WRITING SCIENTIFIC SOFTWARE: A GUIDE FOR GOOD STYLE

SUELY OLIVEIRA AND DAVID E. STEWART

University of Iowa

CAMBRIDGE UNIVERSITY PRESS
## Contents

*Preface*  
*Part I  Numerical Software*  
1 Why numerical software?  
1.1 Efficient kernels  
1.2 Rapid change  
1.3 Large-scale problems  
2 Scientific computation and numerical analysis  
2.1 The trouble with real numbers  
2.2 Fixed-point arithmetic  
2.3 Algorithm stability vs. problem stability  
2.4 Numerical accuracy and reliability  
3 Priorities  
3.1 Correctness  
3.2 Numerical stability  
3.3 Accurate discretization  
3.4 Flexibility  
3.5 Efficiency: time and memory  
4 Famous disasters  
4.1 Patriot missiles  
4.2 Ariane 5  
4.3 Sleipner A oil rig collapse  
5 Exercises  

*Part II  Developing Software*  
6 Basics of computer organization  
6.1 Under the hood: what a CPU does  
6.2 Calling routines: stacks and registers  
6.3 Allocating variables  
6.4 Compilers, linkers, and loaders
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7  Software design</td>
<td></td>
</tr>
<tr>
<td>7.1 Software engineering</td>
<td>57</td>
</tr>
<tr>
<td>7.2 Software life-cycle</td>
<td>57</td>
</tr>
<tr>
<td>7.3 Programming in the large</td>
<td>59</td>
</tr>
<tr>
<td>7.4 Programming in the small</td>
<td>61</td>
</tr>
<tr>
<td>7.5 Programming in the middle</td>
<td>67</td>
</tr>
<tr>
<td>7.6 Interface design</td>
<td>70</td>
</tr>
<tr>
<td>7.7 Top-down and bottom-up development</td>
<td>75</td>
</tr>
<tr>
<td>7.8 Don’t hard-wire it unnecessarily!</td>
<td>77</td>
</tr>
<tr>
<td>7.9 Comments</td>
<td>78</td>
</tr>
<tr>
<td>7.10 Documentation</td>
<td>80</td>
</tr>
<tr>
<td>7.11 Cross-language development</td>
<td>82</td>
</tr>
<tr>
<td>7.12 Modularity and all that</td>
<td>87</td>
</tr>
<tr>
<td>8  Data structures</td>
<td></td>
</tr>
<tr>
<td>8.1 Package your data!</td>
<td>90</td>
</tr>
<tr>
<td>8.2 Avoid global variables!</td>
<td>91</td>
</tr>
<tr>
<td>8.3 Multidimensional arrays</td>
<td>92</td>
</tr>
<tr>
<td>8.4 Functional representation vs. data structures</td>
<td>96</td>
</tr>
<tr>
<td>8.5 Functions and the “environment problem”</td>
<td>97</td>
</tr>
<tr>
<td>8.6 Some comments on object-oriented scientific software</td>
<td>106</td>
</tr>
<tr>
<td>9  Design for testing and debugging</td>
<td></td>
</tr>
<tr>
<td>9.1 Incremental testing</td>
<td>118</td>
</tr>
<tr>
<td>9.2 Localizing bugs</td>
<td>120</td>
</tr>
<tr>
<td>9.3 The mighty “print” statement</td>
<td>120</td>
</tr>
<tr>
<td>9.4 Get the computer to help</td>
<td>122</td>
</tr>
<tr>
<td>9.5 Using debuggers</td>
<td>129</td>
</tr>
<tr>
<td>9.6 Debugging functional representations</td>
<td>130</td>
</tr>
<tr>
<td>9.7 Error and exception handling</td>
<td>132</td>
</tr>
<tr>
<td>9.8 Compare and contrast</td>
<td>135</td>
</tr>
<tr>
<td>9.9 Tracking bugs</td>
<td>136</td>
</tr>
<tr>
<td>9.10 Stress testing and performance testing</td>
<td>137</td>
</tr>
<tr>
<td>9.11 Random test data</td>
<td>141</td>
</tr>
<tr>
<td>10 Exercises</td>
<td>143</td>
</tr>
<tr>
<td>Part III  Efficiency in Time, Efficiency in Memory</td>
<td></td>
</tr>
<tr>
<td>11 Be algorithm aware</td>
<td>147</td>
</tr>
<tr>
<td>11.1 Numerical algorithms</td>
<td>149</td>
</tr>
<tr>
<td>11.2 Discrete algorithms</td>
<td>151</td>
</tr>
<tr>
<td>11.3 Numerical algorithm design techniques</td>
<td>153</td>
</tr>
<tr>
<td>12 Computer architecture and efficiency</td>
<td>156</td>
</tr>
<tr>
<td>12.1 Caches and memory hierarchies</td>
<td>156</td>
</tr>
</tbody>
</table>
### Contents

12.2 A tour of the Pentium 4™ architecture 158  
12.3 Virtual memory and paging 164  
12.4 Thrashing 164  
12.5 Designing for memory hierarchies 165  
12.6 Dynamic data structures and memory hierarchies 168  
12.7 Pipelining and loop unrolling 168  
12.8 Basic Linear Algebra Software (BLAS) 170  
12.9 LAPACK 178  
12.10 Cache-oblivious algorithms and data structures 184  
12.11 Indexing vs. pointers for dynamic data structures 185  
13 Global vs. local optimization 187  
13.1 Picking algorithms vs. keyhole optimization 187  
13.2 What optimizing compilers do 188  
13.3 Helping the compiler along 191  
13.4 Practicalities and asymptotic complexity 192  
14 Grabbing memory when you need it 195  
14.1 Dynamic memory allocation 195  
14.2 Giving it back 197  
14.3 Garbage collection 198  
14.4 Life with garbage collection 199  
14.5 Conservative garbage collection 202  
14.6 Doing it yourself 203  
14.7 Memory tips 205  
15 Memory bugs and leaks 208  
15.1 Beware: unallocated memory! 208  
15.2 Beware: overwriting memory! 208  
15.3 Beware: dangling pointers! 210  
15.4 Beware: memory leaks! 214  
15.5 Debugging tools 215  

**Part IV** Tools  
16 Sources of scientific software 219  
16.1 Netlib 220  
16.2 BLAS 220  
16.3 LAPACK 221  
16.4 GAMS 221  
16.5 Other sources 221  
17 Unix tools 223  
17.1 Automated builds: make 223  
17.2 Revision control: RCS, CVS, Subversion and Bitkeeper 226
Contents

17.3 Profiling: prof and gprof 228
17.4 Text manipulation: grep, sed, awk, etc. 230
17.5 Other tools 232
17.6 What about Microsoft Windows? 233

Part V Design Examples 237
18 Cubic spline function library 239
  18.1 Creation and destruction 242
  18.2 Output 244
  18.3 Evaluation 244
  18.4 Spline construction 247
  18.5 Periodic splines 257
  18.6 Performance testing 260

19 Multigrid algorithms 262
  19.1 Discretizing partial differential equations 262
  19.2 Outline of multigrid methods 264
  19.3 Implementation of framework 265
  19.4 Common choices for the framework 272
  19.5 A first test 273
  19.6 The operator interface and its uses 276
  19.7 Dealing with sparse matrices 279
  19.8 A second test 282

Appendix A Review of vectors and matrices 287
  A.1 Identities and inverses 288
  A.2 Norms and errors 289
  A.3 Errors in solving linear systems 291

Appendix B Trademarks 292

References 293

Index 299