
\]

\[
(p - 4\sqrt{10} - 12)(p + 4\sqrt{10} - 12) > 0
\]

\[
p \geq 25
\]

NODE thus requires fewer critical operations than BRANCH on anything up to \( p=25 \) processors. In reality \( p \) is smaller, as the coefficient to \( p^2 \) is strictly bigger than 1. On four processors BRANCH will require \( 5m^3/3 + O(m) \) compared to NODE's \( m^3/3 + O(m^2) \) divisions and multiplications in the critical part of the calculation.

The second important aspect is communication. To determine when BRANCH requires less critical communication than NODE one must determine

\[
p(p - 1)/2 (2m/p)^2 < pm^2
\]

\[
p^2 - 2p + 2 > 0
\]

The last equation is true for all values of \( p \), so that BRANCH always requires less communication than NODE. On four processors e.g. BRANCH will require the sending of \( 3m^2/2 \) double precision reals, while NODE requires the sending of \( 4m^2 \) double precision reals. To summarise:

<table>
<thead>
<tr>
<th></th>
<th>NODE</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Computation</td>
<td>( O(m^3) )</td>
<td>( O(s^3p^2) )</td>
</tr>
<tr>
<td>Critical Communication</td>
<td>( pm^2 )</td>
<td>( s^2p^2/2 )</td>
</tr>
</tbody>
</table>

On most modern machines, as e.g. the Intel iPSC/860, the processors are very fast, while the communication network is comparatively slow. The number of multiplications and divisions grows to just under 12 times as many (for \( p = 12 \)) for BRANCH than for NODE, but then the communication required for NODE is already more than 6 times the communication required for BRANCH. For all but very small values of \( p \) BRANCH thus seems preferable, especially if it is taken into account that the assumptions made initially were to the disadvantage of BRANCH. In particular the number of critical nodes presupposed here is more than is usually found in practice with BRANCH.

The results presented in chapter 8 will bear out the finding that BRANCH is much more efficient than NODE.
Chapter 7

Reordering systems

Reordering a linear system to block form is required for most algorithms discussed in this thesis. Such a reordering is achieved by tearing the associated graph, that is by removing nodes or branches from it, so that it falls into unconnected subgraphs. Unfortunately the optimal tearing of a graph is generally an NP-complete problem. Heuristics are thus used to obtain a partitioning of a graph. What exactly is meant by 'optimal' or 'good' will usually determine which heuristics are employed. Requirements for a good graph partitioning are detailed below.

In this chapter several approaches to graph partitioning are discussed. A generalisation of an algorithm by Sangiovanni-Vincentelli [117] is developed and shown to perform well on practical problems.

Recall the definition of an undirected graph on page 18. A few additional concepts are needed in this chapter.

Definition 7.1 Let \( G \) be a graph \( G = (V, E) \).

1. If \( u, v \in V \) and \( (u, v) \in E \) then \( u \) and \( v \) are called neighbours.

2. The number of vertices in a set \( U \subset V \) is denoted by \( |U| \).

3. The degree of a node \( v \in V \) is defined as \( |\{u \in V \mid (v, u) \in E\}| \), that is the number of edges originating on it.

4. A graph \( G \) is connected if there is a path connecting nodes \( v \) and \( u \) for all \( v, u \in V \), \( v \neq u \), where a path is a sequence \( < v, v_1, v_2, \ldots, v_n, u > \), \( (v_i \neq v_j) \) with all successive elements connected by an edge \( (v_i, v_{i+1}) \in E \) [49, p.18].

5. A maximal connected subgraph \( C_i = \{V_i, E_i\} \subset G \) is called a component or cluster. Here 'maximal' means that if \( v \in V_i \) is in the subgraph, then all nodes \( u \in V \), such that there is a path from \( v \) to \( u \) are also in the subgraph.
6. A branch cut set is a set of edges $\hat{E} \subseteq E$, such that the graph $\hat{G} = (V, E \setminus \hat{E})$ has at least one component more than $G$.

7. A nodal cut set is a set of nodes $\hat{V} \subset V$, such that the graph $\hat{G} = (V \setminus \hat{V}, \{(u, v) \in E \mid u, v \not\in \hat{V}\})$ has at least one component more than $G$.

8. A partitioning is a set of components of $G$, $\{C_1, \ldots, C_p\}$, such that $C_i \cap C_j = \emptyset$ and $\bigcup_i V_i = V$.

9. A numbering of a graph is the one-to-one assignment of the the numbers $1, \ldots, |V|$ to the nodes in $V$. The assignment of a number $i$ to a node $v$ is denoted by $v \to i$.

Foulds [49] only uses the term cut set for separating sets of branches, not nodes, but here the term will be used for both. Cut sets are also called separators [53]. It is worth noting that a nodal cut-set is not necessarily a component — in fact in most cases cut sets contain several components, most of which are single nodes.

In the definition of partitioning it is necessary that a graph has several components. As power systems are always connected the corresponding graph always consists of a single component, so that the partitioning consists only of this one component. But by removing cut sets from the graphs, new graphs are generated that do have a partitioning with several components. The goal of this chapter is to identify algorithms that generate 'good partitionings'. The definition of a 'good partitioning' varies according to the requirements of the algorithm, but is generally a partitioning generated by small cut sets, with all components having approximately the same size.

Recall that the vertices $v$ correspond to the variables in a linear system, while the edges $(v, u)$ correspond to the nonzero elements in the matrix. Take the linear system and its corresponding graph in figure 4-1 on page 88 as an example: variable $x_3$ corresponds to node $c$ and variable $x_7$ to node $g$, and as $A_{37}$ and $A_{73}$ are nonzero $(c, g) \in E$. In general: if $u \to i$ and $v \to j$, the matrix elements $A_{ij}$ and $A_{ji}$ are nonzero if and only if $(u, v) \in V$. As all the matrices are symmetrically structured only unordered tuples are considered. A reordering of a matrix thus corresponds to a renumbering of the nodes in a graph — the underlying graph does not change, but the matrix structure does.

### 7.1 Nested dissection

The classical way of partitioning graphs is the nested dissection algorithm, which was developed for finite difference methods, and is very successful on regular grids. Gilbert and Tarjan [53] analyse this algorithm for reordering sparse matrices resulting from finite element calculations in order to reduce fill.
Power systems are often planar graphs, in other words they can be embedded in the plane without any edges intersecting. For planar graphs Gilbert and Tarjan [ibid] prove a bound of $O(n \log n)$ fill for graphs with $n^{1/2}$-separators. A graph is said to satisfy an $f(n)$ separator theorem if $|V| = n$ and $V = A \cup B \cup C$, such that there are no edges connecting a node in $A$ to another in $B$, and $|A|, |B| \leq \alpha n$ and $|C| \leq \beta f(n)$. For planar graphs an $n^{1/2}$-separator theorem with $\alpha = \frac{2}{5}$ and $\beta = \sqrt{6}$ has been proven. For 100 nodes, this still means a separator of $\approx 25$ nodes.

Unfortunately nested dissection is not always very effective on irregular, sparse networks, even though it has recently regained some popularity for many problem domains for which it had previously been rejected. A simple nested dissection algorithm was implemented, but gave disappointing results. To attempt to explain why the algorithm performs badly, it is necessary to understand how it works.

Figure 7-1 shows the graph from figure 4-1 split into level sets. These are constructed with the following algorithm.

**Level Set Algorithm**

1. Choose a node with low degree, mark it, and put it into *level set* $L_1$; set $i = 1$.

2. For every node in $L_i$, select all its unmarked neighbours, mark them, and put them in $L_{i+1}$.

3. If there are still unmarked nodes left, set $i \leftarrow i + 1$, and repeat step 2.

In figure 7-1 the node $g$ is chosen as first node, and assigned to $L_1$, $f$ and $c$ are its neighbours, so they are assigned to $L_2$, etc. If any of the sets $L_2, L_3$ or $L_4$ are deleted, the graph would be partitioned, and the chosen level set would be the cut set. The algorithm is called nested dissection, because this process is then repeated on all components until they have reached some predefined size. In this example this algorithm works very well, as the middle level set $L_3$ would give a good separator. The effectiveness of the algorithm thus
### 7.2. The Kernighan & Lin Algorithm

Another idea is to partition the graph arbitrarily into connected components, and then improve the partitioning by exchanging elements between the components in such a way as to minimise the value of some objective function. Such an algorithm was developed by Kernighan and Lin [81]. The advantage of this approach is that the components in the partitioned graph will have a previously chosen size, which would mean good load balancing on a parallel machine. But there is no guarantee that the result will be a 'good' partitioning. Although this algorithm is fast, it was found that 'good' partitionings in power systems very rarely result in components of the same size.

Further clustering algorithms built around the idea of (locally) optimising an objective function are described below.

### Table 7-1: Statistics for level sets generated from real-life power systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of nodes</th>
<th>Number of Level Sets</th>
<th>Size of largest set</th>
<th>Position of largest set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee14</td>
<td>14</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>ieee30</td>
<td>30</td>
<td>7</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>ieee57</td>
<td>57</td>
<td>12</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>ieee118</td>
<td>118</td>
<td>15</td>
<td>18</td>
<td>11</td>
</tr>
<tr>
<td>pow236</td>
<td>236</td>
<td>17</td>
<td>32</td>
<td>11</td>
</tr>
<tr>
<td>pow529</td>
<td>529</td>
<td>19</td>
<td>78</td>
<td>12</td>
</tr>
<tr>
<td>pow795</td>
<td>795</td>
<td>20</td>
<td>115</td>
<td>13</td>
</tr>
<tr>
<td>pow1419</td>
<td>1419</td>
<td>20</td>
<td>252</td>
<td>13</td>
</tr>
</tbody>
</table>

depends on obtaining small level sets, ideally for sets $L_k$ such that the number of nodes $\sum_{j<k} |L_j|$ is roughly equal to $\sum_{j>k} |L_j|$.

However power systems do not generally have small level sets near the centre of the sequence of level sets. To demonstrate this feature table 7-1 lists the number and maximum sizes of level sets for several power systems. The largest level set is always large relative to the average size of the level sets. The table also gives the position of the largest set — which is invariably near the middle of the sequence of level sets. From this table it is clear that nested dissection cannot be effective for power system problems. This algorithm is clearly unsuited for such irregular graphs, and this view is supported by Duff et al. [32, p.258].
7.3 Elimination tree rotations

After reordering a sparse matrix with any fill-reducing algorithm, an elimination tree, as shown in figure 4-3, can be constructed efficiently [52, 32]. As was explained in chapter 4, the nodes corresponding to different subtrees of the elimination tree can be eliminated concurrently. Unfortunately many reordering methods (especially the minimum degree method) lead to unbalanced elimination trees, so that there is little scope for parallelisation. Liu [95] described an elimination tree rotation algorithm, with which a more balanced elimination tree can be constructed without changing the fill-structure. As both the minimum degree algorithm as well as the tree rotation algorithm are expensive, relative to the total solution time, this algorithm was not further considered. The solution time of a load flow analysis is short and a reordering algorithm must be very fast to keep the parallel methods competitive.

7.4 Spectral bisection

In the early seventies the idea to use the eigenvectors of an adjacency matrix to partition graphs was first proposed by Donath and Hoffman at IBM [27, 28]. In recent years these ideas have received much attention, due to their good global properties. Most proposed algorithms use spectral bisection recursively to partition a graph. In the description given here, the paper by Hendrickson and Leland [65] is followed. The idea is the following: index the nodes in \( V \) from 1 to \( n \) and define the adjacency matrix, such that \( A_{ij} = 1 \) if there is an edge from node \( i \) to node \( j \) and \( A_{ij} = 0 \) otherwise. Also define the degree matrix \( D = \text{diag}(d_i) \) where \( d_i \) is the degree of node \( i \). Assign a value \( x_i = \pm 1 \) to node \( i \), such that \( \sum_{V} x_i = 0 \). The assignment partitions the nodes into two sets, and the latter condition ensures that they have the same size (assuming \( n \) is even). The sum over the set of all edges \( f(x) = \frac{1}{4} \sum_{i<j, A_{ij}=1} (x_i - x_j)^2 \) counts the number of edges in the cut-set. This can be rewritten as

\[
\sum (x_i - x_j)^2 = \sum (x_i + x_j)^2 - \sum 2x_i x_j = x^T D x - x^T A x
\]

Then define the matrix \( L = D - A \) so that \( f(x) = \frac{1}{4} x^T L x \). With this define the discrete bisection problem

Minimise \( \frac{1}{4} x^T L x \)

Subject to: \( x^T 1 = 0 \)
\( x_i = \pm 1 \)
This is a hard problem to solve (graph partitioning is NP-hard), so the discreteness constraint on \( x_i \) is relaxed to give the continuous bisection problem.

\[
\begin{align*}
\text{Minimise} & \quad \frac{1}{4} x^T L x \\
\text{Subject to} & \quad x^T 1 = 0 \\
 & \quad x^T x = n
\end{align*}
\]

To continue the description, a theorem proven by Hendrickson and Leland [64] is required.

**Theorem 7.2** Denote the eigenvalues of the matrix \( L \) by \( \lambda_1 \leq \lambda_2 \leq \cdots \lambda_n \), and the corresponding normalised eigenvectors by \( u_1, \ldots, u_n \). Then

1. \( L \) is symmetric positive semi-definite
2. The \( u_i \) are pairwise orthogonal
3. \( u_1 = n^{-\frac{1}{2}} 1, \lambda_1 = 0 \)
4. if \( G \) is connected, then \( \lambda_1 \) is the only zero eigenvalue of \( L \).

Express \( x = \sum \alpha_i u_i \) where \( \alpha_i \in \mathbb{R} \) such that \( \sum \alpha_i^2 = n \). This is possible, as the \( u_i \) are orthogonal, and thus span \( \mathbb{R}^n \). Substitute for \( x \) and find that \( f(x) = \frac{1}{4}(\alpha_2^2 \lambda_2 + \cdots \alpha_n^2 \lambda_n) \).

As \( (\alpha_2^2 + \alpha_3^2 + \cdots + \alpha_n^2) \lambda_2 \leq \alpha_2^2 \lambda_2 + \cdots \alpha_n^2 \lambda_n \), \( f(x) \geq n \lambda_2 / 4 \). Notice that \( f(x) = n \lambda_2 / 4 \) can be achieved, by choosing \( x = \sqrt{n} u_2 \), and that this \( x \) satisfies the balance constraint since \( x^T 1 = u_2^T 1 = 0 \) (according to the theorem).

Consequently this choice of \( x \) solves the continuous minimisation problem and, provided that \( \lambda_3 \neq \lambda_2 \) the solution is unique.

The problem that remains is to map the solution of the continuous problem to that of the discrete problem. In the case of bisection find the median of the values \( x_i \), and map the vertices with corresponding \( x_i \) above the median to 1, and the others to -1. It follows immediately that \( n \lambda_2 / 4 \) is also a lower bound on the number of cuts that is required to partition the graph into two balanced sets, but this bound can underestimate the true number of cuts significantly.

To calculate such a balanced bisection, all that is needed is to calculate the second smallest eigenvalue, and the corresponding eigenvector of \( L \).

The bisection algorithm is then applied recursively to partition the graph into more components. Care must be taken that all the graphs to which this algorithm is applied are connected.

Hendrickson and Leland [65] generalise the basic bisection algorithm to partition a graph into 4, 8 or 16 sections in a single step. They also state that the partitioning can be locally
improved by applying a generalisation of the Kernighan and Lin algorithm. They report very good results for octasection with Kernighan and Lin post-processing.

The main reason that these algorithms are not further pursued, is, firstly, that it was found that in power systems balanced partitions are often far from optimal, and, secondly, that these are complex algorithms and Hendrickson and Leland’s cost estimates show that they are more expensive than the algorithm discussed next. A last objection is that there is no simple way to use these algorithms for node-tearing.

7.5 Clustering methods

After a lot of experimentation it was found that an algorithm devised by Sangiovanni-Vincentelli et al. [117] for branch tearing is effective. The idea underlying this algorithm is to grow clusters within a network in such a way that the number of branches that have to be removed to cut the cluster from the rest of the network remains small. A slightly more general version, that minimises some objective function when adding nodes to a cluster, is described here. Clearly the objective function depends on the definition of a ‘good’ partitioning of a graph. The advantage is that essentially the same algorithm can be used to compute branch as well as node tearing of graphs.

A parallel version of Sangiovanni-Vincentelli’s clustering algorithm on the AMT Distributed Array Processor, was implemented by Saleh and Laughton [116].

7.5.1 Branch tearing

A good partitioning through branch tearing does not necessarily mean that the number of branches in the cut sets is minimised. In the algorithms discussed in chapter 6 it was important that the sizes of the off-diagonal blocks (see figure 6-1 on page 115), caused by the torn branches, are minimised. This implies that the number of nodes connected by torn branches must be minimised. A good algorithm will:

**BT1.** Minimise the number of nodes in a cluster which are connected to nodes outside the cluster.

**BT2.** Minimise the number of nodes outside the cluster which have connections to nodes inside the cluster.

The second condition ensures that there are as few nodes as possible outside the cluster which are involved in connections. If only the first condition was used, other clusters might have large sets of nodes with external connections, even though the cluster under consideration has few nodes with external connections. This happens if nodes with a high
7.5. CLUSTERING METHODS

degree have few internal connections, but many external ones. The function that has to be minimised for a cluster \( C = (V_C, E_C) \) extracted from a set of nodes \( V \) with edge set \( E \) is thus

\[
\psi(C) = |\{u \in V_C | \exists v \in V \setminus V_C : (u, v) \in E\}| + |\{v \in V \setminus V_C | \exists u \in V_C : (u, v) \in E\}| \quad (7-1)
\]

This function is different from the one minimised by Sangiovanni-Vincentelli et al. [117]. Their objective was to minimise the number of branches connecting a cluster to the rest of the system, while here the desire is to keep the off-diagonal blocks as small as possible, without being bothered about the number of nonzero elements in these blocks.

7.5.2 Node tearing

For node tearing an objective function can be defined as well. It is desirable that

\textbf{NT.} The number of nodes that have to be removed to separate the cluster from the rest of the graph is minimal.

The function that should be minimised is

\[
\psi(C) = |\{u \in V \setminus V_C | \exists v \in V_C : (u, v) \in E\}| \quad (7-2)
\]

7.5.3 The clustering algorithm

Clustering algorithms work as follows: a node with low degree is chosen to start a cluster. Subsequently neighbours of any nodes in the cluster are added to the cluster, while attempting to keep the objective function \( \psi \) as small as possible. At any point the neighbours of all nodes in the cluster form a node cut set, while the edges to all the neighbours form a branch cut set.

The question is how to decide which neighbour to add to the cluster. As there are only a finite number of nodes, there is only a finite number of combinations in which nodes can be added to a cluster. These could all be enumerated and the best one could be chosen, but this would be prohibitively expensive. Instead the effect on the function \( \psi \) of adding a single node is considered, and the node leading to the (local) minimum value of \( \psi \) is chosen. This is a simple one-step look-ahead approach. It is of course possible to go one step further, and compute the effect of all combinations of two nodes on \( \psi(C) \). Experiments have shown that this gives better results in some cases, but it is relatively so expensive, that the slight improvement cannot be considered worthwhile.

The description of the algorithm requires three arrays (the notation used here is the same as in [117]):

\( IS \) in which the nodes belonging to a cluster are collected;
AS in which the nodes adjacent to a cluster are collected;

CN in which the value of the objective function for successive clusters is stored.

The parameters $M$ and $m$ indicate the maximal and minimal acceptable size of a cluster.

**Clustering Algorithm**

1. A node with a low degree is chosen as a starting node and stored in IS(1).

2. All the nodes which are adjacent to IS(1), and have not already been allocated to a cluster, form AS(1).

3. Define the subgraph $C = (\mathcal{W}, \mathcal{F}) := (\{IS(1)\}, \emptyset)$.

4. $CN(1) := \psi(C)$.

5. Let $i = 2$.

6. Choose a node $v \in AS(i - 1)$ and define $IS(i) := \{v\}$.

7. Update $AS(i)$ from $AS(i - 1)$ by deleting the node $v$ and adding all the nodes adjacent to $v$ that are not already in $AS(i)$ or any other cluster.

8. $CN(i) = \psi(C)$ with $\mathcal{W} := \bigcup_{j=1}^{k} IS(j)$ and $\mathcal{F} = \{(u, v) \in \mathcal{E} \mid u, v \in \mathcal{W}\}$.

9. If $|AS(i)| > 0$ and $|\mathcal{W}| < M$ then set $i := i + 1$, go to step 6.

10. Let $k$ be the index such that $CN(k) = \min_{j \in \{m, \ldots, M\}} CN(j)$. Define the cluster $C$ with $\mathcal{W} := \bigcup_{j=1}^{k} IS(j)$ and $\mathcal{F} = \{(u, v) \in \mathcal{E} \mid u, v \in \mathcal{W}\}$.

11. In the case of node tearing $AS(k)$ is the nodal cut set. All nodes in $\mathcal{W}$ and $AS(k)$ are ignored when constructing subsequent adjacency sets (step 7).

12. In the case of branch tearing all edges between nodes in $\mathcal{W}$ and $AS(k)$ are the cut-set for branch tearing. Only the nodes in $\mathcal{W}$ are ignored when constructing subsequent adjacency sets.

13. If there are still nodes left in the graph, go to step 1.

As $AS(i)$ always contains all the nodes that are connected to the subgraph defined by the vertices $\mathcal{W} := \bigcup_{j=1}^{k} IS(j)$, $AS(i)$ is obviously a cut set for the graph. Removal of the nodes in $AS(i)$ would cause one subgraph consisting of the nodes in $C$, and one or more subgraphs consisting of the nodes $\mathcal{V} \setminus (\mathcal{W} \cup AS(i))$.

Similarly if all edges between nodes in $C$ and $AS(i)$ are deleted, $C$ and $\mathcal{G} \setminus C$ would be disjoint.
7.5. CLUSTERING METHODS

Figure 7-2: Building the first cluster in the branch tearing algorithm applied to the graph from figure 4-1. Here \( m = 3 \) and \( M = 5 \).

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Step} & \text{IS} & \text{CN} & \text{AS} \\
\hline
1 & g & 1+2=3 & c, f \\
2 & c & 2+2=4 & f, a \\
3 & f & 2+1=3 & a \\
4 & a & 1+1=2 & d \\
5 & d & 1+2=3 & b, e \\
\hline
\end{array}
\]

\( C = \{ g, c, f, a \} \)

Cut-set: \( \{(a,d)\} \)

Figure 7-3: Building the first cluster in the node tearing algorithm applied to the graph from figure 4-1. Here \( m = 3 \) and \( M = 5 \).

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Step} & \text{IS} & \text{CN} & \text{AS} \\
\hline
1 & g & 2 & c, f \\
2 & c & 2 & f, a \\
3 & f & 1 & a \\
4 & a & 1 & d \\
5 & d & 2 & b, e \\
\hline
\end{array}
\]

\( C = \{ g, c, f \} \)

Cut set: \( \{a\} \)

As an example the determination of the first cluster in the branch and node-tearing of the graph from figure 4-1 is demonstrated in figures 7-2 and 7-3 respectively. As there are fewer than \( M \) nodes left after the first cluster is formed, the second cluster contains the remaining nodes in both cases.

In the node-tearing example \( \{d\} \) could also have been chosen as cut-set. It depends whether large or small components are desired: if larger clusters are desired the minimal value for \( \text{CN} \) closest to \( M \) is chosen, if small clusters are desired the value closest to \( m \) is chosen. Sometimes better partitions can be achieved by selecting a large range \([m, M]\), and then choosing a minimum value closest to some target size.

The one-step look-ahead method is used in these examples to choose the node to add in step 6 of the clustering algorithm. In step 3 of the branch-tearing algorithm either node \( f \) or node \( a \) could be added. If node \( f \) is added \( \text{CN}(3) = 3 \), if node \( a \) is added \( \text{CN}(3) = 4 \), so that node \( f \) is preferred. Often several nodes can be added that all lead to the same function value. Tie-breaking is important, but it did not prove possible to find a tie-breaking rule that worked well consistently, without doing further look-aheads. In practice the largest candidate, such that enough nodes are left in the graph to form another cluster of minimum size, is chosen.
<table>
<thead>
<tr>
<th>System</th>
<th>Partitions</th>
<th>Nodes/Partition</th>
<th>Nodes (Ext)</th>
<th>User Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee57</td>
<td>2</td>
<td>25</td>
<td>4</td>
<td>39 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>ieee118</td>
<td>2</td>
<td>49</td>
<td>4</td>
<td>51 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>69</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>37</td>
<td>4</td>
<td>82 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>48</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>pow236</td>
<td>3</td>
<td>64</td>
<td>6</td>
<td>125 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>pow529</td>
<td>3</td>
<td>236</td>
<td>2</td>
<td>270 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>179</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>114</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>pow795</td>
<td>3</td>
<td>236</td>
<td>2</td>
<td>387 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>294</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>265</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>236</td>
<td>2</td>
<td>383 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>179</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>153</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>227</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>pow1419</td>
<td>3</td>
<td>473</td>
<td>3</td>
<td>680 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>474</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>472</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>324</td>
<td>7</td>
<td>711 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>386</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>439</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>270</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Table 7-2: Sizes of partitionings for various test problems. The column *Nodes (Ext)* gives the number of nodes in every partition that have connections to nodes outside their own partition.

This one-step look-ahead algorithm is very fast, and produces reasonably good results. It is however not guaranteed to give an optimal partition. Statistics for branch tearing for the larger example systems are given in table 7-2. The results show that the sizes of the clusters do vary by up to 30%, but, more importantly, that the number of nodes with external connections is usually very small. This algorithm has thus achieved the aim of keeping the interconnection blocks small.

The complexity of the clustering algorithms is very low. If $K$ is the maximum number of neighbours of any cluster that is created (in power systems $K \ll |V|$) a maximum of $K$ look-aheads are needed in every step. As a node is added in every step, and $K$ can
be considered a constant much smaller than \(|V|\), we obtain a complexity of \(O(|V|)\) for
the clustering algorithms. The number of partitions that are desired does influence the
reordering time, as nodes are discarded from a cluster every time a partition is chosen, but
this does not change the complexity provided that \(M - m\) is a small constant. The results
in table 7-2 confirm this complexity.

7.5.4 PABLO

PABLO is a recent algorithm developed by O'Neil and Szyld [102]. Their algorithm is very
similar to the general clustering algorithm described above, the only real difference being
that the function they try to optimise is chosen differently. For a component \(C = \{V_C, E_C\}\)
they introduce the fullness \(\phi(C) = |E_C| / (|V_C|^2 - |V_C|)\), i.e. the ratio between the number of
edges in \(C\) and the number of edges in a fully connected component with the same number
of nodes as \(C\). They then grow clusters while trying to maximise \(\phi\). The only (slight)
drawback is that their algorithm does not include explicit control of block sizes, but this
could be added at the expense of keeping track of the values of the objective function as
described above.

7.6 Reordering a power system

Reordering a power system proceeds by initially determining the graph of the system,
that is constructing the adjacency matrix with some random ordering. Sparse matrices
are a convenient way of representing graphs when coding these algorithms. Using the
representation for sparse matrices specified in chapter 3.8, the nodes adjacent to node
\(v \rightarrow i\) are exactly the nodes corresponding to the column numbers in row \(i\). The degree
of a node is the equal to the number of nonzero off-diagonal elements in row \(i\), that is
\(\text{Row}(i+1) - \text{Row}(i) - 1\). It is also easy to keep track of the nodes that have been assigned to
clusters by maintaining a logical array flagging for every node whether it has been assigned
to a cluster or not.

Initially the system is partitioned with one of the clustering algorithms described above.
This is equivalent to a renumbering of the graph, and defining a partitioning \(P = \{P_1, \ldots, P_p\}\)
of the indices of the nodes, so that \(P_i\) contains the numbers of the nodes in cluster \(i\). In the
case of node tearing a partition \(P_{p+1}\) is added, which specifies all the nodes in the cut sets.

If a reduced algorithm is used, the nodes within every partition are again renumbered,
so that all nodes with edges in a cut-set are ordered last within that partition. This results
in a reduction ordering as described on page 113.

In the case of branch-tearing the clusters are reordered with respect to one another to
achieve a block elimination tree that is low and wide. This is done to create leading empty
off-diagonal blocks to take full advantage of any possible parallelism when computing the block-LU decomposition or using a block-SOR method, as mentioned in section 6.2.

Finally, in order to minimise fill inside every partition, all the partitions are reordered with the minimum degree algorithm. The minimum degree algorithm is expensive (requiring usually about twice as much time as the partitioning), but as it is executed independently on every partition, the reordering can be done in parallel.

For the direct methods requiring node tearing, that is the algorithms described in chapter 4, the reordering proceeds as

Cluster algorithm
(Node tearing) \quad \rightarrow \quad \text{Minimum degree algorithm}

For the branch tearing required for the block-iterative methods of chapter 5 the reordering proceeds as

Cluster algorithm
(Branch tearing) \quad \rightarrow \quad \text{Reordering blocks for parallelism} \quad \rightarrow \quad \text{Minimum degree algorithm}

Finally the reordering required for reduced methods proceeds as

Cluster algorithm
(Branch tearing) \quad \rightarrow \quad \text{Reduction ordering} \quad \rightarrow \quad \text{Reordering blocks for parallelism} \quad \rightarrow \quad \text{Minimum degree algorithm}
Chapter 8

Implementation and Comparison

Solvers for linear systems based on the algorithms discussed here were implemented for the Intel iPSC systems. These are MIMD systems with processors connected by a hypercube network. They are typical for current distributed memory machines in that they require a high computation to message passing ratio to be efficient. All algorithms discussed here are geared towards this type of architecture.

In the next sections several aspects of the implementation are discussed and in section 8.10.1 results are presented.

8.1 Routines implemented

A complete list of all the routines that were implemented is given in table 8-1. Sequential as well as parallel versions of all block algorithms were implemented. Sequential versions of the block algorithms for reduced systems were also implemented (a reduced system was computed and solved on a single processor). No parallel versions with ILU decomposition were implemented, as it is very difficult to implement a parallel ILU-decomposition and results on a single processor with ILU-preconditioning were not very encouraging.

8.2 Design of parallel programs

Parallel programs are notoriously difficult to write, debug and maintain. Especially for the algorithms described here, with complicated, non-homogeneous data movements involving several different data structures, the administration and debugging of data movements is a daunting task. To overcome this difficulty, information tables are constructed in every process, which describe its relationship with other processes. This information includes process and node numbers, the sizes of connecting sub-matrices, and pointers into reduced and full vectors to determine which data corresponds to which processor. The complex-
CHAPTER 8. IMPLEMENTATION AND COMPARISON

<table>
<thead>
<tr>
<th>Method</th>
<th>Preconditioning</th>
<th>Sequential</th>
<th>Red. Seq.</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU-decomposition (node tearing)</td>
<td>—</td>
<td>√</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>LU-decomposition (branch tearing)</td>
<td>—</td>
<td>√</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>Block-Jacobi</td>
<td>—</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Block-SOR</td>
<td>—</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>CGS</td>
<td>Block-Jacobi</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>CGS</td>
<td>Block-SOR</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>CGS</td>
<td>ILU(0)</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Bi-CGStab</td>
<td>Block-Jacobi</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Bi-CGStab</td>
<td>Block-SOR</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Bi-CGStab</td>
<td>ILU(0)</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

√ - implemented  × - not implemented

Table 8-1: List of the routines that were implemented.

<table>
<thead>
<tr>
<th>Vector length</th>
<th>Replicated on 4 processors</th>
<th>Distributed over 4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>0.22 ms</td>
<td>1.02 ms</td>
</tr>
<tr>
<td>10000</td>
<td>1.08 ms</td>
<td>1.32 ms</td>
</tr>
<tr>
<td>20000</td>
<td>2.18 ms</td>
<td>1.62 ms</td>
</tr>
</tbody>
</table>

Table 8-2: Scalar products computed sequentially and in parallel.

The locality of message passing can only be overcome by adhering to a uniform approach to data identification and message construction.

Synchronisation of a program over many processors is difficult to implement on an MIMD machine, leads to delays, and is usually unnecessary. A call to a global synchronisation routine is only used to provide a start for the timing of the routines. Control flow on different processors is otherwise purely data-driven, in other words events are triggered by the arrival of data.

An attempt was made to overlap computation and communication as much as possible. If it is known that the computation on another processor can only continue once a certain subset of data is available, this data is computed at the earliest possible time and sent immediately to the appropriate processor. At times this caused some slightly more elaborate programming, but it is unavoidable for the efficiency of parallel programs with message passing.

A difficulty in the design was the decision concerning which parts of a computation should be done in parallel, and which are better done sequentially. An example is the computation of scalar products required for Krylov-subspace algorithms. If the block rows
of the matrix are distributed over the processors, it is possible to keep only the corresponding section of every vector on every processor. For scalar products this distribution implies that sections of the vectors are multiplied and summed on every processor independently, upon which the results are exchanged and added. As a scalar product vectorises very well, the computation of the whole scalar product on an i860-processor is not significantly slower than computing part of a scalar product, and no communication is needed. As table 8-2 shows, there is a significant speed advantage for short vectors in duplicating scalar products on every node over distributing scalar products. As the vectors in this application are only the size of the reduced systems, it is significantly faster to duplicate the scalar products. This is even more true for DAXPY operations, as whole sections of vectors have to be exchanged. In the event it was decided to duplicate all vectors on all processors. This means that information is only exchanged between processors during the matrix vector multiplications needed for the CG methods.

As soon as the distribution of data and the mechanism of program control (data driven with a separate control flow on every processor versus a single control flow enforced with global communication calls) have been decided, the program can be designed. For a data driven program it is essential that the data-exchange is correct and unambiguous. In particular unambiguity can be quite difficult to enforce, as there is no control over the order in which messages arrive. The approach taken here was to design the data flow structure first and implement it as far as possible without doing any real computation. This leads to a program in which all communication calls are concentrated in a few routines, while most of the work is done in calls to standard sequential subroutines running independently on every processor. By splitting the problem in this way, it is possible to test the computational routines in the same way as on sequential computers and contain the difficulties of correct data exchange. Even so the integration of these routines usually leads to some unforeseen errors.

8.3 BLAS-routines and vectorisation

Extensive use was made of BLAS routines to ensure high performance [90, 30, 29, 24]. On the iPSC/860 the Kuck Mathematical Library provides optimised versions of all BLAS routines. Care was taken to vectorise all loops that allowed vectorisation.

In section 9.3 it will be seen that especially many operations during the initialisation of the matrices can be speeded up significantly by viewing all nonzero elements in the matrix as a long vector. This makes it possible to vectorise statements that would not benefit from vectorisation if every row of the matrix was viewed as a separate vector — many rows only have 6-10 nonzero elements. See also the experiments with vectorising a sparse
matrix-vector-multiplication in appendix C.

8.4 Duplication of code

In code that contains many similar operations there is always the problem that many routines are written that have nearly identical functions, performed on slightly different data structures. This very quickly turns into a maintenance head-ache, as a small change in one place may mandate changes to several routines. Furthermore, as the routines are written for specific purposes, the re-use of code is hampered.

In this project duplication of code was avoided wherever possible — there are for example only two LU-decomposition routines, one for the sparse and the other for the dense case. These are general enough to perform LU-decompositions within blocks in a matrix (also in the sparse case) and allow the user to define off-sets in column and row indices. Of course such generality comes at a cost: these routines are not as efficient as they could be for any specific operation.

The decision to write general purpose routines also proved to be favourable in a different respect: debugging parallel programs is very difficult and it is a great advantage to know that certain parts of the code are working correctly. As there are only two LU-decomposition routines the effort to test the LU-decomposition was significantly less than if 8 or 9 different, but more specific, routines would have had to be tested.

Another example where the duplication of code was avoided are the iterative routines. All scalar products and DAXPY operations are replicated on every processor, so that communication in these routines is only needed during the matrix vector multiplications. It was thus possible to use exactly the same routine for the sequential and parallel versions of the Block-Jacobi, Block-SOR, the CGS and Bi-CGSTab routines. Different versions of the matrix-vector-multiplication were needed, some of which were executed in parallel and required communication. When the parallel version of the iterative methods was written it was known that the fundamental algorithm was working correctly, and that all errors had to be caused by incorrect matrix-vector-multiplications. This is discussed in more detail in section 8.6.

8.5 Implementation of LU decomposition

The algorithms used for sparse LU-decomposition on a single processor, as described in section 3.8, are identical to the ones used on every node in the parallel implementation; the difference is that they operate on a sparse block row of the matrix, rather than the full matrix.
8.5. **IMPLEMENTATION OF LU DECOMPOSITION**

For the setting up of the administration vectors on every processor, it is important to note the following:

- It is possible to work with the structure of the admittance matrix only, and then expand every element to a $2 \times 2$ block.

- The symmetrical structure can be used to determine which rows have nonzero elements in the pivotal column on other processors.

Irving and Sterling [70] claim that the LU-decomposition of the Jacobian during load flow analysis is stable provided that $2 \times 2$ pivots are used, and a motivation for this was given in section 3.8.4. As noted in section 3.8, this assumption has far-reaching consequences — especially for the sparse algorithms:

- Reordering of the matrix can be done before doing any LU-decompositions and the structure including fill can be precomputed — a large advantage if many systems with the same structure have to be solved, as in Newton-raphson iterations.

- The LU-decomposition itself can be speeded up considerably:
  1. It is not necessary to search for an appropriate pivot (although stability is checked).
  2. Rows can be traversed in sets of 2, as it is known beforehand that they have the same structure.
  3. It is not necessary to check whether space was allocated for fill-in.

For the reduced system, as well as the Schur complement, dense data structures are used.

For the dense LU-decomposition the right-looking version is used, that is the pivot-row is multiplied with the inverse of the pivot, and this is the multiplied with the elements in the pivot column before updating the matrix. Two columns are always eliminated simultaneously due to the use of $2 \times 2$ pivots, so that the implementation provided with LAPACK could not be used. An update for the reduced system is demonstrated in figure 8-1. After computing the inverse of the pivotal $2 \times 2$ matrix, a first call to the BLAS-3 routine DGEMM, computes the factors with which the pivotal rows must be scaled, and a second call to DGEMH updates the complete block to the right and below the pivotal block. As the dense matrices do not tend to be very large, a version using larger blocks of pivots was not considered.

The direct methods solved the problems in all cases to a very high precision. If these direct methods are to be used for other network problems they may not perform quite as well, as there would be some overhead in the computation of the structure and the search
Figure 8-1: Elimination of a column in a dense block row in the reduced system: (1) the $2 \times 2$ pivot $A$ is inverted, and (2) the block row $(B_1, B_2, B_3)$ is multiplied with $A^{-1}$ with a call to DGEMM. Then (3) the block row $(A^{-1}B_1, A^{-1}B_2, A^{-1}B_3)$ is distributed to the processors containing the diagonal blocks for columns $B_2$ and $B_3$, so that these can update their part of the reduced matrix. Finally (4) $C$ is multiplied with $(A^{-1}B_1, A^{-1}B_2, A^{-1}B_3)$ and subtracted from $(D_1, D_2, D_3)$ with a second call to DGEMM.

for pivots. Markowitz type algorithms are quite expensive, and would cause considerable overhead. In the parallel case, this work will be done independently on the different partitions of the graph, so that these methods should still perform well; in fact the performance of the parallel version should improve significantly relative to the sequential version. Of course no stable decomposition can be guaranteed in the parallel case.

### 8.6 Implementation of iterative methods

In most cases the main part of the work in any iterative method are the matrix-vector-multiplications. As the matrices are very sparse, the matrix-vector-multiplications do not overshadow the rest of the calculations to the same extent as in more dense applications. The balance is readjusted slightly, because all the other operations needed for the CG-methods are vector-vector operations, which are fully vectorisable, while the matrix-vector-multiplications cannot be vectorised properly due to the irregular structure of the matrices. In appendix C efforts to vectorise matrix-vector-multiplication are described.

As mentioned already, the same piece of code for every iterative method was reused several times. This was made possible by hiding all the differences between the methods in an initialisation routine and the 'matrix-vector-multiplication'. The disadvantage of using preconditioned systems lies mainly in the fact that rounding errors may cause the preconditioned system to have a different solution from the original system. Take as an example block-Jacobi preconditioning: should one of the diagonal blocks in the system be
8.6. IMPLEMENTATION OF ITERATIVE METHODS

nearly singular, the system $D^{-1}Ax = D^{-1}b$ may have quite a different solution from $Ax = b$
in the presence of rounding errors. Such problems did not seem to occur in the tests that
were done — at the end of every run the residual of the original system was computed to
check this assertion.

To understand how preconditioning was used to simplify the code, consider as an ex-
ample the case of CGS with SOR preconditioning. When calling the CGS routine, a pro-
cedure $\text{MVM}$ is passed in as an argument. During the initialisation phase the CGS routine
calls $\text{MVM}$ with a flag set to Right-hand-side. In the SOR version of $\text{MVM}$ the right hand
side $(D + \omega L)^{-1}b$ is computed and returned. In every iteration $\text{MVM}$ is called twice with a
flag set to Matrix-Vector-Multiply. This causes the computation of $(I - (D + \omega L)^{-1}A)x$
as explained in section 5. Of course none of these inverses are computed explicitly, and all
calculations are done in sparse data structures.

The method using reduced systems computes the reduction before calling the CGS
routine. The routine $\text{MVM}$ computes the same quantities as before, but now only for the
reduced system and in dense data structures. It is irrelevant to the CGS routine whether
the matrix passed to it is in sparse or dense format, as the matrix is simply an array of
numbers to it.

More interesting is the parallel case. As mentioned already, it is advantageous to replic-
ate scalar products on all processors, rather than calculating them in parallel. Consequently
the CGS routine is duplicated on all processors, and only the matrix-vector-multiplications
run in parallel. As all the intermediate vectors are available on all the processors, the
CGS routine does not have to know whether it is running as part of a parallel program
or sequentially. Communication is only necessary to compute the updated right hand side
initially and then to compute the matrix-vector-multiplications. These must always return
the complete result of the matrix-vector-multiplication, and not just the part of the result
corresponding to the strip of matrix assigned to a specific processor. In the case of SOR
preconditioning the routine $\text{MVM}$ is fairly complicated, as precedence between processors has
to be taken into account. The routine is sophisticated enough to compute a processor’s re-
sult as soon as the relevant data is available, so that all possible parallelism in the forward
substitution is exploited. The different cases are detailed in figure 8-2.

One of the advantages of duplicating the iterative process on all the processors is that,
provided that all operations are rounded in the same way on all of them, the same residual
is available on all the processors. Thus all the processes will terminate at the same time,
without requiring any additional global communication.

In this program vectors often have to be distributed to all the processors in the system.
Intel provides a series of global operations across the hypercube to collect data from different
nodes and distribute them to all nodes. This can be done very efficiently in a hypercube,
but it does mean that all processors are synchronised during such an exchange. Another problem is that enough processes are needed to fill a complete subcube to make use of these operations. Dummy programs are thus always placed on all the nodes in a subcube that cannot be used by the algorithm. This reduces the time required for broadcasts at the cost of having a synchronisation point.

The parallel version of the iterative methods always uses a reduced system. As the reduced system is very small, this is not very efficient. The speedup in using a parallel method is attained mostly by parallelising the computation of the reduced system.

The appropriate weights, with which the norm of the residual must be multiplied to obtain the residual for the original system from that of the preconditioned system (section 5.4), are computed before calling the iterative routines. In the parallel case the weight is computed in parallel, and it is ensured that the same weight is used on all processors.

8.7 Testing and debugging

Testing such a large program (well over 10000 lines of code) is a very difficult task. The program was designed to be modular and every module was tested in isolation before inclusion in the main program. The main lesson that was learnt is that it is important to write
routines in as general a way as possible. Although performance may be impaired by this approach, the development of similar routines is avoided. This seems a price well worth paying.

As was mentioned before, the debugging of parallel programs is difficult. As no debugger was available a set of debugging routines was written. These print sections of matrices and vectors and transform data so that it is easier to analyse. Calls to these routines were inserted in the code within C-preprocessor statements, so that they can be switched on or off as needed.

A difficulty is the huge amount of output that parallel programs tend to produce. A useful approach is to let the different processors write to different files and then compare the files after a run.

A useful ‘tool’ in debugging the parallel programs was to create simple test data with known solutions, and writing serial programs which perform the same operations on the data as the parallel program. These could then be compared to the debugging output of the parallel program.

In all the tests the computed solutions were substituted in the linear system and the residual was computed. This showed whether the solution was acceptable.

8.8 Data

For a fair and realistic comparison of the algorithms discussed in this part, it is very important that the data used is realistic. In section 2.6 a set of power systems were described. Linear systems for testing the algorithms were obtained from one of the Newton-Raphson steps needed for the solution of the load flow problems posed by these test systems. A question is which iteration of the load flow solution provides a linear system that gives a realistic comparison between different algorithms. A few points to take into account are:

1. A linear system derived from the first Newton-Raphson iteration favours direct algorithms, as the initial systems are usually harder to solve — due to worse initial guesses — than later ones. This is because the systems are solved to a fixed precision and not a to a relative improvement.

2. A linear system derived from one of the last Newton-Raphson iterations favours iterative algorithms, as usually fewer iterations are needed to solve the later systems than the initial ones.

3. Another objection to choosing a system from one of the later iterations, is that this may favour iterative methods with cheap preconditioning. These may be very effective
with starting vectors close to the solution, but unable to solve the linear systems in the first iterations, and thus not a good choice for a general solver.

In the event, linear systems occurring in the second iteration of the load flow method were used. As most systems were solved in $4 - 6$ iterations, this seemed a good compromise.

A second, more difficult question, is to what precision the linear systems should be solved. If for example a truncated Newton method [4] is used, the required precision depends on additional information from the Newton method (the norm of the original power mismatch). In the event a rather large tolerance of $10^{-4}$ was chosen. This is sufficient as the nonlinear systems need not be solved to a very high precision, as the input data may contain large errors (up to 5% for high voltages). Even if a truncated Newton method is used, it is unlikely that linear systems will be solved to a much lower precision halfway through the Newton process.

As the results will show, these problems are not serious, as the comparison between the methods delivers clear cut conclusions.

For the parallel tests the admittance matrix was reordered with one of the clustering algorithms from chapter 7 before the load flow solution. So the linear system obtained from the Newton-Raphson process already has the appropriate structure, and consequently no reordering was needed for the solution of the linear systems. Both node and branch torn systems with varying numbers of partitions were produced. For the sequential Gaussian elimination and ILU(0) tests the system was only ordered with the minimum degree algorithm.

8.9 The iPSC/860

All the results reported in this chapter were obtained on an Intel iPSC/860 at the SERC's Daresbury Laboratories. Every node in this machine consists of an Intel i860 microprocessor running at 40 MHz. Every i860 contains a 64-bit RISC processor, a floating-point adder and multiplier, and respectively 8 and 16 KByte caches for instructions and data [69]. The processor's theoretical peak-rate — not to be attained — is 60 MFlops double precision. As the processor is heavily pipelined, vectorisation is essential to achieve good performance. The nodes are connected in a hypercube and the links are bi-directional pathways delivering data at a rate of 5.6 MBytes/second.

The interconnecting network is relatively slow and, combined with the high speed of the processors, results in an unbalanced system: communication is relatively much more expensive than computation. This means that it is quite difficult for parallel algorithms to compete with sequential ones, unless communication is either negligible or can be overlapped with computation.
8.10. TIMING

All programs are coded in FORTRAN-77 with calls to Intel's communication library for message passing. The if77 vectorising FORTRAN compiler from The Portland Group, was used with the following options

-O3 Level3 optimisation: All scalar optimisations, some global optimisation and software pipelining.

-Mvect=recog,transform Vectorisation of loops and high-level transformations such as loop splitting and interchanging. Loops are also strip-mined and streamed into on-chip cache.

Throughout the BLAS routines for the Intel, as provided in the Kuck Mathematical Library, were used.

8.10 Timing

Timing is always a contentious issue. The following assumptions were made:

1. The linear systems are solved as part of a process which requires the solution of many identically structured linear systems.

2. These linear systems are well enough conditioned that an LU-decomposition can be computed without pivoting.

3. The linear systems are constructed in place on the processors.

For load flow analysis these assumptions are true: an iterative process for the solution of a system of nonlinear equations requires the solution of a system of linear equations in every step, and the structure of all the Jacobians is determined by that of the underlying network; when using $2 \times 2$ pivots a stable decomposition is possible for all the linear systems in all the test cases; and the linear systems can be constructed in place on the processors quite easily.

The first assumption implies that the time needed to reorder the system to a suitable structure can be ignored, as the reordering has to be performed only once. As the systems can be decomposed without pivoting the final filled structure can be computed initially. Only the initial splitting of the system must be done sequentially, which is, using the algorithms described in section 7, very fast. In the case of branch tearing the reordering of blocks, to get maximum parallelism during the solution of the reduced system, must also be done sequentially, but is also very fast. The very expensive minimum degree reordering of every block can be done in parallel on the different processors, as can the computation of the filled structure. As the number of nodes per processor is much smaller in the parallel
case, the minimum degree ordering is much cheaper per processor than for the whole matrix (as needed in the sequential algorithms). Ignoring the reordering thus favours the sequential algorithms.

As the filled structure is computed only once, the nonzero elements of the linear system are written into a structure which will accommodate all fill, so that costly copying is avoided.

Furthermore it can be argued that ignoring pivoting favours the sequential algorithms as well, as looking for pivots is expensive. In the parallel case all the processors are looking for pivots simultaneously, and have far fewer matrix elements to search than in the sequential case. On the other hand, if a block row is singular the parallel algorithms break down, while the sequential algorithms may still be able to find a suitable pivot. In previous work on circuit simulation at Philips [76] singular block rows were only encountered when the underlying system was physically infeasible or very badly conditioned. Thus this is often a situation that should be reported to the user, rather than remedied. If the Jacobian is singular during the solution of systems of nonlinear equations it is possible to solve a related quadratic model for which the Jacobian is nonsingular [26, chap 6.5].

The computation of the Jacobian in-place on the different processors is realistic for a parallel algorithm (see part III). This means that the solution of the linear system can only be considered finished once the solution is available on all the nodes, as this is needed to determine the linear system for the next step. So the initial distribution of data to the nodes is ignored, as it is not required for every linear system solution, but the solution is only considered finished once the results are available on all nodes.

These assumptions do not give the direct methods an unfair advantage in their comparison with iterative methods, as the decomposition of at least the diagonal blocks is required for the preconditioners that make the iterative methods robust enough to solve the problems under consideration.

The methodology for timing the linear routines is now summarised.

The procedure for timing the sequential methods

1. The reordering of the system is excluded.
2. The timing is started once the system is reordered and set up.
3. The computation of the weights for the errors in the iterative methods is included.
4. The timing is terminated as soon as the complete solution has been computed.

The procedure for timing the parallel methods

1. The initial splitting and reordering of the system is excluded.
### Abbreviations for methods and preconditioners

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS</td>
<td>Gaussian elimination</td>
</tr>
<tr>
<td>B-JAC</td>
<td>Block-Jacobi method</td>
</tr>
<tr>
<td>B-SOR</td>
<td>Block-SOR method</td>
</tr>
<tr>
<td>CGS</td>
<td>Conjugate Gradient Squared method</td>
</tr>
<tr>
<td>BCGS</td>
<td>Bi-CGStab method</td>
</tr>
<tr>
<td>R-</td>
<td>Computation of the reduced system and subsequent application of the specified method.</td>
</tr>
<tr>
<td>NODE</td>
<td>Parallel direct solution of a node torn system</td>
</tr>
<tr>
<td>BRANCH</td>
<td>Parallel direct solution of a branch torn system</td>
</tr>
<tr>
<td>JAC</td>
<td>Jacobi preconditioning</td>
</tr>
<tr>
<td>2 × 2</td>
<td>Jacobi preconditioning with 2 × 2 blocks</td>
</tr>
<tr>
<td>B-JAC</td>
<td>Block Jacobi preconditioning</td>
</tr>
<tr>
<td>B-SOR</td>
<td>Block-SOR preconditioning</td>
</tr>
<tr>
<td>ILU(0)</td>
<td>Incomplete LU decomposition with 0 levels of fill</td>
</tr>
</tbody>
</table>

Table 8-3: Abbreviations for methods and preconditioners.

2. The distribution of the data to the processors is excluded, as it is assumed that the data will be generated on the different processors.

3. The computation of the filled data structures is excluded from the timing.

4. Computation of the block fill structure is excluded – this is necessary to determine the precedence of operations for forward substitution or LU-decomposition in the reduced system.

5. A call to the Intel library routine gsync() is used to synchronise all the processes to start the computation.

6. The computation of the weights by which the residuals of the preconditioned system have to be multiplied is included.

7. All other operations concerning the reduction of the system as well as the iterative procedures are included.

8. The time measurement is terminated when the solution is available on all processors (distributed through a call to the global function gco1).

### 8.10.1 Discussion of the results

A full set of results is given in tables 8-4 to 8-7. The abbreviations that are used are listed in table 8-3.
### Sequential methods (m236)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 (10^{10})</td>
<td>24 ms</td>
</tr>
<tr>
<td>B-JAC</td>
<td>—</td>
<td>—</td>
<td>50*</td>
<td>2.41 (10^{-1})</td>
<td>1.96 (10^{-2})</td>
<td>400 ms</td>
</tr>
<tr>
<td>B-SOR</td>
<td>—</td>
<td>—</td>
<td>50*</td>
<td>1.25 (10^{-4})</td>
<td>3.53 (10^{-6})</td>
<td>408 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>JAC</td>
<td>—</td>
<td>—</td>
<td>2.22 (10^{+2})</td>
<td>2.05 (10^{+2})</td>
<td>291 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>2 \times 2</td>
<td>—</td>
<td>50*</td>
<td>5.01 (10^{-1})</td>
<td>1.18 (10^{-1})</td>
<td>277 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>17</td>
<td>9.15 (10^{-6})</td>
<td>1.32 (10^{-6})</td>
<td>306 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-SOR</td>
<td>—</td>
<td>10</td>
<td>1.75 (10^{-5})</td>
<td>1.50 (10^{-5})</td>
<td>215 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>ILU(0)</td>
<td>—</td>
<td>26</td>
<td>8.53 (10^{-5})</td>
<td>7.72 (10^{-5})</td>
<td>454 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>JAC</td>
<td>—</td>
<td>50*</td>
<td>2.72 (10^{-2})</td>
<td>6.31 (10^{-3})</td>
<td>298 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>2 \times 2</td>
<td>—</td>
<td>50</td>
<td>2.84 (10^{-3})</td>
<td>1.30 (10^{-3})</td>
<td>286 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>16</td>
<td>8.49 (10^{-5})</td>
<td>3.45 (10^{-5})</td>
<td>287 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-SOR</td>
<td>—</td>
<td>11</td>
<td>2.94 (10^{-5})</td>
<td>1.94 (10^{-5})</td>
<td>225 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>ILU(0)</td>
<td>—</td>
<td>24</td>
<td>8.64 (10^{-5})</td>
<td>7.73 (10^{-5})</td>
<td>418 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>—</td>
<td>50*</td>
<td>2.33 (10^{+0})</td>
<td>1.96 (10^{-2})</td>
<td>351 ms</td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>—</td>
<td>—</td>
<td>50*</td>
<td>1.37 (10^{-3})</td>
<td>3.53 (10^{-6})</td>
<td>354 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>19</td>
<td>1.37 (10^{-6})</td>
<td>2.86 (10^{-6})</td>
<td>337 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>—</td>
<td>11</td>
<td>2.71 (10^{-5})</td>
<td>2.00 (10^{-7})</td>
<td>312 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>17</td>
<td>4.41 (10^{-5})</td>
<td>7.62 (10^{-7})</td>
<td>328 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>—</td>
<td>11</td>
<td>3.69 (10^{-5})</td>
<td>1.44 (10^{-7})</td>
<td>311 ms</td>
</tr>
</tbody>
</table>

### Parallel methods on three processors (m236.3)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 (10^{+0})</td>
<td>55 ms</td>
</tr>
<tr>
<td>BRANCH</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 (10^{+0})</td>
<td>29 ms</td>
</tr>
<tr>
<td>B-JAC</td>
<td>—</td>
<td>—</td>
<td>40*</td>
<td>6.32 (10^{-1})</td>
<td>7.30 (10^{-2})</td>
<td>93 ms</td>
</tr>
<tr>
<td>B-SOR</td>
<td>—</td>
<td>1.55</td>
<td>24</td>
<td>6.41 (10^{-5})</td>
<td>3.42 (10^{-7})</td>
<td>81 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>17</td>
<td>3.20 (10^{-5})</td>
<td>2.85 (10^{-7})</td>
<td>86 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.55</td>
<td>13</td>
<td>1.61 (10^{-5})</td>
<td>1.49 (10^{-7})</td>
<td>90 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>16</td>
<td>2.56 (10^{-5})</td>
<td>3.54 (10^{-7})</td>
<td>79 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.55</td>
<td>14</td>
<td>1.20 (10^{-5})</td>
<td>1.56 (10^{-7})</td>
<td>94 ms</td>
</tr>
</tbody>
</table>

Table 8-4: Results for linear methods on the iPSC/860 for the test system m236 with 472 variables. (* no convergence)

All sequential methods were run on a single processor of the iPSC/860, while all the parallel methods were run on three or seven processors. The reason for having odd numbers of processors is that one processor is reserved for reading in, reordering and partitioning the system, as well as collecting the results. Due to memory constraints and a limit of one process per processor, it would have been difficult to use this processor in the calculation. These are problems that can be overcome, but for simplicities sake it was decided to leave this processor idle during the parallel computation.

The sole exception is the direct method for node-torn systems (NODE), which was
### 8.10. TIMING

#### Sequential methods (m529)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 10^{-6}</td>
<td>52 ms</td>
</tr>
<tr>
<td>B-SOR</td>
<td>—</td>
<td>1.42</td>
<td>15</td>
<td>5.50 10^{-5}</td>
<td>2.23 10^{-6}</td>
<td>373 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>341 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>352 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>ILU(0)</td>
<td>—</td>
<td>37</td>
<td>8.34 10^{-5}</td>
<td>1.31 10^{-5}</td>
<td>1433 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>7</td>
<td>8.36 10^{-7}</td>
<td>7.08 10^{-8}</td>
<td>330 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>7</td>
<td>1.54 10^{-5}</td>
<td>4.47 10^{-7}</td>
<td>340 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>ILU(0)</td>
<td>—</td>
<td>30</td>
<td>8.07 10^{-5}</td>
<td>1.97 10^{-5}</td>
<td>1139 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>—</td>
<td>40*</td>
<td>1.35 10^{-1}</td>
<td>1.76 10^{-3}</td>
<td>1433 ms</td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>—</td>
<td>1.42</td>
<td>16</td>
<td>6.98 10^{-5}</td>
<td>5.04 10^{-7}</td>
<td>1419 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>1417 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>1419 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>1416 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>7</td>
<td>1.00 10^{-10}</td>
<td>0.00 10^{+0}</td>
<td>1418 ms</td>
</tr>
</tbody>
</table>

#### Parallel methods on three processors (m529.3)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 10^{+0}</td>
<td>56 ms</td>
</tr>
<tr>
<td>BRANCH</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 10^{+0}</td>
<td>28 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>—</td>
<td>40*</td>
<td>1.35 10^{-1}</td>
<td>1.76 10^{-3}</td>
<td>55 ms</td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>—</td>
<td>1.42</td>
<td>16</td>
<td>6.98 10^{-5}</td>
<td>5.04 10^{-7}</td>
<td>43 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>40 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>43 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>7</td>
<td>0.00 10^{+0}</td>
<td>0.00 10^{+0}</td>
<td>41 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>7</td>
<td>1.00 10^{-10}</td>
<td>0.00 10^{+0}</td>
<td>43 ms</td>
</tr>
</tbody>
</table>

Table 8-5: Results for linear methods on the iPSC/860 for the test system m529 with 1058 variables.

Run on four processors: three for the subsystems and a fourth one to compute the Schur complement. In principal the process which computes the update and decomposition of the Schur complement could be run on any of the other processors, but, as the iPSC/860 only allows a single process to run on every node, this is very difficult to organise. Such a NODE calculation is also considered to run on three processors.

Table 8-4 gives results for the standard point-Jacobi preconditioning, as well as Jacobi preconditioning using $2 \times 2$ blocks. As expected, convergence is quite slow for these methods, and they were abandoned for the larger systems.

It is notable how quickly the block-iterative and block-preconditioned methods converge. The block-preconditioned methods are not only better suited for parallelisation, but, for these problems at least, they are also consistently faster than the ILU(0) preconditioned methods. These results, combined with the difficulty of implementing a parallel version of
### Sequential methods (m795)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1.00 $10^{-10}$</td>
<td>71 ms</td>
</tr>
<tr>
<td>B-SOR</td>
<td>—</td>
<td>1.52</td>
<td>20</td>
<td>6.05 $10^{-5}$</td>
<td>2.09 $10^{-6}$</td>
<td>728 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-JAC</td>
<td>1.42</td>
<td>13</td>
<td>0.00 $10^{+0}$</td>
<td>0.00 $10^{+0}$</td>
<td>827 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>10</td>
<td>1.89 $10^{-5}$</td>
<td>2.40 $10^{-6}$</td>
<td>712 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>ILU(0)</td>
<td>40*</td>
<td>4.58</td>
<td>6.04 $10^{-4}$</td>
<td>2199 ms</td>
<td></td>
</tr>
<tr>
<td>BCGS</td>
<td>B-JAC</td>
<td>1.42</td>
<td>10</td>
<td>6.48 $10^{-5}$</td>
<td>5.56 $10^{-6}$</td>
<td>674 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>11</td>
<td>2.23 $10^{-5}$</td>
<td>1.24 $10^{-6}$</td>
<td>742 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>ILU(0)</td>
<td>40*</td>
<td>5.76</td>
<td>3.41 $10^{-5}$</td>
<td>2388 ms</td>
<td></td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>40*</td>
<td>1.32</td>
<td>1.58 $10^{-2}$</td>
<td>2389 ms</td>
<td></td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>1.42</td>
<td>40*</td>
<td>1.02</td>
<td>2.31 $10^{-7}$</td>
<td>2391 ms</td>
<td></td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>13</td>
<td>0.00</td>
<td>0.00 $10^{-10}$</td>
<td>2377 ms</td>
<td></td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>11</td>
<td>1.17 $10^{-5}$</td>
<td>6.29 $10^{-8}$</td>
<td>2376 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>13</td>
<td>0.00</td>
<td>0.00 $10^{-10}$</td>
<td>2377 ms</td>
<td></td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>11</td>
<td>3.44 $10^{-5}$</td>
<td>8.06 $10^{-8}$</td>
<td>2375 ms</td>
</tr>
</tbody>
</table>

### Parallel methods on three processors (m795.3)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>104 ms</td>
</tr>
<tr>
<td>BRANCH</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>39 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>40*</td>
<td>1.32</td>
<td>1.58 $10^{-2}$</td>
<td>69 ms</td>
<td></td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>1.52</td>
<td>23</td>
<td>4.64</td>
<td>1.08 $10^{-7}$</td>
<td>68 ms</td>
<td></td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>13</td>
<td>0.00</td>
<td>0.00 $10^{-10}$</td>
<td>64 ms</td>
<td></td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>11</td>
<td>1.17 $10^{-5}$</td>
<td>70 ms</td>
<td></td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>13</td>
<td>0.00</td>
<td>0.00 $10^{-10}$</td>
<td>63 ms</td>
<td></td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>11</td>
<td>3.51 $10^{-5}$</td>
<td>68 ms</td>
<td></td>
</tr>
</tbody>
</table>

### Parallel methods on seven processors (m795.7)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>457 ms</td>
</tr>
<tr>
<td>BRANCH</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>69 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>40*</td>
<td>1.92</td>
<td>4.95 $10^{-3}$</td>
<td>168 ms</td>
<td></td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>1.52</td>
<td>40*</td>
<td>1.36</td>
<td>1.34 $10^{-3}$</td>
<td>334 ms</td>
<td></td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>27</td>
<td>3.02</td>
<td>3.17 $10^{-8}$</td>
<td>226 ms</td>
<td></td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>18</td>
<td>3.42 $10^{-5}$</td>
<td>317 ms</td>
<td></td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>26</td>
<td>2.36</td>
<td>3.03 $10^{-7}$</td>
<td>215 ms</td>
<td></td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.42</td>
<td>15</td>
<td>1.57 $10^{-5}$</td>
<td>263 ms</td>
<td></td>
</tr>
</tbody>
</table>

Table 8-6: Results for linear methods on the iPSC/860 for the test system m795 with 1590 variables. (* = no convergence)
8.10. TIMING

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{-6}$</td>
<td>118 ms</td>
</tr>
<tr>
<td>B-SOR</td>
<td>—</td>
<td>1.45</td>
<td>18</td>
<td>6.54 $10^{-5}$</td>
<td>1.87 $10^{-6}$</td>
<td>1193 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>13</td>
<td>0.00 $10^{+0}$</td>
<td>0.00 $10^{+0}$</td>
<td>1510 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>B-SOR</td>
<td>1.45</td>
<td>9</td>
<td>1.68 $10^{-5}$</td>
<td>1.11 $10^{-6}$</td>
<td>1228 ms</td>
</tr>
<tr>
<td>CGS</td>
<td>ILU(0)</td>
<td>—</td>
<td>33</td>
<td>3.38 $10^{-5}$</td>
<td>4.53 $10^{-5}$</td>
<td>3209 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>9</td>
<td>6.73 $10^{-5}$</td>
<td>3.34 $10^{-6}$</td>
<td>1166 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>B-SOR</td>
<td>1.45</td>
<td>8</td>
<td>8.29 $10^{-5}$</td>
<td>5.74 $10^{-6}$</td>
<td>1109 ms</td>
</tr>
<tr>
<td>BCGS</td>
<td>ILU(0)</td>
<td>—</td>
<td>26</td>
<td>3.90 $10^{-5}$</td>
<td>3.88 $10^{-5}$</td>
<td>2538 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>208 ms</td>
</tr>
<tr>
<td>BRANCH</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>58 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>—</td>
<td>40*</td>
<td>6.98 $10^{-2}$</td>
<td>6.96 $10^{-4}$</td>
<td>108 ms</td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>—</td>
<td>1.45</td>
<td>20</td>
<td>8.15 $10^{-5}$</td>
<td>4.08 $10^{-7}$</td>
<td>89 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>12</td>
<td>5.27 $10^{-5}$</td>
<td>5.81 $10^{-7}$</td>
<td>91 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.45</td>
<td>9</td>
<td>2.23 $10^{-5}$</td>
<td>1.39 $10^{-7}$</td>
<td>90 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>12</td>
<td>3.11 $10^{-6}$</td>
<td>4.03 $10^{-8}$</td>
<td>89 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.45</td>
<td>9</td>
<td>3.60 $10^{-5}$</td>
<td>2.18 $10^{-7}$</td>
<td>88 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Precon.</th>
<th>Omega</th>
<th># Iter</th>
<th>Error (est.)</th>
<th>Error (real)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>406 ms</td>
</tr>
<tr>
<td>BRANCH</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.00 $10^{+0}$</td>
<td>60 ms</td>
</tr>
<tr>
<td>R-B-JAC</td>
<td>—</td>
<td>—</td>
<td>40*</td>
<td>2.30 $10^{-1}$</td>
<td>4.25 $10^{-3}$</td>
<td>180 ms</td>
</tr>
<tr>
<td>R-B-SOR</td>
<td>—</td>
<td>1.45</td>
<td>40*</td>
<td>2.04 $10^{-2}$</td>
<td>1.81 $10^{-4}$</td>
<td>302 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-JAC</td>
<td>—</td>
<td>24</td>
<td>2.94 $10^{-6}$</td>
<td>3.88 $10^{-8}$</td>
<td>220 ms</td>
</tr>
<tr>
<td>R-CGS</td>
<td>B-SOR</td>
<td>1.45</td>
<td>14</td>
<td>2.31 $10^{-5}$</td>
<td>1.32 $10^{-7}$</td>
<td>233 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-JAC</td>
<td>—</td>
<td>23</td>
<td>2.73 $10^{-5}$</td>
<td>4.16 $10^{-7}$</td>
<td>209 ms</td>
</tr>
<tr>
<td>R-BCGS</td>
<td>B-SOR</td>
<td>1.45</td>
<td>14</td>
<td>3.29 $10^{-5}$</td>
<td>2.12 $10^{-7}$</td>
<td>227 ms</td>
</tr>
</tbody>
</table>

Table 8-7: Results for linear methods on the iPSC/860 for the test system m1419 with 2838 variables.

ILU(0), were the reason for only providing times for a serial version of ILU(0).

Estimates for the error in the iterative methods are compared to the ‘real’ error as found by explicitly computing $||r||_\infty = ||b - Ax_s||_\infty$ with $x_s$ the computed solution. At the start of the iterative method a weight $||M||_\infty$ is computed, as described in section 5.4, and subsequently all estimates of residuals generated by the method are multiplied with this weight. These estimates are within two orders of magnitude, and consistently overestimate the error, for all methods except those using ILU(0) preconditioning. For ILU(0) preconditioning the weight had to be adjusted by between $10^{-5}$ and $10^{-7}$ to get good estimates for the error. $||\hat{L}||_\infty$ is evidently a severe over-estimate of the weight — which is not all that
surprising as a $\|\hat{L}\|_\infty$ is an absolute row sum of a triangular matrix.

The reduced methods are given as reference — they are exactly the parallel iterative methods executed on one processor. After the reduction step a copy from sparse to dense data structures is required, so that the reduced system can be solved in dense data structures; this proved to be a serious bottle-neck. For the power system pow795 (table 8-6) fully 2338 ms was spent in the reduction step and the reduced system was then solved in between 37 ms for Bi-CGSTab with B-SOR preconditioning and 51 ms for B-JAC. Due to memory constraints no results for the reduced system could be obtained for the system pow1419.

The impressive speed-ups found for the parallel iterative methods are thus mainly due to the use of reduced systems. The large amount of work required in the reduction is fully parallel, and it is not necessary to repeat this work in every iteration as with the sequential methods on the full system.

For the system pow236 sequential Gaussian elimination (GAUSS) performs best, but for all the larger systems the parallel direct method for branch torn matrices (BRANCH) is the fastest. As expected the parallel iterative methods are slower than BRANCH, as the work to calculate the reduced system is the same, but the iterative method subsequently require much more communication. After the theoretical discussion in section 6.4 it was concluded that BRANCH may be faster than NODE due to a decrease in the amount of critical communication. The results indicate that this is indeed the case. It was also conjectured that BRANCH should scale better than NODE, which is confirmed by the results in figure 8-3. In figure 8-3 the sizes of various quantities in the systems are also given.

The correspondence between the models constructed in section 6.4 for communication costs and the real results is now examined. The values for the models and the real timings for two linear systems on increasing numbers of processors are listed below. The values $pm^2$ are the number of processors times the size of the Schur complement squared, i.e. the model for the amount of communication needed for the NODE method. The values $s^2p(p - 1)/2$, with $s$ the average size of the part of the reduced matrix on every processor, and $p$ the number of processors, is the model for the amount of communication needed for the BRANCH method. The results are given in table 8-8 on page 158.

Although the full reduction step was included in these timings, there is a high correlation between the theoretical measures for the amount of communication and the actual execution times, as shown in table 8-9. These results clearly indicate the dominance of the communication in these calculations. Of course this strong correlation may change on systems with a better balance between the communication and computation speeds, but current architectural developments point in the opposite direction, as processors speeds are increasing much faster than communication speeds. On the other hand the estimates for
the communication are also a measure for the sequential part of the method, that is the solution of the reduced system over several processors, and certainly for larger numbers of processors, a fairly high correlation is to be expected, as the reduction of the system is fully parallel and always very fast.
Chapter 8. Implementation and Comparison

Results for the linear system m795

<table>
<thead>
<tr>
<th>#Proc</th>
<th>pm²</th>
<th>s²p(p - 1)/2</th>
<th>Result (in msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32</td>
<td>361</td>
<td>53</td>
</tr>
<tr>
<td>3</td>
<td>3072</td>
<td>225</td>
<td>104</td>
</tr>
<tr>
<td>4</td>
<td>3136</td>
<td>1014</td>
<td>171</td>
</tr>
<tr>
<td>5</td>
<td>6480</td>
<td>1960</td>
<td>244</td>
</tr>
<tr>
<td>6</td>
<td>13824</td>
<td>1500</td>
<td>183</td>
</tr>
<tr>
<td>7</td>
<td>23548</td>
<td>4459</td>
<td>457</td>
</tr>
</tbody>
</table>

Results for the linear system m1419

<table>
<thead>
<tr>
<th>#Proc</th>
<th>pm²</th>
<th>s²p(p - 1)/2</th>
<th>Result (in msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>968</td>
<td>16</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>10092</td>
<td>901</td>
<td>208</td>
</tr>
<tr>
<td>4</td>
<td>13456</td>
<td>5954</td>
<td>154</td>
</tr>
<tr>
<td>5</td>
<td>16820</td>
<td>1960</td>
<td>271</td>
</tr>
<tr>
<td>6</td>
<td>32856</td>
<td>240</td>
<td>180</td>
</tr>
<tr>
<td>7</td>
<td>34300</td>
<td>3471</td>
<td>406</td>
</tr>
</tbody>
</table>

Table 8-8: Theoretical measures and real performance of the direct methods.

<table>
<thead>
<tr>
<th>System</th>
<th>pm² w.</th>
<th>s²p(p - 1)/2 w.</th>
<th>(s²p(p - 1)/2 - pm²) w.</th>
</tr>
</thead>
<tbody>
<tr>
<td>m795</td>
<td>0.9004</td>
<td>0.7244</td>
<td>0.8700</td>
</tr>
<tr>
<td>m1419</td>
<td>0.6447</td>
<td>0.7198</td>
<td>0.6827</td>
</tr>
</tbody>
</table>

Table 8-9: Correlation between the theoretical measures and the real performance.

8.11 Event graphs for the direct algorithms

In order to get a better overview over the efficiency of the different algorithms a performance analysis program was used. This allows the user to specify events which are logged. In figures 8-4 and 8-5 the event graphs for the LU decomposition with the BRANCH and NODE algorithms are shown. The numbers indicate flagged events and the grey bars indicate idle processors. In figure 8-5 it can be seen clearly how all other processors have to wait for processor zero to factorise the Schur complement, and how the back substitution is then done simultaneously by all processors. In figure 8-4 processor zero does nothing except collect the results at the end. But here the activity pattern is very similar to that in figure 8-5, except that now processors 1 and 2 wait for processor 3 to finish its calculation. Noting that the block with connections between processors 1 and 2 is empty, there is no obvious reason why the algorithm BRANCH should perform any better than NODE.

The answer to this puzzle is that the superior performance of the one method over the other has nothing to do with algorithmic differences, but only with the partitioning of
8.11. EVENT GRAPHS FOR THE DIRECT ALGORITHMS

Figure 8-4: Events during the solution of the linear system m795 on three processors with the algorithm BRANCH. The events are: 1. Termination of the whole calculation; 2. Reduction of the linear system; 3. Solution of the reduced system; 4. Computing the update to send to other processors; 5. Parallel back substitution. The shaded areas indicate when processors were idle.

the underlying power system. The structures of the matrices are given in appendix E in figures E-2 and E-3 respectively. Not only is the partitioning achieved with branch-tearing much better balanced, but the reduced system is also smaller than the Schur complement generated for NODE. This is mirrored in the activity graph: processor 1 spends much longer on the LU-decomposition (between tags 2 and 3) than processors 2 or 3 in algorithm NODE, while the decomposition in algorithm BRANCH requires about the same amount of time on all processors.

The real advantage of the algorithm BRANCH is expected on more processors. To examine this case the same algorithms were executed on seven processors. The matrix structures are again given in appendix E in figures E-4 and E-5 respectively. The partitioning of the branch-teared matrix seems a lot less favourable than that of the node-teared matrix: its largest block contains 388 elements as opposed to 358, and the reduced system has dimension 102 as opposed to 58 for the Schur complement. But still the algorithm BRANCH performs
much better than NODE. Here the enhancement made to the BRANCH algorithm of ordering unconnected blocks first has significant impact. The first four blocks of the system are not connected to one another and, as can be seen from the event diagram in figure 8-6, the updates from processors 1 to 4 can be sent immediately (bracketed by events 4) when the solution of the reduced system (bracketed by events 3) commences. The gaps in the idle-time on processors 4 to 7 between events 3 and 4 indicate when updates arrive from other processors and are incorporated. There are times when all processors are idle, and this indicates times during which messages are begin transmitted between processors.

The severity of the bottle-neck in the case of NODE is plain to see in figure 8-7: all seven processors fall idle and wait for processor 0 to factorise the Schur complement, before the back substitution can commence. Initially all processors have to wait for the processor with the largest block assigned to it, to finish its LU-decomposition. In this example the
largest block (with 358 variables) was assigned to processor 3 (note that this does not agree with the order of the blocks in figure E-5; after looking at the information for individual processors it was concluded that the tool that generates the event graph numbers the processors differently). This event graph indicates that the same execution time could have been achieved on only four processors, by placing multiple processes on the same processor. So the unbalanced partitioning is partly to blame for the bad performance in this case — but a glance at figure 8-6 shows that the partitioning of the branch-torn matrix was equally unbalanced, and still BRANCH performed well.

8.12 Conclusions

In the last few chapters several methods for solving linear systems derived from networks were discussed. It was shown that there are parallel algorithms that can be used effectively on such problems, even though the structure of the problems is very irregular.

Initially a direct method, which is widely used for double-bordered block-structured systems, was described, but this method contains a severe bottle-neck that prevents it from scaling well. Subsequently it was shown that iterative methods can be used on such problems, provided that a good preconditioner is used. These preconditioners are expensive, and iterative methods are slower than the direct methods.

The discussion of iterative methods led to the idea of reduced systems, which provided a way of speeding up iterative methods with block-preconditioning. This in turn led to a new direct method, which proves to scale better and is generally faster than the direct method described initially. The fact that a method based on branch-tearing is more efficient than those based on node-tearing is not intuitive, and questions the current pre-dominance of node-tearing methods in parallel computing.

This method holds much promise for large systems, as results on up to 7 processors show that it scales much better than the older methods. It seems likely that, provided that the underlying problem is large enough, tens of processors could be used effectively. In the next part of this thesis, which examines the solution of the whole load flow problem, this method will be used in conjunction with the Newton-Raphson method.
Figure 8-6: Events during the solution of the linear system m795 on seven processors with the algorithm BRANCH. The events are: 1. Termination of the whole calculation; 2. Reduction of the linear system; 3. Solution of the reduced system; 4. Computing the update to send to other processors; 5. Parallel back substitution. The shaded areas indicate when processors were idle.
Figure 8.7: Events during the solution of the linear system $m795$ on seven processors with the algorithm NODE. The events are: 1. Termination of the whole calculation; 2. Decomposition of a diagonal block; 3. Extracting and sending the dense Schur complement; 4. Back substitution; 5. Start of updating the Schur complement of the whole system; 6. Solution of the Schur system. The shaded areas indicate when processors were idle.
Part III

Parallel methods for nonlinear systems
In part II of this thesis several methods for the parallel solution of linear systems of equations are examined. The goal of that work is to speed up the solution of the linear systems that have to be solved in every iteration of the Newton-Raphson process. However, it turns out to be very difficult to construct efficient parallel algorithms for the solution of the linear systems due to their difficult structure. In part III methods are investigated that solve the nonlinear systems in parallel, instead of having a sequential linearisation process with parallel solution of the nonlinear systems.

As a benchmark, the Newton method in conjunction with the best linear parallel algorithm found in part II (BRANCH), is applied to the load flow problem in chapter 9. Using the full Newton method is attractive as the classical convergence theory is valid. It is known how to make the Newton method more robust and it is an algorithm trusted by practitioners. The implementation presented here provides real speed-ups compared to an efficient single-processor algorithm, which is remarkable on these difficult problems.

However, the resulting algorithm still does not scale as well as one would wish. For this reason other methods for the solution of nonlinear systems of equations are examined, with the goal of developing new parallel methods that may scale better. The first class of methods that are considered are generalised linear methods and block versions of these (chapter 10). The nonlinear Jacobi method, for example, is very easy to parallelise, but is unsuccessful on these difficult problems. The nonlinear block-SOR method is not that easy to parallelise and, as the results shows, offers few advantages.

One of the main reasons for the failure of the generalised linear methods is that the function value barely decreases in the update directions. In an attempt to overcome this problem a method based on a nonlinear block-Jacobi method is developed, which guarantees that every search direction is a descent direction. It is demonstrated that these descent directions can be computed efficiently and that the required line-searches can be executed without a large overhead.

When evaluating this method the importance of applying the methods to real problems and not artificial ones is seen. As discussed in section 2 power systems contain regulated busses which may change their state if limitations imposed on them are breached. The result is that load flow problems are similar to constrained optimisation problems. However, a problem that can occur in constrained optimisation problems is zigzagging [48, sec. 11.3]. Zigzagging occurs in this descent method. It is not straightforward to transfer the classical remedies for this kind of behaviour from optimisation problems to load flow problems. Furthermore initial results showed that this method is not competitive with the Newton-Raphson method.

Finally, in chapter 11, an attempt is made to transfer the idea of reduced systems to nonlinear systems. For nonlinear systems it is not possible to construct an exact reduced
system, but an approximation can be constructed. The result is a method that requires fewer global iterations than the Newton-method, and which solves the systems considered here in roughly the same time. These approaches could have advantages when scaling to more processors for larger systems, but sacrifice the well-understood convergence behaviour of the Newton method.
Chapter 9

The Newton-Raphson method

As already mentioned, the Newton-Raphson method is a very popular and well-understood method, which makes it attractive for the use in general purpose load-flow analysis packages. In this and the following chapters several assumptions about the way in which the problem is mapped on to the parallel machine and the reordering and partitioning are made. These are summarised in section 9.1. Subsequently the implementation of the parallel Newton-Raphson solution of the load flow equations is discussed in some detail in section 9.2, before results are presented in section 9.3.

9.1 Notation and assumptions

Some notation for nonlinear systems of equations is needed throughout this part of the thesis. Load flow analysis requires the solution of a nonlinear function \( F \in C^2(\mathbb{R}^n, \mathbb{R}^n) \), in other words a function that is at least twice continuously differentiable and maps \( \mathbb{R}^n \) into \( \mathbb{R}^n \). The dimension of the domain and range of \( F \) can change during the solution, but these problems are handled as discussed in section 3.8, and are considered constant for our purposes. The solution is a vector \( x_\ast \in \mathbb{R}^n \) such that \( F(x_\ast) = 0 \).

As a typical parallel load flow program encompasses several thousand lines of code, it is not an easy task to adapt a program to a new data layout on a parallel machine. For this reason some assumptions are made about the way that the nonlinear system is partitioned and distributed over the machine. The assumptions are that

- the nonlinear system is partitioned with a branch-tearing algorithm as discussed in chapter 7,
- that the system is partitioned into exactly as many block rows as there are processors, and,
- that block row \( i \) is placed on processor \( P_i \).
CHAPTER 9. THE NEWTON-RAPHSON METHOD

Assume that there are \( q \) such block-rows, and that there are \( n_i \) nodes without external connections and \( m_i \) nodes with external connections in the \( i \)th block row. On a moderate number of processors the supposition that \( m_i \ll n_i \) is always true. In every block row the \( n_i \) equations without external connections are ordered first and the \( m_i \) equations with external connections last. It immediately follows that all nonzero elements in the off-diagonal blocks of the Jacobian are in the last \( m_i \) rows of block \( i \).

Partition the vector of variables \( x \in \mathbb{R}^n \) similarly to give

\[
x = (x_1, y_1, x_2, y_2, \ldots, x_q, y_q)
\]

with \( x_i \in \mathbb{R}^{n_i} \) and \( y_i \in \mathbb{R}^{m_i} \). Partition

\[
F = (f_1, \hat{f}_1, f_2, \hat{f}_2, \ldots, f_q, \hat{f}_q)
\]

with \( f_i : \mathbb{R}^n \to \mathbb{R}^{n_i} \) and \( \hat{f}_i : \mathbb{R}^n \to \mathbb{R}^{m_i} \). The system of nonlinear equations then has the form

\[
\begin{align*}
f_1 (x_1, y_1, 0, 0, \ldots, 0, 0) & = 0 \\
\hat{f}_1 (x_1, y_1, 0, y_2, \ldots, 0, y_q) & = 0 \\
f_2 (0, 0, x_2, y_2, \ldots, 0, 0) & = 0 \\
\hat{f}_2 (0, y_1, x_2, y_2, \ldots, 0, y_q) & = 0 \\
& \vdots \\
f_q (0, 0, 0, \ldots, x_q, y_q) & = 0 \\
\hat{f}_q (0, y_1, 0, y_2, \ldots, x_q, y_q) & = 0
\end{align*}
\]

with the block equations \((f_i, \hat{f}_i)\) allocated to processor \( P_i \).

The Jacobian for this system is

\[
J = \begin{pmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & 0 & 0 & \cdots & \cdots & 0 & 0 \\
\frac{\partial \hat{f}_1}{\partial x_1} & \frac{\partial \hat{f}_1}{\partial y_1} & 0 & 0 & \cdots & \cdots & 0 & \frac{\partial \hat{f}_1}{\partial y_q} \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} & \cdots & \cdots & 0 & 0 \\
0 & \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} & \cdots & \cdots & 0 & \frac{\partial f_2}{\partial y_q} \\
& \vdots & \vdots & \vdots & \cdots & \cdots & \vdots & \vdots \\
& \vdots & \vdots & \vdots & \cdots & \cdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & \cdots & \frac{\partial f_q}{\partial x_q} & \frac{\partial f_q}{\partial y_q} \\
0 & \frac{\partial f_q}{\partial y_1} & 0 & 0 & \cdots & \cdots & \frac{\partial f_q}{\partial x_q} & \frac{\partial f_q}{\partial y_q}
\end{pmatrix}
\]

Frequent use is also made of the matrix \( \tilde{J} \) which consists of the main diagonal blocks of \( J \).
only:

\[
\begin{pmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & 0 & 0 & \cdots & \cdots & 0 & 0 \\
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & 0 & 0 & \cdots & \cdots & 0 & 0 \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} & \cdots & \cdots & \vdots & \vdots \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} & \cdots & \cdots & \vdots & \vdots \\
\vdots & \vdots & \ddots & \ddots & \cdots & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \cdots & \cdots & 0 & 0 \\
0 & 0 & \cdots & \cdots & 0 & 0 & \frac{\partial f_q}{\partial x_q} & \frac{\partial f_q}{\partial y_q} \\
0 & 0 & \cdots & \cdots & 0 & 0 & \frac{\partial f_q}{\partial x_q} & \frac{\partial f_q}{\partial y_q}
\end{pmatrix}
\]

(9-3)

The sub-matrices in these matrices are denoted by \( J_{ik} \) and \( \tilde{J}_{ik} \) respectively, so that

\[
J_{34} = \begin{pmatrix} 0 & 0 \\ 0 & \frac{\partial f_k}{\partial y_i} \end{pmatrix} \quad \text{and} \quad \tilde{J}_{33} = \begin{pmatrix} \frac{\partial f_k}{\partial x_3} & \frac{\partial f_k}{\partial y_3} \\ \frac{\partial f_k}{\partial x_3} & \frac{\partial f_k}{\partial y_3} \end{pmatrix}
\]

For the off-diagonal blocks the expression \( J_{ik} \) is also used to denote only the nonzero block \( \frac{\partial f_k}{\partial y_i} \). Which way \( J_{ik} \) is used is always clear from the context. The rows corresponding to this block are referred to as the border of the block row.

Due to the way the functions are distributed over the processors, blocks \( J_{ik} \) are assumed to be on processor \( P_i \), in other words the block row \( i \) of the Jacobian is on processor \( P_i \).

The notation \( \frac{\partial f_k}{\partial x_k} \) is also used.

### 9.2 Implementation of the parallel Newton-Raphson method

The Newton-Raphson method is a very effective method to solve load flow equations and, with the parallel linear solvers discussed in part II, provides a benchmark for other parallel methods. The linear solver used for this implementation is the algorithm BRANCH, which was discussed in chapter 6.

The system is partitioned into block rows as shown in equation (9-1). The number of processors \( p \) on which the job is to be executed is determined first and the system is then partitioned into \( p \) parts. The clustering algorithm with branch-tearing discussed in chapter 7 is used for this purpose. The equations within each block row are then reordered to ensure that all equations with ‘external nodes’ are ordered last. Blocks are then reordered to achieve maximum parallelism between blocks, so that as many leading off-diagonal blocks are zero as possible. Lastly the equations in every block are reordered with the minimum
degree algorithm. The reordering of the equations after the initial partitioning can be done in parallel. However, as explained in section 8.10, the reordering is done sequentially, before the distribution of the data, and is excluded from all the timings.

All the information about the partitioning is contained in two simple arrays: Part and BInd (for border index). For the ieee118 power system partitioned for three processors, these are

<table>
<thead>
<tr>
<th>Part</th>
<th>1</th>
<th>38</th>
<th>71</th>
<th>119</th>
</tr>
</thead>
<tbody>
<tr>
<td>BInd</td>
<td>34</td>
<td>66</td>
<td>116</td>
<td></td>
</tr>
</tbody>
</table>

Part contains the indices of the equations with which every partition starts, and BInd contains the indices where the border starts. So the second partition in this example starts with equation 38 and ends with equation 70 = 71-1. Equations 66 to 70 are those that contain external nodes, that is the equations that form \( f_2 \), and 38 to 65 are the equations that form \( f_2 \). The size of the off-diagonal border blocks can also be deduced easily as they have the same number of rows as the border block (i.e. 5 in this case). The number of columns is the same as the number of rows in the other border blocks (i.e. 38-34+1=5 for the first off-diagonal block and 119-116+1 = 4 for the last off-diagonal block).

The data is then distributed to the processors. The time required to distribute the data and assemble it on the different processors is included in all the timings. The data that is distributed is the following:

1. The partitioning information for the whole system
2. The admittances for all internal and external connections for a partition.
3. The power inputs/outputs for every node in the partition.
4. The numbers of the regulated busses and all the regulated bus information.
5. The starting voltages and angles unless a flat start is used.

When all this information is available on a processor the computation starts. Initially some administrative work is done: the equations are re-labelled so that the numbering is relative to the start of the partition. The sparse data structures of the admittance matrix are scanned to determine which diagonal blocks are empty and the sparse data structures for the Jacobian are determined. The structure of the Jacobian is constructed from the admittance matrix by expanding every nonzero element into a \( 2 \times 2 \) block.

The calculation of the power mismatches and the Jacobian are of course completely parallel, and it is here that a large part of the speed-up attained with the parallel method is achieved.
Once the Jacobian and the power mismatches have been calculated, the Newton-Raphson equation
\[ J(x^{(k)})s = -f(x^{(k)}) \]
must be solved. The algorithm that is used here is the direct method described in section 6.3. The reduced system is determined by calculating the Schur complement \( S_{ii} \) of every diagonal block \( J_{ik} \) for \( i = 1, \ldots, q \). These calculations are again executed in parallel.

The decomposition of the reduced system proceeds as follows.

**Reduced System Decomposition**

1. Whenever updates have been received from all processors \( P_k \) with \( k < i \) for which the off-diagonal blocks \( J_{ik} \) are nonzero, then:
   
   (a) Compute the dense LU-decomposition of the Schur complement \( S_{ii} \)
   (b) Compute \( S_{ii}^{-1} J_{ik} \) for \( k > i \) if \( J_{ik} \) is nonzero.
   (c) Send these (dense) matrices \( S_{ii}^{-1} J_{ik} \) to the processors \( P_k \) for which \( J_{ik} \) is nonzero.

   This step can always be executed for at least one row \( i \) and often for several.

2. For block row \( i \) on processor \( P_i \) execute the following steps. For every nonzero off-diagonal block \( J_{ik} \) with \( k < i \) receive the updated off-diagonal blocks \( S_{kk}^{-1} J_{kj} \) for \( k < j \leq q \) from processor \( P_k \). Multiply these with \( J_{ik} \) and subtract them from the corresponding blocks in row \( i \).

For matrices as small as those in the borders of every block row it is not worthwhile to use sparse data structures. As soon as the initial reduction step is finished, the border matrices are copied into dense data structures. This could be improved upon by allocating enough space in the sparse data structures for the Jacobians that the border blocks are always stored densely. But this is at the expense of some additional overhead in the LU-decomposition for the reduction step, as it may be necessary to search for the nonzero element in a row that must be updated. Although this change would be worthwhile and would reduce memory usage as well, it was decided to copy the sparse border rows into dense data structures for simplicity.

These dense data structures are compressed, in other words only those parts of the border in which nonzero elements can occur are stored. This is equivalent to storing the true reduced matrix as shown in figure 6-3 on page 115. When sending an update to another processor, the dense block to the right of the diagonal block is sent to the following processor. The processor that receives the update can multiply all the updated blocks with a single call to the BLAS-3 routine `DGEMM`. 
CHAPTER 9. THE NEWTON-RAPHSON METHOD

The solution of the reduced system is thus a standard block-LU-decomposition, as found in for example LAPACK [1], the only difference being that block rows are on different processors and do not all have the same size. The other major difference is that the occurrence of zero blocks in the reduced matrix is fully exploited to increase parallelism.

The back substitution to find the solution of the linear system proceeds analogously. When the back substitution of the reduced system has finished on processor \( P_k \), that processor immediately proceeds with the back substitution to find the solution for the whole block row \( k \). Once this has been computed, the estimates for the voltage angles and magnitudes are updated and the power mismatches are recomputed.

The next major problem in a parallel implementation of load flow analysis is the handling of the regulated buses. To this end an information vector is kept on every processor which registers the state of the regulated buses on all processors. This is important as the calculation of the section of the Jacobian in the border of block row \( i \) requires the state of any regulated buses in the borders of other block rows. After calculating the newest power mismatches the regulated buses are updated on every processor as described in section 3.8.5, and subsequently the regulated bus information for the border blocks only is exchanged. This may trigger the recalculation of power mismatches, on several processors, if a regulated bus has reverted from a PQ bus to a PV bus. This step is very time consuming, as it requires all processors to wait until the information has been received from all other processors. It is thus, in effect, a synchronisation step. Unfortunately such a reversion to a PQ bus occurs at least during one iteration for all the larger test cases (systems pow529 to pow1419) as is seen from table 3-4 on page 73.

Lastly the error norms are exchanged and the global error is calculated on every processor. The size of this error determines whether the calculation proceeds or not. The decision to proceed is taken on every processor independently.

The most important feature of this implementation is that it is purely data-driven. There are no explicit synchronisation steps and the calculation on every processor proceeds as soon as all the data it requires for the next part of the calculation is available.

9.3 Results for the parallel Newton-Raphson method

Results on an Intel hypercube with this parallel load flow method are given in table 9-1. As mentioned above these times exclude the partitioning and reordering of the system. The time required to distribute the data to the processors and collect the results is included. Every attempt has been made to make this a fair comparison between sequential and parallel methods: after all, the only time in which a user of a load flow program is interested is elapsed wall-clock time for the solution.
9.4. RESULTS WITH PVM

<table>
<thead>
<tr>
<th>System</th>
<th>Iterations</th>
<th>Error</th>
<th>1</th>
<th>3</th>
<th>7</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee118</td>
<td>4</td>
<td>0.000516</td>
<td>142</td>
<td>75</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>pow236</td>
<td>4</td>
<td>0.000656</td>
<td>252</td>
<td>150</td>
<td>124</td>
<td>245</td>
</tr>
<tr>
<td>pow529</td>
<td>4</td>
<td>0.000796</td>
<td>565</td>
<td>246</td>
<td>225</td>
<td>338</td>
</tr>
<tr>
<td>pow795</td>
<td>6</td>
<td>0.000004</td>
<td>1156</td>
<td>454</td>
<td>248</td>
<td>737</td>
</tr>
<tr>
<td>pow1419</td>
<td>4</td>
<td>0.000923</td>
<td>1323</td>
<td>568</td>
<td>575</td>
<td>324</td>
</tr>
</tbody>
</table>

Table 9-1: Results on an iPSC/860 with the parallel Newton-Raphson method.

The results show that reasonable decreases in solution time can be achieved on moderate numbers of processors. The solution time starts increasing again if the number of processors is increased too much, as the communication time then dominates the solution time. The increase in communication is due to three factors:

1. the distribution of the data takes longer as more administration is required and more messages have to be set up and sent

2. the steps where data must be exchanged between all processors, such as when the regulated busses are updated, requires more time

3. the partitioning tends to have many more border rows in total if the system is partitioned too finely.

The last factor is the most important. The changes in the number of border rows is also responsible for the hap-hazard way in which the time decreases as the number of processors is increased. For the system pow1419 this is particularly relevant, as the solution time actually increases when moving from 3 to 7 processors, before decreasing again when executing on 12 processors. When partitioned into 3 parts pow1419 has 52 border rows, when partitioned into 7 parts it has 90 rows and when partitioned into 12 parts only 38.

If the power systems are larger more processors can be used effectively, as there are more opportunities for partitioning the system. These results indicate that for large systems that are currently simulated, i.e. systems with 5000 to 10000 nodes, it should be possible to use at least 12 to 16 processors effectively.

9.4 Results with PVM

To assess the effectiveness of such algorithms on other architectures, this load flow program was also implemented on a Parallel Virtual Machine (PVM). This required some modifications to the code, as communication within PVM requires a task identifier for a process,
Table 9-2: Results under PVM over two DECstations (5000/240 and 5000/200).

<table>
<thead>
<tr>
<th></th>
<th>#Proc.</th>
<th>ieee118</th>
<th>pow236</th>
<th>Power System</th>
<th>pow529</th>
<th>pow795</th>
<th>pow1419</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>elapsed</td>
<td>309 ms</td>
<td>488 ms</td>
<td>1156 ms</td>
<td>2187 ms</td>
<td>4398 ms</td>
<td></td>
</tr>
<tr>
<td>PVM</td>
<td>user</td>
<td>164 ms</td>
<td>387 ms</td>
<td>996 ms</td>
<td>1992 ms</td>
<td>4097 ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>system</td>
<td>145 ms</td>
<td>102 ms</td>
<td>160 ms</td>
<td>195 ms</td>
<td>301 ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>elapsed</td>
<td>227 ms</td>
<td>359 ms</td>
<td>723 ms</td>
<td>1078 ms</td>
<td>2480 ms</td>
<td></td>
</tr>
<tr>
<td>PVM</td>
<td>user</td>
<td>105 ms</td>
<td>219 ms</td>
<td>531 ms</td>
<td>898 ms</td>
<td>2109 ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>system</td>
<td>121 ms</td>
<td>141 ms</td>
<td>191 ms</td>
<td>180 ms</td>
<td>371 ms</td>
<td></td>
</tr>
</tbody>
</table>

which is returned when spawning the process. The iPSC communication libraries only require a processor number, which is known a priori. The task ID's for all processes thus had to be distributed from the main program as well. Initially all other modifications were minor, although the design decision to concentrate all message passing in a few routines simplified the conversion significantly.

First results, which were published earlier in [79], are presented in table 9-2. These were obtained on two DECstations connected with Ethernet. Timing on time-shared systems is even more contentious than on single-user systems. The difficulty is that the user time only included the time that is spent in calculations on the one processor, not the time in which the process is idle. The elapsed time is more informative, as this includes the idle-time, but it also includes all the time spent in other system processes, such as NFS daemons. Here the elapsed, user and system times are given. These are the averages over several runs, with only one user on the system. The elapsed time should be used for comparison.

Recently the same program was executed on an Alpha Farm, consisting of 3 DEC Alpha 3000/300X machines, connected with a GigaSwitch. A GigaSwitch is a private, fibre-optic network with a cross-bar switch between the machines. The hardware-latency in such a communication system is low, while the bandwidth is very high. Unfortunately the software latency in PVM is significant.

Digital's mathematical library, DXML, was used as a source for the BLAS routines. The routines for calculating the power mismatches on the iPSC is highly vectorised, and this routine was rewritten to minimise cache misses on the Alpha. This change gave a speed-up of about 5%.

Results are presented in table 9-3. The results are disappointing, and the reason for this lies in the much worse communication to computation ratio. Both the sequential program and the parallel node programs were profiled; the most heavily used routines in both cases are the LU-decomposition/calculation of the Schur complements, and the calculation of the power mismatches and the Jacobian. All of these routines show a significant speed-up in
9.4. RESULTS WITH PVM

<table>
<thead>
<tr>
<th>System</th>
<th>Error</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee118</td>
<td>0.000516</td>
<td>22</td>
<td>288</td>
<td>490</td>
</tr>
<tr>
<td>pow236</td>
<td>0.000656</td>
<td>42</td>
<td>306</td>
<td>517</td>
</tr>
<tr>
<td>pow529</td>
<td>0.000550</td>
<td>95</td>
<td>391</td>
<td>440</td>
</tr>
<tr>
<td>pow795</td>
<td>0.000004</td>
<td>251</td>
<td>564</td>
<td>635</td>
</tr>
<tr>
<td>pow1419</td>
<td>0.000923</td>
<td>389</td>
<td>553</td>
<td>664</td>
</tr>
</tbody>
</table>

Table 9-3: Results under PVM for 1 to 3 DEC Alpha's connected by GigaSwitch.

the parallel case. The slow-down in these runs is purely due to the communication delays.

Digital claim that this is mainly due to software delays while sending the messages, and that these delays are significantly reduced in their implementation of PVM on newer systems.
Chapter 10

Block methods and descent methods

The oldest methods for solving nonlinear systems of equations are generalised linear methods such as the nonlinear Jacobi or SOR methods. These are the basis of the Gauß-Seidel load flow method described in section 3.2.

In general these methods are unattractive due to their slow (linear) convergence — the Gauß-Seidel method requires approximately as many iterations as there are nodes in the system. Nevertheless these methods are worth investigating as they are easy to parallelise.

Generalised linear methods look attractive if block versions are used, as is explained below. Unfortunately power systems provide difficult nonlinear systems, and this chapter is a catalogue of methods that do not work well.

10.1 Jacobi-Newton and SOR-Newton methods

As an example of a generalised linear method the Gauß-Seidel method is discussed. For a system of equations $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ of the form

\[
\begin{align*}
  f_1(x_1, x_2, \cdots, x_n) &= 0 \\
  f_2(x_1, x_2, \cdots, x_n) &= 0 \\
  \vdots & \vdots & \vdots \\
  f_n(x_1, x_2, \cdots, x_n) &= 0
\end{align*}
\]  

the method proceeds from a point $x$ to a point $x^+$ by successively solving the equations

\[
f_i(x_1^{(k+1)}, x_2^{(k+1)}, \cdots, x_{i-1}^{(k+1)}, x_i^{(k)}, x_{i+1}^{(k)}, \cdots, x_n^{(k)}) = 0 \quad i = 1, \ldots, n
\]  

for $x_i$. To solve equation (10-2) for $x_i$ involves the solution of a 1-dimensional nonlinear
equation. This equation can be solved with the 1-dimensional Newton method, which gives rise to the Gauß-Seidel-Newton method. When using just a single Newton iteration in the inner iteration the method is defined as follows [111, p. 220].

**1-step Gauß-Seidel-Newton method**  Given a starting vector \( (x_1^{(0)}, \ldots, x_n^{(0)}) \) calculate for iterations \( k = 0, 1, \ldots \), for \( i = 1, \ldots, n \)

\[
x_i^{(k+1)} = x_i^{(k)} - \left( \frac{\partial f_i(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k+1)}, \ldots, x_n^{(k+1)})}{\partial x_i} \right)^{-1} f_i(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k)}, \ldots, x_n^{(k)})
\]

until converged.

This method is known as the 1-step Gauß-Seidel-Newton method. If \( m \) Newton iterations are done, the resulting method is called an \( m \)-step method. These methods only converge under restrictive conditions (see the next section). Usually line-searches are needed to obtain convergence.

Using the same approach, it is obvious how to define the Jacobi-Newton and SOR-Newton methods.

**1-step Jacobi-Newton method**  Given a starting vector \( (x_1^{(0)}, \ldots, x_n^{(0)}) \) calculate for iterations \( k = 0, 1, \ldots \), for \( i = 1, \ldots, n \)

\[
x_i^{(k+1)} = x_i^{(k)} - \left( \frac{\partial f_i(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k)}, \ldots, x_n^{(k)})}{\partial x_i} \right)^{-1} f_i(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k)}, \ldots, x_n^{(k)})
\]

until converged.

**1-step SOR-Newton method**  Given a starting vector \( (x_1^{(0)}, \ldots, x_n^{(0)}) \) calculate for iterations \( k = 0, 1, \ldots \), for \( i = 1, \ldots, n \)

\[
x_i^{(k+1)} = x_i^{(k)} - \omega \left( \frac{\partial f_i(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k+1)}, \ldots, x_n^{(k+1)})}{\partial x_i} \right)^{-1} f_i(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k)}, \ldots, x_n^{(k)})
\]

until converged.

Block methods can be defined for the Jacobi-, Gauß-Seidel and SOR-Newton methods as well: the solution of (10-2) now involves the solution of a system of nonlinear equations. The discussion of the convergence of these methods in the next section shows why these methods seem attractive.
10.1.1 Convergence of generalised linear methods

Ortega and Rheinboldt [111, sec. 7.4] classify a Newton method in which the linear system in every iteration is solved by the Gauss-Seidel or SOR method as a generalised linear iteration as well. They call these methods Newton-Gauss-Seidel and Newton-SOR respectively — as opposed to for example Gauss-Seidel-Newton, where the inner iteration is a Newton method and the outer iteration a Gauss-Seidel method. The Newton-Jacobi, -Gauss-Seidel and -SOR methods were investigated in part II of this thesis, and their performance was poor.

The convergence of all these methods is very similar. Ortega and Rheinboldt [111, p.321] give the following result for the root convergence factor\(^1\) of a generalised linear iteration (under milder conditions).

**Theorem 10.1** Let \( F \in C^2(\mathbb{R}^n, \mathbb{R}^n) \) and a point \( x^* \in \mathbb{R}^n \) for which \( F(x^*) = 0 \). Define \( F'(x) = B(x) - C(x) \) and suppose that \( B : S_0 \to L(\mathbb{R}^n) \) is continuous at \( x^* \), and that \( B(x^*) \) is nonsingular. Define \( H(x) = B(x)^{-1}C(x) \) and define the iteration

\[
x_{k+1} = x_k - (I + \cdots + H(x_k)^{m-1})B(x_k)^{-1}F(x_k), \quad k = 0, 1, \ldots \tag{10-3}
\]

If \( \rho(H(x^*)) < 1 \), then, for any \( m \geq 1 \), \( x^* \) is a point of attraction of this iteration \( \mathcal{I} \) and 
\[
R_1(\mathcal{I}, x^*) = \rho(H(x^*))^m.
\]

A Newton-Jacobi method, which always does exactly \( m \) Jacobi iterations in every Newton iteration, has a root-convergence factor of \( \rho([D(x^*)]^{-1}(-L(x^*) - U(x^*))]^m) \), with the Jacobian \( F' = L + D + U \) with \( L \) lower triangular, \( D \) diagonal and \( U \) upper triangular. In other words, if \( m \) Jacobi or SOR iterations are done on the linear system in every Newton iteration, the Newton method will converge \( m \) times faster — asymptotically, \( m \) times fewer Newton iterations are needed, but \( m \) times more SOR iterations are done.

Ortega and Rheinboldt then proceed to show that a similar result holds for the iterations discussed in this chapter. Specifically they prove that the root convergence factor of the 1-step SOR-Newton method is also \( \rho(H(x^*)) \). As the nonlinear SOR-method in which every 1-dimensional iteration is solved exactly also has root convergence factor \( \rho(H(x^*)) \), \( m \)-step methods also have root convergence factor \( \rho(H(x^*)) \). It is thus not worthwhile to do more than 1 Newton-step in every iteration.

The reason that block-methods are of interest is that the convergence speed of the whole system depends on \( \rho(H(x^*)) \). It was seen already in chapter 5, table 5-3, that the use of a block-SOR method significantly decreases the spectral radius. If block methods also turn out to be easier to parallelise they may be competitive.

\(^1\)Root convergence factor, \( R_1(x^{(k)}) = \lim \sup_{k \to \infty} \|x^{(k)} - x^*\|^{1/k} \) [111, sec.9.2]
10.2. GENERALISED LINEAR METHODS AND LOAD FLOW

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Jacobi-Newton Error</th>
<th>GS-Newton Error</th>
<th>SOR-Newton (ω = 0.95) Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9219</td>
<td>0.9219</td>
<td>0.9219</td>
</tr>
<tr>
<td>1</td>
<td>0.9844</td>
<td>1.1644</td>
<td>1.1082</td>
</tr>
<tr>
<td>2</td>
<td>0.3390</td>
<td>0.4498</td>
<td>0.4723</td>
</tr>
<tr>
<td>3</td>
<td>0.3620</td>
<td>0.2233</td>
<td>0.2721</td>
</tr>
<tr>
<td>4</td>
<td>0.1489</td>
<td>0.1254</td>
<td>0.1597</td>
</tr>
<tr>
<td>5</td>
<td>0.2030</td>
<td>0.0642</td>
<td>0.0896</td>
</tr>
<tr>
<td>6</td>
<td>0.0805</td>
<td>0.0360</td>
<td>0.0535</td>
</tr>
</tbody>
</table>

Table 10-1: Block methods for the ieee14 test system.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Jacobi-Newton Error</th>
<th>GS-Newton Error</th>
<th>SOR-Newton (ω = 0.95) Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9317</td>
<td>0.9317</td>
<td>0.9317</td>
</tr>
<tr>
<td>1</td>
<td>0.4673</td>
<td>0.7678</td>
<td>0.7104</td>
</tr>
<tr>
<td>2</td>
<td>0.4963</td>
<td>0.6250</td>
<td>0.6049</td>
</tr>
<tr>
<td>3</td>
<td>0.4133</td>
<td>0.4413</td>
<td>0.4556</td>
</tr>
<tr>
<td>4</td>
<td>0.3964</td>
<td>0.3198</td>
<td>0.3436</td>
</tr>
<tr>
<td>5</td>
<td>0.3158</td>
<td>0.2328</td>
<td>0.2593</td>
</tr>
<tr>
<td>6</td>
<td>0.2965</td>
<td>0.1696</td>
<td>0.1956</td>
</tr>
</tbody>
</table>

Table 10-2: Block methods for the ieee30 test system.

10.2 Generalised linear methods and load flow

Generalised linear methods can now be formulated for nonlinear systems of equations with the same branch-torn structure discussed in part II of this thesis. One can thus assume a nonlinear system of equations as described in section 9.1.

The Jacobi method is trivially parallelisable: the current estimate of $x^{(k)}$ is distributed to all processors and every processor does one iteration independently of all the others. The results are then collated and the next iteration is done. Unfortunately the Jacobi method converges very slowly.

The Gauß-Seidel method tends to converge faster than the Jacobi method. The difficulty with parallelising the Gauß-Seidel method is that the solution of the blocks in the system must be done successively. But the fact that equations $f_k, k = 0, \ldots, m$ do not contain nonzero variables except $x_k$ and $y_k$, makes it possible to exploit a significant amount of parallelism. Viewed from processor $k$ in iteration $i$ this works as follows:
Parallel Gauß-Seidel-Newton method

1. Calculate \( f_k(x_k^{(i)} : y_k^{(i)}) \) and its Jacobian \( \partial_{x_k} f_k \) and \( \partial_{y_k} f_k \).

2. Calculate the LU-decomposition of \( \partial_{x_k} f_k \) and \( \partial_{x_k} f_k^{-1} \partial_{y_k} f_k \) simultaneously.

3. Wait for the all the values \( y_j^{(i+1)} \) with \( j < k \) to become available.

4. Calculate \( \tilde{f}_k(0, y_1^{(i+1)}, \ldots, x_k^{(i)}, y_k^{(i)}, \ldots, 0, y_{i+1}^{(i)}) \) and the Jacobian \( \partial_{x_k} \tilde{f}_k \) and \( \partial_{y_k} \tilde{f}_k \).

5. Calculate \( S = \partial_{y_k} \tilde{f}_k - \partial_{x_k} \tilde{f}_k \partial_{x_k} f_k^{-1} \partial_{y_k} f_k \) (which is a matrix-matrix multiplication and subtraction).

6. Solve \( y_k^{(i+1)} = y_k^{(i)} - S^{-1} \tilde{f}_k \).

7. Broadcast the new value \( y_k^{(i+1)} \) to all other processors.

8. Solve for \( x_k^{(i+1)} \) and repeat.

By exploiting the fact that some off-diagonal blocks are empty, it is also possible to calculate more than one \( y_k^{(i+1)} \) simultaneously. Both the Gauß-Seidel and the SOR method can thus be implemented fairly efficiently on a parallel machine for these problems. The question is whether the simplicity of implementation outweighs their slower convergence.

Results for the convergence of generalised linear iterations are given in tables 10-1 to 10-4. The number of iterations was restricted to six. Convergence is very slow and these methods are clearly not competitive. No line-searches were done in these calculations. Several different relaxation parameters were tried with the SOR method, but none provided better results than the Gauß-Seidel method.

One of the reasons that the convergence is so slow, is that the search direction is in a direction in which the function value hardly decreases. Even if the function value decreases in the search direction a step length of 1 is not always optimal. A remedy to this problem are line-searches, in other words searching along the descent direction for the point at which the power mismatch is minimal. Line-searches require at least one additional function evaluation, which makes them very expensive for load flow analysis problems. For this reason line-searches are only used if the function value does not decrease by more than some fixed value. Here a decrease of less than \( 10^{-3} \) was required before a line-search was done.

Tests were also done with the block Jacobi method and line-searches. This produced a very slight improvement in cases where there had been an increase in the error, such as in the block-Jacobi solution of the IEEE57 system (although the increase in the error there was caused by a change in a regulated bus). But in most cases the error does decrease in every
10.3. DESCENT DIRECTIONS

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Jacobi-Newton Error</th>
<th>GS-Newton Error</th>
<th>SOR-Newton ((\omega = 0.95)) Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.8654</td>
<td>2.8654</td>
<td>2.8654</td>
</tr>
<tr>
<td>1</td>
<td>1.2987</td>
<td>1.0680</td>
<td>0.9860</td>
</tr>
<tr>
<td>2</td>
<td>1.3768</td>
<td>0.9423</td>
<td>0.9119</td>
</tr>
<tr>
<td>3</td>
<td>0.7423</td>
<td>0.8027</td>
<td>0.7931</td>
</tr>
<tr>
<td>4</td>
<td>0.9663</td>
<td>0.6807</td>
<td>0.6850</td>
</tr>
<tr>
<td>5</td>
<td>0.5379</td>
<td>0.5764</td>
<td>0.5905</td>
</tr>
<tr>
<td>6</td>
<td>0.7110</td>
<td>0.4880</td>
<td>0.5087</td>
</tr>
</tbody>
</table>

Table 10-3: Block methods for the ieee57 test system.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Jacobi-Newton Error</th>
<th>GS-Newton Error</th>
<th>SOR-Newton ((\omega = 0.95)) Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.8906</td>
<td>5.8906</td>
<td>5.8906</td>
</tr>
<tr>
<td>1</td>
<td>1.9975</td>
<td>1.9437</td>
<td>1.8752</td>
</tr>
<tr>
<td>2</td>
<td>1.6193</td>
<td>1.3174</td>
<td>1.3344</td>
</tr>
<tr>
<td>3</td>
<td>1.0664</td>
<td>1.0552</td>
<td>1.0629</td>
</tr>
<tr>
<td>4</td>
<td>1.0629</td>
<td>0.8334</td>
<td>0.8542</td>
</tr>
<tr>
<td>5</td>
<td>0.7685</td>
<td>0.6670</td>
<td>0.6940</td>
</tr>
<tr>
<td>6</td>
<td>0.7517</td>
<td>0.5372</td>
<td>0.5681</td>
</tr>
</tbody>
</table>

Table 10-4: Block methods for the ieee118 test system.

iteration, before taking changes in regulated busses into account, so that the line search is not used.

The conclusion from these results is that, to make these generalised linear type methods competitive, the search direction must be 'steeper' or a search direction closer to the actual Newton direction is required. The first possibility is investigated in the next section.

10.3 Descent directions

One way to get convergence for a method to solve optimisation problems, is to ensure that every search direction is a descent direction. If every direction is a descent direction, convergence to a local minimum is guaranteed, provided that a minimum exists and the starting value is in a neighbourhood of this minimum, and line-searches are used. Descent directions are defined with reference to the minimisation problem

\[
\min_x \frac{1}{2} F(x)^T F(x) =: f(x).
\]  

(10-4)

A necessary condition to satisfy this equation is that the derivative at \(x\) must be zero, i.e.
\[ F^T J = 0 \]

Unfortunately this is not a sufficient condition, i.e. it is not necessarily true that the solution of the minimisation problem is a solution for the system of equations. When a solution is found, it is necessary to check whether this is also a solution to the nonlinear system, otherwise the method has to be restarted with a different starting point. In all tests done with load flow data sets this was never the case, this is due to the fact that a good starting value is always available.

A descent direction \( p \) must be in the opposite direction of the gradient, i.e.

\[ F^T J \cdot p < 0 \tag{10-5} \]

must be satisfied. In order to solve the load flow problem efficiently, a descent direction is required which can be computed efficiently with the distribution of the data assumed here. In order to do this, define a descent vector \( p \) as

\[ p = (0 \cdots 0 \ p_i \ \hat{p}_i \ 0 \cdots 0)^T \]

with nonzero elements in exactly those positions corresponding to the \( i \)th diagonal block. In this section the full vector \( p \) as well as the restriction of \( p \) to its nonzero part, i.e. \( p = (p_i \ \hat{p}_i)^T \) are both used. From the context it is always clear whether the full vector \( p \) or its restriction to block row \( i \) is meant.

Then equation (10-5) implies the inequalities

\[
\left( \begin{array}{c}
 f_i \\
 \hat{f}_i
\end{array} \right)^T \left( \begin{array}{cc}
 \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \\
 \frac{\partial \hat{f}_i}{\partial x_i} & \frac{\partial \hat{f}_i}{\partial y_i}
\end{array} \right) \left( \begin{array}{c}
 p_i \\
 \hat{p}_i
\end{array} \right) + \sum_{k \neq i} \left( \begin{array}{c}
 f_k \\
 \hat{f}_k
\end{array} \right)^T \left( \begin{array}{cc}
 0 & 0 \\
 0 & \frac{\partial f_k}{\partial y_i} \frac{\partial f_k}{\partial y_i}
\end{array} \right) \left( \begin{array}{c}
 p_k \\
 \hat{p}_k
\end{array} \right) < 0 . \tag{10-6}
\]

Denote the diagonal block \( i \) of the Jacobian by \( J_{ii} \) and define a vector \( q \) such that

\[
\left( \begin{array}{c}
 q_i \\
 \hat{q}_i
\end{array} \right) = J_{ii} \left( \begin{array}{c}
 p_i \\
 \hat{p}_i
\end{array} \right) . \tag{10-7}
\]

Then equation (10-6) becomes

\[
\left\{ \left( \begin{array}{c}
 f_i \\
 \hat{f}_i
\end{array} \right)^T + \sum_{k \neq i} \left( \begin{array}{c}
 f_k \\
 \hat{f}_k
\end{array} \right)^T \left( \begin{array}{cc}
 0 & 0 \\
 0 & \frac{\partial f_k}{\partial y_i} \frac{\partial f_k}{\partial y_i}
\end{array} \right) \left( \begin{array}{c}
 f_k \\
 \hat{f}_k
\end{array} \right)^T \right\} \left( \begin{array}{c}
 q_i \\
 \hat{q}_i
\end{array} \right) < 0 . \tag{10-8}
\]

Choose \((q_i \ \hat{q}_i)^T\) as the negative transpose of the expression in curly brackets, so that equation (10-8) is satisfied, provided that the expression in brackets is nonzero. The search direction \( p \) can then be found by solving the linear system
\( \begin{pmatrix} \frac{\partial f_k}{\partial x_i} & \frac{\partial f_k}{\partial y_i} \\ \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \end{pmatrix} \begin{pmatrix} \hat{p}_i \\ \hat{q}_i \end{pmatrix} = \begin{pmatrix} \hat{q}_i \\ \hat{q}_i \end{pmatrix} \)

The expression for \((q_i \hat{q}_i)^T\) is

\[
\begin{pmatrix} q_i \\ \hat{q}_i \end{pmatrix} = - \begin{pmatrix} f_i \\ \hat{f}_i \end{pmatrix} - \sum_{k \neq i} \left\{ \begin{pmatrix} f_k \\ \hat{f}_k \end{pmatrix}^T \begin{pmatrix} 0 & 0 \\ 0 & \frac{\partial f_k}{\partial y_i} \end{pmatrix} \begin{pmatrix} \frac{\partial f_k}{\partial x_i} & \frac{\partial f_k}{\partial y_i} \\ \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \end{pmatrix}^{-1} \right\}^T
\]

\[
= - \begin{pmatrix} f_i \\ \hat{f}_i \end{pmatrix} - \begin{pmatrix} \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \\ \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \end{pmatrix}^{-T} \sum_{k \neq i} \begin{pmatrix} 0 & 0 \\ 0 & \frac{\partial f_k}{\partial y_i} \end{pmatrix} \begin{pmatrix} f_k \\ \hat{f}_k \end{pmatrix}
\]

\[ (10-9) \]

One of the difficulties with using this expression in a program is the evaluation of the sum. The assumption throughout is that processor \(P_i\) contains the block row \(J_{ii}\), as described in section 9.1. Consequently the blocks \(J_{ki}\) are on different processors. Define quantities \(c_k\) as

\[
\begin{pmatrix} 0 \\ c_k \end{pmatrix} := \begin{pmatrix} 0 & 0 \\ 0 & \frac{\partial f_k}{\partial y_i} \end{pmatrix}^T \begin{pmatrix} f_k \\ \hat{f}_k \end{pmatrix}
\]

and these can be evaluated in parallel. Subsequently the expression

\[
\begin{pmatrix} \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \\ \frac{\partial f_i}{\partial x_i} & \frac{\partial f_i}{\partial y_i} \end{pmatrix}^{-T} \sum_{k \neq i} \begin{pmatrix} 0 \\ c_k \end{pmatrix}
\]

must be evaluated on processor \(P_i\).

The evaluation of \(J_{ii}^{-T}\) may seem problematic, but existing information can be reused: for block-Jacobi preconditioning, the diagonal blocks of the matrix must be factorised, so that these blocks \(J_{ii}\) are available in decomposed form \(J_{ii} = L_i \cdot U_i\) with \(L_i\) and \(U_i\) lower and upper triangular matrices respectively. Then

\[
J_{ii}^{-T} = (L_i \cdot U_i)^{-T} = (U_i^T \cdot L_i^T)^{-1}
\]

After the LU-decomposition, \(U_i\) is an upper triangular matrix stored by rows, so that \(U_i^T\) is a lower triangular matrix stored by columns. It is straightforward to update the forward and back substitution algorithms to work with triangular matrices in column form. Full use should be made of the fact that the first part of the \((0 \ c_k)^T\) is zero, as \(c_k\) will usually be much shorter than the leading vector of zeros. Most of the work involved in the forward substitution is avoided in this way.

The vector \(q\) can be found with a simple addition to the function result \((f_i \ \hat{f}_i)^T\). To obtain the descent direction \(p\), all that is needed is a final forward and back substitution. So for the calculation of a descent direction two forward and back substitutions are needed.
10.3.1 Another view of this derivation

The previous derivation is not very intuitive. To clarify it a different view of the construction of descent directions is presented in this section. Consider a system with only two block rows. Multiply the Jacobian $J$ from the right with the inverse of the matrix consisting only of the diagonal blocks of the Jacobian $\tilde{J}$:

$$
\left( \begin{array}{cccc}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & 0 & 0 \\
\frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial y_1} & 0 & 0 \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} \\
0 & 0 & \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2}
\end{array} \right) \left( \begin{array}{cccc}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & 0 & 0 \\
\frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial y_1} & 0 & 0 \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} \\
0 & 0 & \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2}
\end{array} \right)^{-1}
$$

The result is the matrix

$$
\left( \begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2}
\end{array} \right) \left( \begin{array}{cccc}
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} \\
0 & 0 & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2}
\end{array} \right)^{-1}
$$

Multiplying this matrix from the left with the transposed function vector gives:

$$
P^T J \tilde{J}^{-1} = \left( \begin{array}{c}
f_1^T \\
f_2^T
\end{array} \right) + \left( \begin{array}{cc}
f_2^T & 0 \\
0 & \frac{\partial f_1}{\partial y_1} \frac{\partial f_1}{\partial y_1}
\end{array} \right) \left( \begin{array}{cc}
\frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial y_2} \\
\frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial y_2}
\end{array} \right)^{-1} \left( \begin{array}{c}
f_2^T \\
f_2^T
\end{array} \right), \quad (10-10)
$$

Define the matrix $I_i$ as a zero matrix, with only an identity matrix in the diagonal block corresponding to block $J_{ii}$ in the Jacobian. An example is the matrix $I_1$ in a system with 2 blocks, which is defined as

$$
I_1 := \left( \begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array} \right)
$$

With this definition the equation (10-8) can be written as

$$
P^T J \tilde{J}^{-1} I_i \left( \begin{array}{c}
q_i \\
\dot{q}_i
\end{array} \right) < 0. \quad (10-11)
$$

Recall that $q = \dot{J} p$ and from equation (10-11) with equation (10-5) it follows that $q_i = -(P^T J \tilde{J}^{-1} I_i)^T$. Then the expression for $p_i$ is
\[ p_i = -\tilde{J}^{-1}I_i \tilde{J}^{-T} J^T F. \]  

(10-12)

To confirm that \( p \) is indeed a descent direction, define

\[ v_i \equiv F^T J \tilde{J}^{-1} I_i. \]

The descent condition (10-5) is then rewritten as

\[
F^T J p_i = -F^T J \tilde{J}^{-1} I_i \tilde{J}^{-T} J^T F \\
= - \left( v_1, \cdots, v_i, \cdots, v_q \right) \cdot (0, \cdots, v_i, \cdots, 0)^T \\
= - \|v_i\|_2^2 < 0.
\]

This is always the case if \( v_i \) is nonzero.

### 10.3.2 Descent in multiple directions simultaneously

It is possible to descend in \( q \) directions simultaneously, by defining the descent direction

\[
s = \sum_{i=1}^{q} -\tilde{J}^{-1} I_i \tilde{J}^{-T} J^T F \\
= \sum_{i=1}^{q} -\tilde{J}^{-1} \cdot (0, \cdots, v_i, \cdots, 0)^T \\
= -\tilde{J}^{-1} \cdot (v_1^T, \cdots, v_q^T)^T \\
\]

That \( s \) is indeed still a descent direction follows from

\[
F^T J s = -F^T J \tilde{J}^{-1} \cdot (v_1, \cdots, v_q)^T \\
= -(v_1, \cdots, v_q) \cdot (v_1, \cdots, v_q)^T \\
= -\|v_1, \cdots, v_q\|_2^2 < 0
\]

provided that any of the \( v_i \) is nonzero.

### 10.3.3 Line searches

Although \( s \) is a vector giving the descent direction, a step of length one is not necessarily optimal. Hence it is necessary to do line searches, which are defined for the related optimisation problem (10-4). For load flow problems line-searches are very expensive, as they require further function evaluations. While unnecessary line-searches have to be avoided, line-searches can reduce the number of iterations that is required, so that total computing time is reduced.
For a line-search the function that must be minimised is

\[ g(\lambda) = \frac{1}{2} \| F(x + \lambda s) \|^2. \]  

(10-13)

The information that is available after a function evaluation for \( \lambda = 1 \) is [26]

\[
\begin{align*}
g(0) &= F(x)^T F(x) \\
g'(0) &= F(x)^T J(x) s \\
g(1) &= F(x + s)^T F(x + s)
\end{align*}
\]

Line-searches are only done if \( g(1) > g(0) + \alpha g'(0) \) (note that \( g'(0) < 0 \) as \( s \) is a descent direction), where \( \alpha \) was chosen as \( \alpha = 10^{-4} \).

The values \( g(0) \) and \( g(1) \) can be calculated in parallel by evaluating \( f_i(x)^T f_i(x) \) on processor \( P_i \) and summing these values across processors. The evaluation of \( g'(0) \) poses more problems.

\[
\begin{align*}
g'(0) &= F(x)^T J(x) s = \sum_{i=1}^{q} \left( \sum_{i=1}^{q} f_i^T J_{ki} \right) s_i \\
&= \sum_{i=1}^{q} \left\{ \sum_{k \neq i} f_i^T J_{ki} + f_i^T J_i \right\} s_i
\end{align*}
\]

From the calculation of the correction (10-9) the values \( f_i^T J_{ki} \) are available on processor \( P_i \). Thus only the values \( f_i^T J_i \) must be computed, and this can be done locally on processor \( P_i \). So no communication is required to calculate every processor’s contribution to \( g'(0) \), and \( g'(0) \) can be computed as a global sum.

If \( g(1) > g(0) + \alpha g'(0) \) for some small \( \alpha \), e.g. \( \alpha = 10^{-4} \), then the minimum of the one-dimensional quadratic

\[ \hat{m}(\lambda) = (g(1) - g(0) - g'(0))\lambda^2 + g'(0)\lambda + g(0) \]  

(10-14)

must be found, which is

\[ \lambda = \frac{-g'(0)}{2(g(1) - g(0) - g'(0))} \]  

(10-15)

and the new estimate is \( x + \lambda s \). That this is indeed a minimiser follows from

\[ \hat{m}''(\lambda) = 2(g(1) - g(0) - g'(0)) > 0 \]

as \( g(1) \geq g(0) + \alpha g'(0) > g(0) + g'(0) \), i.e. from the fact that the search direction is a descent direction.
### 10.3. DESCENT DIRECTIONS

| Test System | \(\rho(G'(x^*))\) | No. Eigenvalues \(e_i\) | \(|e_i| < 10^{-7}\) | \(|e_i| \geq 1\) |
|-------------|-------------------|-----------------|-----------------|-----------------|
| ieee14      | 5.703080          | 28              | 16              | 3               |
| ieee30      | 7.147611          | 60              | 36              | 6               |
| ieee57      | 13.079823         | 114             | 90              | 8               |
| ieee118     | 14.192130         | 236             | 216             | 6               |

Table 10-5: Spectral radius at the solution \(x^*\) for \(G'(x^*)\). The solution was found with a full Newton-Raphson method.

In some cases this choice of \(\lambda\) may still not reduce the function value. This is the case if the function \(g(\lambda)\) is not approximated well by a quadratic. If this is the case a new quadratic can be fitted through the points \(g(0)\) and \(g(\lambda)\) or a cubic function function can be fitted through \(g(0), g(\lambda)\) and \(g(1)\) (see [26, sec.6.3.2]).

A second problem with descent methods is that the step-length may be too small. One way of ensuring that the step-length is big enough is to use the following condition [26]:

\[
g'(\lambda) \geq \beta g'(0) \tag{10-16}
\]

with \(\beta \in (\alpha, 1)\) (e.g. \(\beta = 0.5\)). This requires the rate of decrease at \(x + \lambda s\) to be larger than some fraction of the rate of decrease at \(x\). Choosing \(\beta > \alpha\) ensures that this condition can be fulfilled simultaneously with the condition that

\[
g(\lambda) \leq g(0) + \alpha \lambda g'(0) \tag{10-17}
\]

Using the line-search strategy outlined above means that this condition is very rarely necessary, as the initial step-length is chosen as 1 and then reduced.

#### 10.3.4 Convergence

This descent method can be written in the general form

\[
Gx = x - A(x)^{-1}Fx
\]

\[
= x - (J^{-T} J^T \bar{J})^{-1}(x) Fx \tag{10-18}
\]

Assuming that \(J\) and \(\bar{J}\) are nonsingular in the vicinity of the solution (this follows from the definition of the Jacobian in table 3-1 on page 46), it follows that \(A^{-1}\) exists. The derivative of \(G\) is

\[
G'(x^*) = I - A(x^*)^{-1}F'(x^*) = I - \bar{J}^{-1}(x^*)J^{-T}(x^*)J^T(x^*)J(x^*) \tag{10-19}
\]

and, provided that \(\rho(G'(x^*)) < 1\), the Ostrowski theorem states that \(x^*\) is a point of attraction of \(G\).
The solution of the system \( x^* \) was calculated with a Newton method and the spectral radius of \( G'(x^*) \) determined. The results are given in table 10-5 for some test cases. As \( \bar{J} \) consists of most of the matrix \( J \) it is not surprising that many of the eigenvalues are zero. What is surprising is just how large the remaining eigenvalues are. Note that \( \bar{J} \) and \( J \) appear twice in the expression for \( G' \), which implies that the problem becomes more ill-conditioned. This follows from the fact that, if \( v \) is an eigenvector corresponding to the maximal eigenvalue \( \lambda \) for a matrix \( A \), this means that \( A(Av) = A\lambda v = \lambda^2 v \). Here \( J \) and \( \bar{J} \) are not entirely symmetric, but the spectrum of \( J\bar{J}^{-1}I_i\bar{J}^{-2}J^T \) will still expand in a similar fashion.

As the spectral radius is consistently larger than 1, it follows that the iteration (10-18) will not converge to a fixed point. For convergence the method thus relies on line searches.

It is however possible to show that a descent method converges under surprisingly mild conditions, see for example Dennis and Schnabel [26, th. 6.3.3]. The main conditions are that the function \( f(x) = F(x)^T F(x) \) is a \( C^2(\mathbb{R}^n) \) function, and that its derivative obeys a Lipschitz condition. Provided that the starting point is in a neighbourhood of a minimiser, and that line-searches are used that satisfy the conditions (10-16) and (10-17), the descent method will converge.

### 10.3.5 Results

Results obtained with the descent method are presented in tables 10-6 to 10-9. There are two reasons for the disappointing performance of the descent method.

The first has to do with the search direction. The fact that \( G \) is not a contraction in the neighbourhood of the solution means that the search direction can be ill chosen and badly scaled. This is found in practice. In fact, the step is often so badly scaled, that the factor \( \lambda \) is not close to its optimal value. To alleviate this problem the standard approach is to halve the step length in the search direction until the \( f(x + \lambda s) \) is only slightly larger than \( f(x) \) and then do a line search. This is prohibitively expensive, as the function evaluations in load flow problems are expensive. It furthermore requires synchronisation between processors on a parallel computer in order to make the decision whether to proceed with the bisection process or do a line search.

The second reason for the bad performance has to do with the constraints. The changes in regulated buses are also the reason for the increase in the function value in table 10-6. To exemplify the problem, consider the behaviour of the regulated buses in the case of the ieee14 and ieee30 test systems, given in table 10-10.

In the table it can be seen that the constraints cycle. This is a well-known problem in optimisation, where is is known as zig-zagging [48]. The usual remedies applied in optimisation to avoid it do not transfer directly to the load flow problem as formulated.
### 10.3. DESCENT DIRECTIONS

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Error</th>
<th>$f^T Js$</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9219</td>
<td>-2.0817</td>
<td>1.0000</td>
</tr>
<tr>
<td>1</td>
<td>0.5964</td>
<td>-2.8201</td>
<td>0.3178</td>
</tr>
<tr>
<td>2</td>
<td>0.3683</td>
<td>-2.1602</td>
<td>0.0715</td>
</tr>
<tr>
<td>3</td>
<td>0.2912</td>
<td>-0.7216</td>
<td>1.0000</td>
</tr>
<tr>
<td>4</td>
<td>0.2591</td>
<td>-1.9499</td>
<td>0.0725</td>
</tr>
<tr>
<td>5</td>
<td>0.3101</td>
<td>-0.3474</td>
<td>0.0731</td>
</tr>
<tr>
<td>6</td>
<td>0.2645</td>
<td>-0.0738</td>
<td>0.1266</td>
</tr>
</tbody>
</table>

Table 10-6: Block methods for the ieee14 test system (Partitioned into 2 blocks).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Error</th>
<th>$f^T Js$</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9317</td>
<td>-44.2425</td>
<td>0.1688</td>
</tr>
<tr>
<td>1</td>
<td>1.6054</td>
<td>-20.3153</td>
<td>0.1433</td>
</tr>
<tr>
<td>2</td>
<td>1.3194</td>
<td>-13.4893</td>
<td>0.1036</td>
</tr>
<tr>
<td>3</td>
<td>1.0946</td>
<td>-8.0114</td>
<td>0.3167</td>
</tr>
<tr>
<td>4</td>
<td>0.8266</td>
<td>-13.5373</td>
<td>0.0354</td>
</tr>
<tr>
<td>5</td>
<td>0.7244</td>
<td>-5.2134</td>
<td>0.0780</td>
</tr>
<tr>
<td>6</td>
<td>0.6678</td>
<td>-5.2134</td>
<td></td>
</tr>
</tbody>
</table>

Table 10-7: Block methods for the ieee30 test system (Partitioned into 2 blocks).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Error</th>
<th>$f^T Js$</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.8906</td>
<td>-256.9480</td>
<td>0.0752</td>
</tr>
<tr>
<td>1</td>
<td>5.5526</td>
<td>-148.1196</td>
<td>0.3650</td>
</tr>
<tr>
<td>2</td>
<td>3.7898</td>
<td>-189.1612</td>
<td>0.0100</td>
</tr>
<tr>
<td>3</td>
<td>3.7371</td>
<td>-184.7242</td>
<td>0.0100</td>
</tr>
<tr>
<td>4</td>
<td>3.6862</td>
<td>-147.1266</td>
<td>0.0100</td>
</tr>
<tr>
<td>5</td>
<td>3.6392</td>
<td>-120.7052</td>
<td>0.0149</td>
</tr>
</tbody>
</table>

Table 10-8: Block methods for the ieee57 test system (Partitioned into 3 blocks).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Error</th>
<th>$f^T Js$</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.8906</td>
<td>-256.9480</td>
<td>0.0752</td>
</tr>
<tr>
<td>1</td>
<td>5.5526</td>
<td>-148.1196</td>
<td>0.3650</td>
</tr>
<tr>
<td>2</td>
<td>3.7898</td>
<td>-189.1612</td>
<td>0.0100</td>
</tr>
<tr>
<td>3</td>
<td>3.7371</td>
<td>-184.7242</td>
<td>0.0100</td>
</tr>
<tr>
<td>4</td>
<td>3.6862</td>
<td>-147.1266</td>
<td>0.0100</td>
</tr>
<tr>
<td>5</td>
<td>3.6392</td>
<td>-120.7052</td>
<td>0.0149</td>
</tr>
</tbody>
</table>

Table 10-9: Block methods for the ieee118 test system (Partitioned into 3 blocks).
Table 10-10: The state of regulated busses in successive iterations of the descent method for the ieee14 and ieee30 test systems. Note that some of the busses change their state in every iteration.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>ieee14</th>
<th>ieee30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bus 6</td>
<td>Bus 14</td>
</tr>
<tr>
<td>1</td>
<td>PV ← PQ</td>
<td>PV ← PQ</td>
</tr>
<tr>
<td>2</td>
<td>PQ ← PV</td>
<td>PQ ← PV</td>
</tr>
<tr>
<td>3</td>
<td>PV ← PQ</td>
<td>PV ← PQ</td>
</tr>
<tr>
<td>4</td>
<td>PQ ← PV</td>
<td>PQ ← PV</td>
</tr>
<tr>
<td>5</td>
<td>PV ← PQ</td>
<td>PQ ← PQ</td>
</tr>
<tr>
<td>6</td>
<td>PQ ← PV</td>
<td>PQ ← PV</td>
</tr>
</tbody>
</table>

here. These are formulated for active set methods with linear constraints and require second-order multipliers. See Fletcher [48, p268ff] for details. In the load flow problem the second order multipliers are not readily available. However the basic strategy could be followed, which is to keep the set of constraints fixed if cycling is detected. When suitably close to the solution, the constraints are checked and the active set updated.

The reason for not further pursuing these methods is that, to be competitive with the Newton-Raphson method, these methods have to converge in relatively few iterations. This does not leave much scope for detecting cycling of constraints, then updating the active sets and subsequently doing further iterations to converge to a solution.

These results confirm that generalised linear methods and variants of these are not competitive with the Newton-Raphson method, even though they parallelise much more easily. Their rate of convergence is so much slower that it offsets any gains made through increased parallelism. To obtain methods that are competitive with the standard Newton-Raphson method it is necessary to obtain search directions that are much closer to the Newton-Raphson direction.
Chapter 11

Reduced methods

In chapter 6 a method to transform a linear system to a reduced system is described. The usefulness of that approach lies in the fact that the behaviour of the whole system is modelled by a much smaller system.

In this section the possibilities of adapting this approach for nonlinear systems are investigated. The motivation for pursuing this idea is that it may make it possible to reduce the amount of information that must be exchanged between processors, which should result in a more efficient parallel algorithm.

A reduced system is only known to exist in a neighbourhood of the solution. However, linear approximations serving a similar purpose can be constructed locally around other points. The resulting algorithm reduces the number of global iterations that are required in some cases. Problems with this method are (i) that the approximation to a reduced system does not take account of most regulated busses that have to change their state, (ii) that its convergence is only partially understood and (iii) that it is complicated to implement.

An alternative approach is to do several nonlinear iterations on block rows of the system, interspersed with global iterations. The resulting algorithm is called the multi-level Newton method and is discussed in section 11.6. This algorithm reduces the number of global iterations that have to be executed, which is important due to the increasingly bad ratio between computational speed and communication speed on modern machines. On the iPSC/860 this algorithm is competitive with the Newton method. However, the convergence of this algorithm is again not as well understood as that of the Newton method.

Some of the ideas of the multi-level method are similar to those reported in Rabbat et al. [107] and Zhang et al. [142]. However the algorithm used here differs significantly in its implementation from their methods.
11.1 Reduced nonlinear systems

Consider a function $F$ which is partitioned as shown in equation (9-1) on page 170. The

\begin{align*}
g_1(y_1, y_2, \cdots, y_q) &= 0 \\
g_2(y_1, y_2, \cdots, y_q) &= 0 \\
\vdots & \quad \vdots \\
g_q(y_1, y_2, \cdots, y_q) &= 0
\end{align*}

which approximates the behaviour of the system as a whole. This system is only a fraction of the size of the original system. Such a representation can be achieved if, for $i = 1, \ldots, q$, functions $H_i$ can be found, such that

\begin{equation}
f_i(0, \cdots, H_i(y_i), y_i, \cdots, 0) = 0 \quad \text{for all } y_i \in \mathbb{R}^{m_i}.
\end{equation}

Then (11-1) can be defined as

\begin{equation}
g_i(y) = f_i(0, y_1, \cdots, H_i(y_i), y_i, \cdots, 0, y_q).
\end{equation}

The power mismatch functions are $C^\infty$, i.e. infinitely often differentiable, and the implicit function theorem (see e.g. Rheinboldt and Ortega [111, p.128]) guarantees the existence of functions $x_i = H_i(y_i)$ in a neighbourhood of a point $(x^*, y^*)$ for which $f_i(x^*, y^*) = 0$. Consequently this approach is valid in a neighbourhood of the solution of the load flow problem, but not necessarily at the starting value.

As mentioned in the introduction, an attempt is made to construct a linear approximation to the functions $H_i$. Around the solution $(x^*, y^*)$ such a linear approximation can be obtained from the Taylor expansion of $f_i$:

\begin{equation}
f_i(x, y) \approx f_i(x^*, y^*) + \frac{\partial f_i}{\partial x_i}(x_i - x_i^*) + \frac{\partial f_i}{\partial y_i}(y_i - y_i^*)
\end{equation}

The functions $H_i$ can then be approximated by

\begin{equation}
H_i(y_i) = x_i \approx x_i^* - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} \left\{ f_i(x^*, y^*) + \frac{\partial f_i}{\partial y_i}(y_i - y_i^*) \right\}
\end{equation}

where $f_i(x^*, y^*) = 0$. This choice of $x_i$ is an approximation to the value that will make $f_i$ zero if $y_i \neq y_i^*$.

As the solution $(x^*, y^*)$ is not known, this approximation is not very helpful. However, for values $(x, y)$ in a neighbourhood of the solution, (11-5) will still be a reasonable approximation to the functions $H_i$. How large this neighbourhood depends on the nonlinearity
of the functions $f_i$: if $f_i$ is linear — that is, the second and higher order terms in the Taylor expansion are zero — (11-5) is an exact representation of $H_i$ for any choice of $x \in \mathbb{R}^n$.

As a reasonably good starting value is always available for the load flow problem in the form of a flat start, it seems possible that one could use this starting value to construct an initial approximation to $H_i$. The value of $H_i$ could then be determined numerically by using a fixed point iteration to determine $H_i(y_i)$, but such an iteration would be expensive, so that it will not be competitive with the Newton method. Alternatively, a linear approximation to $f_i$ can be constructed from the Taylor expansion around the starting value, which is refined as more information becomes available.

The first choice for the approximation to $H_i$ is (11-5), but with the Taylor expansion now developed around the starting value $(x^{(0)}, y^{(0)})$

$$H_i(y_i) = x_i \approx x_i^{(0)} - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} \left\{ f_i(x^{(0)}, y^{(0)}) + \frac{\partial f_i}{\partial y_i} (y_i - y_i^{(0)}) \right\}$$

(11-6)

Note that, contrary to the definition for $H_i(y_i)$ in equation (11-5), this choice for $H_i$ changes the value of the function $f_i$ after a change in $y_i$. In fact, (11-6) is not really a definition for a reduced system, but for a given $y_i$ it does a single Newton step for the function $F_i(x_i) = f_i(x, y^{(0)}) + \frac{\partial f_i}{\partial y_i} (y_i - y_i^{(0)}) \approx f_i(x, y)$ for the starting value $x_i^{(0)}$. This can also be interpreted as obtaining the values for $x_i$ from a generalised Newton method (fixed Jacobian method) at $(x_i^{(0)}, y)$, for which the function value $f(x_i^{(0)}, y)$ is approximated with a first order Taylor expansion around $(x^{(0)}, y^{(0)})$.

It is almost certainly not true (that is, except when all the functions $f_i$ are linear) that the reduced system (11-1) with this approximation of $H_i$ has the same solution as the original system. For this approach to work it is thus necessary to periodically update equation (11-6) with new estimates for the solution.

In order to apply the Newton method to the reduced system (11-1) it is necessary to compute

1. the function value

$$g_i(y_i) = \hat{f}(0, y_1, \cdots, H_i(y_i), y_1, \cdots, y_q)$$

(11-7)

2. the derivative

$$\frac{\partial g_i}{\partial y_i} = \frac{\partial \hat{f}_i}{\partial x_i} \frac{\partial H_i}{\partial y_i} + \frac{\partial \hat{f}_i}{\partial y_i}$$

(11-8)

When using (11-6) as estimate for $H_i$, the derivative (11-8) for $g_i$ is

$$\frac{\partial g_i}{\partial y_i} = \frac{\partial \hat{f}_i}{\partial y_i} - \frac{\partial \hat{f}_i}{\partial x_i} \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} \frac{\partial f_i}{\partial y_i}$$
This is the Schur complement of the diagonal block $J_i$ of the Jacobian of the original load flow system, so that the calculation of this derivative is equivalent to calculating the reduced linear system during the Newton solution of the full load flow system.

That the definition (11-6) is not an unreasonable approach to take, i.e. that it at least has the same solution as the original system of equations locally, is seen from the Taylor expansion of $(f_i, \hat{f}_i)$ around $(x_i^{(0)}, y_i^{(0)})$:

$$
\begin{pmatrix}
  f_i \\
  \hat{f}_i
\end{pmatrix}(x, y) 
\approx
\begin{pmatrix}
  f_i \\
  \hat{f}_i
\end{pmatrix}(x^{(0)}, y^{(0)}) + 
\begin{pmatrix}
  \frac{\partial f_i}{\partial x_i} \\
  \frac{\partial \hat{f}_i}{\partial y_i}
\end{pmatrix}(x^{(0)}, y^{(0)}) \Delta x_i + \sum_{k \neq i} \frac{\partial f_k}{\partial y_k} \Delta y_k
$$

The definition of $H_i y_i := x_i$ is chosen so that the first equation becomes zero for all $y_i$, i.e. $f(x, y) \approx 0$. Eliminating $\Delta x$ from the last equation (which is equivalent to filling in $\Delta x$ obtained from (11-6)) gives

$$
\hat{f}(x, y) 
\approx
\hat{f}(x^{(0)}, y^{(0)}) - \frac{\partial \hat{f}_i}{\partial x_i} \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} f(x^{(0)}, y^{(0)}) + 
\left( \frac{\partial \hat{f}_i}{\partial y_i} - \frac{\partial \hat{f}_i}{\partial x_i} \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} \frac{\partial f_i}{\partial y_i} \right) \Delta y_i + \sum_{k \neq i} \frac{\partial f_k}{\partial y_k} \Delta y_k
$$

$$
\approx
\hat{f}(0, y_i^{(0)}, \ldots, x^{(0)} - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} f(x^{(0)}, y^{(0)}) y_i^{(0)}, \ldots, 0, y_q^{(0)})
+ \frac{\partial g_i}{\partial y_i} \Delta y_i + \sum_{k \neq i} \frac{\partial g_i}{\partial y_k} \Delta y_k
$$

$$
= \hat{f}(0, y_i^{(0)}, \ldots, H_i y_i^{(0)}, y_i^{(0)}, \ldots, y_q^{(0)}) + \frac{\partial g_i}{\partial y_i} \Delta y_i + \sum_{k \neq i} \frac{\partial g_i}{\partial y_k} \Delta y_k
$$

It follows that the definition (11-6) for $H_i$ leads to a function $g_i$ which is consistent, insofar that it agrees (up to the approximation error) with the Taylor-expansion of $(f_i, \hat{f}_i)$ in a neighbourhood of $(x^{(0)}, y^{(0)})$. The difficulty with the expansion around a point for which $f(x^{(0)}, y^{(0)}) \neq 0$ is that $H_i y_i^{(0)} \neq x_i^{(0)}$.

Equation (11-6) is not the only option for approximating $H_i$ when doing more than one Newton iteration on the reduced system. It is also possible to define $H_i$ as

$$
H_i(y_i) := x^{(0)} - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} f_i(x^{(0)}, y)
$$

(11-9)

That this definition for $H_i$ is also consistent with the Taylor expansion around $(x^{(0)}, y^{(0)})$ can be confirmed easily, and it avoids the linear approximation to $f_i(x^{(0)}, y_i)$ used in (11-6). For this representation of $H$ the derivative is expensive to calculate, as

$$
\frac{\partial H_i}{\partial y_i} = - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} \left|_{(x^{(0)}, y^{(0)})} \frac{\partial f_i}{\partial y_i} \right|_{(x^{(0)}, y)}
$$
11.2. Definition of the Reduced Algorithm

As the goal of a reduced system is to do a few cheap reduced iterations, interspersed with more expensive, parallel operations to initialise new estimates for $H_i$, the latter approach is not attractive. As was discussed in section 3.7, the cost of calculating the whole function value is higher than the cost of calculating the Jacobian, and the required forward and back substitutions are also expensive. As these operations have to be done in parallel, there is no advantage in doing this much work and sacrificing the quadratic convergence of the Newton method.

It should be emphasised that, while these definitions for $H_i$ may approximate the behaviour of the full system in a neighbourhood of the starting point, they will not usually lead to a solution for the whole system. It is thus essential to recalculate the Jacobians intermittently.

11.2 Definition of the reduced algorithm

The reduced algorithm can be summarised as follows. In this algorithm $m$ iterations are done on the reduced system.

Nonlinear Reduced Algorithm

1. Compute the power mismatches (9-1) at $(x^{(0)}, y^{(0)})$.

2. If the power mismatches are small enough, terminate.

3. Compute the Jacobian (9-2) at $(x^{(0)}, y^{(0)})$.

4. Compute the LU-decomposition of $\partial x_i f_i^{-1}$ for $i = 1, \ldots, q$.

5. Calculate $\partial x_i f_i^{-1}$ and $\partial x_i f_i^{-1} \partial y_i f_i$.

6. For $k = 0, \ldots, m - 1$ perform the reduced iteration:

   (a) For $i = 1, \ldots, q$, calculate $H y_i^{(k)} = x_i^{(0)} - \partial x_i f_i^{-1} f_i - \partial x_i f_i^{-1} \partial y_i f_i (y_i(k) - y_i(0))$ and $g_i(0, y_1, \ldots, H(y_i), y_i, \ldots, y_q)$.

   (b) If $\| (g_1, \ldots, g_q) \|_{\infty}$ is small, exit loop.

   (c) Calculate $\partial y_i f_i$ and $\partial y_i f_i$.

   (d) Solve the linear system defined for $i = 1, \ldots, q$ by

   \[
   \left( \partial y_i \dot{f}_i - \partial x_i \dot{f}_i \partial x_i f_i^{-1} \partial y_i f_i \right) \Delta y_i^{(k)} + \sum_{j \neq i} \partial y_i \dot{f}_j \Delta y_j^{(k)} = -g_i.
   \]

   (e) Set $y^{(k+1)} = y^{(k)} + \Delta y^{(k)}$. 
7. Set $x_i^{(k+1)} = H y_i^{(k+1)}$ for $i = 1, \ldots, q$ and go to 1.

When referring to reduced iterations an iteration as in step 6 is meant; when referring to a global iteration the calculation of $x_i^{(k)}$ and the recalculation of the Jacobian is meant. The reduced iterations are very cheap indeed, as they only involve the calculation of a small subset of power mismatches, and the corresponding block-rows in the Jacobian.

### 11.3 Convergence of the reduced method

To simplify the discussion of the convergence of the reduced method, rewrite the nonlinear system so that the Jacobian has the structure shown in figure 11-1, and define

$$f = (f_1, f_2, \ldots, f_q)$$
$$\hat{f} = (\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_q)$$
$$x = (x_1, x_2, \ldots, x_q)$$
$$y = (y_1, y_2, \ldots, y_q)$$

A moment’s reflection confirms that the reduced system for $(f, \hat{f})$ is identical to the reduced system for $(f_1, \hat{f}_1, \ldots, f_q, \hat{f}_q)$. In the following only the system $(f, \hat{f})$ is considered.

Assume that $(f, \hat{f})$ is sufficiently smooth, say $(f, \hat{f}) \in C^2(R^n)$; in fact, the real and imaginary load flow equations are in $C^\infty(R^n)$.

As mentioned earlier the calculation of $x = Hy$ is in some sense a fixed Jacobi iteration. For the reduced method to converge a minimal demand is that the iteration (for ease of notation subscripts are used for the iteration index in this section)

$$x_1 = x_0 - \left( \frac{\partial f_i}{\partial x_i} \right)^{-1} f(x_0, y_0)$$
11.3. CONVERGENCE OF THE REDUCED METHOD

reduces the error \( \|x - x^*\| \), with \( x^* \) the solution. This is the case if the nonlinear block-Jacobi method converges, i.e. when \( \rho \left( -D(x)^{-1}(L(x) + U(x)) \right) < 1 \), where \( D(x) \) is the block diagonal of the Jacobian, and \( L \) and \( U \) the upper and lower parts respectively [111, p.321].

To simplify the indexing in the following discussion, define \( \Delta y_0 = 0 \) and set \( y_1 = y_0 \). Then a reduced iteration step is equivalent to solving

\[
\begin{pmatrix}
    \partial_x f(x_0, y_0) & \partial_y f(x_0, y_0) \\
    \partial_x \tilde{f}(x_k, y_k) & \partial_y \tilde{f}(x_k, y_k)
\end{pmatrix}
\begin{pmatrix}
    \Delta x_k \\
    \Delta y_k
\end{pmatrix}
= \begin{pmatrix}
    f(x_0, y_0) + \partial_x f(x_0, y_0) \cdot (x_k - x_0) + \partial_y f(x_0, y_0) \cdot (y_k - y_0) \\
    \tilde{f}(x_k, y_k)
\end{pmatrix}
\]

(11-10)

This can be seen from the elimination of \( \Delta x \) and the definition of \( x_k = H y_k \) (11-6)

\[
(\partial_y \tilde{f}(x_k, y_k) - \partial_x \tilde{f}(x_k, y_k) \partial_x f(x_0, y_0)^{-1} \partial_y f(x_0, y_0)) \Delta y_k
= -\tilde{f}(x_k, y_k) + \partial_x \tilde{f}(x_k, y_k) \{(x_k - x_0) + \\
\partial_x f(x_0, y_0)^{-1} \{f(x_0, y_0) + \partial_y f(x_0, y_0) \cdot (y_k - y_0)\}\}
= -\tilde{f}(x_k, y_k) + \partial_x \tilde{f}(x_k, y_k) \{(x_k - x_0) - (x_k - x_0)\}
= -\tilde{f}(x_k, y_k)
\]

It remains to be shown that \( x_{k+1} = x_k + \Delta x_k \), but this follows from the first equation in (11-6)

\[
\partial_x f(x_0, y_0)(x_k - x_0 + \Delta x_k) = -f(x_0, y_0) - \partial_y f(x_0, y_0) \cdot (y_k - y_0 + \Delta y_k)
\]

\[
x_{k+1} - x_0 = -\partial_x f(x_0, y_0)^{-1} \{f(x_0, y_0) + \partial_y f(x_0, y_0) \cdot (y_{k+1} - y_0)\}
= H(y_{k+1})
\]

Define the error as

\[
\epsilon_k = \left\| \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \begin{pmatrix} x^* \\ y^* \end{pmatrix} \right\|_\infty.
\]

The question now is under which circumstances the iteration (11-10) improves the error \( \epsilon \). This question is not fully answered here, but at least it can be shown that the reduced iteration does not diverge. Define \( J_\epsilon \) to be the Jacobian evaluated at the solution \( (x^*, y^*) \) and

\[
J_k = \begin{pmatrix}
    \partial_x f(x_0, y_0) & \partial_y f(x_0, y_0) \\
    \partial_x \tilde{f}(x_k, y_k) & \partial_y \tilde{f}(x_k, y_k)
\end{pmatrix}
\]

**Theorem 11.1** Assume \( \begin{pmatrix} f \\ \tilde{f} \end{pmatrix} \in C^2(\mathbb{R}^n) \), \( (x^*, y^*) \in \mathbb{R}^n \) and \( \begin{pmatrix} f \\ \tilde{f} \end{pmatrix} (x^* y^*) \) \( = 0 \); and, that there is an \( r > 0 \) and a ball \( N((x^*, y^*), r) \) such that \( J, \partial_x f \) and \( \partial_y \tilde{f} \) are Lipschitz
continuous with parameters $\gamma$, $\gamma_A$ and $\gamma_D$ respectively, that $(J_*)^{-1}$ and $\partial_\mathbf{x} f(x_0, y_0)$ exist and that $\|(J_*)^{-1}\|_\infty < \beta$ and $\|\partial_\mathbf{x} f(x_0, y_0)^{-1}\| < \beta_A$. Then there exists a parameter $\delta > 0$ such that for all $x_0 \in N((\mathbf{x}^*, \mathbf{y}^*), \delta)$ the sequence generated by (11-10) is well-defined and $\varepsilon_k \leq \frac{1}{2} \varepsilon_0$.

**Proof:** Consider the first step of the reduced method: as $\partial_\mathbf{x} f$ is Lipschitz continuous and $\|\partial_\mathbf{x} f(x_0, y_0)^{-1}\| < \beta_A$ there is a neighbourhood around $x_0$ in which $\partial_\mathbf{x} f(x_0, y_0)^{-1} f(x_0, y_0)$ converges to a fixed point (see the simplified Newton theorem [111, p.421]). That this indeed reduces the $\varepsilon$ follows from the block contraction theorems discussed by Bertsekas and Tsitsiklis [8, sec.7.5].

In the following the proof for the convergence of the Newton method by Dennis and Schnabel [26] is followed closely. Choose the parameter $\delta$ as

$$\delta = \min \left\{ r, \frac{1}{2 \beta \gamma} \right\}$$

Then it follows for all $y$ such that $\|\mathbf{y} - \mathbf{y}^*\|_\infty < \|\mathbf{y}_0 - \mathbf{y}^*\|_\infty$ that

$$\|(J_*)^{-1}(J_k - J_*)\|_\infty \leq \|(J_*)^{-1}\|_\infty \|J_k - J_*\|_\infty$$

$$\leq \beta \gamma \left\| \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} x^* \\ y^* \end{pmatrix} \right\|_\infty \leq \beta \gamma \delta \leq \frac{1}{2}$$

But this implies (see [26, th.3.1.4]) that

$$\|J_k^{-1}\|_\infty \leq \frac{\|(J_*)^{-1}\|_\infty}{1 - \|(J_*)^{-1}(J_k - J_*)\|_\infty}$$

$$\leq 2 \|(J_*)^{-1}\|_\infty \leq 2 \beta$$

Consequently the iteration (11-10) is well-defined and

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} - \begin{pmatrix} x_* \\ y_* \end{pmatrix} = \begin{pmatrix} x_k \\ y_{k+1} \end{pmatrix} - \begin{pmatrix} x_* \\ y_* \end{pmatrix} - \left( \begin{pmatrix} \partial_\mathbf{x} f(x_0, y_0) \\ \partial_\mathbf{y} f(x_0, y_0) \end{pmatrix} \begin{pmatrix} \partial_\mathbf{x} f(x_k, y_k) \\ \partial_\mathbf{y} f(x_k, y_k) \end{pmatrix} \right)^{-1}$$

$$\begin{pmatrix} f(x_0, y_0) + \partial_\mathbf{x} f(x_0, y_0)(x_k - x_0) + \partial_\mathbf{y} f(x_0, y_0)(y_k - y_0) \\ \hat{f}(x_k, y_k) \end{pmatrix}$$

$$= J_k^{-1} \left\{ \begin{pmatrix} f(x_*, y_*) \\ h(x_*, y_*) \end{pmatrix} - \begin{pmatrix} f(x_0, y_0) \\ \hat{f}(x_0, y_0) \end{pmatrix} - \begin{pmatrix} \partial_\mathbf{x} f(x_0, y_0)(x_* - x_0) + \partial_\mathbf{y} f(x_0, y_0)(y_* - y_0) \\ \partial_\mathbf{x} \hat{f}(x_k, y_k)(x_* - x_0) + \partial_\mathbf{y} \hat{f}(x_k, y_k)(y_* - y_0) \end{pmatrix} \right\}$$

The error in the quadratic model for a Lipschitz continuous function $F$ is bounded by [26, Lemma 4.1.12]

$$\|F(x) - F(x_0) - \partial_\mathbf{x} F(x_0)(x - x_0)\| \leq \frac{\gamma}{2} \|x - x_0\|^2$$

(11-11)
It follows that
\[
\left\| \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} - \begin{pmatrix} x^* \\ y^* \end{pmatrix} \right\|_\infty \leq \frac{\gamma}{2} \max \left\{ \|x_0 - x^*\|_\infty, \|y_0 - y^*\|_\infty \right\}^2 \\
\leq \beta \gamma \max \left\{ \|x_0 - x^*\|_\infty, \|y_0 - y^*\|_\infty \right\}^2 \\
\leq \frac{1}{2} \max \left\{ \|x_0 - x^*\|_\infty, \|y_0 - y^*\|_\infty \right\}
\]
The last step follows from the choice of \( \delta \). This implies that \( \epsilon_k \leq \beta \gamma \epsilon_0 \leq \frac{1}{2} \epsilon_0. \)

This theorem is not very satisfactory: it does show that the reduced iterations do not diverge if the starting value is close enough to the solution, but it is not clear on the conditions under which individual reduced iterations improve the error. Note that equation (11-10) is exactly a Newton iteration if \( f \) is a linear function, and this results in the following lemma.

**Lemma 11.2** If \( (f, \dot{f})^T \) comply to the conditions of theorem 11.1, and \( f(x) \) is linear, then the reduced iterations (11-10) are identical to full Newton iterations, and the convergence obeys \( \epsilon_{k+1} \leq \beta \gamma \epsilon_k^2 \) for \( k > 1 \).

So on the one hand it is known that the reduced method converges like the Newton method if \( f \) is linear, but on the other hand all that is known is that the error does not get worse than it would be after a single Newton step. It is clear that the reduced method will only improve the error if the estimate \( (x_k, y_k) \) is 'close' to the starting point \( (x_0, y_0) \) in the sense that the linear approximation to \( f(x_k, y_k) \) is good, i.e. the Lipschitz constant for \( f \) is small. For load flow problems one finds that 2–3 reduced iterations improve the error, after which no further improvement is achieved. For its convergence the reduced method thus relies on the occasional global iteration.

### 11.4 Initial results

Results with the reduced method are given in table 11-1. For the cases considered here the reduced method converges marginally faster than the Newton method, but every global iteration is more expensive than for the Newton method. One of the reasons that the reduced method works, is that a good starting value is available, so that the iterates \( \Delta y_k \) are only small, and the approximation \( x_i \approx H_i y_i^{(k)} \) does not contain a very large error.

A major problem with this approach is that all the variables corresponding to the \( x_i \) are only explicitly included in the iterative process in every global iteration. The consequence is that the regulated buses (most of which are included in the functions \( f_k \), as these cover the vast majority of the buses) have not got the opportunity to move to their final state.
### Table 11-1: Results with the reduced algorithm. For the reduced algorithm I $m = 2$ and for algorithm II $m = 2$ until the error is smaller than 0.1, after which $m = 3$. In both cases the reduced iterations were terminated once $\|g\|_\infty < 10^{-4}$.

<table>
<thead>
<tr>
<th>System</th>
<th>It</th>
<th>Newton Error</th>
<th>Reduced method I Error</th>
<th>#Reduced It.</th>
<th>Reduced method II Error</th>
<th>#Reduced It.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee14</td>
<td>0</td>
<td>0.921936</td>
<td>0.921936</td>
<td>2</td>
<td>0.921936</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.101207</td>
<td>0.057477</td>
<td>2</td>
<td>0.057477</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.000717</td>
<td>0.000064</td>
<td></td>
<td>0.000064</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ieee30</td>
<td>0</td>
<td>0.931678</td>
<td>0.931678</td>
<td>2</td>
<td>0.931678</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.071452</td>
<td>0.067334</td>
<td>2</td>
<td>0.067334</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.000601</td>
<td>0.000277</td>
<td>2</td>
<td>0.000277</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>ieee57</td>
<td>0</td>
<td>2.865400</td>
<td>2.865400</td>
<td>2</td>
<td>2.865400</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.148785</td>
<td>0.148342</td>
<td>2</td>
<td>0.148342</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.003785</td>
<td>0.003406</td>
<td>2</td>
<td>0.003406</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.000006</td>
<td>0.000004</td>
<td></td>
<td>0.000004</td>
<td></td>
</tr>
<tr>
<td>ieee118</td>
<td>0</td>
<td>5.890557</td>
<td>5.890557</td>
<td>2</td>
<td>5.890557</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.824945</td>
<td>0.824945</td>
<td>2</td>
<td>0.824945</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.010397</td>
<td>0.010397</td>
<td>2</td>
<td>0.010397</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.062346</td>
<td>0.000116</td>
<td>2</td>
<td>0.000116</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.025804</td>
<td>0.000028</td>
<td></td>
<td>0.000028</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.000559</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pow236</td>
<td>0</td>
<td>5.890557</td>
<td>5.890557</td>
<td>2</td>
<td>5.890557</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.538277</td>
<td>0.824945</td>
<td>2</td>
<td>0.824945</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.050693</td>
<td>0.013232</td>
<td>2</td>
<td>0.013232</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.229793</td>
<td>0.000268</td>
<td>1</td>
<td>0.000268</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.098105</td>
<td>0.000091</td>
<td></td>
<td>0.000091</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.069031</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.000065</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider the results in table 11-1, where two reduced algorithms are compared to the Newton method\(^1\). The error after every global iteration and the number of reduced iterations are given in the table. Note that for the ieee30, ieee118 and pow236 systems, there are cases where algorithm II does more reduced iterations than algorithm I, but the error after the global iteration is the same. This is due to regulated busses changing their state, so that

---
\(^1\)The number of iterations for the Newton method are at times higher here than reported in table 9-1 due to the use of a numerical workbench package, Octave, which uses full matrices. The rounding errors are different than in the sparse implementation, and consequently the regulated busses change in different iterations.
the maximal error is encountered at a regulated bus (the infinity norm is used here). This is also the reason for the errors after the first iteration for ieee118 and pow236 begin the same: pow236 was constructed from two copies of ieee118 and contains the same regulated busses.

These results indicate that it is not worthwhile doing more than two inner iterations. For the ieee30, ieee118 and pow236 there was once again a problem with some regulated busses cycling, i.e. changing their state in every iteration.

A possibility to avoid some of these problems is to do several Newton iterations on every subsystem \((f_i, \tilde{f}_i)\) independently first. This way new estimates for \((x_i, y_i)\) are obtained on processor \(i\), which are better than the starting values \((x_i^{(0)}, y_i^{(0)})\), and the regulated busses may change to their final states. During subsequent reduced iterations changes in \(\Delta x_i\) should also be smaller, improving the accuracy of the linear approximations. However, too many independent Newton iterations will converge to the solution of the system \((f_i, \tilde{f}_i)\) subject to \((y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_q)\) so that the estimate to \((x_i^*, y_i^*)\) may deteriorate.

Another problem with this approach: as the values \(y_i\) are not exchanged, they will have different values on different processors after these initial iterations. The solution is to exchange the values \(y_i\) and to allow the value of \(y_i\), calculated with block equation \(i\), to supersede all other estimates for \(y_i\).

The algorithm for which results are given in table 11-2 is the following:

**Nonlinear Reduced Algorithm with an Initial Newton Iteration**

1. In parallel do a single independent Newton-Raphson iteration on the system \((f_i, \tilde{f}_i)\) given values \((y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_q)\) to obtain \((x_i^{[i]}, y_i^{[i]})\) for \(i = 1, \ldots, q\)

2. Construct a starting vector \((x_i^{(0)}, y_i^{(0)})\) such that \(x_i^{(0)} = x_i^{[i]}\) and \(y_i^{(0)} = y_i^{[i]}\).

3. Execute the reduced algorithm with \(m = 2\).

In all cases the estimates for \(x_i\) are improved by the initial Newton iteration. The results show a slight improvement in the final errors, but no reduction in the number of global iterations. Unfortunately this algorithm does not have a large effect on the number of regulated busses changing their state in every iteration.

### 11.5 Corrections

The exchange of values in step 2 of the reduced algorithm with initial iterations causes new errors in the estimates of \(x_i\). It is possible to correct the estimates of these values in all processors for the newest \(y\)-values. This can be achieved cheaply by considering that the
### Table 11-2: The reduced method with initial iterations to find a good starting value for the values $x_i$. For the reduced methods The number of inner iterations was chosen as $m = 2$. NR is the error after the parallel Newton iteration and Corr the error after the parallel Newton iteration followed by a correction step.

<table>
<thead>
<tr>
<th>System</th>
<th>Iteration</th>
<th>Newton-Raphson</th>
<th>No Initial Iterations</th>
<th>1 Initial Iteration</th>
<th>Initial iteration &amp; Correction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.921936</td>
<td>0.921936</td>
<td>0.921936</td>
<td>0.921936</td>
</tr>
<tr>
<td></td>
<td>NR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Corr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.101207</td>
<td>0.057477</td>
<td>0.013231</td>
<td>0.013231</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.000717</td>
<td>0.000064</td>
<td>0.000002</td>
<td>0.000002</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iee14</td>
<td>0</td>
<td>0.931678</td>
<td>0.931678</td>
<td>0.931678</td>
<td>0.931678</td>
</tr>
<tr>
<td></td>
<td>NR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Corr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.071452</td>
<td>0.067334</td>
<td>0.037432</td>
<td>0.013231</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.000601</td>
<td>0.000277</td>
<td>0.000123</td>
<td>0.000002</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.000000</td>
<td>0.000001</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>iee30</td>
<td>0</td>
<td>2.865400</td>
<td>2.865400</td>
<td>2.866400</td>
<td>2.865400</td>
</tr>
<tr>
<td></td>
<td>NR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Corr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.148785</td>
<td>0.148342</td>
<td>0.047451</td>
<td>0.051546</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.003785</td>
<td>0.003406</td>
<td>0.000306</td>
<td>0.000409</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.000006</td>
<td>0.000004</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>iee57</td>
<td>0</td>
<td>5.890557</td>
<td>5.890557</td>
<td>5.890557</td>
<td>5.890557</td>
</tr>
<tr>
<td></td>
<td>NR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Corr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.824945</td>
<td>0.824945</td>
<td>0.012588</td>
<td>0.012752</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.010397</td>
<td>0.010397</td>
<td>0.213749</td>
<td>0.064385</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.062346</td>
<td>0.000116</td>
<td>0.001924</td>
<td>0.000041</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.025804</td>
<td>0.000028</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.000059</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iee118</td>
<td>0</td>
<td>5.890557</td>
<td>5.890557</td>
<td>5.890557</td>
<td>5.890557</td>
</tr>
<tr>
<td></td>
<td>NR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Corr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.538277</td>
<td>0.824945</td>
<td>0.280665</td>
<td>0.253899</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.050693</td>
<td>0.013232</td>
<td>0.220940</td>
<td>0.098212</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.229793</td>
<td>0.000268</td>
<td>0.002080</td>
<td>0.000096</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.098105</td>
<td>0.000091</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.069031</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.000065</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
system solved on processor $i$ is

$$\begin{pmatrix} f_i \\ \hat{f}_i \end{pmatrix} \begin{pmatrix} x_i, y_i \end{pmatrix} \bigg\rvert_{(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_N)} = 0 \quad (11-12)$$

After the initial iterations all $y_k$ are updated by $\Delta y_i$, where $\Delta y_i$ is the solution of

$$0 = \begin{pmatrix} f_i \\ \hat{f}_i \end{pmatrix} (x_i, y_i) + J_{ii} \begin{pmatrix} \Delta x_i \\ \Delta y_i \end{pmatrix} \quad (11-13)$$

with $(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_N)$ fixed. After the initial parallel iterations the corrections to $y_k$, computed on processor $k$, become available on processor $i$. The solution $(x_i, y_i)$ can be corrected with $(\delta_i, \hat{\delta}_i)$ in so far that equation (11-13) becomes true for the updates $(\Delta y_1, \ldots, \Delta y_{i-1}, \Delta y_{i+1}, \ldots, \Delta y_N)$, which gives

$$0 = \begin{pmatrix} f_i \\ \hat{f}_i \end{pmatrix} (x_i, y_i) + J_{ii} \begin{pmatrix} \Delta x_i + \delta_i \\ \Delta y_i + \hat{\delta}_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{k \neq i} J_{ik} \Delta y_k \end{pmatrix} \quad (11-14)$$

As $(\Delta x_i, \Delta y_i)$ is the solution of (11-13), the calculation of $(\delta_i, \hat{\delta}_i)$ involves the solution of

$$\begin{pmatrix} \delta_i \\ \hat{\delta}_i \end{pmatrix} = -J_{ii}^{-1} \begin{pmatrix} 0 \\ \sum_{k \neq i} J_{ik} \Delta y_k \end{pmatrix}$$

As $J_{ii}$ was factorised to solve (11-13), the calculation of $(\delta_i, \hat{\delta}_i)$ amounts to a few small matrix-vector multiplications and a forward and back substitution. The forward substitution is very cheap due to the many zeros in the right hand side.

The values for $y_i$ differ yet again on all the processors and have to be exchanged. This idea was suggested by Zhang et al. [142], although in that work double-bordered systems were considered. This means that only a single border block instead of many off-diagonal blocks occurs. Consequently all the $(\delta_i, \hat{\delta}_i)$ depend on the same $\Delta y_N$, and there is no problem with the interdependence of the $y_i$.

Using this correction improves things slightly, as can be seen in the last column in table 11-2, in some cases saving a global iteration. Note however that in the case of the system ieee57 the corrections actually increase the error; the cause is that an additional regulated bus changes its state after the correction.

### 11.6 The multilevel Newton method

The methods introduced in the first part of this chapter converge, but the approximations depend on the size of the nonlinear component of the equations in the subsystem, that is the size of the second and higher order derivatives. The results discussed in sections 11.4 show
that these are significant in load flow problems (no more than two reduced iterations can be gainfully used between global iterations), which is not surprising as there are many sines and cosines in the functions, so that the second derivative of the system is not negligible.

The results also show that convergence depends strongly on when regulated busses change their state. With the reduced methods discussed up to now, most regulated busses have few opportunities to change their states as this can only happen during global iterations.

A further draw-back is that the implementation of the reduced method is very complicated: the reduced iterations must be done on a single processor, in order to avoid excessive communication overhead, while the global iterations will happen in parallel. This means that the information for the functions \( \tilde{f}_i \) has to be duplicated on two processors and kept updated.

A possible solution to these problems is to do several parallel, independent Newton iterations on the block rows of the system, and then do a global iteration. This should reduce the errors in the estimate of \( x_i \), and prevent a small change in \( y_i \) precipitating a large change in \( H_i y_i \). This can be motivated as follows: set \( y \) equal to the solution \( y^* \) and \( x = x^{(0)} \); then several independent Newton iterations on the subsystem \((f_i, \tilde{f}_i)\) may be required before \( x_i \) is close to \( x_i^* \). For reference these parallel Newton iterations are called inner iterations.

Methods which do several iterations on subsystems, before doing an iteration on the global system, have been reported in the literature for some time. The earliest reference to so-called multilevel Newton methods is by Rabbat et al. [107]. They require explicitly constructed macro-models for the sub networks, which have the same function as the reduced systems introduced earlier. They then do separate Newton iterations on the subsystems and the global system (which can be interpreted as the reduced system). The fact that some subsystems are latent is exploited to reduce the amount of work: if a subsystem's inputs are found not to have changed, no further Newton iterations are done for that subsystem. But if they have changed, Newton iterations are performed until the error is below a preset value.

More recent work on a Newton algorithm which performs many iterations on subsystems was done by Zhang et al. [142]. They consider the case of a double-bordered block structured system, with many iterations being performed on the main-diagonal blocks and occasional iterations on the Schur complement. They show a speed-up on a parallel machine when using several inner iterations with line searches.

Of course care must be taken not to do too many inner iterations, as \( x_i^{(k)} \) converges to a solution that depends on \( y_i \). If \( y_i \) is still far from the solution \( y_i^* \) much work is done to converge to a wrong solution for \( x_i \).
11.6. THE MULTILEVEL NEWTON METHOD

The multilevel algorithm described above can be summarised as follows:

Multilevel Algorithm

1. Do for \( k = 0, \ldots, m - 1 \) the inner iteration

   (a) Calculate \((f_i, \hat{f}_i)\) with \( y_j^{(k)} = y_j^{(0)}, j \neq i \) for \( i = 1, \ldots, g \).

   (b) If \( \max(\|f_i\|_\infty, \|\hat{f}_i\|_\infty) \) small, exit loop.

   (c) Calculate the Jacobian \( J_{ii}(x_i^{(k)}, y_i^{(k)}) \) and solve

   \[
   J_{ii} \begin{pmatrix} \Delta x_i \\ \Delta y_i \end{pmatrix} = - \begin{pmatrix} f_i \\ \hat{f}_i \end{pmatrix} \tag{11-15}
   \]

   (d) Set \((x_i^{(k+1)}, y_i^{(k+1)}) = (x_i^{(k)}, y_i^{(k)}) + (\Delta x_i, \Delta y_i)\)

2. Exchange the values \( y_i^{(m)} \) between processors, and recalculate \( J \) and \((f, \hat{f})\) with the new values.

3. Solve the system \( J(\Delta x, \Delta y)^T = (f, \hat{f})^T \).

4. Update \((x, y)\) and recalculate the function value \((f, \hat{f})\).

5. If converged, terminate, otherwise set \((x^{(0)}, y^{(0)}) = (x^{(k)}, y^{(k)})\) and go to 1.

Equation (11-15) has this simple form as \( \Delta y_j = 0 \) for \( j \neq i \). It is of course possible to do a correction step, as discussed in section 11.5, after the distribution of the \( y \) values derived from the inner iterations (step 2).

This method differs significantly from the methods proposed by Rabbat et al. [107] and Zhang et al. [142]. Rabbat et al. never do an explicit iteration with the whole system, but only do an iteration with an approximate global system (through the use of macro-models for subsystems) and Zhang et al. do not do the inner iterations on all nodes of the system (the border block is excluded).

The reduced and multi-level methods can be combined in the following way: in the initial iterations several inner iterations may be needed to get a reasonable approximation of the reduced system, but in the later iterations (some of the) subsystems may not change much, so that several iterations with the reduced system may be profitable. The best balance between inner and full iterations is problem dependent. See appendix F for some tests that were done with the multilevel method with varying numbers of inner/outer iterations.
11.7 Convergence of the multilevel Newton method

The multi-level algorithm is identical to the Newton method if no inner iterations are done.

The inner iterations are much more difficult to analyse. In the first inner iterations the Jacobian is calculated at the point \((x^{(0)}, y^{(0)})\), which is common to all processors. The first Jacobian is thus the Jacobian \(J(x^{(0)}, y^{(0)})\) and the first inner iteration is a block Jacobi-Newton iteration of the form encountered in chapter 10. This gives the following lemma.

**Lemma 11.3** The multilevel algorithm with one inner iteration will converge whenever the Newton method and the block Jacobi-Newton method both converge.

In subsequent inner iterations the situation becomes more complicated, as there are no unique values for \((x^{(k)}, y^{(k)})\). These differ for the calculation in every block row \(i\). It is not true that a nonlinear system defined by

\[
\begin{pmatrix}
\frac{\partial f_i(x_i, y_i)}{\partial x_i} \\
\frac{\partial f_i(x_i, y_i)}{\partial y_i}
\end{pmatrix} \left| \begin{array}{c}
(z^{(0)}, \ldots, z^{(0)}, y_i^{(0)}, \ldots, y_i^{(0)}) \\
y_1^{(0)}, \ldots, y_{i-1}^{(0)}, y_{i+1}^{(0)}, \ldots, y_q^{(0)}
\end{array} \right| = 0
\]

is being solved on processor \(i\), as the \(y\)-values are included in the calculation of the Jacobian (otherwise the Jacobian may be singular, see Appendix B).

In the first inner iteration a block Newton-Jacobi iterations is done on all block rows. Subsequently the values for \(y_j, j \neq i\) are not updated and the following iteration is done on processor \(P_i\)

\[
J(x_i^{(k)}, y_1^{(0)}, \ldots, y_{i-1}^{(0)}, y_i^{(k)}, y_{i+1}^{(0)}, \ldots, y_q^{(0)}) \cdot (\Delta x \Delta y)^T = f(x_i^{(k)}, y_1^{(0)}, \ldots, y_{i-1}^{(0)}, y_i^{(k)}, y_{i+1}^{(0)}, \ldots, y_q^{(0)})
\]

The main problem is that the solution to the original problem is not even a fixed point for this iteration. The convergence for this iteration, without exchange of information, is always worse than for the Newton-Jacobi method. The method thus relies for its speed of convergence on the occasional global iterations.

It should be mentioned that Bertsekas and Tsitsiklis [8, Prop.7.5.2] study similar methods and prove that such methods converge, for step sizes smaller than a parameter \(\gamma_0\), provided that: (i) the functions are Lipschitz-continuous, (ii) they comply with a descent property

\[
F_i^T J_{si} s_i \leq -\|s_i\|^2 / \beta \quad \forall i
\]

for every search direction \(s_i\), where \(\beta > 0\) and \(\|s_i\| \geq K\|F_i^T J_{si}\|\), and (iii) that the latest estimates are exchanged within some set, finite time. The parameter \(\gamma_0\) depends on the
11.8. RESULTS WITH THE MULTILEVEL METHOD

Lipschitz constant $\gamma$, the value of $\beta$, the number of search directions $q$ and the number of local iterations passing between global exchanges of estimates $B$. Then

$$\gamma_0 = \frac{1}{\gamma\beta(1 + B + qB)}$$

For the multilevel method the descent property is satisfied, as $s_i = -J_{i}^{-1}F_i$, provided that $\|J_{i}^{-1}\|$ is bounded. The latter is a natural condition, as a similar condition is required to demonstrate the convergence of the Newton method. For the multilevel method convergence is guaranteed in a neighbourhood of the solution, provided that all these conditions are met and a global iteration is executed after every two or three local iterations.

11.8 Results with the multilevel method

Some initial results are presented in table 11-3. As the number of global iterations is small, this method is promising. An added advantage of doing several inner iterations is that it allows the regulated busses to change to their final state. Every change of a regulated bus has a major impact on the solution process. Using corrections, as described in section 11.5, improves the convergence slightly. For the smaller systems it is advantageous to use only one inner iteration throughout, as the difference in the errors is too slight to be considered significant.

The question is how the number of inner versus global iterations should be chosen. There are no hard and fast rules, as is clear from the discussion of the convergence. In appendix F tables with convergence under different circumstances are given.

The rules that were finally used are the following:

- In the first iteration, 2 inner iterations are done and subsequently only 1.
- If, however, the error in the power mismatch is $\|\mathbf{r}\|_\infty < 0.1$, no inner iterations are done.
- No corrections are done, as this makes the method slower.

Many tests with and without corrections were performed, and it was found that doing the corrections improves the final error slightly, but only very rarely reduces the iteration count. Consequently the method is slower when corrections are used.

The times and final errors for the multilevel method and the standard Newton method on an Intel iPSC/860 are given in table 11-4. A direct comparison of the normal Newton method and the multilevel Newton method for the different systems is given in figures 11-2 to 11-5. These show little difference between the methods.
### CHAPTER 11. REDUCED METHODS

The error $\|r\|_\infty$ at different stages of the multilevel method.

<table>
<thead>
<tr>
<th>System</th>
<th>It</th>
<th>Inner</th>
<th>Outer</th>
<th>Inner</th>
<th>Outer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieie14</td>
<td>0</td>
<td>0.21936</td>
<td>0.921936</td>
<td>0.921936</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.465590</td>
<td>0.044278</td>
<td>0.337666</td>
<td>0.022174</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.010038</td>
<td>0.000020</td>
<td>0.004198</td>
<td>0.000002</td>
</tr>
<tr>
<td>iee30</td>
<td>0</td>
<td>0.931678</td>
<td>0.931678</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.469477</td>
<td>0.052381</td>
<td>0.478859</td>
<td>0.037734</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.020255</td>
<td>0.000086</td>
<td>0.012477</td>
<td>0.000020</td>
</tr>
<tr>
<td>iee57</td>
<td>0</td>
<td>2.865400</td>
<td></td>
<td></td>
<td>2.865400</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.283219</td>
<td>0.047045</td>
<td>1.373532</td>
<td>0.050754</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.048857</td>
<td>0.000572</td>
<td>0.061357</td>
<td>0.000525</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.010054</td>
<td>0.000039</td>
<td>0.00393</td>
<td>0.000000</td>
</tr>
<tr>
<td>iee118</td>
<td>0</td>
<td>5.890557</td>
<td></td>
<td></td>
<td>5.890557</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.869620</td>
<td>0.127249</td>
<td>1.710936</td>
<td>0.100862</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.798716</td>
<td>0.017260</td>
<td>0.020800</td>
<td>0.000047</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.011774</td>
<td>0.000005</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11-3: Results for the multilevel algorithm with and without corrections. All results were achieved with 2 inner iterations in the first global iterations, and subsequently 1 inner iteration.

<table>
<thead>
<tr>
<th>Proc.</th>
<th>System</th>
<th>Newton Method</th>
<th>Multilevel Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Iterations</td>
<td>Error</td>
</tr>
<tr>
<td>3</td>
<td>ieie118</td>
<td>4</td>
<td>0.000516</td>
</tr>
<tr>
<td></td>
<td>pow236</td>
<td>4</td>
<td>0.000656</td>
</tr>
<tr>
<td></td>
<td>pow529</td>
<td>4</td>
<td>0.000796</td>
</tr>
<tr>
<td></td>
<td>pow795</td>
<td>6</td>
<td>0.000004</td>
</tr>
<tr>
<td></td>
<td>pow1419</td>
<td>4</td>
<td>0.000923</td>
</tr>
<tr>
<td>7</td>
<td>pow236</td>
<td>4</td>
<td>0.000656</td>
</tr>
<tr>
<td></td>
<td>pow529</td>
<td>4</td>
<td>0.000796</td>
</tr>
<tr>
<td></td>
<td>pow795</td>
<td>6</td>
<td>0.000004</td>
</tr>
<tr>
<td></td>
<td>pow1419</td>
<td>4</td>
<td>0.000923</td>
</tr>
<tr>
<td>12</td>
<td>pow236</td>
<td>4</td>
<td>0.000656</td>
</tr>
<tr>
<td></td>
<td>pow529</td>
<td>4</td>
<td>0.000796</td>
</tr>
<tr>
<td></td>
<td>pow795</td>
<td>6</td>
<td>0.000004</td>
</tr>
<tr>
<td></td>
<td>pow1419</td>
<td>4</td>
<td>0.000923</td>
</tr>
</tbody>
</table>

Table 11-4: A comparison between the Newton method and the multilevel method on an Intel iPSC/860.
Figure 11-2: Comparison of the run times for the Newton and the multilevel method on an iPSC/860 for the system pow236.

It should, however, be pointed out that on both 7 and 12 processors, the multilevel had not converged by a very small margin in the same number of iterations as on 3 processors, so that an additional iteration had to be done. This resulted in considerable additional work and a solution with a much higher precision than required, and much higher than for the equivalent Newton method.

On newer machines communication tends to be even slower relative to computation, so that the ability to reduce the number of global iterations is becoming ever more important. These results demonstrate that the techniques discussed in this chapter can reduce the number of global iterations for the load flow problem. However all these methods depend on having good starting values and sacrifice the global convergence properties of the Newton method with line-searches. As the run-times are very short, it seems preferable to use the Newton-method with branch-tearing whenever possible.
Figure 11-3: Comparison of the run times for the Newton and the multilevel method on an iPSC/860 for the system pow529.

Figure 11-4: Comparison of the run times for the Newton and the multilevel method on an iPSC/860 for the system pow795.
Figure 11-5: Comparison of the run times for the Newton and the multilevel method on an iPSC/860 for the system pow1419.
Chapter 12

Conclusions and further work

The work presented in this thesis was governed by a desire to find faster, parallel algorithms for distribution problems, in this case load flow analysis. The results of this work are also applicable to problems in the area of gas and water distribution as well as circuit simulation. Not only is this a fascinating problem, combining the modelling of electrical components, with graph theory, linear algebra, nonlinear function solution and parallel computing methodologies, but it also provides an opportunity to apply mathematical theory in practice.

The original contributions of this thesis are:

* A unified view of the models for power system components that are required for load flow analysis (chapter 2).

* A survey of load flow analysis methods, investigating their convergence characteristics and explaining the limits to some obvious parallelisation approaches (chapter 3).

* A detailed comparison of direct and iterative methods for the solution of linear systems of equations arising in load flow in part II.

* The introduction of the idea of reduced systems, and the application of this idea to provide an effective parallel solution method for linear systems (chapter 6).

* A generic, fast partitioning approach, that is useful for both node and branch tearing (chapter 7).

* An investigation into various parallel methods for the solution of nonlinear systems of equations (part III).

* An implementation and comparison of all these algorithms.
The methods investigated in this dissertation demonstrate that it is possible to achieve improvements in solution time on a parallel machine for these very difficult problems — even if the algorithms are compared to a very efficient sequential implementation. Sparse matrix techniques are essential to keep the problem in memory and to minimise the computational requirements. Here traditional sparse matrix techniques, as described for example in Duff et al. [32], proved to be very effective. A remarkable result is that the use of branch-tearing leads to more efficient linear system solution methods than node-tearing.

Another important factor in the efficient implementation of any numerical method is the exploitation of vendor-provided libraries such as the BLAS routines. These can provide significant improvements in computation time. Any comparison between single-processor and multi-processor algorithms that ignores the availability of optimised libraries is of little value, as their use has a bigger effect on a single processor implementation.

It also became clear that the effectiveness of these methods to a specific problem depend crucially on the following aspects:

- The size of the problem.
- Whether good starting values are available.
- How badly conditioned the problems are.
- Whether a ‘good’ partitioning of the underlying graph can be found.
- Whether there are additional difficulties in the application, such as constraints on some variables.

As a general rule it is true to say that larger problems benefit more from parallelisation. The main reason is that larger graphs provide more scope for finding a good partitioning, and that large problems have a better computation to communication ratio. Due to the lack of data for large problems, the test cases considered here were only of moderate size. Nevertheless it was possible to demonstrate (albeit modest) speed-ups on up to 12 processors. For larger problems the situation should improve dramatically.

All the load flow problems considered here are of moderate size and most of the benefit of the parallel algorithms is achieved on 8 processors or less. As these algorithms require significant effort to implement and maintain, a compelling reason for the improved solution time must be found to justify the coding effort. If the load flow method is part of long simulation runs, a 50 percent improvement in the load flow solution time may be ample justification, but otherwise load flow simulations will have to get much larger to warrant the implementation of these methods.

Furthermore the advent of very fast microprocessors, such as Digital’s Alpha chip, mean that there is a diminishing return from parallelisation. Communication speeds have also
improved dramatically, but have not kept up with the improvements in processor speeds. It should however be noted that parallel algorithms are also needed to fully utilise shared memory machines, and implementations on these machines will benefit from the algorithms presented here.

12.1 Applicability to other areas

Although this thesis focused on the load flow problem, the approaches developed here are also applicable in other areas. Gas and water distribution problems are very similar to load flow analysis in many respects — also with respect to size. Consequently they also suffer from limited problem size, which makes the practical application of these parallel algorithms hard to justify.

In circuit simulation, however, the picture is different. Circuit sizes are growing even faster than computer speeds increase, and the simulation of large circuits can take days. The most commonly used parallel algorithms used in circuit simulation employ a Newton-Raphson method and use node-tearing to solve the linear systems in parallel. As shown in this thesis, much more effective algorithms are available. A further advantage is that, in all cases the author is familiar with, circuit simulation does not require constraints on some variables — or the related changes in the system that is solved. This will improve the efficiency of the parallel algorithms enormously, as the amount of communication that is required can be reduced significantly.

For circuit simulation the improved scaling achieved by the algorithms described here will be of particular benefit, as it will make the use of large parallel computers more efficient. With node-tearing there are virtually no circuit simulation problems that still give a speed improvement beyond 10 to 20 processors and the new algorithms promise to change this.

The multilevel algorithm described in chapter 11 may prove particularly beneficial for circuit simulation. Often large parts of the circuit barely change from one time-step to the next, and solving a partitioned nonlinear system in parallel to a specified precision automatically takes advantage of this fact.

12.2 Further work

It is necessary to test these algorithms on larger problems. The solution time for the test cases considered here is so short, that it is hard to justify the implementation effort for the parallel algorithms.

More work is needed to understand the convergence characteristics of the multi-level method better. The strategy outlined in chapter 11 works well for all the test cases con-
12.2. FURTHER WORK

sidered, but to use the method effectively on more general problems a better understanding of the trade-offs between local and global iterations is required.

It would also be of interest to see a detailed comparison of the algorithm proposed by Zhang et al. [142] with the multilevel method. It seems that the multilevel method may scale better than the algorithm of Zhang, as it avoids the bottle-neck caused by their use of a node-torn system. On the other hand Zhang's algorithm relies less on old estimates of the solution in the partitioning set, and may converge faster.

These algorithms should be modified and applied to circuit simulation problems, where they will be of much greater benefit than on the relatively small load flow problems.

A last area that could be interesting for future investigation is to formulate the load flow problem as a constrained optimisation problem. The objective function of such a problem would be for example the Euclidean norm of the power mismatches, and the equality constraints would be the load flow equations. For every PV bus there would be inequality constraints on the reactive power.

In general

1. Constrained optimisation problems tend to be much harder to solve than nonlinear systems of algebraic equations.

2. The problem loses a lot of its structure, as there are now 3 variables (reactive power, voltage magnitude and voltage angle) for every PV bus.

3. The ad-hoc approach generally used to handle constraints in load flow problems works rather well.

On the other hand good starting values are available, and the additional freedom offered through the inclusion of all variables may lead to faster convergence— after all, a transition of a PV bus to a PQ bus has a large impact on the performance of the load flow solvers used here. Constrained optimisation methods may also be more robust, as no convergence is shown for the interaction between regulated bus changes and the nonlinear system solution steps in the methodsXS discussed in this thesis; in fact, constraint zig-zagging occurs frequently.

A further motivation to look at constrained optimisation techniques is that many of the problems power utilities want to solve involve optimisation problems of which load flow is just one aspect. A very important example is optimal power flow, which is used to allocate generating capacity according to economic and/or environmental constraints.
Appendix A

The operators $\otimes$ and $\odot$

To simplify the representation of the load flow equations the two operators $\otimes$ and $\odot$ are introduced. These operators define component-wise operations on matrices and vectors.

**Definition A.1** 1. $v \otimes w$, $v,w \in \mathbb{R}^n$ implies component-wise multiplication, i.e.

$$
\begin{pmatrix}
  v_1 \\
  \vdots \\
  v_n \\
\end{pmatrix} \otimes
\begin{pmatrix}
  w_1 \\
  \vdots \\
  w_n \\
\end{pmatrix} =
\begin{pmatrix}
  v_1 \cdot w_1 \\
  \vdots \\
  v_n \cdot w_n \\
\end{pmatrix}
$$

2. $v \odot w$, $v,w \in \mathbb{R}^n$ implies component-wise division, i.e.

$$
\begin{pmatrix}
  v_1 \\
  \vdots \\
  v_n \\
\end{pmatrix} \odot
\begin{pmatrix}
  w_1 \\
  \vdots \\
  w_n \\
\end{pmatrix} =
\begin{pmatrix}
  v_1/w_1 \\
  \vdots \\
  v_n/w_n \\
\end{pmatrix}
$$

3. Analogously, for $v \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$, the operations $A \otimes v$ and $A \odot v$ define the multiplication or division of the row $A_i$ by $v_i$ for $i = 1, \ldots, n$.

The operator $\odot$ is associative, distributive with $+$ and commutative.

**Lemma A.2** If $v$, $w$, $u \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$, then

1. $v \otimes (w \otimes u) = (v \otimes w) \otimes u$

2. $v \otimes (w + u) = v \otimes w + v \otimes u$

3. $v \otimes w = w \otimes v$

4. $v \otimes (A \cdot w) = (v \otimes A) \cdot w$

218
**Proof:** The first and third rule follow immediately from the fact that multiplication is associative and commutative, while the second rule follows from the distributive rule for multiplication and addition.

The last rule follows from the first and the second when writing out the matrix-vector multiplication.

As the Jacobian is frequently computed, some differentiation rules are needed. For the differential operator $D_x$, defined as

$$
\left( \frac{\partial}{\partial x_1}, \cdots, \frac{\partial}{\partial x_n} \right)
$$

**Theorem A.3** If $x, v \in \mathbb{R}^n$ and $A, I \in \mathbb{R}^{n\times n}$ with $I$ the identity matrix, then

1. $D_x(x \otimes v) = v \otimes I$
2. $D_x(x \otimes Av) = Av \otimes I$
3. $D_x(v \otimes Ax) = v \otimes A$
4. $D_x(x \otimes Ax) = x \otimes A + Ax \otimes I$

**Proof:** Note that 2 follows from 1 if we define a vector $c = Av$. Equation 1 is easily seen, as

$$D_x\begin{pmatrix} x_1 \cdot v_1 \\ x_2 \cdot v_2 \\ \vdots \\ x_1 \cdot v_1 \end{pmatrix} = \begin{pmatrix} v_1 & 0 & \cdots & 0 \\ 0 & v_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \cdots & 0 & v_n \end{pmatrix} = v \otimes I.$$

Equation 3 is just as easy to see, by writing out the equations

$$D_x\begin{pmatrix} v_1 \cdot (a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n) \\ v_2 \cdot (a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n) \\ \vdots \\ v_n \cdot (a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n) \end{pmatrix} = \begin{pmatrix} v_1 a_{11} & v_1 a_{12} & \cdots & v_1 a_{1n} \\ v_2 a_{21} & v_2 a_{22} & \cdots & v_2 a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_n a_{n1} & v_n a_{n2} & \cdots & v_n a_{nn} \end{pmatrix} = v \otimes A.$$

As $\otimes$ is only a simple multiplication of each row of a matrix with an element of a vector, it is seen that the rule for the differentiation of products holds. 4 thus follows immediately from 2 and 3. \qed
Appendix B

LU-decomposition of the Jacobian

It is important to realise the effect of partitioning a power system into areas which do not include a swing bus. When transferring the ideas of diakoptics to nodal analysis, the input/output of a subcircuit is specified as a parameter, i.e. included in $S_{sp} = P_{sp} + jQ_{sp}$. The tie lines are subsequently not included in the mismatch equations. Consider the equations for a 2 bus partition, connected by a single transmission line,

\begin{align}
  P_1 &= P_1^{sp} + V_1^2 G + V_1 V_2 (G \cos \theta_{12} + B \sin \theta_{12}) \\
  Q_1 &= Q_1^{sp} - V_1^2 (B + B_C) + V_1 V_2 (G \sin \theta_{12} - B \cos \theta_{12}) \\
  P_2 &= P_2^{sp} + V_1 V_2 (G \cos \theta_{21} + B \sin \theta_{21}) + V_2^2 G \\
  Q_2 &= Q_2^{sp} + V_1 V_2 (G \sin \theta_{21} - B \cos \theta_{21}) - V_2^2 (B + B_C)
\end{align}  

where the admittance of the transmission line is $G + jB$ and the capacitance of the transmission line is $B_C$. In the following the capacitance of the transmission line is ignored, but it does not change the argument. The full Jacobian for this problem, as well as the first step in the elimination process with both single and 2 × 2-pivots is given in tables B-1 through B-2. These calculations show clearly that the Jacobians are rank-deficient. This is not surprising, as the intuitive idea behind the nodal model for circuits is that the inputs and outputs of all nodes add up to zero.
Table B-1: The Jacobian for the sample system.

\[
J = \begin{pmatrix}
(B \cos \theta - G \sin \theta)V_1 V_2 & 2GV_1 + (G \cos \theta + B \sin \theta)V_2 & (G - B \cos \theta)V_1 V_2 & (G \cos \theta + B \sin \theta)V_1 \\
(G \cos \theta + B \sin \theta)V_1 V_2 & -2BV_1 + (G - B \cos \theta)V_2 & -(G \cos \theta + B \sin \theta)V_1 V_2 & (G - B \cos \theta)V_1 \\
-(B \cos \theta + G \sin \theta)V_1 V_2 & (G \cos \theta - B \sin \theta)V_2 & (B \cos \theta + G \sin \theta)V_1 V_2 & (G \cos \theta - B \sin \theta)V_1 + 2GV_2 \\
-(G \cos \theta + B \sin \theta)V_1 V_2 & -(B \cos \theta + G \sin \theta)V_2 & (G \cos \theta - B \sin \theta)V_1 V_2 & -(B \cos \theta + G \sin \theta)V_1 - 2BV_2
\end{pmatrix}
\]  

(B-6)

Table B-2: The first elimination step with a 2 \times 2 pivot.

\[
J^{(2)} = \begin{pmatrix}
1 & 0 & -1 & \frac{2V_1 \sin \theta}{(2V_1 \cos \theta + V_2)V_2} \\
0 & 1 & 0 & \frac{V_1}{2V_1 \cos \theta + V_2} \\
0 & 0 & \frac{2G(V_1 \sin^2 \theta + (V_1 \cos \theta + V_2)^2)}{2V_1 \cos \theta + V_2} \\
0 & 0 & \frac{-2B(V_2 \sin^2 \theta + (V_1 \cos \theta + V_2)^2)}{2V_2 \cos \theta + V_2}
\end{pmatrix}
\]  

(B-7)
Appendix C

Matrix vector multiplications

Matrix-vector-multiplications tend to be the most expensive part of many iterative algorithms. As they usually require a large number of multiplications the question is whether a speed-up can be obtained by vectorising these multiplications.

The following variables will be used in the discussion

- **NN** — Number of rows in the matrix
- **NZ** — Number of nonzero elements in the matrix
- **A** — Vector containing all the nonzero elements in the matrix
- **Row** — Indices of the first nonzero element in every row
- **Col** — Column index of every nonzero element in A
- **X** — Vector with which to multiply the matrix
- **B** — Result of the multiplication

The basic matrix-vector multiplication "$B = Ax$" is written as

\[
\begin{align*}
\text{DO } & I = 1, \text{NN} \\
& B(I) = 0.0 \\
& \text{DO } J = \text{Row}(I), \text{Row}(I+1)-1 \\
& \quad B(I) = B(I) + A(J) \times X(\text{Col}(J)) \\
& \text{END DO} \\
& \text{END DO}
\end{align*}
\]

(MVMUL 1)

As all nonzero elements of the matrix are stored in a long vector, and a second vector with their column indices is available, all the multiplications could be done with the following loop

\[
\begin{align*}
\text{DO } & I = 1, \text{NZ} \\
& WX(I) = A(I) \times X(\text{Col}(I)) \\
& \text{END DO}
\end{align*}
\]

The problem now is to do the partial summations of \(WX\) to get the results for every row. Three different methods were used:
1. Summing the elements in a loop as in MVMUL 1.

2. Summing the elements element-wise to get longer vectors.

3. Computing masked cascade sums of the vector.

The first method is simply

\[
\begin{align*}
\text{DO } & I = 1, \text{ NN} \\
& B(I) = 0.0 \\
& \text{DO } J = \text{Row}(I), \text{Row}(I+1)-1 \\
& \quad B(I) = B(I) + WX(J) \\
\end{align*}
\]

(MVMUL 2)

\[
\text{END DO}
\]

\[
\text{END DO}
\]

For the second of these methods an initial computation of the maximum row length and all individual row lengths are needed. The advantage is that one now has long vectors that are being added to B in every step, but the disadvantage is that the indexing into the array WX is complicated, as it uses an index vector as well as an increment.

\[
\begin{align*}
\text{DO } & I = 1, \text{ NN} \\
& \text{RowLen}(I) = \text{Row}(I+1)-\text{Row}(I) \\
\end{align*}
\]

(MVMUL 3)

\[
\text{END DO}
\]

\[
\text{DO } \text{Step} = 0, \text{MaxRowLength} \\
\text{DO } I = 1, \text{ NN} \\
\text{IF } (\text{RowLen}(I) \text{.GT. Step}) \text{ THEN} \\
\quad B(I) = B(I) + WX(\text{Row}(I)+\text{Step}) \\
\text{END IF} \\
\text{END DO} \\
\text{END DO}
\]

For the last method considered here, every row is summed as a cascade sum. In order to ensure that the correct elements are summed, a mask is used. The mask is initialised to the last element in every row:

\[
\begin{align*}
\text{DO } & I = 1, \text{ NZ} \\
& \text{LM}(I) = .\text{FALSE.} \\
\end{align*}
\]

(MVMUL 4a)

\[
\text{END DO}
\]

\[
\text{DO } I = 2, \text{ NN+1} \\
\quad \text{LM}(	ext{Row}(I)-1) = .\text{TRUE.} \\
\end{align*}
\]

\[
\text{END DO}
\]

Subsequently the whole array is shifted by one position, and added to itself, except in the positions that have been masked out. The mask is then updated and the process repeated.
APPENDIX C. MATRIX VECTOR MULTIPLICATIONS

Matrix **m795.3** (N=1590, NZ=12844)

<table>
<thead>
<tr>
<th>Method</th>
<th>Multiply</th>
<th>Summation</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVMUL 1</td>
<td>7.0 ms</td>
<td>1.0 ms</td>
<td>7.0 ms</td>
</tr>
<tr>
<td>MVMUL 2</td>
<td>6.0 ms</td>
<td>2.0 ms</td>
<td>8.6 ms</td>
</tr>
<tr>
<td>MVMUL 3</td>
<td>5.6 ms</td>
<td>3.0 ms</td>
<td>25.0 ms</td>
</tr>
<tr>
<td>MVMUL 4</td>
<td>6.0 ms</td>
<td>3.0 ms</td>
<td>46.0 ms</td>
</tr>
</tbody>
</table>

Matrix **m1419.3** (N=2838, NZ=23052)

<table>
<thead>
<tr>
<th>Method</th>
<th>Multiply</th>
<th>Summation</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVMUL 1</td>
<td>11.0 ms</td>
<td>1.0 ms</td>
<td>11.0 ms</td>
</tr>
<tr>
<td>MVMUL 2</td>
<td>10.0 ms</td>
<td>2.0 ms</td>
<td>15.2 ms</td>
</tr>
<tr>
<td>MVMUL 3</td>
<td>10.2 ms</td>
<td>3.0 ms</td>
<td>45.0 ms</td>
</tr>
<tr>
<td>MVMUL 4</td>
<td>10.0 ms</td>
<td>3.0 ms</td>
<td>81.0 ms</td>
</tr>
</tbody>
</table>

Table C-1: Timings for various versions of the sparse matrix-vector multiplication gained on a single node of the Intel iPSC/860.

```plaintext
Step = 1
DO WHILE (Step .LT. MaxRowLength)
  DO J = 1, NZ-Step
    IF (.NOT. LM(J)) THEN
      WX(J) = WX(J) + WX(J+Step)
    END IF
  END DO
  IF (Step*2 .LT. MaxRowLength) THEN
    DO J = 1, NZ-Step
      LM(J) = (LM(J) .OR. LM(J+Step))
    END DO
    END IF
    Step = Step*2
  END DO
  DO I = 1, NN
    B(I) = WX(Row(I))
  END DO
```

This type of algorithm is very well suited for data parallel machines, as the computation of all the additional partial sums comes at no extra cost. It was hoped that the advantage of having long vectors with simple indexing would outweigh the additional cost of the extra summations.

Results, which are presented in table C-1, were obtained for two of the test matrices on
a single node of an Intel iPSC/860. The programs were compiled with the following options

    if77 -O4 -Mvect=recog,transform

The results were obtained for the test matrices m795.3 and m1419.3 with 1590 and 2838 rows respectively. The results show that the conventional matrix-vector-multiplication (MVMUL 1) is by far the fastest. The reason is probably that the summation and multiplication pipes can be coupled, so that a multiplication and addition can be done in every clock-cycle once the pipes are full. For both matrices the minimum row length is only 4, which is too short to make effective use of pipelining, but the maximum row length is 20, which is long enough to get a real speed-up by using the pipes. In both cases the average row length is just over 8, which indicates that pipelining may be effective. An additional advantage, which may have a bigger influence than pipelining, is that there is a constant flow of data through the machine, so that there won't be as many cache misses as for the other algorithms.

For this reason no real advantage can be gained by splitting the addition and the multiplications from each other, as can be seen clearly from the fact that (MVMUL 2) is slower than (MVMUL 1). This may also be due to the fact that all data will have to be loaded into cache twice for (MVMUL 2) and only once for (MVMUL 1).

With the information that can be obtained from these runs it is not possible to say whether vectorisation will be effective on some pipelined or data parallel machines, where the cache misses could be avoided. What is clear though, is that the standard matrix-vector multiplication is by far the fastest method on the iPSC/860.
Appendix D

Estimate of the SOR parameter $\omega$

In order to get an estimate for the parameter $\omega$ needed for the block SOR method, tests were done with values for $\omega$ varying between 1.2 and 1.67. The results for five different sets of data are given in table D-1. The number of iterations is also plotted in figure D-1. The optimal value for $\omega$ varies between 1.40 and 1.55, and a better choice tends to make a big difference in the number of iterations that are needed for the SOR method.

In table D-2 the results for the systems $m795$ and $m1419$, both partitioned into three blocks, are given. The results for block-SOR as well as Bi-CGStab with block-SOR preconditioning are given. A different value for $\omega$ should be used if the block-SOR method is used as a preconditioner. In general the same value for $\omega$ as for the block-SOR method itself gives quite reasonable results. From these two sets of results at least, it seems as if a smaller value for $\omega$ is slightly better in the case of using the block-SOR method as a preconditioner.

A value for $\omega$ of around 1.45 always seems to give reasonable results, and this value was used throughout. The reason for using a constant value is that it isn’t practical to do a set of tests before every run, so that in the absence of a formula for choosing $\omega$ a reasonable constant must be used.
<table>
<thead>
<tr>
<th>Omega</th>
<th># Iter</th>
<th>Estimated</th>
<th>Real</th>
<th>Omega</th>
<th># Iter</th>
<th>Estimated</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.20</td>
<td>40*</td>
<td>4.26 10^{-4}</td>
<td>4.65 10^{-5}</td>
<td>1.20</td>
<td>33</td>
<td>9.00 10^{-5}</td>
<td>2.93 10^{-6}</td>
</tr>
<tr>
<td>1.30</td>
<td>38</td>
<td>8.16 10^{-5}</td>
<td>7.08 10^{-6}</td>
<td>1.30</td>
<td>27</td>
<td>7.40 10^{-5}</td>
<td>1.86 10^{-6}</td>
</tr>
<tr>
<td>1.35</td>
<td>33</td>
<td>9.42 10^{-5}</td>
<td>7.18 10^{-6}</td>
<td>1.35</td>
<td>23</td>
<td>9.51 10^{-5}</td>
<td>2.05 10^{-6}</td>
</tr>
<tr>
<td>1.40</td>
<td>29</td>
<td>6.88 10^{-5}</td>
<td>4.51 10^{-6}</td>
<td>1.40</td>
<td>20</td>
<td>6.02 10^{-5}</td>
<td>1.06 10^{-6}</td>
</tr>
<tr>
<td>1.42</td>
<td>27</td>
<td>6.47 10^{-5}</td>
<td>3.94 10^{-6}</td>
<td>1.42</td>
<td>18</td>
<td>5.75 10^{-5}</td>
<td>8.66 10^{-7}</td>
</tr>
<tr>
<td>1.45</td>
<td>23</td>
<td>7.45 10^{-5}</td>
<td>3.90 10^{-6}</td>
<td>1.45</td>
<td>16</td>
<td>5.54 10^{-5}</td>
<td>2.83 10^{-6}</td>
</tr>
<tr>
<td>1.47</td>
<td>20</td>
<td>3.83 10^{-5}</td>
<td>1.11 10^{-6}</td>
<td>1.47</td>
<td>16</td>
<td>7.13 10^{-5}</td>
<td>5.75 10^{-6}</td>
</tr>
<tr>
<td>1.50</td>
<td>21</td>
<td>4.19 10^{-5}</td>
<td>2.24 10^{-6}</td>
<td>1.50</td>
<td>18</td>
<td>5.32 10^{-5}</td>
<td>3.91 10^{-6}</td>
</tr>
<tr>
<td>1.52</td>
<td>23</td>
<td>4.71 10^{-5}</td>
<td>7.29 10^{-7}</td>
<td>1.52</td>
<td>18</td>
<td>7.76 10^{-5}</td>
<td>7.85 10^{-6}</td>
</tr>
<tr>
<td>1.55</td>
<td>25</td>
<td>7.71 10^{-5}</td>
<td>1.30 10^{-6}</td>
<td>1.55</td>
<td>20</td>
<td>5.72 10^{-5}</td>
<td>6.57 10^{-6}</td>
</tr>
<tr>
<td>1.57</td>
<td>27</td>
<td>8.90 10^{-5}</td>
<td>1.06 10^{-6}</td>
<td>1.57</td>
<td>21</td>
<td>9.15 10^{-5}</td>
<td>7.53 10^{-6}</td>
</tr>
<tr>
<td>1.60</td>
<td>31</td>
<td>8.22 10^{-5}</td>
<td>4.19 10^{-7}</td>
<td>1.60</td>
<td>24</td>
<td>5.67 10^{-5}</td>
<td>4.91 10^{-6}</td>
</tr>
<tr>
<td>1.62</td>
<td>35</td>
<td>6.32 10^{-5}</td>
<td>4.22 10^{-7}</td>
<td>1.62</td>
<td>24</td>
<td>8.56 10^{-5}</td>
<td>4.08 10^{-5}</td>
</tr>
<tr>
<td>1.65</td>
<td>40*</td>
<td>8.67 10^{-5}</td>
<td>3.71 10^{-7}</td>
<td>1.65</td>
<td>27</td>
<td>5.66 10^{-5}</td>
<td>9.04 10^{-6}</td>
</tr>
<tr>
<td>1.67</td>
<td>40*</td>
<td>2.27 10^{-4}</td>
<td>1.24 10^{-6}</td>
<td>1.67</td>
<td>29</td>
<td>6.84 10^{-5}</td>
<td>9.21 10^{-6}</td>
</tr>
</tbody>
</table>

Table D-1: Number of iterations needed to converge with the block SOR method for different values of \( \omega \). (* means that the maximum number of iterations was reached.)
Figure D-1: Number of iterations needed to converge to a tolerance of $10^{-5}$ for the Block-SOR method with different values for the parameter $\omega$. All systems were partitioned into three blocks.
<table>
<thead>
<tr>
<th>Omega</th>
<th># Iter</th>
<th>Estimated</th>
<th>Real</th>
<th>Omega</th>
<th># Iter</th>
<th>Estimated</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.20</td>
<td>40</td>
<td>7.05 $10^{-4}$</td>
<td>4.41 $10^{-5}$</td>
<td>1.20</td>
<td>9</td>
<td>3.03 $10^{-5}$</td>
<td>5.39 $10^{-6}$</td>
</tr>
<tr>
<td>1.30</td>
<td>40</td>
<td>1.62 $10^{-4}$</td>
<td>8.23 $10^{-6}$</td>
<td>1.30</td>
<td>9</td>
<td>4.56 $10^{-5}$</td>
<td>3.70 $10^{-6}$</td>
</tr>
<tr>
<td>1.35</td>
<td>38</td>
<td>9.30 $10^{-5}$</td>
<td>4.23 $10^{-6}$</td>
<td>1.35</td>
<td>10</td>
<td>7.53 $10^{-5}$</td>
<td>4.62 $10^{-6}$</td>
</tr>
<tr>
<td>1.40</td>
<td>34</td>
<td>7.88 $10^{-5}$</td>
<td>3.18 $10^{-6}$</td>
<td>1.40</td>
<td>11</td>
<td>4.64 $10^{-6}$</td>
<td>1.84 $10^{-7}$</td>
</tr>
<tr>
<td>1.42</td>
<td>32</td>
<td>8.12 $10^{-5}$</td>
<td>3.11 $10^{-6}$</td>
<td>1.42</td>
<td>12</td>
<td>6.53 $10^{-6}$</td>
<td>8.52 $10^{-7}$</td>
</tr>
<tr>
<td>1.45</td>
<td>29</td>
<td>8.22 $10^{-5}$</td>
<td>2.89 $10^{-6}$</td>
<td>1.45</td>
<td>11</td>
<td>2.71 $10^{-5}$</td>
<td>3.21 $10^{-6}$</td>
</tr>
<tr>
<td>1.47</td>
<td>27</td>
<td>7.83 $10^{-5}$</td>
<td>2.57 $10^{-6}$</td>
<td>1.47</td>
<td>11</td>
<td>5.11 $10^{-5}$</td>
<td>4.06 $10^{-6}$</td>
</tr>
<tr>
<td>1.50</td>
<td>23</td>
<td>8.97 $10^{-5}$</td>
<td>2.55 $10^{-6}$</td>
<td>1.50</td>
<td>11</td>
<td>6.92 $10^{-5}$</td>
<td>7.66 $10^{-6}$</td>
</tr>
<tr>
<td>1.52</td>
<td>20</td>
<td>6.05 $10^{-5}$</td>
<td>2.09 $10^{-6}$</td>
<td>1.52</td>
<td>12</td>
<td>3.73 $10^{-5}$</td>
<td>1.02 $10^{-6}$</td>
</tr>
<tr>
<td>1.55</td>
<td>20</td>
<td>9.70 $10^{-5}$</td>
<td>6.42 $10^{-6}$</td>
<td>1.55</td>
<td>12</td>
<td>7.12 $10^{-5}$</td>
<td>2.10 $10^{-6}$</td>
</tr>
<tr>
<td>1.57</td>
<td>21</td>
<td>5.86 $10^{-5}$</td>
<td>7.47 $10^{-6}$</td>
<td>1.57</td>
<td>12</td>
<td>3.63 $10^{-5}$</td>
<td>3.68 $10^{-6}$</td>
</tr>
<tr>
<td>1.60</td>
<td>23</td>
<td>6.11 $10^{-5}$</td>
<td>7.90 $10^{-6}$</td>
<td>1.60</td>
<td>13</td>
<td>2.63 $10^{-5}$</td>
<td>8.68 $10^{-7}$</td>
</tr>
</tbody>
</table>

Table D-2: Number of iterations needed to converge with the block SOR method and the Bi-CGSTab method with block-SOR preconditioning for different values of $\omega$. 
Appendix E

Matrix structures

In this appendix the matrix structures for different tearing algorithms are given.
Figure E-1: Structure of the matrix m795 after reordering with the minimum degree algorithm.
Figure E-2: Structure of the matrix m795 partitioned for three processors with the branch tearing algorithm.
Figure E-3: Structure of the matrix m795 partitioned for three processors with the node tearing algorithm.
Figure E-4: Structure of the matrix m795 partitioned for seven processors with the branch tearing algorithm.
Figure E-5: Structure of the matrix m795 partitioned for seven processors with the node tearing algorithm.
Appendix F

Convergence tests with the multilevel method

In the following pages diagrams are given which compare the convergence of the multilevel Newton method (described in section 11.6) with a variety of rules for alternating inner and global iterations.

The tests were all executed on an Intel iPSC/860 hypercube on varying numbers of processors. Tests were done for the four largest systems: pow236, pow529, pow795 and pow1419. For descriptions of these systems see section 2.7. For every test run two diagrams are given. The top diagram shows the global error (on a log-scale) as computed after every global iteration. The bottom diagram shows the errors as computed in the reduced iterations on the different processors. The areas shaded grey indicate the periods in which

<table>
<thead>
<tr>
<th>Table</th>
<th>#Proc</th>
<th>Correction</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Subsequent iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-2</td>
<td>3</td>
<td>No</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-3</td>
<td>3</td>
<td>Yes</td>
<td>2</td>
<td>1</td>
<td>.1</td>
</tr>
<tr>
<td>F-4</td>
<td>7</td>
<td>No</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-5</td>
<td>7</td>
<td>No</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>F-6</td>
<td>7</td>
<td>Yes</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-7</td>
<td>7</td>
<td>No</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-8</td>
<td>12</td>
<td>No</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>F-9</td>
<td>12</td>
<td>Yes</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-10$^1$</td>
<td>7</td>
<td>No</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>F-11$^1$</td>
<td>7</td>
<td>No</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

$^1$ In these two tests inner iterations were terminated if $\|r^{inner}\|_\infty < \|r\|_\infty/10$.

Table F-1: Description of the tests carried out with the multilevel Newton method. In all cases no inner iterations were done if the global error $\|r\|_\infty < 0.1$. 

236
reduced iterations were done without intermediate global iterations. Note that only the errors after iterations are shown, so that the first data point in every graph is the error after the first iteration.

Tests were done both with and without corrections (see section 11.5) and the results show that the use of corrections is of no great benefit.

The tests are described in table F-1. In all cases no inner iterations were done if the global error $\|r\|_{\infty} < 0.1$. 
Table F-2: In the first iteration 2 inner iterations are done and subsequently only 1. If \( \|r\|_\infty < 0.1 \) no inner iterations are done. Tests on 3 processors.
Table F-3: In the first iteration 2 inner iterations are done and subsequently only 1. If $\|r\|_{\infty} < 0.1$ no inner iterations are done. A correction step is done before every full iteration. Tests on 3 processors.
Table F-4: In the first iteration 2 inner iterations are done and subsequently only 1. If $\|r\|_\infty < 0.1$ no inner iterations are done. Tests on 7 processors.
Table F-5: In the first two full iterations 2 inner iterations are done and subsequently only 1. If $\|r\|_\infty < 0.1$ no inner iterations are done. Tests on 7 processors.
Table F-6: In the first iteration 2 inner iterations are done and subsequently only 1. If $\|r\|_\infty < 0.1$ no inner iterations are done. A correction step is done before every full iteration. Tests on 7 processors.
Table F-7: In the first iteration 2 inner iterations are done and subsequently only 1. If \( \|r\|_{\infty} < 0.1 \) no inner iterations are done. Tests on 12 processors.
Table F-8: In the first two full iterations 2 inner iterations are done and subsequently only 1. If $\|r\|_\infty < 0.1$ no inner iterations are done. Tests on 12 processors.
Table F-9: In the first iteration 2 inner iterations are done and subsequently only 1. If $\|r\|_\infty < 0.1$ no inner iterations are done. A correction step is done before every full iteration.
Tests on 12 processors.
Table F-10: Inner iterations are terminated if $\|r\|_{\infty} < \|r\|_{\infty}/10$. In the first two full iterations 2 inner iterations are allowed and subsequently only 1. If $\|r\|_{\infty} < 0.1$ no inner iterations are done. Tests on 7 processors.
Table F-11: Inner iterations are terminated if $\|r_{\text{inner}}\|_\infty < \|r\|_\infty/10$. In the first two full iterations 3 and 2 inner iterations are allowed respectively and subsequently only 1. If $\|r\|_\infty < 0.1$ no inner iterations are done. Tests on 7 processors.
References


REFERENCES


REFERENCES


REFERENCES


Index

admittance, 16
  self admittance, 31
shunt, 23
alternating current, 12
Amdahl's law, 96

BLAS, 96, 141
BRANCH, 120
  complexity analysis, 123
  comparision with NODE, 124
  results, 156
branch tearing, 77, 106
  clustering method, 132
bus
  data, 37
  generator, 19
  load, see PQ bus
  PQ, 18
  PV, 19, 33, 34, 72, 174
  regulated, see PV bus
  swing, 19, 33

capacitance, 23
  capacitor bank, 13, 28
cluster, 126
  clustering algorithm, 134
  clustering methods, 132
component, see cluster
conjugate gradient method, 99
corrections, 203
cut set
  branch, 127
  nodal, 127
cut sets, 89
descent direction, 183
diakoptics, 78
double-bordered block structure, 89, 91
elimination tree, 90

Fast Decoupled Load Flow, see load flow,
  PDLF
frontal method, 89
Gauß-Seidel-Newton method, see generalised linear methods
generalised linear methods, 178
  m-step method, 179
  convergence, 180
  Gauß-Seidel-Newton method, 48
  Gauß-Seidel-Newton, 78, 179
  Jacobi-Newton, 78, 179
  Jacobi-Newton method, 47
  SOR-Newton, 179
graph
  sparse matrix, 67, 87
impedance, 16, 23
inductance, 23
Intel iPSC/860, 148
iterative methods
  basic, 101
  Bi-CG, 100
  Bi-CGStab, 100, 142
  block, 103
    reduced, 117
  CG, 99
  CGS, 100, 142
  error estimates, 110
  GMRES, 99
  implementation, 144
  Jacobi, 103, 142
  point, 103
  SOR, 103, 142

Jacobi-Newton method, see generalised linear methods

Jacobian, 45
  calculation of, 68, 74
  partitioning, 170
sparse storage, 67
Kernighan and Lin algorithm, 129
Krylov subspace methods, 98
level sets, 128
line searches, 187
load flow, 11
data, 35
decoupled, 63, 75
decoupled Newton method, 47–49
direct current, 43
equations
polar formulation, 22
rectangular formulation, 22
FDLF, 49–54
BX version, 53
convergence, 53–54
XB version, 52
fixed Jacobi method, 58
flat start, 44, 58
Gauß-Seidel-Newton, 181
Gauß-Seidel, 43
parallel, 78
Jacobian, see Jacobian
linear system
parallel algorithms, 75–78
minimum mismatch, 61
Newton-Raphson, see Newton-Raphson
parallel Jacobi, 78
parallelisation, 74
reordering, 137
SOLF, 54–62
interpretation, 58
PV busses, 57
termination criterion, 73
test systems, 39
LU-decomposition, 69, 142
2 × 2 pivots, 88
fill-in, 87
parallel, 92–95, 120
pivoting, 87
sparse, 72
multi-frontal methods, 89
multilevel method, 206
algorithm, 207
comparison with Newton-Raphson, 211
convergence, 208
results, 209
nested dissection, 89, 127
Newton-Raphson, 44–47, 169
comparison with multilevel, 211
implementation, 64
multilevel method, 206
parallel, 78, 171
results, 73, 174
simplified, 58
nodal model, 16–18
NODE, 95
comparison with BRANCH, 124
complexity analysis, 121
results, 156
node
degree, 126
neighbour, 126
node tearing, 77, 89
clustering method, 133
p.u., see per unit
partitioning, 113
per unit, 19
scaling, 70
transformers, 27
phase angle, 12
power
complex, 15
mismatch, 31
calculation of, 69, 74
reactive, 13
preconditioning, 102
2 × 2 Jacobi, 104
block, 106
eigenvalues, 108
M-matrices, 109
error estimates, 111
from the left, 102
from the right, 102
ILU, 104
Jacobi, 103
MILU, 105
reduced system, 118
SOR, 103
symmetric Jacobi, 104
PVM, 175

reactance
  capacitive, 13
  inductive, 13
  shunt, 29
reduced system
  Jacobi, 118
  linear, 114–117
  nonlinear, 194–197
    algorithm, 197
  parallel, 173
  SOR, 117
reduction ordering, 113
reordering, 87
  of a power system, 137
residual, 99
resistance, 23
results
  descent methods, 190
  linear systems, 151
  multilevel method, 209
  Newton-Raphson, 174
Schur complement, 48, 91
second order load flow, see load flow, SOLF
shunt reactance, 29
sparse data structures, 64–66
  graph, 67, 87
static capacitor, 28
  data, 39
super node, 90
synchronous compensator, 13, 29
test data, 147
testing of code, 146
timing, 149
transformers, 25
  data, 39
    per unit calculations, 27
transmission lines, 23
  data, 36
    nominal-π model, 24
zig-zagging, 190