Wissenschaftliches Rechnen

Hans-Joachim Bungartz

7. Oktober 2000

Inhaltsverzeichnis

1 Introduction **7**

1.1 Begriffsklärung, Literatur 7

1.2 Dogmatik des wissenschaftlichen Rechnens 14

2 Mathematical Modelling **19**

2.1 Allgemeines 19

2.2 Beispiele mathematischer Modelle 21

2.2.1 Diskrete Modelle 22

Scheduling 22

Entscheidungs- und Wahlmodelle 25

2.2.2 Kontinuierliche Modelle 27

Populationsmodelle 27

Wärmeleitung 31

2.2.3 Stochastische Modelle 33

2.3 Mehrskaligkeit und hierarchische Modelle 38

2.3.1 Hierarchische Modellierung 38

2.3.2 Mehrskaligkeit 39

2.3.3 Mittelung und Homogenisierung 40

3 Numerical Treatment of Mathematical Models **43**

3.1 Back to the Roots: Linear Algebra 44

3.2 Discretizing Ordinary Differential Equations 46

3.2.1 Initial Value Problems 46

3.2.2 Boundary Value Problems 51

3.3 Discretizing Partial Differential Equations 54

3.3.1 Discretizing Space 55

Finite Differences (FD) 55

Finite Elements (FE) 57

Finite Volumes . . . 61
Other Schemes . . . 61
3.3.2 Discretizing Time . . . 61
3.4 Direct Solution of Systems of Linear Equations . . . 62
3.4.1 Gaussian Elimination and Cholesky . . . 62
3.4.2 Nested Dissection . . . 62
3.5 Iterative Solution of Large Sparse Linear Systems . . . 62
3.5.1 Classical Relaxation Techniques . . . 64
Overview . . . 64
Analysis . . . 65
3.5.2 Krylov Subspace Methods . . . 66
Overview . . . 66
Minimizing Quadratic Forms and Steepest Descent . . . 68
Conjugate Directions . . . 70
Conjugate Gradients . . . 72
Preconditioning . . . 73
3.5.3 Multigrid Methods . . . 75
3.6 Nonlinear Equations, Linearization . . . 79
3.6.1 Newton's Method . . . 81
3.6.2 Inexact Newton Methods . . . 82
3.6.3 Quasi-Newton Methods . . . 82
3.7 Grid Generation . . . 83
3.7.1 Structured Grids . . . 84
Composite Grids . . . 84
Block-Structured Grids . . . 84
Elliptic Generators . . . 84
Hyperbolic Generators . . . 85
Algebraic Generators . . . 86
Adaptive Grids . . . 86
3.7.2 Unstructured Grids . . . 87
Delaunay Triangulations, Voronoi Diagrams . . . 87
Point Creation . . . 88
Other Techniques . . . 89
Adaptive Grids . . . 90
3.7.3 Methods for Varying Geometries . . . 90
Front Tracking Methods . . . 91

Front Capturing Methods . . . 91
Clicking and Sliding Mesh Techniques . . . 91

4 Implementation **93**
4.1 Rechnerstrukturen . . . 93
4.1.1 Moderne Mikroprozessoren . . . 93
4.1.2 Supercomputer: Parallele und verteilte Architekturen . . . 98
4.2 Programmiertechnik . . . 107
4.2.1 Programmierparadigmen und Sprachen . . . 107
4.2.2 Algorithmen und Datenstrukturen . . . 110
4.2.3 Aspekte des Software-Engineering . . . 115
4.3 Parallelization . . . 121
4.3.1 Parallel Programming Paradigms . . . 121
4.3.2 Programming Models: MPI and PVM . . . 125
4.3.3 Lastverteilung . . . 128
Allgemeines . . . 128
Lastverteilung mittels raumfüllender Kurven . . . 130
4.3.4 Leistungsbewertung . . . 136
4.4 Numerics for Supercomputers . . . 140
4.4.1 Parallel Numerical Linear Algebra . . . 140
4.4.2 Substructuring and Domain Decomposition . . . 140
4.5 Werkzeuge des wissenschaftlichen Rechnens . . . 141
4.5.1 Programmbibliotheken und -sammlungen . . . 141
GAMS . . . 141
Matrix Market . . . 142
NETLIB . . . 142
Visual Numerics . . . 144
Diffpack . . . 144
NAG . . . 145
Numerical Recipes . . . 145
4.5.2 Werkzeuge für Entwicklung, Analyse und Prototyping von Algorithmen . . . 146
Maple . . . 146
MATLAB . . . 149

5 Embedding and Integration **151**
5.1 Schnittstellen . . . 151
5.2 Gekoppelte Systeme . . . 157

6 Visualization **167**

6.1 Methoden der Bildverarbeitung 168

6.2 Aspekte der Computergraphik 170

6.3 Visualisierung raum- und zeitaufgelöster Simulationsdaten 173

6.4 Vom Bild zum Film 177

6.5 Bunte Bildchen über alles 178

7 Validation **181**

8 Case Study: Computational Fluid Dynamics **183**

8.1 Das mathematische Modell 183

8.2 Die Diskretisierung 183

8.3 Die Implementierung 183

8.4 Die Visualisierung 183

Kapitel 1

Introduction

1.1 Begriffsklärung, Literatur

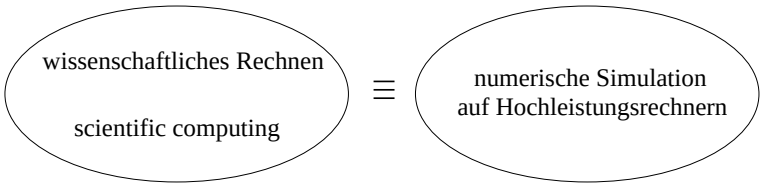


Abbildung 1.1: Was ist wissenschaftliches Rechnen?

Das *Wissenschaftliche Rechnen* (engl. *Scientific Computing*) befasst sich mit der supercomputergestützten numerischen Simulation von Prozessen, wie sie in den Ingenieur-, Natur- oder auch in den Wirtschafts- und Sozialwissenschaften untersucht werden. Als Beispiele für Anwendungsfelder seien genannt die Strömungsmechanik (Aerodynamik von Fahr- und Flugzeugen, Verbrennungsvorgänge, Schadstoffversickerung), die Halbleitertechnologie (Kristallzucht und Oxidationsprozesse, Schaltkreissimulation), die Wetter- und Klimavorhersage (Entstehung und Bewegung von Tornados, globale Erwärmung) sowie die Finanzmathematik (Prognosen von Aktien- und Optionskursen). Dabei tritt die numerische Simulation in zunehmendem Maße als gleichberechtigter Partner neben die beiden klassischen Säulen des Erkenntniserwerbs in den betreffenden Anwendungswissenschaften, die theoretische Untersuchung und das Experiment.

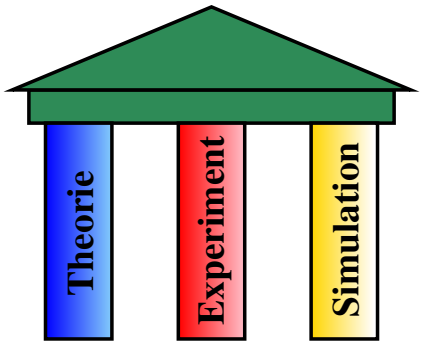


Abbildung 1.2: Die berühmten drei Säulen

Charakteristisch für das wissenschaftliche Rechnen ist, dass praxisrelevante Resultate nur durch

den koordinierten Einsatz aktueller Methoden und Forschungsergebnisse aus unterschiedlichen wissenschaftlichen Fachgebieten erzielt werden können. So bringt die jeweilige Anwendungsdisziplin fachspezifisches und technologisches Know-how sowie die Möglich- und Fertigkeiten zur Verifikation der Simulationsergebnisse mittels experimenteller Techniken ein. Die Mathematik leistet wichtige Beiträge sowohl bei der Modellbildung als auch bei der Entwicklung schneller numerischer Verfahren zur rechnergestützten Lösung der Modelle. Die zentrale Rolle der Informatik schließlich besteht in der Bereitstellung, Nutzbarmachung und Programmierung moderner Hochleistungsrechner sowie in der Entwicklung effizienter Algorithmen und mächtiger Software-Pakete.

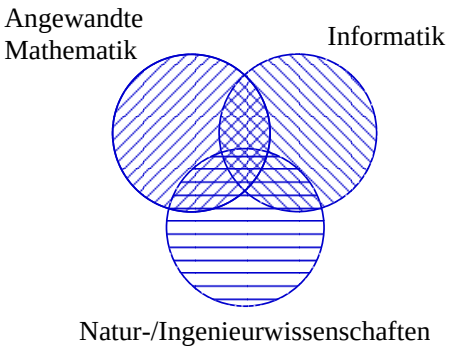


Abbildung 1.3: Im Spannungsfeld ...

Dabei ist es gerade das Zusammenspiel der Disziplinen, die für den Erfolg unabdingbar ist. Ein schlechtes mathematisches Verfahren kann zwar parallelisiert und auf Supercomputern eingesetzt werden, sehr weit kommen wird man dabei aber nicht. Benötigt werden vielmehr eine beständige Fortentwicklung der mathematischen Modelle und numerischen Algorithmen, der Transfer der Algorithmen in leistungsfähige und benutzerfreundliche Software, deren Portierung auf die aktuellen Hochleistungsrechenplattformen sowie eine permanente Leistungssteigerung im Hardware-Sektor selbst. Ein kleines Beispiel soll das veranschaulichen: Die algorithmischen Fortschritte beim Lösen großer schwachbesetzter linearer Gleichungssysteme haben in den letzten Jahrzehnten zu einer vergleichbaren Beschleunigung der Berechnungen geführt wie die Entwicklung der Prozessorleistung in diesem Zeitraum (siehe Abb. 1.4).

Noch ein Wort zur Namensgebung: *Scientific Computing* ist der ursprüngliche englische Terminus. Inzwischen hat sich jedoch eher *Computational Science and Engineering (CSE)* durchgesetzt: Dies gefällt insbesondere den Ingenieuren und Naturwissenschaftlern besser, drückt es doch aus, dass alles nur eine Teildisziplin ihres Tuns ist, was wiederum den Mathematikern weniger gefällt. Die Informatiker ihrerseits sind über *CSE* schon weniger betrübt: Ihnen war die Nähe von *Scientific Computing* zu *Computer Science* ohnehin immer suspekt. Dem auf die Wahrung der deutschen Sprache bedachten Wissenschaftler gefällt der deutsche Begriff *Wissenschaftliches Rechnen* in der Regel besser, doch auch hier lauern Bauchschmerzen: Gibt es denn ein *unwissenschaftliches* Rechnen? Und erinnert uns der Begriff des *Rechnens* nicht zu stark an frühe Grundschultage? Von prominenter Seite wurde deshalb schon das bedeutend intellektueller anmutende Wort von der *Wissenschaftlichen Komputistik* vorgeschlagen, und von da ist es dann nur noch ein kleiner Schritt zur (auch schon, mehr oder weniger ernst, ins Spiel gebrachten) *Scientifistischen Komputistik*. Doch genug davon. CSE hat sich als Begriff etabliert, und es gibt weltweit bereits eine Reihe von Hochschulen, die CSE als eigenständigen (Aufbau-) Studiengang oder als Vertiefungsschwerpunkt anbieten (siehe z. B. `http://www.siam.org/cse/`), darunter die TU München (`http://www.cse.tum.de/`).

Die Notwendigkeit der „dritten Säule Simulation“ kann man daran erkennen, dass das traditionelle Instrumentarium – also theoretische Analyse und Experiment – auf sich alleine gestellt immer öfter an seine Grenzen stößt. So führt die Erfordernis einer möglichst realitätsnahen Beschreibung der zu untersuchenden Prozesse oder Systeme i. d. R. zu Modellen, die sich aufgrund ihres Kom-

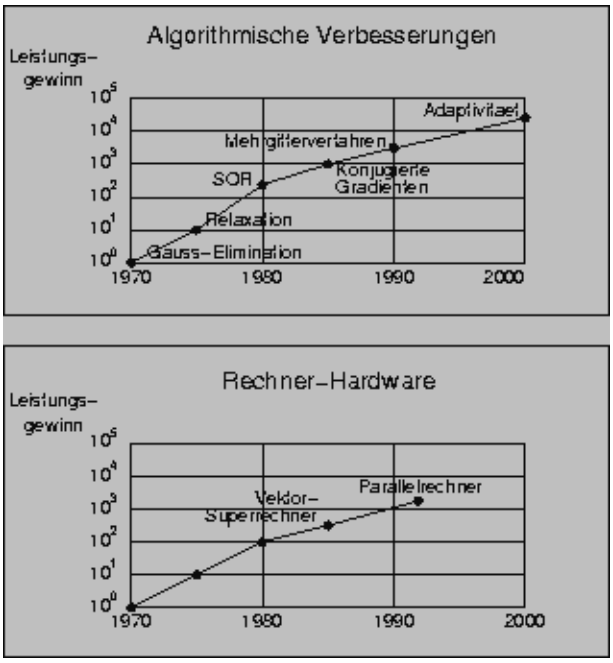


Abbildung 1.4: Leistingsexplosion bei Hard- und Software

plexitätsgrades einer analytischen Behandlung entziehen. Wenn überhaupt, dann sind allenfalls noch Aussagen zur Existenz und Eindeutigkeit von Lösungen zu erwarten, nicht jedoch explizite Formeln zu deren geschlossener Darstellung. Andererseits sind auch den Experimentiermöglichkeiten in vielen Fällen Grenzen gesetzt. Der Treibhauseffekt oder Vorgänge aus der Astrophysik wie beispielsweise die Entstehung von Galaxien laufen bei weitem zu langsam ab (und das ist offensichtlich nur eine der auftretenden Schwierigkeiten), als dass sie im Sinne eines reproduzierbaren Laborexperiments fassbar wären. Chemische Reaktionen weisen häufig die umgekehrte Problematik einer zu hohen Geschwindigkeit auf. Trotz der unbestreitbaren Fortschritte der Messtechnik stellt ferner auch die räumliche Auflösung in vielen Fällen ein Problem dar: Zur Untersuchung von El Niño zum Beispiel sollen Messdaten über schwimmende Sonden flächendeckend (!) im Südpazifik gewonnen werden – klar, dass dies nur in einer sehr groben Auflösung geht. Wer wollte auf der anderen Seite Vorgänge beobachten, die sich auf so kleinen Skalen abspielen, dass sie durch mikroskopische Techniken nicht mehr erfasst werden können? In vielen Bereichen verbieten sich Experimente auch aufgrund unerwünschter (und irreversibler) Begleiterscheinungen – man denke nur etwa an Tests von Kernwaffen – oder mangels eines geeigneten Experimentierfeldes: Die Auswirkungen der Einführung des Euros lassen sich eben nicht durch Feldversuche im klassischen Sinn vorhersagen. Oftmals weist aber ein ganz pragmatischer und zu allen Zeiten hervorragend vermittelbarer Grund in Richtung Simulation: die Kostenfrage. Bei der Entwicklung FCKW-freier Kühlmittel müssen beispielsweise Transportkoeffizienten von Gasen ermittelt werden – im Modell bzw. in der Simulation im Wesentlichen die Berechnung sechsdimensionaler Integrale, im Labor ein mehrere hunderttausend Mark teueres Experiment. In der Automobilindustrie könnten ferner durch simulierte Crash- und – allen Unkenrufen zum Trotz – Elchtests bei der Entwicklung ohne negative Auswirkungen auf die Fahrzeugsicherheit beträchtliche Summen eingespart werden, ja der ganze konventionell gebaute Prototyp könnte bei konsequentem Einsatz simulativer Methoden bald der Vergangenheit angehören.

Some fields of application are always mentioned when people want to emphasize the necessity of numerical simulations (cf. the corresponding above-mentioned statements of SIAM, the Society for Industrial and Applied Mathematics, `http://www.siam.org/cse/`):

- *climate research*:
  Future energy and environmental strategies will require a very high accuracy in order to understand how local or regional events and measures influence the global development. For that, the resolution used in weather forecasting has to be brought to the global predictions, which is not yet feasible because of the very large amounts of data storage and long computation times required. Major advances in CSE will enable scientists to incorporate knowledge about the interactions between the oceans, the atmosphere, and living ecosystems into the models used to predict long-term change, and, hence, to reduce uncertainties regarding long-term climate changes.

- *combustion*:
  Accurate simulations of combustion systems offer the chance to develop the understanding that is necessary to improve efficiency and reduce emissions of combustion processes. Since the combustion of fossil fuels is still the most important source of energy, this is a really crucial point.

- *automobile development*:
  It is now standard in the design of mechanical systems such as vehicles or robots to use computer simulation to observe the dynamic response of the system being designed. Computer-aided design drastically reduces the need to construct and test prototypes. Simulation is used not only to improve performance, but also for safety and ergonomics. Real-time simulation with operator or hardware in the loop presents substantial challenges for algorithms and software.

- *aircraft design*:
  Since the early days of computing, numerical simulations have been used for the desgin and performance analysis of aircraft components like airfoils. As computations become more sophisticated and computers more powerful, the complete design process will be computer-based (the Boeing 777 was the first jet to be completely digitally designed).

- *electronic design automation*:
  Modern electronic systems (including the microprocessors that have enabled CSE to achieve its current importance) are extraordinarily complex. Their development is only possible with the aid of computational tools for their simulation and verification. During the whole development, computation plays an important role: from simulating the processes used to fabricate semiconductor devices (crystal gowth, oxidation, etc.), via simulating and verifying the function of these devices and the logic of a microprocessor system, to laying out the floor plan of VLSI circuitry.

- *biology and medicine*:
  CSE is rapidly becoming indispensable to the biological and medical sciences. Simulation plays a major role in the conceptual development of medical devices (both for diagnostic procedures (electromagnetic, ultrasonic, etc.) and for artificial organs (hearts, kidneys, etc.)). Computational modelling plays a fundamental role in the emerging efforts to combine mathematics and informatics with biology in the genomic sciences (genome sequencing, gene expression profiling, genotyping, etc.). Here, there is also a big economic interest of the pharmaceutical industry (drug design).

- *chemistry and physics*:
  Computational chemistry and computational physics are widely used in academic and industrial research. Typically in today's supercomputing centres, a big part of the overall computing time goes into these fields (think of quantum mechanics (direct and approximate solutions of the Schrödinger equation), protein folding, or astrophysics).

- *material sciences*:
  Here, the challenge is to invent new materials and to perfect existing ones by fabrication and processing so that they have the desired performance and environmental response. For ex-

ample, there are many new and important applications for thin films, including silicon-based microelectronics, compound semiconductors, opto-electronics devices, high-temperature superconductors, and photovoltaic systems. The growth of such thin films, which can be accomplished via processes such as chemical vapoor deposition (CVD), is sensitive to many factors in the manufacturing process. Here, numerical simulations can help to obtain a better understanding of the various influences.

- *financial engineering*:
  While so far mostly topics from science and classical engineering disciplines have dominated the simulation market, economics and finance have entered the stage recently, too. Problems like option pricing, e. g., show the increasing need for a computational approach.

Doch es sind nicht nur die großen Herausforderungen (oder „Grand Challenges“, wie sie in US-Förderprogrammen immer wieder genannt werden), bei denen die numerische Simulation zu Fortschritten beitragen kann, sondern oftmals auch vermeintlich banale Probleme. Durch die Simulation der Dachhinterlüftung konnte beispielsweise die Vermutung bestätigt werden, dass diese für die Reduzierung von Feuchtigkeit und damit für die Erhöhung der Lebensdauer von Dächern von ganz entscheidender Bedeutung ist. Durch Simulation der Temperaturverteilung sowie der Strömung in einem großen Bäckerei-Ofen konnte die optimale Temperatureinstellung ermittelt werden, bei der die Semmeln auch in der dünnen Luft der Zugspitze goldgelb werden. Schließlich stützt sich auch die Entwicklung vieler Hightech-Komponenten in Automobilen (Airbag-Sensoren, Airbag-Systeme, Fahrdynamikregelung etc.) auf den Einsatz simulativer Techniken. Das Ziel solcher Aufgabenstellungen ist nicht, überhaupt etwas machen zu können, sondern Qualität, Zuverlässigkeit und Rentabilität von Produkten und Prozessen zu erhöhen, sei es durch wirtschaftlicheren Einsatz von Ressourcen, durch Reduktion von Abfallstoffen oder durch Herabsetzung des Schadensfallrisikos.

Die Interdisziplinarität des wissenschaftlichen Rechnens spiegelt sich in seinem entsprechend vielschichtigen Aufgabenspektrum wider. Wir wollen bei der Arbeit des numerischen „Simulanten“ (leider lässt sich die kaum ruhmreiche Herkunft des Worts „Simulation“ nicht leugnen) sechs wesentliche Schritte unterscheiden: Modellbildung, numerische Behandlung, Implementierung, Einbettung, Visualisierung und Validierung:

1. *Die Modellbildung.* Am Anfang jeder Simulation steht die Entwicklung eines mathematischen Modells der zu untersuchenden Vorgänge. Dieses soll als vereinfachtes Abbild der Realität möglichst viele relevante Zusammenhänge beschreiben, dabei aber auf eine Art und Weise formuliert sein, die eine rechnergestützte Behandlung gestattet, und ferner (eindeutig) lösbar sein. Abhängig von den jeweiligen Beschreibungsmitteln erhält man unterschiedlichste Modelle, zumeist jedoch ein analytisch nicht zu lösendes System gekoppelter (Differential-) Gleichungen.

2. *Die numerische Behandlung.* Da jeder Rechner nur mit endlich vielen diskreten Zahlen arbeiten kann, müssen die Gleichungen des vorliegenden Modells sowie gegebenenfalls das betrachtete Simulationsgebiet zunächst *diskretisiert* und somit in eine für den Computereinsatz geeignete Form gebracht werden. Für die Lösung des resultierenden diskretisierten Gleichungssystems müssen anschließend effiziente numerische Verfahren bereitgestellt und eingesetzt werden.

3. *Die Implementierung.* Hier ist neben der Auswahl geeigneter Programmiersprachen und Datenstrukturen vor allem die verteilte Bearbeitung von großer Bedeutung, da realitätsnahe Simulationsrechnungen aus Speicherplatz- und Rechenzeitgründen i. d. R. nur auf Rechnernetzen oder auf Supercomputern durchgeführt werden können. Da ferner Simulationscodes üblicherweise sehr umfangreiche Programmpakete darstellen, sind hier auch Techniken des modernen Software-Engineering unerlässlich. Zudem ist zu diskutieren, welche unterstützenden Software-Werkzeuge dem numerischen Simulanten zur Verfügung stehen und wie mit deren Hilfe ein effizientes Prototyping ermöglicht werden kann.

4. *Die Einbettung.* In industriellen Anwendungen stellt die numerische Simulation nur einen

Schritt in der Produktentwicklung dar. Deshalb müssen Schnittstellen bereitgestellt werden, die eine direkte Anbindung der Simulationsprogramme etwa an die CAD-Welt gestatten. Nur so lässt sich beispielsweise die Simulation der Aerodynamik eines Autos zur Ermittlung seines c_w-Wertes bereits frühzeitig in den Design-Prozess der Karosserie integrieren.

5. *Die Visualisierung.* Bei der numerischen Simulation fallen üblicherweise sehr große Datenmengen an, die sich nicht immer wie im zuvor erwähnten Beispiel des Widerstandsbeiwerts in einem oder wenigen Parametern zusammenfassen lassen. Oft ist die Geschwindigkeit im ganzen Strömungsgebiet gefragt, oder es interessiert bei der Optimalsteuerung eines Roboters die Bahnkurve des Roboterarms. Zur Veranschaulichung sowie zur Interpretation solcher Simulationsergebnisse ist deren graphische Aufbereitung und Darstellung, kurz *Visualisierung* genannt, von großer Bedeutung.

6. *Die Validierung.* Wenn nach langen Bemühungen und oftmals nahezu ebenso langen Rechenzeiten endlich ein Simulationsergebnis vorliegt, bei dem nicht offensichtlich ist, dass es unmöglich die Wirklichkeit auch nur annähernd wiedergeben kann, bricht unter den Simulanten oft verfrühter Jubel aus. Jedoch ist gerade die abschließende Phase der Verifikation der Resultate unerlässlich, und hier sind wieder die Expertise des Spezialisten aus der jeweiligen Anwendungsdisziplin sowie – zumindest zur Eichung in einfachen Fällen – experimentell bestimmte Vergleichswerte erforderlich.

Während bei Modellbildung, Numerik und Validierung traditionell die Mathematik bzw. die jeweilige Anwendungsdisziplin den Ton angeben, liegen die Schwerpunkte informatischen Treibens üblicherweise bei den Schritten 3 bis 5. Über die Validierung haben dagegen i. d. R. die Anwender die Hoheit, sieht man einmal von Benchmarks ab, also von präzise definierten Modell-Simulationsläufen, an denen dann die einzelnen mathematischen Modelle und Codes sich (vergleichend) die Zähne ausbeißen können. Da eine solche Validierung von Code A über den Vergleich mit den von Code B produzierten Ergebnissen aber stark an Münchhausen erinnert, der sich selbst an den Haaren aus dem Sumpf zieht, sind Eichungen mit Hilfe experimenteller Daten sicher vorzuziehen.

In dieser Vorlesung beschäftigen wir uns somit hauptsächlich mit den Schritten 1 bis 5. Da hier die unterschiedlichsten Aspekte aus den beteiligten Disziplinen einfließen, gibt es natürlich eine Fülle von Berührungspunkten mit anderen Lehrveranstaltungen:

1. lineare Algebra, Analysis, Funktionalanalysis, Differentialgleichungen, Statistik und Wahrscheinlichkeitstheorie, mathematische Modellbildung, ...

2. numerische Mathematik, Approximationstheorie, Numerik gewöhnlicher und partieller Differentialgleichungen, finite Elemente, iterative Lösung großer Gleichungssysteme, Optimierung, diskrete Mathematik, ...

3. Rechnerarchitektur, Mikroprozessoren, Rechnernetze, parallele Programmierung, Datenstrukturen, rekursive Verfahren und hierarchische Datenstrukturen in der numerischen Analysis, objektorientierter Entwurf, Software-Engineering, Computeralgebra, ...

4. Software-Engineering, geometrische Modellierung, CAD, ...

5. Computergraphik, geometrische Modellierung, ...

Nach der Behandlung der einzelnen Teilaspekte des wissenschaftlichen Rechnens werden wir am Beispiel der numerischen Strömungsmechanik alle Schritte im Stile einer Fallstudie diskutieren. Damit ist die Vorlesung natürlich die geradezu ideale Einstimmung auf das Praktikum „Wissenschaftliches Rechnen und Visualisierung“.

Last, but not least, some references that might be helpful for obtaining further and more detailed information concerning the various topics discussed in the following chapters.

concerning scientific computing as a discipline:

[1] Golub, Ortega: Scientific Computing: An Introduction with Parallel Computing, Academic Press, 1993

[2] Friedman, Littman: Industrial Mathematics, A Course in Solving Real-World Problems, SIAM, 1993

[3] Gander, Hřebíček: Solving Problems in Scientific Computing Using Maple and MATLAB, Springer, 1997

concerning the mathematical modelling:

[1] Dym, Ivey: Principles of Mathematical Modeling, Academic Press, 1980

[2] Jacoby, Kowalik: Mathematical Modeling with Computers, Prentice-Hall, 1980

[3] Krabs: Mathematische Modellierung, Teubner, 1997

[4] Bossel: Modellbildung und Simulation, Vieweg, 1985

concerning the numerical treatment:

[1] Stoer, Bulirsch: Introduction to Numerical Analysis, Springer, 1996

[2] Dongarra, Duff, Sorensen, van der Vorst: Numerical Linear Algebra for High Performance Computers, SIAM, 1998

[3] Hackbusch: Iterative Solution of Large Sparse Systems of Equations, Springer, 1993

[4] Numerical Recipes: see `http://www.nr.com` (C or FORTRAN version available online)

[5] Hackbusch: Elliptic Differential Equations – Theory and Numerical Treatment, Springer, 1992

[6] Briggs: A Multigrid Tutorial, SIAM, 1987

[7] Hackbusch: Multi-Grid Methods and Applications, Springer, 1985

[8] Kelley: Iterative Methods for Linear and Nonlinear Equations, SIAM, 1995

[9] Ortega, Rheinboldt: Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, 1970

[10] Richtmyer, Morton: Difference Methods for Initial-Value Problems, John Wiley, 1967

[11] Anderson, Tannehill, Pletcher: Computational Fluid Mechanics and Heat Transfer, McGraw-Hill, 1984

[12] Gustafsson, Kreiss, Oliger: Time Dependent Problems and Difference Methods, John Wiley, 1995

[13] Bathe: Finite Element Procedures, Prentice Hall, 1996

[14] Braess: Finite Elements, Springer, 1997

[15] Szabo, Babuska: Finite Element Analysis, John Wiley, 1991

[16] Smith, Bjoerstad, Gropp: Domain Decomposition Methods, Cambridge University Press, 1996

[17] Knupp, Steinberg: Fundamentals of Grid Generation, CRC Press, 1993

[18] Thompson, Soni, Weatherill: Handbook of Grid Generation, CRC Press, 1998

concerning the implementation:

[1] Dongarra, Duff, Sorensen, van der Vorst: Numerical Linear Algebra for High Performance Computers, SIAM, 1998

[2] Almasi, Gottlieb: Highly Parallel Computing, Benjamin Cummings, 1994

[3] Gropp, Lusk, Skjellum: Using MPI – Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1999

[4] Geist, Beguelin, Dongarra, Jiang, Manchek, Sunderam: PVM – Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994

[5] Schnekenburger, Stellner: Dynamic Load Distribution for Parallel Applications, Teubner, 1997

[6] Hwang: Advanced Computer Architecture, Mc Graw-Hill, 1993.

[7] Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison Wesley, 1995

[8] Gander, Hřebíček: Solving Problems in Scientific Computing Using Maple and MATLAB, Springer, 1997

concerning the visualization:

[1] Brown, Earnshaw, Jern, Vince: Visualization – Using Computer Graphics to Explore Data and Present Information, John Wiley, 1995

[2] Scientific Visualization: Workshop Material and Web Resource at NCSU, see `http://www2.ncsu.edu/scivis/`

concerning computational fluid dynamics:

[1] Griebel, Dornseifer, Neunhoeffer: Numerische Simulation in der Strömungsmechanik, Vieweg, 1995

[2] Griebel, Dornseifer, Neunhoeffer: Numerical Simulation in Fluid Dynamics, SIAM, 1998

[3] Ferziger, Peric: Computational Methods for Fluid Dynamics, Springer, 1996

1.2 Dogmatik des wissenschaftlichen Rechnens

Durch alle Teilschritte ziehen sich einige Strategien, die sich als vorteilhaft erwiesen haben. Die nachfolgend aufgelisteten „Gebote“ sollen aufzeigen (und festlegen!), wie man im wissenschaftlichen Rechnen arbeiten sollte.

(bislang) 7 Gebote:

(I) rekursiv:

- Datenstrukturen (Bäume, Geflechte)
- Algorithmen (auf Bäumen etc.)

(II) hierarchisch:

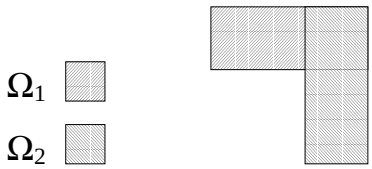
- Datenstrukturen (z. B. record, array und Baum):

```
type   t      =   record      s1: t1;
                               s2: t2;
                               s3: t3
                   end;

type   t      =   array[1..10]  of  real;

type   tree   =   ↑ node;
type   node   =   record      left_son: tree;
                               right_son: tree;
                               value: real
                   end;
```

- Algorithmen (z. B. Gebietszerlegung für PDEs):



$Lu = f$ auf $\Omega = \Omega_1 \cup \Omega_2$;
löse abwechselnd auf Ω_1 und Ω_2
mit entsprechenden Randwerten

- Zahldarstellung:

nicht hierarchisch:	||||||||||||||	(Bierstriche)
hierarchisch:	14	(Dezimalsystem)

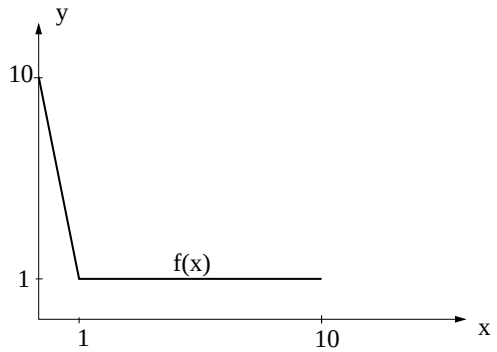
- Modelle:
  Bsp. Katalysator: Interessiert man sich für die genauen Reaktionen zu jeder Zeit und an jedem Ort im Kat (feines Modell mit vielen raumaufgelösten Reaktionsgleichungen), oder sind nur einige wenige Parameter wie etwa der Prozentsatz der herausgefilterten Schadstoffe gefragt (grobes, parametrisiertes Modell ohne Raumauflösung)?

(III) adaptiv:

- Die Auswahl eines effizienten Algorithmus' für ein numerisches Problem ist eine Optimierungsaufgabe unter einer Nebenbedingung:

$$\min_{\text{Aufwand = const}} \|u^{ex} - u^{num}\|.$$

  D. h.: Bei der Diskretisierung sind Punkte dort zu investieren, wo es das Problem erfordert.
  Als Beispiel betrachten wir die numerische Quadratur:



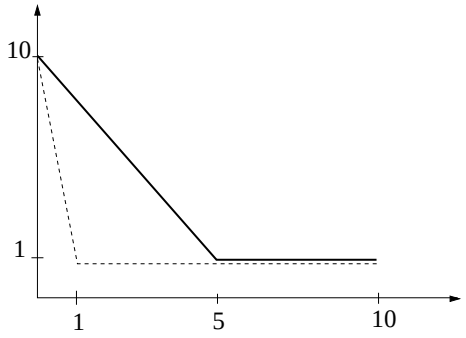
$$\int_0^{10} f(x)dx = 14.5$$

  Wir wollen drei Gitterpunkte investieren, dazwischen wird linear interpoliert. Den Näherungswert $\tilde{\int}$ für obiges Integral erhalten wir über den exakten Integralwert des Inter-

polanten:

→ äquidistanter Fall: $x_0 = 0, x_1 = 5, x_2 = 10$: $\tilde{\int} = 32.5$

→ adaptiver Fall: $x_0 = 0, x_1 = 1, x_2 = 10$: $\tilde{\int} = \int = 14.5$!



——— $\tilde{\int} = 32.5$ (äquidistant)

- - - - - - $\tilde{\int} = 14.5$ (adaptiv)

(IV) verteilt:

- Einsatz von Parallelrechnern
- Teamwork: Teilaufgaben auf Experten auslagern, z. B. bei Fluid-Struktur-Wechselwirkungen (Stausee-Damm)
  → nicht alles neu modellieren

(V) modular:

- Erkenntnis des Software-Engineering allgemein
- in der Numerik:
  - ⋆ PDEs:

    Geometriediskretisierung

    Gleichungsdiskretisierung

    iterativer Löser (cg, Gauß-Seidel, Mehrgitterverfahren)

    Grundbausteine aus der linearen Algebra ($A\,b$, $b^T\,u$)

    → alles möglichst entkoppeln!
  - ⋆ Fluid-Struktur-Wechselwirkungen:
    Fluid- und Strukturteil weitmöglichst entkoppeln

(VI) interdisziplinär:

- ergibt sich aus dem Aufgabenspektrum des wissenschaftlichen Rechnens

(VII) strukturiert:

- Struktur vereinfacht Problemstellung i. d. R.; deshalb empfiehlt es sich, vorhandene Struktur auszunutzen, also beispielweise
  - ⋆ gekoppelte PDEs nicht einfach in einen Topf werfen und System lösen, sondern möglichst weit entkoppeln;
  - ⋆ nicht unnötig komplizierte Elemente verwenden (Tetraedergitter etc.);

(VIII) ???

(IX) ???

(X) ???

Leitgedanke des rekursiven bzw. hierarchischen Vorgehens:

teile und herrsche — divide and conquer — divide et impera

Der Ausspruch wird heute König Ludwig XI. von Frankreich zugeschrieben, allerdings in der französischen Fassung „Diviser pour régner“. Er wird oft zur Charakterisierung der Politik der „Befriedung“ im antiken Rom gebraucht, lässt sich aber erst seit der Renaissance belegen. Ludwig XI. war französischer König von 1461 bis 1483. Er festigte den noch an den Folgen des 1453 beendeten Hundertjährigen Krieges leidenden französischen Nationalstaat (siehe Abb. 1.5). Sein Hauptwidersacher hierbei war Karl der Kühne, der letzte der großen Herzöge Burgunds (Philipp der Kühne 1363–1404, Johann der Unerschrockene 1404–1419, Philipp der Gute 1419–1467, Karl der Kühne 1467–1477). Daneben war Frankreich noch von einer ganzen Reihe weiterer potentieller Feinde umgeben: Navarra, Kastilien, Aragon, England, das habsburgische Kaiserreich sowie die Eidgenossen. Stets versuchte deshalb Ludwig, unter seinen Nachbarn Zwietracht zu säen und so eine gegen Frankreich gerichtete Allianz zu verhindern: divide et impera! Hermann Hesse sah allerdings Philipp von Mazedonien als Schöpfer des Gedankens. Eine Fassung findet sich auch bei Goethe: „Entzwei’ und gebiete!“ (womit auch dieses Skript seinen Beitrag zum Goethe-Jahr leistet).

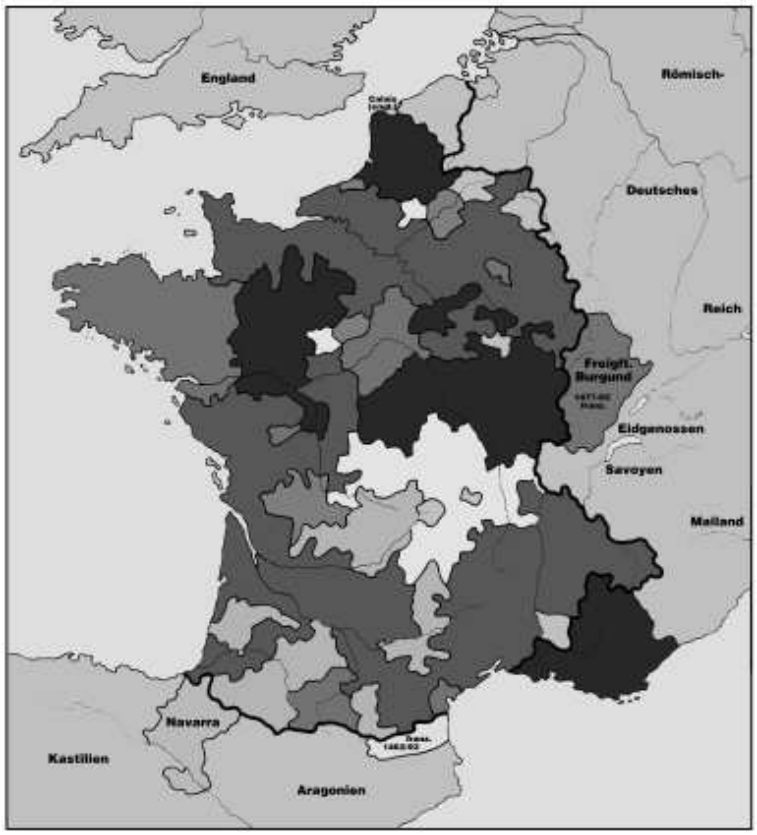


Abbildung 1.5: Frankreich nach dem Hundertjährigen Krieg

Anwendungsbeispiele von divide et impera aus Informatik und Mathematik:

- Sortieren mittels mergesort (siehe Abb. 1.6):
  teile Menge in 2 Hälften, sortiere beide Hälften, mische die sortierten Teilfelder

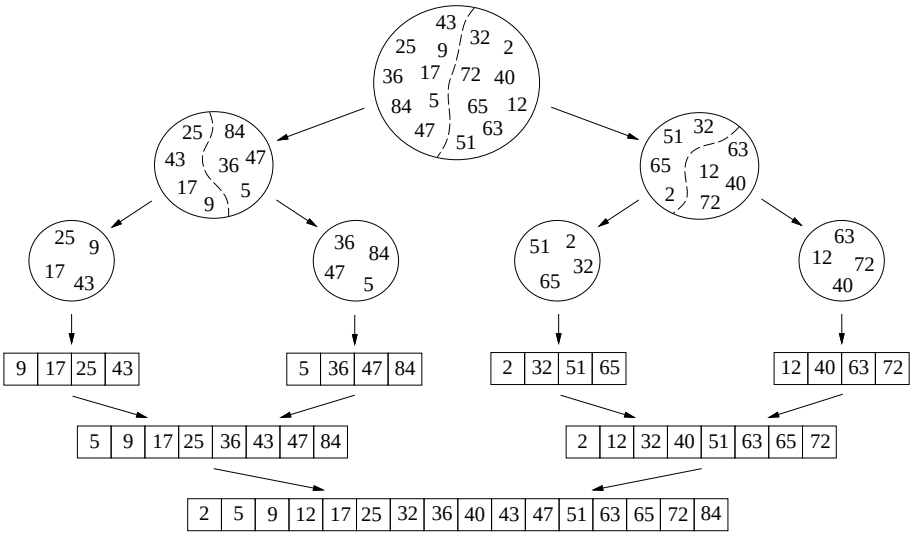


Abbildung 1.6: Mergesort

- Berechnung der konvexen Hülle (siehe Abb. 1.7):

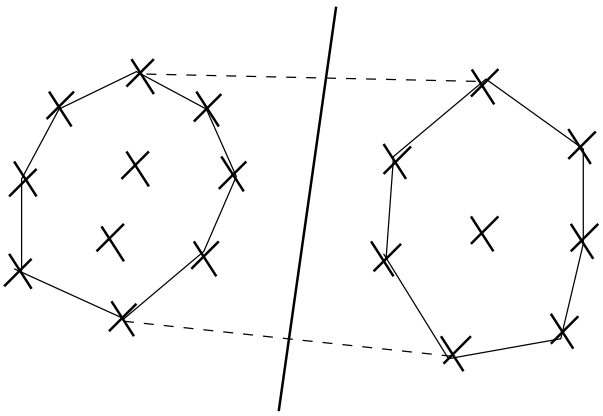


Abbildung 1.7: Konvexe Hülle

- Gebietszerlegung bei partiellen Differentialgleichungen
- numerische Quadratur, Spline-Interpolation:
  konstruiere Polynom mit gewünschten Eigenschaften auf jedem Teilintervall, klebe die (Teil-) Lösungen zusammen

Kapitel 2

Mathematical Modelling

2.1 Allgemeines

Zur formalen Beschreibung sowie zur systematischen Behandel- und damit Lösbarkeit werden Probleme in den unterschiedlichsten wissenschaftlichen Disziplinen *mathematisiert*, d. h. mit Hilfe des Methodenapparats der Mathematik präzise beschrieben. In manchen Fällen liegt das nahe und hat eine lange Tradition: in den (exakten) Naturwissenschaften (insbesondere der Physik) und in vielen Bereichen der Ingenieurwissenschaften. Die Grundgleichungen der Kontinuumsmechanik beispielsweise sind per se mathematischer Natur und heute in weiten Anwendungsfeldern allgemein akzeptiert. Andernorts liegt die mathematische Beschreibung weniger an der Oberfläche: in der Nationalökonomie, in der Spieltheorie oder bei der Klimamodellierung. Hier ist oftmals unklar oder umstritten, wie die zugrunde liegenden Sachverhalte formalisiert werden sollen bzw. welche Modellhypothesen plausibel sind. Das von-Neumannsche Min-Max-Prinzip in der Spieltheorie (wähle die Strategie so, dass das schlechtestmögliche Ergebnis bestmöglich wird) wird sicher einem echten Zocker nicht gerecht. Und der ewige Streit zwischen Anhängern und Gegnern von John Maynard Keynes' wirtschaftspolitischen Thesen (deficit spending etc.) zeigt auch, dass hier bislang kein Konsens über die richtige Strategie staatlicher Wirtschaftspolitik erzielt werden konnte.

Was ist ein mathematisches Modell? Ausgangspunkt ist ein gegebenes Problem einer Anwendungswissenschaft, das auf eine zunächst informale Art beschrieben ist (also z. B. in „Prosa"). Durch Formalisierung, d. h. durch Übergang zu einer Beschreibungsform mit dem Instrumentarium der Anwendungswissenschaft wird daraus ein Modell. Mit ihm versucht der Anwendungswissenschaftler, sein reales Problem möglichst umfassend, präzise und doch einfach formal zu beschreiben. In einem evtl. weiteren Schritt wird das Modell in die Sprache der Mathematik übertragen, wobei Inkonsistenzen sowie Redundanzen entfernt und die numerische Behandlung vorbereitet werden sollen. Entscheidende Fragen bei der Modellgenerierung sind:

- Welche Größen spielen eine Rolle? Welche Größen sind wie wichtig?
  Zwei Beispiele: Will man die bestimmenden Gleichungen für die optimale Flugbahn des Space-Shuttle zusammentragen, so spielt die Gravitation des Pluto hierbei sicher nur eine vernachlässigbare Rolle. In ein Modell für die Entwicklung des Dow Jones Indexes in der nächsten Woche muss die Meinung des Autors sicher nicht einfließen. Die Stellungnahme des US-Notenbankpräsidenten nicht zu berücksichtigen, wäre dagegen ein schweres Versäumnis.

- In welchem Beziehungsgeflecht stehen die Größen miteinander?
  Hierbei sind die Abhängigkeiten sowohl qualitativ (Was beeinflusst was?) als auch quantitativ (Wie sieht die Beeinflussung aus?) zu erfassen. Solche Wechselwirkungen lassen sich beschreiben

  - als *algebraische Gleichungen* und *Ungleichungen*, etwa in Form eines Systems linearer Gleichungen und Ungleichungen $A \cdot x \leq b$ bei der linearen Optimierung,
  - als *System gewöhnlicher* oder *partieller Differentialgleichungen*, z. B. $\ddot{y} + y = 0$ für die Oszillation eines idealen Pendels oder $\Delta u = f$ auf $[0,1]^2$ für die Verformung einer eingespannten Membran unter einer Last,
  - als Anfangs- oder Randbedingungen für Differentialgleichungen, also $y(0) = 1.5$ oder $u(x,y) = 0$ für $x, y \in \{0,1\}$,
  - als Graphen oder Zustandsübergangsdiagramme bei Reihenfolgeproblemen oder Markov-Ketten,
  - ...
- Welche Gestalt hat die resultierende Aufgabenstellung?
  - Löse eine Gleichung bzw. ein System von Gleichungen.
  - Löse eine Existenzaufgabe: Gibt es einen Hamiltonschen Weg in einem gegebenen Graphen?
  - Löse eine Extremalaufgabe: Finde den kürzesten Weg im Graphen.
- Gibt es Aussagen zur Lösbarkeit bzgl.
  - der Existenz bzw.
  - der Eindeutigkeit von Lösungen?
- Hängt die Lösung stetig von den Eingabedaten (Anfangswerte, Randwerte, rechte Seiten etc.) ab?
  Probleme, die die drei Bedingungen Existenz und Eindeutigkeit der Lösung sowie deren stetige Abhängigkeit von den Daten erfüllen, werden nach Hadamard (1923) auch *sachgemäß gestellte Probleme (well-posed problems)* genannt. In den dreißiger Jahren wiesen Tikhonov und John darauf hin, dass die Mehrzahl der Probleme leider als unsachgemäß gestellt (ill-posed) einzustufen ist. Dies gilt vor allem für die in technischen Problemstellungen weit verbreitete Klasse der sog. *inversen Probleme*, bei denen nicht die Antwort auf bestimmte Parametereinstellungen oder Randbedingungen gesucht ist, sondern umgekehrt gefragt ist, welche Bedingungen vorgegeben werden müssen, um ein bestimmtes gewünschtes Ergebnis zu erhalten. In diesem Fall wird oftmals nicht das Ausgangsproblem, sondern ein *regularisiertes* verwandtes Problem gelöst, welches sachgemäß gestellt ist.
- Ist das Modell – falls erforderlich – geeignet für eine numerische Behandlung?
- Haben wir alles richtig gemacht?
  Erforderlich ist eine eingehende Prüfung der Gültigkeit des Modells (Modellfälle rechnen, Plausibilitäts-Check, Vergleich mit experimentellen Daten etc.).

Hinsichtlich der Lösungstechniken, mit denen mathematische Modelle „geknackt“ werden können, lassen sich mehrere unterschiedliche Vorgehensweisen unterscheiden:

- Der *analytische* Ansatz versucht, nicht nur Existenz und Eindeutigkeit formal zu beweisen, sondern auch bei der Konstruktion einer bzw. der Lösung mit formalen Mitteln auszukommen. Eine solche Vorgehensweise ist i. A. nur bei einfachen Modellproblemen möglich.
- Bei vielen diskreten Problemstellungen kann man die Lösung zumindest theoretisch durch (geschicktes) *Ausprobieren* erhalten, da die Zahl der potentiellen Kandidaten endlich ist. Allerdings ist diese Zahl oft so groß, dass dieser Weg praktisch nicht gehbar ist (etwa beim Problem des Handlungsreisenden oder bei anderen NP-vollständigen Aufgaben). Hier kommen dann *Heuristiken* zum Einsatz.

- Ferner gibt es *direkte numerische* Verfahren, die das gegebene Problem zwar algorithmisch und u. U. in mehreren Iterationsschritten lösen, aber sicher bei der exakten Lösung landen. Als Beispiele sei hier der Simplex-Algorithmus zur Lösung von Aufgaben der linearen Optimierung genannt.
- Die *(echt) iterativen numerischen* Verfahren schließlich nähern sich der Lösung in mehreren Iterationsschritten sukzessive und beliebig an, ohne sie jedoch exakt zu treffen. Dies trifft etwa für die numerischen Verfahren zur Lösung gewöhnlicher bzw. partieller Differentialgleichungen zu.

Das wissenschaftliche Rechnen befasst sich schwerpunktmäßig mit Lösungsansätzen aus der letztgenannten Klasse. Da hierbei Modelle auf der Grundlage von Systemen partieller Differentialgleichungen die heftigsten Probleme bereiten, wollen wir uns bei der Diskussion numerischer Techniken in Kapitel 3 hierauf konzentrieren. Zunächst soll jedoch ein breiterer Überblick über mathematische Modelle gegeben werden.

2.2 Beispiele mathematischer Modelle

Eine Klassifikation mathematischer Modellansätze ist sicher ein gewagtes Unterfangen. Die Literatur, die sich – losgelöst von konkreten Problemstellungen oder Disziplinen – systematisch mit den Prinzipien mathematischer Modellierung befasst, ist dazuhin nicht sehr reichhaltig. Trotzdem lassen sich gewisse Unterscheidungen treffen:

- *Diskrete* Modelle beschreiben Phänomene mit einem ganzzahligen oder kombinatorischen Instrumentarium. Hier hat man es also nicht mit reellwertigen Funktionen zu tun. Anwendung finden diskrete Modelle natürlich primär bei diskreten Fragestellungen. Als Beispiele seien genannt die Optimierung des Einsatzplans für Flugzeuge einer Fluglinie (Scheduling) sowie das Aufspüren einer optimalen Route für einen Vertreter oder für ein Datenpaket im Internet (Routing). Aber auch inhärent kontinuierliche Probleme der Physik können diskret modelliert werden. So beschreiben Lattice-Gas-Automaten die Ausbreitung eines Fluids nicht über die (kontinuierlichen) Variablen Geschwindigkeit und Druck der Kontinuumsmechanik, sondern modellieren die Strömung als Folge diskreter Einzelbewegungen von Fluiddichten.
- *Kontinuierliche* Modelle beschreiben Phänomene mit Hilfe reellwertiger Größen. Auch hier gilt, dass der hauptsächliche Anwendungsbereich natürlich bei kontinuierlichen Fragestellungen liegt. Gleichwohl ist auch bei diskreten Problemen nicht immer alles diskret. Man denke etwa an den Verkehrsfluss an Ampeln, wo neben diskreten Größen (Anzahl der wartenden Fahrzeuge zur Zeit t) auch kontinuierliche betrachtet werden (Anzahl der durchschnittlich wartenden Fahrzeuge vor einer Ampel): Damit könnte man den Verkehrsfluss durch eine Stadt auch als Strömung von Wasser durch ein Kanalsystem mit abwechselnd offenen oder geschlossenen Schleusen bzw. Sperren modellieren.
- Neben rein *deterministischen* Modellen kann man auch König Zufall in ein Modell integrieren und gelangt so zu *stochastischen* Modellen. Bei „echten Zufallsexperimenten“ wie dem Würfeln ist alles andere eh sinnlos, aber auch bei im Grunde deterministischen Vorgängen wie der Frage, welche Radios aus einer Großserienproduktion defekt sind, oder bei der Modellierung der Auftragsabwicklung in großen Rechnersystemen erweist sich eine randomisierte Betrachtungsweise oft als vorteilhaft. Auch bei stochastischen Modellen kann man zwischen diskreter und kontinuierlicher Betrachtungsweise unterscheiden (vgl. diskrete und stetige Zufallsvariable).

Im Folgenden wollen wir uns exemplarisch einige Beispiele für mathematische Modelle aus den drei oben genannten Klassen ansehen und dabei auch das jeweils eingesetzte und von Fall zu Fall unterschiedliche mathematische Instrumentarium im Auge behalten.

2.2.1 Diskrete Modelle

Zunächst betrachten wir zwei diskrete Modelle: Ein Modell für das Problem des *Scheduling* bzw. allg. für Reihenfolgeprobleme sowie ein Modell aus der Entscheidungstheorie zur Bewertung von Wahlverfahren.

Scheduling

Beim Scheduling ist man mit folgender Aufgabenstellung konfrontiert: Gegeben seien n Aufträge $A_1, \ldots, A_n$, die auf m Maschinen $M_1, \ldots, M_m$ bearbeitet werden sollen. Dabei kann kein Auftrag gleichzeitig auf mehreren Maschinen sein, und keine Maschine kann gleichzeitig mehr als einen Auftrag bearbeiten (serielle Auftragsabarbeitung). Die verschiedenen Maschinen können aber gleichzeitig arbeiten. Um möglichst viele Probleme mit unserem Modell zu erfassen, führen wir eine Reihe weiterer Beschreibungsparameter ein:

- Parameter, die die Aufträge charakterisieren:
  - m_j: Anzahl der (Ein-Maschinen-) Teilschritte bei Auftrag A_j,
  - p_{ij}: Prozesszeit von Auftrag A_j auf Maschine M_i,
  - r_j: Zeitpunkt der Verfügbarkeit (wann kann mit Auftrag A_j begonnen werden?),
  - d_j: Zeitpunkt der Fälligkeit (wann muss Auftrag A_j fertig sein?),
  - w_j: Gewichtungsparameter zum Ausdruck der Bedeutung von Auftrag A_j,
  - $f_j(t)$: Kostenfunktion, die die Kosten von A_j bei Fertigstellung zur Zeit t angibt (nicht abnehmend in t);
- Charakterisierung der Maschinen und der Auftragsabwicklung:
  - eine Maschine / qualitativ und quantitativ identische Maschinen mit rein auftragsabhängiger Arbeitszeit ($p_{ij} = p_j$) / qualitativ identische Maschinen mit auftrags- und maschinenabhängiger Arbeitszeit, wobei Maschinen und Aufträge unabhängig sind ($p_{ij} = q_i \cdot p_j$) / beliebige Maschinen (allg. p_{ij}),
  - die Teilaufträge von A_j auf verschiedenen Maschinen sind beliebig vertauschbar (*open shop*) / sind an die strikte Einhaltung einer Reihenfolge gebunden (*flow shop*) / Reihenfolgeabhängigkeit und jede Maschine wird höchstens einmal besucht (*job shop*),
  - Teilaufträge sind unterbrechbar / müssen am Stück bearbeitet werden,
  - evtl. Präzedenzrelationen zwischen den einzelnen Aufträgen,
  - ...
- Optimalitätskriterien:
  - c_j: Fertigstellungszeitpunkt von A_j,
  - $l_j = c_j - d_j$: Verzug von A_j,
  - $t_j = \max(0, l_j)$: Verspätung von A_j,
  - $s_i = \max_j c_j \; - \; \sum_{j=1}^{n} p_{ij}$: Stillstandszeit der Maschine M_i;

  Zu bestimmen ist dann eine Abarbeitungsreihenfolge, die eine der obigen Größen minimiert oder das Maximum einer der obigen Größen über alle j bzw. i minimiert oder eine gewichtete Summe einer der obigen Größen über alle j bzw. i minimiert.

Nun betrachten wir den Spezialfall von m unterschiedlichen Maschinen und n Aufträgen $A_j = (A_{1,j}, \ldots, A_{m_j,j})$ als job shop (d. h., die Reihenfolge ist fest vorgegeben, jeder Auftrag besucht jede Maschine höchstens einmal). Zudem gebe es eine Präzedenzordnung zwischen den einzelnen Aufträgen, und die zu minimierende Zielfunktion sei der maximale Fertigstellungszeitpunkt aller Aufträge $c_{\max} = \max_j c_j$. Zur Modellierung bedienen wir uns des Instrumentariums der Graphentheorie:

- *Knoten* im Graph sind alle Teilaufträge $A_{i,j},\ 1 \leq i \leq m_j,\ 1 \leq j \leq n$, sowie ein Start- und ein Zielknoten.
- *Kanten* im Graph gibt es viererlei:
  - *Konjunktivkanten* von $A_{i,j}$ nach $A_{i+1,j}$ (Präzendenz der Teilaufträge eines Auftrags),
  - *Konjunktivkanten* vom Startknoten zu allen $A_{1,j}$ sowie von allen $A_{m_j,j}$ zum Zielknoten (Kennzeichnung von Anfang und Ende jedes Auftrags),
  - *Konjunktivkanten* von $A_{m_j,j}$ nach $A_{1,k}$, falls A_j in der Auftragspräzendenz vor A_k kommt (Präzendenz der Aufträge),
  - *Disjunktivkanten* von $A_{i,j}$ nach $A_{l,k}$ und zurück, falls beide Teilaufträge dieselbe Maschine benötigen (Identifikation der Maschinen; nur eine (aber nicht a priori festliegende) Kantenrichtung bzw. Reihenfolge ist relevant, daher der Begriff *disjunktiv*).
- Zudem werden alle Knoten $A_{i,j}$ mit $p_{l,j}$ bewertet, wenn $A_{i,j}$ auf M_l bearbeitet wird. Start- und Zielknoten werden beide mit Null bewertet.
- Verfügbarkeitszeiten können durch zusätzliche Knoten zwischen Startknoten und $A_{1,j}$ berücksichtigt werden, die mit r_j bewertet sind.

Eine mögliche Lösung wird dann beschrieben als Vektor der Startzeiten $t(A_{i,j})$ aller Teilaufträge $A_{i,j}$, wobei der Startknoten die Anfangszeit $t = 0$ und der Zielknoten die Anfangszeit $t = c_{\max}$ erhalten. Eine solche Lösung ist *zulässig*, wenn alle Präzedenz- und sonstigen Bedingungen erfüllt sind. Man kann zeigen, dass es zu jeder zulässigen Lösung eine eindeutig bestimmte Auswahl aktiver Disjunktivkanten (eine Richtung aus jedem Paar) gibt, wobei der reduzierte Graph azyklisch ist. Eine solche Kantenauswahl nennt man auch *Disjunktivkantenbelegung*.

Damit steht das Modell. Jetzt kann man sich daran machen, Algorithmen für die Lösung des hergeleiteten graphentheoretischen Problems zu suchen. Dazu geht man prinzipiell so vor, dass man von einer Disjunktivkantenbelegung startet, sich eine zulässige Lösung konstruiert (falls existent), die zu dieser Disjunktivkantenbelegung passt, dann eine *kritische Bahn* sucht, d. h. einen Weg maximaler Länge (in aufaddierten Knotenbewertungen) vom Start- zum Zielknoten und schließlich die Lösung (den Vektor der Startzeiten) soweit möglich verringert. Damit hat man die minimale Zeit für diese bestimmte Reihenfolge. Was bleibt, ist die Bestimmung des Optimums über die verschiedenen Disjunktivkantenbelegungen, wofür beispielsweise Heuristiken eingesetzt werden können.

In practice, the selection of a disjunctive edge's actual direction is usually determined by a so-called *policy*. One possibility is to add *priority lists* that say which job has to be preferred if several (without precedence) are ready for the same machine. However, such a policy is neither continuous nor monotone: It might be possible that decreasing all jobs' running times leads to an increased overall running time (see Fig. 2.1). Therefore, more sophisticated policies have to be developed and applied.

Well, that's all about scheduling? Certainly not! Often, it is not reasonable to model the completion times as deterministic quantities. Especially during big projects, many unforeseen events can influence the system's actual behaviour. The result is something we all know: Hardly any huge project has ever been finished in time or with the budget originally estimated (think of the move of Germany's federal parliament and government from Bonn to Berlin, for example). Thus, instead of dealing with deterministic c_j and being astonished about cost or time at the end, we

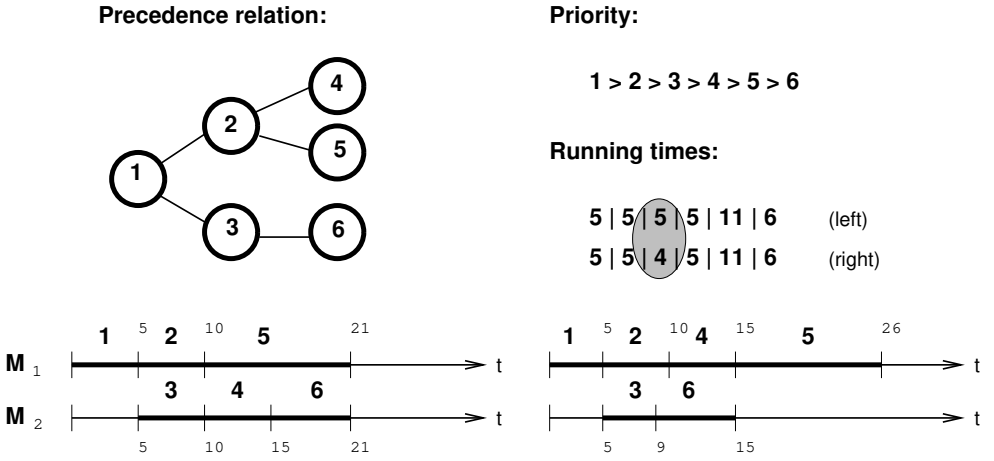


Abbildung 2.1: Priority-based policy for a given precedence relation and a simple priority list. If one of the two equal machines M_1 and M_2 finishes a job, it selects a new available job (due to the precedence graph) with highest possible priority. Although the running time of job A_3 is reduced from 5 to 4, we end up with a higher overall running time (26 compared to 21 before) – a clear drawback of the priority strategy. Obviously, it would be better to accept a short idle period of machine M_2 after the completion of job A_3.

should use random variables C_j. Then, the above phenomenon of *underestimation* has a theoretical background (Fulkerson 1962):

$$\max_j E(C_j) \quad \leq \quad E\left(\max_j C_j\right) . \tag{2.1}$$

That is, maximizing over all expected completion times $E(C_j)$ leads to an earlier or equal overall completion time than maximizing over random variables and forming the moment afterwards.

A very simple example can help to illustrate the above relation. Think of n independent jobs A_j with running times X_j that are all equally distributed on $[0,1]$, i. e. $p(X_j \leq t) = t$. Hence, $E(X_j) = 1/2$. Due to the jobs' independence,

$$p(\max_j X_j \leq t) \;=\; \prod_j p(X_j \leq t) \;=\; t^n \tag{2.2}$$

and, hence,

$$E(\max_j X_j) \;=\; \int_0^1 t \cdot nt^{n-1} dt \;=\; \frac{n}{n+1} \;\to\; 1 \quad \text{for} \quad n \to \infty . \tag{2.3}$$

Consequently, the expected maximum running time tends to the worst case 1, if the number of jobs increases, whereas

$$\max_j E(X_j) \;=\; \max_j \frac{1}{2} \;=\; \frac{1}{2} . \tag{2.4}$$

This simple problem already shows that a stochastic model can lead to completely different results than a discrete one does, and that a stoachastic point of view is necessary to explain certain phenomena like the above-mentioned underestimation of completion times. We will deal with stochastic models later. For the moment, we keep to discrete models.

Finally, note that classical graphs are not the only mathematical tool to model deterministic scheduling problems. If more than one job can be on the machines simultaneously, a *hypergraph*

may be better suited. A hypergraph consists of a set V of nodes and a set HE of *hyperedges*, each of which are a non-empty subset of V. If the cardinality of all hyperedges is 2, the hypergraph is just a classical graph. Now we can interpret nodes as jobs and hyperedges as machines (collecting all jobs that need this machine), and we might look for a colouring of the hypergraph with a small number of colours (where each colour represents a certain time interval, i. e., if a machine can do three jobs in parallel, then only three nodes of the same colour are allowed in its hyperedge). Hence, the message is that for each class of problems tailored modelling tools have been developed – for reasons of theoretical investigations or for obtaining algorithms and solutions for practical problems.

Entscheidungs- und Wahlmodelle

Für unser zweites Beispiel benötigen wir ein paar Grundbegriffe aus der Theorie der Relationen. Eine *Relation* R auf einer Trägermenge X ist eine Teilmenge von $X^2 = X \times X$. Statt $(x, y) \in R$ schreibt man auch xRy. R heißt

reflexiv	$:\Leftrightarrow$	$xRx \;\; \forall x \in R,$
transitiv	$:\Leftrightarrow$	$xRy \;\wedge\; yRz \;\Rightarrow\; xRz,$
symmetrisch	$:\Leftrightarrow$	$xRy \;\Rightarrow\; yRx,$
antisymmetrisch	$:\Leftrightarrow$	$xRy \;\wedge\; yRx \;\Rightarrow\; x = y,$
asymmetrisch	$:\Leftrightarrow$	$xRy \;\Rightarrow\; \neg yRx,$
konnex	$:\Leftrightarrow$	$\neg xRy \;\Rightarrow\; yRx,$
Quasiordnungsrelation	$:\Leftrightarrow$	reflexiv und transitiv,
Äquivalenzrelation	$:\Leftrightarrow$	reflexiv, transitiv und symmetrisch,
Halbordnungsrelation	$:\Leftrightarrow$	reflexiv, transitiv und antisymmetrisch,
Totalordnungsrelation	$:\Leftrightarrow$	konnexe Halbordnungsrelation.

Die meisten Eigenschaften sind sicher bekannt, einzig die Konnexität ist kein so häufig gebrauchter Begriff. Diese Eigenschaft einer Relation garantiert einfach, dass zwei beliebige Elemente der Trägermenge vergleichbar sind, also in der einen oder anderen Richtung miteinander in Relation stehen.

Obige Eigenschaften von Relationen lassen sich leicht am Beispiel der reellen Zahlen und der dort auftretenden Relationen $=, \leq, \geq, <$ sowie $>$ veranschaulichen. Da alle reellen Zahlen paarweise vergleichbar sind, ist hier die Konnexität immer gegeben (vgl. dagegen die komplexen Zahlen).

Jetzt wollen wir uns mit Modellen für die Beurteilung von Wahlsystemen befassen. Dazu betrachten wir eine endliche *Kandidatenmenge* X und eine endliche Menge $I := \{1, \ldots, n\}$ von *Wählern*. $B := X^2$ sei die Menge aller Relationen auf X, und $A \subseteq B$ sei die Menge aller konnexen Quasiordnungsrelationen (also reflexiv, transitiv und konnex). Im Folgenden wollen wir uns auf diese vernünftigen Präferenzrelationen beschränken. Dann heißt jede Abbildung f,

$$f : A^n \;\; \to \;\; B, \qquad (R_1, \ldots, R_n) \;\; \mapsto \;\; R,$$

eine *kollektive Auswahlfunktion*. Nichts anderes geschieht bei einer Wahl: Aus den individuellen Präferenzen R_i der einzelnen Wähler wird eine einzige Präferenz R des Kollektivs ermittelt. Da es (zumindest in Demokratien) durchaus nicht unvernünftig ist, die Meinung der Mehrheit als für den Wahlausgang ausschlaggebend heranzuziehen, definiert man weiter für beliebige individuelle Präferenzen $R_1, \ldots, R_n \in A$:

$$\begin{aligned} N(x, y) &:= \text{Anzahl der } i \in I \text{ mit } xR_iy\,, \\ xMy &:\Leftrightarrow N(x, y) \geq N(y, x)\,, \\ f_M(R_1, \ldots, R_n) &:= M\,. \end{aligned}$$

M heißt die $(R_1, \ldots, R_n)$ zugeordnete *Mehrheitsrelation*, und die kollektive Auswahlfunktion f_M wird *Mehrheitsentscheidungsfunktion* genannt. Es lässt sich zeigen, dass M stets reflexiv und

konnex ist. Allerdings muss M nicht transitiv sein, was man am folgenden *Abstimmungsparadoxon* sieht:

Gegeben seien drei Kandidaten, d. h. $X := \{a, b, c\}$, und drei Wähler, also $I := \{1, 2, 3\}$. Die Wähler haben folgende Präferenzen:

$$\begin{aligned} R_1 &:= \{(a,b),(b,c),(a,c),(a,a),(b,b),(c,c)\}\ , \\ R_2 &:= \{(c,a),(a,b),(c,b),(a,a),(b,b),(c,c)\}\ , \\ R_3 &:= \{(b,c),(c,a),(b,a),(a,a),(b,b),(c,c)\}\ . \end{aligned}$$

Alle R_i sind nicht nur (wie gefordert) konnexe Quasiordnungsrelationen, sondern sogar antisymmetrisch und damit Totalordnungsrelationen. Es ergeben sich folgende Mehrheiten:

$$N(a,b) = 2,\quad N(b,a) = 1,\quad N(a,c) = 1,\quad N(c,a) = 2,\quad N(b,c) = 2,\quad N(c,b) = 1\,.$$

Somit gilt $N(a,b) \geq N(b,a)$, $N(c,a) \geq N(a,c)$ und $N(b,c) \geq N(c,b)$. Daraus ergibt sich M zu

$$M := \{(a,b),(b,c),(c,a),(a,a),(b,b),(c,c)\}\ ,$$

was zu einer zyklischen kollektiven Präferenz führt: M ist *nicht* transitiv. Obwohl alle drei Wähler klare Präferenzen zwischen allen Kandidaten haben, lässt die resultierende kollektive Präferenz keine (vernünftige) Wahl bzw. Rangfolge zu. Ist die Demokratie also unlogisch, führt sie ins Chaos? Sehen wir uns dazu die Alternativen an.

In einem demokratischen System ist die Mehrheitsentscheidungsfunktion f_M sicher eine (wenn nicht die) vernünftige Vorgehensweise. Unser Modell erfasst aber auch andere politische Systeme. Ein externer Diktator, der über die Köpfe des Parlaments hinweg entscheidet, kann mittels

$$f_D(R_1, \ldots, R_n) := S \in A \qquad \forall\ R_i \in A$$

modelliert werden, ein „starker Mann“ $j \in I$, der als Parlamentsmitglied das Parlament beherrscht, mittels

$$f_S(R_1, \ldots, R_n) := R_j \qquad \forall\ R_i \in A\,.$$

In beiden Fällen ist das Ergebnis zwar ein vernünftiges, hier gibt es dafür wieder andere (naheliegende) Vorbehalte.

Modellieren wir also weiter. Da wir im demokratischen Rahmen bleiben wollen, müssen wir nach Alternativen zur (offensichtlich nicht optimalen) Mehrheitsentscheidungsfunktion suchen. Zunächst muss der Begriff der demokratischen Wahl präzisiert werden. Zu diesem Zweck stellen wir für den Fall $n \geq 2$ ein System von *demokratischen Grundregeln* auf, die jeder aufrechte Demokrat sofort als sinnvoll und angemessen akzeptieren wird:

R 1: Es muss eine kollektive Auswahlfunktion f geben, die auf ganz A^n definiert ist.

R 2: f muss stets eine Quasiordnungsrelation liefern, die also reflexiv und transitiv ist.

R 3: Für alle Paare $(x, y) \in X \times X$ gilt stets: $xP_iy\ \ \forall i \quad \Rightarrow \quad xPy$
(dabei $xP_iy \quad :\Leftrightarrow \quad xR_iy \quad \wedge \neg yR_ix$).

R 4: Für jedes Paar individueller Präferenzen $(R_1, \ldots, R_n)$, $(Q_1, \ldots, Q_n)$ gilt:
$(xR_iy \quad \Leftrightarrow \quad xQ_iy\ \ \forall i) \quad \Rightarrow \quad (xRy \quad \Leftrightarrow \quad xQy)$.

R 5: Es gibt keinen Wähler i mit $xP_iy \quad \Leftrightarrow \quad xPy \quad \forall(x, y) \quad \forall(R_1, \ldots, R_n)$.

Regel 1 garantiert die individuelle Freiheit der Wahl: Jede Kombination individueller Präferenzen ist zugelassen, sofern diese konnexe Quasiordnungsrelationen sind. Regel 2 schließt zyklische Kollektivpräferenzen wie beim obigen Beispiel aus. Regel 3 beschreibt die Entscheidungsfreiheit des Kollektivs: Sind sich alle Wähler einig (mit strengen Präferenzen P_i statt R_i, d. h. anschaulich $<$ statt $\leq$), dann setzt sich diese Meinung auch durch. Oder umgekehrt: Jedes kollektive Ergebnis

ist möglich und kann (durch Einstimmigkeit zumindest) herbeigeführt werden. Regel 4 sichert die Unabhängigkeit von irrelevanten Alternativen: Zwei Wahlgänge mit demselben Abstimmungsverhalten bzgl. zweier Kandidaten x und y führen auch zum selben kollektiven Wahlergebnis in Bezug auf x und y. Durch Regel 5 wird schließlich ein Diktator verhindert: Kein Wähler bestimmt bei beliebigen Individualpräferenzen die Wahl allein durch seine Entscheidung.

Man veranschauliche sich die Bedeutung der Regel 4 an folgendem Beispiel:

Fünf Wähler a, b, c, d, e stimmen ab über drei Kandidaten x, y, z. Jeder Wähler muss einem Kandidaten drei Stimmen, einem zwei und einem eine Stimme geben. Für die Kollektivpräferenz ist jeweils die Summe der Stimmen maßgeblich. In zwei unterschiedlichen Wahlgängen ergebe sich folgendes Bild:

Stimmen	1. Wahlgang:	a	b	c	d	e	2. Wahlgang:	a	b	c	d	e
3		x	x	y	y	y		x	x	y	y	y
2		z	z	x	x	x		y	y	x	x	x
1		y	y	z	z	z		z	z	z	z	z

Im ersten Wahlgang ist das Ergebnis 12 Stimmen für x, 11 für y und 7 für z, im zweiten 12 Stimmen für x, 13 für y und 5 für z. Obwohl sich die individuellen Präferenzen zwischen x und y bei keinem der fünf Wähler geändert haben (a und b ziehen x vor, c, d und e dagegen bevorzugen y), ändert sich die Kollektivpräferenz: Die „irrelevante Alternative“ z ist ausschlaggebend!

Man kann beweisen, dass die oben definierte Mehrheitsentscheidungsfunktion f_M die Regeln 1 und $3-5$ erfüllt. Regel 2 gilt für f_M allerdings nur im Falle $|X| = 2$, d. h. bei nur zwei Kandidaten: Das obige Abstimmungsparadoxon hatte ja schon die Schwäche einer möglicherweise nicht gegebenen Transitivität der Kollektivpräferenz aufgezeigt. Nach Arrow ist dieser Sachverhalt eine logische Rechtfertigung für das anglo-amerikanische Zweiparteiensystem (das aber spätestens seit Ross Perot in den USA und den erstarkten Liberalen in Großbritannien eigentlich keines mehr ist).

Jetzt könnte man versuchen, anstelle von f_M nach einer anderen kollektiven Auswahlfunktion zu suchen. Aber das hilft alles nichts, denn Arrow konnte 1951 in seinem berühmten Unmöglichkeitssatz sogar zeigen, dass es bei mehr als zwei Kandidaten und mehr als einem Wähler keine kollektive Auswahlfunktion geben kann, die alle fünf obigen Regeln erfüllt. Im Klartext: Die bunte Parteienlandschaft in Deutschland oder Italien ermöglicht (zumindest in unserem Modell) kein Wahlsystem, welches allen fünf demokratischen Grundregeln genügt. Damit sind wir an einem für die Modellierung ganz typischen Punkt angelangt, nämlich bei der Frage, ob in unserem Modell noch irgendwo der Wurm steckt, oder ob uns die Modellierung eine echte Schwäche der Realität, also in diesem Fall unseres politischen Systems, offenbart hat. Darüber sollen sich aber andere den Kopf zerbrechen. Wir gehen weiter zu den kontinuierlichen Modellen.

2.2.2 Kontinuierliche Modelle

Wir betrachten zwei Beispiele kontinuierlicher Modelle: Populationsmodelle, die durch Systeme gewöhnlicher Differentialgleichungen beschrieben werden, und ein Modell zur Wärmeleitung, das die Modellierung mit Hilfe partieller Differentialgleichungen exemplarisch zeigen soll.

Populationsmodelle

Populationsmodelle beschreiben das Wachstum von Populationen, isoliert (d. h. ohne wesentliche äußere Einflüsse) oder in (friedlicher oder feindlicher) Koexistenz mehrerer Gattungen. Einen typischen und sehr alten Vertreter solcher Modelle (nach Maltus, 1798) wollen wir uns näher anschauen. Wir gehen von einer konstanten positiven *Geburtenrate* γ (Zuwachs pro Zeiteinheit und pro Kopf) und einer ebenfalls konstanten positiven *Sterberate* δ (Abnahme pro Zeiteinheit und

pro Kopf) aus, die in einer konstanten *Wachstumsrate* $\lambda = \gamma - \delta$ (positiv oder negativ) zusammengefasst werden. Für die Größe $p(t)$ einer Population zur Zeit t wird folgender Zusammenhang angenommen:

$$p(t + \Delta t) \;=\; p(t) \;+\; \lambda \cdot p(t) \cdot \Delta t\,,$$

d. h., das Wachstum der Population ist proportional zur Populationsgröße und zur Zeit. Division durch Δt und Grenzübergang $\Delta t \to 0$ führen auf die gewöhnliche Differentialgleichung

$$\dot{p}(t) \;=\; \lambda \cdot p(t)$$

mit der – bei Vorgabe des Anfangswerts $p(0) = p_0$, d. h. der Populationsgröße zur Zeit $t = 0$ – eindeutigen Lösung

$$p(t) \;=\; p_0 \cdot e^{\lambda \cdot t}\,.$$

Damit haben wir schon den Sprung von einer *diskreten* Realität zu einem *kontinuierlichen* Modell vollzogen: Die Stärke einer Population ist eine ganzzahlige Größe, obige Funktion $p(t)$ ist hingegen reellwertig. Doch auch eine ganzzahlige Betrachtungsweise führt auf die Exponentialfunktion. Unserer anfänglichen Modellhypothese einer konstanten Wachstumsrate λ entspricht die Annahme, dass sich die Größe der Population alle τ Jahre verdoppelt mit konstantem τ. Startet man mit der Anfangsstärke p_0 zur Zeit $t = 0$, so ergibt sich nach $k \cdot \tau$ Jahren eine Stärke von

$$p(k \cdot \tau) \;=\; 2^k \cdot p_0 \;=\; p_0 \cdot e^{\frac{ln\,2}{\tau} \cdot k \cdot \tau}\,,$$

λ und τ sind also über den Zusammenhang $\lambda = \frac{ln2}{\tau}$ verknüpft.

Ein exponentielles Wachstum anzunehmen ist sicher nur über einen begrenzten Zeitraum sinnvoll. Es hat sich beispielsweise gezeigt, dass die Erdbevölkerung zwischen 1700 und 1960 ziemlich präzise einem solchen Gesetz gefolgt ist, mit einer Wachstumsrate $\lambda = 0.02$ bzw. mit einer Verdopplungszeit von 34 Jahren und 8 Monaten. Das kann aber nicht ewig so weitergehen: Die Erde hat nur eine begrenzte Kapazität, und der wachsende Konkurrenzdruck um Ressourcen (Luft, Wasser, Nahrung) wird das Wachstum zwangsläufig abbremsen. Dies versuchen Modelle des 19. Jahrhunderts zu berücksichtigen (nach Verhulst), bei denen die Populationsgröße einem Sättigungsgrenzwert zustrebt. Im einfachsten Fall geht man von linearen Geburts- und Sterberaten (Zu- bzw. Abnahme pro Zeit) aus:

$$\gamma(t) \;=\; \gamma_0 - \gamma_1 \cdot p(t)\,, \qquad \delta(t) \;=\; \delta_0 + \delta_1 \cdot p(t)\,,$$

wobei $\gamma_0 > \delta_0 > 0$ und $\gamma_1, \delta_1 > 0$. Man beachte, dass der Begriff der Rate jetzt die Entwicklung pro Zeit insgesamt beschreibt, also nicht mehr pro Kopf. Wie man sieht, wirkt sich eine große Population jetzt bremsend auf die Geburtenrate und beschleunigend auf die Sterberate aus, die Sättigung ist damit in das Modell integriert. Als Differentialgleichung legt man

$$\dot{p}(t) \;=\; \gamma(t) - \delta(t) \;=\; -\,m \cdot (p(t) - p_\infty)$$

zugrunde, wobei

$$m \;=\; \gamma_1 + \delta_1 \;>\; 0\,, \qquad p_\infty \;=\; \frac{\gamma_0 - \delta_0}{\gamma_1 + \delta_1} \;>\; 0\,.$$

Der Anfangswert $p(0) = p_0$ führt dann auf die Lösung

$$p(t) \;=\; p_\infty + (p_0 - p_\infty) \cdot e^{-m \cdot t}\,.$$

Offensichtlich gilt $p(0) = p_0$ und $\lim_{t \to \infty} p(t) = p_\infty$. Ein solches beschränktes Wachstum ist in vielen Fällen bereits realistischer als das exponentielle Wachstum, es bleiben aber immer noch Wünsche offen. So ändert, bedingt durch die Charakteristik der Exponentialfunktion, die Krümmung der Populationsfunktion $p(t)$ ihr Vorzeichen nicht. Untersuchungen der Bevölkerungszahl der USA zwischen 1790 und 1950 zeigen jedoch einen S-förmigen Verlauf. Einen solchen erhält man etwa, wenn man von der Differentialgleichung

$$\dot{p}(t) \;=\; a \cdot p(t) \;-\; b \cdot p^2(t)$$

mit positiven Konstanten $a \gg b > 0$ ausgeht. Hier soll modelliert werden, dass bei noch kleinem $p(t)$ der quadratische Term kaum ins Gewicht fällt und sich so nahezu exponentielles Wachstum einstellt. Nimmt $p(t)$ dagegen zu, so gewinnt er an Bedeutung und bremst das Wachstum. Dass hierbei gerade die zweite Potenz angesetzt wird, ist eine – zu verifizierende – Modellannahme. Rechtfertigen lässt sich das etwa so, dass ein Individuum von der Population in einem Ausmaß gestört wird, das proportional zu deren Größe $p(t)$ ist. Die Beeinträchtigung der gesamten Population ergibt sich somit zu $O(p^2(t))$. Mittels Separation der Variablen kann man obige Gleichung lösen und erhält

$$p(t) \;=\; \frac{a \cdot p_0}{b \cdot p_0 + (a - b \cdot p_0) \cdot e^{-a \cdot t}}\,, \qquad p(0) \;=\; p_0\,.$$

Der Rest ist Kurvendiskussion (vgl. Abb. 2.2):

- $\lim_{t\to\infty} p(t) \;=\; \frac{a}{b}$,
- $p_0 > \frac{a}{b} \;\Rightarrow\; p(t) > \frac{a}{b} \quad \forall\, t \geq 0$, kein Wendepunkt,
- $p_0 < \frac{a}{b} \;\Rightarrow\; p(t) < \frac{a}{b} \quad \forall\, t \geq 0$, ein Wendepunkt,
- $p_0 = \frac{a}{b} \;\Rightarrow\; p(t) = p_0 \quad \forall\, t \geq 0$.

Im Falle der oben erwähnten Bevölkerungsentwicklung in den USA zwischen 1790 und 1950 erhält man $a = 0.03134$ und $b = 1.5587 \cdot 10^{-10}$. Doch auch bei diesem verfeinerten Modell ist unklar, ob es die Wirklichkeit hinreichend genau beschreibt. Dies gilt für die Entwicklung einer geschlossenen Population ohne äußere Einflüsse, und für eine heftigen Einflüssen ausgesetzte Spezies wie den Homo Americanicus sowieso.

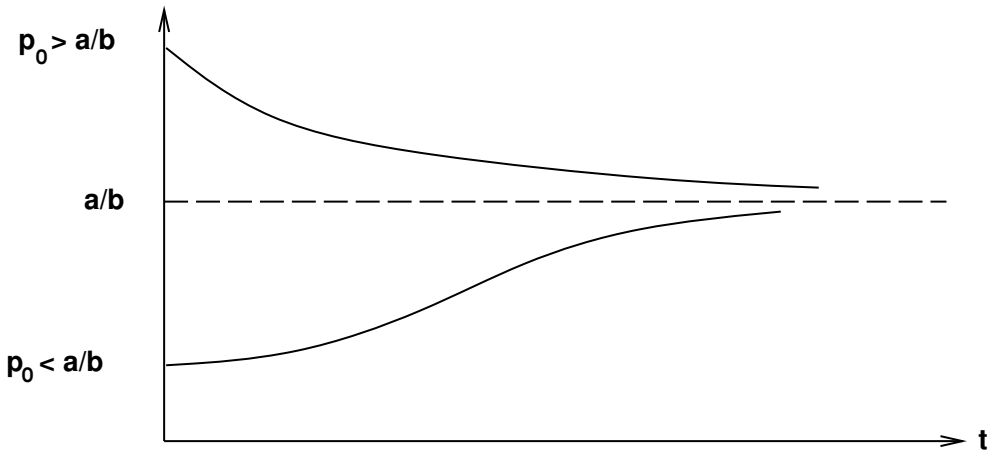


Abbildung 2.2: Entwicklung der Population bei unterschiedlichen Anfangswerten

Natürlich ist obige Differentialgleichung nicht die einzige Möglichkeit, ein S-förmiges (oder *logistisches*) Wachstum zu beschreiben. Bei der mathematischen Modellierung des Wachstums von Tumoren spielen Differentialgleichungen der Gestalt

$$\dot{p}(t) \;=\; \lambda(t) \cdot p(t)$$

mit positivem, monoton fallendem und stetigem $\lambda(t)$ eine Rolle. Ausgangspunkt ist hierbei die (empirirsch begründete) Annahme zeitlich abklingender Wachstumsraten. Wählt man für $\lambda(t)$ etwa ein negativ-exponentielles Wachstum, so erhält man wiederum eine logistisch wachsende Funktion $p(t)$, obwohl man von unterschiedlichen Modellannahmen gestartet ist. Eine genauere Analyse zeigt, dass die beiden Differentialgleichungen eine gewisse Verwandtschaft aufweisen und beide vom Typ

$$\dot{p}(t) \;=\; f(p(t)) \cdot p(t)$$

mit positivem, monoton fallendem und mit wachsendem t gegen Null strebendem f sind. Dies lässt den (zunächst spekulativen, aber empirisch begründeten) Schluss zu, dass das Wachstum geschlossener Populationen diesem Typ von Differentialgleichungen gehorcht, wobei die Wachstumsrate $f(p)$ eine nichtnegative und monoton fallende Funktion der Populationsgröße ist.

Jetzt gehen wir wieder einen Schritt weiter, denn Populationen, die sich ohne äußere Einflüsse über eine längere Zeit ungestört entwickeln können, dürften eher selten sein. Deshalb betrachten wir im Folgenden zwei Populationen p und q, deren zeitliche Entwicklung wir aufgrund der bisherigen Betrachtungen durch das System

$$\begin{aligned} \dot{p}(t) &= f(p(t), q(t)) \cdot p(t)\,, \\ \dot{q}(t) &= g(p(t), q(t)) \cdot q(t) \end{aligned}$$

beschreiben. f und g seien dabei für positive Werte von p und q reellwertige Funktionen, die die Wachstumsraten von p bzw. q darstellen. Eine (als existent angenommene) Lösung $\bar{p},\ \bar{q}$ von

$$f(\bar{p}, \bar{q}) \;=\; g(\bar{p}, \bar{q}) \;=\; 0$$

mit positiven $\bar{p},\ \bar{q}$ heißt *Gleichgewichtslösung* und beschreibt einen stabilen Zustand des Systems: Beide Populationsgrößen ändern sich nicht. Entscheidend ist nun die Frage, ob sich die Populationen im Laufe der Zeit einer solchen Gleichgewichtslösung immer mehr nähern oder nicht, ob diese also *attraktiv* ist. Aus der Theorie gewöhnlicher Differentialgleichungen ist bekannt, dass eine hinreichende Bedingung hierfür negative Realteile der Eigenwerte der Jacobimatrix zu

$$F(p, q) \;:=\; \begin{pmatrix} f(p, q) \cdot p \\ g(p, q) \cdot q \end{pmatrix},$$

also der Eigenwerte von

$$J_F(\bar{p}, \bar{q}) \;=\; \begin{pmatrix} f_p(\bar{p}, \bar{q}) \cdot \bar{p} + f(\bar{p}, \bar{q}) & f_q(\bar{p}, \bar{q}) \cdot \bar{p} \\ g_p(\bar{p}, \bar{q}) \cdot \bar{q} & g_q(\bar{p}, \bar{q}) \cdot \bar{q} + g(\bar{p}, \bar{q}) \end{pmatrix} \;=\; \begin{pmatrix} f_p(\bar{p}, \bar{q}) \cdot \bar{p} & f_q(\bar{p}, \bar{q}) \cdot \bar{p} \\ g_p(\bar{p}, \bar{q}) \cdot \bar{q} & g_q(\bar{p}, \bar{q}) \cdot \bar{q} \end{pmatrix},$$

sind. Aus der Vielfalt möglichen Wachstumsverhaltens wählen wir zwei Spezialfälle aus: *Konkurrenz* und *Räuber-Beute-Charakteristik*.

Bei *Konkurrenz* kämpfen beide Gattungen um denselben Lebensraum. Beide Wachstumsraten nehmen mit wachsender Größe beider Populationen ab:

$$f_p(p, q), \quad f_q(p, q), \quad g_p(p, q), \quad g_q(p, q) \quad < \quad 0$$

für positive p und q. Die Eigenwertbedingung lässt sich in diesem Fall schreiben als

$$f_p(\bar{p}, \bar{q}) \cdot g_q(\bar{p}, \bar{q}) \;-\; f_q(\bar{p}, \bar{q}) \cdot g_p(\bar{p}, \bar{q}) \quad > \quad 0\,.$$

Vereinfacht gesprochen liegt also Attraktivität vor, wenn die Behinderung des jeweils eigenen Wachstums stärker ist als die Behinderung des Fremdwachstums.

Die einfachste Wahl der Raten f und g ist deren Modellierung als lineare Funktionen:

$$\begin{aligned} f(p, q) &= a + b \cdot p + c \cdot q\,, \\ g(p, q) &= d + e \cdot p + f \cdot q \end{aligned}$$

mit positiven a und d, wobei aufgrund unserer Modellannahmen gelten muss:

$$b, c, e, f \quad < \quad 0, \qquad b \cdot f \quad > \quad c \cdot e\,.$$

Damit hat das lineare Gleichungssystem

$$\begin{aligned} f(p, q) &= a + b \cdot p + c \cdot q \;=\; 0\,, \\ g(p, q) &= d + e \cdot p + f \cdot q \;=\; 0 \end{aligned}$$

genau eine Lösung

$$\begin{aligned} \bar{p} &= \frac{cd - af}{bf - ce}, \\ \bar{q} &= \frac{ae - db}{bf - ce}. \end{aligned}$$

Damit dies auch tatsächlich eine Gleichgewichtslösung ist, muss $\bar{p}, \bar{q} > 0$ gelten, was durch

$$\frac{b}{e} > \frac{a}{d} > \frac{c}{f}$$

sichergestellt ist. In diesem (linearen) Modell kann man auch den Fall behandeln, dass die Behinderung des eigenen Wachstums gleich groß wie die Behinderung des Fremdwachstums ist ($bf = ce$). Man erhält dann ausgeartete Gleichgewichtszustände, etwa das Aussterben einer Population oder unbegrenztes Wachstum beider Populationen.

Bei *Räuber-Beute-Verhalten* dient die eine Population – etwa q – der anderen als Beute. Man denke z. B. an Füchse und Hasen. Das Wachstum der Räuberpopulation p wird gebremst bei zunehmender eigener Stärke, es nimmt jedoch zu bei wachsender Beutepopulation. Die Beutepopulation q verhält sich dagegen wie im vorigen Fall. Somit haben wir folgende Situation:

$$f_p(p,q), \quad g_p(p,q), \quad g_q(p,q) \quad < \quad 0\,, \qquad f_q(p,q) \quad > \quad 0\,.$$

Für eine (wiederum als existent angenommene) Gleichgewichtslösung $\bar{p}, \bar{q}$ sind die Realteile der Eigenwerte der Jacobimatrix von $F(p,q)$ immer negativ. Gleichgewichtslösungen sind bei Räuber-Beute-Verhalten also immer attraktiv.

Geht man wieder vom linearen Modell $f(p,q) = a + bp + cq$, $g(p,q) = d + ep + fq$ aus, so sind die obigen Bedingungen gleichbedeutend mit

$$b, e, f \quad < \quad 0, \qquad c \quad > \quad 0\,,$$

so dass $b \cdot f > c \cdot e$ in der Tat automatisch erfüllt ist.

Im klassischen Räuber-Beute-Modell nach Volterra und Lotka wird $f = b = 0$ angenommen, die jeweils eigene Populationsgröße ist also ohne Einfluss auf die Wachstumsraten. Weiterhin geht man von $a < 0$ und $d > 0$ aus. Dies führt zu

$$\bar{p} \;=\; -\frac{d}{e}, \qquad \bar{q} \;=\; -\frac{a}{c}.$$

Sowohl $\bar{p}$ als auch $\bar{q}$ sind positiv, und beide Eigenwerte der Jacobimatrix sind rein imaginär. Trotzdem kann die Stabilität bzw. Attraktivität der Gleichgewichtslösung gezeigt werden.

Wärmeleitung

Unsere obigen Populationsmodelle kennen als einzige Variable die Zeit. Eine räumliche Verteilung wird nicht betrachtet. Deshalb lassen sich alle Zusammenhänge mittels gewöhnlicher Differentialgleichungen beschreiben. Eine detailliertere Betrachtung, die auch die räumliche Auflösung mit einbezieht, könnte allerdings beispielsweise dann sinnvoll sein, wenn wir nicht die Bevölkerungsentwicklung in den gesamten USA, sondern in jedem Staat oder County separat modellieren wollen. In einem solchen Fall müssen ortsabhängige Parameter (Sterberaten, Geburtenraten) eingeführt werden, und die Fluktuation von einer räumlichen Zelle in die benachbarten muss berücksichtigt werden (etwa aufgrund des Siedlerstroms nach Westen oder während des Goldrausches in Kalifornien). Somit hängt die Populationsstärke $p(t)$ nicht mehr nur von der Zeit t, sondern auch vom betrachteten Ort ab. Ein solches Modell beschreibt die Realität sicher präziser, es ist jedoch komplizierter als das rein zeitabhängige und macht die (numerische) Lösung aufwendiger.

Was im Falle der Populationsmodelle als etwas gekünstelt erscheinen mag, ist für die Modellierung vieler physikalischer Phänomene Standard. Das gebotene Hilfsmittel sind dann *partielle* Differentialgleichungen, bei denen (partielle) Ableitungen nach verschiedenen Variablen auftreten (verschiedene Ortskoordinaten bei *stationären* Problemen oder Ortsvariable und die Zeit bei *instationären* oder *zeitabhängigen* Problemen). Als einfaches Beispiel betrachten wir die Wärmeleitung.

Die Wärmeleitung spielt in der Thermodynamik eine zentrale Rolle. Wir stellen uns etwa eine quadratische Metallplatte oder ein mit Wasser gefülltes Gefäß vor. Vom Rand des jeweils betrachteten Gebiets geht dabei ein definierter Einfluss aus (Kühlung, Heizung, Isoierung etc.). Man interessiert sich nun dafür, wie sich die Randvorgabe im Verlauf der Zeit im Inneren des Gebiets auswirkt, wie sich also die Temperatur $T(x, y, t)$ bzw. $T(x, y, z, t)$ als Funktion von Raum und Zeit entwickelt. Der Verlauf von T wird natürlich stark vom jeweiligen Material und von dessen Wärmeleitfähigkeit abhängen. Das mathematische Modell hierfür ist die sogenannte *Wärmeleitungsgleichung*, die im dreidimensionalen Fall folgende Gestalt hat (im positiven Parameter $c > 0$ stecken dabei die Materialeigenschaften):

$$c^2 \cdot \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) \quad = \quad \frac{\partial T}{\partial t} \,,$$

bzw. mit dem Laplace-Operator und der üblichen Indizierung der Ableitungsrichtung

$$c^2 \cdot \Delta T \quad = \quad T_t \,.$$

Zusätzlich müssen – um die Eindeutigkeit der Lösung sicherzustellen – Rand- oder Anfangs- und Randbedingungen vorgegeben werden. Man unterscheidet

- *Dirichlet*-Randbedingungen: $T(x, y, z) = \varphi(x, y, z)$ für (x, y, z) auf dem Rand $\delta\Omega$ des betrachteten Gebiets Ω,
- *Neumann*-Randbedingungen: $\frac{\partial T}{\partial n}(x, y, z) = \varphi(x, y, z)$ für $(x, y, z) \in \delta\Omega$.

Im Falle $\varphi(x, y, z) \equiv 0$ spricht man von *homogenen* Randbedingungen. Natürlich ist auch eine Mischung von Dirichlet- und Neumann-Randbedingungen möglich.

Hängt der Temperaturverlauf nicht von der Zeit ab, so handelt es sich um ein stationäres Problem. Ein Beispiel hierfür ist die Temperaturverteilung in einem konstant beheizten Raum. Die Ableitung nach der Zeit verschwindet, und man erhält die Laplace-Gleichung, eine *elliptische Differentialgleichung* zweiter Ordnung. Zu deren eindeutiger Lösbarkeit ist bspw. der Temperaturverlauf am Rand des Gebiets vorzuschreiben (Dirichlet-Randbedingungen). Man kann aber auch die Temperatur nur an Teilen des Randes vorgeben und am Rest des Randes homogene Neumann-Randbedingungen vorschreiben (verschwindende Ableitung der Temperatur in Normalenrichtung bzw. Wand ohne orthogonalen Wärmetransport (keine Kühl- oder Heizwirkung)). Überall Neumann-Randbedingungen anzusetzen reicht jedoch nicht: Wie man leicht sieht, wäre dann zu einer gefundenen Lösung $T(x, y, z)$ auch $T + \alpha$ für eine beliebige Konstante α Lösung unseres Problems. Eine solche Aufgabenstellung (löse eine elliptische Differentialgleichung unter Randbedingungen) nennt man auch ein *elliptisches Randwertproblem*.

Bei instationären Problemen hängt die Temperatur auch von der Zeit ab. Ein Beispiel hierfür ist die Temperatur in einem Raum mit Fenstern, die einer im Tagesverlauf wechselnden Sonneneinstrahlung ausgesetzt sind. In diesem Fall ergibt sich eine *parabolische Differentialgleichung* zweiter Ordnung. Üblicherweise wird hier eine Mischung aus Anfangs- und Randwerten vorgeschrieben, etwa

$$\begin{aligned} T(x, y, 0) &:= \tau(x, y) \quad \text{für} \quad (x, y) \in [0, 1]^2 \,, \\ T(x, 0, t) &:= \varphi_0(x, t) \quad \text{für} \quad x \in [0, 1], t > 0 \,, \\ T(x, 1, t) &:= \varphi_1(x, t) \quad \text{für} \quad x \in [0, 1], t > 0 \,, \end{aligned}$$

und homogene Neumann-Randbedingungen für $x = 0$ und $x = 1$. Ein solches Problem heißt *Anfangsrandwertproblem*. Was bedeuten obige Vorgaben? Bzgl. der Zeit werden Anfangswerte

vorgegeben: Zur Startzeit $t = 0$ sei der Temperaturverlauf auf unserer Metallplatte durch $\tau(x, y)$ beschrieben. Zu jedem späteren Zeitpunkt sind die Werte an zwei Seiten der quadratischen Platte vorgeschrieben (Dirichlet-Werte für $y \in \{0, 1\}$), die restlichen zwei Seiten seien perfekt isoliert (homogene Neumann-Randwerte für $x \in \{0, 1\}$).

Sowohl der stationäre als auch der instationäre Fall sind in der Regel nur numerisch lösbar, wobei letzterer offensichtlich aufwendiger ist. Die Unterscheidung in elliptische, parabolische sowie hyperbolische Probleme ist in den z. T. erheblich unterschiedlichen Eigenschaften der Klassen (analytisch wie numerisch) begründet. Wir werden hierauf im nächsten Kapitel zurückkommen.

In den vorigen Beispielen haben wir gesehen, dass manche Phänomene mit Raumauflösung modelliert werden müssen, für andere dagegen Modelle mit oder ohne Raumauflösung existieren. Die entscheidenden Fragen bei der Auswahl des geeigneten Modells sind dabei, welche Phänomene im Modell berücksichtigt werden sollen bzw. müssen (Genauigkeit des Modells) und welche Genauigkeit praktikabel ist im Hinblick auf den zu erwartenden Aufwand bei der numerischen Lösung des Modells. Oft stellt sich die Frage nach der Betrachtung auch des Orts nicht nur als 0-1-Entscheidung, sondern man hat auch die Wahl, in welcher Dimensionalität das Problem modelliert werden soll. Niedrigdimensionale Modelle (z. B. eine Raumdimension plus Zeit) können dabei ein Kompromiss zwischen den (billigen, aber ungenauen) Modellen ohne Raumauflösung und den (genauen, aber teueren) hochdimensionalen Modellen (drei Ortskoordinaten plus Zeit) sein. Denn die numerische Lösung partieller Differentialgleichungen ist ja vor allem deshalb so aufwendig, weil die Auflösung des Raumes die Zahl der Freiheitsgrade (Gitterpunkte) stark anwachsen lässt. Standardverfahren benötigen eine exponentiell in der Anzahl der betrachteten Dimensionen wachsende Zahl von Gitterpunkten, also etwa $O(h^{-d})$, um eine partielle Differentialgleichung in d Dimensionen (Raum plus Zeit) mit einer Maschenweite (Feinheit der Diskretisierung, tritt auch beim bekannten Maschendrahtzaun auf) h numerisch zu lösen – ein Dilemma, das auch als *Fluch der Dimensionalität* bezeichnet wird. Deshalb liegt es nahe, sich zunächst zu überlegen, ob nicht auch ein in der Zahl der betrachteten Dimensionen reduziertes und damit vereinfachtes Modell verwendet werden kann. Dies ist offensichtlich beispielsweise dann ohne Genauigkeitsverlust der Fall, wenn ein zylinderförmiges Rohr von außen konstant gekühlt wird: Das Problem ist rotationssymmetrisch, die dritte Raumdimension ist entbehrlich. In anderen Fällen bedeutet eine solche Modellreduktion zwar vielleicht einen Informationsverlust, erlaubt aber eine schnellere (wenn auch ungenauere) Lösung. Mit dieser Problematik werden wir uns im Abschnitt über hierarchische Modelle noch näher befassen.

2.2.3 Stochastische Modelle

Schließlich wollen wir den rein deterministischen Bereich verlassen und den Zufall in die Modelle (diskret oder kontinuierlich) einbauen. Als Beispiel hierfür sollen *Wartesysteme* und *Wartenetze* betrachtet werden. Diese dienen zur Modellierung einer Vielzahl realer Systeme, beispielsweise des Verkehrsflusses durch eine Stadt mit Ampeln oder auf stark befahrenen Autobahnen, wo man mit (mehr oder minder ausgefeilten) Verkehrsleitsystemen Staus vermeiden möchte. Wir studieren ein „informatischeres“ Beispiel, den Verkehrsfluss in Rechensystemen. Zunächst brauchen wir einiges an Begriffen.

- Eine *Funktionseinheit* (FE) ist ein durch Aufgabe oder Wirkung abgrenzbares Gebilde. Wir unterscheiden *Bedieneinheiten* (BE, kein Datentransport) und *Kanäle* (Datentransport zwischen Bedieneinheiten). Ein *System* sei eine aus mehreren FEen zusammengesetzte FE.
- Ein *Auftrag* beschäftigt eine FE. Vom Auftragsbeginn bis zur Fertigstellung des Auftrags *verweilt* der Auftrag in der FE (*Verweilzeit* y). Die *Füllung* f einer FE gibt an, wie viele Aufträge gerade in der FE verweilen: Bei $f = 0$ ist die FE *frei*, bei $f > 0$ ist sie *beschäftigt*. Die maximale Füllung heißt *Kapazität* k. Im Falle $k = 1$ spricht man von *einfachen* FEen, bei $f = k$ ist die FE *belegt*. Der Quotient f/k bezeichnet die *relative Füllung*.

- Unter dem *Durchsatz* d einer FE versteht man die Anzahl der von ihr in einem Zeitintervall vollendeten Aufträge, der maximal mögliche Durchsatz wird als *Grenzdurchsatz* c bezeichnet. Der Quotient $\rho = d/c$ gibt die *Auslastung* einer FE an.
- Unter der *Bedienzeit* b versteht man schließlich die Netto-Verweilzeit eines Auftrags in einer FE, d. h. die Verweilzeit bei $f = 1$.

Von zentraler Bedeutung in der sogenannten Verkehrstheorie ist die *Formel von Little*. Für ein bestimmtes Zeitintervall $[t_1, t_2]$ besagt sie, dass die mittlere Füllung einer FE gleich dem Produkt aus Durchsatz und mittlerer Verweilzeit ist (es gelte $f(t_1) = f(t_2) = 0$):

$$\bar{f}(t_1, t_2) \;=\; d(t_1, t_2) \cdot \bar{y}(t_1, t_2)\,.$$

Anstelle von auf ein Zeitintervall bezogenen Durchschnittsgrößen kann man die Formel auch mittels Erwartungswerten von entsprechenden Zufallsvariablen schreiben:

$$E(F) \;=\; E(D) \cdot E(Y)\,.$$

Die Formel von Little erweist sich bei näherer Betrachtung eigentlich als Binsenweisheit: Wenn in einer Fabrik täglich durchschnittlich hundert Autos produziert werden und die Fertigstellung eines Autos einen halben Tag erfordert, dann sind stets durchschnittlich fünfzig Autos in der Fabrik in Arbeit.

Im Sonderfall einer einfachen FE ist die Verweilzeit gleich der Bedienzeit (siehe obige Definition der Bedienzeit), die ihrerseits dann gleich dem Kehrwert des Grenzdurchsatzes ist (die Bedienzeit bestimmt den maximalen Output). Somit gilt

$$E(F) \;=\; E(D) \cdot E(B) \;=\; E(D) \cdot \frac{1}{c} \;=\; E(\rho)\,.$$

Jetzt gehen wir einen Schritt weiter und bauen Funktionseinheiten zusammen. Ein *Wartesystem* ist dabei ein System, in dem Aufträge nicht verloren gehen. Bei belegten Bedieneinheiten warten sie in Kanälen. Einfachster Vertreter ist das *elementare Wartesystem* (siehe Abb. 2.3), bestehend aus einem Kanal der Kapazität k_{WP} als Wartepool und einer Bedieneinheit der Kapazität k_{BE} bzw. k_{BE} einfachen Bedieneinheiten (oder *Bedienstationen BS*).

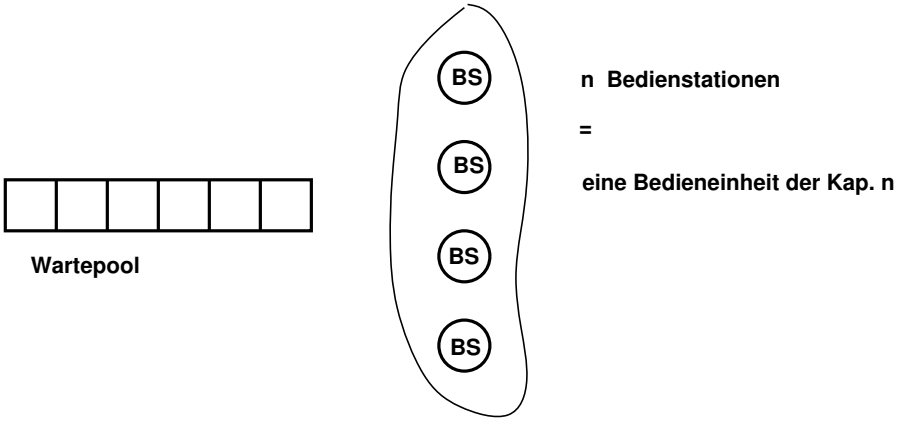


Abbildung 2.3: Das elementare Wartesystem

Für solche Wartesysteme soll die Auftragsabwicklung modelliert werden, und zwar mit Hilfe *stochastischer Prozesse*. Obwohl Auftragseingang und Auftragsbearbeitung in Rechensystemen i. a. deterministische Prozesse sind (es sei denn, an einer Stelle wird Wahrscheinlichkeit gezielt ins Spiel gebracht, etwa bei einer zufälligen Auswahl des nächsten zu bearbeitenden Auftrags), erweist sich

eine stochastische Betrachtungsweise als vernünftig. Dies liegt zum einen daran, dass wir bei der Beurteilung der Leistungsfähigkeit eines Rechners aufgrund der großen Zahl von Aufträgen mehr an gemittelten Größen als am Schicksal eines einzelnen Auftrags interessiert sind. Zum anderen sind aus der lokalen Sicht einer Funktionseinheit der Auftragseingang und aus der Sicht der „Umwelt" die Bedienzeit in der Tat „zufällig". Ein stochastischer Prozess ist nun eine Schar von (i. d. R. gleichverteilten) Zufallsvariablen, die eine Folge (gleichartiger) Ereignisse beschreiben. Alternativ kann man sich auch eine Zufallsvariable vorstellen, die fortlaufend Ausprägungen produziert. Der *Ankunftsprozess* beispielsweise modelliert den Auftragseingang. Für die Abstände A zwischen zwei Auftragseingängen, die sog. *Zwischenankunftszeiten*, nimmt man an, dass sie alle derselben Verteilung gehorchen, etwa

- D: deterministisch, d. h. konstant,
- M: Markovian, d. h. negativ-exponentiell verteilt mit Dichte $1 - e^{-\lambda \cdot t}$ und Erwartungswert $1/\lambda$,
- G: general, d. h. beliebig verteilt.

Beim ersten, deterministischen Fall „D" hat man es also streng genommen gar nicht mit stochastischen Prozessen zu tun. Im zweiten Fall „M" ist der Ankunftsprozess dagegen völlig zufällig und *gedächtnislos*, d. h. unabhängig von der Vorgeschichte:

$$p(A > t_2 \mid A > t_1) \;=\; p(A > t_2 - t_1)\,.$$

Ein solcher Prozess wird *Poisson-Prozess* genannt. Der Parameter λ heißt *Ereignisrate* und gibt den Auftragseingang an:

$$\lambda \;=\; E\left(\frac{\text{Zahl der Ankünfte in } [t_1, t_2]}{t_2 - t_1}\right)\,.$$

Falls $\lambda < c$, dem Grenzdurchsatz unseres elementaren Wartesystems, dann gilt $E(D) = \lambda$, denn dann gehen weniger Aufträge ein, als das elementare Wartesystem bearbeiten kann. Dieser Markovsche Fall ist bei den Modellierern sehr beliebt, weil er in einer Vielzahl von Fällen die Realität hinreichend genau beschreibt und zudem sehr schön zu analysieren ist. Man beachte, dass dieser Markovsche Fall unserer Vorstellung allerdings oft widerspricht: Auf Landstraßen treten Autos gern in Rudeln auf, ein vorbeifahrendes Auto scheint also die Wahrscheinlichkeit eines weiteren zu erhöhen, die erwartete Zwischenankunftszeit also zu reduzieren. Umgekehrt vermuten wir aus „Fairnessgründen" oft eine Gleichverteilung: Der Betreiber einer Würstchenbude rechnet nach einer mehrminütigen Pause fest damit, dass wieder ein Kunde kommt, die Vorgeschichte fließt also sehr wohl in sein Kalkül ein.

Der allgemeine Fall „G" schließlich bietet zwar alle Möglichkeiten einer detailgetreuen Modellierung, dafür lassen sich allgemeingültige Aussagen hierfür jedoch nur sehr selten herleiten.

Soviel zum Auftragseingang. Auf der anderen Seite steht der *Bedienprozess* mit von den Auftragseingängen völlig unabhängigen Bedienzeiten B. Wie zuvor unterscheidet man die Klassen D, M und G. Daraus resultiert dann folgende Beschreibung eines Wartesystems:

$$\text{Ankunftsprozess} \mid \text{Bedienprozess} \mid \text{Zahl der Bedienstationen } n \mid k_{WP} \mid N\,.$$

Hierbei bezeichnet N die Gesamtzahl der Aufträge im System, und k_{WP} gibt die Kapaität des Wartepools an. Die letzten beiden Parameter sind optional, man schreibt also etwa $M \mid M \mid 1$ oder $M \mid G \mid 1$ für elementare Wartesysteme mit negativ-exponentiell verteiltem Ankunftsprozess, einer Bedienstation und negativ-exponentiell bzw. beliebig verteiltem Bedienprozess. Zusätzlich kann man eine *Bedienstrategie* spezifizieren, also etwa

- nicht verdrängend: random, first-come-first-served, last-come-first-served, priority, shortest-job-next, ...

- verdrängend: round robin, LCFS-preemptive, priority, ...

Als wichtige Beziehungen in elementaren Wartesystemen halten wir fest:

$$\begin{aligned}
c_{BE} &= \frac{n}{E(B)} = n \cdot c_{BS}\,, \\
\rho_{BE} &= \frac{E(D)}{c_{BE}}\,, \\
E(F_{WP}) &= E(D) \cdot E(W)\,, \qquad \text{Wartezeit } W \\
E(F_{WS}) &= E(D) \cdot E(Y_{WS}) = E(D) \cdot (E(W) + E(B))\,, \\
E(F_{WS}) &= E(F_{WP}) + E(F_{BE})\,.
\end{aligned}$$

Jetzt gehen wir wieder einen Schritt weiter: Für moderne Rechensysteme ist das elementare Wartesystem als Modell natürlich zu primitiv. Deshalb betrachtet man nun *Wartenetze*, d. h. Netze aus elementaren Wartesystemen. Ein Wartenetz modelliert man z. B. als Graph mit elementaren Wartesystemen als Knoten und potentiellen Auftragswegen als Kanten. Jeder einzelne Auftrag durchläuft seriell bestimmte Knoten, in denen Teilaufträge bearbeitet werden, das Gesamtnetz bearbeitet aber mehrere Aufträge im Simultanbetrieb. Man unterscheidet *geschlossene* Wartenetze (konstante Füllung) und *offene* (schwankende Füllung). Alle bisher eingeführten Größen werden jetzt mit einem Index $i, 1 \le i \le M$, für die Nummer des Knotens indiziert: $\lambda_i, E(D_i), c_i, \rho_i, E(B_i)$, außerdem gibt es diese Größen für das Gesamtnetz: $\lambda_S = \sum_{i=1}^{M} \lambda_i, c_S, \ldots$. Jeder Knoten j mit maximaler Auslastung $\rho_j = \max \rho_i$ heißt *Verkehrsengpass* des Systems. Die *Besuchszahl* v_i des Knotens i gibt schließlich das Verhältnis des Durchsatzes in i (in Teilaufträgen) zum Durchsatz des Systems (in Aufträgen) an:

$$v_i := \frac{E(D_i)}{E(D_S)}\,.$$

Damit ist man jetzt in der Lage, Aussagen über relevante Größen in derartigen Wartenetzen zu treffen. Die Analyse geschlossener Netze etwa führt zu folgenden Resultaten (vgl. Abb. 2.4):

- Es gibt eine *Sättigungsfüllung* $f_S^* = c_s \cdot E(B_S)$: Ab dieser Füllung kann der Durchsatz nicht mehr wesentlich gesteigert werden, die Verweilzeit steigt aber munter linear weiter.
- Für die Grenzfälle $f_S = 1$ und $f_S = \infty$ ergibt sich ein asymptotisches Anschmiegen. Die Annäherung an die Asymptoten $E(D_S) = c_S$ und $E(D_S) = \frac{f_s}{E(B_S)}$ beim Durchsatz bzw. $E(Y_S) = E(B_S) = \sum_{i=1}^{M} v_i \cdot E(B_i)$ und $E(Y_S) = \frac{f_S}{c_S}$ ist umso besser, je geringer die B_i streuen.
- Der Übergang von serieller ($f_S = 1$) zu simultaner Auftragsbearbeitung ($f_S > 1$) verbessert den Verkehrsfluss bei kleiner Füllung, bei großer ist er dagegen ohne Einfluss – das Warten überwiegt. Die Kenntnis der Sättigungsfüllung kann somit zur Steuerung des Systems, d. h. zur Kontrolle des Zulaufs an Aufträgen benutzt werden.

Mit diesem Modell kann man allerlei beobachtbare Erscheinungen in Rechensystemen darstellen, so etwa auch das *Seitenflattern*: Die Besuchszahl der Platte pro Aufrag wächst proportional zur Seitentauschwahrscheinlichkeit. Steigt der Mehrprogrammgrad über einen bestimmten Wert, dann wird die Platte zum Engpass, und jede weitere Erhöhung der Füllung bzw. des Mehrprogrammgrads ruiniert den Durchsatz. Wir haben somit ein mächtiges mathematisches Modell zur Beschreibung des Verkehrsflusses in Rechensystemen zur Verfügung, das – in erweiterter Form – auch tatsächlich zur Leistungsanalyse heutiger Anlagen herangezogen wird.

Bisher haben wir die stochastischen Prozesse selbst noch nicht zur Analyse benutzt. Das wollen wir zum Schluss dieses Ausflugs in die Stochastik noch tun. Rechensysteme besitzen einen diskreten, endlichen Zustandsraum. Alle Zustände sind (u. U. nur indirekt) untereinander erreichbar. Werden die Zustandsübergänge von Poisson-Prozessen getrieben (zufällig, negativ exponentialverteilt,

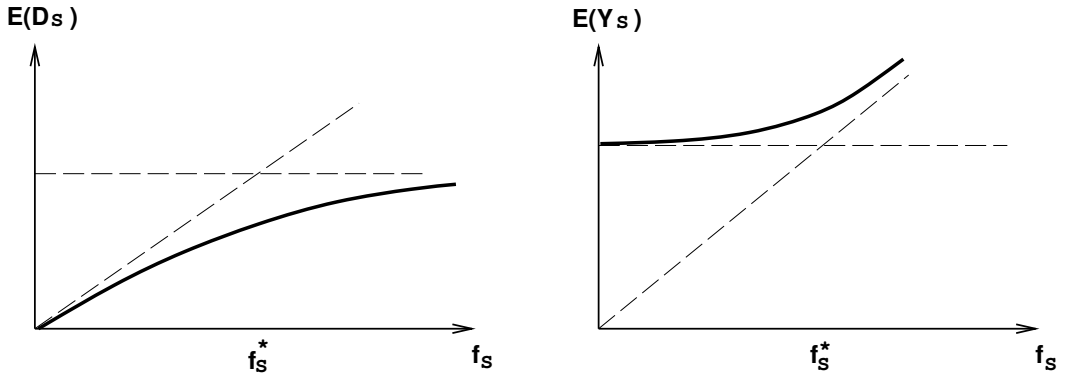


Abbildung 2.4: Sättigung bei der Auftragsbearbeitung

gedächtnislos), dann spricht man auch von *homogenen Markovschen Prozessen in kontinuierlicher Zeit* bzw. von *Markovschen Systemen*. Genauer:

- Gegeben seien n Zustände mit *Übergangsraten* λ_{ij}:

$$\lambda_{ij} \;=\; \lim_{\Delta t \to 0} \frac{p(j \text{ in } t+\Delta t \mid i \text{ in } t)}{\Delta t}\,.$$

- Alle Aufenthaltszeiten T_i in i sind negativ-exponentiell verteilt mit Raten $\lambda_i = \sum_{j \neq i} \lambda_{ij}$.
- Unabhängig vom Startzustand bestehen asymptotisch Zustandswahrscheinlichkeiten p_i, die zeitkonstant sind und für die gilt:

$$\sum_{j=1}^{n} p_i \;=\; 1, \qquad p_i \cdot \sum_{j \neq i} \lambda_{ij} \;=\; \sum_{j \neq i} p_j \cdot \lambda_{ji}\,.$$

Für unser System $M \mid M \mid 1$ bedeutet das bei einem Ankunftsprozess mit Ankunftsrate λ und einem Bedienprozess mit Bedienrate μ, wenn Zustand i gerade den Füllgrad des Systems angibt, $0 \leq i \leq \infty$:

$$p_i \;=\; \left(\frac{\lambda}{\mu}\right)^i \cdot p_0\,.$$

Damit ergibt sich p_0 zu $1 - \lambda/\mu$. Man kann zeigen, dass

$$\frac{\lambda}{\mu} \;=\; \rho\,.$$

Man beachte, dass der Fall $\lambda > \mu$ nicht sinnvoll ist: Wenn die Aufträge schneller kommen, als sie erledigt werden können, muss das Ganze in einer Katastrophe enden!

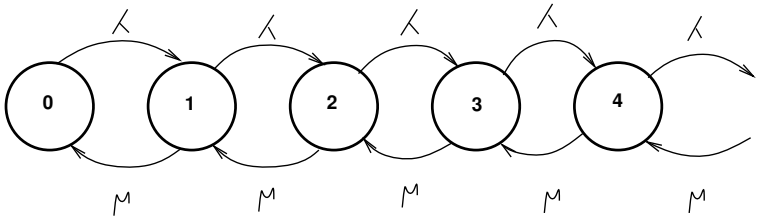


Abbildung 2.5: Markov-Kette

Der Anwendungsbereich stochastischer Modelle ist natürlich nicht auf die Verkehrstheorie beschränkt. So findet man beispielsweise in der Physik zahlreiche Modellieransätze, die sich auf Wahrscheinlichkeiten abstützen (Brownsche Bewegung, Aufenthaltswahrscheinlichkeit von Elektronen in Orbitalen, Verharren von Makromolekülen in semistabilen Zuständen (Konformationen) etc.).

2.3 Mehrskaligkeit und hierarchische Modelle

2.3.1 Hierarchische Modellierung

Vor der Modellierung muss man sich Gedanken machen, auf welcher *Skala* man die zu modellierenden Phänomene betrachten will. Ein Beispiel für solch unterschiedliche Betrachtungsebenen desselben Phänomens haben wir bei den Populationsmodellen schon gesehen: Die Entwicklung von Populationen in ihrem Lebensraum kann entweder ohne Ortsauflösung modelliert werden (eine grobe Skala ohne räumliche Details) als System gewöhnlicher Differentialgleichungen, oder man nimmt auch den Ort in das Modell auf (feine Skala mit Orts- und Zeitvariablen) und gelangt zu einem System partieller Differentialgleichungen. Auch bei Berücksichtigung einer räumlichen Auflösung gibt es je nach Anzahl der verwendeten Raumdimensionen Unterschiede in der Modellpräzision. Die jeweils gebotene Vorgehensweise ergibt sich im Wesentlichen aus der Beantwortung zweier Fragen:

- Welcher Grad von Detailliertheit ist für die konkret interessierende Fragestellung überhaupt erforderlich?
- Welches Modell kann ich mit den derzeit verfügbaren numerischen Verfahren und Rechnern in sinnvoller Zeit numerisch behandeln?

Dabei ist klar, dass die Machbarkeitsgrenze eine harte ist. Auf der anderen Seite ist aber keinesfalls immer alles erforderlich und damit sinnvoll, was numerisch machbar ist. Schließlich empfiehlt sich auch oft ein *hierarchisches* Modellieren: Für eine erste grobe Simulation im Sinne eines Prototyping reicht vielleicht ein grobes Modell, sei es ein Modell ohne Raumauflösung oder ein in seiner Dimensionalität reduziertes Modell (2 D statt 3 D). Wenn dann die Details durch Simulation festgelegt werden sollen (der optimale Zündzeitpunkt eines Otto-Motors, die exakte Positionierung der Einströmdüsen einer Klimaanlage), ist jedoch ein genaueres Modell erforderlich. Man betrachtet die zu modellierenden Phänomene also auf verschiedenen Skalen.

Ein typisches Beispiel dafür, dass im Laufe der Zeit immer höhere Genauigkeitsanforderungen an die Simulation (und somit auch an die zugrunde liegenden Modelle) gestellt werden, ist die Halbleitertechnologie. Hochintegrierte Schaltkreise werden nach der Entwurfsphase durch numerische Simulation getestet, bevor man an das (teure) Erstellen von Prototypen oder die (noch teuerere) Serienproduktion geht. Angesichts der Vielzahl von einzelnen Transistoren auf dem Chip (mehrere Millionen bei modernen Mikroprozessoren) ist dabei an eine raumauflösende Simulation aller Transistoren auf der Grundlage der Gleichungen der Elektrodynamik (Maxwell-Gleichungen) nicht zu denken. Deshalb gelangen hier Schaltkreissimulatoren wie SPICE oder TITAN zum Einsatz, die die Simulation von großen Systemen gewöhnlicher Differentialgleichungen gestatten. Als Modell dienen hier also im Wesentlichen das kennlinienbasierte Verhalten der einzelnen Bauelemente sowie die rein zeitabhängigen Kirchhoffschen Gesetze. Durch den immer höheren Integrationsgrad der Chips stößt diese Vorgehensweise jedoch an Grenzen: Die Transistoren und Leiterbahnen liegen so dicht beieinander, dass sie beginnen, sich (jenseits der gewollten und bisherigen Effekte) wechselseitig zu beeinflussen. Es kommt zu sogenannten *parasitären Effekten* (man spricht in diesem Zusammenhang auch vom Problemkreis der *elektromagnetischen Verträglichkeiten*). Auch wenn man nicht zur völligen Integration des Raums übergehen kann, müssen solche und andere räumlich bedingten Effekte in das bisherige Modell aufgenommen werden, sei es durch zusätzliche Parameter, durch Ankopplung niedrig- (d. h. i. d. R. ein-) dimensionaler partieller Differentialgleichungen,

durch Verbesserung der Kennlinien und Parameter der gewöhnlichen Differentialgleichungen mittels raumauflösender Simulation einzelner kritischer Bauelemente oder auf eine andere Weise.

2.3.2 Mehrskaligkeit

Wie wir soeben gesehen haben, kann man ein gegebenes Problem auf verschiedenen Ebenen oder Skalen betrachten bzw. modellieren und gelangt so zu unterschiedlichen Modellen. Es gibt aber auch den Fall, dass innerhalb einer Betrachtungsebene, also innerhalb eines Grundmodells, Phänomene auf verschiedenen Skalen auftreten. Damit bei der numerischen Behandlung nicht die kleinste auftretende Skala zur die Auflösung bestimmenden Größe wird, werden hier oftmals die kleinskaligen Strukturen speziell modelliert.

Als Beispiel betrachten wir *turbulente* Strömungen. Im Gegensatz zu *laminaren* Strömungen, bei denen die Vorgänge langsam und geordnet ablaufen (fließender Honig, Strömung des Mississippi zu Normalzeiten), treten bei Turbulenz starke und unregelmäßige Verwirbelungen auf. Man denke etwa an die Strömung eines Flusses über ein Wehr bzw. eine Stufe oder an Luftverwirbelungen im Nachlaufbereich eines startenden Düsenflugzeugs. Letztere sind so stark, dass sie für die unteren Sicherheitsschranken bei den Startintervallen (derzeit i. A. ca. 45 Sekunden) verantwortlich sind – die Turbulenzen hinter einem startenden Jumbo-Jet könnten bei Nichteinhaltung eine Boeing B 737 durch die Luft schleudern! Diese Wirbel im Nachlaufbereich sind auch ein Problem beim derzeit in der Vorentwicklung befindlichen Airbus 3XX: er würde nach Berechnungen Startintervalle von fast fünf Minuten erfordern! Als wichtige Eigenschaften turbulenter Strömungen sind festzuhalten: Turbulente Strömungen sind instationär und i. d. R. dreidimensional (d. h. nicht in einer Ebene verlaufend), die starke Verwirbelung führt zu einer hohen Diffusivität (Stoff- und Energietransport in alle Richtungen), es treten Wirbel unterschiedlichster Größenordnungen auf, wobei zwischen den Skalen ebenfalls ein starker Energietransport stattfindet (Energie-Kaskade), und es wird viel Energie vernichtet (d. h. dem Primärzweck entzogen). Letzteres zeigt sich etwa bei Pipelines, in denen Turbulenzen die zum Transport des Fluids notwendige Energie drastisch erhöhen. Hier lassen sich durch Beigabe polymerer Additive (geeignete Makromoleküle, die in der Strömung die Form langgestreckter Stäbchen annehmen und somit reibungs- und damit widerstandsvermindernd wirken) Energieeinsparungen bis zu 70% erzielen.

Die Mehrskaligkeit zeigt sich bei turbulenten Strömungen also vor allem in der Charakteristik der Wirbel. Sehr große und langsame Wirbel überlagern sich mit sehr kleinen und schnellen Wirbeln unterschiedlichster Ausrichtung. Will man nun auf der Ebene der Erhaltungsgleichungen der Kontinuumsmechanik für Energie, Masse und Impuls (Navier-Stokes-Gleichungen) alle auftretenden Wirbel numerisch, d. h. mit einem Gitter auflösen (*Direkte Numerische Simulation*, DNS), so liefert das sogenannte $k^{-5/3}$-Gesetz von Kolmogorov, dass die kleinsten auftretenden relevanten Wirbel von der Größenordnung $\nu^{3/4}$ sind, wobei ν die *Viskosität* oder *Zähigkeit* des Fluids angibt. Typische Werte für Luft, etwa bei aerodynamischen Untersuchungen, liegen bei $\nu < 10^{-6}$, was also eine relevante Wirbelgröße von $h < 10^{-9/2}$ bedeutet. Dementsprechend engmaschig müssen die verwendeten Gitter sein. Dreidimensionale Berechnungen mit einer derart kleinen Maschenweite sind auf absehbare Zeit nicht durchführbar. Auf der anderen Seite sind die gängigen Turbulenzmodelle, die auf die Auflösung der Wirbel verzichten und stattdessen statistische (d. h. gemittelte) Größen verwenden, oftmals zu ungenau.

Zur Abhilfe der Problematik der Mehrskaligkeit wurden deshalb die sogenannten *Feinstrukturmodelle* (auch *Large Eddy Simulation* oder kurz LES genannt) eingeführt. Hier werden die groben und konfigurationsabhängigen Strukturen bzw. Wirbel direkt aufgelöst, wohingegen die feinskaligen Wirbel durch vereinfachende Modelle (*subgrid scale models*) erfasst werden. Insofern stellen die Feinstrukturmodelle sowohl hinsichtlich Realitätsnähe als auch hinsichtlich Rechenzeit- und Speicherplatzbedarf bei der numrischen Lösung einen Mittelweg zwischen der direkten numerischen Simulation und den statistischen Turbulenzmodellen dar.

Unterschiedliche Skalen gibt es aber nicht nur bei der Betrachtung des Raumes, sondern auch in der

Zeit. In der Molekulardynamik etwa überlagern sich hochfrequente Oszillationen mit extrem niederfrequenten Zustandsänderungen, wobei die Unterschiede mehrere Größenordnungen betragen. Beim Wachstum von Biofilmen (Agglomerationen von Mikroorganismen, etwa auf Kieselsteinen in Seen, als Plaque auf den Zähnen oder in biologischen Abwasserreaktoren) kommt es im Innern der Biofilme zu Strömungs- und Transportprozessen, die z. T. bis zu zehn Größenordnungen schneller ablaufen als dann das resultierende und beobachtbare Wachstum der Biomasse. Es ist klar, dass eine Simulation mit alle Skalen berücksichtigenden Zeitschritten außerhalb des Machbaren liegt. Auch hier müssen Techniken entwickelt werden, mit deren Hilfe der Einfluss der feinskaligen Phänomene ohne wesentlichen Genauigkeitsverlust in ein an sich grobes Modell integriert werden kann.

2.3.3 Mittelung und Homogenisierung

Ein wunderschönes Beispiel für unterschiedliche Betrachtungsebenen bei der Modellierung stellt das Studium von Transportvorgängen in porösen Medien dar. Man denke dabei beispielsweise an das Versickern von Schadstoffen im Erdreich. Denkbare Szenarien hierfür sind etwa Unfälle von Tanklastzügen oder die Untersuchung der Eignung von Salzstöcken zur Endlagerung radioaktiver Abfälle. Es gibt aber auch vielfältige technische Anwendungen, beispielsweise Katalysatoren oder Kugelschüttungen. Der Boden besteht nun aus dem Feststoffskelett, der sog. *Bodenmatrix*, und aus dem *Porenraum*, der von einem oder mehreren Fluiden ausgefüllt wird (z. B. verschmutztes Bodenwasser und Bodenluft oder Wasser, Öl und Luft). Die Verunreinigung des Wassers äußert sich darin, dass das Wasser Schadstoffe trägt – aufgelöst oder als Partikel. Neben vielen anderen Phänomenen wie chemischen Reaktionen oder Adsorption an der Oberfläche der Bodenmatrix ist es vor allem die Ausbreitung der Schadstoffe, die hier von Interesse ist.

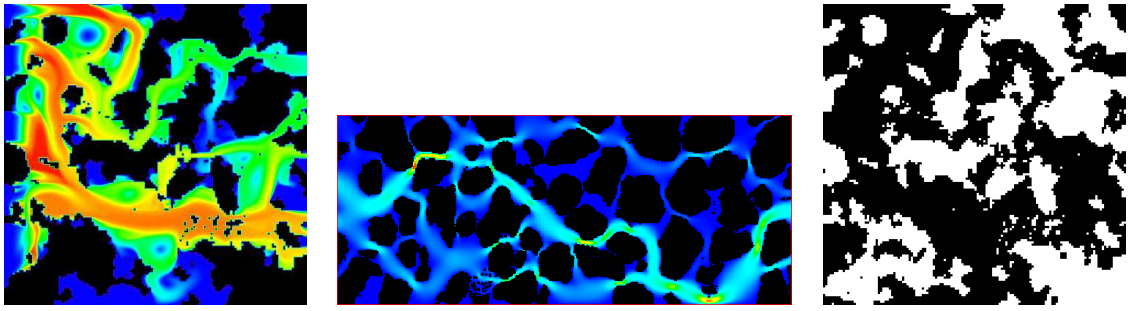


Abbildung 2.6: Poröse Medien groß (Schadstoffausbreitung im Gangesdelta, links), poröse Medien klein (Schadstoffe im Boden des Wattenmeers, Mitte) und poröse Medien winzig (poröse Struktur eines Biofilms (aggregierte Mikroorganismen), rechts

Es gibt nun (mindestens) zwei unterschiedliche Betrachtungsebenen:

- Auf der *mikroskopischen Ebene* oder *Mikroskala* wird die zerklüftete Geometrie des Porenraums vollständig aufgelöst, und man arbeitet mit den klassischen physikalischen Größen (Geschwindigkeit $\vec{u}$, Druck p, Konzentration C). Hier erhält man für die Strömung die bereits erwähnten *Navier-Stokes-Gleichungen*,

$$\begin{aligned} \frac{\partial}{\partial t}\vec{u} + (\vec{u}\cdot \mathrm{grad})\vec{u} + \mathrm{grad}p &= \frac{1}{Re}\Delta\vec{u} + \vec{g}\,, \\ \mathrm{div}\vec{u} &= 0\,, \end{aligned}$$

  hergeleitet aus den Erhaltungsgleichungen für Impuls und Masse, und für die Stoffkonzentration die *Transportgleichung*

$$\frac{\partial C}{\partial t} + \vec{u}\cdot \mathrm{grad}C = \lambda\Delta C + Q(t,x,y,\vec{u},C)\,.$$

Dabei bezeichnen $\vec{g}$ äußere Kräfte (Schwerkraft), Re die sog. *Reynoldszahl* und Q Quellterme (für Reaktionen etc.); für Details sei auf die Fallstudie zur numerischen Strömungsmechanik am Ende dieses Skripts verwiesen. Man beachte, dass die Dichte ρ in die Reynoldszahl einfließt und konzentrationsabhängig ist. Somit liegt ein gekoppeltes System von fünf partiellen Differentialgleichungen vor. Diese Betrachtungsebene ermöglicht ein sehr detailliertes Studium der Verhältnisse und eignet sich daher bestens, einzelne lokale Phänomene zu untersuchen (Abhängigkeit des Strömungsfelds von der Größe der Poren etc.). Allerdings führt sie aufgrund der komplizierten Geometrie zu extrem aufwendigen numerischen Berechnungen. Zudem ist man oft nicht an den Versickerungseigenschaften eines bestimmten Kubikmeters Boden interessiert, sondern vielmehr an den Eigenschaften so gearteter poröser Medien allgemein. Als „Eingabe“ nimmt man dann nicht eine konkrete Geometrie, sondern vielmehr einen mittels möglichst weniger Parameter spezifizierten Geometrietyp (z. B. Porösitätszahl, Charakteristik der Poren etc.). Auch auf der Ausgabeseite sind oftmals andere Variablen als die obigen gefragt, da man das lokale Geschwindigkeitsfeld oder die lokale Konzentration nicht oder unter unter großem Aufwand messtechnisch ermitteln. In der Regel liegen nur gemittelte Messgrößen vor, also etwa der Durchfluss durch ein kleines Teilwürfelchen der Geometrie.

- Auf der *makroskopischen Ebene* oder *Makroskala* wird deshalb mit gemittelten Größen gearbeitet, um eine Übereinstimmung zwischen Modellierungs- und Beobachtungsebene zu erhalten. Dieser Mittelungsprozess muss formal nachvollzogen werden. Als Kalkül wurde dafür auf Seiten der Naturwissenschaften die *Mittelungstechnik* entwickelt, von mathematischer Seite wurde das Verfahren der *Homogenisierung* beigesteuert.

Bei der Mittelung geht man von einem sogenannten *repräsentativen Elementarvolumen*[1] (REV) V aus, das klein im Vergleich zum Gesamtgebiet, jedoch hinreichend groß im Vergleich zu den Strukturen der Mikroskala (Porengröße, Kanaldurchmesser) ist. Anstelle der realen Geometrie betrachtet man jetzt nur noch dieses Muster. Der Zusammenhang des Porenraums wird dabei vorausgesetzt, es darf also keine Einschlüsse etc. geben. Eine in der fluiden Phase gegebene (mikroskopische) Größe u, also etwa eine Komponente des Geschwindigkeitsvektors, wird dann außerhalb des Porenraums (d. h. im Feststoffskelett) mit Null besetzt, und es wird ein sog. *Phasenmittel* berechnet:

$$\langle u \rangle \;:=\; \frac{1}{vol(V)} \cdot \int_V u \, \mathrm{d}V \,.$$

Nun werden die zugehörigen mikroskopischen Erhaltungsgleichungen über V gemittelt, um daraus Gleichungen herzuleiten, die ausschließlich in den gemittelten Größen leben. Dabei entstehen Anteile, die in den mikroskopischen Gleichungen nicht existieren (z. B. der hier nicht weiter behandelte Massenfluss durch mechanische Dispersion), für die ebenfalls Zusammenhänge postuliert werden müssen (etwa auf der Grundlage experimenteller Ergebnisse). Insgesamt ist wesentlich, dass die gemittelten Gleichungen nicht von der Wahl des REV abhängen (sonst sind die REV eben nicht repräsentativ).

Die Homogenisieriung ist der strengere und mathematischere Zugang, er erfordert aber noch einschränkendere Voraussetzungen. Grundlegend ist hier die Annahme einer zwar komplizierten, aber regelmäßigen Struktur. Genauer ist erforderlich, dass das betrachtete (makroskopische) Grundgebiet eine periodische Struktur aufweist. Für künstliche poröse Medien (Schwämme, Katalysatoren, Verbundwerkstoffe) ist das keine Einschränkung. Beim Boden kann es schon anders aussehen, wenn etwa verschiedene Bodenschichten das Grundgebiet schneiden. Gänzlich unzutreffend ist die Annahme der Periodizität bei stark heterogenen Medien. Trotzdem kann auch in solchen Fällen die Homogenisierung oft gewinnbringend eingesetzt werden. Man startet von einem Würfel C der Kantenlänge 1, der durch einen Faktor ε geeignet skaliert wird. Durch Reproduktion und Translation

[1] Die erste Schwierigkeit liegt hier oftmals schon in der Bestimmung solcher (funktional, nicht ästhetisch) repräsentativen Elementarvolumina – was ist schließlich repräsentativ?

des so entstandenen Würfels εC wird dann das Grundgebiet völlig ausgefüllt. In C werden nun eine offene Teilmenge M als *repräsentatives Korn*, $P := C \setminus \bar{M}$ als *repräsentatives Porenvolumen* und $\Gamma := \partial M$ als *repräsentative Matrixoberfläche* eingeführt. Daraus ergibt sich die gewünschte periodische Struktur für die Bodenmatrix

$$\Omega_{\varepsilon,M} \;:=\; \bigcup \left\{ \varepsilon \cdot (M + k) \subset \Omega,\, k \in \mathbb{Z}^3 \right\} ,$$

für den Porenraum

$$\Omega_{\varepsilon,P} \;:=\; \Omega \setminus \bar{\Omega}_{\varepsilon,M}$$

und für die Matrixoberfläche

$$\Gamma_\varepsilon \;:=\; \bigcup \left\{ \varepsilon \cdot (\Gamma + k) \subset \Omega,\, k \in \mathbb{Z}^d \right\} .$$

Auf dieser periodisierten Geometrie werden nun die (mikroskopischen) Erhaltungsgleichungen formuliert. Anschließend wird für die Lösungen sowie für die Gleichungen der Grenzübergang $\varepsilon \to 0$ untersucht. Alle in der Grenzgleichung auftretenden Koeffizienten sind dabei explizit definiert.

Im Falle des Stofftransports in durchströmten porösen Medien führen Mittelung und Homogenisierung auf neue Größen und sog. *effektive Parameter* (*Porosität, Permeabilität*). Als Gleichungen erhält man die *Darcy-Gleichung*

$$\tilde{q} \;=\; -\frac{k}{\mu} \mathrm{grad} \tilde{p}$$

mit dem gemittelten Druck $\tilde{p}$, dem *mittleren Massenfluss* $\tilde{q}$ und der *Permeabilität* k, bzw. – bei größeren Reynoldszahlen – die *Darcy-Forchheimer-Gleichung*.

An dieser Stelle wollen wir die Diskussion der mathematischen Modellierung beenden und uns der Problematik der Diskretisierung zuwenden. Die numerische Strömungsmechanik wird uns im letzten Kapitel nochmals begegnen.

Kapitel 3

Numerical Treatment of Mathematical Models

Since, in general, mathematical models cannot be solved in a direct, i. e. analytical way, their approximate *numerical* treatment on a computer is of crucial importance. Actually, this aspect is the heart of scientific computing, and most researchers working in the field of numerical simulation deal with the development of efficient numerical methods.

In the vast majority of mathematical models used for numerical simulations of problems from science and engineering, ordinary or partial differential equations play a predominant part. Therefore, their numerical treatment is in the centre of interest in our context and in this chapter. Nevertheless, there are other important mathematical tools for modelling, like

- integral equations,
- stochastic differential equations, or
- differential-algebraic equations,

for which numerical methods have to be developed, too.

In this text, we suppose that the reader is familiar with the basic ideas from numerical analysis (such as floating point numbers, rounding errors, condition, or stability) and, especially, with the meaning of the process of *discretization*, i. e. the restriction of continuous problems or quantities to suitable discrete and finite counterparts of these. Discretization concerns

- the used numbers: floating point numbers and floating point arithmetic instead of real numbers;
- the occurring functions: approximation of trigonometric functions with the help of truncated series of extremely fast convergence;
- the underlying domain (space and time): finite sets of grid points instead of a continuum, polygonal lines and triangular meshes instead of arbitrary curves and surfaces;
- the components of the differential equations: differences instead of derivatives;
- the dimension of the space of solutions: restriction to finite dimensional approximation spaces.

Hence, the step of discretization is the first crucial task during the numerical treatment of differential equations. After that, usually, methods from numerical linear algebra (eigenvalue computations, solution of linear systems, and so on) have to be applied to solve the generated discrete problems.

This chapter on numerical methods is organized as follows. After summarizing important basics from linear algebra in Sect. 3.1, in Sect. 3.2, we discuss some important topics and methods for the numerical solution of ordinary differential equations occurring in initial or boundary value problems. Sect. 3.3 is dedicated to discretization schemes for partial differential equations. Here, the focus is on finite difference, finite element, and finite volume methods. Furthermore, some aspects of the combination of discretization in both space and time are discussed. Afterwards, we turn to techniques for the direct (Sect. 3.4) or iterative (Sect. 3.5) solution of large systems of linear equations, which is the, perhaps, most crucial and most challenging task from numerical linear algebra. Since many relevant models are nonlinear, methods for the iterative solution of systems of nonlinear equations will be presented in Sect. 3.6. Finally, in Sect. 3.7, a short sortie to the fascinating world of generating two- and three-dimensional grids will conclude this chapter.

3.1 Back to the Roots: Linear Algebra

First of all, we start with a summary of some notations, definitions, and results from linear algebra which will occur at crucial points throughout this chapter. Since we suppose that the reader is familiar with the basic vocabulary of linear algebra, we restrict ourselves to the essentials.

- $\mathbb{R}^n$ is the n-dimensional vector space of all n-dimensional real vectors, i. e. of all n-tuples of real numbers.
- $\mathbb{R}^{m,n}$ is the mn-dimensional vector space of all linear maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ or all real m,n-matrices $A = (a_{ij})$.
- A *submatrix* of $A \in \mathbb{R}^{m,n}$ is obtained from A by deleting some rows and columns; if $m = n$ and if corresponding rows and columns are removed, the resulting submatrix is *principal*, and it is *leading principal*, if rows and columns are removed starting from the last ones.
- The *transpose* $A^T \in \mathbb{R}^{nmn}$ of $A \in \mathbb{R}^{m,n}$ is obtained by taking rows as columns and vice versa; if $A = A^T$, then A is called *symmetric*.
- Concerning regular patterns of nonzero entries, we distinguish *diagonal* ($a_{ij} = 0$ for $i \neq j$), *tridiagonal* ($a_{ij} = 0$ for $|i-j| > 1$), *lower triangular* ($a_{ij} = 0$ for $i < j$), and *upper triangular* ($a_{ij} = 0$ for $i > j$) matrices.
- If the a_{ij} are not numbers, but matrices themselves, we get the notion of *block matrices* (with the corresponding special cases as above). Note that the single blocks are denoted by A_{ij}.
- Vector addition is straightforward, vector products can be done in several ways, for example as an *inner* or *scalar* product $x^T y = \sum_{i=1}^{n} x_i y_i$ or as an *outer* product $xy^T := B \in \mathbb{R}^{n,n}$ with $b_{ij} = x_i y_j$.
- Matrix-vector products Ax with $A \in \mathbb{R}^{m,n}$ and $b \in \mathbb{R}^n$ result in a vector $y \in \mathbb{R}^m$ with inner products of A's rows with x as components; the definition of matrix-matrix products is straightforward.
- Two nonzero vectors $x, y \in \mathbb{R}^n$ are *orthogonal* if $x^T y = 0$ and *orthonormal* if, additionally, $x^T x = y^T y = 1$.
- $x^{(1)}, \ldots, x^{(k)} \in \mathbb{R}^n$ are *linearly independent*, if $\sum_{j=1}^{k} \alpha_j x^{(j)} = 0$ implies that all α_j must be zero; otherwise, the vectors are *linearly dependent*. A set of orthogonal vectors is always linearly independent.
- All *linear combinations* $\sum_{j=1}^{k} \alpha_j x^{(j)}$ of $x^{(1)}, \ldots, x^{(k)} \in \mathbb{R}^n$ span a subspace of $\mathbb{R}^n$ called $\mathrm{span}(x^{(1)}, \ldots, x^{(k)})$; its dimension is the maximum number of linearly independent vectors in it (the maximum length of a *basis*, i. e. a set of linearly independent vectors that span the subspace).

- The maximum number of linearly independent rows or columns of A is called its *rank*. If $A \in \mathbb{R}^{n,n}$ has maximum rank n, it is *invertible* or *nonsingular*, i. e. its *inverse* $A^{-1} \in \mathbb{R}^{n,n}$ with $A^{-1}A = AA^{-1} = I$ with the identity I exists.
- $A \in \mathbb{R}^{n,n}$ is called *orthogonal*, if $A^T A = AA^T = I$. Obviously, orthogonal matrices are nonsingular.
- $A \in \mathbb{R}^{n,n}$ is called *positive definite* (*positive semidefinite*), if $x^T Ax > 0$ ($x^T Ax \geq 0$) for all vectors $x \neq 0$. Positive definite matrices are nonsingular. If A is symmetric and positive definite, then each principal submatrix is so, too.
- *Eigenvalues* λ and *eigenvectors* $x \neq 0$ are solutions of $Ax = \lambda x$ for $A \in \mathbb{R}^{n,n}$. The eigenvalues are the roots of the *characteristic polynomial* $\det(A - \lambda I)$, where $\det(A)$ is the so-called *determinant* of A. Thus, there is a maximum of n (not necessarily distinct) real eigenvectors. The set of all eigenvalues is called the *spectrum* of A. A is nonsingular, if and only if all eigenvalues are nonzero. For each different eigenvalue λ_i of *multiplicity* k_i, there exists at least one eigenvector. The *eigenspace* of such a λ_i (the subspace formed by all eigenvectors to λ_i) has a maximum dimension of k_i.
- For some polynomial p and a matrix $A \in \mathbb{R}^{n,n}$ with eigenvalues $\lambda_1, \ldots, \lambda_n$, the eigenvalues of the matrix $p(A)$ are just the $p(\lambda_i)$. Each eigenvector of A is one of $p(A)$, too.
- For symmetric matrices, all eigenvalues are real, and there exists a basis of eigenvectors. If A is positive definite, then all eigenvalues are positive.
- A *similarity transform* BAB^{-1} does not change the spectrum of A. If A is symmetric or if A has n distinct eigenvalues, then there exists a B such that BAB^{-1} becomes diagonal. In the first case, the similarity matrix B can even be taken to be orthogonal. In many other situations, diagonality can not be obtained. Then, the most simple form is the so-called *Jordan canonical form*, which is 'nearly' diagonal (eigenvalues in the diagonal, and some 1's in the first superdiagonal).
- Another useful concept is the *singular value decomposition*. For each $A \in \mathbb{R}^{n,n}$, there exist orthogonal matrices U and V such that $A = UDV$ with a diagonal matrix D. The diagonal elements of D are all non-negative and called the *singular values* of A. Obviously, these are the square roots of the eigenvalues of $A^T A$. Note that the above definition of a singular value decomposition also extends to the case of rectangular matrices with $m \neq n$.
- A *norm* $\|\cdot\|$ in a vector space V maps V to $\mathbb{R}_+$ with $\|x\| \geq 0$ for all $x \in V$ and zero only if $x = 0$ (positivity), $\|\alpha x\| = |\alpha| \|x\|$ for each scalar α, and $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in V$ (triangle inequality).
- For the case of $V = \mathbb{R}^n$, there are several important *vector norms*: $\|x\|_2 := (x^T x)^{1/2}$ (the *Euclidean* or l_2 norm), $\|x\|_1 := \sum_{i=1}^n |x_i|$ (the l_1 norm), and $\|x\|_\infty := \max_{i=1}^n |x_i|$ (the l_∞ or *maximum* norm). Furthermore, for some symmetric positive definite $A \in \mathbb{R}^{n,n}$, $\|x\|_A := (x^T Ax)^{1/2}$ defines a vector norm, too (resulting in the Euclidean norm if A is the identity I).
- A *scalar* or *inner product* $(\cdot, \cdot)$ is a mapping $V \times V \to \mathbb{R}$ with $(x, x) \geq 0 \; \forall x \in V$ and zero only if $x = 0$, $(\alpha x, y) = \alpha(x, y)$ for each scalar α, $(x, y) = (y, x)$, and $(x + y, z) = (x, z) + (y, z) \; \forall x, y, z \in V$. Each inner product can be used to define a norm: $\|x\| := (x, x)^{1/2}$ (as it is done with the standard inner product $x^T y$ to define the Euclidean norm). Note that the concept of *orthogonality* can be defined with respect to each inner product.
- Since, in finite dimensional vector spaces, all norms are *equivalent* (i. e., they bound each other), some sequence $x^{(k)}$ converges to some limit vector x (i. e. $\|x^{(k)} - x\| \to 0$ for $k \to \infty$) with respect to all norms or none.
- For $V = \mathbb{R}^{n,n}$, *matrix norms* can be derived from the respective vector norms via $\|A\| := \max_{\|x\|=1} \|Ax\|$. In addition to the standard properties of a norm, we have the submulti-

plicativity: $\|AB\| \leq \|A\| \cdot \|B\|$ for all $A, B \in \mathbb{R}^{n,n}$. A short calculation shows that $\|A\|_1$ (corresponding to the l_1 vector norm) is just the maximum absolute value *column* sum of A and that $\|A\|_\infty$ (corresponding to the l_∞ vector norm) is just the maximum absolute value *row* sum of A.

- The Euclidean matrix norm is more complicated. We get $\|A\|_2 = (\rho(A^T A))^{1/2}$, where $\rho(B)$ is the *spectral radius* or maximum absolute value eigenvalue of B. For a symmetric A, $\|A\|_2 = \rho(A)$ holds. Furthermore, since $|\lambda|\|x\| = \|\lambda x\| = \|Ax\| \leq \|A\| \cdot \|x\|$ for each eigenpair λ, x and each matrix-vector norm pair $\|\cdot\|$, we get $|\lambda| \leq \|A\|$ for all eigenvalues of A and any matrix norm.
- A famous theorem for estimating the eigenvalues was given by *Gershgorin*. Define disks in the complex plane centered at the diagonal elements a_{ii} and with radius $r_i := \sum_{j \neq i} |a_{ij}|$ for $i = 1, \ldots, n$. Then all eigenvalues of A lie in the union of all n disks. Moreover, if the union of k disks is disjoint from the other $n - k$ disks, then this union contains exactly k eigenvalues (with multiplicities counted).
- A is called *(strictly) diagonally dominant* if $|a_{ii}| \geq r_i$ ($|a_{ii}| > r_i$) for all $i = 1, \ldots, n$. Each strictly diagonally dominant matrix is nonsingular. If we have only diagonal dominance, the matrix A is nonsingular if it is *irreducible* and if '$>$' holds for at least one row i. (A is irreducible if no permutation P can result in PAP^T being a 2×2 block upper triangular matrix with quadratic diagonal blocks.)

3.2 Discretizing Ordinary Differential Equations

In this section, we want to discuss methods for the numerical solution of problems based on *ordinary* differential equations (ODEs), i. e. usually equations where time is the only variable involved. Here, there are two basic classes of problems: *initial value problems (IVPs)*, where some properties of the solution are prescribed only on one side of the interesting (and possibly infinite) time interval, and *boundary value problems (BVPs)*, where we have conditions at both ends.

3.2.1 Initial Value Problems

Look at the equation

$$\dot{y}(t) \;=\; f(t, y(t))\,, \qquad y(a) \;=\; y_a\,, \qquad t \geq a\,, \tag{3.1}$$

or at the system

$$\dot{y}_i(t) \;=\; f_i(t, y_1(t), \ldots, y_n(t))\,, \qquad y_i(a) \;=\; y_{i,a}\,, \qquad t \geq a\,, \qquad i = 1, \ldots, n\,, \tag{3.2}$$

which are simple prototypes of an initial value problem. If f does not depend on its second argument y, then (3.1) becomes a simple integration or quadrature problem, and this is why we sometimes speak of the *integration* of ODEs instead of using the term *solution*. In (**??**) in Sect. 2.2.2, we have already seen an example from biology of a mathematical model leading to systems of ODEs, the dynamic development of populations (for example in a predator-prey situation). Other (more technical) examples are optimal control (of robots or a spacecraft) or processes involving chemical reactions.

First, let us deal with single equations. All standard approaches use *finite-difference* approximations for the derivatives, for example the difference quotient $(y(t + \delta t) - y(t))/\delta t$ for the first derivative $\dot{y}(t)$. With this approximation, (3.1) for $t = a$ reads

$$y(a + \delta t) \;\doteq\; y(a) + \delta t \cdot f(t, y(a)) \tag{3.3}$$

or, going on this way starting from a,

$$y_{k+1} := y_k + \delta t \cdot f(t_k, y_k)\,, \qquad t_k = a + k\delta t\,, \qquad k = 0, 1, \ldots\,, \tag{3.4}$$

which is called *Euler's method* and which is the simplest strategy to produce approximations y_k to $y(t_k)$ and, thus, to solve a first-order initial value problem. Note that (3.4) can also be derived via a truncated Taylor expansion of the exact solution $y(t)$ around t_k:

$$y(t) = y(t_k) + (t - t_k)\dot{y}(t_k) + R \doteq y(t_k) + (t - t_k)f(t_k, y(t_k))\,, \tag{3.5}$$

where $t := t_{k+1}$ leads to Euler's scheme.

Of course, we will make a discretization error that will probably become more important for increasing t. Therefore, this discretization error has to be studied carefully. For that, on some time interval $[a, b]$, $b = a + N\delta t$, we distinguish the *local discretization error*

$$l(\delta t) := \max_{a \le t \le b - \delta t} \{(y(t + \delta t) - y(t))/\delta t - f(t, y(t)\} \,, \tag{3.6}$$

which has nothing to do with the computed approximation, but indicates just the influence of the approximation to the derivatives, and the *global discretization error*

$$e(\delta t) := \max_{k=0,\ldots,N} \{|y_k - y(t_k)|\} \,, \tag{3.7}$$

which gives the maximum difference between the computed discrete approximations y_k and the exact solution's values at the respective points t_k. Obviously, the global discretization error is the crucial quantity concerning the quality of the overall method, whereas, usually, the local discretization error is the quantity that can be studied much easier. If $l(\delta t) \to 0$ for $\delta t \to 0$, the scheme is called *consistent*, if $e(\delta t) \to 0$ for $\delta t \to 0$, the scheme is called *convergent*. For Euler's method, Taylor expansion shows that $l(\delta t) = \mathcal{O}(\delta t)$ if $\ddot{y}$ is bounded. However, the more important point is the behaviour of $e(\delta t)$. Fortunately, if both y's second derivative and f's partial derivative with respect to its second argument y are bounded, it can be shown that Euler's method converges for $\delta t \to 0$ (which means, especially, that reducing the step size δt will improve the accuracy of our computed approximations) and that $e(\delta t) = \mathcal{O}(\delta t)$, too. Due to this property, Euler's method is called a *first-order discretization scheme*.

Of course, first-order convergence is quite slow. Thus, methods of higher order with $e(\delta t) = \mathcal{O}(\delta t^p)$ for some $p > 1$ are of great importance. The most straightforward way to derive such schemes is to start, again, from y's Taylor expansion and to take into account more terms of the series, for example

$$y(t_{k+1}) \doteq y(t_k) + \delta t \dot{y}(t_k) + \frac{\delta t^2}{2}\ddot{y}(t_k)\,, \tag{3.8}$$

where $\ddot{y}$ can be replaced by an expression in f, y, and $\dot{y}$ due to $\dot{y} = f$. However, such an approach needs derivatives of f and results in rather complicated formulas, especially for greater p. More promising is the strategy to use additional evaluations of f during each step in order to get an accuracy of higher order. The two most famous representatives of this family of discretization schemes are the method of *Heun*,

$$y_{k+1} := y_k + \frac{\delta t}{2}\big(f(t_k, y_k) + f(t_{k+1}, y_k + \delta t f(t_k, y_k))\big)\,, \tag{3.9}$$

which is of second order ($e(\delta t) = \mathcal{O}(\delta t^2)$), and the fourth-order *Runge Kutta* method

$$y_{k+1} := y_k + \frac{\delta t}{6}\left(T_1 + 2T_2 + 2T_3 + T_4\right)\,, \tag{3.10}$$

where

$$\begin{aligned} T_1 &:= f(t_k, y_k)\,, \\ T_2 &:= f\left(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} T_1\right)\,, \\ T_3 &:= f\left(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} T_2\right)\,, \\ T_4 &:= f(t_{k+1}, y_k + \delta t T_3)\,. \end{aligned}$$

Still higher order is possible, but not very widespread, as the number of evaluations of f (which may be quite expensive if f is not given in closed form) gets too large. For those readers that are familiar with numerical quadrature, it is intereting to note that the above schemes of Euler, Heun, and Runge-Kutta correspond to composite quadrature rules (rectangle, trapezoid, and Simpson, resp.).

A third possibility to construct schemes of higher order is to profit from former results $y_{k-i}, i > 1$, when computing y_{k+1}. This avoids additional evaluations of f in intermediate points. One of the simplest representatives of these so-called *multistep methods* is the *second-order Adams-Bashforth* scheme:

$$y_{k+1} := y_k + \frac{\delta t}{2}\big(3f(t_k, y_k) - f(t_{k-1}, y_{k-1})\big)\,. \tag{3.11}$$

For the general case of order $p \geq 1$, p previous approximations have to be taken into account. For example, think of a polynomial P interpolating $f_i := f(t_i, y_i)$ in t_i, $i = k, k-1, \ldots, k-p+1$, and start from

$$y_{k+1} := y_k + \int_{t_k}^{t_{k+1}} \dot{y}(t)\mathrm{d}t = y_k + \int_{t_k}^{t_{k+1}} f(t, y(t))\mathrm{d}t \doteq y_k + \int_{t_k}^{t_{k+1}} P(t)\mathrm{d}t\,. \tag{3.12}$$

A short calculation shows that $p = 1$ results in Euler's method, whereas $p = 2$ leads to (3.11), the second-order Adams-Bashforth scheme. In principle, schemes of arbitrary order can be constructed in this way. In practice, however, methods with $p > 4$ are not very popular. Note that the problem of missing information at the beginning (no 'previous points' are available when computing y_1, e. g.) can be circumvented by using a single-step method of the same order for the first time steps and turning to the multistep one as soon as enough f_i are available.

All of the above schemes are *explicit* ones: The new value y_{k+1} can be computed directly based on the data acquainted so far. The drawback of this comfortable approach is that, often, the time step δt has to be chosen very small in order to avoid instabilities. A remedy for that offer *implicit* methods which are characterized by an occurrence of (the unknown) y_{k+1} on the right-hand side of y_{k+1}'s defining equation, too. Hence, computing y_{k+1} becomes more complicated. The most important class of implicit methods are the *Adams-Moulton* schemes. Here, interpolants P are constructed as above, but now interpolating t_{k+1}, f_{k+1}, too. Two representatives are the second-order method

$$y_{k+1} := y_k + \frac{\delta t}{2}\big(f_k + f_{k+1}\big) \tag{3.13}$$

and the fourth-order scheme

$$y_{k+1} := y_k + \frac{\delta t}{24}\big(f_{k-2} - 5f_{k-1} + 19f_k + 9f_{k+1}\big)\,. \tag{3.14}$$

Now, how can we solve the above equations in order to obtain y_{k+1}? One possibility is to solve the (generally nonlinear) equation with an appropriate method like Newton's method, e. g. A more elegant and efficient approach, however, is the so-called *predictor-corrector* technique. Here, an explicit method is used to construct a preliminary or 'predicted' value for y_{k+1} and, thus, for

f_{k+1}, which is afterwards used in an implicit formula to compute the final y_{k+1}. Note that the resulting scheme is entirely explicit, but has the properties of an implicit method.

All that has been said so far for the numerical solution of a single first-order ODE can be generalized to systems of such ODEs. Since higher-order ODEs (ODEs with higher derivatives) can be reduced to systems of first-order ODEs (see the discussion at the end of this section), the above methods cover the whole field of initial value problems.

As we know from the basics of numerical analysis, ill-conditioned problems with small changes in the input data leading to huge differences in the results are hard to deal with. This holds especially for the solution of differential equations. Let us consider the small example $\ddot{y} - N\dot{y} - (N+1)y = 0$ with initial conditions $y(0) = 1, \dot{y}(0) = -1$. Obviously, $y(t) = e^{-t}$ is the desired solution. If we use $y_\varepsilon(0) = 1 + \varepsilon$ instead, we get the solution $y_\varepsilon(t) = (1 + \frac{N+1}{N+2}\varepsilon)e^{-t} + \frac{\varepsilon}{N+2}e^{(N+1)t}$, which is of a completely different characteristics (for arbitrary $\varepsilon > 0$, it tends towards infinity for t increasing!). Hence, the numerical solution of such an IVP causes a lot of problems, since disturbances like the above one (due to rounding or discretization errors) can not be avoided in a numerical solution process.

But even if the *problem* is well-conditioned, the *algorithm* may be *unstable* and cause problems. For that, consider the *midpoint rule*

$$y_{k+1} \;:=\; y_{k-1} + 2\delta t f_k\,, \tag{3.15}$$

an explicit two-step scheme of second order. Now, we apply this scheme to the well-conditioned IVP $\dot{y} = -2y+1$, $y(0) = 1$, with solution $y(x) = \frac{1}{2}(e^{-2x}+1)$ (with the initial condition $y(0) = 1+\varepsilon$ instead, the change in the solution is only εe^{-2x}). Then, we get

$$y_{k+1} \;=\; y_{k-1} + 2\delta t(-2y_k + 1) \;=\; y_{k-1} - 4\delta t y_k + 2\delta t\,, \qquad y_0 = 1\,. \tag{3.16}$$

Since we have a two-step method, we need some y_1, too, and we choose $y(\delta t)$, the exact solution's value at δt, for that. For any $\delta t > 0$, a short calculation shows that the sequence of the y_k oscillates with absolute values tending towards infinity for $k \to \infty$, instead of approaching y's limit $1/2$.

δt	y_9	y_{10}	y_{79}	y_{80}	y_{999}	y_{1000}
1.0	-4945.9	20953.9				
0.1	0.5820	0.5704	-1725.3	2105.7		
0.01	0.9176	0.9094	0.6030	0.6010	-154.6	158.7

Thus, although the midpoint rule is of second order, we observe a highly unstable behaviour. How can this happen? The point is that the midpoint rule is *consistent* of second order, but, obviously, not *convergent*. While for single-step methods, both terms can be shown to be equivalent, consistent multistep methods need an additional *stability condition* to avoid instabilities and to ensure convergence.

For the discussion of such a stability condition, we write linear s-step methods in their general form

$$y_{k+1} \;=\; \sum_{i=1}^{s} \alpha_i y_{k+1-i} \;+\; \delta t \sum_{i=0}^{s} \beta_i f_{k+1-i}\,. \tag{3.17}$$

If $\beta_0 = 0$, the method is explicit, otherwise it is implicit. Associated with the above scheme is the polynomial

$$P(\zeta) \;:=\; \zeta^s - \sum_{i=1}^{s} \alpha_i \zeta^{s-i}\,. \tag{3.18}$$

Then, the method (3.17) is *stable* if all complex roots of $P(\zeta)$ are located on the boundary of or within the unit disk, and if all roots with $|\zeta| = 1$ are simple. The method is *strongly stable* if, additionally, there is at most one root with $|\zeta| = 1$. One can show that all s-step Adams-Bashforth and Adams-Moulton ($s > 1$) methods are strongly stable, as well as all explicit 1-step methods. However, the midpoint rule introduced in (3.15) has $P(\zeta) = \zeta^2 - 1$ and, hence, is only stable. For strongly stable methods, the terms consistency and convergence are equivalent (including the order p). If the interval of interest is finite, then this holds for stable methods, too. The example of the midpoint rule shows what can happen if strong stability on an infinite interval is not fulfilled.

The last phenomenon we want to discuss in the context of IVPs is *stiffness*. For that, consider the IVP

$$\dot{y} \;=\; -1000y + 1000\,, \qquad y(0) = y_0 = 2\,, \tag{3.19}$$

with the solution $y(t) = e^{-1000t} + 1$. It is easy to show that this problem is well-conditioned. Euler's method results in

$$y_{k+1} \;=\; y_k + \delta t(-1000y_k + 1000) \;=\; (1 - 1000\delta t)y_k + 1000\delta t\,, \tag{3.20}$$

or, by complete induction,

$$y_{k+1} \;=\; (1 - 1000\delta t)^{k+1} + 1\,. \tag{3.21}$$

Although $y(t)$ decreases very quickly to its limit 1 and although the Euler method is strongly stable, the above sequence will show oscillations and will diverge if $\delta t > 0.002$. This shows that the notion of stability, actually, is an asymptotic one, describing the behaviour in the limit case $\delta t \to 0$, but saying nothing for some specific time step δt. Of course, we understand that a very small step size is necessary at the beginning when $y(t)$ changes rapidly, but afterwards, there seems to be no reason for such a small time step. This is what is called *stiffness*: The solution contains a rather unimportant component (here the exponential term), but that requires to be approximated in a very accurate way. That is, stiff equations force us to use very small time steps even if the solution does hardly vary. Here, the remedy are implicit methods. If we use the *implicit* or *backward* Euler method instead,

$$y_{k+1} \;=\; y_k + \delta t f(t_{k+1}, y_{k+1})\,, \tag{3.22}$$

we end up with

$$y_{k+1} \;=\; \frac{y_k + 1000\delta t}{1 + 1000\delta t} \;=\; \frac{1}{(1 + 1000\delta t)^{k+1}} + 1\,, \tag{3.23}$$

which clearly shows that there are no problems of oscillations or divergence at all. There is a very simple explanation for this miracle: Explicit methods try to approximate the solution with polynomials, whereas implicit schemes are based upon rational functions – and no polynomial can approximate e^{-t} for $t \to \infty$, whereas rational functions do.

Usually, especially with systems of equations, implicit methods require the iterative solution of the resulting nonlinear equation(s), which can lead to a significant amount of computing time. Nevertheless, implicit schemes are the suitable choice for stiff equations.

Finally, *extrapolation methods* shall not be forgotten. Here, starting from a series expansion of the global discretization error for $\delta t = (a - b)/n$ and $n = 2, 4, 8, 16, \ldots$ or $n = 4, 6, 8, 12, \ldots$ (*Bulirsch sequence*), error terms of lower order can be cancelled by suitable linear combinations, and one gets a significant gain in the order of accuracy.

In this section, we have treated IVPs with first derivatives, only. In the higher (say n-th) order case, the ODE is usually converted to a system of n first-order equations by introducing the derivatives of $y(t)$ as additional variable functions. For example, the ODE

$$y^{(n)}(t) \;=\; f(t, y(t), \ldots, y^{(n-1)}(t))\,, \tag{3.24}$$

with

$$y_k(t) \;:=\; y^{(k)}(t)\,, \qquad k = 0, \ldots, n-1\,, \tag{3.25}$$

can be written as first-order system consisting of

$$y'_{n-1}(t) \;=\; f(t, y_0(t), \ldots, y_{n-1}(t)) \tag{3.26}$$

and

$$y'_k(t) \;=\; y_{k+1}(t)\,, \qquad k = 0, \ldots, n-2\,. \tag{3.27}$$

3.2.2 Boundary Value Problems

Next, we deal with ODEs with constraints at more than one point, typically on both sides of the interval of interest. While initial conditions are quite reasonable for population dynamics, for example, such *boundary conditions* occur often in technical applications (a rocket, e. g., usually has both a defined starting point and a defined target point). Since a first-order ODE allows only one constraint to determine the solution uniquely, we consider the following simple example:

$$\ddot{y}(t) \;=\; f(t, y(t), \dot{y}(t))\,, \qquad a \le t \le b\,, \qquad y(a) = y_a\,, \quad y(b) = y_b\,, \tag{3.28}$$

or, for some linear f,

$$\ddot{y}(t) \;=\; a(t)\dot{y}(t) + b(t)y(t) + c(t) \tag{3.29}$$

with the same boundary conditions. This is called a *two-point boundary value problem*. We start with the linear case and $a(t) \equiv 0$. In this case, $b(t) \ge 0$ on $[a, b]$ is a sufficient condition for the BVP to have a unique solution. We define $\delta t := (a-b)/n$ for some $n \in \mathbb{N}$, $t_0 := a, t_n := b$, and $t_i := a + i\delta t$ for $i = 0, \ldots, n$. That is, we introduce a discrete grid on $[a, b]$ with $n+1$ equidistant grid points. For $\ddot{y}$, we, again, use a finite difference approximation:

$$\ddot{y}(t) \;\doteq\; \frac{\frac{y(t+\delta t)-y(t)}{\delta t} - \frac{y(t)-y(t-\delta t)}{\delta t}}{\delta t} \;=\; \frac{y(t+\delta t) - 2y(t) + y(t-\delta t)}{\delta t^2}\,. \tag{3.30}$$

Now, with the abbreviations $y_i := y(t_i)$ and so on, the following equations, together with the boundary conditions $y_0 := y_a$ and $y_n := y_b$, are a discrete analogue to (3.29) with vanishing $a(t)$:

$$\frac{1}{\delta t^2}(y_{i+1} - 2y_i + y_{i-1}) - b_i y_i \;=\; c_i\,, \qquad i = 1, \ldots, n-1\,. \tag{3.31}$$

This system of linear equations can be written in matrix form:

$$\begin{pmatrix} 2 + b_1\delta t^2 & -1 & & & \\ -1 & 2 + b_2\delta t^2 & \ddots & & \\ & \ddots & \ddots & & \\ & & & -1 & \\ & & & -1 & 2 + b_{n-1}\delta t^2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} -\delta t^2 c_1 + y_a \\ -\delta t^2 c_2 \\ \vdots \\ -\delta t^2 c_{n-2} \\ -\delta t^2 c_{n-1} + y_b \end{pmatrix}. \tag{3.32}$$

Note that it can be shown that the matrix is positive definite and, thus, nonsingular. Hence, there is a unique solution of the system.

Concerning the error of our discrete approximations $y_1, \ldots, y_{n-1}$, we distinguish again the *local* discretization error

$$\begin{aligned} L(t, \delta t) \;&:=\; (y(t+\delta t) - 2y(t) + y(t-\delta t))/\delta t^2 - b(t)y(t) - c(t) \\ &=\; (y(t+\delta t) - 2y(t) + y(t-\delta t))/\delta t^2 - \ddot{y}(t) \end{aligned} \tag{3.33}$$

and the *global* discretization error

$$e(\delta t) \;:=\; \max_{1\le i\le n-1}\{|y_i - y(t_i)|\}\,. \tag{3.34}$$

As for IVPs, Taylor expansion (now of both $y(t+\delta t)$ and $y(t-\delta t)$) provides the consistency result, if $y(t)$ is sufficiently smooth:

$$L(t,\delta t) \;=\; \mathcal{O}(\delta t^2)\,. \tag{3.35}$$

Evaluating (3.33) in the grid points, subtracting (3.31) from it, and denoting by e and l the vectors of the global and local discretization error, resp., we get

$$Ae \;=\; -\,\delta t^2 l\,, \tag{3.36}$$

where A is the coefficient matrix from (3.32). Equation (3.36) gives the basic relationship between the global and the local discretization error. At that point, the properties of A are crucial for getting any statement concerning the behaviour of the global discretization error for $\delta t \to 0$. It can be shown that for our case of $b(t) \ge 0$, the global discretization error is of second order, too.

Now, we skip our above restriction $a(t) \equiv 0$ and allow the occurrence of first derivatives $\dot y(t)$ in (3.29). A widespread finite difference approximation for $\dot y(t)$ is the second-order *centered difference*

$$\dot y(t) \;\doteq\; \frac{y(t+\delta t) - y(t-\delta t)}{2\delta t}\,. \tag{3.37}$$

Evaluating these differences at our grid points, we get instead of (3.31):

$$\left(-1 - \frac{a_i\delta t}{2}\right) y_{i-1} + \left(2 + b_i\delta t^2\right) y_i + \left(-1 + \frac{a_i\delta t}{2}\right) y_{i+1} \;=\; -\,\delta t^2 c_i\,, \quad i = 1,\ldots,n-1\,. \tag{3.38}$$

This leads to the following matrix representation:

$$\begin{pmatrix} 2+b_1\delta t^2 & -1+p_1 & & \\ -1-p_2 & 2+b_2\delta t^2 & \ddots & \\ & \ddots & \ddots & -1+p_{n-2} \\ & & -1-p_{n-1} & 2+b_{n-1}\delta t^2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} -\delta t^2 c_1 + (1+p_1)y_a \\ -\delta t^2 c_2 \\ \vdots \\ -\delta t^2 c_{n-2} \\ -\delta t^2 c_{n-1} + (1-p_{n-1})y_b \end{pmatrix}, \tag{3.39}$$

where $p_i = a_i\delta t/2$. What about positive definiteness? We, again, assume that all $b_i \ge 0$. In order to get a diagonally dominant matrix, we need

$$|2 + b_i\delta t^2| \;\ge\; |1 + \frac{a_i\delta t}{2}| + |1 - \frac{a_i\delta t}{2}|\,, \qquad i = 1,\ldots,n-1\,. \tag{3.40}$$

This holds, especially, if $|a_i\delta t| \le 2$ for $i = 1,\ldots,n-1$, which is a rather hard restriction to the time step and, actually, enforces very small δt if the first derivates are dominant in the ODE (large a_i). We can avoid this, if we use a *one-sided* or *upwind* discretization for $\dot y(t)$ instead of (3.37):

$$\dot y(t) \;\doteq\; \frac{1}{\delta t}\cdot \begin{cases} (y_{i+1} - y_i) & \text{for} \quad a_i < 0\,, \\ (y_i - y_{i-1}) & \text{for} \quad a_i \ge 0\,, \end{cases} \tag{3.41}$$

where the direction of the one-sided difference depends on the sign of a_i. In this case, it is easy to verify that diagonal dominance holds (if $b_i \ge 0$, as assumed). Note that, here, it can be shown that diagonal dominance already ensures nonsingularity. The drawback of the one-sided approach is its reduced accuracy: the upwind scheme is of first order, only.

Up to now, we have studied boundary conditions that fix the solution's value at some given point(s). However, these so-called *Dirichlet* boundary conditions are not the only possibility to impose boundary conditions to an ODE. *Neumann* boundary conditions specify the (first) derivative of the solution:

$$\dot{y}(a) \;=\; \dot{y}_a\,, \qquad \dot{y}(b) \;=\; \dot{y}_b\,. \tag{3.42}$$

Hence, $y_0 = y(a)$ and $y_n = y(b)$ are no longer known values, but have to be computed. Compared to (3.32), the matrix becomes bigger, with rows 1 and $n-1$ now corresponding to rows $2, \ldots, n-2$, and with two additional rows 0 and n for the (new) unknowns y_0 and y_n. The latter can be derived with the help of some virtual grid points t_{-1} and t_{n+1} (with virtual function values y_{-1} and y_{n+1}, respectively) outside the interval $[a, b]$. Starting from the centered difference approximation

$$\dot{y}_a \;=\; \dot{y}(a) \;\doteq\; \frac{y_1 - y_{-1}}{2\delta t}\,, \tag{3.43}$$

we get

$$\ddot{y}(a) \;\doteq\; \frac{y_{-1} - 2y_0 + y_1}{\delta t^2} \;=\; \frac{2y_1 - 2y_0 - 2\dot{y}_a\delta_t}{\delta t^2}\,. \tag{3.44}$$

In an analogous way, we get a corresponding equation in b. Note that, of course, a mixture of Dirichlet and Neumann boundary conditions is possible, too. Hence, for the pure Neumann case, the resulting system of equations reads

$$\begin{pmatrix} 2+b_0\delta t^2 & -2 & & \\ -1 & 2+b_1\delta t^2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -2 & 2+b_n\delta t^2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} \;=\; \begin{pmatrix} -\delta t^2 c_0 - 2\dot{y}_a\delta t \\ -\delta t^2 c_1 \\ \vdots \\ -\delta t^2 c_{n-1} \\ -\delta t^2 c_n + 2\dot{y}_b\delta t \end{pmatrix}. \tag{3.45}$$

This coefficient matrix is no longer symmetric (but can be symmetrized), and it is nonsingular if $b_i > 0$ for all $i = 0, \ldots, n$. If, however, all b_i are zero, the matrix is singular (all row sums vanish). This is reasonable, since $b(t) \equiv 0$ in the ODE (3.29) means that if we add a constant to the solution $y(t)$ of our BVP (3.29) and (3.42), then this new function solves the BVP, too. That is, in this situation, the BVP itself doesn't have a unique solution.

Another type of relevant boundary conditions are *periodic* ones:

$$y_0 = y(a) \;=\; y(b) = y_n\,. \tag{3.46}$$

As before, y_0 and y_n are both unknowns. However, because of (3.46), only one of them needs to be taken into account. Thus, compared to (3.32), we need one further equation, which can be derived in a similar way to the above one:

$$\ddot{y}(a) \;\doteq\; \frac{y_{-1} - 2y_0 + y_1}{\delta t^2} \;=\; \frac{y_{n-1} - 2y_0 + y_1}{\delta t^2}\,. \tag{3.47}$$

Finally, in the equation for y_{n-1}, y_n is replaced by y_0, and we get the system

$$\begin{pmatrix} 2+b_0\delta t^2 & -1 & & -1 \\ -1 & 2+b_1\delta t^2 & \ddots & \\ & \ddots & \ddots & -1 \\ -1 & & -1 & 2+b_{n-1}\delta t^2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix} \;=\; \begin{pmatrix} -\delta t^2 c_0 \\ -\delta t^2 c_1 \\ \vdots \\ -\delta t^2 c_{n-2} \\ -\delta t^2 c_{n-1} \end{pmatrix}. \tag{3.48}$$

As before, we loose uniqueness in the case of $b(t) \equiv 0$ (adding a constant to a solution leads to another solution).

The above finite difference approach can be extended to the general situation of a nonlinear f in (3.28) by discretizing the derivatives as before and by, afterwards, applying Newton's scheme, for example (see Sect. 3.6). Another very successful and quite different approach is the so-called *shooting* method. Here, instead of directly tackling (3.28), the corresponding IVP

$$\ddot{y}(t) \;=\; f(t, y(t), \dot{y}(t))\,, \qquad y(a) = y_a\,, \quad \dot{y}(a) = s\,, \tag{3.49}$$

is solved. Hence, the task is to find the parameter $\bar{s}$ and the corresponding IVP-solution $y(t;\bar{s})$ that fulfil

$$y(b;\bar{s}) \;=\; y_b\,. \tag{3.50}$$

To some extent, the shooting approach regards a BVP as an inverse problem: to a given result, the correct input parameters are looked for. In fact, the shooting apmethod requires an iterative process, each step of which entails the solution of an IVP. This process can be realized in a systematic way by searching for a root of $y(b;\bar{s}) - y_b$ with standard techniques (bisection, Newton, etc.).

3.3 Discretizing Partial Differential Equations

ODEs involve only one indpendent variable, which usually is time. If we have problems where both time and space are to be taken into account, or if we want to treat stationary problems in more than one spatial variable x, y, z, we need *partial* derivatives and end up with *partial* differential equations (PDEs). Again, we have to deal with two basic classes of problems: *boundary value problems*, where the situation at the boundary of the respective domain is prescribed, and *initial-boundary problems*, where we have both initial values (with respect to time) and boundary conditions (with respect to space). As before, both initial and boundary conditions shall guarantee the uniqueness of the solution.

A simple and important class of PDEs is the *general linear PDE of second order in d dimensions*:

$$\sum_{i,j=1}^{d} a_{ij}(\mathbf{x}) u_{x_i x_j}(\mathbf{x}) \;+\; \sum_{i=1}^{d} a_i(\mathbf{x}) u_{x_i}(\mathbf{x}) \;+\; a(\mathbf{x})u(\mathbf{x}) \;=\; f(\mathbf{x})\,. \tag{3.51}$$

Here, $\mathbf{x} := (x_1, \ldots, x_d)$ is the vector of variables x_j, which may denote space coordinates or time. Three subtypes can be distinguished:

elliptic in $\mathbf{x}$:	the (w. l. of. g. symmetric) matrix $A(\mathbf{x}) = \big(a_{ij}(\mathbf{x})\big)_{i,j}$ is positive definite or negative definite;
hyperbolic in $\mathbf{x}$:	A has one positive and $n-1$ negative Eigenvalues or vice versa;
parabolic in $\mathbf{x}$“:	one Eigenvalue of A is xero, $n-1$ Eigenvalues have equal sign, and $\operatorname{rank}(A(\mathbf{x}), c(\mathbf{x})) = n$ with $c(\mathbf{x}) = (a_i(\mathbf{x}))_i$.

According to the above classification a PDE is called elliptic, hyperbolic, or parabolic, if the respective relations hold for all $\mathbf{x} \in \Omega$. In Ch. 2, we have already seen some of the simplest representatives of these three classes:

elliptic	potential or Laplace equation	$\Delta u = u_{xx} + u_{yy} = 0$
hyperbolic	wave equation	$u_{xx} - u_{yy} = 0$
parabolic	heat equation	$u_{xx} - u_t = 0$

All three types are characterized by strongly different properties and need different techniques for their numerical treatment. Thus, in the following, we will restrict ourselves to the case of PDEs

that are elliptic with respect to space and (if time-dependent at all) parabolic with respect to time. In spite of this restriction, we will only cover a small part of the wide field of the numerical treatment of PDEs.

In the next two sections, we discuss two of the most important discretization schemes for PDEs, finite difference and finite element methods. Finally, some other approaches like finite volumes and spectral elements will be mentioned.

3.3.1 Discretizing Space

Finite Differences (FD)

Hier wird das betrachtete Gebiet Ω mit einem i. A. regelmäßigen Gitter mit (evtl. verschiedenen) Maschenweiten h_x, h_y und h_z überzogen. Der Standardfall sind (in jeder Richtung) äquidistante Gitter, aber auch adaptive Verfeinerung ist möglich (i. A. blockweise), wobei man sich dann aber für die sog. hängenden Knoten (Gitterpunkte ohne Nachbarn auf einer Seite) etwas überlegen muss. Die Ableitungen (Differentialquotienten) werden unmittelbar durch entsprechende Differenzenquotienten approximiert:

$$\frac{\partial u}{\partial x}(\xi) \doteq \frac{u(\xi+h)-u(\xi)}{h}, \frac{u(\xi+h)-u(\xi-h)}{2h}, \frac{u(\xi)-u(\xi-h)}{h}$$

(Vorwärtsdifferenz, zentrale Differenz, Rückwärtsdifferenz)

$$\frac{\partial^2 u}{\partial x^2}(\xi) \doteq \frac{u(\xi+h)-2u(\xi)+u(\xi-h)}{h^2}$$

...

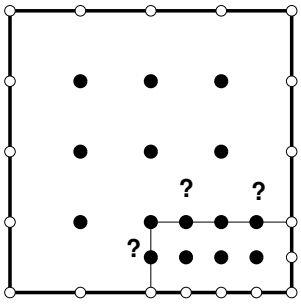


Abbildung 3.1: Innere Punkte, Randpunkte und hängende Knoten in 2 D

In jedem Gitterpunkt, der auch einen Freiheitsgrad darstellt, wird nun die entsprechende Differenzengleichung aufgestellt, d. h. die FD-diskretisierte Differentialgleichung. Bei Dirichlet-Randwerten gibt es also eine diskrete Gleichung für jeden inneren Punkt, die Randwerte wandern als vorgegebene Größen auf die rechte Seite der jeweiligen diskreten Gleichung. Bei Neumann-Randbedingungen sind auch die Werte in den Randpunkten Unbekannte, und es sind auch in diesen entsprechende Differenzengleichungen anzugeben. Hierbei wird i. A. auf (virtuelle) Werte außerhalb des Gebiets zugegriffen, die durch Berücksichtigung der Randbedingung anschließend wieder eliminiert werden müssen (siehe das nachfolgende Beispiel). In beiden Fällen erhält man schließlich ein System von M linearen Gleichungen in M Unbekannten.

Ein einfaches Beispiel: Zu lösen sei die zweidimensionale Laplace-Gleichung $-\Delta u = 0$ auf dem Einheitsquadrat $\Omega = [0,1]^2$. Die Schreibweise mit dem Minuszeichen wird i. d. R. gewählt, um in der Matrix positive Diagonalelemente zu erhalten. Hierzu soll ein quadratisches und äquidistantes

$(N+1) \times (N+1)$ Gitter verwendet werden, also mit $h_x = h_y = 1/N$. Im Falle von Dirichlet-Randbedingungen gibt es somit $M := (N-1)^2$ Unbekannte (die inneren Punkte), im Falle von Neumann-Randbedingungen gilt $M := (N+1)^2$. Für die Matrix A gilt $A \in \mathbb{R}^{M \times M}$, für die rechte Seite b gilt $b \in \mathbb{R}^M$, und für den Lösungsvektor x, der die zu bestimmenden Werte von u an den jeweiligen Gitterpunkten enthält, gilt ebenfalls $x \in \mathbb{R}^M$. Verwendet man obige diskrete Approximation für die zweiten Ableitungen, dann ist A *dünn besetzt*: Unabhängig von M gibt es in jeder Matrxizeile höchstens fünf von Null verschiedene Einträge. Bei passender Anordnung der Gitterpunkte hat A zudem Bandstruktur. Im Dirichlet-Fall sind alle Diagonalelemente 4, und es gibt zwei, drei oder vier Stellen mit dem Eintrag -1. Weniger als vier -1-Einträge gibt es in den Zeilen zu Nachbarpunkten von Randpunkten, da die festen Randwerte nicht als Unbekannte behandelt werden, sondern in die rechte Seite b des Gleichungssystems wandern, siehe Abb. 3.2.

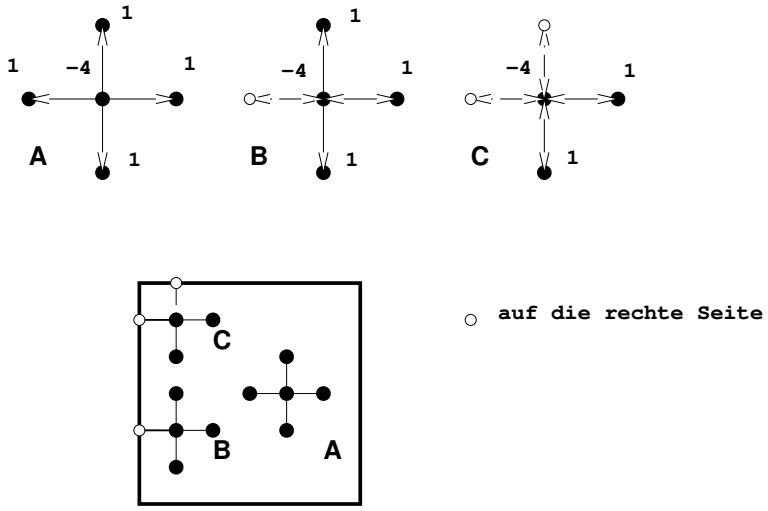


Abbildung 3.2: Unterschiedliche „Differenzensterne“ in 2 D (Dirichlet)

Im Neumann-Fall steht in der Diagonale der Zeilen zu inneren Punkten der Wert 4, die Diagonalelemente der Zeilen zu Randpunkten haben den Wert 2 (Ecken) bzw. 3 (sonstige Randpunkte). Die Anzahl der -1-Einträge ist ebenfalls zwei, drei oder vier. Die modifizierten Diagonalelemente auf dem Rand kommen daher, dass hier (virtuelle) Punkte außerhalb des Gebiets zur Vervollständigung des Sterns herangezogen werden, die aber gemeinsam mit einem entsprechenden Anteil des Diagonalelements durch den vorgegebenen (und auf die rechte Seite wandernden) Wert der Normalableitung ersetzt werden, siehe auch Abb. 3.3.

Die Güte der gewählten Diskretisierung spiegelt sich in ihrer Konvergenzordnung wider, d. h. in der Größenordnung des Fehlers der numerisch berechneten Lösung u_h im Vergleich zur exakten Lösung u des gegebenen Problems. Typischerweise sind die Fehler (gemessen in einer geeigneten Norm) von der Ordnung $\|u - u_h\| = O(h^k)$ ($k = 2$ im Falle der obigen Differenzenquotienten für die zweite Ableitung), wenn h die (jetzt in alle Richtungen als gleich angenommene) Maschenweite bezeichnet. Man kann das Ergebnis somit auf zwei Arten verbessern: durch Verringerung der Maschenweite (teuer, da die Maschenweite h bei einem d-dimensionalen Problem mit $O(h^{-d})$ Punkten einhergeht) oder durch Erhöhung der Ordnung k des Verfahrens durch den Übergang zu besseren diskreten Approximationen der Ableitungsterme. Letzteres kann man durch „Verbreiterung“ der Differenzensterne, d. h. durch Involvierung einer größeren Anzahl umliegender Knoten erreichen (was natürlich zu einer stärkeren Füllung der Matrix A führt).

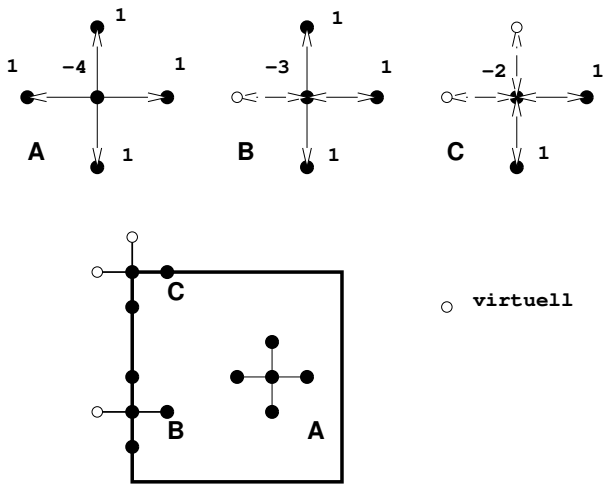


Abbildung 3.3: Unterschiedliche „Differenzensterne“ in 2 D (Neumann)

Finite Elements (FE)

Bei der Methode der finiten Elemente wird ein gänzlich anderer Zugang gewählt. Anstelle einer Abschwächung der Differentialgleichung $L\,u = f$ mit dem Operator L (also z. B. Δ) und der rechten Seite f über die direkte Diskretisierung der einzelnen Ableitungsterme wird jetzt verlangt, dass $L\,u = f$ nur im *schwachen* Sinne, d. h. im Sinne eines Skalarprodukts mit einer endlichen Anzahl von Testfunktionen ψ_l, erfüllt ist:

$$L\,u \;=\; f \qquad \longrightarrow \qquad < L\,u, \psi_l > \;=\; < f, \psi_l > \qquad 1 \le l \le N\,.$$

Es wird sich zeigen, dass dies einem Variationsansatz entspricht: Die gesuchte Lösung u löst auch ein Minimierungsproblem eines geeignet zu wählenden Energiefunktionals. Die Testfunktionen ψ_l sind einfach gebaute Funktionen mit kompaktem Träger, sie leben nur auf kleinen Teilgebieten. Mit dieser Abschwächung geht folglich die Partitionierung des Grundgebiets in *finite Elemente* einher. Im Gegensatz zu finiten Differenzen ist der Spielraum bei der Wahl der Positionen der Gitterpunkte sowie bei der Wahl der Form der Parzellen (Element-Typen) wesentlich größer. Hier treffen sich die soeben knapp beschriebene Sehweise des Mathematikers und die des Ingenieurs (die Urpsrünge der anwendungsorientierten FE-Methode liegen im Bereich der Mechanik bzw. Statik), für den diese Technik primär zur Partitionierung seines (i. d. R. komplizierten) Gebildes in einfacher zu handhabende Elemente bestimmter vorgegebener Formklassen dient. Auf den Elementen werden dann geeignete lokale Lösungen generiert, die zu einer Gesamtlösung des Problems zusammengebaut oder *kompiliert* werden.

Die einzelnen Bearbeitungsschritte:

1. **Gittergenerierung:**
   Zerlegung des Gebiets in einzelne „Elemente“; die Eckpunkte, an denen verschiedene Elemente zusammentreffen (Knoten), bilden ein Gitter Ω_n.

   Jedem Gitterpunkt x_k ist eine *Ansatzfunktion* φ_k zugeordnet, beispielsweise eine Hutfunktion im eindimensionalen Fall oder eine Pagodenfunktion in 2 D (siehe Abb. 3.5).

   Diese verschwindet auf allen außer den an x_k angrenzenden finiten Elementen (endlicher Träger; evtl. können sich die Ansatzfunktionen auch auf einen weiteren, aber immer noch lokalen Bereich erstrecken). Alle Ansatzfunktionen zusammen spannen einen linearen Raum V_n auf, den sogenannten *Ansatzraum*:

$$V_n \;:=\; \mathrm{span}\{\varphi_k : x_k \in \Omega_n\}\,.$$

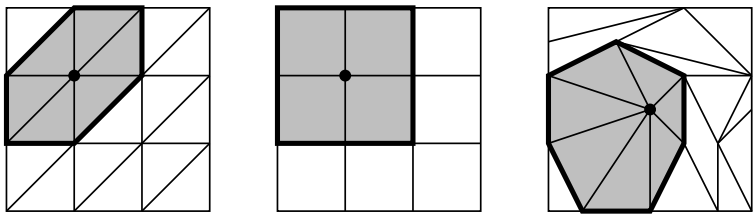


Abbildung 3.4: Regelmäßige Dreiecks- und Viereckgitter sowie unregelmäßiges Gitter mit Trägern potentieller Basisfunktionen

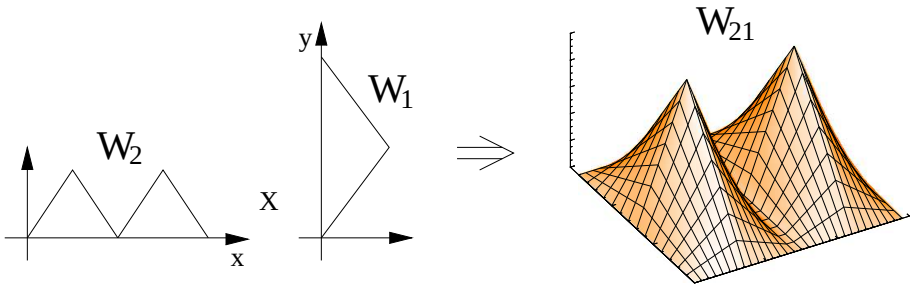


Abbildung 3.5: Hutfunktion (1 D) und Pagodenfunktion (2 D) als typische Basisfunktionen auf Intervallen bzw. Rechtecken

In der Tat bilden die φ_k eine Basis von V_n. In V_n suchen wir nun eine Approximation der Lösung unserer partiellen Differentialgleichung bzw. unseres Randwertproblems.

2. **Schwache Form der partiellen Differentialgleichung:**
   Wir schwächen die partielle Differentialgleichung $L\,u = f$ auf Ω ab zu

$$\int_\Omega Lu \cdot \psi_l \, \mathrm{d}\mathbf{x} \;=\; \int_\Omega f \cdot \psi_l \, \mathrm{d}\mathbf{x}$$

   für bestimmte *Testfunktionen* ψ_l (*Methode der gewichteten Residuen, Galerkin-Ansatz*). Neben dem Ansatzraum V_n hat man jetzt also einen zweiten endlich-dimensionalen Raum, den von den ψ_l aufgespannten *Testraum*. Die Testfunktionen ψ_l kann man auf verschiedene Art wählen. Sind die Testfunktionen mit den Ansatzfunktionen identisch, spricht man von *Ritz-Galerkin-Verfahren*, andernfalls von *Petrov-Galerkin-Verfahren*. Im ersten Fall bedeutet die

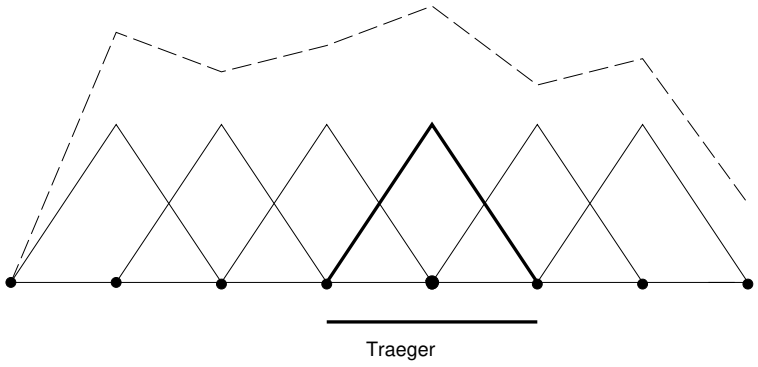


Abbildung 3.6: Stückweise lineare Stützpunktbasis

schwache Form also

$$\int_\Omega Lu \cdot \varphi \, \mathrm{d}\mathbf{x} \quad = \quad \int_\Omega f \cdot \varphi \, \mathrm{d}\mathbf{x} \qquad \forall \varphi \in V_n \ .$$

Aufgrund der Linearität von V_n reicht es, wenn man dies für alle *Basisfunktionen* φ_k fordert. Jetzt beschränken wir uns auf *lineare* Differentialoperatoren L, also bspw. den Laplace-Operator Δ. Mit der *Bilinearform*

$$a(u,v) \quad := \quad \int_\Omega Lu \cdot v \, \mathrm{d}\mathbf{x}$$

und der *Linearform*

$$b(v) \quad := \quad \int_\Omega f \cdot v \, \mathrm{d}\mathbf{x}$$

für $u, v \in V_n$ erhält man schließlich folgendes System linearer Gleichungen:

$$a(u, \varphi_k) \quad = \quad b(\varphi_k) \qquad \forall \ \varphi_k \in V_n.$$

3. **Diskrete Approximation:** Jetzt bedenken wir, dass wir unsere Approximation u_n in V_n suchen, also als Linearkombination der Basisfunktionen φ_l:

$$u_n = \sum_l \alpha_l \cdot \varphi_l \ .$$

Diesen Ansatz für u_n setzen wir jetzt in das aus der schwachen Form abgeleitete Gleichungssystem ein und erhalten

$$a(u, \varphi_k) \quad = \quad a\left(\sum_l \alpha_l \cdot \varphi_l \ , \ \varphi_k \right) \quad = \quad \sum_l \alpha_l \cdot a(\varphi_l, \varphi_k) \quad = \quad b(\varphi_k) \qquad \forall \varphi_k \in V_n \ .$$

Alle $a(\varphi_l, \varphi_k)$ und $b(\varphi_k)$ hängen nur vom gegebenen Problem, nicht jedoch von der aktuellen Näherungslösung ab. Folglich müssen sie nur einmal (z. B. zu Beginn) berechnet werden.

Somit haben wir ein lineares Gleichungssystem $Ax = b$ in den Unbekannten α_i:

$$\begin{aligned} A &= (a(\varphi_i, \varphi_j))_{i,j} \ , \\ b &= (b(\varphi_i))_i \ + \ < \text{Einfluß der Randbedingungen} > \ , \\ x &= (\alpha_i)_i \ . \end{aligned}$$

Die Matrix A wird hierbei *Steifigkeitsmatrix* genannt.

4. **Lösung des linearen Gleichungssystems:** Die Lösung x des linearen Gleichungssystems $Ax = b$ stellt gerade die Finite-Elemente-Approximation in V_n zur Lösung unseres Ausgangsproblems dar. Somit ist die Steifigkeitsmatrix mit ihrer Gestalt und ihren Eigenschaften für den Lösungsprozess von entscheidender Bedeutung. Optimal ist natürlich eine Diagonalmatrix (d. h., die Basis $\{\varphi_k\}$ ist a-orthogonal), dann ist das Lösen von $Ax = b$ trivial. Einige abschließende Bemerkungen zu A:

   - Oft ist A symmetrisch positiv definit.
   - Bei Verwendung von Stützpunktbasen (alle Basisfunktionen haben dieselbe Gestalt und Trägergröße) ist A dünn besetzt und hat Bandstruktur. Denn für alle Basisfunktionen φ_i und φ_j, deren Träger sich nicht überlappen, gilt

$$a_{i,j} = a(\varphi_i, \varphi_j) \quad = \quad \int_\Omega L\varphi_i \cdot \varphi_j \, \mathrm{d}\mathbf{x} \quad = \quad 0 \ .$$

     Hier wird der Term „finite“ Elemente klar.

– Bei hierarchischen Basen (unterschiedliche Trägergröße der einzelnen Basisfunktionen) überlappen wesentlich mehr Elemente. Der Träger der hierarchisch höchsten Basisfunktion etwa überlappt mit allen Elementen. Man bekommt einen fill-in, die Matrix ist aber nach wie vor dünn besetzt (die Träger der Basisfunktionen einer Ebene sind ja paarweise disjunkt). Durchschnittlich ergeben sich pro Zeile $O(n)$ Nicht-Null-Einträge, wenn n die Auflösung des Raumes angibt ($h = 2^{-n}$).

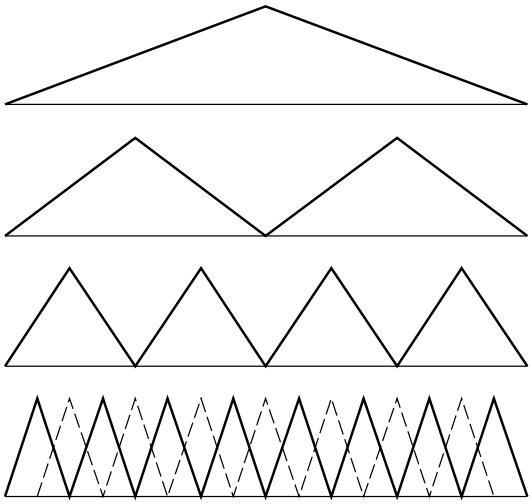


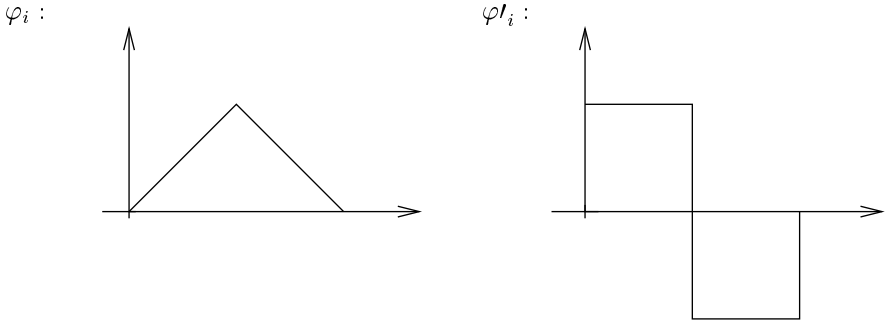
Abbildung 3.7: Stützpunktbasis (untere Zeile) und hierarchische Basis (alle Zeilen, nur die fett gezeichneten Funktionen) in 1 D

– Das Ziel muß eine Basis sein, die zu möglichst einfach bzw. schnell zu lösenden linearen Gleichungssystemen führt. Im Idealfall ist A diagonal, zumindest aber sollte A so beschaffen sein, dass die gängigen Iterationsverfahren (siehe Abschnitt 3.5) schnell konvergieren.

Wir betrachten als einfaches Beispiel die eindimensionale Poisson-Gleichung $-\Delta u = f$ bzw. $-u'' = f$. Die erste Greensche Formel liefert

$$a_{i,j} \;=\; \int\limits_{\Omega} \nabla\varphi_i \cdot \nabla\varphi_j \; \mathrm{d}\mathbf{x} \;=\; \int\limits_{\Omega} \varphi_i' \cdot \varphi_j' \; \mathrm{d}x\,.$$

Bei Verwendung der stückweise linearen hierarchischen Basis ergibt sich folgendes Bild:



Offensichtlich gilt $a_{i,j} \neq 0 \;\Leftrightarrow\; i = j$. Somit haben wir einen der seltenen Fälle, in denen es gelingt, eine diagonale Steifigkeitsmatrix zu bekommen!

Was Konvergenzaussagen angeht, so ist die folgende Eigenschaft der *Best-Approximation* (Céa-Lemma) von zentraler Bedeutung:

Sei $u_n^{FE} \in V_n$ die Finite-Elemente-Approximation zur Lösung u des Randwertproblems mit symmetrischem und positiv definitem a bzw. A. Dann gilt in der Norm $\|u\|_a := \sqrt{a(u,u)}$, d. h. in der sog. *Energie-Norm*:

$$\|u - u_n^{FE}\|_a \quad \leq \quad \inf_{u_n \in V_n} \|u - u_n\|_a \ .$$

Die Finite-Elemente-Lösung ist also Best-Approximation von u in V_n.

Die Abschätzung des Interpolationsfehlers in der Energienorm in V_n, die in der Regel ohne größere Probleme zu bestimmen ist, ist folglich obere Schranke für den Fehler der Finite-Elemente-Lösung u_n^{FE}, da der Interpolant $u_n^I \in V_n$ von u in V_n zulässiger Kandidat bei der Suche nach dem Infimum ist. Daraus folgt, dass Konvergenzaussagen in der Finite-Elemente-Welt bzgl. der (problemabhängigen) $\|\cdot\|_a$ -Norm i. A. leicht, bzgl. der (problemunabhängigen) L_∞- bzw. L_2-Norm dagegen oft ungleich schwerer sind.

Die zentrale Aufgabe in der (mathematischen) Finite-Elemente-Welt lautet somit:

Finde endlichdimensionale Approximationsräume V_n mit Basen $\{\varphi_k\}$, so dass erstens die jeweiligen Näherungslösungen u_n die Lösung u gut (und mit wachsendem n immer besser) approximieren und zweitens die u_n leicht (d. h. billig) zu berechnen sind.

Finite Volumes

Other Schemes

3.3.2 Discretizing Time

Bisher haben wir uns mit der Diskretisierung der Ortsableitungen befasst. Wie gestaltet sich nun die Behandlung der Zeit? Hier gibt es zwei prinzipiell verschiedene Wege: Man kann entweder die Zeit einfach als $d+1$. Raumdimension ansehen und somit wie eine Ortskoordinate behandeln. Unsere zeitabhängige Wärmeleitungsgleichung wird dann einfach zu einer parabolischen Differentialgleichung in $d+1$ Raumdimensionen, die mit den zuvor vorgestellten Techniken diskretisiert werden kann. Oder aber man betrachtet die Zeit als etwas grundlegend anderes als den Raum. Der einfachste Weg, dann die Zeit zu berücksichtigen, ist die *quasi-stationäre* Vorgehensweise: Man schaltet die Randbedingungen sukzessive um einen Zeitschritt hoch und löst dann jeweils das resultierende stationäre Problem. Offensichtlich vernachlässigt diese Strategie zeitliche Änderungen nicht völlig, es ist aber auch klar, dass dies nur für Probleme mit sehr kleinen Änderungen in der Zeit einigermaßen korrekt ist – schließlich geht man bei der Lösung der stationären Wärmeleitungsprobleme von verschwindenden Zeitableitungen aus, was i. A. bei zeitabhängigen Problemen nicht der Fall ist. Genauso wenig ist übrigens die umgekehrte Vorgehensweise ein Allheilmittel, die Ortsabhängigkeit zu ignorieren und das Ganze als eine Menge gewöhnlicher Differentialgleichungen zu betrachten, also etwa, ausgehend vom Anfangswert, für jeden Gitterpunkt separat die Zeit einem geeigneten Abklinggesetz (etwa $u + u_t = 0$) folgend hochzuschalten.

Trotz ihrer Mängel werden beide Ansätze oft als vereinfachende Näherungen herangezogen, wenn das Originalproblem als zu aufwendig erscheint. So tastet man sich oft über eine quasistationäre Simulation an die echte Instationarität heran. Beim Übergang von einem rein zeitabhängigen Modell ohne Ortsauflösung zu einem Modell mit Berücksichtigung des Raums beginnt man andererseits oftmals damit, anstelle einer gewöhnlichen Differentialgleichung für das gesamte betrachtete Gebiet einige wenige unterschiedlich parametrisierte auf Teilgebieten zu betrachten.

Die instationäre Simulation erfordert dann gleichzeitige bzw. alternierende Diskretisierungsschritte in Raum und Zeit, wobei jeweils die Zeitableitungen bei der Ortsdiskretisierung und die Ortsableitungen bei der Zeitdiskretisierung als von der Lösung nicht abhängende rechte Seiten angesehen werden. Für die konkrete Realisierung der Zeitintegration steht das Instrumentarium der numerischen Lösung gewöhnlicher Differentialgleichungen zur Verfügung (explizt/implizit, 1-Schritt/s-Schritt etc.).

Eine abschließende Bemerkung: Bei der Wahl der Zeitschrittweite Δt und der Maschenweiten h_x, h_y und h_z für den Raum hat man nur eingeschränkte Wahlmöglichkeiten. Will man *Instabilitäten* vermeiden (also Oszillationen anstelle von Konvergenz gegen die Lösung des Problems), dann müssen Zeitschrittweite und Maschenweiten aufeinander abgestimmt werden. Dabei gilt i. A., dass eine feinere Raumauflösung auch eine feinere Diskretisierung der Zeit erfordert. Dies erscheint auch als plausibel: Schließlich macht es wenig Sinn, kleinste Raumdetails zu berücksichtigen, aber nur im Sekundentakt das Szenario zu betrachten. Eine formale Gesetzmäßigkeit werden wir bei der Fallstudie zur Strömungsmechanik noch kennenlernen.

3.4 Direct Solution of Systems of Linear Equations

The numerical treatment of mathematical models nearly always leads to some basic tasks from (numerical) linear algebra, like determining some or all eigenvalues of a given matrix, solving large and sparse systems of linear equations, or, in a wider sense of the term 'linear algebra', the linearization of nonlinear equations. Therefore, the development of efficient algorithms for these tasks which are well-suited for the type of computer architecture they shall run on is of crucial importance. In contrast to lectures on numerical analysis where, ususally, the first aspect is dealt with only, for us, parallelization and vectorization properties are as important issues as sole numerical efficiency is. Thus, we have to discuss them both. In this section, some basics of the direct solution of large linear systems of equations will be presented, whereas the discussion of aspects concerning the derived algorithms' use in scientific computing is postponed to Chapter 4, when an overview of computer architectures and their properties will have been provided. Furthermore, note that there are several numerical libraries that offer optimized modules for the tasks discussed in the following (see also Sect. 4.5 on tools for scientific computing). Therefore, before programming each module on his or her own, the reader is strongly encouraged to look for available solutions.

3.4.1 Gaussian Elimination and Cholesky

3.4.2 Nested Dissection

3.5 Iterative Solution of Large Sparse Linear Systems

The discretization of our partial differential equation(s) led us to a finite system of discretized equations in a finite number of unknowns. For our example of heat conduction, this system is already a *linear* one, as we know it from linear algebra. Generally, however, the resulting discrete system will be *nonlinear*: The derivatives do not only occur in linear combinations, but there are products like $u \cdot u_x$ oder u_x^2, too. In this case, a linearizing step has to be applied, first. We will discuss suitable methods for that in Sect. 3.6. After such a linearization, we have to deal with the central task of solving a linear system of equations $Ax = b$ with $A \in \mathbb{R}^{n,n}$, too. Since the dimension n directly depends on the resolution of the underlying grid, n will be very large for realistic problems of numerical simulation[1]. Furthermore, typically, A has only a small number of nonzeroes, it is *sparse*. To see that, think of reasons of nonzeroes in discretization matrices. For both finite differences and finite elements, these result from couplings between grid points, which occur only between (in some sense) neighbouring points. Thus, each line of A usually contains only a limited number of nonzero entries that does not depend on n. Such types of linear systems have to be solved in an efficient way that profits from the sparsity of the matrix. This is especially important since this standard step must be made a possibly large number of times (during each time step, during each step of the linearization procedure of the nonlinear algorithm).

[1] A three-dimensional equidistant grid with 1000 points in each direction leads to $n = 10^9$, for example.

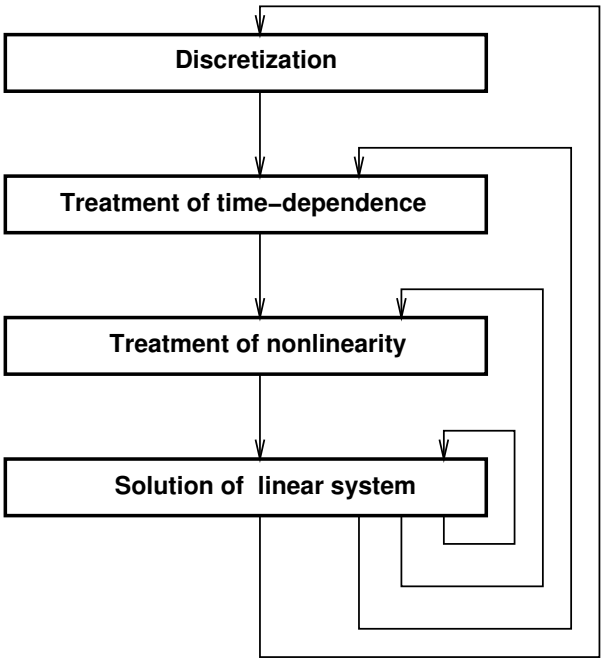


Abbildung 3.8: Numerical treatment of a time-dependent nonlinear PDE

Therefore, in the following, we study linear systems $Ax = b$ with very large and sparse system matrices $A \in \mathbb{R}^{n,n}$. In many cases, A will be symmetric positive definite, and a direct solution will be prohibitive due to reasons of memory capacity or runtime. As we have seen in Sect. 3.4, the classical direct solvers need $\mathcal{O}(n^3)$ or, in the best case of a full exploitation of sparsity, $\mathcal{O}(n^2)$ operations. Thus, there is an evident need for *iterative* solvers. The final objective of the construction of iterative solvers is to obtain a linear, i. e. an $\mathcal{O}(n)$ behaviour of the computational complexity of the solution of $Ax = b$. More can hardly be expected, since each step of the iteration that visits and updates each unknown once needs already $\mathcal{O}(n)$ operations[2]. Consequently, the number of steps of the iteration that are necessary to reduce the error by some prescribed factor must not depend on n and, thus, on the resolution of the mesh, or must be bounded from above with an upper bound that does not depend on n, respectively.

As an example, we briefly discuss a discretization on a rectangular equidistant grid of mesh width $h_l = 2^{-l}$. In the d-dimensional case, we already know that this leads to $n := (2^l - 1)^d$ inner grid points (relevant for Dirichlet boundary conditions) or $n := (2^l + 1)^d$ overall grid points (relevant for Neumann boundary conditions). If we are only interested in one physical quantity (the temperature, for example), then this means n degrees of freedom, too, i. e. $A \in \mathbb{R}^{n,n}$.

Starting from some given $x^{(0)}$, we apply our (not yet defined) iterative scheme a number of times. If the iteration converges towards the (discrete) solution x, we have

$$x^{(0)} \longrightarrow \ldots \longrightarrow x^{(i)} \longrightarrow x^{(i+1)} \longrightarrow \ldots \longrightarrow \lim_{i\to\infty} x^{(i)} \equiv x^{\mathrm{exact}} \equiv x\,. \tag{3.52}$$

Now, let $\gamma \in \,]0,1[\,$ denote the factor by which the error is reduced in each iterative step:

$$\|x - x^{(i+1)}\| \;<\; \gamma \cdot \|x - x^{(i)}\|\,. \tag{3.53}$$

[2] Note that this is not always necessary. There are iterative schemes which update only a selection of unknowns during each step – those which look promising in the sense that their update seems to be worthwhile for the reduction of the overall error. In this text, however, we won't discuss such methods.

Hence, i steps lead to an overall reduction of γ^i. Here, it is important to note that most iterative schemes show a behaviour like

$$\gamma \;=\; \mathcal{O}(1 - n^{-k})\,. \tag{3.54}$$

for some positive k. Obviously, this n-dependence leads to a convergence behaviour that is inacceptable: If the mesh is refined, n increases, and γ gets that close to 1 that the positive effect of one single step of the iteration becomes negligible. Thus, during the following attempts to construct iterative methods, we must keep in mind the point that we need

$$\gamma \;<\; 1 - \text{const.} \tag{3.55}$$

This can be achieved, as the section on multigrid methods will show. But, first, let us turn to the 'classics'.

3.5.1 Classical Relaxation Techniques

Overview

The probably oldest iterative schemes for systems of linear equations $Ax = b$ with $A \in \mathbb{R}^{n,n}$ and $x, b \in \mathbb{R}^n$ are the so-called *relaxation* or *smoothing* methods: the *Richardson* iteration, the *Jacobi* iteration, the *Gauß-Seidel* iteration, and SOR, the *successive over relaxation*. The name relaxation or smoother stems from a certain property we will have a look at later. We begin with the algorithms, followed by a short derivation and analysis.

For all of these methods, the starting point is the *residual* $r^{(i)}$,

$$r^{(i)} \;:=\; b - Ax^{(i)} \;=\; Ax - Ax^{(i)} \;=\; A\left(x - x^{(i)}\right) \;=\; -Ae^{(i)}\,, \tag{3.56}$$

where $x^{(i)}$ denotes the approximation to x after i steps of the iteration, and $e^{(i)}$ denotes the respective error. Since, obviously, $e^{(i)}$ is not available without knowing the exact solution x, it turns out to be reasonable to take $r^{(i)}$ as the direction where to look for an improvement. The Richardson iteration is the most straightforward one and takes the residual as it is. More sophisticated are the Jacobi and Gauß-Seidel scheme, correcting $x_k^{(i)}$ in a way that $r_k^{(i)}$ vanishes (for the moment, at least). Finally, SOR or its counterpart, a *damped* relaxation, take into account that such a correction is often a too small or a too big step, respectively.

Richardson iteration:

```
for i = 0,1,...
          for k = 1,...,n:      x_k^(i+1) := x_k^(i) + r_k^(i)
```

Jacobi iteration:

```
for i = 0,1,...
          for k = 1,...,n:      y_k := (1/a_ii) · r_k^(i)
          for k = 1,...,n:      x_k^(i+1) := x_k^(i) + y_k
```

Hence, in each substep k of a step i, a correction y_k is computed and stored, which would lead (if applied immediately) to a vanishing k-component of the residual $r^{(i)}$. However, all corrections of one step of iteration are only applied at the end of this step.

Gauß-Seidel iteration:

```
for i = 0,1,...
          for k = 1,...,n:
```
$$y_k := \frac{1}{a_{ii}} \cdot r_k^{(i)}, \qquad x_k^{(i+1)} := x_k^{(i)} + y_k$$

Here, we compute the same correction y_k as we do in the Jacobi case, but we apply it at once instead of postponing the update.

Sometimes, in each of the above three methods, some *damping* ($0 < \alpha < 1$) or *over relaxation* ($1 < \alpha < 2$) leads to a better convergence behaviour:

$$x_k^{(i+1)} := x_k^{(i)} + \alpha y_k \,. \tag{3.57}$$

If applied to the Gauß-Seidel scheme, the over relaxation version is called *SOR* or *successive over relaxation* method.

Analysis

For some convergence analysis of the above iterative schemes, we need a more algebraic formulation of them. All are based on the simple idea to write the matrix A as a sum $A = M + (A - M)$, where $Mx = b$ should be easy to solve and $A - M$ should not be too important with respect to some matrix norm. With the help of such an M, Richardson, Jacobi, Gauß-Seidel, and SOR iterations can all be written as

$$Mx^{(i+1)} + (A - M)x^{(i)} = b \tag{3.58}$$

or

$$x^{(i+1)} := M^{-1}b - M^{-1}(A - M)x^{(i)} = M^{-1}b - (M^{-1}A - I)x^{(i)} = x^{(i)} + M^{-1}r^{(i)} \,. \tag{3.59}$$

Furthermore, we split A into its diagonal part D_A, its strictly lower triangular part L_A, and its strictly upper triangular part U_A:

$$A =: L_A + D_A + U_A \,. \tag{3.60}$$

Then, for our concrete iterative schemes, the following relations hold:

$$\begin{aligned} \text{Richardson}: \quad M &:= I \,, \\ \text{Jacobi}: \quad M &:= D_A \,, \\ \text{Gauß-Seidel}: \quad M &:= D_A + L_A \,, \\ \text{SOR}: \quad M &:= \frac{1}{\alpha} D_A + L_A \,. \end{aligned} \tag{3.61}$$

If we look at the algorithmic formulation of the Richardson and of the Jacobi method, the first two relations are obvious. Since Gauß-Seidel is a special case of SOR ($\alpha = 1$), we have to show (3.61) for the latter only. From the algorithm, we immediately get

$$\begin{aligned} & x_k^{(i+1)} := x_k^{(i)} + \alpha \left(b_k - \sum_{j=1}^{k-1} a_{kj} x_j^{(i+1)} - \sum_{j=k}^{n} a_{kj} x_j^{(i)} \right) / a_{kk} \\ \Leftrightarrow \quad & x^{(i+1)} := x^{(i)} + \alpha D_A^{-1} \left(b - L_A x^{(i+1)} - (D_A + U_A) x^{(i)} \right) \\ \Leftrightarrow \quad & \frac{1}{\alpha} D_A x^{(i+1)} = \frac{1}{\alpha} D_A x^{(i)} + b - L_A x^{(i+1)} - (D_A + U_A) x^{(i)} \\ \Leftrightarrow \quad & \left(\frac{1}{\alpha} D_A + L_A \right) x^{(i+1)} + \left((1 - \frac{1}{\alpha}) D_A + U_A \right) x^{(i)} = b \\ \Leftrightarrow \quad & Mx^{(i+1)} + (A - M)x^{(i)} = b \,, \end{aligned}$$

which completes the proof of (3.61) for SOR.

Concerning convergence, there are two direct consequences of (3.58):

- If the sequence $\left(x^{(i)}\right)$ converges, then the limit is the exact solution x of our system $Ax = b$.
- The spectral radius ρ of the iteration matrix $-M^{-1}(A - M)$ is the crucial quantity for convergence:

$$\lim_{i\to\infty} x^{(i)} = x = A^{-1}b \quad \Leftrightarrow \quad \rho < 1\,. \tag{3.62}$$

  To see this, subtract $Mx + (A - M)x = b$ from (3.58) to get

$$Me^{(i+1)} + (A - M)e^{(i)} = 0 \quad \Leftrightarrow \quad e^{(i+1)} = -M^{-1}(A - M)e^{(i)}\,. \tag{3.63}$$

  Obviously, all error components are reduced if and only if $\rho < 1$.

There are a number of convergence results, and we just want to cite some of the most important:

- A necessary condition for the convergence of the SOR iteration is $0 < \alpha < 2$.
- If A is positive definite, then both the SOR iteration (for $0 < \alpha < 2$) and, thus, the Gauß-Seidel iteration converge.
- If A and $2D_A - A$ are both positive definite, then the Jacobi iteration converges.
- If A is strictly diagonally dominant (i. e. $a_{ii} > \sum_{j\neq i} |a_{ij}|$ for all i), then both the Jacobi and the Gauß-Seidel iteration converge.
- For special types of matrices A, the optimum parameter α (i. e. the one resulting in the smallest number of steps of iteration) can be computed in advance.

Note that the Gauß-Seidel iteration is not generally better than the Jacobi iteration (as one might suppose due to the somewhat better update strategy). There are examples where the first converges whereas the latter diverges, and vice versa. In many situations like the standard finite difference discretization of the Laplacian, however, the Jacobi method needs twice the number of steps of iteration than the Gauß-Seidel method does and, thus, is actually better.

Obviously, ρ is not only crucial for the question whether the iteration converges at all, but also for its quality, i. e. its speed of convergence: The smaller ρ is, the faster all components of the error $e^{(i)}$ are reduced in each step of iteration, and the faster, thus, $e^{(i)}$ gets smaller. Actually, the convergence results cited above are more of a theoretical importance. In practice, ρ is often so close to 1 that the number of iterations that are necessary to obtain some prescribed accuracy is much too high. Take our primary context of matrices A resulting from discretized partial differential equations as an example. Here, ρ usually depends on the number n of unknowns and, thus, on the resolution h of the underlying grid. For a 2 D Poisson equation (equidistant rectangular grid with mesh width $h_l = 2^{-l}$, piecewise bilinear finite elements), for example, we get

$$\rho \;=\; \mathcal{O}(1 - h_l^2) \;=\; \mathcal{O}\left(1 - \frac{1}{4^l}\right)\,. \tag{3.64}$$

This is a tremendous drawback: The finer and, thus, more accurate our grid is, the poorer the convergence behaviour of our iterative method becomes. Hence, much more has to be invested here, even if A is symmetric and positive definite.

3.5.2 Krylov Subspace Methods

Overview

In this section, we want to study an alternative approach in order to construct iterative methods for the solution of large sparse systems of linear equations $Ax = b$. As before, we start from

$A = M + (A - M)$, where $Mx = b$ should be easy to solve and $A - M$ should be small with respect to some matrix norm. The most evident choice of $M := I$ led us to the Richardson iteration

$$x^{(i)} := x^{(i-1)} + r^{(i-1)} \qquad \text{or} \qquad x^{(i)} := x^{(i-1)} + \alpha_i r^{(i-1)} \tag{3.65}$$

with some relaxation parameter α_i. Here, obviously, the residual $r^{(i)} = b - Ax^{(i)}$ fulfils

$$r^{(i)} = b - A(x^{(i-1)} + \alpha_i r^{(i-1)}) = (I - \alpha_i A) r^{(i-1)} = \ldots = p_i(A) r^{(0)} \tag{3.66}$$

with the polynomial $p_i(z) := \prod_{j=1}^{i} (I - \alpha_j z)$, where the choice of the parameters α_j offers a lot of possibilities to construct more sophisticated variants of the classical Richardson scheme.

Now, let us look at the case of a general M, hopefully a better approximation to A than the identity I. Then, instead of (3.65), we get (3.59) and, thus,

$$x^{(i)} := x^{(i-1)} + M^{-1} r^{(i-1)} \qquad \text{or} \qquad x^{(i)} := x^{(i-1)} + \alpha_i M^{-1} r^{(i-1)} \,. \tag{3.67}$$

Note that M^{-1} is almost never constructed explicitly in practical methods. (3.59) shows that the iterative method with a general M applied to $Ax = b$ is formally equivalent to the standard Richardson scheme applied to the modified or *preconditioned* system $M^{-1}A = M^{-1}b$. In this sense, Jacobi, Gauß-Seidel, and SOR iterations are nothing else but a preconditioned Richardson iteration, with the respective M given in (3.61). Due to the equivalence of I applied to $Ax = b$ and M applied to $M^{-1}A = M^{-1}b$, we can restrict ourselves in the following to the simple case of $M = I$ without loss of generality. But, in contrast to the situation in the previous section, we now use the damping parameters α_i to construct certain polynomials p_i which ensure further properties of the new approximation x_i. Obviously, with the assumption $x^{(0)} = 0$ and, thus, $r^{(0)} = b$, (3.66) provides for $i \geq 1$

$$x^{(i)} = \sum_{j=1}^{i} \alpha_j r^{(j-1)} = \sum_{j=1}^{i} \alpha_j p_{j-1}(A) r^{(0)} = \sum_{j=1}^{i} \alpha_j p_{j-1}(A) b \,, \tag{3.68}$$

i. e.

$$x^{(i)} \in \left\{ b, Ab, \ldots, A^{i-1}b \right\} =: \mathcal{K}_i(A, b) \,, \tag{3.69}$$

where the finite sequence $(b, Ab, \ldots, A^{i-1}b)$ is called a *Krylov sequence* of length i, and where the subspace $\mathcal{K}_i(A, b) \subseteq \mathbb{R}^n$ is called the *Krylov subspace* of dimension i, generated by A and b[3]. If a linear dependence occurs, i. e., if $\mathcal{K}_{i+1} = \mathcal{K}_i$, the Krylov sequence ends. For practical Krylov subspace methods, this means that the solution x has been found.

When j increases, eigenvectors of the (w. r. t. its absolute value) largest eigenvalue of A more and more dominate A^j, such that a (numerical) linear dependence in the basis provided by the Krylov sequence and, hence, numerical problems in identifying the optimum approximations in the Krylov subspaces may occur. Therefore, it is advantageous to work with better suited, i. e. orthonormal bases of $\mathcal{K}_i$. *Arnoldi* and *Lanczos* provided methods for obtaining such bases via a modified Gram-Schmidt orthogonalization.

At that point, the question of the appropriate choice of the α_j arises. Two evident possibilities are minimizing $\|r^{(i)}\|_2$ or forcing $r^{(i)}$ to be orthogonal to $\mathcal{K}_i(A, b)$. But there are more possible strategies to select the i-th approximate solution $x^{(i)}$ in $\mathcal{K}_i$, all of projection-type:

- a *Ritz-Galerkin* approach, using $r^{(i)} \perp \mathcal{K}_i$ as criterion (realized in CG (*conjugate gradients*, see below), FOM (*full orthogonalization method*), or IOM (*incomplete orthogonalization method*)),

[3] A Krylov sequence ends by definition if a linear dependence occurs; therefore, if $b, Ab, \ldots, A^{i-1}b$ form a Krylov sequence, it is of length i, and the corresponding Krylov subspace $\mathcal{K}_i$ is of dimension i.

- a *Petrov-Galerkin* approach, using $r^{(i)} \perp \mathcal{L}_i$ for some other suitable subspace $\mathcal{L}_i$ of dimension i (realized in the Bi-CG or QMR (*quasi minimum residual*) methods, which both use the Krylov subspace of A^T as $\mathcal{L}_i$),
- a *minimum residual* approach, requiring $\|r^{(i)}\|_2$ to be minimal on $\mathcal{K}_i$ (realized in MINRES (*minimum residual*) and GMRES (*generalized minimum residual*)),
- hybrid combinations of the above three approaches, for example CGS (*conjugate gradient squared*), Bi-CGSTAB, Bi-CGSTAB(l), and TFQMR (*transpose-free quasi minimum residual*),
- a *minimum error* approach, requiring $\|x - x^{(i)}\|_2$ to be minimal on $A^T\mathcal{K}_i(A^T, b)$ (realized in SYMMLQ or GMERR).

There are a lot of Krylov subspace methods that have been developed for the iterative solution of large linear systems. Besides the above classification, this variety can be structured based upon the methods' history and their algorithmic relations:

- The conjugate gradient method (CG) can deal with symmetric positive definite matrices A. If A is only symmetric, MINRES can produce a (minimim residual) solution. For the general case, CG's variants CGNE (*conjugate gradient normal equation*) or CGNR (*conjugate gradient normal equation for minimum residual*) tackle the normal equations $A^TAx = A^Tb$ instead of the original system. The LSQR algorithm provides a stable way for the solution of the normal equations.
- GMRES and its more memory-efficient restart[4] version GMRES(m) can treat general nonsymmetric systems.
- Bi-CG and its improved variant QMR solve general nonsymmetric systems using a biorthogonal basis approach. There are close relations to CGS and its variant TFQMR.
- Bi-CGSTAB (*bi-conjugate gradient stabilized*) and its restart version Bi-CGSTAB(l), a combination of Bi-CG and GMRES(m), for general nonsymmetric systems.

We will restrict ourselves to a discussion of the conjugate gradient algorithm as the 'mother of all Krylov subspace methods', starting with some more heuristic derivation from quadratic optimization. For further information, see the book of Dongarra et al., for example.

Minimizing Quadratic Forms and Steepest Descent

For the following, let $A \in \mathbb{R}^{n,n}$ be a symmetric and positive definite matrix. In this case, it is well-known that solving the linear system $Ax = b$ is equivalent to minimizing the quadratic form

$$f(x) \;:=\; \frac{1}{2}x^TAx - b^Tx + c \tag{3.70}$$

for any scalar constant c. Since A is positive definite, the surface defined by $f(x)$ is a paraboloid in $\mathbb{R}^{n+1}$ with ellipsoidal isolines in $\mathbb{R}^n$ (see Fig. 3.9), and there exists one global minimum x. The equivalence is obvious:

$$f'(x) \;=\; \frac{1}{2}A^Tx + \frac{1}{2}Ax - b \;=\; Ax - b \;=\; 0 \quad \Leftrightarrow \quad Ax = b\,. \tag{3.71}$$

[4] There are two standard approaches to increase runtime and memory efficieny of Krylov methods, *truncation* and *restart*. The first one takes into account only the k freshest vectors, the second one restarts the complete procedure after k iterations.

Actually, for each perturbation $e \neq 0$ of the solution x of $Ax = b$, we get due to (3.70)

$$\begin{aligned} f(x+e) &= \frac{1}{2}x^T Ax + e^T Ax + \frac{1}{2}e^T Ae - b^T x - b^T e + c \\ &= f(x) + e^T(Ax - b) + \frac{1}{2}e^T Ae \\ &= f(x) + \frac{1}{2}e^T Ae \;>\; f(x)\,. \end{aligned} \tag{3.72}$$

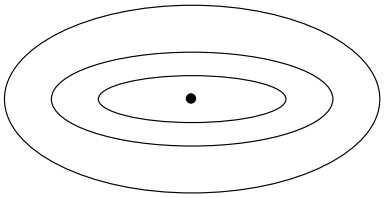


Abbildung 3.9: Isolines of $f(x)$, minimum in the centre

A simple way of finding the minimum x in the above situation is provided by the *method of steepest descent*. Here, repeat for $i = 0, 1, \ldots$

$$\begin{aligned} r^{(i)} &:= b - Ax^{(i)}\,, \\ \alpha_i &:= \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i)}}^T A r^{(i)}}\,, \\ x^{(i+1)} &:= x^{(i)} + \alpha_i r^{(i)}\,, \end{aligned} \tag{3.73}$$

or, start with $r^{(0)} := b - Ax^{(0)}$ and repeat for $i = 0, 1, \ldots$

$$\begin{aligned} \alpha_i &:= \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i)}}^T A r^{(i)}}\,, \\ x^{(i+1)} &:= x^{(i)} + \alpha_i r^{(i)}\,, \\ r^{(i+1)} &:= r^{(i)} - \alpha_i A r^{(i)}\,, \end{aligned} \tag{3.74}$$

which avoids one of the two matrix-vector products. The method of steepest descent looks for an improvement in the direction of the negative gradient $-f'(x^{(i)}) = r^{(i)}$, which explains its name. The direction of the steepest descent is chosen since the direction of the error $e^{(i)} := x^{(i)} - x$ is not known. The crucial question is how long to march in the direction of $r^{(i)}$. For that, we have to look where the partial derivative of $f(x^{(i)} + \alpha_i r^{(i)})$ with respect to α_i vanishes. A short piece of calculus provides (3.73) and (3.74), resp.

Note that choosing the residuals $r^{(i)}$ as search directions is not the most straightforward choice. Actually, we could look for an improvement along the coordinate axes, i. e. with the unit vectors $u^{(i)}$ as search directions. It can be shown that this is exactly what the Gauß-Seidel iteration does: n successive search steps in order to minimize $f(x^{(i)} + \alpha_i u^{(i)}), i = 1, \ldots, n$ are equivalent to one Gauß-Seidel step. This shows the close relations between the relaxation and the minimization approach.

For a convergence analysis of the method of steepest descent, note that the eigenvectors of A are directed along the axes of the paraboloid formed by our quadratic form f. Thus, if the actual error $e^{(i)}$ is an eigenvector of A, then $r^{(i)} = -Ae^{(i)}$ is an eigenvector, too, and the direction of correction is one of those axes and leads us to the minimum in one step. The same holds if all eigenvalues of A are equal: the ellipsoidal isolines are circles, so the gradient at each point is directed towards the minimum, and one step of steepest descent takes us to the minimum. However, these are the best

cases. In general, the component of some eigenvector may even increase from $x^{(i)}$ to $x^{(i+1)}$, a fact that can be responsible for arbitrary slow convergence. This property distinguishes the method of steepest descent (and CG, by the way) from so-called *smoothers* like Jacobi or Gauß-Seidel iteration, where such an increase of some eigenvector component can never happen, and explains why the methods discussed here are sometimes called *roughers*.

To quantify the convergence behaviour, it can be shown that the *energy norm* $\|e^{(i)}\|_A$ of the error $e^{(i)}$,

$$\|v\|_A \;:=\; \sqrt{v^T A v}\,, \tag{3.75}$$

fulfils

$$\|e^{(i)}\|_A \;\leq\; \left(\frac{\kappa-1}{\kappa+1}\right)^i \|e^{(0)}\|_A\,, \tag{3.76}$$

where

$$\kappa \;:=\; \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} \tag{3.77}$$

denotes the *spectral condition number* of A[5]. From that, with (3.72), we get the following bound for the convergence behaviour of the method of steepest descent:

$$\frac{f(x^{(i)})-f(x)}{f(x^{(0)})-f(x)} \;=\; \frac{{e^{(i)}}^T A e^{(i)}}{{e^{(0)}}^T A e^{(0)}} \;\leq\; \left(\frac{\kappa-1}{\kappa+1}\right)^{2i}. \tag{3.78}$$

Obviously, for very big κ, the gain in accuracy in each step of the steepest descent iteration may be close to nothing. A spectral condition close to 1, however, allows a huge gain. Again, this is a hint for *preconditioning*, i. e. solving $M^{-1}Ax = M^{-1}b$ instead of $Ax = b$ with a substantially improved condition of $M^{-1}A$ compared to that of A itself. We will discuss preconditioning later.

Conjugate Directions

The main reason for steepest descent's slow convergence is the fact that it may search in the same direction for several times (due to the lack of orthogonality in the search directions). However, if all search directions were orthogonal, and if the error after i steps were orthogonal to all previous i search directions, then we would never come back to a certain direction and would find the solution in at most n steps of iteration. Strictly speaking, such a method is no longer an iterative one, but a direct solver. That's the reason why CG and others are sometimes called *semi-iterative methods*.

Hence, instead of the residuals or negative gradients $r^{(i)}$, we are now looking for alternative search directions $d^{(i)}$, $i = 0, 1, \ldots$:

$$x^{(i+1)} \;:=\; x^{(i)} + \alpha_i d^{(i)}\,. \tag{3.79}$$

For the moment, we suppose the ideal case that the present error $e^{(i)}$ is orthogonal to the subspace spanned by all former search directions $d^{(0)}, \ldots, d^{(i-1)}$. Consequently, the new correction $\alpha_i d^{(i)}$ shall be chosen such that $d^{(i)}$ is orthogonal to $d^{(0)}, \ldots, d^{(i-1)}$ and that the resulting new error $e^{(i+1)}$, additionally, is orthogonal to $d^{(i)}$, i. e.

$$0 \;=\; {d^{(i)}}^T e^{(i+1)} \;=\; {d^{(i)}}^T \left(e^{(i)} + \alpha_i d^{(i)}\right) \quad\Rightarrow\quad \alpha_i \;:=\; -\frac{{d^{(i)}}^T e^{(i)}}{{d^{(i)}}^T d^{(i)}}\,. \tag{3.80}$$

[5] Note that, if A is not symmetric positive definite, then κ is defined as the ratio of the biggest and smallest positive absolute value of all eigenvalues.

Unfortunately, (3.80) is not very helpful, since we do not know the error $e^{(i)}$, of course. Therefore, as we did in the method of steepest descent, we focus on the residuals instead of the errors and look for *A-orthogonal* or *conjugate* search directions instead of standard orthogonal ones. Two vectors $u \neq 0$ and $v \neq 0$ are called *A-orthogonal* or *conjugate* with respect to A, if $u^T A v = 0$. Note that within a set of conjugate vectors (as in a set of orthogonal vectors), no linear dependence can occur. With this definition, we now request that the new error $e^{(i+1)}$ shall be conjugate to $d^{(i)}$, i. e.

$$0 \;=\; {d^{(i)}}^T A e^{(i+1)} \;=\; {d^{(i)}}^T A \left(e^{(i)} + \alpha_i d^{(i)} \right) \quad \Rightarrow \quad \alpha_i \;:=\; -\frac{{d^{(i)}}^T A e^{(i)}}{{d^{(i)}}^T A d^{(i)}} \;=\; \frac{{d^{(i)}}^T r^{(i)}}{{d^{(i)}}^T A d^{(i)}} . \tag{3.81}$$

Note that this A-orthogonality condition (3.81) is equivalent to searching for the minimum along a given search direction (here $d^{(i)}$), as we did it in the method of steepest descent:

$$0 \;=\; \frac{\partial f}{\partial \alpha_i}(x^{(i+1)}) \;=\; f'(x^{(i+1)}) \frac{\partial x^{(i+1)}}{\partial \alpha_i} \;=\; -\,{r^{(i+1)}}^T d^{(i)} \;=\; {d^{(i)}}^T A e^{(i+1)} .$$

In contrast to (3.80), all expressions in (3.81) are known and can be calculated. Thus, (3.79) and (3.81) lead us to the following algorithm, still without a rule how to construct the (conjugate) search directions $d^{(i)}$. We start with $d^{(0)} := r^{(0)} = b - A x^{(0)}$ and iterate for $i = 0, 1, \ldots$:

$$\begin{aligned} \alpha_i &:= \frac{{d^{(i)}}^T r^{(i)}}{{d^{(i)}}^T A d^{(i)}} , \\ x^{(i+1)} &:= x^{(i)} + \alpha_i d^{(i)} , \\ r^{(i+1)} &:= r^{(i)} - \alpha_i A d^{(i)} . \end{aligned} \tag{3.82}$$

Compare (3.74) on the one hand with (3.82) on the other hand to see the close relations between steepest descent and our new method of conjugate directions.

What about convergence? Since our conjugate directions $d^{(i)}$ are linearly independent, we can represent the starting error $e^{(0)}$ as a linear combination of $d^{(0)}, \ldots, d^{(n-1)}$,

$$e^{(0)} \;=\; \sum_{j=0}^{n-1} \beta_j d^{(j)} \quad \text{with} \quad \beta_j \;=\; \frac{{d^{(j)}}^T A e^{(0)}}{{d^{(j)}}^T A d^{(j)}} , \tag{3.83}$$

which can be easily verified (multiply both sides with ${d^{(k)}}^T A$ from the left). Because of the A-orthogonality of the $d^{(i)}$, we can add the first $j-1$ corrections to $e^{(0)}$ and get with (3.81)

$$\beta_j \;=\; \frac{{d^{(j)}}^T A e^{(0)}}{{d^{(j)}}^T A d^{(j)}} \;=\; \frac{{d^{(j)}}^T A (e^{(0)} + \sum_{k=0}^{j-1} \alpha_k d^{(k)})}{{d^{(j)}}^T A d^{(j)}} \;=\; \frac{{d^{(j)}}^T A e^{(j)}}{{d^{(j)}}^T A d^{(j)}} \;=\; -\,\alpha_j . \tag{3.84}$$

From (3.79) and (3.84), we learn that step i eliminates exactly the starting error's $d^{(i-1)}$-component, which leads us to the minimum x after at most n steps. For the energy norm $\|e^{(i)}\|_A$, this means that

$$\|e^{(i)}\|_A^2 \;=\; {e^{(i)}}^T A e^{(i)} \;=\; \sum_{j=i}^{n-1} \sum_{k=i}^{n-1} \alpha_j \alpha_k {d^{(j)}}^T A d^{(k)} \;=\; \sum_{j=i}^{n-1} \alpha_j^2 {d^{(j)}}^T A d^{(j)} . \tag{3.85}$$

Apart from A-orthogonality, note that there is a standard orthogonality relation between the conjugate directions $d^{(i)}$ and the residuals $r^{(i)}$:

$${d^{(i)}}^T r^{(j)} \;=\; -\,{d^{(i)}}^T A e^{(j)} \;=\; {d^{(i)}}^T A \sum_{k=j}^{n-1} \alpha_k d^{(k)} \;=\; 0 \quad \text{for} \quad j > i . \tag{3.86}$$

Finally, we must discuss the construction of the conjugate directions $d^{(i)}$. The most obvious choice would be A's eigenvectors $v^{(i)}$. Since A is symmetric positive definite, the $v^{(i)}$ are orthogonal and, thus, A-orthogonal:

$$v^{(i)\,T} A v^{(j)} \;=\; \lambda_j v^{(i)\,T} v^{(j)} \;=\; 0 \qquad \text{for} \quad i \neq j\,.$$

However, computing all $v^{(i)}$ is even a harder job than solving $Ax = b$ and, hence, is not feasible. Another possiblity is a so-called *Gram-Schmidt A-orthogonalization* or *Gram-Schmidt conjugation*, which is completely analogous to the well-known standard Gram-Schmidt orthogonalization (start from some basis $u^{(0)}, \ldots, u^{(n-1)}$ and subtract from $u^{(i)}$ all components that are not A-orthogonal to the conjugate directions $d^{(0)}, \ldots, d^{(i-1)}$ constructed before $(i = 0, \ldots, n-1)$). Obviously, this is not very attractive: all of the search vectors $d^{(i)}$ constructed so far have to be kept in memory at all times, and, in addition to that, $\mathcal{O}(n^3)$ arithmetic operations are necessary to generate all of them. Actually, it can be shown that the method of conjugate directions is equivalent to standard Gaussian elimination, if we take the axial unit vectors as our starting basis for the Gram-Schmidt A-orthogonalization process mentioned above. Consequently, we have to look for a more efficient way to construct conjugate directions.

Conjugate Gradients

If the unit vectors don't work sufficiently, why not take the residuals $r^{(i)}$ as our vectors to be conjugated? The residuals have proven to be feasible as search directions in the method of steepest descent, and, according to (3.86), $r^{(i)}$ is orthogonal to all former search directions, unless it is zero (and the problem, thus, is solved).

For getting an idea why this choice is much more efficient than conjugating the axial unit vectors, we must have a closer look at its consequences. Define

$$\mathcal{D}_i \;:=\; \operatorname{span}\left\{d^{(0)}, \ldots, d^{(i-1)}\right\} \tag{3.87}$$

as the (never again visited) subspace of our first i corrections. Since we conjugate the residuals in order to get our search directions, $\mathcal{D}_i$ is spanned by $r^{(0)}, \ldots, r^{(i-1)}$, too. Hence, (3.86) says

$$r^{(i)\,T} r^{(j)} \;=\; 0 \quad \text{for} \quad i \neq j\,, \tag{3.88}$$

i. e., the residuals are orthogonal. Furthermore,

$$r^{(i)} \;=\; -Ae^{(i)} \;=\; -A(e^{(i-1)} + \alpha_{i-1} d^{(i-1)}) \;=\; r^{(i-1)} - \alpha_{i-1} A d^{(i-1)} \tag{3.89}$$

shows that $r^{(i)}$ is just a linear combination of the previous residual $r^{(i-1)}$ and $Ad^{(i-1)}$, where $d^{(i-1)} \in \mathcal{D}_i$. Thus, the $\mathcal{D}_i$ are Krylov subspaces,

$$\mathcal{D}_i \;=\; \operatorname{span}\left\{d^{(0)}, Ad^{(0)}, \ldots, A^{i-1} d^{(0)}\right\} \;=\; \operatorname{span}\left\{r^{(0)}, Ar^{(0)}, \ldots, A^{i-1} r^{(0)}\right\}\,, \tag{3.90}$$

and (3.86) and $A\mathcal{D}_i \subset \mathcal{D}_{i+1}$ imply that $r^{(i+1)}$ is not only orthogonal to $\mathcal{D}_{i+1}$, but also A-orthogonal to $\mathcal{D}_i$. This makes the step of the A-orthogonalization very simple, since conjugation of $r^{(i+1)}$ has only to be done explicitly with respect to $d^{(i)}$.

Now we are ready for the CG algorithm. We start with $d^{(0)} := r^{(0)} = b - Ax^{(0)}$ and iterate for

$i = 0, 1, \ldots$:

$$\begin{aligned} \alpha_i &:= \frac{{r^{(i)}}^T r^{(i)}}{{d^{(i)}}^T A d^{(i)}}, \\ x^{(i+1)} &:= x^{(i)} + \alpha_i d^{(i)}, \\ r^{(i+1)} &:= r^{(i)} - \alpha_i A d^{(i)}, \\ \beta_{i+1} &:= \frac{{r^{(i+1)}}^T r^{(i+1)}}{{r^{(i)}}^T r^{(i)}}, \\ d^{(i+1)} &:= r^{(i+1)} + \beta_{i+1} d^{(i)}. \end{aligned} \tag{3.91}$$

Two comments: First, the calculation of α_i profits from ${d^{(i)}}^T r^{(i)} = {r^{(i)}}^T r^{(i)}$ (an immediate consequence of the Gram-Schmidt process) and, thus, avoids a further scalar product. Second, the parameter β_{i+1} has entered the scene for realizing the conjugation of $r^{(i+1)}$ and $d^{(i)}$ (see a detailed description of the Gram-Schmidt process for a derivation).

What about convergence behaviour? In principle, CG is a direct solver. However, numerical problems (roundoff and cancellation errors, especially disturbing the A-orthogonality of our search directions), and the fact that n is usually too large in problems of modern numerical simulation to allow n steps of iteration are responsible for its primary use as an iterative scheme, today. This is not the place for a detailed convergence analysis, but its results are important, nevertheless. Concerning the energy norm of the error after i steps, the following bound can be given:

$$\|e^{(i)}\|_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa + 1}} \right)^i \|e^{(0)}\|_A, \tag{3.92}$$

where κ again denotes A's spectral condition number. Compare this with the corresponding result (3.76) for the method of steepest descent. For $\kappa = 100$, there, the factor of reduction of each step of the iteration was about 0.99, and it is about 0.82 for CG. Or, in other words, suppose that we want to reduce the starting error $e^{(0)}$ by a factor ε, i. e., we look for the smallest i that ensures $\|e^{(i)}\|_A \leq \varepsilon \|e^{(0)}\|_A$. A short calculation based on (3.76) and (3.92), resp., shows

$$\begin{aligned} i &\leq \frac{1}{2} \kappa \ln\left(\varepsilon^{-1}\right) \qquad \text{(steepest descent)}, \\ i &\leq \frac{1}{2} \sqrt{\kappa} \ln\left(2\varepsilon^{-1}\right) \qquad \text{(CG)}. \end{aligned} \tag{3.93}$$

What does that mean in (our) reality? Standard finite difference or piecewise linear finite element discretizations of second order elliptic boundary value problems in d dimensions often lead to an $\mathcal{O}(n^{2/d})$- or $\mathcal{O}(h^{-2})$-behaviour of κ, if h denotes the characteristic mesh width of the underlying grid of n grid points. Hence, we have a time complexity of $\mathcal{O}(h^{-2})$ for steepest descent versus $\mathcal{O}(h^{-1})$ for CG, which means that, of course, the Krylov subspace method CG is a significant improvement compared to the simple method of steepest descent. However, it is evident, too, that we can not stop here in developing solvers for large sparse linear systems: If we increase the accuracy of our discretization by refining the mesh and, thus, reducing h, then the number of iterations still increases quickly.

Preconditioning

As we have seen in the previous section, the aspect that limits the efficiency of the CG algorithm (and that of Krylov subspace methods in general, by the way) is the spectral condition number κ of the system matrix A. A remedy is *preconditioning*, where we apply a *preconditioner* M to the system in order to improve (i. e. to reduce) the condition of the new system matrix $M^{-1}A$:

$$M^{-1} A x = M^{-1} b. \tag{3.94}$$

Note that a reduction of the spectral condition number κ means a better clustering of the spectrum of the resulting matrix. However, even if both M and A are symmetric positive definite, the product $M^{-1}A$ in general is not. Nevertheless, it is helpful to restrict oneself to a symmetric positive definite M. For, in this case, with M's Cholesky factorization $WW^T = M$, the matrices $M^{-1}A$ and $W^{-1}AW^{-T}$ are similar, and we can tackle the solution of

$$W^{-1}AW^{-T}y \;=\; W^{-1}b\,, \qquad y \;=\; W^T x\,, \tag{3.95}$$

(first for y, then for x) with a symmetric positive definite system matrix $W^{-1}AW^{-T}$ instead of the original system or (3.94). Now, we can reformulate the CG algorithm (3.91) for $W^{-1}AW^{-T}$ (which means that we need not only M, but also its above factorization). Fortunately, some skilful variable substitutions can make W disappear from the algorithm, and we get the following preconditioned version of (3.91), now starting with $r^{(0)} = b - Ax^{(0)}$ and $d^{(0)} := M^{-1}r^{(0)}$ and iterating for $i = 0, 1, \ldots$:

$$\begin{aligned}
\alpha_i &:= \frac{{r^{(i)}}^T M^{-1} r^{(i)}}{{d^{(i)}}^T A d^{(i)}}\,,\\
x^{(i+1)} &:= x^{(i)} + \alpha_i d^{(i)}\,,\\
r^{(i+1)} &:= r^{(i)} - \alpha_i A d^{(i)}\,,\\
\beta_{i+1} &:= \frac{{r^{(i+1)}}^T M^{-1} r^{(i+1)}}{{r^{(i)}}^T M^{-1} r^{(i)}}\,,\\
d^{(i+1)} &:= M^{-1} r^{(i+1)} + \beta_{i+1} d^{(i)}\,.
\end{aligned} \tag{3.96}$$

As we can see, it is not even necessary to compute M or M^{-1} explicitly. All we need is to apply M^{-1} to a vector, i. e. to form the matrix vector product with M^{-1}.

Obviously, the ideal preconditioner M is the original matrix A itself: The spectrum of the resulting system matrix I is narrowed to 1, and one step leads us to the solution. However, constructing this preconditioner is equivalent to solving the original problem. So nothing has been gained. On the other hand, we could try our 'approximations' M to A from (3.61), e. g. $M := D_A$ in the simplest (Jacobi) case or $M := D_A + L_A$ for Gauß-Seidel. The first, called *diagonal* or *Jacobi* preconditioner, is very cheap, but without a too impressive performance. Due to lack of symmetry of M, Gauß-Seidel and SOR are not used for preconditioning. However, a symmetric version called *SSOR* consisting of two successive SOR steps with reversed roles of L_A and U_A is quite widespread:

$$M^{(1/2)} \;:=\; \frac{1}{\alpha}D_A + L_A\,, \qquad M^{(1)} \;:=\; \frac{1}{\alpha}D_A + U_A\,, \tag{3.97}$$

with a resulting symmetric overall preconditioner

$$M \;:=\; \frac{\alpha}{2-\alpha}\left(\frac{1}{\alpha}D_A + L_A\right)^{-1} D_A^{-1}\left(\frac{1}{\alpha}D_A + U_A\right)\,. \tag{3.98}$$

The potential that lies between $M := A$ and $M := D_A$ is huge, and we just want to mention some principles of deriving efficient preconditioners.

- **Incomplete factorizations**:
  If an explicit factorization of A and, thus, the direct solution of $Ax = b$ are prohibitive, an approximate or *incomplete* factorization of A may be a good approximation to A and, thus, suitable as a preconditioner. In the symmetric positive definite case, we speak of *incomplete Cholesky factorizations*, in the general case, the term is *incomplete LU* or *ILU factorizations*. Instead of the classical and exact factorization $A = LL^T$ or $A = LU$, resp., approximate and sparse factors $\tilde{L}$ $\tilde{U}$ are used for the definiton of M:

$$M \;:=\; \tilde{L}\tilde{L}^T \qquad \text{or} \qquad M \;:=\; \tilde{L}\tilde{U}\,. \tag{3.99}$$

The product $z := M^{-1}b$ is done solving the two sparse systems $\tilde{L}w = b$ and $\tilde{L}^T z = w$ or $\tilde{U}z = w$, resp. Two prominent representatives are

$$\begin{aligned} M &:= (L_A + D_A)\, D_A^{-1} \left(L_A^T + D_A\right) , \\ M &:= (L_A + D_A)\, D_A^{-1} \left(U_A + D_A\right) , \end{aligned} \tag{3.100}$$

where L_A, D_A, and U_A denote the lower triangular, diagonal, and upper triangular part of A, respectively. Note that this is just the above-mentioned SSOR preconditioner for $\alpha = 1$.

- **Sparse approximate inverses (SPAI)**:
  It is a well-known fact in numerical linear algebra that computing the inverse A^{-1} is no reasonable way to solve linear systems $Ax = b$. For sparse matrices, in addition to that, an explicit matrix factorization like LU usually leads to factors that are structurally dense. One way to overcome this are factorizations of the ILU type. Another possibility is to approximate A^{-1} directly with the help of a sparse matrix B, i. e. to solve

  $$\min_B \|I - AB\|^2 \tag{3.101}$$

  for some matrix norm, for example the Frobenius norm

  $$\|A\|_F := \sqrt{\sum_{i,j=1}^{n} a_{i,j}^2} ,$$

  where B is of some fully or partially prescribed sparsity pattern, and to take this sparse approximate inverse B for preconditioning, i. e. $M^{-1} := B$. One of the main reasons why sparse approximate inverse preconditioners may be advantageous is the fact that, usually, ILU preconditioners are difficult to parallelize in an efficient way, whereas SPAI-based preconditioners are often very well-suited for parallelization.

- **Polynomial preconditioners**:
  If you represent A^{-1} as a series (formally, write $A = I - (I - A)$ and take the geometric series), it becomes evident that polynomials in A can be used to approximate A^{-1}. Though Krylov subspace methods already look for solutions in polynomial spaces of A and though it is, thus, not so clear why an additional polynomial used as a preconditioner might improve the situation, it is, often, again the parallel argument that counts.

- **Multilevel preconditioners**:
  The perhaps most sophisticated way to construct preconditioners is to put hierarchical or *multilevel* information into it. The most prominent examples, like the famous BPX preconditioner, are based on projection arguments. A short introduction to multilevel or multigrid methods (there as an explicit solver, not as a preconditioner) will be given in the next section.

3.5.3 Multigrid Methods

As we have learned in (3.64) and in the previous section, both classical relaxation techniques and Krylov subspace methods suffer from the fact that an improved resolution of the underlying grid leads to a significant slow-down concerning the speed of convergence, i. e. the number of iterations that are necessary to obtain the solution of the discrete system. Whereas preconditioning offers possibilitites for a remedy in the Krylov case, the *multigrid* principle can take the part of an accelerator for Jacobi or Gauß-Seidel iterations.

A more detailed analysis of what happens with standard relaxation techniques provides the so-called *Fourier mode analysis*. Here, as during Fourier transform, the error is decomposed into its components of different frequencies. It can be shown that, typically, the above-mentioned standard relaxation techniques reduce the highly oscillating (i. e. high-frequency) components quite quickly,

whereas the low-frequency part decreases only very slowly (actually, the more slowly the finer the grid is). That's why those methods are sometimes called *smoothers*. Figure 3.10 illustrates this phenomenon for a very simple example. Take the one-dimensional Laplace equation (i. e. $u''(x) = 0$) on the unit interval with Dirichlet boundary conditions $u(0) = u(1) = 0$ and a random starting approximation (which is just the error itself, since $u(x) \equiv 0$ is our solution). On the equidistant grid of mesh width $h = 2^{-6}$, use a finite difference discretization with the standard three-point stencil, and apply a damped Jacobi iteration with damping parameter $\alpha = 0.5$. After 100 iterations (which is quite much for such a simple problem), the originally rough error has become very smooth, but it is still rather large. As it can be seen in Fig. 3.10, this is mainly due to the low-frequency error components. However, with a more clever multigrid method to be discussed below, a homogeneous and fast error reduction is possible (look at the curve with tiny oscillations around zero which shows the error after only two multigrid iterations).

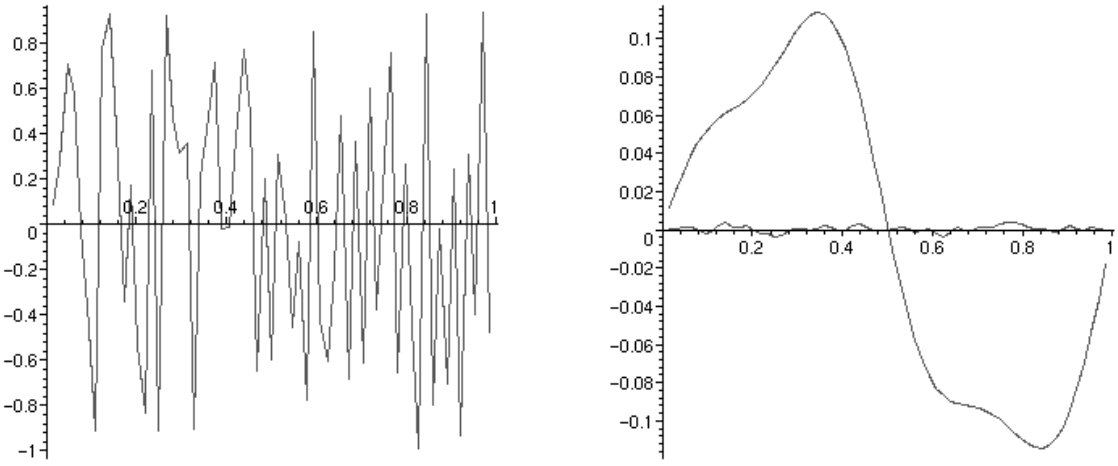


Abbildung 3.10: Error for the damped Jacobi iteration ($\alpha = 0.5$, matrix stemming from the 1 D standard finite difference stencil of the second derivative u'', mesh width $h = 2^{-6}$, random starting values): At the beginning, there are both high- and low-frequency error components (left); after 100 iterations, the high-frequency part has nearly disappeared, whereas the low-frequency components are still large (right, high-amplitude curve). Compare this with the error that is the result of just two multigrid steps (right, tiny curve).

So what is the consequence of the above observation? Obviously, a fine grid has problems with long-wave error components, but can reduce short-wave error components quite well (as long as they can be represented on the grid at all, of course). Thus, since the terms 'long-wave' and 'short-wave' are relative ones and depend on the mesh width of the underlying grid, all we have to do in order to tackle the long-wave error components effectively is to move on to a coarser grid. Fortunately, this is a very economic decision, too, since working on a coarse grid is much cheaper due to the smaller number of grid points. If we repeat this use of coarser grids, we come quite close to the essential idea of multigrid or *multilevel* techniques.

At that point, one might now think of discretizing the given PDE and solving the resulting systems in an independent way on several grids of different resolution. However, in each single solution, there is only part of the error reduction done, and the question remains what to do with the different and non-related solutions. Hence, a *correction-type* scheme, where the computation on the coarser grid is used as a correction to the result obtained on the respective finer grid, turns out to be the better choice. Before explaining real multigrid, we discuss the simpler two-grid case. Suppose we have an approximate solution x_l on level l (i. e. with mesh width $h_l = 2^{-l}$, for

example), and suppose that A_l, b_l, and Ω_l denote the corresponding matrix, right-hand side, and grid. Note that, in the regular equidistant case, $h_{l-1} = 2^{-(l-1)}$, usually, i. e., the mesh width is increased by a factor of 2, and the number of grid points is reduced by a factor of 2^d, where d denotes the dimensionality of the problem. For irregular or unstructured grids, the coarsening process is controlled in a way that we get a comparable reduction of the number of unknowns from the fine to the next coarser level. Starting from these prerequisites, the following algorithm provides a *coarse grid correction* e_{l-1} that can be used to update and improve the original fine grid solution x_l:

$$\begin{aligned}
&\text{smooth the current solution } x_l;\\
&\text{form the residual } r_l := b_l - A_l x_l;\\
&\text{restrict } r_l \text{ to the coarse grid } \Omega_{l-1};\\
&\text{provide a solution to } A_{l-1} e_{l-1} = r_{l-1};\\
&\text{prolongate } e_{l-1} \text{ to the fine grid } \Omega_l;\\
&\text{add the resulting correction to } x_l \text{ (perhaps, smooth again);}
\end{aligned} \tag{3.102}$$

Let us have a look at the different modules the above algorithm consists of:

- The **pre-smoothing**: Its task is to smooth the error, i. e. to reduce high-frequency components and, thus, to prepare the error or the residual for the transfer to the coarse grid. Usually, multigrid algorithms use a small number of damped Jacobi or Gauß-Seidel steps, here.
- The **restriction**: The most obvious and simple choice is to use *injection*, i. e. to take the fine grid values at the coarse grid points and to forget the values at the other points of Ω_l. A more sophisticated strategy is *(full) weighting*, where the coarse values are formed by an averaging process of the function's values in this point itself and in the neighbouring fine grid points. This is more natural, since it can be seen as the inverse process to interpolation (see Fig. 3.11).
- The **coarse grid solution**: Obviously, it is useful to solve the above-mentioned coarse grid equation $A_{l-1} e_{l-1} = r_{l-1}$: If we did this on the fine grid Ω_l, the respective correction would result in the exact solution in one step due to

  $$A_l (x_l + e_l) \;=\; A x_l + A e_l \;=\; b_l - r_l + r_l \;=\; b_l \,. \tag{3.103}$$

  For the solution of the coarse grid equation, there are several possibilities: If Ω_{l-1} is coarse enough and if, thus, the resulting linear system is small enough, then a direct solution can be realized. Otherwise, we can apply some smoothing steps, or we can repeat the coarse grid correction recursively and move on to some Ω_{l-2}. This latter choice leads us from the *two-grid* to the *multigrid* algorithm.
- The **prolongation**: When the coarse grid equation is solved, the computed correction e_{l-1} has to be transferred back to the fine grid Ω_l. This is done by some interpolation or prolongation process that, usually, just takes the coarse grid values as they are and forms an average of the values in the coarse grid points neighbouring the points of $\Omega_l \setminus \Omega_{l-1}$ (see Fig. 3.11).
- The **post-smoothing**: Sometimes, it is reasonable to do some more smoothing at the end in order to avoid new low-frequency components.

If applied to a hierarchy of grids Ω_k, $k = l, \ldots, 1$, the coarse grid solution step can be done by a coarse grid correction itself, now on Ω_{l-2}, and so on, until one reaches the coarsest grid Ω_1 of the sequence. Here, both a direct solution or some smoothing steps are popular, depending on the

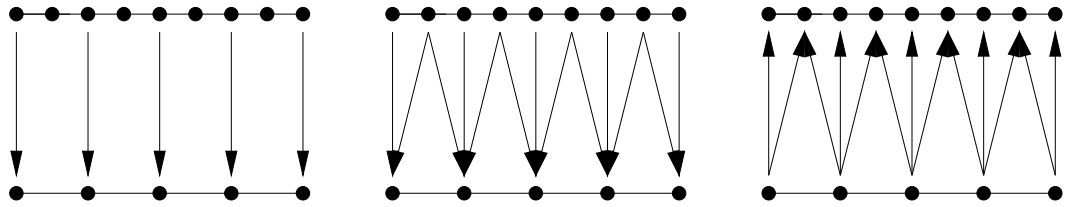


Abbildung 3.11: The processes of restriction (injection (left) or full weighting (centre)) and prolongation (right) between two grids Ω_l and Ω_{l-1} in 1 D

corresponding problem's size. In this recursive or multigrid case, the algorithmic scheme of (3.102) is called a *V-cycle*. Of course, there are other potential schemes. The *W-cycle*, for example, is sometimes advantageous with respect to speed of convergence. For an explanation of the names, see Fig. 3.12.

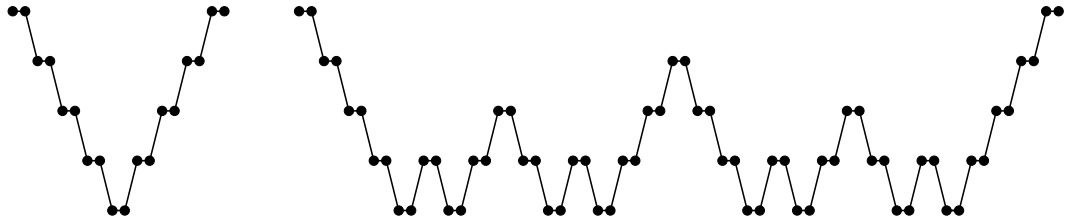


Abbildung 3.12: V-cycle (left) and W-cycle (right): two famous multigrid representatives (top: finest grid Ω_l, bottom: coarsest grid Ω_1)

Finally, one might think of profiting from the multigrid idea also in order to produce starting approximations u_l. In fact, this is possible and realized in the so-called *nested iteration*. There, the following scheme is applied:

$$\begin{aligned}
&\text{smooth or solve the problem on the coarsest grid } \Omega_1;\\
&\text{interpolate the obtained } u_1 \text{ to } \Omega_2;\\
&\text{smooth on } \Omega_2;\\
&\text{interpolate the obtained } u_2 \text{ to } \Omega_3;\\
&\text{smooth on } \Omega_3;\\
&\vdots\\
&\text{get a starting solution on the finest grid } \Omega_l;
\end{aligned} \tag{3.104}$$

Furthermore, if the scheme of (3.104) is applied not only to get a starting solution, but to calculate multigrid solutions on each level (replace 'smooth' by 'solve' in (3.104)), then the result is an algorithm that ends up not only with a starting solution, but with a solution to our problem (see Fig. 3.13).

Finally, we have to mention two important features of multigrid methods:

- First, the amount of work is dominated by the finest level. If C denotes the number of operations for one smoothing step on Ω_l, then $C/2$ or $C/4$ or $C/8$ operations are necessary on Ω_{l-1} in the 1 D, 2 D, or 3 D case, if we halven the number of grid points in each coarsening step. Thus, we get a geometric series for the overall smoothing cost (as well as for restriction

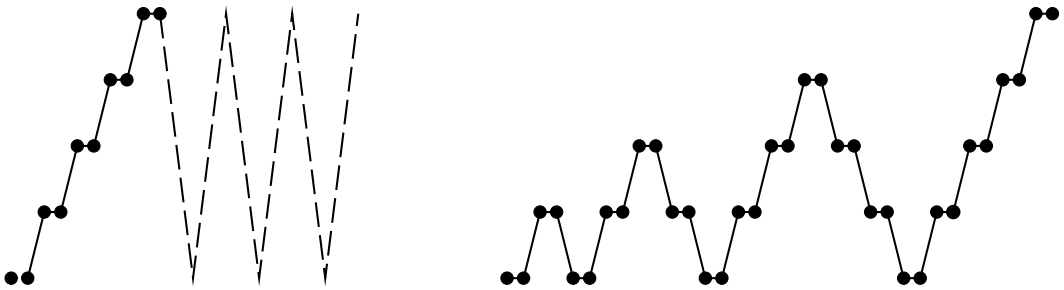


Abbildung 3.13: Nested iteration to get a starting solution on the fine grid, followed by several V-cycles (left) and full multigrid scheme (right; top: finest grid Ω_l, bottom: coarsest grid Ω_1)

and prolongation etc.) and end up with a total of kC operations for one whole V-cycle with some small constant k. The same holds for the memory requirements, by the way.

- Concerning the speed of convergence, in many cases, multigrid methods show the ideal $\gamma = O(1-c)$ behaviour already mentioned in (3.55). I. e., the reduction γ of the error per step of the iteration is bounded by some constant smaller than 1 (and often very small, like 0.1). Thus, γ does not depend on the resolution of the grid (or depends on it only very slightly), and convergence does not slow down (or slows down only slightly) if we increase the accuracy of our discretization.

Today, the multigrid idea is used in a much more general way than the original and basic scheme described here does. First of all, it is important to mention that multigrid methods are not restricted to the linear case. The so-called *full approximation scheme (FAS)* (where both the residual and the current solution are transferred to the coarser grid) provides an algorithmic framework for systems of nonlinear equations, too. Moreover, *multilevel preconditioners* use the presence of a hierarchy of levels in order to construct optimal (with respect to the resulting spectral condition number) preconditioners. *Multiresolution approaches* use the related wavelet theory to obtain fast solvers. Finally, *algebraic multigrid methods* do without the notion of underlying grids (i. e. without a *geometric* point of view) and construct coarse grid operators, restrictions, and prolongations just based upon the fine grid operator or matrix A_l. One important advantage is that the multigrid acceleration can be used for the solution of linear systems that do not stem from the discretization of a PDE, too.

With this short presentation of the multigrid idea, we close the discussion of iterative solvers for linear systems and turn to methods for systems of nonlinear equations.

3.6 Nonlinear Equations, Linearization

The task of solving a linear system of equations $Ax = b$ can be interpreted as the problem to find the root of some linear function $F(x) := b - Ax$, $F : \mathbb{R}^N \to \mathbb{R}^N$. Hence, the natural generalization is to allow arbitrary (especially nonlinear) functions $F(x)$ and to look for their root(s). This is important, since many PDEs occurring in mathematical models are nonlinear (think of the term $(\vec{u} \cdot \mathrm{grad})\vec{u}$ in the Navier-Stokes equations in fluid dynamics, for example) and, after discretization, lead to systems of nonlinear equations.

In the following, let f_i denote the ith vector component of F. If all f_i are differentiable in some $x \in \mathbb{R}^N$, we define the *Jacobian matrix* $F'(x)$ as

$$(F'(x))_{i,j} := \left(\frac{\partial f_i}{\partial x_j}(x) \right)_{i,j} . \tag{3.105}$$

As in the linear case (cf. (3.53)), an iterative method produces a sequence $x^{(i)}, i = 0, 1, \ldots$, that (hopefully) converges to the exact solution or root denoted by $\bar{x}$. In addition to the mere decision of convergence as given in (3.54), we distinguish the quality or speed of convergence with the help of its order and write, in general,

$$\|x^{(i+1)} - \bar{x}\| \;\leq\; c\|x^{(i)} - \bar{x}\|^{\alpha}\,. \tag{3.106}$$

Now, an iterative method is called *locally linearly* or *locally quadratically* convergent (and so on), if the produced $x^{(i)}$ fulfil (3.106) with $\alpha = 1$ and $0 < c < 1$ or $\alpha = 2$, respectively, if the starting vector $x^{(0)}$ is sufficiently good (or close to $\bar{x}$).

Many nonlinear systems are naturally formulated as *fixed-point problems*

$$x \;:=\; \Phi(x) \tag{3.107}$$

with a nonlinear mapping $\Phi : \mathbb{R}^N \to \mathbb{R}^N$. Any solution $\bar{x}$ of (3.107) is called a *fixed point* of Φ (once the iteration has reached $\bar{x}$, it is trapped and won't leave $\bar{x}$ any more). Note that our derivation of linear iterative schemes in (3.58) in the previous section was nothing else but a (linear) fixed-point formulation:

$$M\bar{x} + (A - M)\bar{x} \;=\; b \quad\Leftrightarrow\quad \bar{x} \;=\; M^{-1}\left(b - (A - M)\bar{x}\right)\,. \tag{3.108}$$

A straightforward (and important) iterative method for fixed-point problems is the *fixed-point iteration*

$$x^{(i+1)} \;:=\; \Phi(x^{(i)})\,, \tag{3.109}$$

which is also called *Richardson* or *Picard* iteration. The most important result for this type of iteration is the *fixed-point theorem* or *contraction mapping theorem* of Banach and Weissinger. It says that under certain assumptions ($\Phi : U \to U$ for some closed $U \subseteq \mathbb{R}^n$ is a contraction mapping, i. e. Lipschitz continuous[6] with a Lipschitz constant smaller 1), there exists a unique fixed point $\bar{x}$ of Φ in U, and the fixed-point iteration (3.109) converges linearly with with reduction factor $c := \gamma$ for all starting vectors $x^{(0)} \in U$.

A simple application of the fixed-point iteration and of the above theorem is the numerical integration of ODEs by implicit methods. If we apply a backward Euler method with time step h to the initial value problem $y' = f(y)$, $y(t_0) = y_0$, we get

$$y^{k+1} - y^k \;=\; hf(y^{k+1}) \quad \text{or} \quad y = \Phi(y) \;:=\; y^k + hf(y)\,.$$

If $f(x)$ is bounded and Lipschitz continuous, say $|f(x)| \leq C$ and $|f(x) - f(y)| \leq C|x - y|$, then

$$\|\Phi(x) - \Phi(y)\| \;=\; h|f(x) - f(y)| \;\leq\; hC|x - y|\,.$$

Thus, for a sufficiently small time step h, we can integrate forward in time, and the process will converge.

In practice, the simple fixed-point iteration does not perform sufficiently well (take the demand for small time steps in the above problem as an example). Apart from nonlinear variants of classical relaxation methods, Newton's method and related techniques are very popular. But before we turn to their discussion, we make some assumptions. For the following, we assume that

- $F(x) = 0$ has a solution $\bar{x}$,
- $F' : U \to \mathbb{R}^{n,n}$ is Lipschitz continuous with Lipschitz constant γ,
- $F'(\bar{x})$ is not singular.

.

[6] $G : \mathbb{R}^n \supseteq U \to \mathbb{R}^m$ is called *Lipschitz continuous with Lipschitz constant γ on U*, if $\|G(x) - G(y)\| \leq \gamma\|x - y\|$ for all $x, y \in U$.

3.6.1 Newton's Method

Newton's method uses the root of F's two-term Taylor expansion around the current iterate $x^{(i)}$,

$$F(x^{(i)}) + F'(x^{(i)})(x - x^{(i)}),$$

to construct the new iterate $x^{(i+1)}$. Thus, it reads

$$x^{(i+1)} := x^{(i)} - F'(x^{(i)})^{-1} F(x^{(i)}). \tag{3.110}$$

Under the above standard assumptions, locally quadratic convergence can be shown. It is important to keep in mind that all we can get with Newton's method is *lcoal* convergence. I. e., we need a starting point $x^{(0)}$ which is already quite close to $\bar{x}$. However, this can often be ensured.

There are several possibilities to get a stopping criterion. The simplest one (based directly on the quadratic convergence) says to terminate the iteration when $\|x^{(i+1)} - x^{(i)}\|$ becomes sufficiently small. Another rule could be to stop iterating when the relative nonlinear residual $\|F(x^{(i)})\|/\|F(x^{(0)})\|$ becomes sufficiently small. Here, we might get problems if our initial guess was already too good, which could entail a very long iteration.

As we can see in (3.110), the classical Newton iteration requires the exact, i. e. direct solution of a linear system (remember that F' is a matrix!) in each step. And before that, the partial derivatives (the entries of the Jacobian matrix) have to be provided (in some rare cases analytically, ususally approximately via finite differences, sometimes also in an elegant way via automatic differentiation[7]). Thus, an implementation of (3.110) may say to repeat the following steps as long as the stopping criterion allows to do this:

$$\begin{aligned}
&\text{compute } F'(x);\\
&\text{factorize } F'(x) := LU;\\
&\text{solve } LUs = -F(x);\\
&\text{update } x := x + s;\\
&\text{evaluate } F(x);
\end{aligned} \tag{3.111}$$

A cheaper alternative is the so-called *chord method*. Here, the Jacobian matrix is constructed and inverted once for the whole iteration, thus replacing (3.110) by

$$x^{(i+1)} := x^{(i)} - F'(x^{(0)})^{-1} F(x^{(i)}). \tag{3.112}$$

The resulting iterative process can be shown to be locally linearly convergent. A variant, sometimes named *Shamanskii method*, is based upon an alternation of a Newton step with some chord steps, i. e. it uses a Jacobian and its factorization for a certain number of steps and computes a new one afterwards. Of course, such simplifications create new sources of error, since not the real Jacobian, but some approximation is used. Yet there are even more possible simplifications or sources of error:

- errors occurring during the computation of $F(x)$ or $F'(x)$ (roundoff errors);
- errors occurring due to the use of an *approximate inverse* of the Jacobian instead of factorizing it;
- errors due to a difference approximation to the partial derivatives in the Jacobian.

For those situations, theoretical results are available. In some cases, we only get linear convergence (the chord method, for example), in others, we see quadratic convergence at the beginning, but may run into stagnation where no further improvement is possible (difference approximations).

[7] Automatic or algorithmic differentiation tries to avoid approximations of derivatives of F by an automatically or algorithmically controlled repeated use of the chain rule in the algorithms to compute F.

3.6.2 Inexact Newton Methods

Up to now, the linear system with the Jacobian has been solved directly, with the help of some factorization. For reasons of efficiency, of course, the question arises whether to use iterative methods instead, some *inner* iteration within the *outer* Newton iteration. Approaches following this idea are called *inexact Newton methods*. One usually demands

$$\|F'(x^{(i)})s + F(x^{(i)})\| \;\leq\; \eta_i \|F(x^{(i)})\| \,. \tag{3.113}$$

Since the smaller η_i is, the better the approximation to a direct solution of the respective linear system is, the overall convergence behaviour depends on the choice of the sequence η_i. If all η_i are bounded by a sufficiently small constant, then we get linear convergence. If the η_i converge to zero (or fulfil comparable restrictions), then even superlinear convergence is possible.

Now, any iterative scheme discussed in Sect. 3.5 can be used to construct an update s to $x^{(i)}$ that fulfils (3.113). Hence, there are Newton-SOR, Newton-multigrid, Newton-CG, and Newton-GMRES (or, in general, Newton-Krylov) approaches. Note that the inner iteration sometimes requires additional evaluations of F, besides the iterates $x^{(i)}$. Today, inexact Newton methods are very popular for the iterative solution of systems of nonlinear equations.

3.6.3 Quasi-Newton Methods

Another strategy to reduce computational effort during Newton's method was introduced by Broyden. His method is the prototype of the so-called *quasi-Newton methods*. Here, the idea of an *update* that we have already seen for the approximation $x^{(i)}$ to the solution $\bar{x}$ is applied to the Jacobian, too. Thus, the general scheme is

$$x^{(i+1)} \;:=\; x^{(i)} + s \;=\; x^{(i)} - \left(B^{(i)}\right)^{-1} F(x^{(i)}) \,, \tag{3.114}$$

where $B^{(i)}$ is the ith approximation to the Jacobian at $\bar{x}$, $F'(\bar{x})$. After the update of $x^{(i)}$, $B^{(i)}$ is updated itself to form $B^{(i+1)}$. Even the inverse can be updated, such that there is no need for a direct solution of a linear system or for an inner iteration. The crucial point is that updates of the approximate Jacobians are, usually, much cheaper to compute than the sequence of Jacobians $F'(x^{(i)})$ in all iterates $x^{(i)}$. Broyden's method, e. g., is locally superlinearly convergent, and, hence, an attractive alternative to Newton's method or to the chord method. Compared to inexact Newton methods, quasi-Newton methods avoid the work and function evaluations associated with the inner iteration. Thus, if a sufficiently good starting approximation $B^{(0)}$ to $F'(\bar{x})$ is available, the quasi-Newton approach may be more favourable than inexact Newton methods.

We do not want to go into detail, here, but just give the original *Broyden update* to $B^{(i)}$:

$$B^{(i+1)} \;:=\; B^{(i)} + \frac{F(x^{(i+1)})s^T}{s^T s} \,, \tag{3.115}$$

where s, as before, denotes the x-update $x^{(i+1)} - x^{(i)}$. Finally, we mention the basic result from linear algebra that helps to avoid to solve a linear system with $B^{(i+1)}$. If we have an invertible matrix B and two vectors u, v, than the *rank-1 update* $(B + uv^T)$ is invertible if and only if $1 + v^T B^{-1} u \neq 0$, and we get

$$\left(B + uv^T\right)^{-1} \;=\; \left(I - \frac{(B^{-1}u)v^T}{1 + v^T B^{-1} u}\right) B^{-1} \,.$$

A last remark shall close the discussion of methods for nonlinear equations. So far, we have only discussed *locally convergent* methods. Thus, the question arises how to get sufficiently good

starting approximations that ensure convergence. That is, there is some need for *globally convergent* methods, too. There are several approaches, the simplest of which is the *line search paradigm*. Here, the starting point is the observation that the search direction in Newton-type methods is usually fine (and pointing towards the root), but the length of the step is too big. Thus, it might be a good idea to develop a criterion to get a better length. Actually, based on this concept, it is possible to construct globally convergent Newton-based methods.

3.7 Grid Generation

The numerical treatment of partial differential equations requires the approximate description and discretization of the underlying computational domain. There exists a real zoo of strategies for generating and refining suitable *grids* or *meshes*. Of course, the selection of an appropriate strategy depends on the chosen discretization technique for the PDE (*point* discretizations, for example finite differences, or *cell* discretizations, for example finite volumes or finite elements) as well as on the complexity of the given geometry. Especially for real-life technical objects, it can take orders of magnitude more man-power to construct a grid than it does to compute and analyze the approximate solution on the grid. Therefore, grid generation is a very important issue for scientific computing. Hence, in this section, we want to give a short overview of some of the most relevant approaches for grid generation and mesh refinement.

During grid generation, a number of aspects have to be taken into account:

- **Accuracy**: The generated mesh must be (locally) dense enough to represent the essential physical phenomena.
- **Boundary approximation**: The generated mesh should allow a sufficiently accurate representation of the domain's boundaries and, thus, of the respective boundary conditions.
- **Computational efficiency**: The overhead for the organization of the grid structure (storage, CPU time) should not be too big, and the performance of the whole solution process on supercomputers (distributing the mesh!) should not suffer from the chosen mesh type.
- **Numerical adequacy**: Features with a negative impact on numerical efficiency (too small or big angles of the elements, too big distortions of elements or grids, etc.) should be avoided.

In general, we distinguish *structured* and *unstructured* grids. We speak of structured grids, if the points or elements are constructed following some regular process from which the geometric and topological information (coordinates, neighbour relationships) can be directly derived. In contrast to that, unstructured grids allow more flexibility, with grids points whose position might even result from a random process, but require the explicit storage of the grid's relevant geometric and topological information. However, in both situations, the generation of some starting grid is only part of the job, since, in many cases, the optimal position of grid points in the sense of a cost-benefit ratio can not be predicted *before* the solution process, but depends on the computed solution and, thus, has to take into account properties of it. Hence, dynamically adaptive meshes that are coupled with the physical solution are of crucial importance, too.

Finally, standard parallelization strategies for PDEs are based on some subdivision or decomposition of the underlying domain and, thus, of the respective grid. Therefore, the questions how easy an equal distribution of the grid points among the processors can be obtained, and how communication between the different subsets of grid points can be reduced (small interfaces, graph partitioning strategies) are important, too, but will not be discussed in detail.

3.7.1 Structured Grids

Composite Grids

The notion of *composite grids* was the key to the treatment of general 3 D geometries with the help of structured grids. The idea is just to subdivide the region of concern into a number of subdomains which are of a simpler form that allows the generation of a structured mesh. While the gain won't be that much for domains arising with porous media, e. g., it definitely is a breakthrough for many technical configurations, if one thinks of objects that have been created with the help of some CAD system and that are represented via the CSG (constructive solid geometry[8]) scheme, for example. There are several variants (see Fig. 3.14):

- **Block** or **patched** grids: Here, the different structured grids on the subregions are fitted together along interfaces (with or without continuity).
- **Overlaid** or **chimera** grids: Here, the grids are composed of completely independent subdomain grids that overlap an overall background grid or other component grids. A lot of attention has to be paid to the organziation of data transfer between grids via appropriate interpolation techniques.

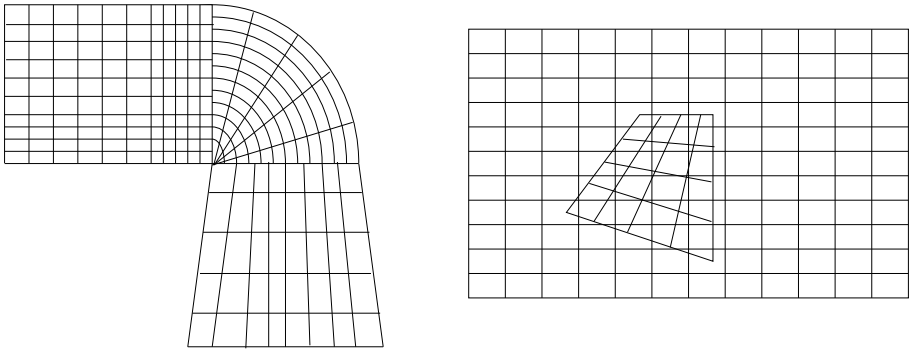


Abbildung 3.14: Composite grids: patched (left) and chimera (right)

Block-Structured Grids

Block-structured grids are very popular in computational fluid dynamics. Actually, their arrival opened the door to real-life CFD applications. In a block-structured approach, the domain is subdivided into several logically rectangular subdomains which fit together in an unstructured way. Afterwards, each of the subdomains gets a logically rectangular grid with possibly curvilinear coordinates. At the interfaces, continuity is ensured – the grid points coincide (see Fig. 3.15). The block-structured approach enables us to combine the advantages of both boundary-fitted grids and strictly structured grids.

Elliptic Generators

A lot of grid generation techniques are based on the idea of a transformation $\Psi : \Omega' \to \Omega$ from the unit square or unit cube Ω' to the computational domain Ω on which the given PDE is to be solved. Of course, the respective transformation has to be applied to the PDE, too. A famous representative of this class are *elliptic generators*, where the coordinates on the respective domain

[8] CSG is a standard CAD representation scheme for rigid bodies. Here, simple parametrized primitives like bricks or cylinders are used to form complicated objects with the help of the Boolean operations union, intersection, and set difference.

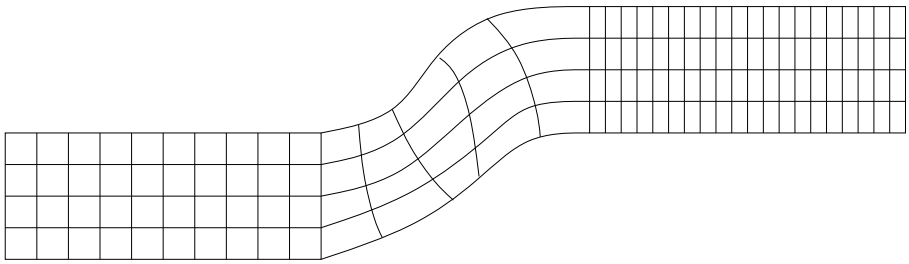


Abbildung 3.15: A block-structured grid

are obtained as solutions of systems of elliptic PDEs. This ensures that we get very smooth grid lines in the interior – even if the boundaries are not smooth (see the left part of Fig. 3.16). It is mainly this advantage that makes elliptic grid generators that popular, in spite of the relatively high cost. However, grid control, i. e. the explicit control of the grid points' and lines' position, is often quite difficult.

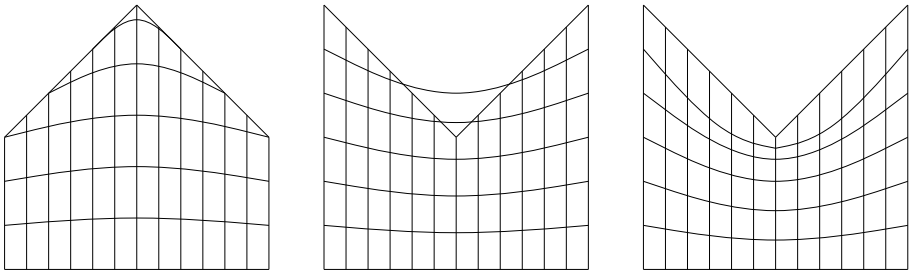


Abbildung 3.16: Grids by elliptic methods (convex (left) and nonconvex case (centre and right))

We regard a simple 2 D example. A grid shall be generated on a (logically rectangular) domain Ω bounded by four curves $c_{0,y}$, $c_{1,y}$, $c_{x,0}$, and $c_{x,1}$ (see Fig. 3.17). Then, one possibility would be to define a system of Laplacians for the components $\xi(x,y)$ and $\eta(x,y)$ of the mapping $\Psi(x,y)$,

$$\begin{aligned} \Delta\xi(x,y) &= 0 \qquad \text{on} \quad \Omega' =]0,1[^2\,, \\ \Delta\eta(x,y) &= 0 \qquad \text{on} \quad \Omega'\,, \end{aligned} \tag{3.116}$$

with Dirichlet boundary conditions

$$\begin{aligned} \begin{pmatrix} \xi(x,0) \\ \eta(x,0) \end{pmatrix} &= c_{x,0}(x)\,, \qquad \begin{pmatrix} \xi(x,1) \\ \eta(x,1) \end{pmatrix} = c_{x,1}(x)\,, \\ \begin{pmatrix} \xi(0,y) \\ \eta(0,y) \end{pmatrix} &= c_{0,y}(x)\,, \qquad \begin{pmatrix} \xi(1,y) \\ \eta(1,y) \end{pmatrix} = c_{1,y}(x)\,. \end{aligned} \tag{3.117}$$

A drawback of this simple approach is the fact that, in the nonconvex case, the resulting grid lines may leave the domain (see the centre part of Fig. 3.16). Here, *inverse elliptic methods*, where the Laplacians are defined on the curvilinear domain Ω and then transformed to the unit square, can serve as a remedy (see the right part of Fig. 3.16). The price for this is a more complicated and coupled system of PDEs to be solved.

Hyperbolic Generators

Instead of solving an elliptic system, a computational grid can be constructed by solving a hyperbolic system, too. Due to the character of hyperbolic PDEs, however, only one boundary with

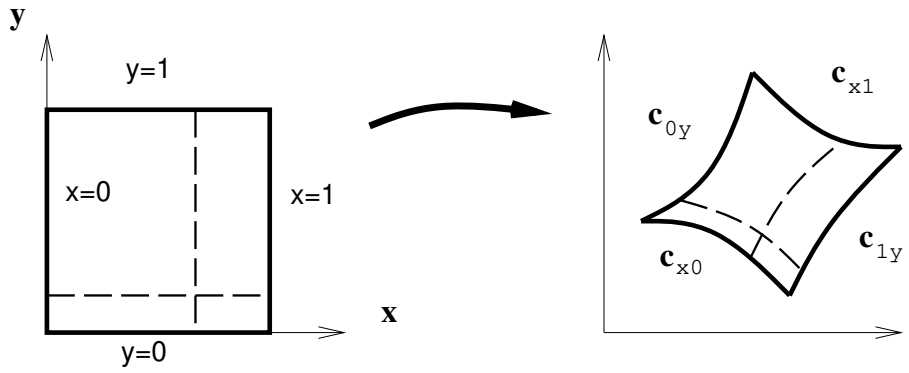


Abbildung 3.17: Mapping and elliptic grid generation

respect to each coordinate can be specified. Hence, hyperbolic generators are appropriate for calculations on physically unbounded regions. Here, the advantage of the hyperbolic approach is a gain in speed compared with elliptic generators.

Algebraic Generators

Algebraic or *interpolation-based* generators do not require to solve a whole system of PDEs in order to get a grid for another PDE's solution, but they are based on simple interpolation techniques. The most famous one is the so-called *transfinite interpolation* or, as known in computer-aided geometric design (CAGD), the *Coons patch*. Again, we restrict ourselves to the 2D case and define a grid for the domain Ω given in the right part of Fig. 3.17 whose boundaries are specified by the four curves $c_{0,y}, \ldots, c_{x,1}$. We can summarize all four curves in one single definition $c(x,y)$, defined for $x \in \{0,1\}$ or $y \in \{0,1\}$, and introduce three interpolaton operators F_1, F_2, and F_1F_2 that map the function $c(x,y)$ defined on the boundary of Ω', only, to a function given on the whole Ω':

$$\begin{aligned} F_1(x,y) &:= (1-x)\cdot c(0,y) \;+\; x\cdot c(1,y)\,, \\ F_2(x,y) &:= (1-y)\cdot c(x,0) \;+\; y\cdot c(x,1)\,, \\ F_1F_2(x,y) &:= (1-x)(1-y)\cdot c(0,0) \;+\; x(1-y)\cdot c(1,0) \\ &\quad +\; (1-x)y\cdot c(0,1) \;+\; xy\cdot c(1,1)\,. \end{aligned} \tag{3.118}$$

Then, the transfinite interpolation operator TF is defined as

$$TF(x,y) \;:=\; (F_1 + F_2 - F_1F_2)\,(x,y)\,. \tag{3.119}$$

Obviously, TF performs an interpolation of all four boundary curves into the interior of Ω.

The 3D extension is straightforward. The transfinite interpolation allows a very cheap and easy to control grid generation (explicit lines which shall become coordinate lines can be taken into account as additional inner or virtual boundaries). However, boundaries that are not smooth are inherited into the interior of the domain, which is a serious drawback. Furthermore, once again, the resulting grid lines may leave the domain. In Fig. 3.18, two examples shall illustrate the principle of transfinite interpolation.

Adaptive Grids

Many numerical simulations require the dynamical and adaptive refinement of meshes during the computation, if one reaches the following point: Investing more everywhere is no longer feasible for

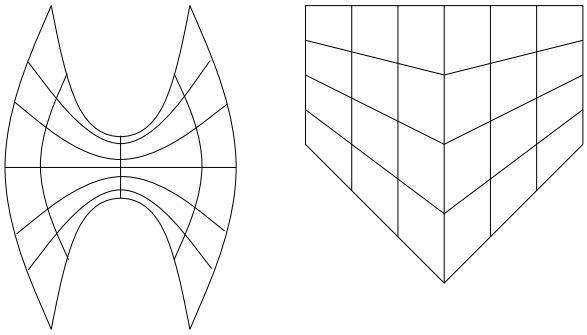


Abbildung 3.18: Transfinite interpolation

economic reasons, stopping already now means that you haven't obtained a sufficiently accurate solution. Thus, more has to be invested locally at some points or in some subregions, whether others are already treated sufficiently well. There are several strategies for going in that direction. First, one might decide to *refine* the grid in the critical regions (the classical mesh refinement or *h-adaptation*). Second, one might increase the order of approximation (by using higher order finite elements or higher order stencils, which does the *p-adaptation*[9]. Finally, if no more work can be invested, the idea might be to *rearrange* the grid points in order to get an error that is balanced better. In the context of grid generation, we are primarily interested in the first case. For structured grids, the most common way to refine a given mesh is block adaptation: Identify the critical regions (with the help of some *error indicator* estimating the local error in that point or region, and start a simple block refinement (double mesh width in the respective block, for example). Special attention has to be paid to *hanging nodes*, since discontinuities have to be avoided. See Fig. 3.19 for a simple example.

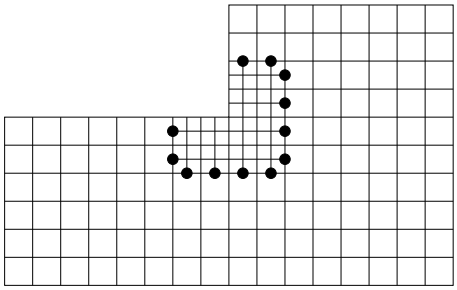


Abbildung 3.19: Block adaptation and hanging nodes: If there is a non-zero in a point marked with a black circle, then the point has this value from the fine grid's point of view, but it is just the difference of its two neighbours' values from the adjacent coarse cell's view.

3.7.2 Unstructured Grids

Unstructured grids are closely related to finite element methods, and one usually thinks of (nearly) arbitrary triangulations or tetrahedral grids.

Delaunay Triangulations, Voronoi Diagrams

Suppose you have to find a grid for some domain Ω. Moreover, suppose you have already determined discrete points for your grid. How can you derive elements in the finite element sense? One solution

[9] The parameter p stands for the order or polynomial degree of the used approach.

is very old and goes back to Dirichlet and Voronoi. For a given set of points $P_i, i = 1, \ldots, N$, the *Voronoi regions* V_i can be defined as

$$V_i := \{P : \|P - P_i\| < \|P - P_j\| \; \forall j \neq i\} . \tag{3.120}$$

This means that V_i contains all points the are closer to P_i than to any other of our grid points P_j. Overall, the result is the so-called *Voronoi diagram*, a subdivision of the 2 D or 3 D domain into polygones or polyhedra, respectively. If one draws a line between all pairs of points that are neighbouring in the sense that they are located in adjacent Voronoi regions, the result is a set of disjoint triangles (a *Delaunay triangulation*, see Fig. 3.20 for a simple example) or tetrahedra covering the convex hull of the P_i. Delaunay trianglations are used very often, since they have some properties both favourable for efficient algorithms for their construction and for the numerical solution process afterwards.

In the following, for reasons of simplicity, we will speak of triangles and triangulations for general dimensionality d.

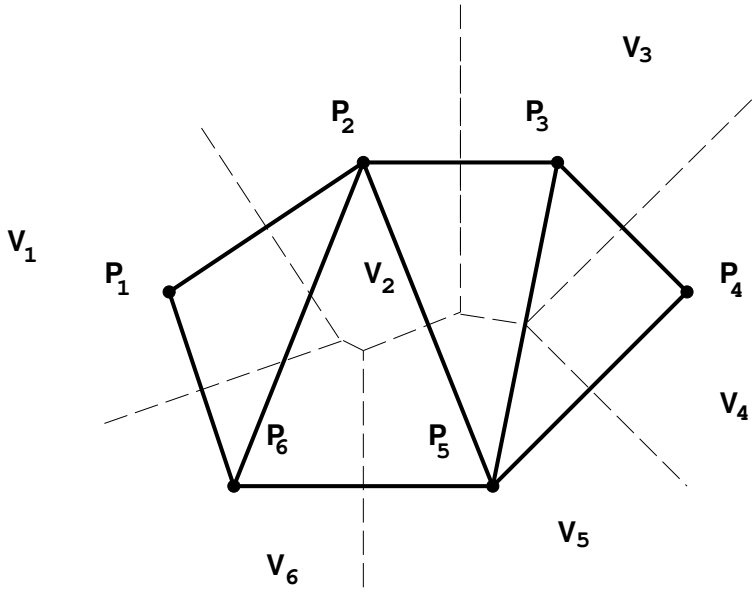


Abbildung 3.20: Voronoi diagram (dashed lines) and Delaunay triangulation (solid lines) in 2 D

Point Creation

Now, we know what to do when we have grid points – but how to get these? There are several possible ways:

- **Independent generation**: Use the grid points of a structured grid (with has nothing to do with the later triangulation), for example a polar grid in the case of a ring domain, as starting point.
- **Superposition and successive subdivision**: Superimpose a regular grid over the domain with the help of a recursive subdivision approach (quad- or octrees, for example).
- **Successive subdivision of a boundary-based triangulation**: Start with a given boundary point distribution, generate the Delaunay triangulation, and continue with subdividing the triangles by introducing new points at the triangles' centroids according to certain rules (reduce big triangles etc.), until resolution is high enough.

- **Point and line sources**: Apart from points on the boundary, point or line sources in the interior of the domain can be introduced. They shall help to ensure that the successive subdivision of the initial Delaunay triangulation gets especially refined near the sources. This is helpful, for example, if one knows the position of singularities or boundary layers.

Other Techniques

Of course, there are more strategies to generate unstructured grids than the Delaunay approach:

- **Advancing front methods**: Here, the idea is to start from the boundary (the starting *front* – a line in 2 D, a surface in 3 D) and advance step by step into the interior of the domain. In 2 D, with points P_i, $i = 1, \ldots, n$, and edges P_iP_j, the basic algorithmic scheme looks like

  choose an edge on the current front, say P_1P_2;
  create a new point P at equal distance d from P_1 and P_2;
  determine all points $Q \in \{P, P_1, \ldots, P_n\}$ lying within a circle of radius r around P;
  order these points Q in distance from P;
  for all those points Q, form triangles (P_1P_2Q) and accept the first suitable one;
  if the accepted Q is a new point, add it to the list of points;
  add new edges to the front and remove the old edge P_1P_2; (3.121)

  A new triangle is suitable if it does not produce edge intersections and if it satisfies certain quality conditions (concerning angles etc.). This loop is repeated until the front is empty (see Fig. 3.21).

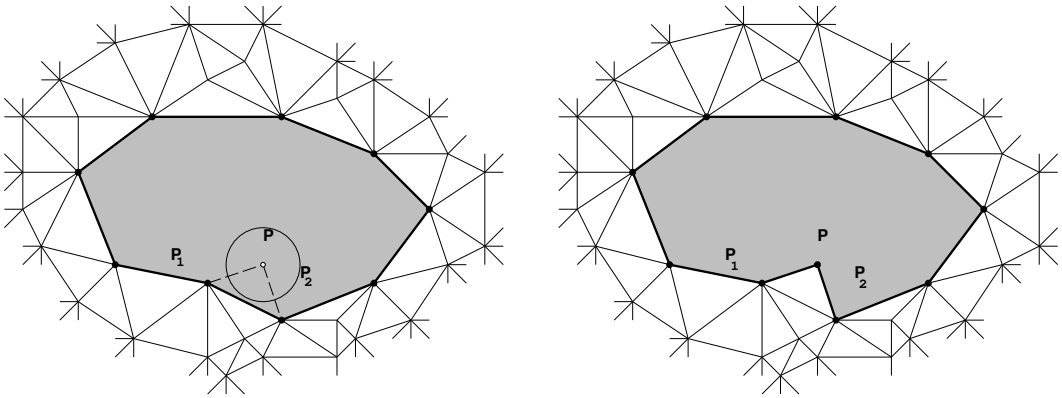


Abbildung 3.21: Advancing front algorithm: one step

  Note that the resolution is controlled via the parameter d, and that an alignment to specific directions is possible, if additional directional parameters are introduced.

- **Spacetrees**: Spacetrees (quadtrees in 2 D, octrees in 3 D) are based upon a recursive regular subdivision of a d-dimensional cube containing the domain. Until some given tolerance is reached, a square or a cube is subdivided into four squares or eight cubes, respectively, if the domain's boundary intersects it. Otherwise, it can be regarded as 'completely inside' or 'completely outside' the domain. In order to get a boundary conforming grid, the pure spacetree is modified by appropriate cutting and reconnecting to triangles and tetrahedra.

- **Hybrid grids**: In order to achieve an optimum compromise between regularity and flexibility, it is possible to combine the two grid types in the form of *hybrid* or *structured-unstructured* grids.

Adaptive Grids

Adaptive grids are somewhat natural in an unstructured context, since we have dynamic data structures, anyway. Thus, mesh adaptation is much more popular for unstructured grids than it is in the structured case. Nevertheless, it is important to note that it is not at all restricted to unstructured grids.

Once we have produced an unstructured mesh of our domain, the need for adaptive refinement usually occurs during the computational part, i. e. during the solution of the PDE(s). Hence, the solution process should provide information whether and, if at all, where to refine. Overall, we need

- a *global error estimator* telling us at the end whether our computed result is sufficiently accurate or whether we should start another computation with a finer mesh;
- a *refinement criterion* that determines the aim of refinement (balancing the error over all grid points, keeping it below a certain bound everywhere, etc.);
- a *local error estimator* or *indicator* telling us during the computation whether it would be helpful to refine the mesh locally at that point;
- a *refinement procedure* that determines the process how the grid shall be (technically) refined. Figure 3.22 shows several possible strategies for that. Note that, as in the structured case, the problem of hanging nodes may occur.

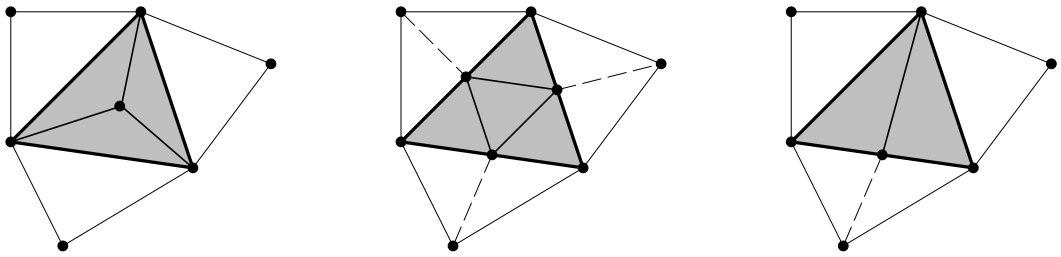


Abbildung 3.22: Some ways of refinement (with treatment of hanging nodes): adding the centroid (left), *red* subdivision (centre), and *green* one (right)

3.7.3 Methods for Varying Geometries

Recently, the focus of research interest has turned to varying geometries or moving grids instead of stationary configurations. The spectrum of applications is huge, and we want to mention only some interesting fields:

- free surfaces (injection moulding, melting or freezing processes, e. g.);
- multiphase flows (steam bubbles transporting heat in water in coolers, e. g.);
- fluid-structure interactions (mechanical valves opened or closed by a flowing liquid, tent-roof constructions subject to wind).

Of course, a permanent remeshing after each modification of the configuration is far beyond present and future computational possibilities. Thus, other strategies than the iteration of the techniques presented above have to be applied.

Front Tracking Methods

The so-called *front tracking methods* describe the boundary or *interface* between the two phases (two liquid phases or liquid and solid phase, etc.) directly, i. e. they update a geometric representation (free-from curves or surfaces) due to the current movements. Obviously, this allows to take into account any modification in a quite accurate way. However, if we have to deal with complicated 3 D geometry changes, or if even changes of topology occur (two bubbles unite), we run into methodical and computational problems. Therefore, in these cases, we can not afford the direct representation of the interface.

Front Capturing Methods

At that point, *front capturing methods* enter the game. They indicate the position of the interface in an indirect way, with the help of some global quantity. Thus, we are less precise, but front capturing methods allow a much more straightforward handling of varying geometries. Two representatives of this class are *marker-and-cell (MAC)* methods and the *volume-of-fluid (VOF)* method. Both were developed for CFD applications, and we, hence, base the following short discussion on flow problems. The MAC approach is based on structured rectangular grids. In each cell, we have a number of (virtual) particles moving with the flow field. Cells without particles do not belong to the liquid phase, cells with particles adjacent to empty cells form the surface or interface, and the other cells are located in the interior of the liquid's domain. The VOF method uses a global variable, the relative volume of the liquid phase in each cell, in order to describe the position of the interface. According to the flow field, the values of this VOF variable are updated after each time step.

Clicking and Sliding Mesh Techniques

If the movements are regular or periodic (oscillations, slow rotations of a stirrer), one might think of keeping most of the fluid grid fixed and moving only the respective structures and the adjacent flow cells. This part of the grid *slides*, and a *click* is necessary, when neighbour relations jump form one grid point to the next.

One final remark shall close the discussion of grid generation. Global grid generation is a problem that primarily occurs when we look at things from the so-called *Eulerian point of view*. Here, we assume a fixed reference system or grid, into which moving structures have to be integrated in a somewhat artificial way. Another way of looking at things provides the *Lagrangian point of view*, where no global reference system is imposed, but all 'players' have their local reference systems. This approach, which seems to be less natural at first glance, is sometimes advantageous for problems with varying geometry. The so-called *Arbitrary Lagrangian-Eulerian (ALE)* method combines both viewpoints for an efficient realization of moving structures. For example, moving bubbles have their own local description when they move in the (regular and fixed) mesh of the fluid domain. At the end, we just want to do some name dropping and mention some methods with reduced importance of a certain grid and, thus, a big potential with respect to movements: *particle methods*, *Lattice Boltzmann Automata (LBA)*, and *gridless discretizations*.

Kapitel 4

Implementation

4.1 Rechnerstrukturen

4.1.1 Moderne Mikroprozessoren

Bevor wir uns den Supercomputern zuwenden, kurz ein paar Bemerkungen zu aktuellen Monoprozessoren. Denn mit dem rasanten Anstieg der Leistungsfähigkeit von Mikroprozessoren werden auch immer mehr Simulationscodes für „normale“ PCs und Workstations geschrieben. Dort lassen sich zwar keine Klimakatastrophen simulieren, aber kleinere industrielle Anwendungen sehr wohl.

Eine Bemerkung vorneweg: Auch wenn die heutigen Monoprozessoren nach wie vor (erweiterte) von-Neumann-Rechner sind, bestehend aus den Komponenten CPU (Central Processing Unit) mit Leitwerk und Rechenwerk, Speicher sowie Ein- und Ausgabeeinheit, so ist doch bereits hier (bei komplexeren Architekturen versteht sich dies eh von selbst) mehr als ein oberflächliches Verständnis der Architekturprinzipien erforderlich, wenn man mit Simulationscodes die Leistung moderner Hardware voll ausnutzen möchte. In der Praxis zeigt sich immer wieder, dass die von den Herstellern in den Vordergrund gestellte Peak-Performance nicht nur nicht erreicht, sondern oft um bis zu vier Größenordnungen verfehlt wird.

Eine ausführliche und regelmäßig aktualisierte Zusammenstellung über die wichtigsten Prozessorfamilien findet man beispielweise unter http://bwrc.eecs.berkeley.edu/CIC/summary/. An dieser Stelle genüge ein kleiner Überblick:

Prozessor	Rechner	Taktrate (MHz)	Transistoren (Mio)	Wortbreite (Bit)
intel Pentium III Xeon	PC	≤ 550	9.5	32
intel Pentium III (Coppermine)	PC	≤ 1000	28.0	32
AMD Athlon (K7)	PC	≤ 1000	22.0	32
PowerPC 7400 (G4)	IBM	≤ 500	6.5	64
Sparc Ultra III	SUN	600	?	64
Alpha 21364	DEC	1000	100	64
MIPS R12000	SGI	≤ 300	?	64
PA-RISC 8600	HP	560	?	64

Vergleicht man diese Leistungsdaten mit denen früherer Prozessorgenerationen, so wird das „schneller-höher-weiter“-Prinzip offenkundig:

- Die Taktraten nehmen zu, die Prozessoren werden *schneller*. Damit sind auch die (billigeren)

Produkte des Massenmarkts auf CMOS[1]-Basis gegenüber den schnelleren, aber teuereren ECL[2]-basierten Chips nicht mehr so viel langsamer. Mit der Taktrate wachsen auch Maßzahlen wie MIPS (Million Instructions Per Second) oder – für unsere Anwendungen wichtiger und nicht immer proportional zu MIPS – FLOPS (FLoating point Operations Per Second). Mit den heutigen Taktraten, die schon in die Nähe von 1 GHz kommen, klopft man bereits an die natürliche Grenze der Lichtgeschwindigkeit an: 1 GHz entspricht einer Zykluszeit von 1 ns, was wiederum für Licht eine Wegstrecke von ca. 30 cm bedeutet – nur unwesentlich mehr, als in Rechnern tatsächlich an Ausbreitungsgeschwindigkeit von Pulsen beobachtet werden kann (derzeit maximal ca. 27 cm pro ns).

- Die Packungsdichte nimmt zu, und mit ihr die mögliche Funktionalität auf dem Chip.
- Zumindest im Bereich der Workstations überwiegen bereits die Prozessoren mit 64-Bit-Wortbreite. In die breiteren Befehlsworte kann mehr Information gesteckt werden, was das Potential an (feinkörniger) Parallelität erhöht.

Entgegen früheren Einschätzungen hat sich der von-Neumann-Monoprozessor als sehr zähes und langlebiges Konstrukt erwiesen. Dies liegt vor allem daran, dass es immer wieder gelungen ist, durch geeignete Erweiterungen des Grundprinzips signifikante Leistungssteigerungen zu erreichen. Zu nennen sind hier etwa:

- Die *RISC* Technologie (Reduced Instruction Set Computer) im Gegensatz zu *CISC* (Complex Instruction Set Computer):
  Im Laufe der Entwicklung der Mikroprozessoren sind die Befehlssätze immer größer geworden. So gab es beispielweise eine Vielzahl von Additionsbefehlen (addiere Konstante und Register, Konstante und Speicher, Register und Register etc.). Dementsprechend wurden die Befehle immer komplizierter (unterschiedliche Formate, unterschiedliche Anzahl von Operanden etc.), so dass es schließlich nötig wurde, die einzelnen Befehle richtiggehend zu interpretieren – die Geburtsstunde der Mikroprogrammierung. Ende der siebziger bis Anfang der achtziger Jahre setzte aus Effizienzgründen eine Gegenbewegung ein mit dem Ziel, sich auf kleine Sätze einheitlich strukturierter und ohne großen Interpretieraufwand ausführbarer Befehle zu beschränken. Heute hat sich die RISC-Technologie bei Mikroprozessoren durchgesetzt. Ihre Charakteristika:
  - relativ kleiner Befehlssatz (im zig-Bereich)
  - einfache Maschinenbefehle, wenige einfache Adressiermodi, festes Befehlsformat
  - nur ein Zyklus für einen Befehl
  - load-and-store-Prinzip: nur Lese- und Schreib-Befehle greifen auf den Speicher zu, die restlichen arbeiten ausschließlich auf Registern
  - keine Mikroprogrammierung
- Das ausgeprägte *Pipelining*: Beim Pipelining (oder deutsch Fließbandverarbeitung) werden die einzelnen Befehle in kleine Schritte zerlegt (z. B. Befehl holen, Befehl dekodieren, Befehl ausführen etc.), die alle von unabhängigen Einheiten der CPU abgearbeitet werden. Dadurch kann die Bearbeitung mehrerer Befehle *überlappend* erfolgen. Falls ein solcher Schritt gerade einen Takt benötigt, kann die Zahl der erforderlichen Takte, die ohne Überlappung identisch mit der Anzahl der Teilschritte des Befehls ist, im Optimalfall auf einen einzigen Takt reduziert werden. Dass dies eine signifikante Beschleunigung darstellen kann, sieht man etwa am Alpha 21164, bei dem sich die Gleitpunkt-Division in doppelter Genauigkeit über 61 Takte erstreckt.

  Eine weitere Leistungssteigerung ist u. U. durch Umordnung der Teilschritte eines Befehls möglich. Werden alle lesenden Speicherzugriffe so früh wie möglich und alle schreibenden

[1] Complementary Metal-Oxide Semiconductor
[2] Emitter Coupled Logic

Speicherzugriffe so spät wie möglich durchgeführt, so erhöht dies die Wahrscheinlichkeit, dass beispielsweise bei arithmetischen Operationen alle Operanden bereits in Registern vorliegen und ihre Bereitstellung zu keiner Verzögerung führt bzw. dass die zu speichernden Resultate bereits berechnet wurden. Dies zeigt die Vorteile des oben erwähnten load-and-store-Prinzips: Die (wenigen) Befehle mit Speicherzugriff können klar identifiziert und daher gezielt geeignet positioniert werden.

Ein kleines Beispiel: Ein Befehl zerfalle in die vier eintaktigen Teilschritte Load (L, Befehl laden), Decode (D, Befehl dekodieren), Register (R, Register reservieren) und Write (W, Ergebnis schreiben) sowie in den dreitaktigen Teilschritt Execute (E, Befehl ausführen). Für jeden der insgesamt sieben Takte eines Befehls sei eine andere Einheit der CPU zuständig. Betrachtet wird ein kurzer Programmausschnitt, der in insgesamt sieben Befehlen drei Operanden aus dem Speicher liest, zwei arithmetische Operationen durchführt und zwei Resultate in den Spreicher schreibt (ein horizontaler Strich steht für einen erzwungenen Takt im Wartezustand, die zuvor benutzte Einheit der CPU ist dabei blockiert):

```
-------------------------------------------------------------------------
Befehl             | 0                  10                  20      24
-------------------------------------------------------------------------
...
Reg1    = Mem(A1)    L D R E E E W
Reg2    = Mem(A2)      L D R E E E W
Reg3    = Reg1+Reg2      L D - - - R E E E W
Mem(A3) = Reg3             L - - - D - - - R E E E W
Reg4    = Mem(A4)                  L - - - D R E E E W
Reg5    = Reg4*Reg2                        L D - - - R E E E W
Mem(A5) = Reg5                               L - - - D - - - R E E E W
...
-------------------------------------------------------------------------
Befehl             | 0                  10      14
-------------------------------------------------------------------------
...
Reg1    = Mem(A1)    L D R E E E W
Reg2    = Mem(A2)      L D R E E E W
Reg4    = Mem(A4)        L D R E E E W
Reg3    = Reg1+Reg2        L D - - R E E E W
Reg5    = Reg4*Reg2          L - - D R E E E W
Mem(A3) = Reg3                     L D R E E E W
Mem(A5) = Reg5                       L D R E E E W
...
-------------------------------------------------------------------------
```

Ohne Umordnung (oben) dauert die Ausführung 25 Takte. Misst man die Zeit vom Anfang (L) des ersten Teilschritts bis zum Anfang des letzten, so ergibt sich sogar eine Dauer von nur 13 Takten. Dieses Maß ist sinnvoller, da wir ja nur einen Programausschnitt betrachten: Relativ zur Ausführungszeit des umgebenden Gesamtprogramms darf nur der Bereich zwischen erstem und letztem Load dem lokalen Ausschnitt zugerechnet werden. Diese 25 bzw. 13 Takte sind zu vergleichen mit sieben Befehlen zu sieben Takten, also 49 Takten, die ohne Pipelining benötigt wurden. Mit der Umordnungsvariante (unten) benötigt man gar nur 15 bzw. 9 Takte.

Manchmal wird genauer zwischen *Pipelining* und *Overlapping* unterschieden, wobei Pipelining dann das strenge Prinzip (die aufeinanderfolgenden Operationen sind völlig unabhängig voneinander, benötigen alle dieselbe Anzahl von Zyklen etc.) und Overlapping die allgemeinere Auslegung bezeichnen. Die optimale Situation für ein strenges Pipelining liegt natürlich

vor, wenn hintereinander lauter identische Operationen auszuführen sind – genau dies ist bei den Vektorbefehlen eines Vektorrechners der Fall. Das Prinzip des Pipelining ist relativ alt und wurde bereits in den sechziger Jahren in Rechnern eingeführt.

Natürlich erfordert das Pipelining, dass es in der CPU unabhängige Funktionseinheiten gibt, die die einzelnen Teilschritte L, D, R, E und W ausführen. Noch besser wird's, wenn diese einzelnen Funktionseinheiten mehrfach vorhanden sind. Dann hat man de facto schon mehr als einen Prozessor. Solche Architekturen werden *superskalar* genannt.

- Die *superskalare Prozessororganisation* basiert auf dem mehrfachen Einbau bestimmter Funktionseinheiten. Beim MIPS R 10000 gibt es beispielsweise fünf Ausführungspipelines (Gleitpunktaddition, Gleitpunktmultiplikation, zwei ganzzahlige ALUs[3] und eine Adress-Pipeline). Damit kann das Pipelining noch weiter forciert werden. Die superskalare Prozessororganisation ist heute ebenfalls Standard.

- Das Prinzip des *Cache-Speichers* (vom frz. cacher: verstecken, verbergen) wurde eingeführt, um das Dilemma der Zugriffs- bzw. *Latenzzeit* bei Mikroprozessoren zu entschärfen. Hierunter versteht man das Problem, dass die CPU-Leistung viel rasanter gewachsen ist als das Tempo beim Hauptspeicherzugriff, der die Prozessoren oft regelrecht ausbremst (die Balance zwischen Rechenleistung (Operationen pro Sekunde) und Durchsatz (Bytes pro Sekunde) ist erheblich gestört). Erste Realisierungen eines Cache-Speichers gab es bereits in den fünfziger Jahren, den Durchbruch brachte die Implementierung in den Baureihen 360 und 370 von IBM.

  Der Cache ist ein im Vergleich zum Hauptspeicher kleiner und schneller Speicher, der oftmals auf dem Chip untergebracht ist und in dem ein aktueller kleiner Ausschnitt des Hauptspeichers gehalten wird. Gelingt es, den Cache zu jedem Zeitpunkt so zu füllen, dass der Großteil der beim Programmablauf benötigten Daten bereits im Cache ist, also nicht mehr aus dem Hauptspeicher geladen werden muss, dann wird die tatsächliche Lage dieser Daten *verschattet* oder vor dem Benutzer *verborgen* – am Programm ändert sich dabei aber überhaupt nichts. Bezeichnen t_m und t_c die Zugriffszeit auf Hauptspeicher und Cache und p die sogenannte *Trefferwahrscheinlichkeit*, also die Wahrscheinlichkeit, dass ein benötigtes Datum bereits im Cache liegt, dann gilt für die *effektive Zugriffszeit* t_e

  $$t_e \;=\; p \cdot t_c \;+\; (1-p) \cdot t_m \,,$$

  die im Idealfall also gleich t_c ist. Das Ziel, p möglichst nahe an 1 zu realisieren, wirft eine Reihe von Fragen auf:

  - Wahl des *Ausschnitts*: Was soll im Cache gehalten werden?
  - Sicherstellung der *Lokalität* der Daten: Wie erreiche ich, dass von einem im Cache liegenden Programmausschnitt aus vornehmlich auf Daten bzw. Adressen zugegriffen wird, die ebenfalls aktuell im Cache sind?
  - *Lade- und Verdrängungsstrategien*: Wie große Einheiten sollen wann in den Cache geladen bzw. aus ihm entfernt werden?
  - *Assoziation* Hauptspeicher-Adresse – Cache-Adresse: Wie prüfe ich schnell nach, ob ein benötigtes Datum im Cache liegt, ob also die entsprechende Hauptspeicheradresse zum lokalen Ausschnitt gehört, und wo im Cache das Datum gegebenenfalls liegt?
  - *Konsistenz* bzw. *Kohärenz*: Sind Cache und Hauptspeicher immer auf demselben Stand, beispielsweise nach einem schreibenden Zugriff auf den Cache?
  - *Updating*: Wann aktualisiere ich den Hauptspeicher nach einer Änderung der Daten im Cache – nach jeder schreibenden Operation im Cache oder vor der Verdrängung des entsprechenden Blocks aus dem Cache?

[3] Arithmetic Logical Unit

Inzwischen ist es mit einem Cache nicht mehr getan. Im Falle der SGI Power Challenge etwa haben wir einen *primary cache* von 32 kB auf dem Chip und einen (langsameren) *secondary cache* von 0.5 – 16 MB außerhalb des Chips, dessen Steuerung aber immerhin noch auf dem Chip untergebracht ist. Auch dreistufige Caches wurden schon realisiert (zwei Stufen on-chip), haben sich aber (noch) nicht voll durchgesetzt. Man erhält somit eine *mehrstufige Cache-Hierarchie* bzw., auf alle Speicherarten bezogen, eine *mehrstufige Speicher-Hierarchie*.

Als Beispiel einer typischen Speicherhierarchie heutiger Architekturen betrachten wir den DEC Alpha Server 8400 mit einer mit 300 MHz getakteten Alpha 21164 CPU:

Speicherart	Größe	Latenzzeit	Durchsatz
Register	64×8 Byte	3 ns	34 GByte/s
level-1-cache (on-chip)	8 kByte	3 ns	9.6 GByte/s
level-2-cache (on-chip)	96 kByte	10 ns	4.8 GByte/s
level-3-cache (off-chip)	4 MByte	20 ns	1 GByte/s
Hauptspeicher	2 GByte	200 ns	300 MBye/s

Die Unterschiede in den Latenzzeiten sind dabei keineswegs überdurchschnittlich groß. Nach Astfalk (1999) sind folgende *slow down factors* typisch (der Zugriff auf ein Register koste eine Zeiteinheit):

Register	1
level-1-cache	2 – 3
level-2-cache	6 – 12
level-3-cache	14 – 20
near main memory	60 – 100
far main memory	100 – 250
Hintergrundspeicher	O(100)
remote (Netz, MPI)	O(1000) – O(10000)

Die Speicherhierarchie ist gerade bei numerischen Algorithmen immer im Auge zu behalten, wie das einfache Beispiel des Matrix-Vektor-Produkts $A \cdot u$ mit $A \in \mathbb{R}^{N \times N}$ und $b \in \mathbb{R}^N$ bei extrem großem N zeigt. Die Standard-Implementierung (äußere Schleife über die Zeilen der Matrix, innere Schleife zum Skalarprodukt einer Zeile von A mit u über die Spalten von A) ist in Ordnung, wenn die Matrix zeilenweise (im Cache bzw. im Hauptspeicher, wenn A nicht ganz in den Hauptspeicher passt) gespeichert ist. In diesem Fall muss immer nach ein paar Zeilen nachgeladen werden. Anders sieht es aus, wenn A spaltenweise gespeichert ist. Dann verlässt man während jedes Skalarprodukts einer Zeile von A mit u mehrfach den lokalen Bereich und muss ständig nachladen. In diesem Fall müsste man die Implementierung ändern und die äußere Schleife über die Spalten der Matrix gehen lassen.

- *Multiprozessor-Unterstützung*: Moderne Mikroprozessoren unterstützen die bei Parallelverarbeitung zusätzlich anfallenden Aufgaben (Kommunikation etc.).

Schaut man sich die fünf erwähnten Entwicklungen RISC-Technologie, Pipelining, superskalare Prozessororganisation, Speicher- und Cache-Hierarchie sowie Multiprozessor-Unterstützung näher an, so wird deutlich, dass sie alle, abgesehen von den Caches, Schritte in Richtung Parallelverarbeitung darstellen. In den ersten drei Fällen geht es dabei vor allem um *feingranulare* Parallelität (also etwa auf Befehlsebene), im letztgenannten Fall eher um *grobgranulare* (Programmebene).

Ohne ins Detail zu gehen, seien noch einige weitere (neuere) Entwicklungen genannt, die im Bereich der Mikroprozessoren zu beobachten sind und die für unsere hier betrachteten Anwendungen von Bedeutung sind:

- *VLIW* (Very Long Instruction Word): Hier wird die Wortbreite heraufgesetzt, um im Befehlswort das Potential für feinkörnigen Parallelismus weiter zu erhöhen. Bei VLIW-Rechnern handelt es sich um RISC-Architekturen mit mehreren parallelen bzw. Pipeline-artigen Ausführungseinheiten, die jedoch von einem einzigen Kontrollstrom gesteuert werden.

- *MTA* (Multi Threaded Architecture): Kleine und kleinste Prozesse (Fäden, Threads) werden parallel abgearbeitet (beim Tera MTA Rechner kann jeder Prozessor bis zu 16 solcher Minijobs parallel erledigen); auch dies ein Schritt in Richtung eines feinkörnigen Parallelismus. Als Vorteil muss man sich hier weniger Gedanken über die Cache-Problematik machen: Wenn auch ein paar Threads aufgrund eines Spreicherzugriffs ausgebremst werden, so kann doch irgend einer bestimmt weiter bearbeitet werden.
- *On-chip-multiprocessors*: Mit Entwicklungen wie den Pentium Dual-Boards rückt das Ende des klassischen Monoprozessors vielleicht wirklich in greifbare Nähe.
- *Codeoptimierung*: Mittels Werkzeugen zur automatisierten Codeoptimierung (Schleifenoptimierung, Sprungoptimierung, Umordnung (s. o.) etc.) werden Programmes gezielt für bestimmte Architekturen getrimmt (v. a. für Vektorrechner gebräuchlich).

4.1.2 Supercomputer: Parallele und verteilte Architekturen

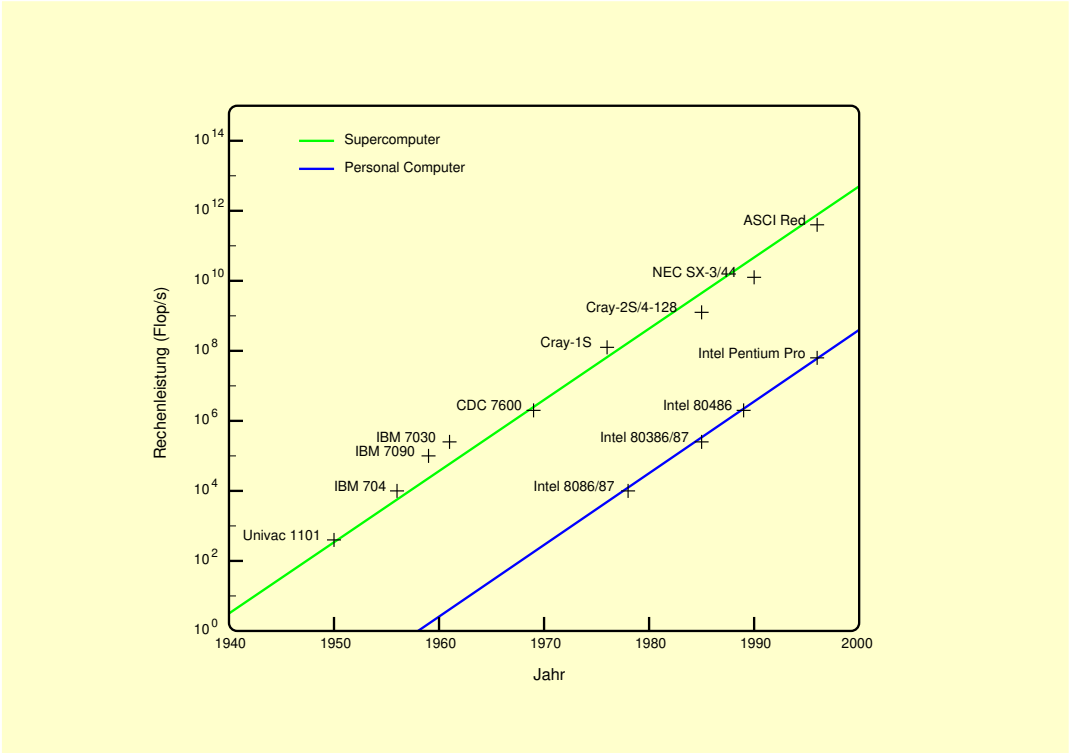


Abbildung 4.1: Entwicklung der Rechenleistung von PC-Mikroprozessoren und Supercomputern: Der Leistungszuwachs folgt *Moore's Gesetz*, wonach etwa alle 18 Monate ein Faktor zwei bzw. alle fünf Jahre ein Faktor zehn erreicht wird, was für beide Rechnerklassen gilt. Der Vorsprung von Supercomputern gegenüber PCs ist dabei nahezu konstant geblieben.

Supercomputer oder *Höchstleistungsrechner* waren zunächst durch einzelne Hochleistungsprozessoren gekennzeichnet (Vektorrechner). Zumindest aus der Perspektive des wissenschaftlichen Rechnens kann die legendäre CRAY-1 als Mutter aller Supercomputer angesehen werden [4]. Durch ihre Erstauslieferung an das Los Alamos National Laboratory in New Mexico im Jahre 1976 sowie ihre (für Maschinen solcher Leistung und solchen Preises) weite Verbreitung erhielt das wissenschaftliche Rechnen wichtige neue Impulse. Inzwischen beherrschen jedoch Multiprozessorsyste-

[4] Der Begriff des Supercomputers kam in den frühen sechziger Jahren auf, insbesondere mit der Entwicklung der CDC 6600 von Control Data, die 1964 die Schallmauer von 1 MFLOPS durchbrach.

me den Markt, was vor allem ökonomische Gründe hat: Es rechnet sich natürlich eher, mehrere Standard-Prozessoren für den Massenmarkt zusammenzuschalten, als teuere Spezialprozessoren zu entwickeln, die nie in bedeutender Stückzahl hergestellt werden können. Kaufen kann man Parallelrechner seit Mitte der achtziger Jahre (eine Alliant war 1985 die erste), erste sog. *massiv parallele* Rechner mit hoher Prozessorenzahl waren dann ab Ende der achtziger Jahre erhältlich.

Wie Abbildung 4.1 zeigt, sind die Supercomputer in puncto Gleitpunkt-Rechenleistung (gemessen in FLOPS oder Flop/s) den jeweils aktuellen Monoprozessoren des Massenmarkts immer um einen Faktor tausend bis zehntausend überlegen. Auch wenn die heutigen Programme des europäischen Wetterzentrums in Reading (einem der prominentesten Supercomputing-Anwender in Europa) nur auf Hochleistungsrechnern hinreichend schnell Daten für den Wetterbericht liefern, so ist doch auch absehbar, dass die „Grand Challenges“ von heute ein paar Rechnergenerationen später den PC nicht mehr aus der Fassung bringen werden.

The International Top 20

Nr.	Hersteller	Typ	Rmax	Institution	Land	# Proz.
1	Intel	ASCI Red	2379.6	Sandia NL, Albuquerque	USA	9632
2	IBM	ASCI Blue Pacific SST, SP 604e	2144	Lawrence Livermore NL, Livermore	USA	5808
3	SGI	ASCI Blue Mountain	1608	Los Alamos NL, Los Alamos	USA	6144
4	IBM	SP Power3 375 MHz	1417	IBM/NAVOCEANO, Poughkeepsie	USA	1336
5	Hitachi	SR8000-F1/112	1035	Leibniz Rechenzentrum, München	D	112
6	Hitachi	SR8000-F1/100	917.2	High Energy Acc. Res. Org., Tsukuba	J	100
7	Cray	T3E1200	891.5	Government	USA	1084
8	Cray	T3E1200	891.5	US Army HPC Res. Center , Minneapolis	USA	1084
9	Hitachi	SR8000/128	873.6	Univ. of Tokyo (Comp. Center)	J	128
10	Cray	T3E900	815.1	Government	USA	1324
11	IBM	SP Power3 375 MHz	723.4	Oak Ridge NL, Oak Ridge	USA	704
12	SGI	ORIGIN 2000 250 MHz	690.9	Los Alamos NL, Los Alamos	USA	2048
13	Cray	T3E900	675.7	NAVOCEANO, Bay St. Louis	USA	1084
14	Cray	T3E1200	671.2	CSAR at Univ. of Manchester	UK	812
15	Cray	T3E1200	671.2	Deutscher Wetterdienst, Offenbach	D	812
16	IBM	SP Power3 222 MHz	613.02	UCSD Supercomp. Center, San Diego	USA	1152
17	Hitachi	SR8000-F1/60	577.5	Univ. of Tokyo (Solid Physics)	J	60
18	Cray	T3E900	552.92	UK Meteorological Office, Bracknell	UK	876
19	IBM	SP PC604e 332 MHz	547	Charles Schwab Corp., San Francisco	USA	2000
20	Cray	T3E1200	526.6	UK Meteorological Office, Bracknell	UK	636

Abbildung 4.2: Die TOP 20 weltweit (Stand: Juni 2000; Quelle: http://www.top500.org/)

Supercomputer sind nach wie vor die Aushängeschilder von Rechenzentren oder Institutionen, die sich mit wissenschaftlichem Rechnen beschäftigen. Es gibt sogar eine alle sechs Monate aktualisierte *TOP 500-Rangliste*, in der die aktuell weltweit 500 leistungsfähigsten Rechner geführt werden.

In Abbildung 4.2 sind die 20 schnellsten Maschinen weltweit aufgelistet, in Abbildung 4.3 die 20 besten in Deutschland (Stand: November 1999). Allerdings sorgt ein Eintrag in diese Liste oft nur für kurze Freude, da das Geschäft einfach zu kurzlebig ist. Man beachte, dass die derzeitigen Spitzenreiter, die ASCI-Rechner der amerikanischen National Laboratories, im Grunde nichts anderes sind als extrem schnelle Vernetzungen einer sehr großen Zahl von Standardprozessoren (Pentium bzw. Power PC bzw. MIPS).

The German Top 20

Nr.	Hersteller	Typ	Rmax	Institution	# Proz.
5	Hitachi	SR8000-F1/112	1035	Leibniz Rechenzentrum, München	112
15	Cray	T3E1200	671.2	Deutscher Wetterdienst, Offenbach	812
32	Cray	T3E1200	447.8	Forschungszentrum Jülich	540
36	Cray	T3E	355.1	MPI/IPP, Garching	812
39	Cray	T3E900	341.3	HWW/Universität Stuttgart	540
54	Cray	T3E900	253.8	Konrad Zuse-Zentrum, Berlin	404
61	Cray	T3E	234.9	Forschungszentrum Jülich	540
75	IBM	SP Power3 375 MHz	171	Deutsche Telekom, Darmstadt	168
87	IBM	SP Power3 375 MHz	148	GWDG, Göttingen	144
92	IBM	SP PC604e 332 MHz	138	DeTeCSM, Bielefeld	452
108	NEC	SX-5/32Me2	121.5	HWW/Universität Stuttgart	32
112	IBM	SP Power3 375 MHz	118	Bayer AG, Leverkusen	114
123	Fujitsu	VPP700/52	106.3	Leibniz Rechenzentrum, München	52
125	IBM	SP PC604e 332 MHz	105.2	DeTeCSM	342
131	IBM	SP PC604e 332 MHz	101	Thyssen	328
155	Sun	HPC 10000 400 MHz	79.36	Deutsche Telekom, Bamberg	128
163	IBM	SP PC604e 332 MHz	79.17	BASF, Ludwigshafen	256
164	IBM	SP PC604e 332 MHz	79.17	Bayer AG, Leverkusen	256
165	IBM	SP PC604e 332 MHz	79.17	Bayer AG, Leverkusen	256
168	NEC	SX-4/40H2	77.2	HWW/Universität Stuttgart	40
⋮					
and formerly (99/2)					
454	Self-made	Parnass2 Cluster	34.23	Universität Bonn/Angewandte Mathematik	128

Abbildung 4.3: Die TOP 20 in Deutschland (Stand: Juni 2000; Quelle: http://www.top500.org/)

Der Begriff des Parallelrechners ist alles andere als eindeutig. Historisch gesehen haben wir es mit zwei Grundtypen paralleler Rechnersysteme zu tun, dem klassischen *Parallelrechner* (mehrere i. d. R. gleichartige Prozessoren räumlich zusammengefasst, enge und schnelle Kopplung) sowie dem *Rechnernetz* (räumlich verteilt angeordnete und u. U. heterogene Prozessoren, lose und langsame Kopplung). Oft werden erstere als *parallele* Systeme im engeren Sinne und letztere als *verteilte* Systeme bezeichnet. Solche *Workstation-Cluster* oder *PC-Cluster* haben aufgrund ihrer Verfügbarkeit sowie ihrer oftmals geringen Auslastung stark an Bedeutung gewonnen. In den letzten Jahren sind zudem die Übergänge fließend geworden, da neben dem Ethernet-Standard[5] durch

[5] Netztechnologie für LANs (Local Area Networks), ursprünglich 10 Mbit/s, heute als Fast Ethernet 100 oder

neue leistungsfähige Netztechnologien wie FDDI[6], ATM[7], HiPPI[8] oder SCI[9] sowie durch einheitliche Programmiermodelle für beide Klassen von Systemen wie MPI (Message Passing Interface) oder PVM (Parallel Virtual Machine) eine strenge begriffliche Trennung oft nur noch bedingt möglich bzw. sinnvoll ist.

Was Rechner angeht, so gilt im wissenschaftlichen Rechnen natürlich die olympische Maxime *schneller – höher – weiter*, da (wie in Kapitel 2 dargelegt)

- Probleme mit Raumauflösung nur dreidimensional hinreichend genau modelliert und simuliert werden können;
- die meisten Probleme instationär sind und somit auch die Zeit berücksichtigt werden muss;
- die somit de facto vierdimensionalen Probleme in jeder Dimension möglichst hoch aufgelöst (diskretisiert) werden müssen;
- oftmals eine größere Anzahl von Variablen in jedem Raum-Zeit-Punkt zu betrachten ist.

Ohne raffinierte Diskretisierungstechniken wachsen offensichtlich sowohl die Rechenzeit als auch der Speicherplatzbedarf einer Simulationsrechnung exponentiell in der Anzahl der Dimensionen. Soll bei $d = 4$ die Auflösung in jeder Richtung verdoppelt werden, zieht dies bereits ein Anwachsen des Aufwands um einen Faktor 16 nach sich – ein Phänomen, das wir ja schon in Kapitel 2 als *Fluch der Dimensionalität* kennengelernt hatten. Dies zeigt, dass Hochleistungsrechner wichtig sind und bleiben werden. Andererseits reicht es aber sicher nicht aus, sein Glück nur in der Verbesserung der Hardware zu suchen: Dafür sind die zu lösenden Simulationsaufgaben trotz allen technischen Fortschritts zu gewaltig. Auch die Parallelisierung allein ist kein Allheilmittel: Ein hoffnungslos schlechter sequentieller Algorithmus lässt sich vielleicht perfekt parallelisieren, er wird jedoch dadurch im Kern nicht besser.

Fazit: Was gebraucht wird, sind bestmögliche sequentielle Algorithmen, die sich effektiv parallelisieren lassen und dann auf den größten verfügbaren Rechnern eingesetzt werden können. Gefragt ist also das Zusammenspiel von Methoden aus den Kapiteln 3 und 4.

Unter den Oberbegriffen Parallelisierung, Parallelismus und Parallelrechner finden sich ganz unterschiedliche Strategien, wobei jeweils nicht nur die Extremlösungen, sondern auch Mischformen anzutreffen sind:

- *was* wird parallelisiert?
  - Codeparallelität: Aufteilung des Programms (parallel ausführbare Programmteile werden auch parallel abgearbeitet)
  - Datenparallelität: Aufteilung der Daten (jeder Prozessor wendet das gegebene Programm auf einen Teil der Daten an, etwa beim Sortieren oder bei der Lösung partieller Differentialgleichungen mittels Gebietszerlegung[10])
  - Wettbewerbsparallelität: verschiedene Algorithmen werden simultan ausprobiert (etwa bei der Suche: verschiedene Such-Algorithmen laufen parallel, der erste Finder gewinnt)
- *wie* wird parallelisiert?
  - Nebenläufigkeit: „echt“ parallele Bearbeitung

1000 Mbit/s

[6] Fiber Distributed-Data Interface, Standard für Datenübertragung in LANs bis 200 km via Glasfaserkabel, 100 Mbit/s

[7] Asynchronous Transfer Mode, verbindungsorientiert mit fixer Paketgröße

[8] High-Performance Parallel Interface, Cu-basierter Standard mit maximal 1.6 Gbit/s

[9] Scalable Coherent Interface

[10] Bei der Gebietszerlegung (engl. Domain Decomposition) wird das betrachtete Gebiet in Teilgebiete zerlegt, auf denen dann die PDE abwechselnd gelöst wird (Kopplung über die mehreren Teilgebieten angehörenden Variablen im Überlappungsbereich). Der Ansatz stammt aus dem 19. Jahrhundert, wurde jedoch erst im Zuge der aufkommenden Parallelverarbeitung wieder aktuell.

  - Pipelining: überlappende Bearbeitung
  - kontrollgetriebener Parallelismus: explizite Anweisungen legen fest, welcher Prozessor was ausführt (imperative Sprachen)
  - datengetriebener Parallelismus: der Datensatz bestimmt implizit, was an ihm parallel ausgeführt werden kann (z. B. für Datenflussmaschinen oder funktionale Sprachen)
  - reduktionsgetriebener Parallelismus: top-down-Auswertung von (z. B. baumartig strukturierten) Ausdrücken (angemessen für Ausdruckskalkül oder funktionale Sprachen)
- *wo* wird parallelisiert?
  - ganz grobkörnig: auf Programmebene
  - ziemlich grobkörnig: auf Prozessebene
  - ziemlich feinkörnig: auf Maschinenbefehlsebene
  - ganz feinkörnig: auf Mikrobefehlsebene, also im Mikroprogramm
- *wer* parallelisiert?
  - manuell/explizit: nur der Anwender
  - interaktiv: der Anwender im Zusammenspiel mit dem System
  - automatisch/implizit: nur das System (vor allem feinkörnig, es gibt aber auch vektorisierende und parallelisierende Compiler bzw. Quellcode-Optimierer, die etwa die Schleifenparallelisierung durchführen)
- *Topologie* des Systems:
  - Anordnung von Speichern und Prozessoren
  - Verbindungsstrukturen
  - statisch (fest verdrahtet, passive Elemente, siehe Abb. 4.4) oder dynamisch (aktive Schalter, Bsp. Crossbar Switch oder Shuffle Exchange, siehe Abb. 4.5)

  Die Frage der Topologie ist sowohl für Parallelrechner im engeren Sinn als auch für Rechnernetze von entscheidender Bedeutung. Hier werden die für Kosten und Leistung verantwortlichen Größen festgelegt wie die Anzahl der Verbindungen und Schalter (Switches), die Anzahl der erforderlichen Anschlüsse an jedem Prozessor bzw. Speicher, die mittlere Weglänge von einem Prozessor zu einem anderen etc. Bus-Topologien beispielweise sind nur bedingt skalierbar, da der Bus offenkundiger Flaschenhals ist. Crossbar Switches gestatten direkte Verbindungen von jedem Prozessor zu jedem beliebigen Speicher, sie sind jedoch sehr aufwendig.
- *Synchronisation*:
  - enge Kopplung (Parallelrechner i. e. S.)
  - lose Kopplung (Netze: LAN (local area network), WAN (wide area network))
- *Kommunikation*:
  - über gemeinsamen Speicher
  - explizit über Nachrichten

Dementsprechend groß ist die Anzahl der Möglichkeiten, Parallelrechner bzw. Rechner allgemein zu klassifizieren. Da wäre zunächst die Klassifizierung nach Flynn (1972):

- *SISD: Single Instruction Single Data*
  $\longrightarrow$ der klassische Monoprozessor nach dem von-Neumann-Prinzip

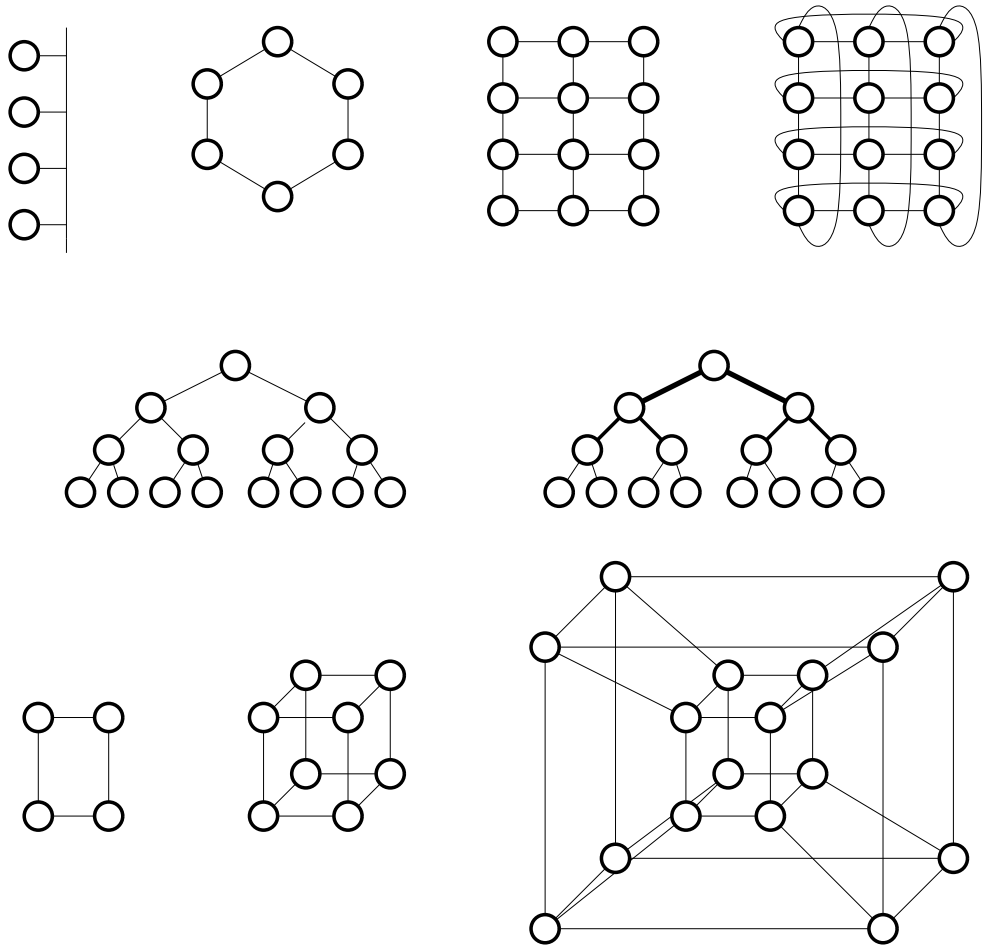


Abbildung 4.4: Statische Netztopologien: Bus, Ring, Gitter, Torus (oben), Tree und Fat Tree (Mitte), Hypercube der Dimension 2, 3 und 4 (unten); Kreise symbolisieren Rechner (Prozessor und Speicher)

- *SIMD: Single Instruction Multiple Data*
  $\longrightarrow$ Vektor- und Feldrechner
  - *Vektorrechner*: Verarbeitung von Vektoren nach dem Prinzip des Pipelining mit speziellen Vektorbefehlen in optimierten Vektoreinheiten (aufwendiges Speichersystem, aber einfache Programmierung und hohe Leistung; Bsp.: CRAY 1, 2, X-MP, Y-MP, J 90, C 90, T 90; NEC SX-n; Fujitsu VP n); heute meist als parallele Vektorrechner
  - *Feldrechner*: Verarbeitung von Daten nach dem Prinzip der Nebenläufigkeit, Feld von Prozessoreinheiten, Ablaufsteuerung durch zentrale Instanz (Programmierung schwierig; Bsp.: Thinking Machines CM-2: MasPar MP-1, MP-2)
- *MIMD: Multiple Instruction Multiple Data*
  $\longrightarrow$ Multiprozessorsysteme, aber auch Netze
  - *Multiprozessoren*: *verteilter* Speicher (lose Kopplung, lokale Adressräume, explizite Kommunikation via Nachrichten; Bsp.: intel Paragon, IBM SP 2) oder *gemeinsamer* Speicher (enge Kopplung, globaler Adressraum, implizite Kommunikation via gemeinsame Variable; Bsp.: Workstation-Server)

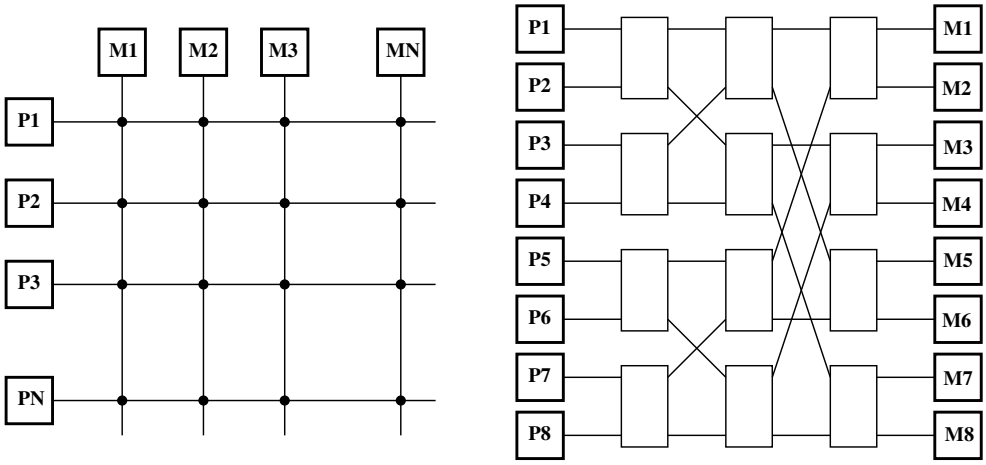


Abbildung 4.5: Dynamische Netztopologien: Crossbar Switch (links) und Shuffle Exchange Network (rechts); P steht für Prozessor, M für Speicher.

  - *Netze*: unterschiedliche Topologien (Ring, Torus, Hypercube, n-Cube, Fat Tree) mit unterschiedlichen Leistungsdaten (Ports pro Knoten, Maximalabstand); Verbindungsaufbau und Übertragung (Leitungsvermittlung, Paketvermittlung, Routing)

- *MISD: Multiple Instruction Single Data*
  $\longrightarrow$ ungewöhnlich und auf den ersten Blick wenig sinnvoll; ein Datenstrom durchläuft eine Folge von Prozessoren, die unterschiedliche Anweisungen ausführen (z. B. *systolisches Array*)

Insbesondere die Zugehörigkeit von Parallelrechnern im eigentlichen Sinne und Workstation-Clustern zur selben Kategorie, aber auch die Tatsache, dass die CRAY X-MP bis zur vier Prozessoren (als MIMD) hat, die aber jeder intern mittels Pipelining vektorisieren (also SIMD), zeigt die Schwächen dieser Klassifizierung. Deshalb ein zweiter Versuch: Jetzt wird anhand der Prozessor-Speicher-Anordnung unterschieden, wobei die einzelnen Klassen gleich hinsichtlich wichtiger Eigenschaften bewertet werden:

- *Skalierbarkeit*: Wie einfach kann das System um weitere Prozessoren erweitert werden?
- *Programmiermodell*: Kann wie gehabt programmiert werden?
- *Portierung*: Wie einfach ist die Portierung von Programmen, d. h. der Transfer von einem Prozessor zum anderen unter Erhaltung der Lauffähigkeit und Funktionalität?
- *Lastausgleich*: Wie einfach ist die gleichmäßige Auslastung aller Prozessoren zu erzielen?

Wir unterscheiden die Klassen *UMA*, *NORMA* und *NUMA*:

- *UMA: Uniform Memory Access*
  $\longrightarrow$ Systeme mit gemeinsamem Speicher (shared memory): symmetrische Multiprozessoren, parallele Vektorprozessoren

  - Vorteile: Programmiermodell, Portierung, Lastausgleich
  - Nachteile: Skalierbarkeit
  - Beispiele: Server aus der Workstation- und PC-Welt; CRAY T 90, Y-MP

- *NORMA: NO Remote Memory Access*
  $\longrightarrow$ Systeme mit verteiltem Speicher (distributed memory)

  - Vorteile: Skalierbarkeit

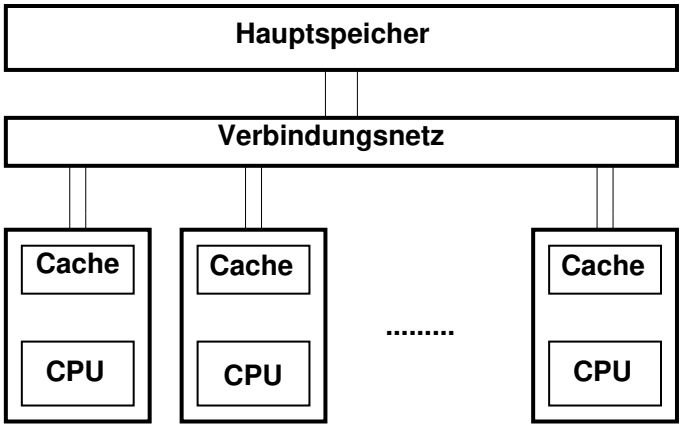


Abbildung 4.6: Schematischer Aufbau einer UMA-Konfiguration

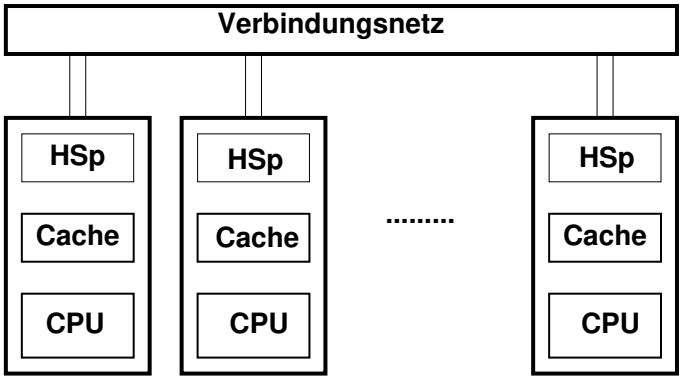


Abbildung 4.7: Schematischer Aufbau einer NORMA-Konfiguration

  - Nachteile: neues Programmiermodell, Portierung, Lastausgleich
  - Beispiele: Workstation- und PC-Netze; IBM SP-2; intel iPSC/860, Paragon; Thinking Machines CM-2, CM-5; vereinzelt auch parallele Vektorrozessoren (Hitachi)

- *NUMA: Non-Uniform Memory Access*
  $\longrightarrow$ Systeme mit virtuell gemeinsamem Speicher
  - Vorteile: Programmiermodell, Skalierbarkeit, Portierung
  - Nachteile: Kommunikation, Cache-Steuerung (Kohärenzproblem)
  - Beispiele: KSR-1; CRAY T3D, T3E; CONVEX SPP

Auch hier gibt es Mischformen, etwa UMA/NORMA (Cluster symmetrischer Multiprozessoren: NEC SX-4).

Für die zukünftige Entwicklung sind einige Fragen von herausragender Bedeutung:

- Bleibt der Leistungsabstand zwischen PC und Workstation auf der einen und Supercomputer auf der anderen Seite bestehen oder verringert sich die Lücke (On-chip-Multiprozessoren)?
- Haben teuere Hochleisungsprozessoren kleiner Stückzahl eine Zukunft oder läuft alles auf vernetzte Systeme hinaus (Bsp. ASCI Red)? Werden das eng gekoppelte Systeme sein, oder wird das *Metacomputing* dominieren, also der Verbund leistungsstarker Rechnung auch über größere Distanzen hinweg?

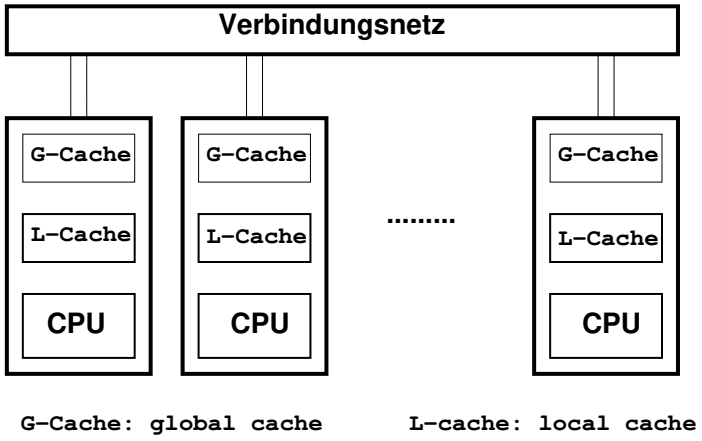


Abbildung 4.8: Schematischer Aufbau einer NUMA-Konfiguration

- Wie sieht die Zukunft von UNIX im Bereich des Hochleistungsrechnens aus?
- (virtually) shared memory vs. distributed memory?

Zur zweiten Frage ein kleines Beispiel. Es ist mit entsprechendem Know-how möglich, aus marktüblichen Standard-Komponenten einen leistungsfähigen Parallelrechner zu bauen – auch ohne einen Rechnerhersteller im Hintergrund und ohne größere Löt-Aktionen. Für Details sei auf das Projekt *Parnass*$_2$ an der Abteilung für wissenschaftliches Rechnen des Instituts für Angewandte Mathematik der Universität Bonn verwiesen (URL: http://wwwwissrech.iam.uni-bonn.de/). Dort wurden 32 Pentium II Dual Boards (400 MHz)[11], also 64 Einzelprozessoren, mittels der Hochleistungs-Netztechnologie *Myrinet* in Fat-Tree-Topologie verschaltet. Auf allen Knoten läuft das Betriebssystem LINUX. Heraus kommt ein Parallelrechner mit 8 GB Hauptspeicher, 270 GB Plattenspeicher und einer Spitzenleistung von 25.6 GFLOPS (was ihn in die TOP 500 katapultiert; vgl. Abb. 4.3, dort bereits mit 128 Prozesoren), der mit knapp 250 000 DM einen Bruchteil des Anschaffungspreises von Supercomputern derselben Leistungsklasse kostet. Programmiert wird das Ganze mit dem Standard-Programmiermodell MPI (kommt noch im Abschnitt über Parallelisierung). Erste Tests mit Simulationsanwendungen aus den Bereichen Strömungsmechanik und Molekulardynamik zeigen ein vielversprechendes Leistungsverhalten. Auch wenn solche Konstrukte den Nachweis erst noch erbringen müssen, ähnlich stabil und breit leistungsfähig wie die herkömmlichen Supercomputer zu sein, so zeigt es doch den Weg in die wohl technologisch vielversprechendste Richtung. Im Grunde genommen sind viele Parallelrechner heute nichts anderes als ein solches Konstrukt – wohl verborgen hinter einem ansprechenden Gehäuse, versteht sich.

[11] Im Vorgänger *Parnass* wurden verschiedene Rechner von SGI mit insgesamt 49 Prozessoren zusammengechaltet.

4.2 Programmiertechnik

Nach unserem Ausflug in konventionelle und parallele Rechnerarchitekturen blasen wir nun zum Angriff auf die Software. Einer kurzen Diskussion von Programmiersprachen und Sprachkonzepten folgen dabei einige Bemerkungen zu den wesentlichen Bausteinen jedes Programms, Algorithmen und Datenstrukturen. Zum Abschluss dieses Abschnitt's gibt's dann einen kurzen Abstecher in das Software-Engineering, also, knapp (und etwas vereinfachend) gesagt, in die Methodik von Programmentwurf, -erstellung und -wartung.

4.2.1 Programmierparadigmen und Sprachen

Die (wie gesehen heute noch im Wesentlichen aktuelle) Architektur des von-Neumann-Rechners verlangt ein (maschinenabhängiges) Programm, bei dessen Bearbeitung einzelne Befehle in vorgegebener Reihenfolge abgearbeitet werden. Auf der Ebene des Programmierers von Anwendungsprogrammen ist eine Festlegung auf ein derart einschränkendes Sprachkonzept jedoch nicht zwingend – hier sorgt der Compiler für eventuell erforderliche Transformationen. Eher selten ist aber der Fall, dass das Sprachkonzept auf Anwendungsebene die Architektur bestimmt. Die resultierenden unkonventionellen Architekturen wie beispielsweise die Datenflussmaschine haben sich nicht durchgesetzt.

Was also sind die verbreiteten Sprachkonzepte, und welche können wir hier gebrauchen?

- Bei den *prozeduralen* oder *imperativen* Programmiersprachen ist das Programm eine Folge von Anweisungen, die in der programmierten Reihenfolge (evtl. durch Sprünge etc. modifiziert) sequentiell abgearbeitet werden. Im Vordergrund steht der zeitliche Ablauf, das Fortschreiten in einem dem Programm zugrunde liegenden Algorithmus. Als Beispiele seien FORTRAN, Pascal oder C genannt – die gängigen general-purpose Sprachen sind also prozedural.
  Im wissenschaftlichen Rechnen ist das prozedurale Konzept ganz klar der Platzhirsch. Die meisten Simulationscodes, die in der Industrie eingesetzt werden, sind nach wie vor in FORTRAN programmiert. Die Gründe liegen in der seinerzeit extrem schnellen Verbreitung von FORTRAN, im „Wachsen“ von solchen Codes über Jahre und Generationen von Mitarbeitern hinweg, in der Einfachheit der Programmierung und – für die FORTRAN-Gegner leider – oftmals in der Laufzeit-Effizienz. Die meisten sinnvollen oder trickreichen oder komfortablen Features modernerer Sprachen wie C, etwa im Datenstruktur-Bereich, wirken sich negativ auf die Effizienz aus. So müssen komplizierte Geflechte erst entflochten werden, insb. für den Einsatz auf Vektor- oder Parallelrechnern. Die seit Jahren im Einsatz bewährten vektorisierenden FORTRAN-Compiler tun sich da angesichts der beschränkten Möglichkeiten von FORTRAN wesentlich leichter. Im akademischen Bereich dagegen ist C inzwischen gleichberechtigt – insb. bei „ingenieur-ferneren“ Arbeitsgruppen.

- *Deklarative* Sprachen rücken weniger den Lösungsalgorithmus, sondern vielmehr das zu lösende Problem in den Mittelpunkt – es wird beschrieben. Aus einer derartigen Beschreibung kann dann der Compiler ein prozedurales (Maschinen-) Programm erzeugen. Wie diese Beschreibung zu erfolgen hat, dafür gibt es verschiedene Ansätze. *Logikbasierte* Sprachen wie PROLOG beruhen auf dem Kalkül der Prädikatenlogik, bei den *funktionalen* Sprachen wie LISP erfolgt die Beschreibung über funktionale Abhängigkeiten (im mathematischen Sinn). Für unsere Anwendungen sind die deklarativen Sprachen eher Exoten. Dass aber insbesondere das funktionale Prinzip dennoch gewinnbringend eingesetzt werden kann, liegt eigentlich nahe: Schließlich beschäftigen wir uns ja mit der numerischen Lösung mathematischer Modelle. Es gibt keine explizite Ablaufsteuerung, jede Funktion ist rechenbereit, sobald ihre erforderlichen Eingabedaten vorliegen. Die oftmals so lästigen *Seiteneffekte* sind ebenfalls unbekannt, da nichts anderes als die Abbildung von Eingabe- auf Ausgabedaten realisiert

wird. Damit gibt es auch keinen globalen Zustand (Speicherinhalt, Variablenwerte). Als „Programm“ erhält man somit einen zyklenfreien gerichteten Datenflussgraphen, der den Weg der Programmeingabe zur Programmausgabe funktional beschreibt. Im Idealfall sollen solche Programme leichter verständlich sein (der ganze Variablen-Overhead entfällt) und leichter zu verifizieren sein. Bei konventionellen Sprachen sind formale Korrektheitsbeweise dagegen in aller Regel nicht durchführbar. Ein weiterer Vorteil des funktionalen Ansatzes liegt in der Möglichkeit einfacher und automatischer Parallelisierung, da die Reihenfolge der Abarbeitung nicht mehr explizit vorgegeben wird bzw. werden muss, sondern sich automatisch aus dem Fortschreiten der Bearbeitung ergeben kann.
Fazit: Im Simulationseinsatz Fehlanzeige, im akademischen Bereich vereinzelt anzutreffen (insb. im Parallelisierungsumfeld).

- *Objektorientierte* Sprachen stellen wieder etwas anderes in den Mittelpunkt, nämlich – wie der Name schon sagt – Objekte, also Daten im weitesten Sinne. Objekte haben Eigenschaften, können ein bestimmtes Repertoire an *Methoden* ausführen und kommunizieren miteinander. Wichtige Prinzipien sind die *Klassenbildung* (Objekte mit gemeinsamen Eigenschaften und Methoden), die *Vererbung* (von Eigenschaften oder Methoden) sowie die *Verschattung* oder *Kapselung*: Jedes Objekt stellt nur bestimmte Informationen nach außen zur Verfügung, bspw. *was* es alles tun kann, andere bleiben geheim, z. B. *wie* etwas erledigt wird.
  Die „echt“ objektorientierten Sprachen wie SMALLTALK oder EIFFEL werden im wissenschaftlichen Rechnen so gut wie nicht eingesetzt, die „gemäßigt“ objektorientierten (prozedurale Grundsprache, erweitert um objektorientierte Möglichkeiten wie Klassendefinition, Kapselung etc.) wie C++ erfreuen sich dagegen zunehmender Beliebtheit.

Eine immer größere Bedeutung erlangen außerdem (interpretierte) Sprachen zum *(Rapid) Prototyping*, d. h. zur schnellen Umsetzung einer algorithmischen Idee in Code. Hierfür sind Sprachen wie C oder gar C++ denkbar ungeeignet, da – insbesondere bei kleineren Programmen – der Overhead für Deklarationen etc. einen signifikanten Teil der Programmier- und Testzeit beansprucht. Diesen (für die Endprodukte durchaus sinnvollen) Aufwand will man beim Rapid Prototyping aber gerade vermeiden. In diese Klasse von Sprachen fallen zum einen die aus der völlig anderen Welt der Systemadministration entstammenden Script-Sprachen, wie sie jede UNIX-Shell in einfacher Form zur Verfügung stellt, oder perl, zum anderen aber auch die in den gängigen mathematischen Software-Paketen wie MATLAB, MATHEMATICA oder MAPLE integrierten Sprachen, mit denen sich Programme für Aufgaben der numerischen linearen Algebra bis hin zur zur Lösung einer Differentialgleichung elegant (da auf einem sehr hohen Level) realisieren lassen. Wir werden hierauf noch im Absatz über Werkzeuge zurückkommen.

An einem kleinen Beispiel wollen wir einige der unterschiedlichen Konzepte näher studieren. Dazu betrachten wir die *Quadratur nach Archimedes*. Archimedes gab als Erster eine Formel für die Berechnung der Fläche unter einer Normalparabel an, indem er sie durch eine (unendliche) Folge von Dreiecken mit geometrisch abfallendem Flächeninhalt ausschöpfte, siehe Abb. 4.9.

Wir wollen diesen Ansatz auf die rekursive Berechnung des bestimmten Integrals einer Funktion f verallgemeinern. Hierzu definieren wir das Funktional

$$F(x \mapsto f(x), a, b) \;:=\; \int_a^b f(x)\ \mathrm{d}x\,,$$

das das zu berechnende Integral bezeichnet, und zerlegen es gemäß

$$\begin{aligned} F(x \mapsto f(x), a, b) &= S(x \mapsto f(x), a, b) + T(x \mapsto f(x), a, b) \\ &= S(x \mapsto f(x), a, b) + (b-a) \cdot \frac{f(a)+f(b)}{2} \end{aligned}$$

in die Funktionale T, welches die Fläche des trapezförmigen Sockels angibt, und S (vgl. Abb. 4.10). Für das Funktional S kann man nun die folgende rekursive Beziehung aufstellen:

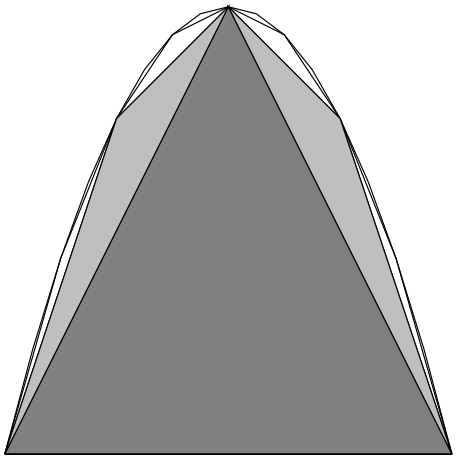


Abbildung 4.9: Rekursive Zerlegung des Parabelsegments in Dreiecke

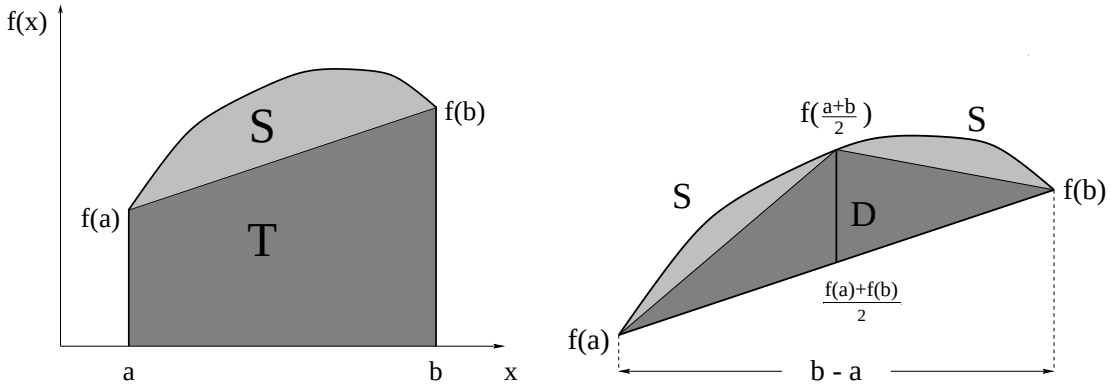


Abbildung 4.10: Segment S (links) und rekursive Definition (rechts)

$$S(x \mapsto f(x), a, b) \;=\; D(x \mapsto f(x), a, b) \;+\; S\left(x \mapsto f(x), a, \frac{a+b}{2}\right) \;+\; S\left(x \mapsto f(x), \frac{a+b}{2}, b\right),$$

wobei

$$D(x \mapsto f(x), a, b) \;:=\; \frac{b-a}{2} \cdot \left(f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2}\right)$$

gerade den Flächeninhalt des einbeschriebenen Dreiecks angibt (siehe Abb. 4.10). Dies ist ein schönes Beispiel für einen rekursiven divide-et-impera Ansatz: Reduziere die Komplexität eines gegebenen Problems, indem seine „einfachen“ Teile (hier die Flächenbestimmung von Trapez und Dreieck) berechnet werden und indem für die unbekannten Größen funktionale Beziehungen aufgestellt werden. Natürlich brauchen wir noch eine Abbruchbedingung: Die Rekursion soll beendet werden, wenn der Flächeninhalt des Dreiecks unter eine vorgegebene Genauigkeit ε fällt. Mit den obigen Definitionen und Beziehungen haben wir im Prinzip bereits ein funktionales Programm aufgestellt. Mit einem geeigneten Compiler könnten wir loslegen, was zeigt, dass die Programmierung eleganter und weniger fehleranfällig als in prozeduralen Sprachen ist.

Doch nun zu etwas wirklichkeitsnäheren Prototypen. Das Programmstück in Abb. 4.11 realisiert die eindimensionale numerische Quadratur nach Archimedes in perl, das Programm in Abb. 4.12 erledigt (hoffentlich) dasselbe in MAPLE.

MAPLE und perl können folglich schon so neckische Dinge wie Rekursion, womit FORTRAN ja nicht viel anzufangen weiß. Auch diese beiden Beispiele zeigen, wie hilfreich solche auf einer hohen

```
$integral = ar1 (0,1,0.001);
sub f   { my ($x) = @_;
    2 + $x * (1-$x);
 }
sub ar1 { my ($a,$b,$eps) = @_;
      (f($a)+f($b)) * ($b-$a)/2 + s1($a,$b,$eps);
 }
sub s1  { my($a,$b,$eps) = @_;
    my ($d);
    $d = (f(($a+$b)/2) - (f($a)+f($b))/2) * ($b-$a)/2;
    if (abs($d) < $eps) { $d * 4/3; }
    else  { $d + s1($a,($a+$b)/2,$eps) + s1(($a+$b)/2,$b,$eps);}
}
```

Abbildung 4.11: Eindimensionale Quadratur nach Archimedes in perl

```
> s:= proc(f,x,a,b,eps) local value;
value:=(subs(x=(a+b)/2,f)-(subs(x=a,f)+subs(x=b,f))/2)*(b-a)/2;
if abs(evalf(value))<eps then value
else value+s(f,x,a,(a+b)/2,eps)
+s(f,x,(a+b)/2,b,eps) fi; end;
```

$s := \mathbf{proc}(f,\ x,\ a,\ b,\ eps)$
$\mathbf{local}\ value;$
$value := (\mathrm{subs}(x = (a+b)/2,\ f)$
$\quad - (\mathrm{subs}(x = a,\ f) + \mathrm{subs}(x = b,\ f))/2) \times (b-a)/2;$
$\mathbf{if}\ \mathrm{abs}(\mathrm{evalf}(value)) < eps\ \mathbf{then}\ value$
$\mathbf{else}\ value + \mathrm{s}(f,\ x,\ a,\ (a+b)/2,\ eps)$
$\quad + \mathrm{s}(f,\ x,\ (a+b)/2,\ b,\ eps)$
$\mathbf{fi}$
$\mathbf{end}$

```
> ar:= proc(f,x,a,b,eps)
(subs(x=a,f)+subs(x=b,f))*(b-a)/2 +s(f,x,a,b,eps); end;
```

$ar := \mathbf{proc}(f,\ x,\ a,\ b,\ eps)$
$(\mathrm{subs}(x = a,\ f) + \mathrm{subs}(x = b,\ f)) \times (b-a)/2 + \mathrm{s}(f,\ x,\ a,\ b,\ eps)$
$\mathbf{end}$

Abbildung 4.12: Eindimensionale Quadratur nach Archimedes in MAPLE

Ebene ansetzenden Sprachen bei der schnellen Implementierung von Prototypen sein können. In puncto Laufzeit darf man sich aber keine Illusionen machen: Programmläufe für „echte “ Anwendungen kann man so nicht abwarten.

4.2.2 Algorithmen und Datenstrukturen

Die klassische Sprache der Simulationscodes, FORTRAN, lebte bis zur Version FORTRAN 90 mit *statischen* Strukturen, deren Größe zur Übersetzungszeit feststeht. Dies gilt für Datenstrukturen, bei denen array-artige Gebilde dominierten, sowie für Unterprogramme, bei denen der Speicherbedarf, den ein Aufruf nach sich zieht, im Wesentlichen bekannt ist.

Sprachen wie Pascal oder C und eben auch FORTRAN 90 erlauben dagegen zudem *dynamische*

Größen, also etwa Felder variabler Länge sowie Geflechte im Datenstrukturbereich oder rekursive Prozeduren, bei denen gekellert werden muss. Diese Möglichkeit der Dynamik ist für viele Simulationsanwendungen von herausragender Bedeutung (automatische Schrittweitensteuerung bei gewöhnlichen Differentialgleichungen, adaptive Gitterverfeinerung bei partiellen Differentialgleichungen).

Ein beliebtes Lehrbuchbeispiel für eine dynamische bzw. rekursive Datenstruktur liefert die Syntax arithmetischer Ausdrücke, die in Pascal-artiger Notation wie folgt aufgeschrieben werden kann:

```
type  expr  =  record  op1: term;
                       op2: term;
                       op: operator
               end;
type  term  =  record  case t boolean of
                         true : (id: real)
                         false : (expression: expr)
               end;
```

Andere (und für uns spannendere) Beispiele sind verkettete Listen oder Binärbäume, beides Spezialfälle von *Geflechten*. Diese werden über *Zeiger* (*Pointer, Verweise*) realisiert, der (für jede rekursive Struktur erforderliche) Abbruch erfolgt über den leeren Zeiger.

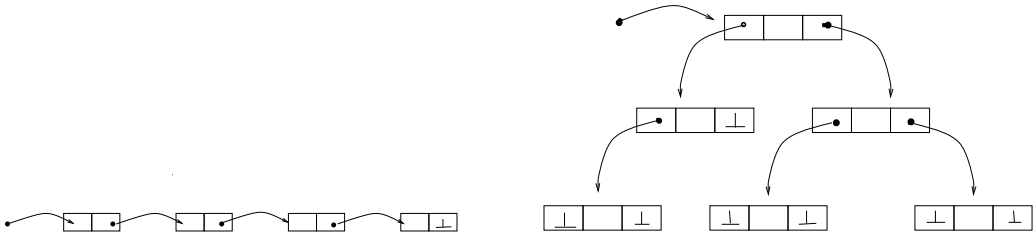


Abbildung 4.13: Lineare Liste und Binärbaum

Lineare Listen bieten eine Möglichkeit zur kompakten Speicherung dünn besetzter Matrizen. Erinnern wir uns an die Diskretisierung von Ableitungen mittels finiter Differenzen in Kapitel 3. Die entstehende Matrix wird bestimmt von den verwendeten Differenzenquotienten. Da nur lokale Gitterpunkte wechselwirken, sind die Matrizen dünn besetzt, enthalten also pro Zeile nur ein paar Nicht-Nullen. Deshalb ist die übliche Speicherung über ein zweidimensionales Feld (array) äußerst ineffizient. Als Abhilfe bietet sich an, die Matrix als Vektor von Zeilen abzuspeichern, wobei jede Zeile als verkettete Liste mit den Eintragspaaren (Spaltennummer, Matrixelement) realisiert ist.

Binärbäume werden beispielsweise bei finiten Elementen zur Speicherung der in Abschnitt 3.1 vorgestellten hierarchischen Basis verwendet. Allgemein sind Geflechte zur Implementierung von Finite-Element-Netzen die übliche Datenstruktur. Man stelle sich etwa im zweidimensionalen Fall ein adaptiv verfeinertes Dreiecksgitter vor, in dem die einzelnen Elemente auf verschiedene Art und Weise (Unterteilung der längsten Seite, Halbierung aller drei Seiten) in eine unterschiedliche Zahl von Sohn-Dreiecken zerlegt werden können. Bei den sog. *unstrukturierten* Gittern, bei denen die Lage und Nachbarschaftsrelationen der einzelnen Punkte und Elemente nicht automatisch aus der Hierarchie folgen (wie dies etwa bei den strukturierten Rechtecksgittern (Tensorprodukt-Netzen) der Fall ist), sondern explizit gespeichert werden müssen, ist es nämlich mit einem einfachen Baum nicht getan. Hier müssen zusätzlich zur Repräsentation der Hierarchie beispielsweise Kanten identifiziert, Dreiecke mit Kanten und Kanten mit Punkten in Bezug gebracht werden etc. Hierfür sind weitere „Verzeigerungen“ erforderlich, die den Umfang der eigentlich zu speichernden Daten (Funktionswerte) bei weitem übersteigen können. Und das ist nur der zweidimensionale Fall: bei einem adaptiven Tetraedergitter wird alles noch erheblich aufwendiger! Ein Beispiel: Obwohl im Strömungsfall an jedem Gitterpunkt nur drei oder vier Variablen anfallen (die Komponenten der

Geschwindigkeit und der Druck), kann der Gesamt-Speicherbedarf pro Knoten Werte von ca. 0.5 kB (2 D) oder 1.5 kB (3 D) pro Gitterpunkt erreichen.

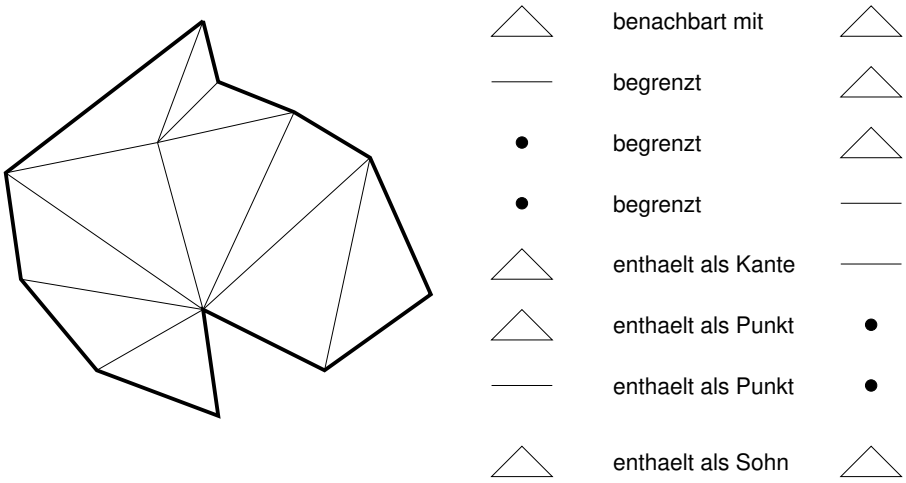


Abbildung 4.14: Verzeigerung bei unstrukturierten FEM-Netzen: Die einzelnen Relationen können als Tabellen, als Einzelgeflechte oder in einem integralen Geflecht organisiert werden.

Dem Vorteil der Dynamik, der Geflechte geradezu ideal für adaptiv verfeinerte Netze oder Gitter macht, stehen einige Nachteile gegenüber, die (mit) für die bereits angesprochenen Probleme zahlreicher mit solchen Strukturen arbeitender Simulationscodes hinsichtlich Laufzeiteffizienz sind (Verfehlen der Peak-Performance um Größenordnungen):

- indirekte Adressierung: Eine Verletzung des Lokalitätsprinzips ist möglich, benachbarte Elemente bzw. Punkte, die wechselwirken, können physikalisch weit entfernt im Hauptspeicher liegen (schlecht für Vektorisierung (effiziente Bearbeitung von referenzierten Daten in den Vektorprozessoren erst in den neueren Generationen teilweise möglich), Parallelisierung (Partitionierung komplizierter), Speicherzugriffszeit (Cache-Wirkung droht zu verpuffen, wenn benachbarte und wechselwirkende Größen nicht im Speicher beieinander liegen)).
- eingeschränkte Möglichkeit des Direktzugriffs auf Knoten (vielmehr aufwendiges „Durchhangeln“)
- Bäume können zu Listen degenerieren ⇒ Balancierung erforderlich (wie man das etwa von AVL-Bäumen kennt); balancierte Bäume sind wichtig etwa für die Parallelisierung (sonst ist die Last zu ungleich auf die Prozessoren verteilt)
- organisatorischer Overhead der Verzeigerung (bei Tetraedernetzen z. T. einige kByte pro Knoten)

Zumindest im Falle strukturierter Gitter gibt es jedoch Alternativen zu den Geflechten, d. h. zu mit Hilfe von Zeigern realisierten Datenstrukturen. Die adaptiv verfeinerten strukturierten Gittern zugrunde liegenden Bäume können auch linearisiert abgespeichert werden, beispielsweise als *preorder (depth first) traversal* (siehe Abb. 4.15). Das erspart die Verzeigerung, erfordert jedoch ein Verschieben des Restvektors bei nachträglicher Einfügung oder Entfernung von Punkten.

Eine weitere Alternative zu Geflechten bei der Abspeicherung dynamischer und gestreuter Datensätze können *Hash-Tabellen* sein. Die übliche Situation, bei der Hash- oder Streuspeicherverfahren zum Einsatz kommen, ist der Fall eines sehr großen *Universums* U von Elementen, aus dem aber i. A. nur wenige Elemente $x \in U$ wirklich benötigt werden. Jedes Element x wird eindeutig durch einen *Schlüssel* $k(x)$ identifiziert. Man denke etwa an die Einträge in einer Personenkartei (Nachnamen als Schlüssel), an diskrete Gitterpunkte mit der zugehörigen Information (Variablen)

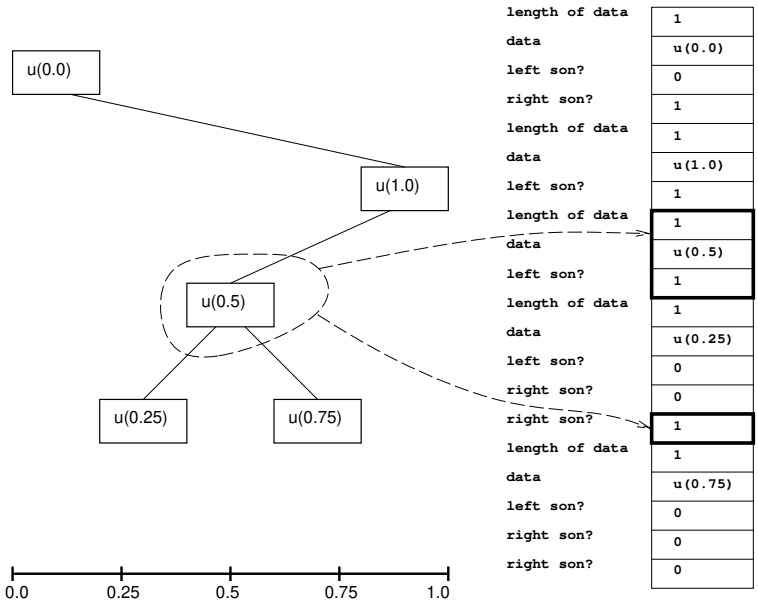


Abbildung 4.15: Beispiel der Linearisierung einer Baumstruktur (depth first)

in einem Grundgebiet (Kooedinaten als Schlüssel) oder an die Nicht-Null-Einträge einer dünn besetzten Matrix (Positionen als Schlüssel). Im Beispiel der stückweise linearen hierarchischen Basis im Eindimensionalen aus Kapitel 3 könnte man als Schlüssel eine hierarchische bzw. eine levelweise Nummerierung der Gitterpunkte wählen (siehe Abb. 4.16). Im d-dimensionalen bieten sich d-Tupel aus solchen Nummern als Schlüssel an. Im folgenden wollen wir der Einfachheit halber zwischen

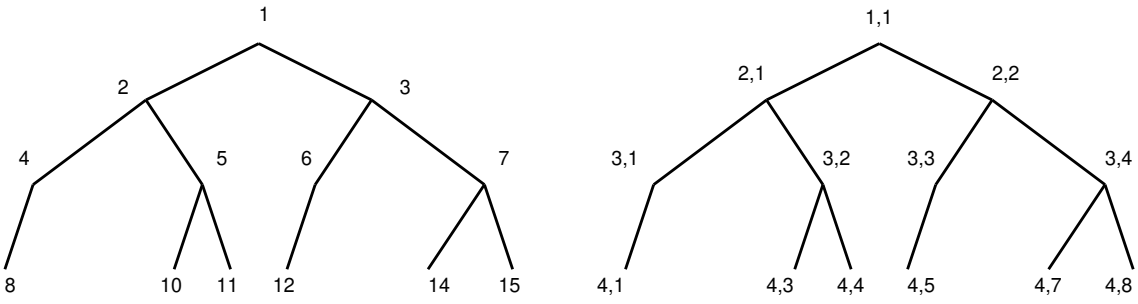


Abbildung 4.16: Schlüsselzuordnung bei der hierarchischen Basis: total hierarchische (links) und levelweise hierarchische Nummerierung (rechts)

Elementen und Schlüsseln nicht unterscheiden. In allen genannten Fällen wäre es irgendwo zwischen ungeschickt und unmöglich, für jedes potentiell auftretende Element einen Speicherplatz zu reservieren. Deshalb sehen wir einen statischen Speicherbereich zur Aufnahme der aktuellen Schlüssel vor und definieren eine geeignete *Hash-Funktion* h, die jedem Element bzw. Schlüssel seinen Platz in diesem Bereich zuordnet:

$$\text{Gitterpunkt } x \in [0,1]^d \quad \text{bzw.} \quad \text{Schlüssel } k(x) \qquad \overset{h}{\mapsto} \qquad \text{Adresse } i \text{ im array } a[0:n-1]\,.$$

Typischerweise wird $n \in [m : 2 \cdot m]$ gewählt, wenn m die Anzahl der (erwarteten) auftretenden

Schlüssel bezeichnet. Beispiele für Hash-Funktionen sind etwa

$$\begin{aligned} h(k) &:= k \bmod n\,, \\ h(k) &:= \lfloor n \cdot ((k-a) \bmod 1) \rfloor\,, \qquad a = \frac{\sqrt{5}-1}{2}\,, \end{aligned}$$

wobei n keine Zweierpotenz, keine Zehnerpotenz und nicht von der Art $2^j - 1$ sein sollte, bspw. prim und „nicht zu nahe“ an einer Zweierpotenz. Es ist klar, dass die Hash-Funktion h das Herzstück der ganzen Sache ist und mit Bedacht konstruiert werden muss. Einerseits soll h möglichst gut streuen, andererseits aber billig zu berechnen sein.

Trotz aller Bemühungen kann es natürlich vorkommen, dass zwei verschiedene Schlüssel denselben Platz zugewiesen bekommen. Da wir ja Platz sparen wollen, kann h sinnvollerweise nicht injektiv sein. Bei einer solchen *Kollision* müssen geeignete Maßnahmen ergriffen werden wie etwa die Einführung von *Überlauflisten* (der Platz im Array zeigt auf eine lineare Liste, in der alle hierauf abgebildeten Schlüssel untergebracht sind; ermöglicht die (virtuelle) Speicherung beliebiger vieler Daten im Hash-Feld), das sog. *Probing* (Suche nach freien Plätzen vom belegten Platz ausgehend; führt zu Clusterbildung) oder das *double hashing* (Anwendung einer zweiten Hash-Funktion auf der Grundlage von Schlüssel und erster (belegter) Position). Dass solche Kollisionen gar nicht so selten sein werden, sieht man am Beispiel der Geburtstage: Bereits bei 23 Personen ist die Wahrscheinlichkeit, dass wenigstens zwei von ihnen am selben Tag Geburtstag haben, größer als 50 Prozent. Deshalb kann das alles nur eine gewisse Zeit gut gehen. Spätestens, wenn die Anzahl der aktuellen Schlüssel in die Nähe von n, der Größe unseres Speicherbereichs kommt, wird die Kollision zum Regelfall. Der Hash-Bereich „platzt“ entweder aus allen Nähten, oder aber die Überlauflisten werden immer länger und dominanter, so dass die Listencharakteristik wieder das Sagen hat. Deshalb muss jetzt eine Neuorganisation (*Rehashing*) mit neuem (d. h. größerem) n und neuem h erfolgen. Während sich Personaldateien i. A. in ihrer Größe nur langsam ändern, gilt für unsere Anwendungen, dass der Gesamtbedarf an Gitterpunkten oftmals a priori nur schlecht abzuschätzen ist (etwa, wenn eine bestimmte zu erzielende Genauigkeit vorgegeben ist) und dass sich bei Adaptivität die Gitterpunktzahlen u. U. schnell und stark ändern können. Auf solche Rehashing-Schritte müssen wir uns also einstellen, wenn wir adaptiv verfeinerte Gitter, wie sie bei der Diskretisierung von PDEs entstehen, mittels Hash-Technik organisieren wollen. Festzuhalten ist, dass Hash-Techniken eine exzessive Verzeigerung zu vermeiden helfen. Die Lokalität ist besser (bei Verfahren ohne Überlauflisten zum. liegen die Daten innerhalb genau definierter Bereiche und nicht wahllos im Speicher), aber nicht perfekt (keine Lokalität innerhalb des Hash-Felds).

So viel zu dynamischen (rekursiven) Datenstrukturen. Auch bei den Algorithmen zur Lösung von Teilaufgaben erweist sich die Rekursion als vorteilhaft. Unter Rekursion versteht man bekanntlich die Rückführung einer gegebenen Aufgabe auf eine „einfachere“ Aufgabe desselben Typs. Diese Problemreduktion kann dabei sehr unterschiedlich aussehen. Einige Beispiele:

- *direkte* Rekursion: Berechnung der Fakultät

  $fac(n) = n \cdot fac(n-1)$
  $fac(1) = 1$

- *indirekte* Rekursion: Test auf gerade/ungerade

  $iseven(n) = isodd(n-1)$
  $iseven(1) = FALSE$

  $isodd(n) = iseven(n-1)$
  $isodd(1) = TRUE$

Wann ist die Formulierung als rekursiver Algorithmus sinnvoll und wann nicht?

- egal: Fakultät

Bei der Berechnung der Fakultät haben obige Rekursion sowie die naheliegende Iteration

$$fac := 1; \quad i = 2..n: \quad fac := fac \cdot i;$$

dieselbe Komplexität (sowohl im Hinblick auf das Niederschreiben als auch, was die Rechenzeit angeht).

- schädlich: Fibonacci-Zahlen ($a_0 = a_1 = 1, \quad a_n = a_{n-1} + a_{n-2}$ für $n \geq 2$)

  Im rekursiven Programm

```
function fib(n: integer): integer;
  if n=0 then fib:=1 else
  if n=1 then fib:=1 else
            fib:= fib(n-1) + fib(n-2);
```

  wächst die Anzahl der Aufrufe exponentiell, es kommt zu Mehrfachaufrufen mit demselben Argument. Eine solche *kaskadische* Rekursion ist der einfachen Iteration

$$x_{alt} := 1; \quad x_{neu} := 1; \quad i = 2..n: \quad x_{neu} := x_{alt} + x_{neu}; \quad x_{alt} := x_{neu} - x_{alt};$$

  hoffnungslos unterlegen!

- nützlich:

  Hier sind die Fälle rekursiv definierter Daten (z. B. Bäume) sowie rekursiv formulierte Aufgabenstellungen zu nennen:

  - Selbstähnlichkeit, Fraktale
  - Hilbert-Kurven zeichnen (vgl. den Abschnitt über Lastverteilung)

  Ferner fallen in die Kategorie „nützlich“ Problemstellungen mit starker „Eigendynamik“:

  - rekursive Substrukturierung bei der Finite-Elemente-Methode in der Statik
  - adaptive Unterteilung und rekursive Substrukturierung bei Radiosity-Verfahren in der Computergraphik
  - Backtracking: finde einen Weg aus einem Labyrinth bzw. finde eine Springerzugfolge, die genau einmal über jedes Feld des Schachbretts führt

  Rekursion ist naturgemäß auch dann ein hilfreiches Prinzip, wenn die Ausgangsaufgabenstellung zu komplex ist und nicht direkt angegangen werden kann (→ divide & conquer). In diese Kategorie fallen die Beispiele zum Sortieren aus der Einleitung sowie die rekursive Quadratur nach Archimedes, insb. im höher-dimensionalen Fall.

4.2.3 Aspekte des Software-Engineering

Mit dem Fortschreiten der Computerisierung unserer Welt ist die *Software-Technik* oder das *Software-Engineering* zu einem eigenen Forschungsgebiet und zu einem Grundpfeiler der modernen Informatik geworden. Auch wenn der Typus des „Hackers“ so schnell nicht aussterben wird, so sind doch große Software-Systeme zu komplex und langlebig und oft an zu neuralgischen Stellen im Einsatz, als dass man sich länger auf die gute alte Programmierung ins Blaue getreu dem Motto „trial and error“ verlassen könnte. Inzwischen sind alle Lebensphasen von Software von einer tiefgreifenden Systematisierung erfasst. Wenn dies im (uns interessierenden) Bereich von Programmen zur numerischen Simulation erst in einem viel geringeren Umfang der Fall ist als etwa im klassischen Datenverarbeitungssektor oder im kaufmännischen Bereich (bis hin zu Gesamtlösungen à la

R/3[12]), so hat dies mehrere Gründe:

- Software für Aufgabenstellungen des wissenschaftlichen Rechnens entsteht schwerpunktmäßig in mathematisch, ingenieur- oder naturwissenschaftlich geprägten Arbeitsgruppen. Informatiker und dementsprechend auch informatisches Gedankengut sind da eher randständig.
- Die Entwicklung von Programmen zur numerischen Simulation findet nach wie vor hauptsächlich im akademischen sowie allenfalls im vor-kommerziellen Bereich statt. Die Codes werden also v. a. an der Hochschule oder an einschlägigen Forschungsinstituten im Rahmen von Diplomarbeiten, Dissertationen oder Forschungsprojekten erstellt, was sowohl einem längerfristigen Vorgehen als auch einem intensiven Teamwork oftmals im Wege steht. Da Personalkosten zudem an Universitäten eine weitaus geringere Rolle spielen als in der Industrie, wird auf eine effiziente Programm-Entwicklung in vielen Fällen nur wenig Wert gelegt. Eine gezielte Verbreitung wird meistens überhaupt nicht angestrebt, und wenn, dann als Public Domain Software, über kostenfreie Lizenzen oder allenfalls als Shareware[13] – in der Regel jedoch ohne die Übernahme von Gewährleistung. Nur im Ausnahmefall erfolgt eine kommerzielle Verwertung, die dann zudem meist (halb-kommerzielle) Spin-off-Unternehmen, also etwa Ausgründungen von Lehrstühlen, vornehmen. Ohne Verbreitung und Verwertung entfällt aber oft der Zwang zu Dokumentation, Wartung und regelmäßigen Updates etc.
- Simulationsprogramme werden oftmals immer noch als maßgeschneiderte Lösung für ein spezielles Problem entwickelt, nicht jedoch als multi-purpose-Lösungen. Die Möglichkeit einer größeren Allgemeinheit wird oft sogar von den „Experten“ verneint – hauptsächlich aus Effizienzgründen, aber auch, um der eigenen Expertise hinreichend Geltung zu verschaffen.
- Es gibt nur wenige international tätige und größere Software-Häuser, die sich auf das hier behandelte Gebiet spezialisiert haben. Fündig wird man am ehesten noch bei Finite-Elemente-Codes für Anwendungen aus der Strukturmechanik, wo beispielsweise die Programme NASTRAN, ANSYS oder ADINA Referenz-Charakter haben. Im Bereich der Strömungsmechanik wird das Eis schon dünner, und in vielen naturwissenschaftlichen Feldern sucht man meist vergeblich nach kommerziellen Produkten.
- Simulationsprogramme sind zumeist (noch) Programme mit einer zwar komplizierten Entwicklung, einer jedoch trotz langer Laufzeiten einfachen Ablaufcharakteristik (als Batch-Job ohne Interaktivität). Angesichts eines solchen im Vergleich zu einem Produkt wie bspw. R/3 geradezu primitiven Einsatz-Szenarios erscheint die Laufzeit-Effizienz oft als wichtiger als etwa Strukturierung oder Modularität.

Trotzdem gibt es eine ganze Reihe von Argumenten für den verstärkten Einsatz moderner Techniken des Softare-Engineering auch im wissenschaftlichen Rechnen:

- Die Aufgabenstellungen werden immer komplexer und immer umfangreicher. Damit wird das klassische „one-man-one-Ph.D.“-Programm zunehmend zur Ausnahme: An der Entwicklung der Programme sind mehrere Personen über einen drei Jahre übersteigenden Zeitraum hinweg beteiligt. Dadurch verlassen Themenkreise wie Konzeption, Implementierungsdetails, Schnittstellendefinition oder Dokumentation notwendigerweise den Bereich des Privatvergnügens eines Programmierers und müssen geplant, abgesprochen und entsprechend realisiert werden. Zudem müssen Programme über einen längeren Zeitraum hinweg wart-, modifizier- und erweiterbar bleiben. Deshalb kommt man in Zukunft um eine Beschäftigung etwa mit modularen oder objektorientierten Ansätzen nicht herum. Auch wenn das langfristig FORTRAN Probleme bereiten dürfte, so ist es doch nur sekundär eine Frage der

[12] R/3 ist das Hauptprodukt der Walldorfer Software-Schmiede SAP und dient zur Erfassung und Steuerung der gesamten Abläufe in Unternehmen, also bspw. Buchhaltung, Controlling, Materialkontrolle, Qualitätsmanagement, Verkauf, Personal- und Projektmanagement.

[13] Unter Shareware versteht man Software, die (zunächst) kostenfrei abgegeben wird, deren Nutzung dem Anwender jedoch nur befristet oder eingeschränkt gestattet ist. Oft handelt es sich hierbei um befristete Probier-Lizenzen oder um „abgespeckte“ Demo-Versionen kostenpflichtiger Produkte – Appetit-Häppchen, sozusagen.

Programmiersprache.

- Aus ökonomischen Gründen wird auch hier der Anteil an außer Haus erstellter Standardsoftware steigen (in den USA lag bereits Ende der achtziger Jahre der Anteil an Standardsoftware über alle Branchen hinweg bei zwei Dritteln, mit steigender Tendenz). Dies schließt ganze Programme ein, etwa zum Pre- und Postprocessing (Gittergenerierung, Visualisierung), aber auch die Verwendung einzelner Programmmodule (z. B. kann ein schneller Löser für lineare Gleichungssysteme aus einer Programmbibliothek geladen und in die eigene Anwendung eingebunden werden). Interoperabilität und Kompatibilität werden an Bedeutung gewinnen.
- Aus ökonomischen Gründen werden reine one-purpose-Lösungen verschwinden. Da der Anteil an Drittmittelfinanzierung an den Hochschulen langfristig zunehmen wird, werden industrielle Anforderungen eine immer größere Rolle spielen. Ein industrieller Kontext gestattet aber nicht die zeitintensive Entwicklung maßgeschneiderter Modelle und Codes, sondern erfordert vielmehr die Möglichkeit kurzfristiger Simulationslösungen durch ein weitgehendes Abstützen auf vorhandene Module. Damit nimmt die Bedeutung der reinen Lauftzeit-Effizienz im Vergleich zur Entwicklungs-Effizienz ab.
- Simulationsprogramme werden in zunehmendem Maße nicht mehr isoliert von Experten, sondern in einen größeren Kontext eingebettet und weitgehend automatisiert bzw. von Nicht-Kennern der Codes aufgerufen. Solche Einbettungen sieht man heute schon beispielsweise bei Echtzeit-Simulationen, bei der Ankopplung von CAD an die Simulation oder bei der Einbindung von Simulationswerkzeugen in den Gesamtablauf des Unternehmens: Der Designer ändert die Karosserie eines Autos, ein Strömungsprogramm ermittelt zu diesen Vorgaben die optimale Platzierung der Einströmdüsen des Belüftungssystems, und das Kostenkalkulationsprogramm berechnet aus diesen Daten wiederum die Herstellungs- und Einbaukosten der Klimaanlage. Wir haben es also mit einer wachsenden Komplexität der Einsatz-Szenarios von Simulationscodes zu tun.

Im Folgenden sollen einige Aspekte des Software-Engineerings diskutiert werden, die für unsere Thematik als besonders wichtig erscheinen. Grundsätzlich kann man die Hauptbereiche *Software-Entwicklung*, *Software-Management*, *Qualitätssicherung* und *Wartung* unterscheiden.

Die *Entwicklung* von Software-Systemen wird in mehrere Phasen untergliedert:

- die *Planungsphase*:
  - Auswahl des Produkts
  - erste Voruntersuchungen (wichtigste Anforderungen (Pflichtenheft), Funktionen, Daten etc.)
  - Machbarkeitsstudien
  - Aufwands- und Terminschätzungen
- die *Definitionsphase*:
  - Definition bzw. Spezifikation des Produkts (Daten, Funktionen, Dynamik und Benutzeroberfläche betreffend), d. h. vollständige und konsistent formulierte Anforderungen und Funktionalitäten
  - verschiedene Techniken, z. B. Funktionsbäume, Datenflussdiagramme, Entity-Relationship-Modelle, objektorientierte Ansätze, Regelmodelle, Automaten, Petri-Netze etc.)
- die *Entwurfsphase* („Programmieren im Großen“):
  - Entwickeln einer software-technischen Lösung im Sinne einer Software-Architektur
  - Berücksichtigung verschiedener Einflussfaktoren (Einsatzbedingungen, Umgebungsbedingungen (Zielplattform etc.), Qualitätsbedingungen)

  - Treffen von Grundsatzentscheidungen (Datenhaltung, Benutzeroberfläche, Hilfssysteme etc.)
  - Spezifikation des Funktions- und Leistungsumfangs
  - Festlegung von Schnittstellen
  - Konzepte auf verschiedenen Abstraktionsebenen (keine Abstraktion (Assembler), funktionale Abstraktion (abstrakte Funktionen und Prozeduren), abstrakte Datenobjekte, abstrakte Datentypen, objektorientierter Entwurf)
  - mögliche Vorgehensweisen:
    strukturierter Entwurf: Entwurf via hierarchisch angeordnete funktionale Module, Strukturdiagramme und Modulspezifikation; rein funktionale Abstraktion, d. h. Algorithmen werden vor dem Anwender verborgen;
    modularer Entwurf: zusätzlich Datenabstraktion, d. h. Datentypen und Zugriffsalgorithmen werden vor dem Anwender verborgen;
    objektorientierter Entwurf: noch höherer Abstraktionsgrad mit Klassenhierarchie und Vererbung;

- die *Implemetierungsphase* („Programmieren im Kleinen“):
  - Konzeption von Datenstrukturen und Algorithmen
  - Programmstrukturierung
  - Dokumentation
  - Umsetzung der Konzepte in die Konstrukte der Zielsprache
  - Komplexitätsbetrachtungen und Tests
  - Endprodukte: Quellcode mit Dokumentation, Objektcode, Testprotokoll
  - Prinzipien:
    Verbalisierung (aussagekräftige, also möglichst mnemonische Namensgebung für Variablen, Prozeduren etc., geeignete Kommentare);
    problemadäquate Datentypen;
    schrittweise Verfeinerung (Strukturierung durch Abstraktionsebenen);
    integrierte Dokumentation;
- die *Abnahme- und Einführungsphase*:
  - Produktübergabe mit Abnahmetests inkl. Abnahmeprotokoll
  - Installation, Schulung, Inbetriebnahme
- die *Wartungs- und Pflegephase*:
  - korrigierende Aktivitäten (ca. 20%)
  - anpassende Aktivitäten (ca. 25%)
  - perfektionierende Aktivitäten (ca. 55%)

Das *Software-Management* hat als wichtige zu optimierende Zielgrößen die Qualität des Programmsystems und den Herstellungsaufwand. Diese Thematik, die Aktivitäten in den Bereichen Planung, Organisation, Personal, Leitung und Kontrolle umfasst, hat offensichtlich zahlreiche Berührpunkte mit allgemeinen Fragestellungen der Betriebswirtschaftslehre, wobei die besonderen Eigenschaften von Software (Software ist immateriell, Entwicklungsfortschritte sind nicht oder nur schwer objektiv zu ermitteln und zudem nicht-deterministisch) hier zu berücksichtigen sind. Wir wollen an dieser Stelle nicht in die Diskussion einsteigen.

Bei der *Qualitätssicherung* stellt sich zunächst die Frage, wie der Begriff der Qualität bei Software zu definieren ist. In der Norm DIN ISO 9126 werden beispielsweise sechs Qualitätmerkmale definiert: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit (Portabilität). Welches Merkmal wie wichtig bzw. wie ausgeprägt ist, ist sicher von Fall zu Fall verschieden. (Laufzeit-) Effizienz ist traditionell die kritische Eigenschaft schlechthin im wissenschaftlichen Rechnen. Portabilität hat durch die Vielzahl verschiedener Parallelrechnerarchitekturen stark an Bedeutung gewonnen, verliert aber wieder an Gewicht durch den wachsenden Einfluss standardisierter Programmiermodelle (MPI z. B.). Benutzbarkeit und Änderbarkeit führen üblicherweise ein Schattendasein – viele Simulationscodes sind nur von ihren Entwicklern zu bedienen, geschweige denn zu ändern. Die Funktionalität beinhaltet Aspekte wie Korrektheit – traditionell wichtig – oder Interoperabilität (Fähigkeit zur Zusammenwirkung mit anderen Systemen) – zunehmend wichtig. Zuverlässigkeit meint vor allem eine geringe Zahl von Versagensfällen, Fehlertoleranz und Wiederherstellbarkeit. Ersteres ist immer wichtig, die beiden letzteren Punkte sind (noch) eher sekundär: Jeder Aufwand nach einem Absturz bspw. ist lästig, das Risiko eines ernsten Datenverlusts bei Simulationsprogrammen ist aber eher gering.

Bei den Maßnahmen zur Erzielung eines hohen Qualitätsstandards unterscheidet man zwischen *konstruktiven* und *analytischen* Maßnahmen sowie zwischen *produktorientierten* und *prozessorientierten* Maßnahmen:

- Konstruktive Maßnahmen sind Methoden, Werkzeuge, Richtlinien etc., die a priori bestimmte Eigenschaften sicherstellen.
  Bsp.: Der Einsatz einer Sprache mit statischer Typprüfung verhindert das Auftreten von Typfehlern während der Laufzeit.
- Analytische Maßnahmen sind diagnostischer Art und dienen zur Evaluierung des Qualitätsniveaus.
- Produktorientierte Maßnahmen betreffen die Qualität des Endprodukts.
- Prozessorientierte Maßnahmen betreffen die Qualität des Erstellungsprozesses.

Während früher der Schwerpunkt auf den produktorientierten Maßnahmen lag (im konstruktiven Bereich durch geeignete Vereinbarungen, im analytischen durch automatisierte oder manuelle Prüfmethoden), haben sich in jüngerer Zeit Ansätze zur Verbesserung der Prozessqualität durchgesetzt, um so indirekt eine hohe Produktqualität sicherzustellen. Das vielleicht prominenteste Beispiel hierfür ist das Normenwerk nach ISO 9000. Es legt für allgemeine Auftraggeber-Lieferanten-Verhältnisse einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung von materiellen und immateriellen Produkten, also bspw. Software, fest. Software-Hersteller, deren Qualitätsmanagementsystem die dort festgelegten Forderungen erfüllt, können sich nach einem von einer unabhängigen Instanz durchgeführten *Audit* ein entsprechendes Zertifikat ausstellen lassen. Dieses Zertifikat ist heute Quasi-Standard und ist in vielen Fällen notwendige Voraussetzung für eine Auftragserteilung.

Die Produktqualität wird jedoch auch direkt avisiert. Hierfür existieren ein Vielzahl *testender*, *validierender* oder – die schärfste Form – *verifizierender*, d. h. formal beweisender Verfahren. Am Ende stehen immer der *Integrationstest*, bei dem das Zusammenspiel der Systemkomponenten geprüft wird, der *Systemtest*, bei dem das System seine Funktionsfähigkeit als Black Box zeigen muss, sowie der abschließende *Abnahmetest*.

Wer schon jemals kleinere numerische Programme oder vielleicht sogar Programme für numerische Simulationsaufgaben geschrieben hat, bei dem werden die obigen Bemerkungen zum Software-Engineering u. U. einen ambivalenten Eindruck hinterlassen, irgendwo zwischen „böhmische Dörfer“ und „belangloses Geblubber“. Fakt ist aber, dass in der modernen Software-Industrie diese Techniken nicht mehr wegzudenken sind. Im Rahmen des CASE (Computer Aided Software Engineering) wurde sogar für die Software-Erstellung eine umfassende Computerunterstützung entwickelt. Dem wird sich auch das wissenschaftliche Rechnen nicht auf Dauer entziehen können.

Abschließend wollen wir nochmals kurz auf das einführende Kapitel dieses Skripts, insb. auf die dort erwähnte „Dogmatik“ des wissenschaftlichen Rechnens zurückkommen. Denn wie für die Algorithmik, so ist auch für das Software-Engineering (und hier insbesondere für die Software-Entwicklung) zu beobachten, dass sich gewisse Prinzipien, Methoden oder Entwurfsmuster herauskristallisiert haben, die von übergeordnete Gültigkeit sind und deren Beachtung sich bei der Programmentwicklung als geradezu unerlässlich erweist. Interessanterweise treffen wir auch hier wieder auf alte Bekannte, die wir in Kapitel 1 bereits unter algorithmischen Gesichtspunkten diskutiert haben.

- *Abstraktion*: Anstelle einer konkreten Realisierung oder Implementierung betrachtet man nur das Wesentliche, etwa (implementierungsunabhängige) Eigenschaften und Beziehungen. Ein Beispiel bilden *abstrakte Datentypen*, bei denen eine bestimmte Struktur (z. B. Mengen, Keller oder Warteschlangen) mit Hilfe bestimmter Typen, Funktionen auf diesen Typen und Gesetzen formal spezifiziert wird, ohne dass man auf die Realisierung eingeht. Der Blick für das Grundlegende wird somit nicht durch (bei jeder konkreten Implementierung u. U. unterschiedlich realisierte) Details versperrt. Mathematische Modellbildung, wie wir sie in Kapitel 2 besprochen haben, ist im Grunde nichts anderes als Abstraktion. Da Abstraktion immer von der relativen Betrachtungsebene ausgeht, ist sie nichts Absolutes, sondern kann vielmehr wiederholt (bzw. rekursiv) angewandt werden. Man gelangt so zu einer Hierarchie von Abstraktionsebenen.
- *Strukturierung*: Komplizierte Systeme ohne Struktur zu erfassen, ist nahezu unmöglich – das gilt für technische Systeme wie für Programme, und für Codes ebenso wie für deren Entstehungsprozesse. Die Struktur muss dabei aber nicht statisch sein, sie darf sich weiterentwickeln. Als prominentes Beispiel kann man hier die Implementierungsphase der Software-Entwicklung heranziehen: Die in den vorausgehenden Phasen definierte Struktur muss im Programmsystem ihren Niederschlag finden.
- *Hierarchisierung*: Ohne Struktur keine hierarchische Sehweise – insofern ist dieses Prinzip ein Spezialfall der Strukturierung. Hierarchie ist eine vertikale Struktur, die eine Betrachtung auf verschiedenen Ebenen und somit beispielsweise schrittweise Verfeinerung oder Vererbungskonzepte ermöglicht.
- *Modularisierung*: Auch hierfür ist die Strukturierung Voraussetzung. Konsequente Modularisierung setzt die funktionale Struktur eines Systems und ermöglicht den direkten Komponentenzugriff und -austausch.
- *Verschattung*: Die Verschattung, auch *Geheimnisprinzip* oder *Information Hiding* genannt, stellt eine verschärfte Form der Modularisierung dar, bei der die einzelnen Module nur einen Teil ihrer internen Information nach außen geben. Im Extremfall einer *Black Box* wird nur über die definierte Schnittstelleninformation kommuniziert, das innere Geschehen ist unsichtbar (z. B. beim Aufruf eines Unterprogramms. Eine solche Vorgehensweise hilft, unerwünschte und unkontrollierbare Seiteneffekte zu verhindern.
- *Lokalität*: Die relevante Information sollte stets größtenteils lokal verfügbar sein. Andernfalls sind große Programmsysteme kaum zu verstehen oder ändern. Dies erfordert u. a. eine klare Strukturierung und Modularisierung.

Abstraktion und Hierarchisierung eröffnen zudem als grundlegenden Freiheitsgrad die Entscheidungsmöglichkeit zwischen einer *top-down* und einer *bottom-up* Vorgehensweise. Im ersten Fall bewegt man sich vom Allgemeinen zum Speziellen, vom Abstrakten zum Konkreten oder vom Übergeordneten zum Untergeordneten, im zweiten Fall umgekehrt.

Alle diese Prinzipien sollten jedoch nie zum Selbstzweck entarten, sondern stets Zielen wie Les-, Bedien- oder Wartbarkeit untergeordnet sein.

4.3 Parallelization

4.3.1 Parallel Programming Paradigms

The important language paradigms

- *imperative*: FORTRAN, C
- *logical/relational*: PROLOG
- *object-oriented*: SMALLTALK
- *functional/applicative*: LISP

and the respective programming languages are all, in principle, well-suited for both explicit (manual) and implicit (automatic) parallelization. In our context of scientific computing, the latter one is still mostly realized via special compilers (High Performance FORTRAN, e. g.). Another approach will be presented later in this section. In the case of an explicit parallelization, usually, appropriate libraries which contain modules for basic tasks like matrix-vector multiplication are linked to standard languages (FORTRAN 90, HPF, C, C++), thus allowing for their use (see Sect. 4.5 for some examples) in a standard environment. Sometimes, the fully explicit programming of a parallel algorithm may be necessary, too. The development of special languages like OCCAM has never had an important impact on numerical simulations. Concerning the above-listed paradigms, the imperative concept (FORTRAN, C) is still the dominant male, with a recently increasing touch of object-oriented features (C++). The old rule of thumb 'take a FORTRAN code, push it through a vectorizing compiler, and let the CRAY work' is still widespread (today with a broader spectrum of machines, perhaps)!

Um unsere Gebote aus der Einleitung nicht aus den Augen zu verlieren, soll hier noch kurz ein neuerer und unkonventioneller Ansatz zur Unterstützung automatischer Parallelisierung vorgestellt werden, die funktionale Sprache FASAN (**F**unctions **A**pplied to **S**ubstructured Data in **A**lgorithms of **N**umerical Computing; für Literatur zu FASAN siehe beispielsweise die URL `http://www5.in.tum.de/persons/ebner.html`). FASAN ist eine *funktionale* Sprache, die sequentielle Programmteile aus Standardsprachen parallel auf Workstationnetze verteilt. Sie ist vor allem für rekursive numerische Algorithmen auf baumartigen Datenstrukturen geeignet (einem klassischen Problemfall der Parallelisierung), da sie die automatische Ausnutzung maximaler Parallelität ermöglicht. Die Stromsemantik der Sprache unterstützt auch pipeline-artige Parallelisierung. Durch sog. Kabelbäume werden unnötige Synchronisationen und Kommunikationen aufgelöst.

Das in Abschnitt 4.2 diskutierte funktionale Prinzip soll also ausgenutzt werden, die bei imperativen Sprachen erforderliche explizite Zerlegung eines Algorithmus' in möglichst unabhängige Teile (mit allen damit verbundenen Problemen wie versteckte Abhängigkeiten, Seiteneffekte, globaler Speicherzustand) zu umgehen. Nur die Datenabhängigkeit soll beschrieben werden, was eine automatische (und somit nicht auf die jeweilige Zielarchitketur hin zu tunende) Parallelisierung ermöglicht.

Dass eine solche explizite Zerlegung in parallel ausführbare Teilprobleme bei numerischen Aufgabenstellungen durchaus problematisch sein kann, zeigt das folgende einfache Beispiel. Wir betrachten die zweidimensionale Laplace-Gleichung $\Delta u = 0$ (Dirichlet-Randbedingungen), die auf einem äquidistanten Gitter mittels finiter Differenzen (Standard 5-Punkte-Stern) diskretisiert sei. Zur Lösung des zugehörigen linearen Gleichungssystems verwenden wir die Gauß-Seidel-Iteration mit lexikographischer (d. h. zeilen- oder spaltenweiser) Abarbeitungsreihenfolge (vgl. Abb. 4.17):

$$u_{i,j}^{\text{neu}} := \frac{1}{4} \cdot \left(u_{i-1,j}^{\text{neu}} + u_{i,j-1}^{\text{neu}} + u_{i+1,j}^{\text{alt}} + u_{i,j+1}^{\text{alt}} \right) .$$

Der Zeilenindex i laufe dabei von oben nach unten, der Spaltenindex j von links nach rechts. An

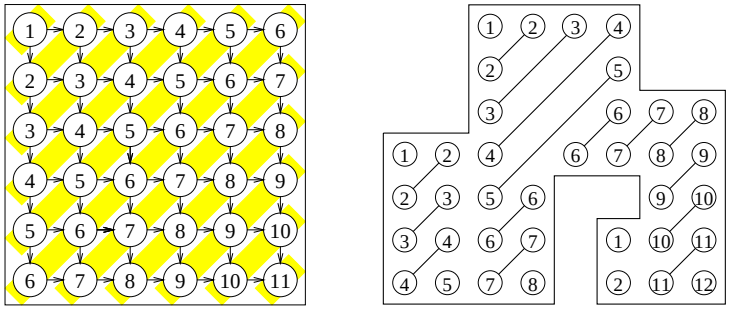


Abbildung 4.17: Links: Frühestmögliche Neuberechnung bei lexikographischer Abarbeitungsreihenfolge bei der Gauß-Seidel-Iteration. Die Pfeile beschreiben den Datenfluß zwischen den Gitterpunkten. Rechts: Ein Gebiet mit komplexer Geometrie. Punkte auf den Diagonalen können gleichzeitig bearbeitet werden.

den jeweils kleineren Indexpositionen $(i-1, j)$ und $(i, j-1)$ werden bereits die aktualisierten Näherungswerte benutzt, in $(i+1, j)$ und $(i, j+1)$ noch die alten (Randwerte liegen immer vor). Die Abhängigkeiten sowie das Parallelisierungspotential liegen damit also bereits fest. In einer imperativen Programmiersprache würde man nun die Iteration durch zwei verschachtelte Schleifen über die Zeilen und über die Spalten beschreiben. Aber weder die Zeilen noch die Spalten sind voneinander unabhängig, so dass sie sequentiell berechnet werden müssten. Die explizite Beschreibung eines Algorithmus', der die voneinander unabhängigen Gitterpunkte in den Diagonalen parallel aktualisiert, ist dagegen schon merklich aufwendiger – insbesondere für kompliziertere Geometrien wie im linken Teil von Abb. 4.17. Dies ist ein Grund dafür, warum bei der Parallelisierung des Gauß-Seidel-Verfahrens oft andere Abarbeitungsreihenfolgen wie Red-Black (Schachbrettmuster: alle „roten" Punkte können unabhängig voneinander und somit parallel aktualisiert werden, ebenso – anschließend – alle „schwarzen") bevorzugt werden, sofern man nicht gleich zum trivial zu parallelisierenden Jacobi-Verfahren übergeht. Dieses einfache Beispiel ist übrigens kein Einzelfall: Im wissenschaftlichen Rechnen ist die effiziente Parallelisierung oft eine Herausforderung, die guten sequentiellen Algorithmen leisten gegen ihre Parallelisierung oft hartnäckiger Widerstand als die schlechten.

FASAN wurde entworfen, um diesen Mangel zu beheben und – noch weiter gehend – allgemein hierarchische numerische Algorithmen (z. B. auf Bäumen) auf einem Workstationcluster automatisch parallelisieren zu können. Natürlich geht die Umsetzung der funktionalen Ebene zu Lasten der Effizienz, es gibt aber eine Reihe von Strategien zur Beschränkung dieses Mehraufwands. Beispielsweise wird möglichst viel an bereits bestehendem (imperativen) Code (optimierte Routinen etc.) wiederverwendet – die funktionale Ebene von FASAN dient lediglich der parallelen Komposition der sequentiellen Module.

Die Grundbausteine von FASAN sind Agenten, also funktionale Einheiten mit einer festen Zahl von Ein- und Ausgängen. Sie werden durch Ströme zu einem Netz verbunden. Diese Ströme übertragen unidirektional Sequenzen von Daten. Ein FASAN-Programm ist somit eine Menge stromverarbeitender Funktionsdeklarationen. Die Abarbeitung eines FASAN-Programms kann man sich allgemein als einen sukzessiven Entfaltungs- oder Aufklappungsprozess vorstellen: Eine Funktion mit bereitliegenden Eingangsdaten wird durch ihren Datenflussgraphen ersetzt. Wie das Beispielprogramm in Abb. 4.18 zeigt, erscheinen die Ströme dem Programmierer als normale Ein- bzw. Ausgabeparameter. Die Stromsemantik ist aber auf der Implementierungsebene wichtig. Ein bereits erzeugtes Agentennetz kann immer wieder für verschiedene Daten verwendet werden, im Falle des Gleichungssystemlösers etwa für verschiedene rechte Seiten oder bei iterativen Verfahren für sich ändernde Näherungswerte. So können Kosten für die Erzeugung des Netzes gespart werden. Es klappt sogar bei Rekursion: Wenn beispielsweise bereits einmal ein rekursives, komplexes Netz namens iterate entfaltet worden ist, so sorgt

NewResult = **recycle** iterate (ResultFromLastIteration)

dafür, dass das bestehende Netz mit den neuen Startdaten gefüttert wird. Fehlt das Schlüsselwort **recycle**, wird das Netz dagegen vollständig neu aufgebaut (vorteilhaft etwa bei gewünschter Neuplatzierung der parallelen Agenten).

FASAN unterscheidet zwei Arten von Agenten: Elementaragenten und Netzagenten. Erstere werden nicht weiter untergliedert, sie stützen sich auf sequentielle Funktionen, bspw. aus einer externen Programmbibliothek (in beliebiger Sprache). Über die Größe der eingebundenen Module lässt sich die Granularität der Parallelisierung steuern. Netzagenten hingegen repräsentieren ihrerseits ein Netz von Agenten und stellen die Verbindung der Ein- und Ausgangsströme von Elementaragenten her. Diese können somit hinter- und nebeneinander geschaltet werden, wobei auch Rekursionen erlaubt sind. Das Hauptprogramm ist als ein initiales Agentennetz vorgegeben, das zur Laufzeit durch schrittweise Ersetzung der Netzagenten entfaltet wird.

Eine Besonderheit von FASAN stellen die sog. *Kabelbäume* dar. Sie eliminieren unnötige Kommunikationen und Synchronisationspunkte, die durch eine komfortablere textuelle Beschreibung zustande gekommen sind. Wesentliche Idee ist dabei, dass Ströme direkt vom Sender zu den jeweiligen Empfängern gezogen werden, auch wenn es im Programmtext so erscheint, als liefen sie über verschiedene Zwischenagenten. Um die maximale Parallelität gewährleisten zu können, werden Netzagenten bereits dann entfaltet, wenn einer der Eingangsströme das erste Datum liefert. Ein Elementaragent ist allerdings erst dann rechenbereit, wenn er auf allen Eingangsströmen Daten erhält.

Durch Begrenzung der Auffaltungstiefe des Netzes kann die Granularität gesteuert werden. Ab dieser Tiefe wird die weitere rekursive Entfaltung des Agentennetzes durch einfache sequentielle Auswertung ersetzt. Die Lastverteilung, über die wir in diesem Kapitel noch eingehender sprechen werden, geschieht über vom Benutzer definierte Funktionen. So bewirkt z. B.

x := f (y) **on** next_proc,

dass der Elementar- oder Netzagent f auf dem Rechner gestartet wird, den next_proc zur Laufzeit ermittelt.

Der FASAN-Compiler übersetzt FASAN-Programme nach C. Diese werden mit einem Laufzeitsystem auf Basis von PVM zusammengebunden, das für die automatische Migration und Kommunikation der Agenten sorgt.

Anhand des (bereits eingangs gerühmten) Divide & Conquer Schemas schauen wir uns FASAN etwas genauer an. Hierbei handelt es sich um eine Methode, die ein gegebenes Problem durch Aufspaltung in mehrere gleichartige, aber kleinere und damit leichter zugängliche Teilprobleme zu lösen versucht. Beispiele sind etwa Gebietszerlegungsverfahren für partielle Differentialgleichungen oder die rekursive Substrukturierung bei der Methode der finiten Elemente. Als Aufrufgraph hat man es hier mit einem Baum zu tun, bspw. einem Binärbaum, dessen Knoten etwa die jeweiligen Teilgebiete der Berechnung repräsentieren. Zu den einzelnen Teilproblemen gehören bei numerischen Anwendungen typischerweise große lokale Datenmengen (z. B. Matrizen bei FE-Berechnungen), die der zugehörige Prozessor in der Iteration lokal verwaltet und modifiziert. Dieses Management großer lokaler Datenmengen und die daraus resultierende Kommunikationsproblematik ist die eigentliche Besonderheit numerischer Anwendungen, die von den gängigen (parallelen) funktionalen Sprachkonzepten nicht abgedeckt wird und deshalb spezielle Entwicklungen wie z. B. FASAN erfordert. In Abbildung 4.18 ist ein allgemeiner Divide & Conquer Algorithmus in FASAN angegeben. Daneben steht das zugehörige Agentennetz, das den Datenfluss veranschaulichen soll.

Zunächst wird geprüft, ob man bereits an einem Blatt des Aufrufbaumes angekommen ist, und ggfs. wird die sequentielle Elementarfunktion work angestoßen. Ansonsten wird mittels Dereferenzierung (Selektoren) der Eingangsstrom bzw. -kabelbaum entflochten, der Agent sieht jetzt die nächste Ebene: zwei Ströme left und right sowie lokale Daten ld. Nun findet zunächst ein Preprocessing (pre) auf den eigenen, lokalen Daten (ld) statt. Informationen vom Vater (parentdata), z. B. Werte auf den

```
agent divconq(datatree,parentdata): newtree,result;
begin
  if isLeaf(datatree) then
    ld := val(datatree);
    newtree, result := work(ld,parentdata);
  else
    <left,ld,right> := datatree;
    ld1,ld2,nldtemp := pre(ld,parentdata);
    nleft, nld1 := divconq(left,ld1);
    nright,nld2 := divconq(right,ld2) on nextproc;
    nld,result := post(nld1,nldtemp,nld2);
    newtree := <nleft,nld,nright>
  fi
end

agent iterate(datatree, eps): newtree,result;
begin
  nld,r := divconq(datatree, nodata());
  if error2big(r,eps) then
    newtree,result := recycle iterate(nld,eps);
  else
    result := postprocess(r);
    newtree := nil;
  fi
end
```

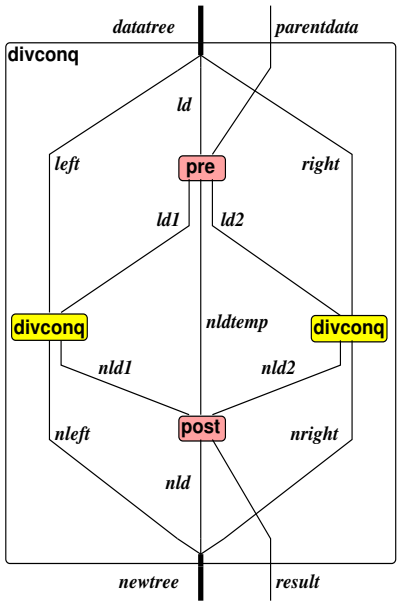


Abbildung 4.18: Links: FASAN-Programm eines abstrakten Divide & Conquer-Algorithmus. Rechts: Datenflussgraph des **else**-Zweiges des Agenten divconq.

Rändern des Teilgebiets, fließen hier mit ein. Außerdem werden Daten für die Söhne bereitgestellt. Danach erfolgt die Aufspaltung in zwei Teilprobleme, die parallel in den Sohnknoten berechnet werden (divconq). Die Ergebnisse der Söhne werden dann in einem Postprocessing (post) in die lokalen Daten eingearbeitet. Mittels Referenzierung (Konstruktor <>) wird der Ausgangsstrom bzw. -kabelbaum aus den drei Komponenten nleft, nright und nld gebildet.

Nun passen aber lokale Daten, die iterativ verändert werden, nicht in das oben entworfene Bild funktionaler Sprachen, stellen sie doch quasi einen globalen Zustand dar. FASAN soll aber seiteneffektfrei sein und Variablen nur einmal zuweisen. In FASAN kann dies trotz rein funktionaler Beschreibung effizient umgesetzt werden. Der funktionalen Idee folgend, müssten die alten lokalen Daten eines Knotens aus der vergangenen Iteration als Eingabestrom in den Agenten fließen und die modifizierten Daten als Ausgabestrom erscheinen. Konsequenterweise müssten dann für jeden Knoten des Aufrufbaumes die lokalen Daten bis zur Wurzel gereicht und von dort wieder beim Absteigen auf die Knoten verteilt werden. Es entstünde so ein Baum von lokalen Daten. Dieser würde aber bei großen Anwendungen die Kapazitäten der oberen Knoten und des Verbindungsnetzwerkes sprengen. Hier helfen die bereits erwähnten Kabelbäume, indem sie dafür sorgen, dass die lokalen Daten am Platz bleiben und keine überflüssigen Informationen über das Netz gehen. Kabelbäume sind Ströme, die selbst wieder Ströme (und Kabelbäume) enthalten können. Damit

Abbildung 4.19: Kabelbaum

lassen sich Teilströme syntaktisch zu Listen und Bäumen zusammenfassen, die in einem einzigen Strom gebündelt sind. In unserem Beispiel werden die lokalen Daten als Teilströme gekapselt, bis zur Wurzel geleitet und beim Abstieg wieder in den einzelnen Agenten entflochten. Die jeweiligen

Knoten haben dabei nur eine lokale Sicht. Wenn sie die äußere Hülle des Kabelbaums (datatree) entfernen, können sie lediglich ihren eigenen Datenstrom (ld) lesen. Die beiden Teilkabelbäume (left, right) sind dagegen gekapselt, ihre Teilströme für diesen Agenten nicht sichtbar. Wie bei ihren Vorbildern aus der Technik macht es aber für den Fluss durch die einzelnen Teilkabel keinen Unterschied, wie diese verlegt oder zusammengefasst werden. Der Strom fließt direkt vom Erzeuger zum Verbraucher.

Auch in FASAN fließen die Daten direkt von Senderagenten zu den Empfängeragenten. So bleiben die lokalen Daten stets am gleichen Ort, wenn Sender und Empfänger im gleichen Knoten sitzen. Genau dies garantiert die **recycle**-Anweisung. Trotz rein funktionaler Syntax ist es so gelungen, lokale Daten effizient zu implementieren. Die direkten Verbindungsströme, die durch die Kabelbaumentflechtung entstehen, sorgen überdies auch für optimale Kommunikationswege in einem komplexen Netz paralleler Tasks. Unnötige Zwischenstationen oder Synchronisationspunkte, die nur wegen einer übersichtlicheren textuellen Beschreibung entstanden sind, können so aufgelöst werden.

Man sieht: Auch wenn sich im sequentiellen Bereich „neumodische“ informatische Sprachparadigmen im wissenschaftlichen Rechnen nicht durchgesetzt haben, werden für die Parallelisierung die Karten neu gemischt. Insbesondere bei einer größeren Anzahl von Prozessoren (Netze von Workstations, massiv parallele Rechner) kann es dabei sein, dass die einfachere und bessere Parallelisierung das ansonsten immer dominierende (Laufzeit-) Effizienz-Argument aussticht.

Dies soll an dieser Stelle genügen, und wir wollen uns jetzt der herkömmlichen Vorgehensweise im wissenschaftlichen Rechnen, der expliziten Parallelisierung mittels MPI zuwenden.

4.3.2 Programming Models: MPI and PVM

Finally the crucial question: How to write parallel programs? For users of UMA machines, the answer is quite simple: Just program as usual in a UNIX environment, since there is no need for changing the *programming model*. However, the situation with distributed memory is quite different. Especially for applications of scientific computing, the *MPI* (*Message Passing Interface*) model has become some kind of a standard (cf. `http://www.cs.usfca.edu/mpi/index.html`). During its development, the big objective in mind was the straightforward transfer of parallel codes to loosely coupled and heterogeneous systems with a varying number of processors like networks of workstations (NOWs) or PC clusters. The first standard MPI-1 was established between 1992 and 1994 after long discussions among parallelization experts, users, and computer manufacturers. Hence, today, nearly all of the latter support this standard: On massively parallel systems, a tuned MPI implementation comes with the machine's installation, and for a standard UNIX environment (i. e. single processor workstations forming a cluster), several implementations of MPI are available, among which *PVM* (*Parallel Virtual Machine*, see `www.epm.ornl.gov/pvm/pvm_home.html`) is perhaps the best known. Other examples are *LAM* (*Local Area Multicomputer*, see `www.mpi.nd.edu/lam`) and *MPICH* (*MPI CHameleon*, see `www.mcs.anl.gov/mpi/mpich`).

Of course, heterogeneous distributed-memory architectures show a bigger need for manual parallelization or explicit communication, respectively, than uniform shared-memory machines do: the more distributed, the more manual! Hence, MPI is based upon the explicit exchange of messages between arbitrary nodes. On the one hand, this explicit exchange leads to a higher amount of programming work (esp., if complicated data or program structures like trees, graphs, or recursion are involved). On the other hand, the progammer's possibilities to tune and optimize the parallelization of some given code increase, too.

We want to mention some important features of MPI:

- A (parallel) program consists of n processes with completely separate address space without any possibility of a remote access.

- Communication and data exchange happen via messages only (system calls `send` and `receive`), which are exchanged between participating processors.
- The kernel of MPI is a library of communication routines which allow to integrate MPI commands into standard languages (FORTRAN: `mpif.h`; C: `mpi.h`). These MPI libraries provide routines for the initialization (`init`), for getting information about the number of processes (`comm_size`) or the own process number (`comm_rank`), for communication (see below), for termination (`finalize, abort`), for error handling, and much more.
- A message consists of a header (recipient, buffer, type, context of communication) and of its body (the content). The type is important for heterogeneous architctures, for example, in order to enable the communication system to correctly convert the formats. A context of communication defines the range within which communication is possible or allowed, respectively.
- Messages are buffered (send buffer, receive buffer).
- Sending a message can be *blocking* (finished only after the message has left the node) or *non-blocking* (finished immediately; the buffer may be cleared (and the message sent) later).
- The same holds for receiving a message (blocking: waiting; non-blocking: looking for it from time to time).

Since communication is often quite costly in an MPI application, the efficiency of the implemented exchange of messages is very important for the evaluation of a parallel system. The cost of passing some message of length N via some buffer of capacity K is usually modelled as

$$t(N) \;:=\; \alpha \cdot \frac{N}{K} \;+\; \beta \cdot N\,,$$

where α denotes the cost of initializing a send process via the buffer (latency), and where β denotes the cost of transportation per data unit (direct communication from sender to recipient assumed). A longer path of communication may further increase the cost of transportation.

A simple example:

```
P1:  compute something              P2:  compute something
     store result in SBUF                 store result in SBUF
     SendBlocking(P2,SBUF)                SendBlocking(P1,SBUF)
     ReceiveBlockin1(P2,RBUF)             ReceiveBlocking(P1,RBUF)
     read data in RBUF                    read data in RBUF
     compute again                        compute again
```

Obviously, without buffering, deadlocks where both are waiting for an answer to complete their send job) are possible. The actual buffering mechanism can be selected by the programmer:

- nothing specified: buffering is possible, but not imperative (no influence of the programmer; typically default)
- never: no buffering (efficient, but possibly risky)
- always: secure, but sometimes costly

Without buffering, measures to avoid deadlocks have to be taken (atomic send&receice calls, for example).

Further features:

- collective communication:
  - *broadcast* (one-to-all) and *multicast* (one-to-some)

- *gather* (all-to-one)
- *gather-to-all* (collect from all, send result to all)
- *scatter* (subdivide and distribute among all)
- *all-to-all* (first gather, then scatter)
- *reduce* (collect from all, one global operation (e. g. sum or minimum), and result sent to one)

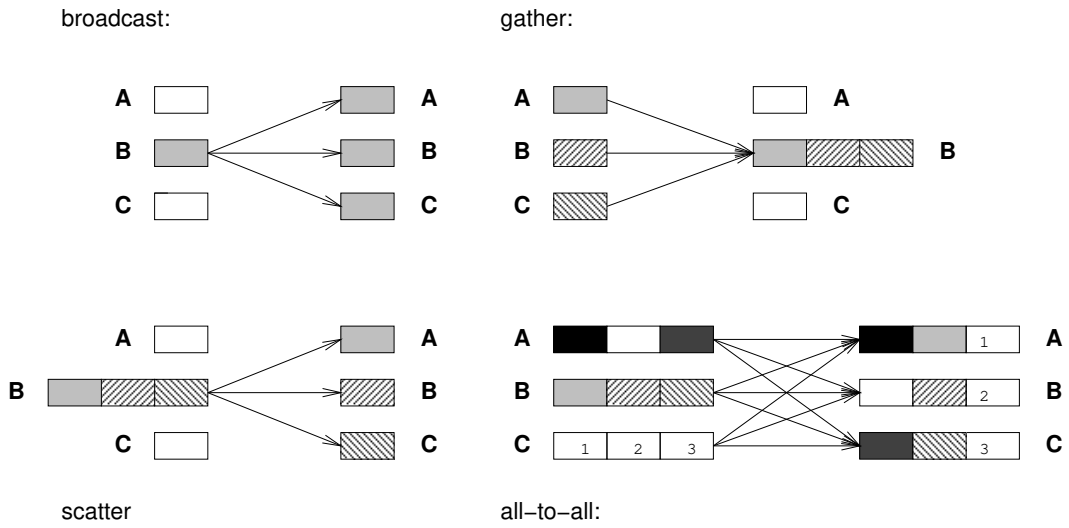


Abbildung 4.20: Possibilities of collective communication within MPI: broadcast (top left), gather (top right), gather-to-all (bottom left) and all-to-all (bottom right)

- synchronization: barriers as points of synchronization
- different process models (static/dynamic, heavy-/lightweight)
- start and end of a parallel application: prologue, epilogue
- some truisms (nevertheless important!):
  - avoid large numbers of short messages (latency!)
  - computation time should dominate communication time
  - data parallelism: try to obtain an optimal size of data subsets
  - reduce number of communication points

Finally some remarks on PVM. The first version was released in 1991 (for further informations, see `http://www.netlib.org/pvm3/book/pvm-book.html`). Concerning its objective:

PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computations.

In contrast to the *standard* MPI, PVI, thus, is a concrete system that has been developed at the Oak Ridge National Laboratory and at the University of Tennessee. Much of the experience obtained during using PVM and other comparable systems like P4 at the ETH Zürich had an important influence on MPI. Today, PVM can be regarded as one implementation of the message passing concept. Some manufacturers offer an own optimized PVM for their architecture.

4.3.3 Lastverteilung

Allgemeines

In verteilten Umgebungen (Netze, lose gekoppelte Systeme) kann es zu einer ungleichen Lastverteilung kommen: Einzelne Prozessoren arbeiten noch, während andere bereits ruhen. Dies beeinträchtigt den Effekt des Einsatzes mehrerer Prozessoren negativ und soll so weit wie möglich verhindert werden. Diesem Zweck dient die *Lastverteilung* bzw. der *Lastausgleich* (engl. *load distribution*, *load balancing* oder *scheduling*). Wichtig ist dabei, dass die eingesetzten Mechanismen ohne signifikanten Overhead ablaufen können: Schließlich geht auch der zur Lastverteilung erforderliche Rechenaufwand zu Lasten des Systems und damit zu Lasten der parallelen Effizienz. Man unterscheidet

- *global* scheduling (wo laufen welche Prozesse) und *local* scheduling (wann bearbeitet ein Prozessor welchen Prozess);
- *statische* Lastverteilung (a priori) und *dynamische* Lastverteilung (während der Laufzeit).

In den meisten Fällen reicht ein statisches Vorgehen nicht aus, da es in heterogenen Netzen zu unvorhersagbaren Prozessen kommt oder da – insb. in Anwendungen des wissenschaftlichen Rechnens – die Last des Programms selbst nicht vorhersagbar ist (etwa bei adaptiver Gitterverfeinerung).

Vor der Entwicklung einer Lastverteilungsstrategie sind einige Fragen zu klären:

- die verfolgten Ziele betreffend:
  - Soll die Systemlast optimiert oder ein bestimmtes Anwednungsprogramm möglichst schnell beendet werden?
  - Sollen nur neue Prozesse platziert werden, oder sollen Prozesse bei Bedarf auch verlagert werden (*Migration*)?
- die Integrationsebene betreffend:
  - Wer soll die erforderlichen Aktionen (Lasterfassung, Strategie anwenden, Maßnahmen einleiten) durchführen: das Anwendungsprogramm, das Laufzeitsystem oder das Betriebssystem?
- die Struktur der jeweiligen Anwendung betreffend:
  - Gibt es Einschränkungen bei der Zuordnung von Prozessen auf Prozessoren (oft bei Simulationsaufgaben der Fall)?
- die zu verteilenden Einheiten betreffend:
  - Sollen Prozesse (grobkörnig), Threads (feinkörnig), Objekte oder Daten (Simulationsanwendungen) verteilt werden?

Im Laufe der Zeit ist eine Vielzahl von Lastverteilungsstrategien entwickelt worden. Diese lassen sich klassifizieren

- hinsichtlich ihrer Herkunft:
  - Ideen aus der Physik (Diffusionsmodell), aus der Kombinatorik (Lastverteilung als diskretes Problem, Algorithmen der Graphentheorie), aus der Ökonomie (Bidding, Maklersysteme)
  - Für welche Zieltopologie sind sie (ursprünglich) bestimmt (Netze, Bustopologien)?
  - Mit welchen Arten von Datentopologien hat man es in den Anwendungen zu tun (Gitter, Bäume, Mengen)?

- hinsichtlich der Verteilungsmechanismen:
  - Wird Last nur an Nachbarknoten abgegeben oder über beliebige Distanzen?
  - Werden nur neue Prozesse verteilt oder gibt es Migration?
- hinsichtlich des Informationsflusses:
  - Wem teilt ein Knoten seine Lastdaten mit, von wem erhält er Informationen?
- hinsichtlich der Koordinierung:
  - Wird zentral oder dezentral über anzustoßende Aktionen entschieden?
  - Wie kommen Entscheidungen zustande (autonom: jeder Knoten allein / kooperativ: mittels Absprachen / kompetitiv: über Konkurrenz)?
  - Wer nimmt ggfs. an Absprachen teil (alle Knoten oder nur die jeweiligen Nachbarn)?
- hinsichtlich der Algorithmen:
  - Ist der Entscheidungsprozess statisch oder dynamisch?
  - Wer stößt ggfs. Lastausgleichsmaßnahmen an (der untätige Knoten, der überlastete Knoten, die Zentrale, eine Uhr)?
  - Adaptivität: Liegt die Strategie unwiderruflich fest, ist sie an die aktuelle Situation anpassbar oder gar lernfähig?
  - Spielen Kostenüberlegungen eine Rolle?
  - Gibt es Sicherheitsmechanismen gegen Auswüchse (Lastverteilung dominiert Rechenzeit und zwingt die Effizienz in die Knie)?

Einige Beispiele für Lastverteilungsstrategien:

- *Diffusionsmodell*: Analog zu Diffusionsprozessen in der Physik, wie man sie bspw. bei der allmählichen Verteilung einer anfangs stark ungleichmäßigen Konzentration eines gelösten Stoffes in einem ruhenden Fluid beobachten kann, findet ein permanenter Lastausgleichsprozess zwischen Nachbarprozessoren statt. Dabei wird ein bestimmter Prozentsatz einer eventuellen Überlast an unterbeschäftigte Nachbarprozessoren abgegeben.
- *Bidding*: Hier regeln Angebot und Nachfrage die Last. Der unterbeschäftigte Prozessor sucht nach Arbeit, derjenige mit Überlast versucht, Arbeit loszuwerden. Somit regelt ein Markt für Aufträge bzw. für Rechenkapazität die Auftragsvergabe.
- *Maklersystem*: Broker-basiert, insb. für heterogene und hierarchische Rechner-Topologien (Bäume), skalierbar. Jeder Prozessor hat einen Makler (Broker, realisiert als kooperierender Agent) mit Teilbaumwissen. Tasks gelangen über einen Application Server zum lokalen Broker und werden – je nach Budget – lokal im Teilbaum bearbeitet oder an den Vaterprozessor hochgereicht (evtl. rekursiv). Auf einer bestimmten Ebene (spätestens im Wurzelknoten) wird dann entschieden und zugewiesen. Dabei werden Preise sowohl für die Ressourcennutzung als auch für das Makeln erhoben.
- *Balanced Allocation*: Hier erfolgt die Auftragsvergabe an einen zufällig ausgewählten Prozessor. Zur Verbesserung der Qualität werden r Prozessoren zufällig ermittelt, den Zuschlag erhält derjenige mit der geringsten Last.
- *Random Matching*: Im Topologiegraphen des Rechners wird ein Matching konstruiert (Teilmenge der Kantenmenge, so dass jeder Knoten maximal einmal vorkommt). In diesem Matching wird dann die Last paarweise längs jeder Kante ausgeglichen.

- *Vorausberechnung* der Last: Alle bisher genannten Strategien beruhen auf *lokaler* Information, und Lasttransfers werden lokal angestoßen. Dies kann zu unökonomischen Lastverschiebungen führen, so dass es manchmal besser sein kann, a priori bzw. zu bestimmten Zeitpunkten die optimale *globale* Lastverteilung zu berechnen (z. B. im Graphen) und dann die Last möglichst kommunikationsextensiv zu verschieben.

Zur Erfassung bzw. Schätzung der Last benötigt man *Lastmodelle*. Diese beruhen fast alle auf *Lastindizes*, quantitativen Maßen für die Last von Anbietern von Rechenkapazität. Es gibt einfache und zusammengesetzte Lastindizes (eine bzw. mehrere Kenngrößen), diese können verschiedene Funktionseinheiten betreffen (CPU, Bus, Speicher). Ferner unterscheidet man Snapshot-Größen und gemittelte bzw. integrierte Größen. Die Lastindizes fließen dann (gewichtet) in eine Formel für die Last ein. Die Gewichte können dabei a priori festgelegt sein oder sich dynamisch ändern. Oft sind auch stochastische Größen zur Erfassung äußerer Einflüsse integriert. Ein guter Index sollte die Zielgröße präzise sowohl in der Gegenwart widerspiegeln als auch für die Zukunft vorhersagen, glättend wirken (um Peaks auszugleichen) sowie einfach und billig zu berechnen sein. Als Beispiel sei der UNIX Load Average genannt (xload), der die gemittelte Anzahl von Prozessen in der CPU-Warteschlange angibt.

Neben Lastmodellen und -verteilungsstrategien sind schließlich *Migrationsprinzipien* festzulegen:

- Was migriert (Prozesse, Threads, Daten)?
- Wie groß sind die wandernden Einheiten?
- Werden Migrationen sofort oder verzögert ausgeführt?
- Erfolgt die Übergabe auf einmal oder als *lazy copying*?
- Wohin kann migriert werden (zu Nachbarknoten oder überall hin)?
- Ist eine Migration nur zwischen Knoten desselben Typs möglich oder kann auch in heterogenen Netzen beliebig migriert werden?

Lastverteilung mittels raumfüllender Kurven

Zum Abschluss unserer Betrachtungen zum Lastausgleich wollen wir noch kurz eine spezielle, sehr einfache und dennoch äußerst wirkungsvolle Lastausgleichsstrategie studieren, die ihre Wurzeln in der Analysis hat. Auch dies ist ein Beispiel dafür, wie Methoden der (in diesem Fall reinen) Mathematik, denen zunächst ein Potential für das wissenschaftliche Rechnen gar nicht anzusehen ist, dennoch hier gewinnbringend eingesetzt werden können. Es geht um *raumfüllende Kurven*, der Einfachheit halber nur im Zweidimensionalen. Versteht man unter einer Kurve das Bild einer stetigen Abbildung vom Einheitsintervall $[0,1]$ in das Einheitsquadrat $[0,1]^2$, dann heißt eine solche Kurve genau dann *raumfüllend*, wenn sie das gesamte Einheitsquadrat überstreicht (und somit ein zweidimensionales Volumen hat):

$$f:\ [0,1] =: I \quad \mapsto \quad Q := [0,1]^2, \qquad f \ \text{ stetig und surjektiv}.$$

Bei der Konstruktion des prominentesten Vertreters raumfüllender Kurven, der *Hilbert-Kurve*, stand die geometrische Anschauung Pate: Wenn I raumfüllend auf Q abgebildet werden kann, dann auch jedes der vier kongruenten Teilintervalle von I auf einen der Quadranten von Q. Wendet man diesen Unterteilungs- und Zuordnungsprozess unter Wahrung von Nachbarschafts- und Teilmengenrelationen rekursiv an, so erhält man im Grenzfall die Hilbert-Kurve. Die Zuordnung von Urbild- zu Bildpunkten ergibt sich dabei aus der Entsprechung von Intervallschachtelungen in I und Quadratschachtelungen in Q. In der Praxis ist man jedoch weniger an der Hilbert-Kurve selbst, sondern vielmehr an den einzelnen Iterierten im Erzeugungsprozeß interessiert. Man startet mit einem *Generator* oder *Leitmotiv*, das die Besuchsreihenfolge der Teilquadrate angibt. Durch

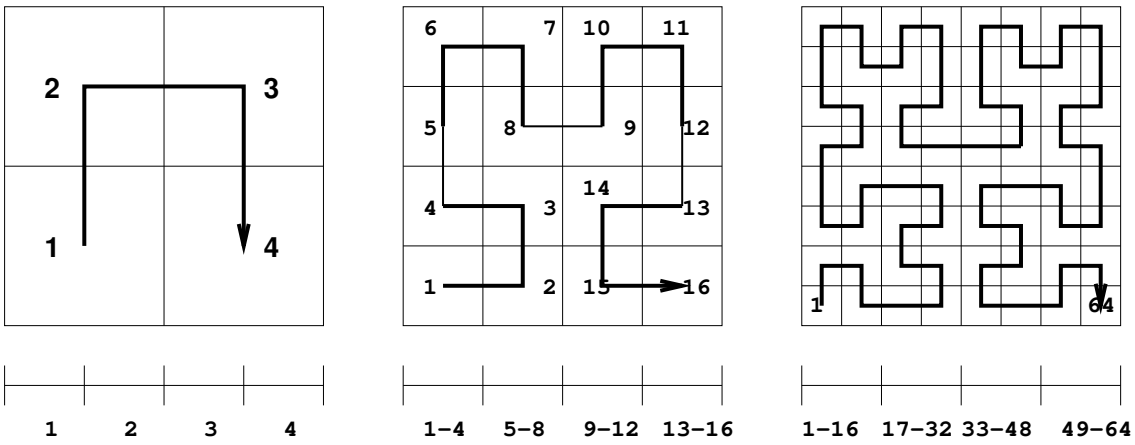


Abbildung 4.21: Erzeugung der Hilbert-Kurve aus ihrem Leitmotiv

Ähnlichkeitstransformationen wird der Generator dann in jedem Teilquadrat angewendet, und die offenen Enden werden verbunden (siehe Abb. 4.21).

Nun zeigt Abb. 4.21 nicht die einzige Möglichkeit, aus dem Hilbert-Generator unter Erhaltung von Inklusion und Nachbarschaft eine raumfüllende Kurve zu konstruieren. Von Moore stammt eine zweite Variante, die in Abb. 4.22 angegeben ist. Das war's dann aber (modulo Symmetrie),

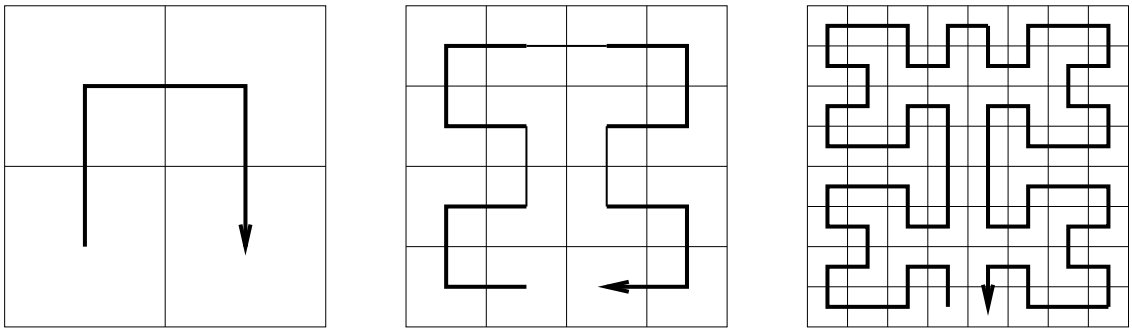


Abbildung 4.22: Erzeugung der Moore-Kurve aus dem Hilbert-Leitmotiv

mehr als diese beiden Varianten gibt es nicht.

Die resultierenden Abbildungen sind in der Tat stetig und surjektiv. Obwohl jede Iterierte injektiv ist, ist die Hilbert-Kurve selbst nicht injektiv, wie man leicht etwa an den inneren Eckpunkten der Quadrate sieht: Die Quadratschachtelungen in Teilquadrat 1 und 3 definieren denselben Bildpunkt, die zugehörigen Intervallschachtelungen im Urbildraum führen aber auf verschiedene Urbildpunkte (siehe Abb. 4.23).

Die Hilbert-Kurve ist zwar die berühmteste, jedoch nicht die älteste raumfüllende Kurve. Die Diskussion über die ja gar nicht so selbstverständliche Existenz solcher „topologischen Monster", wie sie zunächst genannt wurden, kam in Gang, als *Cantor* 1878 zeigte, dass es zwischen zwei beliebig-, aber endlich-dimensionalen glatten Mannigfaltigkeiten eine Bijektion gibt. Ein Jahr später bewies *Netto*, dass eine solche Bijektion bei ungleicher Dimensionalität von Urbild- und Bildraum notwendigerweise unstetig sein muß, also keine Kurve in unserem Sinne definieren kann. Gibt man die Injektivität auf, so klappt's aber, was 1890 von *Peano* durch Angabe der ersten raumfüllenden Kurve konstruktiv gezeigt wurde. In der Folge sprudelten die Monster dann nur so: Ein Jahr nach Peano brachte Hilbert seine Kurve auf den Markt, und weitere Exemplare wurden von Moore,

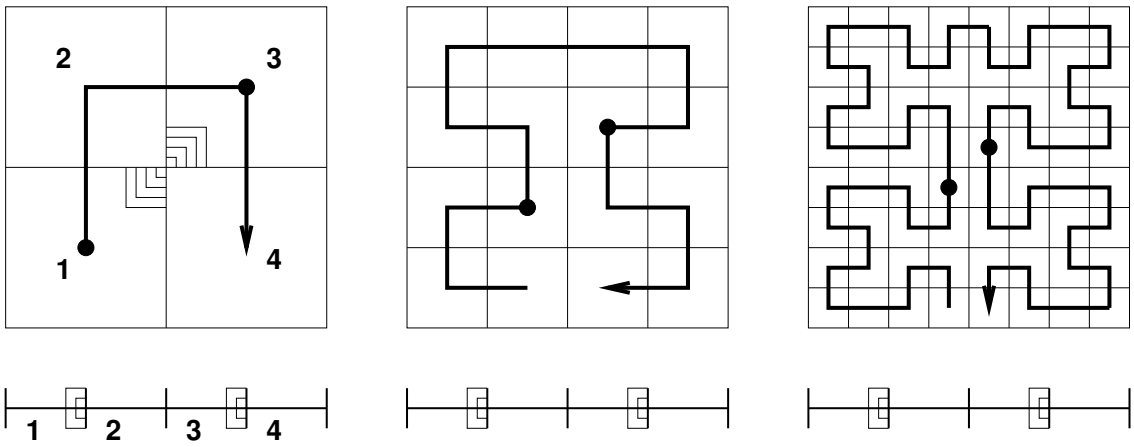


Abbildung 4.23: Fehlende Injektivität am Beispiel der Moore-Kurve

Lebesgue sowie Sierpiński angegeben, um nur einige zu nennen.

Beim Urahn der raumfüllenden Kurven, der Peano-Kurve, werden I und Q in neun kongruente Teilbereiche aufgeteilt. Der Generator liefert wieder die Besuchsreihenfolge (siehe Abb. 4.24).

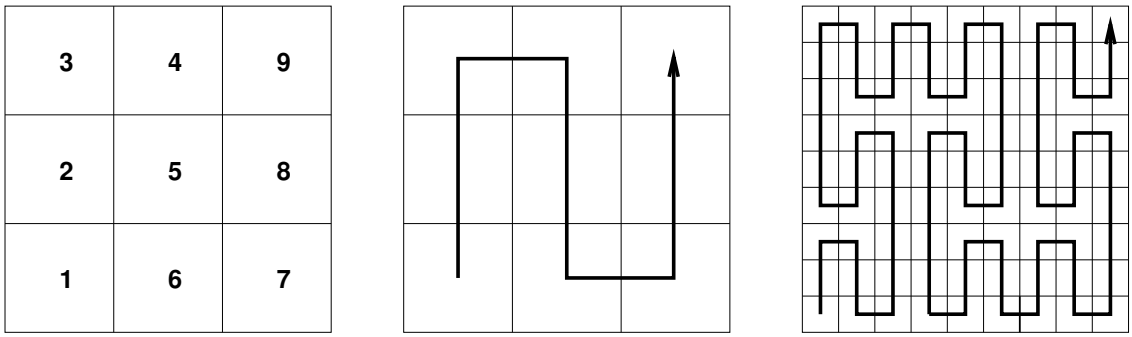


Abbildung 4.24: Leitmotiv und Generierung der Peano-Kurve

Im Gegensatz zur Hilbert-Kurve gibt es jetzt 273 modulo Symmetrie verschiedene Möglichkeiten, den Generator auf den neun Teilquadraten unter Erhaltung von Nachbarschaft und Inklusion rekursiv anzuwenden. Demgemäß gibt es ebenso viele verschiedene Klassen raumfüllender Peano-Kurven. Einige davon zeigt Abb. 4.25.

Eine weitere raumfüllende Kurve mit Anwendungen in der Informatik führte *Lebesgue* ein. Sie unterscheidet sich in zweierlei Hinsicht von den beiden bisher vorgestellten. Diese sind nirgends differenzierbar, die Lebesgue-Kurve ist fast überall differenzierbar. Hilbert- und Peano-Kurven sind *selbstähnlich*, d. h., die Abbildung angewandt auf ein Teilintervall von I ergibt wieder eine raumfüllende Kurve gleicher Struktur, die Lebesgue-Kurve ist dies nicht. Ihre Definition erfolgt zunächst auf der *Cantor-Menge*, die aus I durch sukzessives Entfernen der mittleren Drittel aller verbliebenen Teilintervalle entsteht und das Lebesgue-Maß Null hat. Man definiert

$$f_l(0_3.(2t_1)(2t_2)(2t_3)\dots) := \begin{pmatrix} 0_2.t_1t_3t_5\dots \\ 0_2.t_2t_4t_6\dots \end{pmatrix}, t_i \in \{0,1\},$$

wobei die ternären Argumente aus Nullen und Zweien gerade die Cantor-Menge abdecken, und setzt dies durch lineare Interpolation auf ganz I fort. Interessanterweise gibt es auch hier einen Generator, der in Abb. 4.26 angegeben ist.

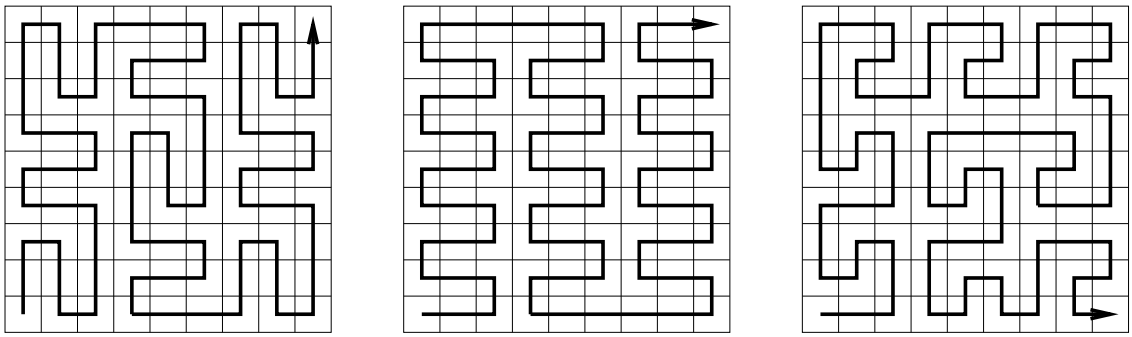


Abbildung 4.25: Peano-Kurven nach Wunderlich: Serpentinentyp (links und Mitte) und Mäandertyp (rechts)

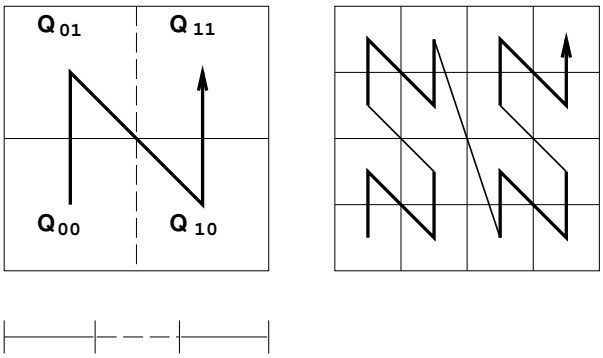


Abbildung 4.26: Leitmotiv der Lebesgue-Kurve

Auch wenn die etwas merkwürdige Art der Verbindung der Leitmotive die Unstetigkeit der Grenzkurve nahelegen mag, so ist dennoch auch die Lebesgue-Abbildung stetig und definiert eine Kurve in unserem Sinn. Die Besuchsreihenfolge im Leitmotiv entspricht der lexikographischen Anordnung und erinnert an Quadtrees bzw. – im Dreidimensionalen – an Octrees. Die fehlende Selbstähnlichkeit der Lebesgue-Kurve ist sofort einsichtig: Das entfernte mittlere Drittel des Einheitsintervalls wird bei obiger Konstruktion auf die vertikale Mittellinie $x = 0.5$ des Einheitsquadrats abgebildet – offensichtlich keine raumfüllende Kurve!

Was auf den ersten Blick wie eine bloße mathematische Spielerei aussieht, hat doch erstaunlich viele Anwendungen, die alle mehr oder weniger mit der Sequentialisierung mehrdimensionaler Daten zu tun haben, etwa bei Suchindizes in Datenbanksystemen. Obwohl man es hier in aller Regel mit hochdimensionalen Datensätzen zu tun hat (Eintrag gleich Dimension), sind eindimensionale Suchindizes immer noch die Regel (B-Bäume basieren auf einem eindimensionalen Primärindex), die dann bei mehrdimensionalen Daten und Suchanfragen entsprechend hintereinandergeschaltet werden müssen (Secondary-Index-B-Bäume). Der Nachteil ist offenkundig: Obwohl nur wenige Deutsche männlich sind und Schuhgröße 58 haben, würden bei der ersten Anfrage rund die Hälfte aller Einträge in der Datenbank aller Deutschen als Treffer gemeldet und weiter bearbeitet werden müssen. Ein echt mehrdimensionaler Suchindex hat dagegen die viel bessere Komplexität $\prod_{i=1}^{d} p_i$, wenn p_i die Trefferquote in einer Dimension angibt. Um jedoch multidimensional an die Daten heranzukommen (etwa in Form eines kleinen Hyperwürfels bei einer Range-Query), ist eine lokalitätserhaltende (stetige) Sequentialisierung ideal, weil man hier einen eindimensionalen Index (B-Bäume verwendbar!) bekommt, der die Mehrdimensionalität der Daten jedoch in sich trägt: Die Daten werden wie Perlen auf einer Schnur aufgereiht, benachbarte Punkte im Einheitsintervall liegen aufgrund der Stetigkeit auch im hochdimensionalen Raum beieinander; umgekehrt gibt's aufgrund der mangelnden Injektivität zwar Ausnahmen (die Urbildbereiche zu einem Volumen

können separiert liegen), die Sequentialisierung mittels raumfüllender Kurven (hier: Lebesgue-Kurve) bewährt sich im Datenbankbereich jedoch bestens (implementiert etwa in den *universellen B-Bäumen (UB-Bäumen)*).

Die uns hier interessierende Anwendung raumfüllender Kurven hat auch mit der Sequentialisierung mehrdimensionaler Daten zu tun, liegt aber in einem ganz anderen Bereich, der Parallelisierung numerischer Simulationsprobleme. Zwei Beispiele:

Bei den sogenannten *Vielteilchen-* oder *N-Körper-Problemen* stehen N Körper über Kräfte (z. B. Gravitationskräfte) in Wechselwirkung. Als sehr unterschiedliche Beispiele seien etwa Astrophysik und Molekulardynamik genannt. In modernen Anwendungen erreicht N Größenordnungen von 10^7 bis 10^8, und gegen mehr hätten Physiker und Chemiker auch nichts einzuwenden. Das Modell führt auf ein System von N gewöhnlichen Differentialgleichungen mit jeweils einem Potential als rechter Seite, in dem die Einflüsse von allen anderen $N-1$ Partikeln auf das jeweils betrachtete Partikel aufakkumuliert werden. Typischerweise nimmt der Einfluß mit zunehmendem Abstand stark ab, was zu Vereinfachungen benutzt werden kann. Vernachlässigt werden dürfen weit entfernte Partikel jedoch nicht ohne weiteres. Es liegen folglich *globale* Kopplungen vor. Die erforderliche Parallelisierung der sehr aufwendigen Berechnungen hat zudem mit der Problematik zu kämpfen, dass die Körper beliebig und stark ungleichmäßig im Raum verteilt liegen können. Außerdem ändern sich die Partikelpositionen mit jedem Zeitschritt – die Wechselwirkungskräfte induzieren eine Bewegung der Partikel.

Mit einer leicht anderen Situation ist man bei *adaptiven Finite-Elemente-* oder *Finite-Differenzen-Verfahren* zur Diskretisierung partieller Differentialgleichungen konfrontiert. Hier liegen N Gitterpunkte an beliebigen Orten im Raum (bei unstrukturierten Gittern) oder an bestimmten potentiellen Orten (bei strukturierten Gittern). Kopplungen finden in der Regel nur *lokal*, also etwa zwischen benachbarten Punkten statt. Im allgemeinen sind die Punkte *stationär*, d. h. sie bewegen sich nicht, dafür kommt aber mit der für die Effizienz unabdingbaren Adaptivität der Gitter ein neues Phänomen ins Spiel, da jetzt im Lauf der Berechnung ständig neue Gitterpunkte erzeugt werden.

Wie können nun raumfüllende Kurven zur effizienten Parallelisierung der geschilderten Problemklassen beitragen? Die grundlegende Idee ist, den Punkten im Raum Punkte auf einer Iterierten einer raumfüllenden Kurve zuzuordnen, die Punkte dann über die entsprechende Abbildung auf dem Einheitsintervall linear anzuordnen und die Partitionierung, d. h. die Zuteilung von Punkten bzw. Teilgebieten zu Prozessoren, gemäß dieser Anordnung vorzunehmen.

Für den ersten Schritt werden vor allem zwei Techniken eingesetzt:

- Man wandelt die (kontinuierlichen) Koordinaten (x, y) in Q in einen Binär- bzw. Quartärcode der Länge k um:

$$\begin{pmatrix} x \\ y \end{pmatrix} \quad \mapsto \quad \begin{pmatrix} 0_2.x_1x_2x_3\ldots x_k \\ 0_2.y_1y_2y_3\ldots y_k \end{pmatrix} \quad \mapsto \quad 0_4.w_1w_2\ldots w_k\,;$$

  diese Vorgehensweise liefert gerade die Quadtree-Blattadressen der Tiefe k bzw. die Anordnung auf der k-ten Iterierten der Lebesgue-Kurve.

- Oder man startet wieder von einer Binärdarstellung der Länge k und bestimmt daraus rekursiv die Position auf der k-ten Iterierten der Hilbert-Kurve.

In beiden Fällen geben die so erhaltenen Binär- bzw. Quartärschlüssel auch bereits Positionen in I an. Anschließend müssen die Schlüssel parallel sortiert werden, was den Aufwand des Verfahrens bestimmt. Abgesehen von der Start-Konfiguration hat man jedoch i. A. die Situation nahezu sortierter Eingaben, da sich Partikel in einem Zeitschritt nur gering bewegen bzw. da pro Iteration nur eine begrenzte Zahl neuer Gitterpunkte erzeugt wird (vgl. Abb. 4.27. Nach einem Berechnungsschritt werden die u. U. neu positionierten Punkte bzw. Partikel auf der Basis der momentanen Partitionierung den Prozessoren zugewiesen (erster Migrationsschritt). Diese sortieren die erhaltenen Punkte lokal und parallel, was aufgrund der Zuteilung ohne Kommunikationsaufwand erfolgen

kann. Anschließend wird die Partitionierung auf der Grundlage der neuen Daten aktualisiert (zweiter Migrationsschritt), um eine eventuell aufgetretene Störung der Lastbalance auszugleichen.

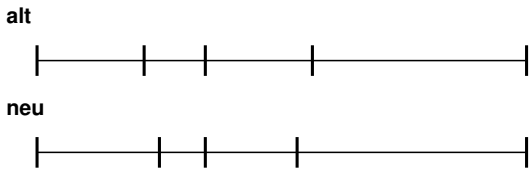


Abbildung 4.27: Änderung der Partitionierung von einem Iterationsschritt zum nächsten

Die Partitionierung kann ungewichtet erfolgen (etwa bei den Gitterpunkten, die typischerweise durch einen konstanten Rechenaufwand pro Punkt gekennzeichnet sind). In diesem Fall erhält jeder Prozessor einfach N/p Punkte. Bei den Partikeln treten dagegen große Unterschiede im jeweiligen Rechenaufwand auf. Hier wird gewichtet partitioniert. Schließlich sei bemerkt, dass die Schlüssel nicht nur zur Sequentialisierung und Partitionierung, sondern auch zur Adressierung und Speicherung der Punkte mittels *Hashing* herangezogen werden.

Für eine Leistungsanalyse der Lastverteilung über raumfüllende Kurven betrachten wir vier Eigenschaften: die Lokalität, den Kommunikationsaufwand, die Qualität sowie die Berechnungskosten der Lastverteilung. Bislang beruhen die meisten Aussagen noch auf empirischen Untersuchungen.

- Zur Lokalität:
  Die Stetigkeit der Abbildungen garantiert, dass nahe beieinanderliegende Urbildpunkte nicht beliebig weit auseinandergerissen werden können, und es gibt für selbstähnliche Kurven Aussagen, die dies im Sinne einer Lipschitz-Stetigkeit quantifizieren. Für die Praxis ist jedoch die umgekehrte Frage wichtiger. Aufgrund der fehlenden Injektivität kann eine perfekte Lokalität natürlich nicht sichergestellt werden. Es stellt sich aber heraus, dass Urbilder von Clustern in Q wieder in wenigen Clustern in I liegen. Im Vergleich der raumfüllenden Kurven untereinander liegt die Hilbert-Kurve vorn.

- Zum Kommunikationsaufwand:
  Die Gebietsgrenzen, die durch die Partitionierung entstehen, sind komplizierter geartet als etwa bei der sukzessiven Koordinatenbisektion in kd-Bäumen (halbiere die Last durch einen Schnitt in y-Richtung, halbiere anschließend in beiden Sektoren die Last durch je einen Schnitt in x-Richtung usw.; vgl. Abb. 4.28). Größe und Gestalt dieser Grenzen werden derzeit untersucht. Insgesamt ist der Kommunikationsaufwand leicht höher als bei der Bisektion.

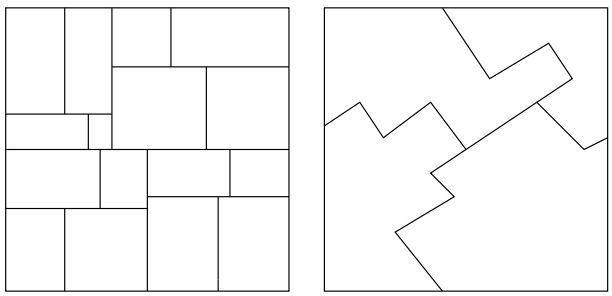


Abbildung 4.28: Partitionierung: kd-Bäume vs. raumfüllende Kurven

- Zur Qualität der Lastverteilung:
  Der Kommunikationsnachteil wird aber durch exzellente Partitionierungseigenschaften mehr als ausgeglichen. Auch bei größerer Prozessorzahl werden hohe Lastverteilungseffizienzen schon bei relativ kleiner Problemgröße erreicht.

- Zu den Kosten der Lastverteilung:
  Selbst bei hoher Prozessorzahl sind die Kosten der Lastverteilung sehr gering. Hier ist nur ein Sortiervorgang erforderlich, im Gegensatz zu den log$p$ Sortiervorgängen bei der Koordinatenbisektion.

Eine abschließende Bemerkung zu raumfüllenden Kurven: Die Begriffe „Selbstähnlichkeit“ und „rekursive Anwendung des Generators auf ein Teilgebiet“ erinnern stark an *Fraktale*, bei deren Definition ja ebenfalls rekursiv angewandte Ähnlichkeitstransformationen im Spiel sind. Die Abbildungen 4.29 und 4.30 zeigen zwei Beispiele für Fraktale, die *Koch-Kurve* und das *Sierpiński-Dreieck*. Beide sind typische Beispiele von „Dimensions-Zwittern“, sie sind weder Kurve noch

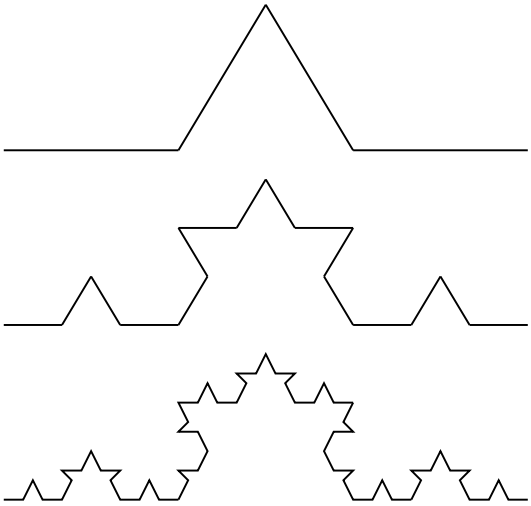


Abbildung 4.29: Die Koch-Kurve

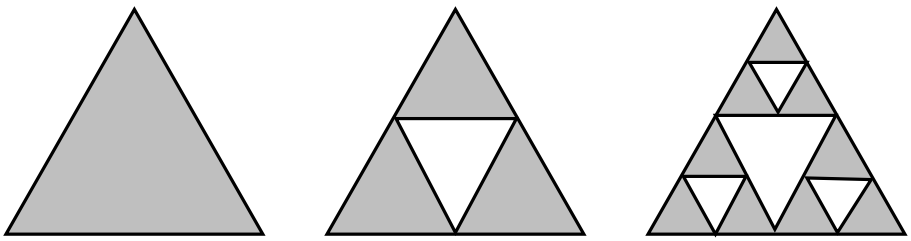


Abbildung 4.30: Das Sierpiński-Dreieck

Fläche. Ihre *fraktale Dimension* liegt bei $\log(4)/\log(3) \approx 1.2619$ bzw. $\log(3)/\log(2) \approx 1.5850$. Die fraktale Dimension ist dabei definiert als $\log(n)/\log(r^{-1})$, wobei n die Anzahl der kleineren Versionen des aktuellen Gebildes im nächsten Verfeinerungsschritt angibt und $r \in]0,1[$ den Verkleinerungsfaktor bezeichnet. Im Falle von „echten“ Kurven oder Flächen ist die fraktale gleich der üblichen topologischen Dimension (d. h. 1 bzw. 2). Da raumfüllende Kurven ja ganz Q überdecken, stellen sie keine Dimensionszwitter oder Fraktale im üblichen Sinne dar und haben demgemäß die fraktale Dimension 2 in 2 D bzw. 3 in 3 D.

4.3.4 Leistungsbewertung

Im Zusammenhang mit der Bewertung der Leistung von Rechnern und Algorithmen sind verschiedene Fragestellungen von Interesse. Einerseits benötigt man geeignete Modelle zur Quantifizierung der Leistung sowie zur Vorhersage der Laufzeitreduktion, andererseits sind Messgrößen

zur Erfassung, genauen Analyse und Beurteilung der Leistung sowie zur Validierung der Modelle erforderlich.

Zunächst muss man zwei Dinge voneinander trennen: Die Qualität eines parallelen Programms bzw. der Parallelisierung eines sequentiellen Programms und die Qualität der Leistungsfähigkeit des parallelen oder verteilten Systems, auf dem das Programm abläuft. Für den Anwender (also den wissenschaftlichen Rechner) ist das jedoch schwer, weil er in aller Regel nur sein Programm auf einem oder allenfalls auf wenigen Rechnern testet.

Die gesamte erbrachte Arbeit bei einer parallelen Berechnung kann angegeben werden als

$$W(p) \;:=\; l \cdot \sum_{i=1}^{p} i \cdot t_i \,,$$

wobei l die Leistung eines Prozessors, p die Zahl der Prozessoren und t_i die Zeit, die genau i Prozessoren beschäftigt sind, angibt. Als *durchschnittlichen Parallelismus* definiert man nun

$$A(p) \;:=\; \frac{\sum_{i=1}^{p} i \cdot t_i}{\sum_{i=1}^{p} t_i} \;=\; \frac{1}{l} \cdot \frac{W(p)}{\sum_{i=1}^{p} t_i} \,.$$

Zum durchschnittlichen Parallelimus gibt es eine Reihe theoretischer Abschätzungen, die meist sehr pessimistisch waren und deshalb oft argumentativ gegen massiv parallele Systeme benutzt wurden. Eine Abschätzung von Minsky aus dem Jahr 1971 prognostiziert für Probleme, bei denen im ersten Schritt noch alle p Prozessoren benutzt werden, im zweiten jedoch nur noch $p/2$ usw. (beispielsweise die parallele Addition von $2p$ Zahlen mittels p Prozessoren), auf SIMD-Systemen einen Wert von $A(p) \;=\; \log_2(p)$. In heutigen Anwendungen spielt diese Problemklasse jedoch keine wesentliche Rolle.

Wichtige Messgrößen sind (wobei jetzt der Einfluss des Algorithmus' und der des Rechners zusammen betrachtet werden):

- der *Speedup* S:
  Sei $T(1)$ die Zeit, die ein Rechner braucht, um ein gegebenes Problem mit dem sequentiellen Programm zu lösen, und sei $T(p)$ die Zeit, die p Rechner für dasselbe Problem mit dem parallelen Programm benötigen. Dann wird definiert:

  $$S \;:=\; \frac{T(1)}{T(p)}$$

- die *Effizienz* E:

  $$E \;:=\; \frac{S}{p} \;=\; \frac{T(1)}{p \cdot T(p)}$$

Bei der Verwendung bzw. Interpretation der Begriffe Speedup und Effizienz ist jedoch Vorsicht geboten. Oft sind diese Werte gerade dadurch besonders beeindruckend, weil ein zwar gut parallelisierbarer, aber schlechter und deshalb kaum eingesetzter sequentieller Algorithmus gewählt wurde (z. B. die Jacobi-Iteration zur Lösung linearer Gleichungssysteme). Es gibt daher auch die strengere Version, in der Definition von Speedup und Effizienz stets den für das gegebene Problem besten bekannten sequentiellen Algorithmus zu verwenden.

In der Regel gilt $S < p$ und somit $E < 1$. Bei Wettbewerbsparallelität etwa kann es aber auch einen *superlinearen* Speedup geben. Ein Beispiel: Gegeben seien n Sortieralgorithmen. Gesucht ist der (für eine bestimmte Eingabe) beste. Beim sequentiellen Test lässt man die n Verfahren hintereinander ablaufen und unterbricht die Bearbeitung jeweils, sobald das aktuelle Minimum überschritten wird. Der sequentielle Vergleich benötigt als Laufzeit somit mindestens $n \cdot t_{\min}$, wenn $t_{\min}$ die

Laufzeit des besten Verfahrens angibt, der parallele Vergleich dagegen kommt bei n Prozessoren mit $t_{\min}$ aus, was – vorausgesetzt, nicht alle Laufzeiten sind gleich – eine Beschleunigung um einen Faktor größer n bedeutet!

Eine (Problem- bzw. Algorithmus-bedingte) Schranke für den Speedup liefert *Amdahls Gesetz*. Ausgangspunkt ist die Erkenntnis, dass jedes Programm einen gewissen Anteil $s \in]0,1[$ an nur sequentiell ausführbaren Operationen enthält. Die als 1 angenommene Ausführungszeit auf einem Prozessor zerfällt damit in einen nicht parallelisierbaren Teil s und in einen parallelisierbaren Teil $1-s$. Daher gilt

$$S \;\leq\; \frac{1}{s + \frac{1-s}{p}} \;\leq\; \frac{1}{s}.$$

Der nicht parallelisierbare Anteil kann also dramatische Auswirkungen auf den Speedup haben. Bei einem sequentiellen Anteil von nur einem Prozent ($s = 0.01$) und einhundert Prozessoren ($p = 100$) ergibt sich ein Speedup von maximal 50.25. Auch auf massiv parallelen Rechnern mit sehr großer Prozessorzahl ist mehr als ein Speedup von 100 nicht drin. Aus diesem Zusammenhang resultiert das Bemühen, den sequentiellen Anteil eines Programms möglichst klein zu halten, was oft auch gelingt: Drei Viertel aller LINPACK-Routinen (siehe Abschnitt 4.4) erfüllen $s < 0.1$. Für präzise Vorhersagen ist Amdahls Gesetz natürlich zu ungenau, grobe Abschätzungen sind jedoch möglich.

Ein alternatives Modell stammt von Gustafson. Er normiert die Bearbeitungszeit auf der parallelen Maschine zu 1, geht dort von einem nicht parallelisierbaren Anteil σ aus und folgert daraus eine Bearbeitungszeit von $\sigma + p \cdot (1-\sigma)$ bei nur einem Prozessor, was unmittelbar in einen Speedup von

$$S \leq \sigma + p \cdot (1-\sigma) \;=\; p + \sigma \cdot (1-p)$$

mündet. Was ist der Unterschied zu Amdahl? Zunächst ist der sequentielle Anteil $s = s(p)$ bezogen auf die Bearbeitungszeit auf einem Prozessor nicht konstant, sondern nimmt mit wachsendem p ab:

$$s(p) \;:=\; \frac{\sigma}{\sigma + p \cdot (1-\sigma)} \;\in\;]0,1[.$$

Dies ist in vielen Fällen realistisch, da mehr Prozessoren vor allem bei steigender Problemgröße eingesetzt werden und hier die parallelisierbaren Teile in der Tat oft zunehmen.

Eine weitere wichtige Größe ist das *Kommunikations-Berechnungs-Verhältnis* (KBV). Es gibt das Verhältnis von Zeit, während der kommuniziert wird, und Zeit, zu der gerechnet wird, an. Wir wissen bereits, dass ein großes KBV für eine Parallelisiserung problematisch ist. Wir betrachten folgendes Problem: Auf eine voll besetzte $N \times N$-Matrix wird ein iteratives Verfahren angewendet, das in jedem Schritt das Element $a_{i,j}$ durch das arithmetische Mittel seiner acht Nachbareinträge ersetzt (am Rand entsprechend). Für einen Iteratonsschritt brauchen wir somit in jeder Zeile der Matrix zwei Nachbarzeilen. Zur Verfügung stehen p Prozessoren. Wir zerlegen zunächst die Matrix in p Blöcke zu N/p Zeilen. Die Berechnungszeit ergibt sich somit zu $N/p \times N$ Zeiteinheiten (o. B. d. A. eine Zeiteinheit pro arithmetischem Mittel), als Kommunikationszeit erhält man $2 \cdot (p-1) \times N$ Zeiteinheiten, und als KBV erhalten wir schließlich $\frac{2 \cdot (p^2-p)}{N}$. Erhöht man p, so verschlechtert sich das KBV. Die Qualität der Parallelisierung hängt also in hohem Maße von der Problem- bzw. Partitionsgröße ab. Außerdem lohnt es sich in den meisten Fällen mehr, in eine schnellere Netzverbindung zu investieren, als die CPUs zu verbessern. Deshalb sind Speedup-Werte für verschiedene Prozessorzahlen, aber gleiche Problemgrößen wenig aussagekräftig.

Eine vor allem bei Vektorrechnern entscheidende Größe ist die Vektorlänge. Aufgrund ihres strikten Pipelinings lieben Vektorprozessoren lange Vektoren. Hockney und Jesshope entwickelten ein Modell, das die Leistung $r(N)$ als Funktion der Vektorlänge N und zweier Parameter r_∞ und $n_{1/2}$ angibt:

$$r(N) \;:=\; \frac{r_\infty}{n_{1/2}/N + 1}\,\text{MFLOPS}.$$

Hierbei gibt r_∞ die Leistung in MFLOPS bei optimaler Vektoränge an, und $n_{1/2}$ bezeichnet die Vektorlänge, bei der man wenigstens noch die halbe Leistung $0.5 \cdot r_\infty$ MFLOPS erzielt.

Abschließend noch eine Bemerkung zur praktischen Ermittlung der Leistung von Rechnern. Hierfür haben sich sogenannte *Benchmarks* eingebürgert, Sammlungen von genau definierten Programmen, die auf genau definierte Probleme angesetzt werden. Im Bereich des Hochleistungsrechnens ist vor allem der LINPACK Benchmark von Interesse, der die zwei Routinen DGEFA (LU-Zerlegung einer Matrix mit partial Pivoting) und DGESL (Lösung eines linearen Gleichungssystems auf der Grundlage dieser LU-Zerlegung) der LINPACK Bibliothek (siehe Abschnitt 4.5) verwendet. Auch für die TOP 500-Liste wird der LINPACK Benchmark eingesetzt.

4.4 Numerics for Supercomputers

4.4.1 Parallel Numerical Linear Algebra

- parallel solution of large linear systems (Wavefront, Red-Black, ...)
- parallel cg

4.4.2 Substructuring and Domain Decomposition

4.5 Werkzeuge des wissenschaftlichen Rechnens

In diesem Abschnitt wollen wir uns mit verschiedenen Klassen von Werkzeugen beschäftigen, die für die Aufgaben des wissenschaftlichen Rechnens zur Verfügung stehen und die über Standard-Tools wie Compiler, Editoren oder Debugger hinausgehen. Obwohl wir dies aus naheliegenden Gründen im Kapitel über Implementierung tun, werden solche Werkzeuge heute in praktisch allen Teilgebieten des wissenschaftlichen Rechnens eingesetzt:

- *Modellbildung*: Programme zur symbolischen Formelmanipulation wie Mathematica, Maple, Axiom oder Reduce können bei Beweisen etc. Unterstützung bieten.
- *Numerik*: Für die Entwicklung sowie für die Analyse von Algorithmen stehen neben den oben erwähnten Computeralgebra-Systemen mächtige Programmpakete wie MATLAB zur Verfügung. Mit den dort jeweils implementierten High-Level-Sprachen lassen sich Prototypen einfach und schnell realisieren.
- *Implementierung*: Die meisten Codes für Simulationsaufgaben bestehen zu einem Großteil aus Programmstücken für immer wiederkehrende Teilaufgaben der numerischen linearen Algebra. Für solche Aufgaben bieten für die gängigen Programmiersprachen zahlreiche Programmbibliotheken fertige Lösungen an.
- *Visualisierung*: Auch hierfür gibt es kommerzielle Programmsysteme wie etwa IDL, IRIS Explorer oder AVS/Express, mit deren Hilfe Ergebnisdaten aus Simulationsläufen aufbereitet und auf vielerlei Arten graphisch dargestellt werden können.

Im Folgenden werden einige Programmbibliotheken bzw. Programmpakete sowie die Systeme Maple und MATLAB kurz vorgestellt. Der Grad der Kommerzialisierung ist dabei sehr unterschiedlich: Von der Public Domain Bibliothek über Produkte von non-profit Einrichtungen zum Selbstkostenpreis bis hin zur teueren Lizenz ist dabei alles vertreten. Für alle Systeme gilt jedoch, dass sie keine fertigen Codes für eine bestimmte eingeschränkte Klasse von Anwendungen darstellen, sondern als Werkzeuge die Entwicklung und Implementierung solcher Programme erleichtern.

4.5.1 Programmbibliotheken und -sammlungen

GAMS

Siehe auch die WWW-Seite `http://gams.nist.gov/`.

Eine Übersicht über mathematische Software ganz allgemein (mit den Schwerpunkten Modellierung, Simulation und Statistik) bietet der *Guide to Available Mathematical Software (*GAMS*)* des US-amerikanischen *National Institute of Standards and Technology (NIST)* auf seiner Web-Seite `http://gams.nist.gov/`. Über einhundert Programmpakete und -bibliotheken mit einigen zigtausend Routinen sind dort katalogisiert, referenziert und beschrieben. Suche und Zugriff können problemklassengetrieben über einen hierarchischen Baum (Zahlentheorie, Statisik, lineare Algebra oder ...?), mittels Schlagworten (welches Problem soll gelöst werden?) oder über Paket- bzw. Modulname erfolgen. Mehrheitlich handelt es sich hierbei um FORTRAN-Programme für mathematische (Teil-) Aufgaben von Problemstellungen des wissenschaftlichen Rechnens wie die Lösung linearer Gleichungssysteme, Eigenwertberechnungen, Nullstellen- und Extremwertbestimmung nichtlinearer Funktionen, (einfache) Systeme von Differentialgleichungen, Regression etc.

Die im GAMS katalogisierten Pakete liegen z. T. auf Servern des NIST, z. T. in der öffentlich zugänglichen Programmbibliothek NETLIB (siehe unten), teilweise handelt es sich auch um kommerzielle (d. h. lizenzgebundene) Produkte.

Matrix Market

Siehe auch die WWW-Seite `http://math.nist.gov/MatrixMarket/`.

Der Matrix Market ist eine Sammlung anwendungsrelevanter Testdaten. Er enthält derzeit ca. 500 dünn besetzte Matrizen aus den verschiedensten Anwendungsfeldern sowie Programme zur Generierung von Testmatrizen.

NETLIB

Siehe auch die WWW-Seiten `http://www.netlib.org/` bzw. `http://www.netlib.bell-labs.com/`.

Das NETLIB „Lager“ enthält frei erhältliche Software für numerische Probleme und Aufgabenstellungen des wissenschaftlichen Rechnens sowie weitere Dokumente und Datenbanken zu diesen Themenkreisen. NETLIB wird betrieben von der University of Tennessee und vom Oak Ridge National Laboratory in Knoxville, TN, sowie von den Bell Labs von Lucent Technologies in Murray Hill, NJ. Um weltweit einen schnellen Zugriff sicherstellen zu können, wird die Programmsammlung an mehreren Standorten gespiegelt, d. h. als Kopie gespeichert und automatisch bzw. regelmäßig mit dem Original synchronisiert. Hauptstandorte mit automatischer täglicher Synchronisation sind neben Knoxville und Murray Hill die *Higher Education National Software Archives (HENSA)* an der University of Kent in Canterbury sowie ein Server in Bergen. Daneben gibt es ein gutes Dutzend lose angekoppelter Kopien, bspw. in Deutschland am *Konrad Zuse Zentrum für Informationstechnik Berlin (ZIB).*

Der Zugriff kann via Internet (WWW, email, ftp) sowie über CD-ROM erfolgen. Die Zahl der Online-Zugriffe von 1985 bis Mitte April 1999 lag übrigens bei über 41 Millionen, davon allein in 1999 mehr als 5.5 Millionen. Diese verteilen sich wie folgt: ca. 90 Prozent WWW/http, ca. 8 Prozent ftp und ca. 2 Prozent email. Die Verwendung der meisten erhältlichen Programme ist nicht eingeschränkt. Natürlich kann man auch eigene Programme zur Aufnahme einreichen.

Derzeit (Stand: April 1999) enthält NETLIB knapp 160 Bibliotheken. Einige Beispiele:

- BLAS (Basic Linear Algebra Subprograms):
  Bei den BLAS, die zu einer Art Standard für numerische Software im Bereich elementarer Vektoroperationen geworden sind, handelt es sich um eine Sammlung robuster, effizienter und portabler Programmbausteine zur Ausführung von Vektor- und Matrixoperationen, die ihrerseits für die Entwicklung leistungsfähiger Software zur numerischen linearen Algebra (beispielsweise die Bibliotheken LAPACK oder LINPACK, s. u.) verwendet werden. Ziel war es, dem Stand der Technik entsprechende Module für die in Simulationscodes immer wiederkehrenden Aufgaben der numerischen linearen Algebra bereitzustellen und durch das modulare Konzept das einfache Upgraden großer Simulationscodes durch Austausch der Unterprogramm-Aufrufe zu ermöglichen. Ferner wird dem Anwender die ständige Berücksichtigung von Fehlerfällen wie Überlauf o. Ä. abgenommen. Die Referenz-Sammlung umfasst FORTRAN-Routinen, es gibt aber auch für spezielle Rechnertypen optimierte BLAS-Bibliotheken (aufrufbar aus FORTRAN-, C- und C++-Programmen). Neuerdings sind auch Java BLAS erhältlich. Die Erweiterung SPARSE-BLAS bietet spezielle Module für dünn besetzte Matrizen.
  Die BLAS Routinen gliedern sich in drei Gruppen. Der sogenannte Level 1 enthält Module für einfache Vektor- und Vektor-Vektor-Operationen (Norm, Skalarprodukt, Addition, die berühmte SAXPY-Operation $y := \alpha \cdot x + y$ etc.). In der Regel ist hier das KBV zu schlecht, als dass sie das Potential von Hochleistungsrechnern voll ausschöpfen könnten. Im Level 2 sind Matrix-Vektor-Operationen wie $\alpha Ax + \beta y$, $A + \alpha xy^T$ (Rang-1-Modifikation) oder die Lösung tridiagonaler Gleichungssysteme zusammengefasst. Wie die Erfahrung gezeigt hat, sind die Routinen des Level 2 für Vektorprozessoren hervorragend geeignet, für Parallelrechner im engeren Sinne sowie für Monoprozessoren mit ausgeprägter Speicherhierarchie erwies

sich das KBV jedoch als noch immer nicht ausreichend. Deshalb wurden mit dem Level 3 Routinen für Matrix-Matrix-Operationen wie $\alpha AB + \beta C$ oder Rang-k-Modifikationen erstellt. Bei Operationen dieses Typs benötigt man oft $O(n^3)$ Gleitpunkt-Operationen bei nur $O(n^2)$ auszutauschenden Daten, hat also ein wesentlich günstigeres KBV.

- EISPACK:
  Sammlung von FORTRAN-Routinen zur Berechnung von Eigenwerten und Eigenvektoren, im Wesentlichen ersetzt durch LAPACK

- LINPACK:
  Sammlung von FORTRAN-Routinen zur Lösung von linearen Gleichungssystemen sowie von linearen Ausgleichsproblemen; entwickelt für die Hochleistungsrechner der siebziger und der frühen achtziger Jahre, heute im Wesentlichen ersetzt durch LAPACK

- LAPACK (Linear Algebra PACKage):
  Bei der LAPACK Bibliothek handelt es sich um eine populäre Sammlung von FORTRAN-Programmen für die gängigsten Probleme aus der numerischen linearen Algebra (lineare Gleichungssysteme, Ausgleichsprobleme, Eigenwert- und Singulärwertprobleme etc.). Es werden Routinen für voll besetzte Matrizen sowie für Bandmatrizen angeboten, jedoch nicht für allgemeine dünn besetzte Matrizen. Die älteren Bibliotheken EISPACK und LINPACK wurden zusammengefasst und vor allem im Hinblick auf den Einsatz der Programme auf modernen Monoprozessor-Architekturen, Vektor- und Parallelrechnern mit gemeinsamem Speicher weiterentwickelt (Reduktion der Speicherzugriffe, Block-Operationen zur Abstimmung auf mehrschichtige Speicherhierarchien). Auch LAPACK stützt sich auf die BLAS ab. Erweiterungen von LAPACK sind LAPACK90 (FORTRAN 90-Schnittstelle), CLAPACK (C-Version), LAPACK++ (C++-Implementierung), sowie ScaLAPACK (Implementierung ür MIMD-Systeme mit verteiltem Speicher (via MPI, PVM)).

- FFTPACK:
  Sammlung von FORTRAN-Unterprogrammen zur schnellen Fourier- (Sinus-, Cosinus- ...) Transformation

- FISHPACK:
  Sammlung von FORTRAN-Unterprogrammen zur Finite-Differenzen-Diskretisierung elliptischer Randwertprobleme (z. B. Helmholtz-Gleichungen in unterschiedlichen Koordinatensystemen)

- HOMPACK:
  Sammlung von FORTRAN-Unterprogrammen zur Lösung nichtlinearer Gleichungssyteme mittels Homotopiemethoden

- ITPACK:
  Sammlung von FORTRAN-Unterprogrammen zur iterativen Lösung großer dünn besetzter linearer Gleichungssysteme

- MINPACK:
  Sammlung von FORTRAN-Unterprogrammen zur Lösung nichtlinearer Gleichungen und nichtlinearer Least-squares-Probleme

- MPI:
  MPI-Standard

- ODEPACK:
  Sammlung von FORTRAN-Unterprogrammen zur Lösung von Anfangs- und Randwertproblemen gewöhnlicher Differentialgleichungen

- PDES:
  verschiedene Codes: MGGHAT (2D elliptische partielle Differentialgleichungen auf polygonal

berandeten Gebieten; Adaption, Mehrgitter, lineare, quadratische und kubische Elemente), CLAW (hyperbolische Systeme von Erhaltungsgesetzen) etc.

- PLTMG:
  Programm zur Lösung elliptischer partieller Differentialgleichungen auf gewissen zweidimensionalen Gebieten (inkl. lokaler Gitteradaption, Mehrgitterlöser, Gittergenerator, graphische Ausgabe)
- PVM3:
  Bibliothek für eine PVM-Implementierung
- QUADPACK:
  Sammlung von FORTRAN-Unterprogrammen zur Berechnung bestimmter univariater Integrale
- ScaLAPACK:
  ScaLAPACK, eine Weiterentwicklung von LAPACK, ist eine Sammlung von Programmen zur numerischen linearen Algebra, die speziell für die Anwendung auf MIMD-Rechnern mit verteiltem Speicher konzipiert bzw. optimiert worden sind. Das Zauberwort ist hierbei die Skalierbarkeit: Während LAPACK nur für Maschinen mit moderater Prozesorzahl (etwa kleiner hundert) gedacht ist, sind die Routinen von ScaLAPACK für eine beliebige Anzahl von Prozessoren geeignet. ScaLAPACK stützt sich ab auf die BLAS, auf die BLACS (Basic Linear Algebra Communication Subroutines) sowie auf die PBLAS (Parallel Basic Linear Algebra Subroutines), einer Version für verteilten Speicher der sequentiellen BLAS Routinen.
- SPARSE:
  Sammlung von C-Unterprogrammen zur Lösung großer dünn besetzter linearer Gleichungssysteme mittels LU-Zerlegung
- TEMPLATES:
  Programmsammlung zum Buch *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM
- VORONOI:
  Voronoi Diagramme und Delaunay Triangulierungen

Visual Numerics

Siehe auch die WWW-Seite `http://www.vni.com/`.

Visual Numerics unterhält ebenfalls große Bibliotheken mathematischer Software. Der Vorgänger IMSL (The International Mathematical and Statistical Libraries) war über Jahre hinweg die Adresse für mathematische Bibliotheksprogramme.

Diffpack

Siehe auch die WWW-Seite `http://www.nobjects.com/top-index.html`.

Diffpack ist eine Entwicklungsumgebung für Programme zur numerischen Simulation von Mehrphysik-Problemen (siehe Kapitel 5). Diffpack wird von der norwegischen Firma *Numerical Objects* vertrieben und ist für die meisten UNIX-Plattformen erhältlich. Als modernes Werkzeug für den Einsatz objektorientierter Numerik in Simulationsanwendungen fährt Diffpack eine Doppelstrategie, um die Vorteile des objektorientierten Entwurfs mit Rechenzeiteffizienz zu verbinden: Die rechenzeitintensiven numerischen Berechnungen werden auf herkömmliche Art, also im Stil eines optimierten imperativen Unterprogramms realisiert, während die hierarchisch höheren Programmteile bzw. Schnittstellen den Prinzipien des objektorientierten Entwurfs folgen.

Dementsprechend kann man Diffpack auf zwei Arten benutzen:

- als Sammlung zuladbarer Bibliotheken, um Diffpacks Funktionalität auch von anderen Programmen aus zu nutzen, oder
- als eigenständige Entwicklungsumgebung, mit deren Hilfe Code für eigene Simulationsanwendungen effizient entwickelt werden kann.

Diese Flexibilität erlaubt es dem Anwender, Diffpack für ganz unterschiedliche Probleme einzusetzen.

NAG

Siehe auch die WWW-Seite `http://www.nag.co.uk/`.

Die *Numerical Algorithms Group (NAG)* ist eine non-profit Software-Firma, die sich auf die Entwicklung von Programmen für technische und wissenschaftliche Anwendungen spezialisiert hat. Ausgehend von Oxford hat die NAG im 29. Jahr ihres Bestehens Niederlassungen in mehreren Ländern und mehrere Tausend Kunden im Hochschul- und im industriellen Bereich. Zu den für uns interessanten Produkten zählen

- die Fortran Library: über 1200 Subroutinen für numerische und statistische Aufgaben (via C Header Files auch von C bzw. C++ aus aufrufbar),
- die Fortran 90 Library,
- die C Library: über 250 Subroutinen für numerische und statistische Aufgaben,
- die Parallel Library: eine numerische Bibliothek, basierend auf MPI oder PVM, für verteilte Berechnungen,
- die Fortran SMP Library: eine numerische Bibliothek, optimiert für symmetrische Multiprozessor-Maschinen,
- die Finite Element Library: FORTRAN-Unterprogramme für Finite-Element-Berechnungen,
- AXIOM: ein System für symbolische Berechnungen,
- die Graphics Library: eine FORTRAN-Graphikbibliothek (via C Header Files auch von C bzw. C++ aus aufrufbar),
- IRIS Explorer: ein Visualisierungspaket, aufbauend auf OpenGl und Open Inventor (siehe Kapitel 6),
- Fastflo: ein 3D Finite-Element-Paket für struktur- und strömungsmechanische sowie andere Anwendungen.

Für die Produkte von der NAG müssen Lizenzen erworben werden.

Numerical Recipes

Siehe auch die WWW-Seite `http://www.nr.com`.

Numerical Recipes: The Art of Scientific Computing ist der Titel einer Reihe von Büchern, die von Numerical Recipes Software entwickelt wurden und von Cambridge University Press verlegt werden. Der Name Numerical Recipes bezieht sich auch auf die (nicht frei verfügbare) Software, die in obigen Büchern beschrieben ist. Bücher und Programme sind erhältlich für die Sprachen C und FORTRAN (77 und 90), es gibt aber auch authorisierte Versionen für PASCAL, BASIC, LISP, MODULA 2 oder IDL. Die C- und FORTRAN-Versionen enthalten jeweils ca. 350 Einzelroutinen für Problemstellungen der Numerik allgemein und des wissenschaftlichen Rechnens

(lineare Gleichungssysteme, Interpolation und Extrapolation, numerische Quadratur, Funktionsauswertung (inkl. Differentiation und Approximation), spezielle Funktionen, Zufallszahlen, Sortieren, Nullstellen, Extremwertbestimmung, Eigenwerte, FFT, Wavelets, Statistik, gewöhnliche Differentialgleichungen, Integralgleichungen, partielle Differentialgleichungen).

Für die Programme muss eine Lizenz gekauft werden, interessant und lehrreich ist aber auch oder gerade das Studium der in den Büchern beschriebenen Algorithmen.

- *Press, Teukolsky, Vetterling, and Flannery*, *Numerical Recipes in C*, Cambridge University Press, 1993.
- *Press, Teukolsky, Vetterling, and Flannery*, *Numerical Recipes in Fortran 77*, Vol. 1 of Fortran Numerical Recipes, Cambridge University Press, 1993.
- *Press, Teukolsky, Vetterling, and Flannery*, *Numerical Recipes in Fortran 90*, Vol. 2 of Fortran Numerical Recipes, Cambridge University Press, 1996.
- *Press, Teukolsky, Vetterling, and Flannery*, *Numerical Recipes in Pascal*, Cambridge University Press, 1991.

Neben den „Hauptbüchern“ gibt es auch ergänzende Bücher mit Beispielen.

4.5.2 Werkzeuge für Entwicklung, Analyse und Prototyping von Algorithmen

Während Programmbibliotheken erprobte und effiziente Bausteine für Einzelaufgaben in einem Simulationsprogramm bereitstellen und somit die Erstellung und Implementierung des „Endprodukts“ unterstützen, wollen wir uns jetzt mit Programmpaketen befassen, die den Entwurf, das Testen und die Analyse von Algorithmen erleichtern. Solche Systeme dienen also vor allem dem *Rapid Prototyping*, d. h. der schnellen Erstellung von Programmen, die in wesentlichen Teilen bereits die Funktionalität des Endprodukts haben, aber auf jeden Ballast (zur Laufzeit- oder Speicheroptimierung etwa) verzichten. Insbesondere kann man in intuitiven (script-ähnlichen) High-Level-Sprachen programmieren, die ohne den Deklarations-Overhead (Typvereinbarungen, Variablendeklaration, Klassendefinition) auskommen.

Zunächst sind hierbei die verbreiteten Computeralgebra-Programme zu erwähnen, also hauptsächlich Mathematica und Maple. Während diese wie auch Reduce oder Axiom ursprünglich zum Zwecke symbolischer Berechnungen (Formelmanipulation) entworfen worden waren, mit Anwendungen auch in Feldern der reinen Mathematik (Algebra, Zahlentheorie), haben sie sich inzwischen auch in ganz anderen Bereichen als Werkzeug etabliert, beispielsweise in der mathematischen Ausbildung oder eben auch im wissenschaftlichen Rechnen. Wir wollen uns Maple etwas genauer anschauen.

Maple

Siehe auch die WWW-Seite `http://www.maplesoft.com/`.

Das Computeralgebra-Programm Maple wird von *Waterloo Maple Inc.* entwickelt und vertrieben, einem 1988 gegründeten Spin-off-Unternehmen der University of Waterloo in Ontario. Maple sieht sich heute jedoch zurecht nicht mehr nur als reines Computeralgebra-Programm, sondern vielmehr als „interaktive Umgebung zur mathematischen Problemlösung und Programmierung“, und bietet Möglichkeiten der symbolischen und numerischen Berechnung sowie der Visualisierung. Dementsprechend liegen die Schwerpunkte

- bei symbolischen Berechnungen, also bei der Formelmanipulation algebraischer Ausdrücke (z. B. Primfaktorzerlegung, Differentiation und Integration, Potenzreihenentwicklung),
- bei numerischen Berechnungen in beliebiger Genauigkeit,

- bei zwei- und dreidimensionaler graphischer Ausgabe sowie
- bei der Programmierung, d. h. bei der Entwicklung eigener Module.

Die Maple-Bibliotheken umfassen dabei mehr als 2700 verschiedene Routinen.

Symbolisch rechnen heißt mit Symbolen (Variablen) rechnen. Für herkömmliche mathematische Programme ist dies unmöglich – unabhängig vom Schwierigkeitsgrad für den menschlichen Bearbeiter, der mit so einfachen Ausdrücken wie $x + x$ oder $\mathrm{d}x^2/\mathrm{d}x$ durchaus noch etwas anzufangen weiß. Ein konventionelles Programm kommt hier nur weiter, wenn die Variable mit einem konkreten Wert belegt wird und wenn im zweiten Fall zusätzlich numerische Näherungen zugelassen werden. Symbolische Programme eröffnen dagegen die Möglichkeit, Integrale, Eigenwerte, lineare Gleichungssysteme, Systeme von Differentialgleichungen etc. soweit möglich analytisch zu behandeln, berechnen oder lösen. Ergänzt durch numerische Lösungsalgorithmen in beliebiger Genauigkeit, Graphikroutinen sowie eine einfache und mächtige Interpreter-Sprache, entstehen so breit einsetzbare Systeme für verschiedenartige mathematische Aufgabenstellungen.

Die interaktive graphische Oberfläche von Maple kann vielfältig eingesetzt werden: zur interaktiven Problemlösung, zu parametrisierten Testläufen oder zur Erstellung hierarchischer Dokumente, etwa im Ausbildungsbereich. Das Standard-Dokument einer Maple-Sitzung ist das *Arbeitsblatt (worksheet)*. Es kann Text, Formeln und Prozeduren sowie berechnete Ergebnisse oder Graphiken und Visualisierungen enthalten. Die Möglichkeit der hierarchischen Strukturierung von Arbeitsblättern erlaubt es zudem, bspw. Details eines Beweises zunächst auszublenden und erst bei späterer genauerer Betrachtung „aufzuklappen“.

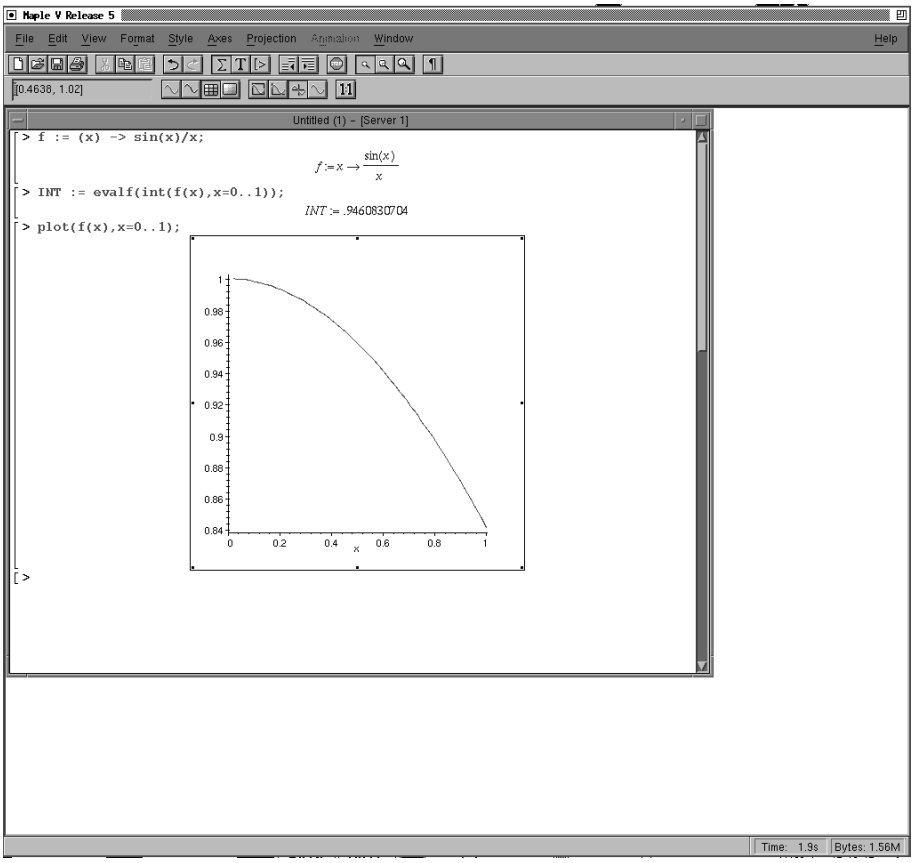


Abbildung 4.31: Maple-Oberfläche mit einem Arbeitsblatt

Auch für Maple gilt natürlich in erster Linie das Prinzip des *learning by doing*, und in den Übungen

wird ja auch Maple an ganz verschiedenen Stellen eingesetzt. An dieser Stelle geht es mehr darum, ein Gefühl für die Möglichkeiten von Maple zu bekommen. Deshalb soll hier ein kleiner Überblick über seine Möglichkeiten genügen.

- Als symbolisches Programm rechnet Maple mit ganzen und rationalen Zahlen exakt. Aus `1+1/2` wird folglich `3/2` und nicht `1.5`. Dies gilt auch für irrationale Zahlen, die nicht als Gleitpunktzahlen dargestellt sind. So wird `Pi/6` als solches und nicht als Gleitpunktzahl geführt. Damit wird der (negative) Einfluss der Numerik weitgehend zurückgedrängt: `sin(Pi)` ist immer Null, $\sqrt{2}^2$ ist immer zwei, aus `sin(3.1415927)` oder 1.4142^2 kann im Laufe der Zeit vieles werden. Die Stellenzahl bei ganzen Zahlen ist nur durch die Leistungsfähigkeit des Rechners und die (kaum zu erreichende) Zahl von ca. 500 000 Stellen begrenzt. Mittels `evalf` wird auf Gleitpunktdarstellung umgeschaltet, wobei die zu verwendende Stellenzahl mittels `Digits` vorgegeben werden kann. Dieses Verharren bei exakter Rechnung gilt auch für Funktionsauswertungen. So lässt Maple `ln(Pi)` lieber unverändert, bevor es den Ausdruck in eine nur angenäherte Form umwandelt.
- Maple kennt neben Zahlen eine Reihe von Objekttypen wie Folgen, Listen, Mengen, Felder und Tabellen.
- In Maple können Ausdrücke vereinfacht (`simplify`, wobei die Meinung von System und Anwender über den Begriff „einfacher“ manchmal voneinander abweichen können), faktorisiert (`factor`), entwickelt (`expand`) oder anderweitig manipuliert werden.
- Ein Schwerpunkt liegt bei klassischen Aufgaben der Analysis (Differentiation, Integration) sowie auf der Bestimmung von Lösungen (direktes Lösen allgemeiner Gleichungen, numerische Lösung, ganzzahlige Lösungen, Lösungen modulo n, Lösung gewöhnlicher Differentialgleichungen, Lösung (einfacher) partieller Differentialgleichungen).
- Maple besteht aus seinem *Kern* (fundamentale und primitve Befehle, optimierter C-Code, ca. 10% des gesamten Systems, wird zu Beginn geladen), aus der *Hauptbibliothek* (am häufigsten verwendete Maple-Befehle, werden bei Bedarf automatisch geladen), aus der *gemischten Bibliothek* (seltenere Befehle, explizit vom Benutzer zu laden) und aus den über 30 *Paketen*. Jedes Paket muss bei Bedarf geladen werden und enthält eine Gruppe von Befehlen für verwandte Aufgabenstellungen. Beispiele für Pakete sind `linalg` (über 100 Befehle zur linearen Algebra), `plots` (Befehle für Klassen spezieller Zeichnungen), `simplex` (lineare Optimierung), `stats` (Statistik) oder `student` (Analysis, Kurvendiskussion).
- Ohne graphische Ausgabe wäre natürlich ein Programm wie Maple heute undenkbar. Dementsprechend sind eine Vielzahl von Möglichkeiten vorgesehen für graphische Darstellungen in zwei und drei Dimensionen (inkl. rudimentärer Schattierung und Beleuchtung), für einfache Animationen, für Visualisierungen (z. B. Vektorfelder), für die Kombination von Text und Bildern sowie für die Manipulation graphischer Objekte (Skalierung, Rotation etc.).
- Von besonderem Interesse sind die Schnittstellen von Maple nach LaTeX und nach HTML. Sowohl einzelne Maple-Befehle als auch ganze Arbeitsblätter können als LaTeX-Ausdruck bzw. -Datei exportiert werden, ebenso wie als HTML-Datei. Das HTML-Dokument gibt dabei die Gliederungsstruktur mittels Hyperlinks wieder, Maple-Animationen werden als animated gifs integriert und sorgen somit für Bewegung.
- Die Programmiersprache Maple schließlich erlaubt ein effizientes Prototyping. Verfahren zur numerischen Lösung von Differentialgleichungen, iterative Löser für große lineare Gleichungssysteme, ja sogar rekursive Algorithmen können mit ihr schnell und ohne großen Ballast entworfen und getestet werden. Als Beispiel sei hier das in Abb. 4.12 vorgestellte Maple-Programm zur numerischen Quadratur nach Archimedes genannt.

Bisher wurde vor allem die Rolle von Maple als interaktives Werkzeug zur Lösung mathematischer Probleme beschrieben. Wie eingangs erwähnt, kann Maple aber auch zur Erstellung interaktiver Dokumente verwendet werden, beispielsweise eines technischen Berichts, eines Vortrags oder eines

Vorlesungsskripts. Im Gegensatz zu nicht-interaktiven Medien (Papier, Folien, Dias) besteht jetzt die Möglichkeit, die einzelnen Rechenschritte explizit vorzführen. Mit Hilfe des Entfaltens und Zusammenklappens von Teilen des Dokuments kann zudem Struktur in die Präsentation gebracht werden: Man zeigt zunächst das Inhaltsverzeichnis oder die groben Schritte eines Algorithmus' und klappt dann bei Bedarf einzelne Punkte auf. Dies geht immer einher mit der Möglichkeit interaktiven Eingreifens in die Berechnungen (Parameter anders setzen etc.), was ein signifikanter Unterschied zum Browsen durch HTML-Dokumente ist.

Zur Demonstration der Verwendbarkeit von Maple im wissenschaftlichen Rechnen sind mit diesem Skript auch ein Maple-Worksheet sowie sein entsprechendes HTML-Dokument erhältlich. Darin werden diskutiert

- einführende Beispiele,
- Beispiele partieller Differentialgleichungen (die eindimensionale Wellengleichung sowie die eindimensionale Wärmeleitungsgleichung samt Fourier-Analyse),
- Beispiele gewöhnlicher Differentialgleichungen (zu den Polpulationsmodellen aus Abschnitt 2.2.2),
- iterative Verfahren zur Lösung linearer Gleichungssysteme (Jacobi-Iteration, Mehrgitterverfahren).

MATLAB

Siehe auch die WWW-Seite `http://www.mathworks.com/`.

Im Gegensatz zu den oben angeführten Computeralgebra-Programmen war MATLAB zunächst hauptsächlich für den Einsatz in der (Mathematik-) Ausbildung gedacht, vor allem vor dem Hintergrund technisch-wissenschaftlicher Anwendungen. Heute sieht sich MATLAB als „High Performance Numeric Computation und Visualization Software“ und somit als Entwicklungs-, Programmier-, Berechnungs- und Visualisierungsumgebung für Aufgabenstellungen des wissenschaftlichen Rechnens. MATLAB sollte vor allem einen direkteren und komfortableren Zugang zur rechnergestützten Mathematik erlauben, als dies etwa mittels C- oder FORTRAN-Programmen möglich ist. MATLAB wird von *The Math Works Inc.* entwickelt und vertrieben, einer 1984 gegründeten Firma bei Boston mit heute über 500 Mitarbeitern. Seine Verbreitung ist beachtlich: In über 100 Ländern wird MATLAB an über 2000 Hochschulen, in außeruniversitären Forschungseinrichtungen sowie in Verwaltung und Industrie eingesetzt.

Kapitel 5

Embedding and Integration

Die numerische Simulation hat sich als isolierte Dienstleistung entwickelt, die auf Interaktion mit anderen Aufgabenstellungen in nur sehr eingeschränktem Maße bedacht war. Was benötigt wurde, wurde selbst erzeugt: Routinen zum Pre-Processing für die Eingabenerzeugung bzw. Netzgenerierung, Routinen zum Post-Processing für die Nachbearbeitung der Ausgabe (Visualisierung etc.). Zudem sind die meisten Simulationscodes hochspezialisierte One-purpose-Lösungen, die die Simulation eines physikalischen Phänomens oder einer technischen Konfiguration gestatten, aber nur bedingt allgemeiner eingesetzt werden können.

Beides ist heute nicht mehr akzeptabel. Die Simulation findet eben nicht lösgelöst statt, sondern ist integriert in einen Produktionsprozess, eine Ablaufsteuerung oder eine Optimierung. Bei der Konzeption eines neuen PKW beispielsweise stehen am Anfang per Hand erzeugte Skizzen des Designers. Diese werden heute computergestützt in rudimentäre Oberflächenmodelle umgewandelt. Dann setzt das klassische CAD an, der detaillierte Entwurf wird ausgeführt. Im Idealfall sollte bereits dieser Prozess begleitet werden von numerischen Simulationen: Ist die entworfene Karosserie aerodynamisch zufriedenstellend, lässt sich der Innenraum bequem belüften, ist der Fahrzeugschwerpunkt nicht zu hoch für eine stabile Straßenlage? Die Simulation liefert also ein Feedback an das CAD zurück, sie ist *eingebettet* in den gesamten Entwurfsprozess. Damit erlangen Fragestellungen wie eine einheitliche Datenbasis für Modellierung, Simulation und Visualisierung oder Schnittstellen der Simulation nach außen eine zentrale Bedeutung.

Ferner hat man es immer seltener mit der Simulation einzelner Effekte wie Strömungsmechanik, Strukturmechanik oder Elektrodynamik zu tun. Vielmehr spielen *Wechselwirkungen* oder *Kopplungen* eine immer größere Rolle.

Im Folgenden befassen wir uns deshalb kurz mit diesen beiden Problemkreisen Schnittstellen und Wechselwirkungen.

5.1 Schnittstellen

Eine zentrale Thematik bei der Einbettung der Simulation wie auch in industriellen Abläufen ganz allgemein ist das Problem der Geometriebeschreibung. Jeder Arbeitsschritt verwendet hierfür seine eigenen Techniken und erzeugt neue geometrische Modelle:

- Die *geometrische Modellierung* liefert erste rechnergestützte Beschreibungen der betreffenden Objekte. Bei technischen Objekten sind hier die Darstellungsarten des *Computer Aided Design (CAD)* gefragt (siehe das Kapitel zur Visualisierung), bei natürlichen Objekten (Präparaten, Bodenproben) kommen u. U. tomographische Techniken zum Einsatz.

- Die numerische Simulation erzeugt sich anschließend ihre eigenen Rechen-Geometrien (Finite-Element-Netze etc.).
- Die Visualisierung schließlich verwendet entweder maßgeschneiderte Interfaces zu bestimmten Simulationscodes, oder aber sie arbeitet wiederum mit einer eigenen Geometriebeschreibung.

Im günstigsten Fall kommen hier aufwendige Interpolatonstechniken zum Einsatz, um die Transformation von einer in eine andere Darstellungsart durchzuführen. Oftmals passiert aber nichts automatisch, die neuen Modelle werden vielmehr von Hand auf der Grundlage der alten Modelle erzeugt – in einem Prozess, der mehrere Mann-Monate erfordern kann. Gerade die effizienten Einzellösungen machen bei der Umwandlung oft Probleme (u. U. geht die Effizienz verloren). Zudem ist die Transformation oft eine Einbahnstraße, die keine bequeme Interaktion oder Rückkopplung gestattet.

Als Beispiel betrachten wir den Automobilbau. Am Anfang eines neuen Modells stehen hier die Bleistiftskizzen des Designers. Aus diesen werden heute bereits teilweise automatisiert mittels perspektivischer Kalibrierung (mit Hilfe bekannter Maßzahlen wie Radstand etc.) erste (grobe) Drahtgitter-, Flächen- und Volumenmodelle erzeugt.

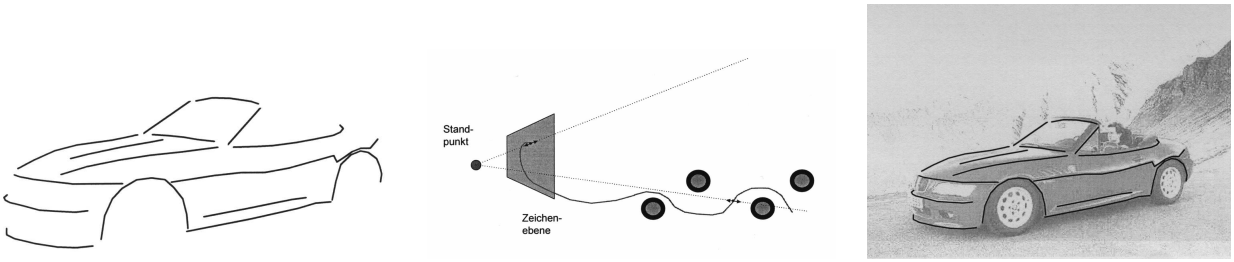


Abbildung 5.1: Designskizze (links), perspektivische Kalibrierung zur automatischen Generierung eines Grobmodells (Mitte), Projektion auf Foto des Autos (rechts)

Der Entwurf im Detail erfolgt dann im CAD, wobei hier die obigen groben Modelldaten i. d. R. nicht einfließen, sondern wieder von vorne begonnen wird. Als Resultat erhält man ein detailliertes Oberflächenmodell (B-Rep-Modell).

Dieses CAD-Modell wird nun i. A. wieder nicht weiterverwendet. Vielmehr erzeugen sich die Simulanten (abhängig von der jeweiligen Aufgabenstellung) neue Geometriemodelle:

- Kollisionstests: Passt die Tür in die Karosserie? Schließt der Kofferraumdeckel auch wirklich den Kofferraum? Hierfür werden i. A. *Voxel-Modelle* eingesetzt.
- Steifigkeit der Karosserie: Bei Crash-Tests werden die Stabilität und die Verformungseigenschaften des Chassis getestet. Die hierbei anfallenden strukturdynamischen Berechnungen erfolgen i. A. mittels Finite-Elemente-Netzen.
- Aerodynamik: Zur Bestimmung des Widerstandsbeiwerts c_w muss die Umströmung des Fahrzeugs im Windkanal simuliert werden, wozu – je nach Diskretisierung – Finite-Elemente-, Finite-Differenzen- oder Finite-Volumen-Netze eingesetzt werden.
- Belüftung: Zur optimalen Auslegung des Belüftungssystems (Düsen, Temperierung etc.) muss der Fahrzeuginnenraum durchströmt werden.
- Wieder andere Beschreibungen erfordern eventuell Kopplungen, bspw. Akustik-Struktur-Wechselwirkungen zur Simulation des Lärms im Innenraum.

Die Visualisierung erfolgt schließlich angepasst an bestimmte Simulationscodes (im Sinne eines Post-Processing) oder durch erneute Umwandlung der Datenformate. Weitergehende Visualisie-

rungstechniken (Virtual Reality (mit der Datenbrille durch den Fahrzeuginnenraum) oder Augmented Reality (Stromlinien in Realbilder (Fotos, Filme) integrieren) erfordern i. A. eine völlig neue Beschreibung.

Es ist offensichtlich, dass dieser Overhead lästig und einer computergestützten Arbeitsweise nicht angemessen ist. Deshalb sind einheitliche Darstellungsarten zu entwickeln, die als Grundlage für verschiedene Arbeitsschritte dienen können. Die Begehrlichkeiten gehen dabei noch wesentlich weiter: Lagerhaltung, Buchführung, Kostenrechnung etc. – all dies könnte auf einer gemeinsamen Basis aufsetzen.

Auf der Suche nach einem geeigneten integralen Datenmodell sind einige Überlegungen vorneweg hilfreich. Zunächst ist zu beachten, dass die „spannenden“ Gebiete, d. h. die Bereiche, in denen eine erhöhte Auflösung erforderlich ist, sich von Aufgabenstellung zu Aufgabenstellung unterscheiden. Für die geometrische Modellierung sind Bereiche mit viel Detailstruktur interessant, für die Simulation dagegen Bereiche mit heftiger physikalischer Aktivität. Das kann mit der Geometrie zu tun haben (bei der Umströmung des Außenspiegels etwa), muss aber nicht, wie das Beispiel eines eingespannten und belasteten Bretts zeigt: An der Sollbruchstelle tut sich geometrisch nichts, die Berechnung erfordert dort aber eine starke Verfeinerung des Gitters. Ferner muss stets hinterfragt werden, wieviel an geometrischer Information man für welche Aufgabe wirklich benötigt. Bei Finite-Elemente-Berechnungen etwa ist in der Regel nicht die Geometrie selbst, sondern die Berechnung gewisser Integrale über diese Geometrie verlangt. Schließlich soll unsere Geometriebeschreibung grobe Betrachtungen im Sinne eines Prototyping und auch Detailstudien gestatten.

Gesucht ist also eine einheitliche Darstellung, die – je nach Aufgabenstellung – flexibel verfeinert oder vergröbert werden kann, die hierarchisch ist und somit unterschiedliche Detailliertheitsstufen beeinhaltet, und die einfach verwaltet und in ihrer Dynamik gesteuert werden kann.

Hierfür bieten sich zunächst strukturierte Gitter an, also solche Netze, bei denen regelmäßige Elementformen die einfache Berechnung der Position von Punkten, der Größe bzw. Maschenweite von Elementen sowie der Nachbarschaftsrelationen gestatten. Den auf den ersten Blick als Nachteil erscheinenden schlechteren Approximationseigenschaften gekrümmter Ränder stehen eine Reihe von Vorteilen gegenüber:

- Geringer Aufwand für das Management der Struktur: Implementierungen auf Bit-Ebene via Bit-Operationen (Shifts etc.) sind möglich.
- Die Speichereffizienz gestattet u. U. sogar eine feinere Auflösung, als dies mit unstrukturierten Gittern möglich ist.
- Der Verfeinerungsprozess ist i. d. R. einfach.
- Man erhält leichten Zugang zu Interaktion/Feedback: Änderungen in der Geometrie sind leicht realisierbar.

Deshalb wollen wir uns *Quad-* bzw. *Octrees* als natürliche Verallgemeinerung von Binärbäumen in 2 D bzw. 3 D näher anschauen. Bei den Quadtrees startet man von einem Quadrat aus, das rekursiv in vier kongruente und disjunkte Teilquadrate zerlegt wird, falls es nicht ganz innerhalb oder ganz außerhalb des zu beschreibenden Gebiets liegt. Quadtrees zur Beschreibung einer Fläche unterscheiden demnach drei Fälle für ein Teilquadrat: drinnen, draußen oder teils teils. Im letzteren Fall wird weiter unterteilt – im Extremfall, bis eine vorgegebene Maximaltiefe erreicht wird. Soll mit dem Quadtree nur eine Kurve beschrieben werden, reichen sogar zwei Fälle: Entweder das Quadrat trifft die Kurve, oder es trifft sie nicht.

Octrees basieren analog auf der rekursiven Zerlegung eines Würfels in acht Subwürfel.

Quad- und Octrees sind in der Informatik weit verbreitet. Fast immer geht es dabei um die Organisation bzw. Strukturierung des Raums. Als Anwendungsfelder seien genannt:

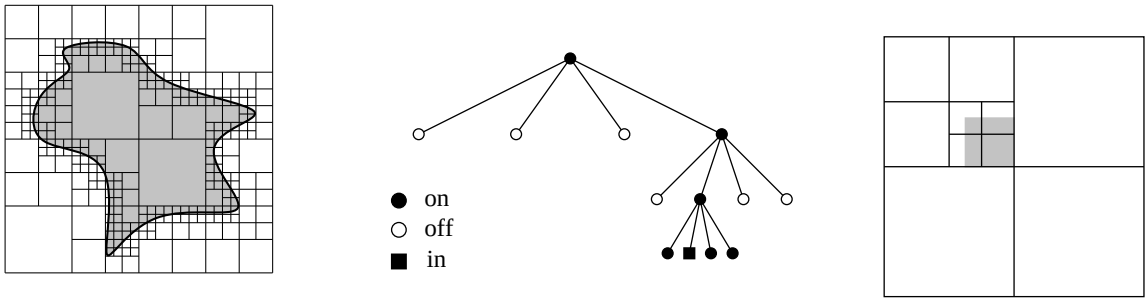


Abbildung 5.2: Quadtrees

- Datenbanken (multidimensionale Indizes, Datensätze als Punkte im mehrdimensionalen Raum: Sequentialisierung zur effizienten Beantwortung von Suchanfragen)
- Bildverarbeitung (Segmentierung)
- geometrische Modellierung: Voxel als Volumen- und Oberflächenmodell
- Computergraphik: Organisation bzw. Partitionierung des Raums (effizientere Lösungen für den Sichtbarkeitsentscheid oder die Schnittproblematik beim Ray Tracing: trifft ein Strahl den großen Würfel nicht, dann auch keines der Objekte in ihm)
- Netzgenerierung (Pre-Processing der Simulation, Generierung von (auch unstrukturierten) Netzen mit Hilfe von Oktalbäumen)
- Visualisierung (Isoflächen, Post-Processing der Simulation)
- digital mock-up: integrierter Entwurfs- und Konstruktionsprozess

Als wichtige Vorteile von Quad- und Octrees halten wir fest:

- Dimensionsreduktion in der Komplexität (Oberfläche bestimmt Aufwand); Problemfälle: fraktale Strukturen etc.
- effiziente Realisierung der Mengenoperationen (linear in der Zeit)
- schnelle Algorithmen zur Suche von Nachbarn verfügbar
- inhärente Sequentialisierungsinformation (Quadtrees und Octrees sind Iterationen der raumfüllenden Kurve nach Lebesgue, vgl. Abschnitt 4.3), die eine effiziente Lastverteilung bzw. Parallelisierung ermöglicht

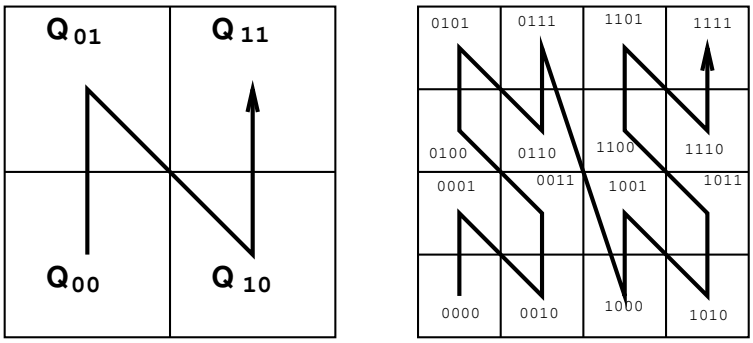


Abbildung 5.3: Die raumfüllende Kurve nach Lebesgue: Quadtree-Aufzählung

- Implementierung: Geflechte, Linearisierung (preorder traversal), locational codes (Volumendarstellung: 2 Bits (drinnen/draußen/verfeinern), Oberflächendarstellung: 1 Bit (trifft/trifft nicht))

Wie können wir nun Oktalbäume als einheitliche Datenstruktur für Modellierung, Simulation und Visualisierung einsetzen? Dazu definieren wir eine *hybride Datenstruktur*: Eine *Basisstruktur* trägt die geometrische Information in voller Tiefe, eine (in ihr enthaltene) *Simulationsstruktur* dient zur numerischen Berechnung. Erstere ist realisiert als ein Octree extrem feiner Auflösung:

- Er trägt die volle geometrische Information.
- Er ist realisiert als Bit-kodierter linearisierter Baum (depth first).
- Er kann ggfs. auf den Hintergrundspeicher ausgelagert werden (um wirklich die volle Geometrie ohne Vereinfachungen aufzunehmen).
- Die Modellierung folgt einem strengen Baukastenprinzip: Für jedes Basis-Bauteil wird ein Octree angelegt. Kompliziertere Objekte (ein Auto bspw.) werden just-in-time modelliert (ohne Speicherung der Zwischenstufen). Diese Vorgehensweise bietet einige Vorteile:
  - Die Modellierung kann via UNIX-Pipes mit Zugriff auf Hintergrundspeicher und Kellerung der Zwischenresultate erfolgen.
  - Involviert sind maximal drei Module bzw. Octrees, kein Objekt muss vollständig geladen werden.
  - Mengen- bzw. Boolesche bzw. CSG Operationen sind leicht zu realisieren.

Die Simulationsstruktur besteht aus einem (kleinen) Ausschnitt variabler Größe, wird explizit im Hauptspeicher aufgebaut und etwa als Geflecht realisiert. Im Gegensatz zur Basisstruktur, die ausschließlich geometrische Information trägt, ist sie mit den Variablen des jeweiligen Simulationsproblems (z. B. Geschwindigkeiten und Druck im Falle einer Strömungssimulation) versehen.

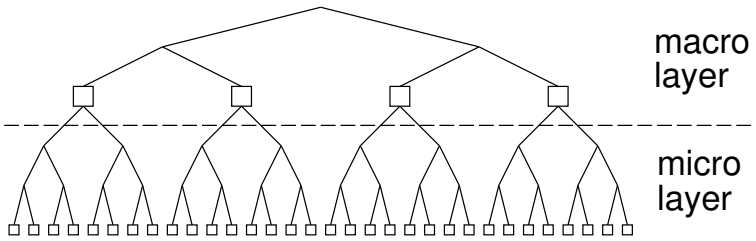


Abbildung 5.4: Macro- und Microlevel der hybriden Octree-Datenstruktur

Die Vorgehensweise erinnert an Techniken aus der Computergraphik:

- Subsampling zur Vermeidung von Aliasing-Effekten (Pixel werden in Sub-Pixel unterteilt, um Verfremdungseffekte an harten Grenzen zu vermeiden)
- Dithering zur Erzeugung von Graustufen (Bild-Pixel werden in Sub-Pixel unterteilt, die – je nach gewünschter Graustufe – schwarz oder weiß eingefärbt werden)

Wie hat man sich nun eine Simulation auf dieser hybriden Struktur vorzustellen?

- Zu Beginn muss die Information in den Blättern der Simulationsstruktur per Durchlauf durch die Basisstruktur (kellern, nur Ausschnitt laden) bereitgestellt werden. Dies kann bedeuten
  - die Akkumulation lokaler FE-Integrale
  - die Akkumulation lokaler Diskretisierungsmatrizen im Sinne der Nested Dissection bzw. rekursiven Substrukturierung

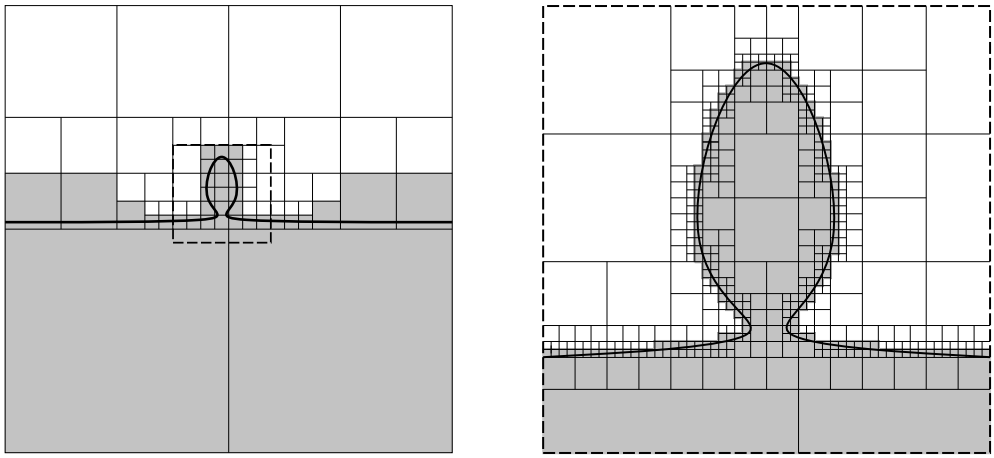


Abbildung 5.5: Macro- und Microlevel im Beispiel

- Die anschließende Iteration erfolgt dann rein auf der Simulationsstruktur.
- Bei erforderlicher adaptiver Verfeinerung muss erneut auf die Basisstruktur zugegriffen werden. Werden zusätzlich die den Blättern der Makroebene entsprechenden Indexpositionen in der Basisstruktur gespeichert, muss jetzt nur der entsprechende Teilbaum eingelesen werden.

Somit ist die einfache Ankopplung von Modellierung und Simulation (zumindest im Prinzip) geklärt. Ebenso spannend ist die unmittelbare Anknüpfung der Visualisierung an die Simulation. Hier hat man es mit der Problematik zu tun, dass viele Visualisierungsprogramme auf regulären Strukturen und mit Stützpunktdaten arbeiten. Als Konsequenz ergibt sich, dass vor der Visualisierung hierarchische Simulationsdaten auf adaptiven Gittern auf ein reguläres Gitter interpoliert werden. Wählt man dieses Gitter zu grob, gehen Details verloren, wählt man es zu fein, ist der Effizienzgewinn der Adaptivität dahin, und man steht vor Speicherproblemen. Ziel muss deshalb eine direkte Visualisierung adaptiver hierarchischer Daten sein.

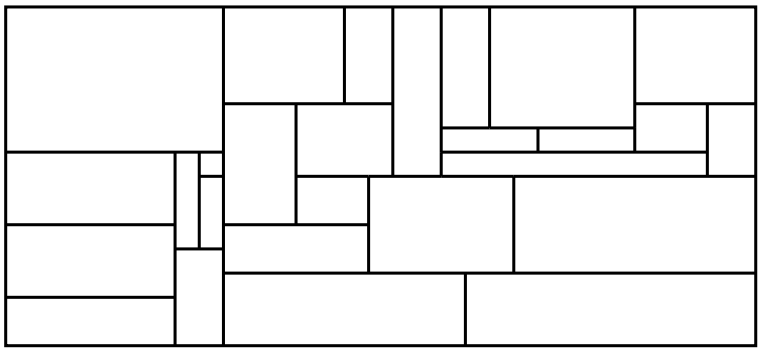


Abbildung 5.6: Adaptives, hierarchisches, strukturiertes Gitter

Dies ist möglich. Z. B. wurde für das Visualisierungssystem AVS/Express eine allg. Visualisierungsschnittstelle für adaptive, hierarchische, strukturierte Geometriedaten entwickelt, die die unmittelbare Visualisierung solcher Datensätze ohne Umweg über eine reguläre dehierarchisierte Struktur gestattet. Dieses Interface kann dabei mehr, als Daten auf Oktalbäumen zu visualisieren: behandelt werden können allg. Daten auf rechtwinkligen Gittern mit rationalen Skalierungsfaktoren in

allen Richtungen (siehe Abb. 5.6), worunter auch kd-Bäume[1] oder dünne Gitter[2] fallen.

Somit kann die Hierarchie der Daten direkt für die Visualisierung ausgenutzt werden.

5.2 Gekoppelte Systeme

Zunächst wollen wir uns klarmachen, warum Wechselwirkungen oder Kopplungen in der Simulation eine immer größere Rolle spielen. Ausgehend von der Tatsache, dass Simulationen nichts anderes als Experimente an Modellen und diese ihrerseits nichts anderes als vereinfachte Abbilder realer Systeme sind, kann man folgende Beobachtungen machen:

- Der Komplexitätsgrad von Systemen und Modellen nimmt zu.
- Die Genauigkeitsanforderungen, die an die Simulation gestellt werden, wachsen ebenfalls.
- Eine einzelne Simulation erfordert deshalb immer mehr an Expertenwissen.

Als Konsequenzen ergeben sich hieraus:

- Die einzelnen Systemkomponenten werden ihrerseits zu Subsystemen.
- Eine hierarchische Strukturierung wird erforderlich.
- Die Wechselwirkungen von Komponenten und Subsystemen gewinnen an Bedeutung.
- Die gestiegene Komplexität beendet die Ära der „One-man-Modelle“. Deshalb muss Expertenwissen so weit wie möglich ausgelagert werden.
- Die Berücksichtigung von Kopplungen wird eine zentrale Aufgabe bei der Simulation. Man unterscheidet dabei
  - *vertikale* Kopplungen, die Wechselwirkungen zwischen unterschiedlichen Modellierungsebenen (Komponentenebene, Systemebene) betreffen, und
  - *horizontale* Kopplungen, die unterschiedliche Effekte auf derselben Modellierungsebene betreffen (etwa Fluid-Struktur-Wechselwirkungen auf der Ebene der Kontinuumsmechanik).

In Abb. 5.7 sind Beispiele horizontaler Kopplungen in einer Matrix zusammengestellt (im linken unteren Dreieck der Typ der Wechselwirkung, im rechten oberen Dreieck technische Bauteile oder Anwendungen, bei denen sie auftreten). Für solche Probleme kommen wir mit unseren bisherigen Modellen und Codes nicht aus: Ein Strömungslöser fängt mit Strukturmechanik nichts an, im Modell, das einem Programm zur Strukturmechanik zugrunde liegt, ist Elektrostatik nicht vorgesehen usw. Abbildung 5.8 zeigt eine Simulationshierarchie mit unterschiedlichen Modellierungsebenen am Beispiel des Menschen. Vertikale Kopplungen wären etwa Einflüsse, die die detailliertere Schicht auf die Modellierung der nächst gröberen Ebene hat. Hier ist das Problem, dass man sich vielleicht im selben thematischen Umfeld bewegt (z. B. CFD), aber mit ganz unterschiedlichen Modellierungsmitteln arbeitet (PDEs vs. ODEs). Die Frage stellt sich somit, wie eine Modellierung auf einer feineren Ebene (z. B. mit PDEs, raumaufgelöst) dazu benutzt werden kann, ein gröberes Modell (z. B. mit ODEs, ohne Raumauflösung) zu verbesseren (bessere Kennlinien etc.).

Die ersten Versuche zu numerischen Simulationen (horizontal) gekoppelter Probleme erfolgten im Bereich der Fluid-Struktur- bzw. Akustik-Struktur-Wechselwirkungen, etwa zur Untersuchung der Erdbebensicherheit von Bauwerken oder der Stabilität von Staudämmen. Ein spektakuläres

[1] kd-Bäume dienen der Partitionierung des d-dimensionalen Raumes. Abwechselnd wird dabei der Raum in x_i-Richtung, $i = 1, \ldots, d$, so in zwei Teile zerlegt, dass ein bestimmtes Balancierungskriterium erfüllt ist (z. B. gleich viele Punkte auf beiden Seiten).

[2] Dünne Gitter basieren auf hierarchischen Tensorproduktzerlegungen des Raumes, wobei überlappende Elemente mit sehr unterschiedlichen Aspektverhältnissen auftreten.

components / kind of coupl.	structural dynamics	heat conduction	fluid dynamics	electromagnetics
structural dynamics		– rolling processes – sensors/actuators (resonators)	– turbines – microvalves, micropumps – sound absorption – accelerometers	– motors, generators – electrostatic and piezoelectric sensors and actuators
heat conduction	thermoelasticity h–>s: material law s–>h: energy dissipation		– cooler – flow– and thermal sensors or actuators	– thermal switches – thermal actuators
fluid dynamics	f–>s: surface forces s–>f: – geometry – bound. cond.	coupling via the energy equation		– sensors and pumps working with the electrohydrodynamic principle
electro– magnetics	e–>s: surface and volume forces s–>e: – geometry – moving conductor	e–>h: electromagnetic losses h–>e: material properties	electrohydrodynamics e–>f: surface and volume forces f–>e: – geometry – mv. conductor	

Abbildung 5.7: Kopplungsmatrix: bilaterale Wechselwirkungen und Anwendungsbeispiele

Beispiel einer Fluid-Struktur-Wechselwirkung lieferte der Einsturz der Tacoma Bridge im US-Bundesstaat Washington 1940 aufgrund einer sogenannten Resonanzkatastrophe. Besonders große Bedeutung haben Wechselwirkungen heute jedoch in der Mikrosystemtechnologie. Hierfür sind vor allem drei Gründe verantwortlich:

- Die Größenverhältnisse führen zur Dominanz von Oberflächeneffekten über Volumeneffekte (bei abnehmender Maschenweite h wird h^3 schneller klein als h^2). Somit werden die oft über Grenzflächen wirksamen Kopplungen wichtiger.
- In Sensoren und Aktuatoren, den bevorzugten Bauteilen der Mikrosystemtechnik, werden gezielt mehrere physikalische Effekte ausgenutzt. Als Beispiel seien Beschleunigungssensoren in Airbagsystemen genannt (siehe Abb. 5.9), in denen eine Metallzunge flüssigkeitsgedämpft im elektrischen Feld zwischen zwei Kondensatorplatten schwingt und somit die Kapazität beeinflusst. Es handelt sich hierbei also um eine Elektrostatik-Fluid-Struktur-Kopplung.
- Die hohe Integrationsdichte führt neben den beabsichtigten Wechselwirkungen zu zusätzlichen ungewollten oder *parasitären* Effekten (cross effects). Man denke hier etwa an den Bereich der elektromagnetischen Verträglichkeit, bspw. bei der wechselseitigen Beeinflussung der Signale auf zwei sehr nah beieinander verlaufenden Leiterbahnen.

Ein weiteres Beispiel eines Mikrosystems, das auf Kopplungen basiert, ist die in Abb. 5.10 schematisch dargestellte ventilgesteuerte Mikropumpe. Abbildung 5.11 zeigt die Pumpe dreidimensional als Explosionsmodell. Die Abmessungen betragen in jeder Dimension nur wenige Millimeter. Eine elektrisch zur Schwingung angeregte Membran an der Oberseite der Pumpkammer erzeugt die Pumpwirkung, durch die das Einlass- und das Auslassventil abwechselnd geöffnet bzw. geschlossen werden. Einsatzgebiete der (industriell gefertigten) Mikropumpe sind medizinische Implantate (etwa zur automatisch geregelten Zufuhr von Insulin bei Diabetikern) oder mobile Umweltanalysesysteme.

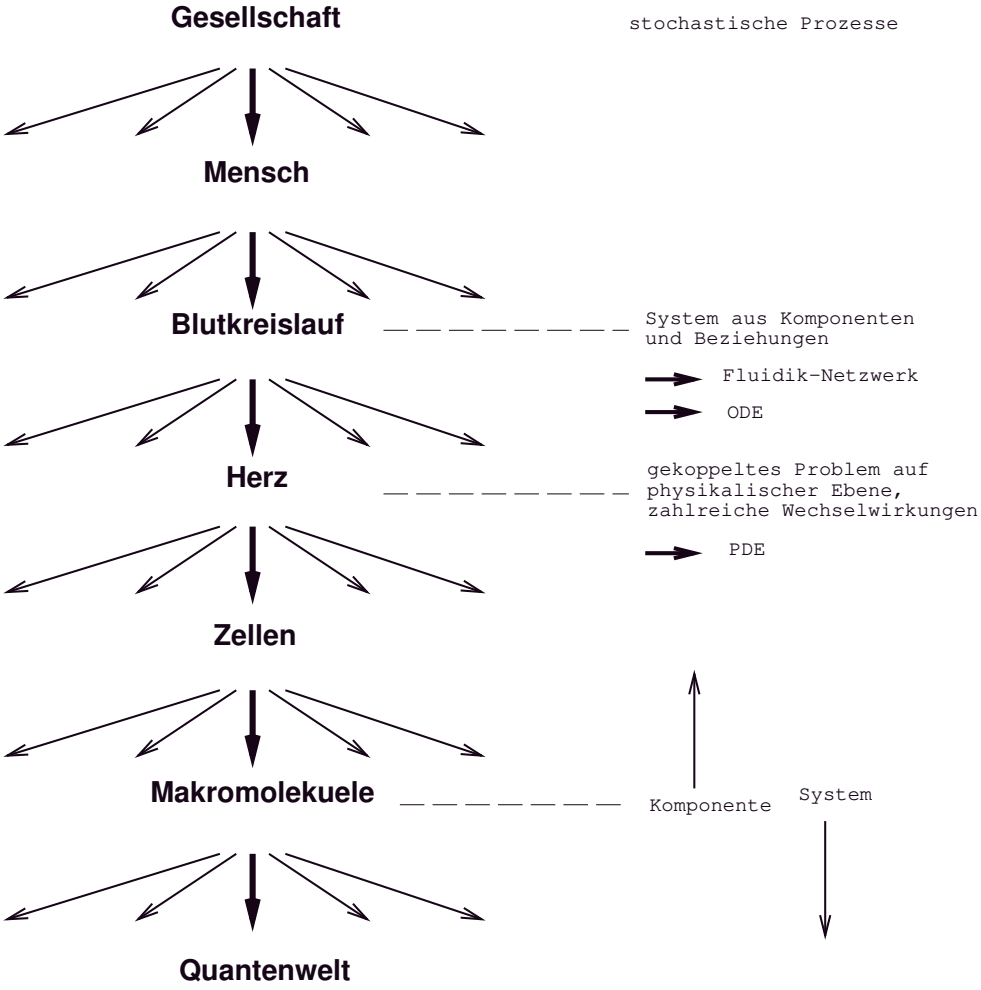


Abbildung 5.8: Beispiel einer Simulationshierarchie: Simulation rund um den Menschen

Zienkiewicz gab 1984 eine formale Definition eines gekoppelten Problems an. Demnach sind gekoppelte Probleme Probleme mit bidirektionalen Wechselwirkungen von i. d. R. unterschiedlichen physikalischen Effekten, wobei

- kein Teilproblem auf seinem Gebiet unabhängig gelöst werden kann und
- kein Satz von Variablen explizit eliminiert werden kann.

In einer allgemeinen Notation betrachten wir demnach n Teilprobleme mit Kopplungen, definiert durch

$$\begin{aligned} L_i[\pi_i, \kappa_i](u_i) &= 0 \quad \text{auf} \quad \Omega_i\,, \qquad 1 \le i \le n\,, \\ C_j(u_1, \ldots, u_n, \kappa_1 \ldots, \kappa_n) &= 0 \quad \text{auf} \quad \Gamma_j\,, \qquad 1 \le j \le m\,, \end{aligned}$$

wobei L_i den Operator für Teilproblem i (TP_i) bezeichnet, C_j den koppelnden Operator j, u_i den Variablensatz von TP_i, π_i nicht koppelnde Parameter von TP_i und κ_i dessen koppelnde Parameter. Nicht koppelnde Parameter sind etwa Materialparameter (relevant für TP_i, aber nicht für andere Teilprobleme), ein Beispiel für einen koppelnden Parameter ist etwa der Druck im Fluid bei einer Fluid-Struktur-Wechselwirkung, der für die Kraft auf die bewegliche Struktur und somit für deren Auslenkung verantwortlich ist. Aber auch die Geometrie selbst kann Kopplungsparameter sein, wenn etwa bei der Mikropumpe durch den Druck des Fluids die Ventilklappen auf- oder zugedrückt

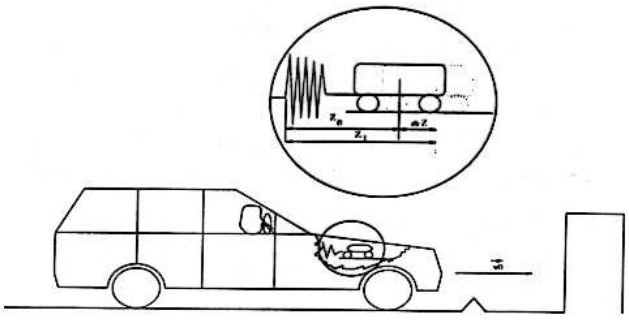


Abbildung 1.2: Prinzip eines Beschleunigungssensors

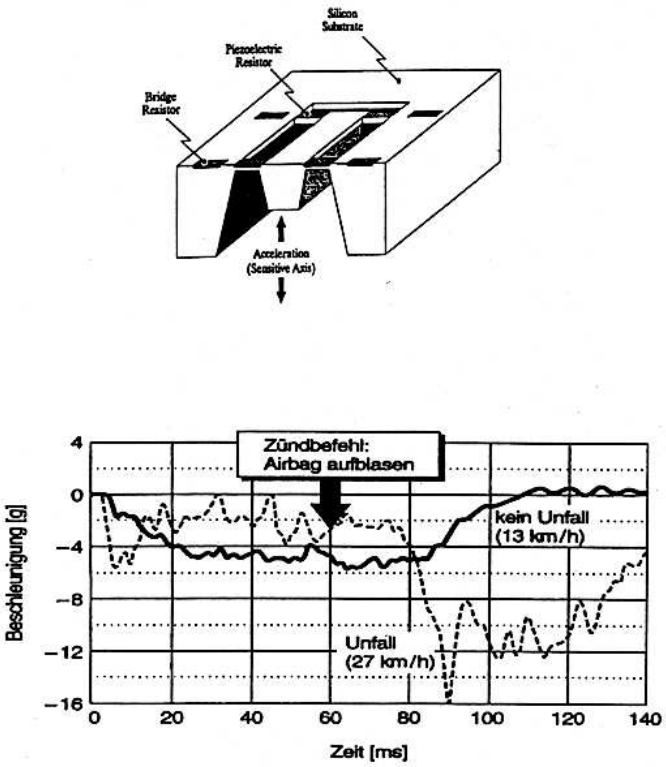


Abbildung 5.9: Beschleunigungssensor in Airbagsystemen: Prinzip (oben), Aufbau (Mitte) und Verlauf der Beschleunigung bei einem Unfall und bei lediglich starkem Bremsen (unten). Man beachte, dass bis über den Zeitpunkt einer potentiellen Zündung hinaus die Verzögerungswirkung beim Nicht-Unfall stärker als beim Unfall ist, was das Erkennen von Crashs und damit Zündsituationen erschwert.

werden. Das gesamte Wechselwirkungsgeflecht kann man auch als Graphen hinschreiben (siehe Abb. 5.12).

Ein nächster wichtiger Punkt ist die Klassifizierung gekoppelter Probleme. Sie kann erfolgen

- anhand der Art des Interface-Gebiets (Zienkiewicz 1984):

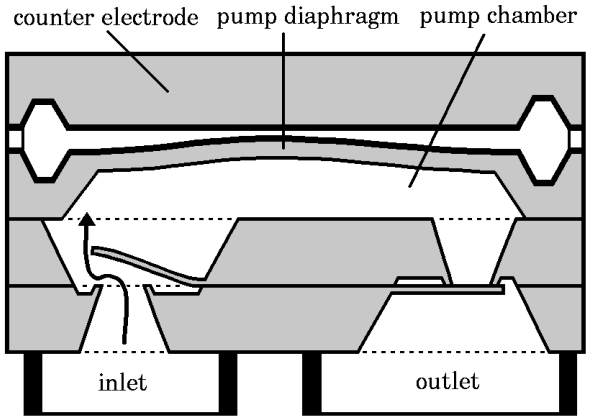


Abbildung 5.10: Schematischer Aufbau einer ventilgesteuerten Mikropumpe mit Pumpkammer, Membran, Ein- und Auslassventilklappe sowie Zu- und Abfluss

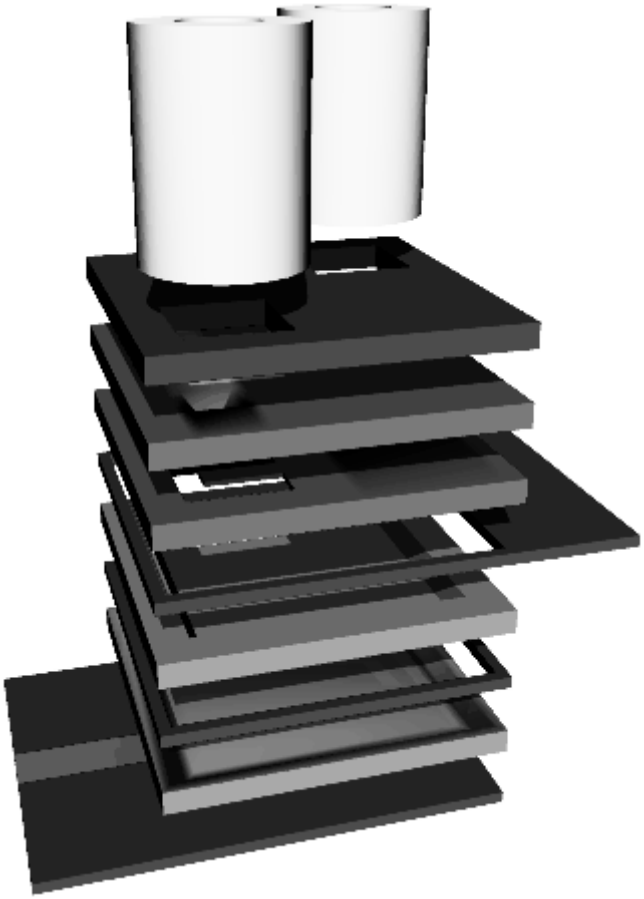


Abbildung 5.11: Die ventilgesteuerte Mikropumpe im Explosionsmodell: Die beiden Röhren stellen die Einlass- und Auslassverbindungen dar, in der Mitte sind die Ventilklappen zu erkennen, unten links befinden sich die Kontakte für die Stromversorgung der Membran.

- überlappende Teilproblemgebiete
- nicht überlappende Teilproblemgebiete mit gleichen oder verschiedenen physikalischen

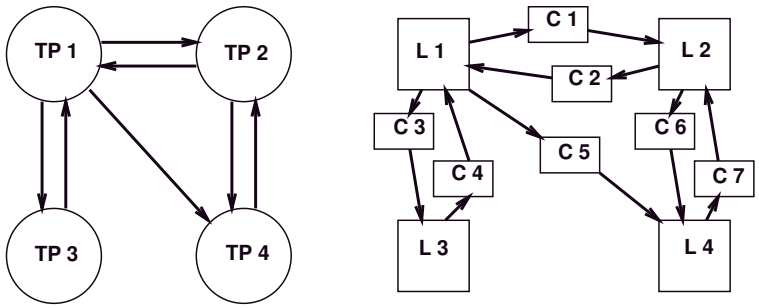


Abbildung 5.12: Kopplungsgraph: Einzeloperatoren L und Kopplungsoperatoren C

Modellen

- anhand der relativen Stellung der Teilprobleme in einem hierarchischen Modell:
  - Kopplungen innerhalb einer Modellierungsebene (horizontal)
  - Kopplungen zwischen verschiedenen Ebenen (vertikal)
- anhand der Komplexität des gekoppelten Problems (Eigenschaften des Koppelgraphen: Zyklen, ...)
- anhand der Eigenschaften der koppelnden Bedingungen:
  Auflösbarkeit, mathematischer Typ (AE, ODE, PDE, IE, ...)
- anhand der Sensitivität der Teilproblemlösung gg. Änderungen in den koppelnden Parametern
  - starke / schwache Kopplungen
  - hoch- / niederfrequente Kopplungen

Was für Lösungsansätze haben sich nun für solche gekoppelten Probleme etabliert? Hier gibt es wiederum mehrere Unterscheidungsmöglichkeiten:

- vom Standpunkt der Modellbildung aus:
  - *gemeinsame* Lösung (*simultaneous* solution): Kombination der kontinuierlichen Modelle mit anschließender Behandlung des kontinuierlichen Gesamtsystems oder Kombination der diskretisierten Modelle mit anschließender Behandlung des diskreten Gesamtsystems
  - *partitionierte* Lösung *partitioned* solution): Trennung der Lösung der einzelnen Teilprobleme von der Berücksichtigung der eigentlichen Kopplungen, Iteration über das Lösen der Einzelprobleme
- vom Standpunkt der Code-Entwicklung aus:
  - *Erweiterung* vorhandener Programme:
    ein CFD-Löser erhält Transportkomponente (FASTEST); ein Strukturlöser erhält eine Strömungskomponente (ADINA-F, ANSYS-FLOTRAN) oder eine Elektrostatik-Komponente (ANSYS)
  - Entwicklung neuer (problemspezifischer bzw. *maßgeschneiderter*) Modelle und Programme
  - Entwicklung einer *Kopplungsumgebung* zum modularen gekoppelten Einsatz vorhandener Programme
- hinsichtlich der Zielvorstellung:

  - *problemorientierter* Ansatz: Lösung einer konkret vorgegebenen Aufgabenstellung
  - *methodischer* Ansatz: Entwicklung eines (möglichst allgemeinen) Verfahrens
- abhängig von der Standeszugehörigkeit:
  - *mathematischer* Ansatz: exakte Modellbildung, effiziente Numerik, beliebige Entwicklungszeit
  - *ingenieurwissenschaftlicher* Ansatz: exakte Modellbildung, Entwicklungszeit = ein Dr.-Ing.
  - *softwaretechnischer* Ansatz: zumindest in der Theorie hohe Wiederverwertbarkeit und geringe Entwicklungszeit, oft bescheidene Effizienz
  - *industrieller* Ansatz: Entwicklungszeit „bis Ende der Woche“, oft fragwürdige Genauigkeit und Effizienz

Wir denken wieder an unsere Leitsätze aus der Einleitung und propagieren natürlich einen modularen oder partitionierten Ansatz, bei dem existierende Teilproblemlöser (Codes) über effiziente Schnittstellen und eine robuste äußere Iteration (für die eigentliche Kopplung) zu einem Löser des Gesamtproblems zusammengeschaltet werden. Das Prinzip der partitionierten Lösung zeigt Abb. 5.13. Somit ergibt sich folgendes Grundmuster einer Iteration:

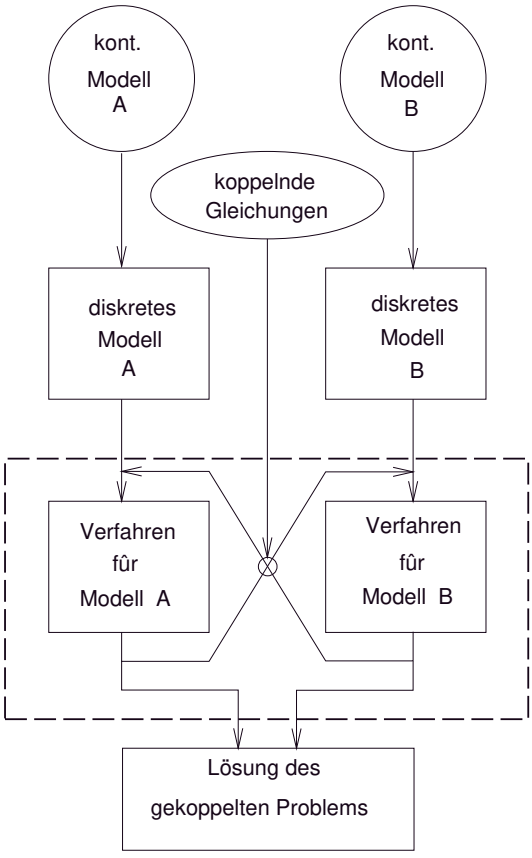


Abbildung 5.13: Prinzip der partitionierten Lösung

1: identifiziere π_i und κ_i für alle Teilprobleme;
π_i: Startwerte gemäß Teilproblem i
κ_i: wähle Startwerte (z. B. entkoppelter Zustand)

2: separate Lösung aller Teilprobleme

3: Auswertung der koppelnden Bedingungen:
$C_j(\dots) = 0 \; \forall j$

4: solange die koppelnden Bedingungen nicht mit der gewünschten Genauigkeit erfüllt werden:
– Korrektur der κ_i
– Teilproblemlösung
– Überprüfung der koppelnden Bedingungen

Als Vorteile des modularen Ansatzes halten wir fest:

- Verfügbare (Standard-) Codes können eingesetzt werden, der Einarbeitungsaufwand auch in eine kompliziertere Koppelproblematik ist vergleichsweise gering.
- Einzelne Komponenten (Teilproblemlöser) können bequem verglichen und ausgetauscht werden.
- Neu zu erstellen sind lediglich Programme zur Kopplungs- und Ablaufsteuerung, die Modellierung der Teilprobleme wird dagegen wiederverwendet.
- Es ergibt sich eine größere Flexibilität: Für andere Probleme werden die einzelnen Komponenten einfach entsprechend neu zusammengeschaltet.
- Die Modularität des Ansatzes beinhaltet einen grobkörnigen Parallelismus. Der Einsatz knotengebundener Programme ist möglich.
- Schließlich wird auch mit menschlichen Ressourcen sparsam umgegangen: Strömungsprobleme werden vom Strömungsfachmann gelöst, der „Koppler“ kümmert sich nur um sein übergeordnetes gekoppeltes Problem.

Analogien finden sich mannigfach. Einige Beispiele:

- numerische Lösung von Gleichungssystemen (Jacobi-Iteration, Nested Dissection)
- Gebietszerlegungsverfahren, rekursive Substrukturierung
- allg. divide & conquer
- Informatik: Software-Engineering

Der heikle Punkt ist natürlich die Realisierung der äußeren Iteration. Obwohl die Kopplungen in aller Regel nichtlinear sind und man folglich auch ein nichtlineares Iterationsverfahren (z. B. Newton-ähnlich) bräuchte, greift man dennoch derzeit noch meistens auf ein Gauß-Seidel- bzw. SOR-ähnliches Vorgehen zurück: Löse Teilproblem 1, passe die Randbedingungen etc. für Teilproblem 2 daran an und löse Teilproblem 2, modifiziere die Randbedingungen von Teilproblem 1 entsprechend und löse dieses wieder usf. Darüber hinaus kann hier aber auch das Spektrum der linearen und nichtlinearen Iterationsverfahren der Numerik eingesetzt werden. Hier gilt

- erster Lösungsansatz:
  verallgemeinerte lineare Iterationsverfahren
  (Gauß-Seidel, SOR)
  - Vorteile:
    + leicht zu implementieren
    + wenn konvergent, dann i. a. schnell
  - Nachteile:
    – Konvergenz nicht gesichert
    – Flattern / Oszillationen
  - Abhilfe:
    Relaxationsparameter, die jedoch nicht allg. zur Verfügung stehen

- zweiter Lösungsansatz:
  nichtlineare Iterationsverfahren
  (Newton-ähnliche, Fixpunktiterationen, ...)
  - Vorteile:
    + allgemeiner anwendbar
    + robuster
  - Nachteile:
    - aufwendig
    - Berechnung von Ableitungen

Dass man aber auch im linearen Fall nicht auf Gauß-Seidel-Derivate beschränkt ist, zeigt das folgende Beispiel. Gegeben sei ein Kragbalken (ein einseitig eingespannter Balken), der an seinem freien Ende belastet werde. Wir wählen einen Gebietszerlegungsansatz mit zwei Teilgebieten und fassen das Ganze als Struktur-Struktur-Kopplung auf (siehe Abb. 5.14). Für die weitere mathema-

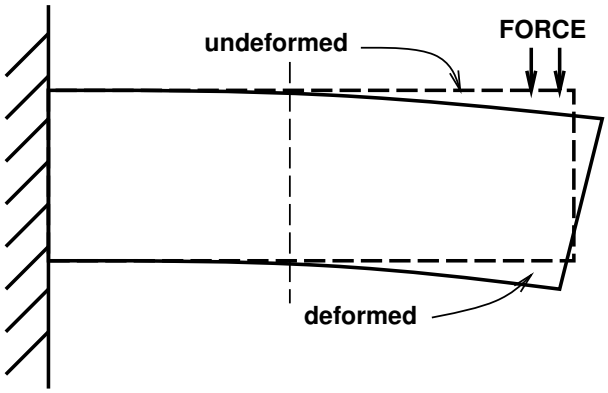


Abbildung 5.14: Kragbalken unter Last mit seinen zwei Teilgebieten

tische Behandlung gehen wir von drei Teilgebieten aus: den inneren Variablen x_1 in Ω_1, den inneren Variablen x_2 in Ω_2 und den Knoten x_3 auf der Trennfläche (Interface) Ω_3. Die Diskretisierung der entsprechenden Gleichungen der Statik bzw. Elastizitätstheorie mit einem Finite-Element-Ansatz führt auf ein blockstrukturiertes lineares Gleichungssystem mit Steifigkeitsmatrix A und Lastvektor b, wobei die Unbekannten x_i gerade die Verschiebungen in Ω_i angeben:

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & (A_{33})_1 + (A_{33})_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} .$$

Bei Vorgabe des Verschiebungsvektors x_3 auf dem Interface Ω_3 lassen sich die beiden Teilprobleme im Inneren lösen:

$$\begin{aligned} x_1 &:= A_{11}^{-1} (b_1 - A_{13} x_3) , \\ x_2 &:= A_{22}^{-1} (b_2 - A_{23} x_3) . \end{aligned}$$

Der wesentliche Rechenaufwand entsteht dabei durch die Invertierung der Matrizen A_{11} und A_{22}, d. h. durch die Lösung der Teilprobleme, wobei die jeweilige Belastung aus einer Kombination aus externen Lasten (repräsentiert durch b_i) und aus der Vorgabe der Verschiebung x_3 am Interface resultiert. Die Reaktionskräfte auf dem Interface ergeben sich nun unter Verwendung der Lösungen x_1 und x_2 sowie der vorgegebenen Verschiebung x_3 zu

$$\begin{aligned} f_1 &:= (b_3)_1 - A_{31} x_1 - (A_{33})_1 x_3 , \\ f_2 &:= (b_3)_2 - A_{32} x_2 - (A_{33})_2 x_3 . \end{aligned}$$

Die gesuchte Lösung des Gesamtproblems ist dadurch gekennzeichnet, dass sich die Reaktionskräfte f_1 und f_2 zu Null addieren müssen, um die Forderung nach einem Kräftegleichgewicht zwischen linkem und rechtem Schnittufer zu erfüllen. Die Summe der Reaktionskräfte wird naheliegenderweise auch Residuum genannt. Formal lässt sich das Residuum also als eine Funktion der angenommenen Verschiebungen x_3 auf dem Interface auffassen:

$$r(x_3) \;:=\; f_1(x_3) + f_2(x_3) \;=:\; Kx_3 - c$$

mit

$$K \;:=\; -\left(A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}\right)$$

und

$$c \;:=\; A_{31}A_{11}^{-1}b_1 + A_{32}A_{22}^{-1}b_2 - b_3\,.$$

Die Matrix K, deren Größe durch die Anzahl der Freiheitsgrade auf dem Interface Ω_3 bestimmt ist, wird dabei im Kontext der Gebietszerlegung *Schurkomplement* genannt. K kann aus den Blöcken A_{ij} berechnet werden und ist symmetrisch positiv definit.

Die gesuchte Lösung x_3 wird nun aus der Forderung

$$r(x_3) \;=\; 0$$

ermittelt, indem ein geeignetes Iterationsverfahren angewandt wird. Damit ist die Kopplung der Teilprobleme 1 und 2 auf ein Interface-Problem für x_3 reduziert.

Die Standard-Schurkomplementmethode aus der Gebietszerlegungswelt stellt K explizit auf und löst anschließend das lineare Gleichungssystem $r(x_3) = 0$. Diese Vorgehensweise ist jedoch oft zu teuer, so dass man in der Regel Iterationsverfahren vorzieht, die mit der *Anwendung* von K auskommen, d. h. dem Matrix-Vektor-Produkt $K \cdot x$ (wozu man, wie man leicht sieht, K nicht explizit kennen muss: es genügt die Hintereinanderausführung von Lösungen bzw. Anwendungen von Blöcken A_{ij}).

Wir gehen noch einen Schritt weiter: Als Verfechter der modularen Strategie wollen wir das Interface-Problem lösen und dabei aber auf K überhaupt nicht mehr zugreifen müssen, sondern nur noch die bei den Teilproblemen verfügbaren Informationen (also A_{11}, A_{13}, A_{31} und $(A_{33})_1$ für Teilproblem 1 und entsprechend für Teilproblem 2) verwenden. Das versuchen wir mit dem cg-Verfahren. Dieses Verfahren benötigt die Matrix K nur, um sie auf die konjugierten Richtungen $d^{(i)}$ anzuwenden. Wegen

$$K \cdot d^{(i)} \;=\; r(d^{(i)}) + c$$

und wegen

$$c = -r(0)$$

kann diese Anwendung der Schurkomplementmatrix K rein durch Bestimmung von Residuen $r(v)$ erfolgen, was wiederum nach der obigen Herleitung allein die Lösung der Teilprobleme erfordert. Damit haben wir unser Ziel erreicht: eine strikt modulare Lösung unseres (zugegebenermaßen sehr einfachen) gekoppelten Problems. Den Clou führe man sich nochmals vor Augen: Durch die Entkopplung der Teilprobleme spricht jetzt nichts mehr dagegen, auf Ω_1 und Ω_2 unterschiedliche Codes einzusetzen.

Der Einsatz der partitionierten Lösung und insb. allgemeiner Iterationsverfahren für kompliziertere gekoppelte Probleme (nichtlinear, unterschiedliche Operatoren) ist natürlich weit problematischer. Für das Beispiel der Mikropumpe wurde er jedoch bereits erfolgreich eingesetzt.

Kapitel 6

Visualization

Nach einer Simulationsrechnung, also am Ende der numerischen Lösung eines mathematischen Modells, liegt das Ergebnis als (i. a. sehr großer) Datensatz vor: die Temperaturverteilung in einem Reaktor, das Geschwindigkeitsfeld um ein Auto im Windkanal oder die Geschwindigkeiten, Beschleunigungen und Kräfte eines Autos beim Elchtest. Wenn auch einige der zentralen Fragestellungen (maximale Temperatur im Reaktor, erliegt das Auto dem A-Klassen-Syndrom?) oftmals durch bloße Bearbeitung der Ergebnisdaten geklärt werden können, so ist doch deren Aussagekraft eher gering. Hier kann die *Visualisierung* helfen, die visuelle bzw. optische Darstellung der Ergebnisse. Deren Anwendungsbereich geht weit über das wissenschaftliche Rechnen hinaus. Der Gewinn liegt dabei in der Tatsache begründet, dass erstaunlich große Anteile des menschlichen Gehirns dem Sehvorgang und der natürlichen Bildverarbeitung gewidmet sind und dass die Kapazität oder Bandbreite der Informationsverarbeitung unseres visuellen Systems größer als bei allen anderen Sinnen ist. Dadurch können Bilder mit (noch wahrnehmbarer) Information geradezu vollgepackt werden: Sie bieten einen globale Ansicht des Ganzen und erlauben zudem die Kodierung lokaler Phänomene via Größe, Formen, Farben, Musterung, Helligkeit etc.

Um den gesamten Prozess der natürlichen Bildverarbeitung zu verstehen und damit Visualisierungen optimal gestalten zu können, sind detaillierte Kenntnisse aus Physik, Medizin und Psychologie erforderlich. So weit wollen wir hier nicht gehen, sondern uns auf den technischen Teil der Bilderzeugung konzentrieren. Je nach konkreter Aufgabenstellung sind dabei ganz unterschiedliche Methoden gefragt:

- Bei der Visualisierung medizinischer Datensätze, wie sie beim Einsatz bildgebender Verfahren wie Ultraschall, CT oder NMR (MRI) entstehen, sind insbesondere Methoden der *Bildverarbeitung* und des *Volume Rendering* erforderlich.
- Für die Visualisierung von raumauflösenden Simulationsdatensätzen, also bspw. des Geschwindigkeitsfelds und der Temperaturverteilung einer Strömung, spielt die effiziente Generierung von Isoflächen, Höhenlinien, Schnittbildern oder Streichlinien (s. u.) eine herausragende Rolle (also auch hier wieder klassische Aufgaben des *Volume Rendering*).
- Oftmals sind grundlegende Techniken aus der Computergraphik wie die geometrische Modellierung dreidimensionaler Szenen, Beleuchtungsmodelle oder Sichtbarkeitsentscheide von Bedeutung. Man denke etwa die Simulation der Fahrdynamik eines PKW auf einer Teststrecke, oder an simulierte Bewegungen eines Industrieroboters.
- Schließlich fließen auch Methoden der Virtual Reality ein, wenn etwa ein Architekturmodell visualisiert werden soll, um einen virtuellen Spaziergang durch das Gebäude zu gestatten, oder bei synthetischen Geländemodellen für Flugsimulatoren etc.

Im Folgenden befassen wir uns kurz mit einigen für die Visualisierung interessanten Methoden

der Bildverarbeitung. Anschließend werden einige relevante Themen aus der Computergraphik diskutiert und die gängigen Methoden zur Visualisierung dreidimensionaler Datensätze vorgestellt. Schließlich vollziehen wir den Schritt vom statischen Bild zum dynamischen Film.

6.1 Methoden der Bildverarbeitung

Die Visualisierung streift auch den Bereich bildgebender Verfahren, bei denen Messwerte (Radar-Aufnahmen der Satelliten ERS-1 und ERS-2, Computertomogramme, Aufnahmen von Rasterelektronenmikroskopen etc.) in Bilder im herkömmlichen Sinne umgewandelt werden sollen. Unter einem Bild ist dabei ein diskreter Datensatz zu verstehen, genauer ein zwei- oder (in einem verallgemeinerten Sinne auch) dreidimensionales Feld von Graustufen- oder Farbwerten. Im Gegensatz zur Visualisierung von Simulationsdaten, bei der aus künstlich errechneten Daten Bilder erzeugt werden sollen, handelt es sich hier also eher um eine optische Rekonstruktion. Hierbei finden Techniken aus der Bildverarbeitung starke Anwendung.

In einer groben Einteilung kann man fünf Kategorien bildverarbeitender Operationen unterscheiden. Man beachte, dass diese – trotz der in der Bildverarbeitung oft üblichen Beschränkung auf den zweidimensionalen Fall – auch auf dreidimensionale Bilddatensätze anwendbar sind.

- *Geometrische Verarbeitung (geometric processing)*: Veränderung der Form eines Bildes bzw. von Teilen davon; solche *geometrischen Operationen* hängen von der Position des Pixels ab, nicht jedoch von den Grauwerten im Pixel und in seinen Nachbarpunkten.
  Beispiele:
  - Vergrößerung/Verkleinerung,
  - Drehung,
  - Verzerrung.
- *Punktweise Bearbeitung (point-to-point processing)*: Modifikation des Bildes durch einfache lokale Veränderungen in den Pixelwerten; solche *Punktoperationen* hängen i. A. vom lokalen Grau- bzw. Farbwert ab, nicht jedoch von Position oder Umgebung.
  Beispiele:
  - *Konstantenaddition*: Modifikation aller Pixelwerte um eine Konstante;
  - *contrast stretching*: Aufblähen der Helligkeitswerte auf den vollen darstellbaren Bereich (Erhöhung des Kontrasts, Aufspüren von Details);
  - *Falschfarbendarstellung*: unterschiedlichen Graubereichen werden verschiedene Farbtöne zugewiesen, wobei jede Farbe den gesamten Intensitätsbereich durchläuft; große Unterschiede in den Pixelwerten äußern sich damit als Farbunterschiede, kleinere wie zuvor in der Intensität (weitere Erhöhung des Kontrasts);
  - *Exponentialtransformation*: die Belichtung von Filmen erfolgt nicht proportional, sondern logarithmisch zur Lichtstärke; eine Exponentialtransformation kann die ursp. Intensitätswerte restaurieren.
- *Lokale Bearbeitung (local-to-point processing)*: Die Umgebungsgrauwerte fließen in den neuen lokalen Wert ein.
  Beispiele:
  - *lokale gewichtete Mittelungsoperatoren*: Auswertung einer lokalen Umgebung (Rechteck, Kreis, ...), gewichtete Mittelung (*Faltung (convolution)*, allg. Glättungsoperatoren);
  - *lokale Rangoperatoren*: Auswertung einer lokalen Umgebung, Sortieren der auftretenden Werte, Auswahl bzgl. der sortierten Werte (Median, Mittel der Extremwerte etc.);

  - *Segmentierung*: Zuordnung zu Klassen via Schwellwerte, *Filterung*;
  - *Flanken- bzw. Kantendetektion*: Pixelwerte in glatten Bereichen werden abgeschwächt oder bleiben unverändert, Unterschiede werden zur Konturierung hervorgehoben.
  - *Merkmalsextraktion*: Lokalisierung bestimmter Merkmale (Schwellwertüberschreitung, Flanken, Oszillationen, Muster, Texturen etc.).

- *Vergleichende Verarbeitung (ensemble processing)*: Auswertung mehrfacher Bilder zur Ableitung eines neuen Bilds.
  Beispiele:
  - *Bewegungsdetektion*: Erkennen von Veränderungen oder Bewegungen durch Vergleich zweier zu verschiedenen Zeitpunkten aufgenommener Bilder, Anwendungen in Medizin und bei Satellitenaufnahmen;
  - *Differenzenbilder*: Anwendungen wiederum in der Medizin (Visualisierung des Blutkreislaufs über Vergleich vor und nach der Injektion von Kontrastmittel, da Blut, Blutbahnen und das umgebende Gewebe nahezu dieselben Röntgen-Absorptionseigenschaften haben und somit schlecht sichtbar gemacht werden können) und bei Satellitenaufnahmen (Aufnahmen des Landsat-Satelliten als Kombination der Antworten auf vier Frequenzbänder zur Ermittlung von Vegetation oder Aufspürung von Bodenschätzen).

- *Globale Verarbeitung (domain processing)*: komplizierte Pixelwertmodifikationen auf der Grundlage lokaler und globaler Information (Fourier- und verwandte Transformationen); diese Verfahren stellen die aufwendigsten und mächtigsten Methoden dar und sind vielseitig einsetzbar, etwa zur Bildschärfung, Flankenhervorhebung, Kontrasterhöhung, Detailrestaurierung sowie zur Komprimierung.
  Beispiele zur Kompression:
  - *Cosinustransformation*: auf ihr beruht der JPEG-Standard (Joint Photographic Experts Group) zur Komprimierung digitaler Bilddaten;
  - *Wavelet-Transformation*: verwandt mit der Fourier-Transformation, aber mit in Ort/Zeit *und* Frequenz lokalisierten Basisfunktionen (*Wavelets*); hohe Kompressionsraten, effiziente Algorithmen;

  Am Anfang (medizinischer) bildgebender Verfahren standen Röntgenaufnahmen. Diese stellen Summationsbilder dar, die häufig wegen der gegenseitigen Überlagerung Details nicht erkennen lassen. Mit der *Schichtuntersuchung* (*Tomographie*) können morphologische Einzelheiten in einer wählbaren Schichttiefe scharf abgebildet werden. Die darüber- und darunterliegenden Schichten werden verwischt. Aus den einzelnen Schichtbildern können dann dreidimensionale Ansichten generiert werden. Man unterscheidet Verfahren der *Transmissionstomographie*, bei denen die Passage von Strahlen durch das abzubildende Objekt betrachtet wird, und Verfahren der *Emissionstomographie*, bei denen von inkorporierten Radionukliden ausgesandte Strahlen erfasst werden. Beispiele für erstere sind etwa:
  - *Röntgenbasierte Computertomographie (CT)*: Sie basiert mathematisch auf der *Radon-Transformation*. Anhand der Abschwächung von gebündelten Röntgenstrahlen beim Durchgang durch das zu untersuchende Gewebe werden einzelne Schnittbilder erzeugt, die die Dichte des Gewebes visualisieren.
  - *Kernspintomographie (NMR, MRI)*: Hier wird ein starkes Magnetfeld anstelle von Röntgenstrahlen zur Generierung der Schnittbilder eingesetzt. Visualisiert wird wiederum die Dichte der Materie. Vorteile gg. CT sind u. a. die höhere Auflösung, die größere Empfindlichkeit gegenüber weichem Gewebe sowie die geringere Belastung des Patienten.

– *Confokale Laser-Scanning-Mikroskopie (CLSM)*: Hier werden statt Röntgen- Laserstrahlen durchgeschossen. Verwendung findet die CLSM vor allem in der Biologie (Präparate, nicht am Menschen).

Beispiele für Emissionstomographie sind:

– *Positronen-Emissions-Tomographie (PET)*: Gemessen wird hier die beim Positronenzerfall entstehende Strahlung. Sie dient somit der Darstellung Positronen-emittierender Radionuklide. Das Verfahren ist sehr aufwendig, da die Positronen in einem Zyklotron erzeugt und dann innerhalb kurzer Zeit (Halbwertszeit 60 bis 120 Minuten) eingesetzt werden müssen.

– *Single Photon Emission Computed Tomography (SPECT)*: Hier werden Gammaemitter verwendet. Die SPECT ist der PET hinsichtlich Qualität unterlegen, aber weniger aufwendig und für den Patienten weniger belastend, billiger und oftmals hinreichend gut.

Erfasst wird bei der Emissionstomographie durch Injektion schwach radioaktiv markierter Substrate deren metabolische (d. h. Stoffwechel-) Aktivität (also biologische statt physikalischer Struktur). Beispielsweise werden Zucker oder Aminosäuren von tumorbefallenem Gewebe heftiger abgebaut als von gesundem, so dass diese Substrate die Radionuklide an die richtige Stelle transportieren. Beim Zerfall der Positronen (PET) bzw. durch unmittelbare Emission (SPECT) werden dann Paare von Gammaquanten frei, die sich in exakt entgegengesetzter Richtung vom Ort ihrer Freisetzung entfernen und die als Strahlung erfasst werden können.

In der Medizin kommen heute auch Kombinationen von Transmissions- und Emissionstomographien zum Einsatz, etwa als Überlagerungen einer Kernspintomographie mit einer SPECT. Diese hybride Vorgehensweise bietet den Vorteil, dass sowohl strukturelle als auch metabolische Information sichtbar gemacht und zueinander in Bezug gesetzt werden kann. Das CT-Prinzip kann auch noch allgemeiner eingesetzt werden, etwa für Visualisierungen der inneren Struktur der Erde auf der Grundlage der Ausbreitung mechanischer (seismischer) Wellen.

6.2 Aspekte der Computergraphik

Da die Visualisierung vor allem eine wichtige Anwendung von Methoden aus der Computergraphik ist, spielen diese natürlich auch in unserem Zusammenhang eine bedeutende Rolle. Dies gilt insbesondere für Visualisierungen, bei denen nicht nur drei- oder vierdimensionale Datensätze in fest vorgegebenen Geometrien graphisch dargestellt werden sollen, sondern bei denen Szenen ganz oder teilweise synthetisch generiert werden müssen. Einige Beispiele:

- Bei Crash-Simulationen möchte man nicht nur ein eingedrücktes Finite-Element-Netz zu sehen bekommen, sondern den gesamten Prozess visualisiert haben, entsprechend den Filmaufnahmen der Automobilhersteller und -tester. Dazu müssen PKW und Umgebung geometrisch modelliert werden.

- Die Visualisierung der Simulation einer Fließbandsteuerung erfordert ebenfalls die geometrische Modellierung der ganzen betreffenden Anlage.

- Um solche Simulationen realistischer erscheinen zu lassen, werden oftmals Beleuchtungsmodelle integriert.

- Bei Visualisierungen von Wettersimulationen („Wetterflug“ im Wetterbericht) sind natürliche Phänomene wie Wolken oder Nebel graphisch darzustellen.

- Zur Steigerung der Anschaulichkeit werden Simulationsdaten in virtuelle Szenen integriert und können dann VR-mäßig „begangen“ werden (Simulationsdaten der Umströmung eines Autos im Windkanal werden auf ein hochaufgelöstes CAD-Modell projiziert, das anschließend aus beliebiger Perspektive betrachtet werden kann etc.).
- Einen Schritt weiter als bei der *Virtual Reality* geht man bei der *Augmented Reality*, bei der Simulations- oder Modelldaten in Fotos und Videofilme der Umgebung integriert werden.

Dementsprechend sind verschiedene Themen der Computergraphik auch für die Visualisierung von Bedeutung:

- Die *geometrische Modellierung* befasst sich mit der Erzeugung von Modellen sowohl für einzelne Objekte als auch für ganze Szenen. Man unterscheidet volumen-, oberflächen- und kantenorientierte Modelle. Zu Ersteren zählen beispielsweise:
  - *Normzellenschema*: Aufbau aus Würfeln einheitlicher Größe;
  - *Oktalbäume (Octrees)*: rekursive Zerlegung eines Ausgangswürfels in acht Subwürfel halber Kantenlänge;
  - *CSG (Constructive Solid Geometry)*: aus Grundelementen (Primitiven, etwa Kugeln, Zylinder, Quader etc.) werden mittels der Mengenoperationen Vereinigung, Durchschnitt und Differenz komplizierte Objekte gebaut.

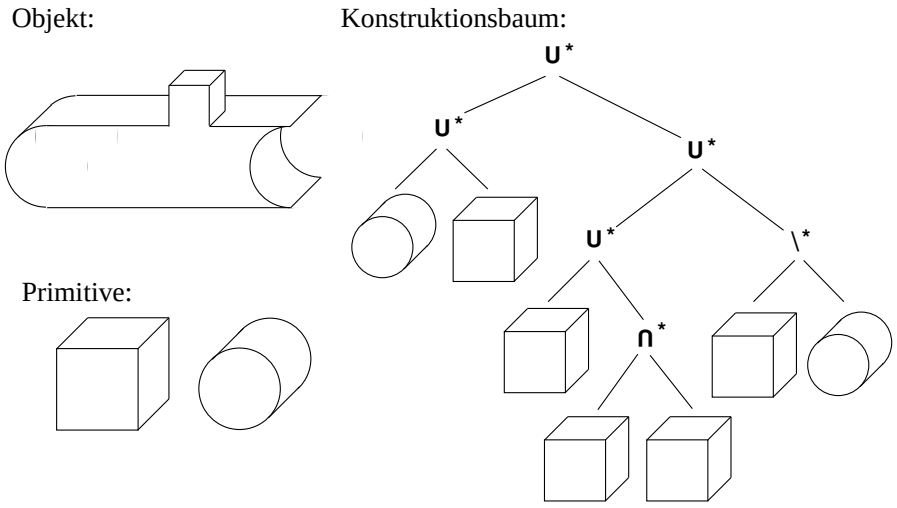


Abbildung 6.1: Konstruktionsbaum im CSG-Schema

Prominentester Repräsentant der Kantenmodelle ist das *Drahtmodell (Wireframe)*. Oberflächenmodelle (*Boundary-Representation-* oder kurz *B-Rep-Modelle*) spielen vor allem im CAD ein große Rolle, da sie wesentlich kompliziertere Oberflächen (Freiformflächen: Bézierflächen, NURBS) zulassen. Bei Kanten- und Oberflächenmodellen tritt neben die *geometrische* Information (Form der Objekte bzw. Flächenstücke, Lage der Punkte, Art der Verbindung der Punkte) noch *topologische* Information, die die Nachbarschaftsbeziehungen zwischen Punkten, Flächen und Kanten beinhaltet. Die Topologie ist dabei grundlegender als die Geometrie, da zwei Körper mit derselben topologischen Information, d. h. demselben Beziehungsgeflecht aus Punkten, Kanten und Flächen, unterschiedliche geometrische Information tragen, also von unterschiedlicher Gestalt sein können. Objekte werden deshalb über ihre topologische Struktur beschrieben, die dann mit geometrischer Information attributiert wird.

- Aufgabe des *Rendering* ist es, aus den modellierten Szenen realistische Computerbilder zu generieren. Wichtige Teilaufgaben sind dabei:

  - *Perspektive*: Mittels Parallel- oder Zentralprojektion wird eine Ansicht erzeugt. Techniken zum Sichtbarkeitsentscheid (*Hidden Line/Surface Removal*) sorgen dafür, dass nur das für den Betrachter Sichtbare gezeichnet wird.

  - *Lokale Beleuchtung* und *Reflexion*: Wie berechnet sich die Lichtintensität in einem Punkt der Szene, wie können diffuse und spiegelnde Reflexion modelliert werden?

  - *Shading*: Für Echtzeit-Anwendungen (z. B. Flugsimulatoren) sind globale Beleuchtungsmodelle zu teuer. Hier muss das Bild aus einzelnen Polygonen aufgebaut werden, wobei die Polygone effizient „beleuchtet“ werden müssen.

  - *Globale Beleuchtung*: Photorealismus ist nach wie vor eines der Hauptziele der Computergraphik. Hierfür reichen einfache Shading-Techniken nicht mehr aus, es müssen vielmehr aufwendige Verfahren zur Realisierung von Spiegelungen und von diffuser Beleuchtung eingesetzt werden (*Ray Tracing, Path Tracing, Radiosity, Light Ray Tracing, Photon Tracing*).

- *Stereoskopie*: Bei den meisten in diesem Kapitel angesprochenen Visualisierungstechniken bleibt die Problematik, dass sich dreidimensionale Daten nur schwer direkt darstellen lassen. Auch wenn ein aufwendiges Rendering den Raumbildeindruck unterstützen kann, auch wenn mittels Iso- oder Schnittflächen (siehe den folgenden Abschnitt) nahezu beliebige Ausschnitte ausgewählt werden können, die Betrachtung bleibt im Wesentlichen zweidimensional. Echt dreidimensionale Phänomene lassen sich aber nur mittels stereoskopischem Raumbildsehen voll erfassen. Kann man etwa virtuell ein durchströmtes poröses Medium durchwandern und dabei geometrische Details, Wirbelstrukturen o. Ä. plastisch wahrnehmen, so erhöht sich der für den Betrachter verwertbare Informationsgehalt der Visualisierung beträchtlich.

  Stereoskopie erfordert, analog zum natürlichen Sehvorgang, zwei getrennte Bilder für das linke und das rechte Auge. Durch die stereoskopische Überlagerung entsteht dann der räumliche Eindruck. Zur Erzeugung der beiden getrennten Bilder gibt es eine Vielzahl von Lösungsansätzen, die grob in zwei Klassen eingeteilt werden können. Bei den sogenannten *feldsequentiellen* oder *zeit-multiplexen* Verfahren werden die Bilder für das linke und das rechte Auge abwechselnd auf dem Bildschirm dargestellt. Damit jedes Auge nun auch tatsächlich nur „seine“ Bilder sieht, muss ein synchronisiertes Verschlusssystem eingesetzt werden, das jeweils ein Auge abdeckt. Man unterscheidet hier zwischen *aktiven Systemen*, bei denen der Verschlussmechanismus in eine Brille integriert ist (*Shutter-Brille*, teuer), und *passiven Systemen*, bei denen der Verschluss als abwechselnd polarisierender Filter vor dem Bildschirm realisiert ist, wobei der Betrachter dann eine Brille mit entsprechenden Polarisierungsfiltern (billig) tragen muss.

  Bei den sogenannten *zeit-parallelen* Verfahren werden dagegen die beiden Ansichten nicht abwechselnd, sondern gleichzeitig gezeigt. Dies erfordert natürlich optische Sehhilfen für die korrekte Trennung der beiden Bilder. Bei der Methode der *Anaglyphen* werden beide Ansichten in Komplementärfarben (Rot und Grün oder Rot und Cyan) erzeugt. Farbdarstellungen sind nicht (bzw. nur sehr eingeschränkt) möglich. Der Betrachter trägt dabei eine Brille mit zwei entsprechenden Farbfiltern (billig). Allerdings ist das Verfahren mit zahlreichen Nachteilen behaftet: Der Betrachter ist an eine feste Position gebunden, es kommt zu Restbildern bei nicht perfekter Überlappung, zu Trübungen des Raumbildeindrucks bei nicht exakter Positionierung sowie zu Aliasing-Effekten. Eine alternative zeit-parallele Möglichkeit ist die Aufeinanderprojektion zweier unterschiedlich polarisierter Bilder. Der Betrachter trägt eine billige Polarisationsbrille und ist in seiner Position nicht gebunden. Die Qualität des Raumbildeindrucks ist hier nahezu perfekt, auch Farben und Bildsequenzen stellen kein Problem dar. Im letzteren Fall muss natürlich eine Synchronisation der beiden Bilder erfolgen, am einfachsten über die Überlagerung von Zwillingsbildern (siehe Abb. 6.2). Akzeptiert man eine aufwendigere Technologie, so können die beiden Bilder auch mittels eines Head-Mounted Displays unmittelbar vor den Augen gezeigt werden.

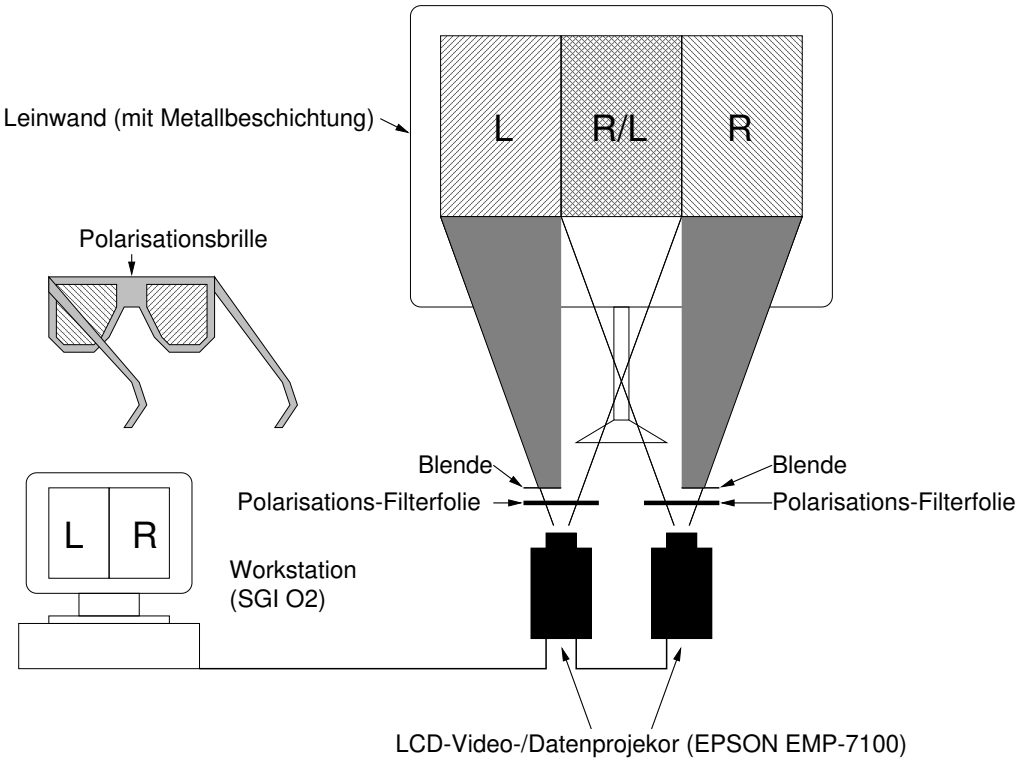


Abbildung 6.2: Stereoprojektion mittels zweier LCD-Beamer

Man beachte, dass die beschriebenen Verfahren der Stereoskopie zwar realistischen Raumeindruck vermitteln, sich aber dennoch auf (zwei) rein zweidimensionale Abbildungen derselben Szene abstützen. Das (derzeit) einzige *echt* dreidimensionale Verfahren zur Bildaufzeichnung und -rekonstruktion stellt die *Holographie* dar, bei der auf *einem* Bild der Raumeindruck festgehalten wird. Allerdings ist die Holographie ein rein photographisches Verfahren und somit kein Verfahren zur Visualisierung in unserem Sinne (es gibt keine digitalen, synthetischen Hologramme). Bei der Holographie wird die Interferenz von Lichtwellen ausgenutzt, wobei zur Erzielung perfekter Darstellungen phasenreines Licht (also z. B. Laserlicht) erforderlich ist. Grob gesprochen, beruht das Prinzip darauf, dass das Interferenzmuster des von der Szene reflektierten Lichts mit einem Referenzstrahl auf Film aufgezeichnet wird. Durch Projektion des Referenzstrahls im Idealfall kann das reflektierte Licht und damit das dreidiemnsionale Bild rekonstrukiert werden, und zwar unabhängig von der Perspektive des Betrachters. Glücklicherweise braucht man nicht immer das exakt gleiche Laserlicht und manchmal sogar überhaupt keinen Laser.

6.3 Visualisierung raum- und zeitaufgelöster Simulationsdaten

Gegeben sei das (in Form einer großen Menge von Zahlen vorliegende) Ergebnis einer numerischen Simulation, also i. A. eine skalare oder vektorwertige Funktion $f : \mathbb{R}^d \supset \Omega \to \mathbb{R}^m$, $m \in \mathbb{N}$, über einem d-dimensionalen Feld. Im Folgenden diskutieren wir kurz einige Techniken bzw. Möglicheiten zur optischen Darstellung solcher Datensätze. Der uns vor allem interessierende Fall ist dabei $d = 3$ bzw., bei Zeitabhängigkeit, $d = 4$.

- **Auswahl und Darstellung von (niedrigdimensionalen) Teilbereichen:**
  Die direkte Visualisierung von im drei- oder höherdimensionalen Raum gegebenen Funktionen ist schwierig. Deshalb beschränkt man sich oft auf die Darstellung bzw. Hervorhebung zwei- oder eindimensionaler Teilbereiche (*Dimensionsreduktion*). Die populärsten hiervon sind:

  – *Schnittebenen (Slices, siehe Abb. 6.3)*: Man betrachtet die interessierenden Größen nicht im Grundgebiet Ω, sondern im Schnitt einer Ebene E mit Ω. Auf der Schnittebene können dann zweidimensionale Techniken angewandt werden.

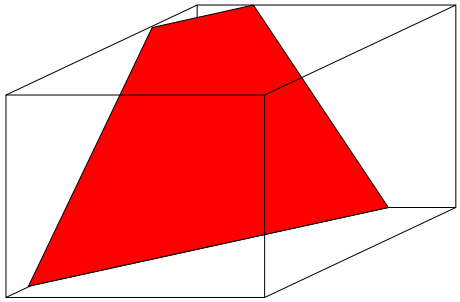


Abbildung 6.3: Beispiel einer Schnittebene

  – *Isoflächen (Isosurfaces, siehe Abb. 6.4)*: Dargestellt wird die Hyperfläche, auf der eine der berechneten skalaren ($m = 1$) Funktionen f einen fest vorgegebenen Wert annimmt: $I(c) := \{\mathbf{x} \in \Omega : f(\mathbf{x}) = c\}$. Will man Isoflächen interaktiv erzeugen und verschieben (Änderung der Konstante c), so bedarf es einerseits effizienter Algorithmen zur schnellen Bestimmung der Isoflächen (etwa der Marching-Cubes-Algorithmus), andererseits einer hinreichend leistungsfähigen Graphik-Hardware (bei einem typischen Beispiel einer Simulation von Strömung und Transport in einer komplizierten Hindernisgeometrie (Auflösung 512^3 Zellen) besteht eine Isofläche bspw. aus 2 Millionen Polygonen). Auf der Isofläche lassen sich zweidimensionale Techniken zur Visualisierung weiterer Größen einsetzen. Das Pendant zu Isoflächen in zweidimensionalen Fall sind *Isolinien* bzw. *Höhenlinien*.

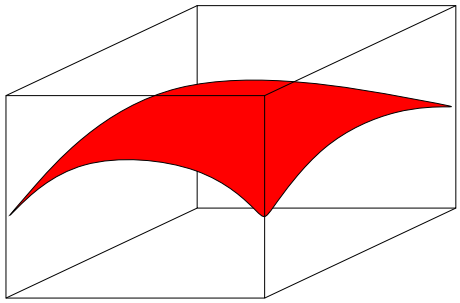


Abbildung 6.4: Beispiel einer Isosurface

  – *Stromlinien (Streamlines)*: Stromlinien dienen zur Visualisierung von Vektorfeldern ($m > 1$), d. h. von vektorwertigen Daten. Sie sind definiert als Kurven in Ω mit in jedem Punkt zum jeweiligen Ortsvektor des Feldes paralleler Tangente. Die Auswahl der Stromlinie erfolgt durch die Auswahl eines Startpunkts.

  – *Bahnlinien (Particle Tracing)*: Bahnlinien werden zur Visualisierung zeitabhängiger Vektorfelder eingesetzt. Ein virtuelles Teilchen wird an einem Startpunkt in das Gebiet

Ω eingebracht und folgt nun in der Zeit dem jeweiligen Ortsvektor des Vektorfeldes. Die Bahn des Teilchens (theoretisch eine kontinuierliche Kurve, praktisch eine Folge diskreter Positionen zu diskreten Zeitpunkten) definiert dabei die Bahnlinie.

- *Streichlinien (Streaklines)*: Auch Streichlinien dienen der Visualisierung zeitabhängiger Vektorfelder. Ab einem Startzeitpunkt werden an einem Startpunkt (theoretisch kontinuierlich, praktisch in festen Abständen) Partikel injiziert, die sich alle den jeweiligen Ortsvektoren des Feldes folgend weiterbewegen. Die dadurch zu einer vorgegebenen Zeit entstandene Linie wird Streichlinie genannt. Bahnlinien geben also die Bewegung *eines* Partikels über ein Zeitintervall wieder, Streichlinien dagegen die Position *vieler* vom gleichen Ausgangspunkt startenden Partikel zu einem festen Zeitpunkt. Im stationären Fall stimmen Streichlinien und Bahnlinien mit Stromlinien überein.
  Neben Streichlinien sind auch *Streichbänder* gebräuchlich. Hier werden die Partikel nicht an einem einzigen Punkt, sondern längs eines Intervalls injiziert. Im zeitlichen Fortlauf entstehen somit keine Kurven, sondern breite Bänder. Vorteile von Streichbändern sind die Möglichkeit zur Darstellung weiterer Information auf ihnen sowie das Sichtbarmachen von Rotationen. Zur Verbesserung des dreidimensionalen Eindrucks werden auch *Streichröhren* eingesetzt.

- **Farbgebung**:
  Die Intensität einer (skalaren) Größe kann durch den Farbton oder durch die Intensität ausgedrückt werden. Dabei sollte die Visualisierung die Intuition unterstützen (im Falle der Temperatur z. B. blau für kalt und rot für warm und nicht umgekehrt, bei einer Stoffkonzentration schwache Intensität für geringe Werte etc.). Außerdem können verschiedene Farben für verschiedene Größen benutzt werden (rote Töne für die Temperatur, Blautöne für die Stoffkonzentration). Hierbei setzt die Wahrnehmungsfähigkeit des Menschen allerdings Grenzen: Bei mehr als fünf Farben nimmt die parallele Verarbeitungsfähigkeit ab. Wichtig ist in jedem Fall die passende Wahl der Skala: Deckt diese einen zu großen Bereich ab, verschwinden kleinere Unterschiede in den Daten, ist der darstellbare Bereich zu klein, erscheinen marginale Unterschiede als gewaltig.

- **Dimensionserhöhung**:
  Eine auf einem zweidimensionalen Gebiet gegebene Größe kann in einem dreidimensionalen Bild als Funktion über dem Grundgebiet aufgetragen werden (*Funktionsplot*). Solche *Surface Plots* machen allerdings nur in 2 D Sinn.

- **Verwendung von Darstellungsprimitiven**:
  Die meisten Visualisierungstechniken stellen die ausgewählten niedrigdimensionalen Teilbereiche als Kurven oder Flächen dar. Alternativ dazu können auch bestimmte Darstellungsprimitive aufgebracht werden. Ein Beispiel hierfür sind Pfeile im Falle eines Geschwindigkeitsfelds, die via Größe und Farbe dann noch weitere Informationen tragen können. Kugeln oder Bälle sind eine andere gebräuchliche Primitivenart, die vor allem bei Bahn- oder Streichlinien eingesetzt wird und die einzelnen virtuellen Teilchen darstellen soll. Werden sie in festem Zeitabstand in die Szene eingebracht, ist ihr Abstand zugleich ein Maß für die absolute Geschwindigkeit im Fluid.

- **Unterstützung realistischer Darstellung**:
  Projektionen (Zentral- oder Parallelprojektion) ergeben einen perspektivischen Eindruck und unterstützen die räumliche Wahrnehmung. Beleuchtungstechniken (Lichtquellen, Schatten, Spiegelungen, globale Beleuchtung) können zu realistischeren Bildern führen. Die realistische Darstellung spielt bei der Visualisierung von Datensätzen kaum eine Rolle, bei Aufgaben der Visualisierung in Virtual-Reality-Anwendungen kommt ihr allerdings eine entscheidende Bedeutung zu.

- **Verwendung von Texturen**:
  Iso- oder Schnittflächen bzw. Bänder können zur Wiedergabe zusätzlicher Information mit einer Textur versehen werden, d. h., ein Muster wird auf die Fläche aufgetragen. Dies kann

ein fest vorgegebenes Muster zur Identifizierung der Fläche oder ein durch eine bestimmte Problemgröße definiertes Muster sein (Isofläche der Stoffkonzentration, Norm bzw. Absolutbetrag der Geschwindigkeit via Farbe, Visualisierung eines Feldes über eine Vektorfeldprojektion auf die Fläche, vergleichbar Eisenfeilspänen in einem Magnetfeld). Die Textur kann aber auch die Beschaffenheit des Grundgebiets visualisieren (Versickerung von Öl im Erdreich: auf einer Isofläche der Ölkonzentration wird die Charakteristik des Erdbodens (Sand, Lehm, Kies etc.) dargestellt).

- **Beschriftung und Legende**:
  Ohne Legende sind Visualisierungen oftmals nicht zu verstehen oder können leicht missverstanden werden (beispielsweise aufgrund einer Fehlinterpretation der Skalen). Beschriftungen im Bild können hilfreich sein, sie können aber auch den Blick des Betrachters von der eigentlichen Information ablenken.

Die Visualisierung stützt sich auf die diskreten Ergebnisdaten eines Simulationslaufs. Damit erbt sie in natürlicher Weise auch die zugrunde liegende diskretisierte Geometrie, also das Gitter (strukturiert oder unstrukturiert, regulär oder adaptiv verfeinert). Kann ein Visualisierungsalgorithmus die ererbte Geometrie direkt behandeln, dann können Berechnung und graphische Darstellung auf derselben Gitterstruktur erfolgen. Andernfalls müssen die berechneten Daten auf ein geeignetes Gitter abgebildet (d. h. interpoliert) werden.

Zur Beschreibung von Datensätzen verwenden Visualisierungssysteme wie z. B. AVS/Express eine Komponentenhierarchie (vgl. Abb. 6.5).

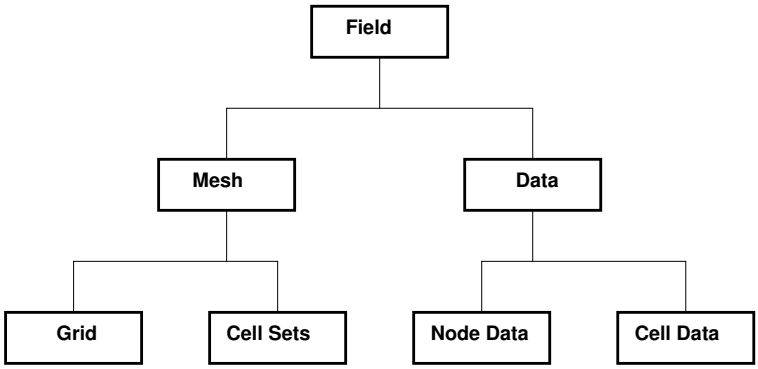


Abbildung 6.5: Komponentenhierarchie in AVS

- **Field**: Das Feld, die oberste Komponente in der Hierarchie, repräsentiert die gesamte Information (in voller Allgemeinheit). Es setzt sich zusammen aus einer Netz- und einer Datenkomponente.
- **Mesh**: Die Netz-Komponente liefert die geometrische Beschreibung des zugrunde liegenden Gebiets (eigentliche und faktische Dimensionalität des Gebiets[1], Typ des Netzes: unstrukturiert, strukturiert, rechtwinklig, uniform, vgl. Abb. 6.6) und definiert, wo und wie die Daten organisiert sind. Die entsprechenden Informationen werden mittels einer Gitter- und einer Zellen-Komponente gehalten.
- **Grid**: Die Gitter-Komponente enthält die Positionen der Gitterpunkte. Auch hier wird zwischen unstrukturierten und strukturierten Gittern unterschieden. Je mehr Struktur ein Netz bzw. ein Gitter trägt, umso weniger Information muss explizit gehalten werden. Im uniformen Fall etwa liegen Positionen und Nachbarschaftsrelationen der Gitterpunkte implizit fest.

[1] Eine Isosurface ist ein eigentlich zweidimensionales Gebilde, lebt faktisch aber im dreidimensionalen Raum.

- **Cells**: Eine Menge von Punkten mit zugeordneten Werten reicht zur Erzeugung eines Bildes (einer Isosurface etwa) noch nicht aus. Die Frage der Nachbarschaft bzw. des Zusammenhangs (*connectivity*) muss beantwortet werden. Dies leistet die Zellen-Komponente. Zellen stellen dabei allgemeine geometrische Einheiten dar und können Punkte, Kanten, Flächen (Dreiecke, Vierecke) oder Volumen (Tetraeder, Hexaeder) sein. Bei strukturierten Gittern ergibt sich das topologische Beziehungsgeflecht (wer grenzt an wen) automatisch aus der Information der Gitter-Komponente, bei unstrukturierten Gittern muss es separat gespeichert werden.

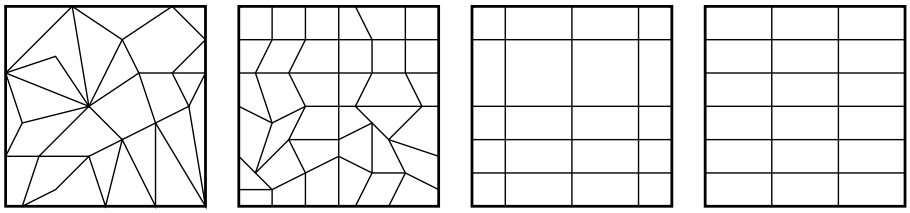


Abbildung 6.6: Netz- bzw. Gittertypen am Beispiel AVS: unstrukturiert (links) und strukturiert (konform, rechtwinklig, uniform; v. l. n. r.). Im unstrukturierten Fall müssen alle Gitterpunkte und die Zusammenhangsinformationen explizit gespeichert werden. Im konformen Fall reichen die Positionen der Punkte, im rechtwinkligen Fall die Vektoren der Maschenweiten, und bei uniformen Gittern reichen die konstanten Maschenweiten.

- **Data**: Die Daten-Komponente repräsentiert die Daten des Gitters (also die berechneten bzw. gespeicherten bzw. darzustellenden Funktionen.
- **Node Data**: Knotendaten sind diejenigen Daten, die den Knoten (Gitterpunkten) im Gitter zugeordnet sind.
- **Cell Data**: Zelldaten sind die Daten, die den Zellen zugeordnet sind. In der Strömungsmechanik arbeitet man bspw. oftmals mit *versetzten Gittern (staggered grids)*, bei denen die Druckwerte den Mittelpunkten und die Geschwindigkeitswerte den Seitenmitten der quaderförmigen Zellen zugeordnet werden.

Die unterschiedliche Zuordnung der jeweiligen Größen zu Punkten oder Zellen ist für die Visualisierung durchaus von Bedeutung: Im ersten Fall erhält nur der Punkt einen entsprechenden Wert bzw. eine entsprechende Farbe, dazwischen wird interpoliert. Sind Größen Zellen zugeordnet, werden die ganzen Zellen den Werten entsprechend konstant eingefärbt (was aus ästhetischen Gesichtspunkten natürlich unerwünscht ist, da es zu harten Übergängen führt).

Ein weiterer zentraler Punkt ist die Schnittstelle zwischen Simulations- und Visualisierungsdaten. Moderne Diskretisierungstechniken reduzieren den Datenaufwand bei der Berechnung zum Teil erheblich (hierarchische Strukturen, adaptive Verfeinerung), Visualisierungssysteme setzen oftmals auf wesentlich einfacheren und reguläreren Strukturen auf und verschenken somit viel an Effizienz. Zum Beispiel ist es mehr als ärgerlich, wenn zu Visualisierungszwecken die Daten eines adaptiven Gitters erst auf ein reguläres interpoliert werden müssen, oder wenn die Beschreibungsart geändert werden muss. Maßgeschneiderte Schnittstellen helfen hier, Reibungsverluste zu vermeiden.

Schließlich sei darauf hingewiesen, dass auch bei der in diesem Abschnitt behandelten Visualisierung von raum- und zeitaufgelösten Simulationsdaten Techniken der Bildverarbeitung hilfreich sein können, beispielsweise die Merkmalsextraktion zum Aufspüren bestimmter Details.

6.4 Vom Bild zum Film

Bei zeitabhängigen Simulationsdaten befriedigt das Einzelbild noch nicht alle Wünsche, da immer nur ein isolierter Zeitpunkt betrachtet werden kann. Sequenzen von Einzelbildern erhöhen die

Aussagekraft bereits, Ziel bei zeitabhängigen Problemen muss jedoch der Film sein. „Ich könnte mir sogar vorstellen, dass man diesen Bildern das Laufen beibringen kann, indem man, was auch sonst, viele Bilder aneinanderhängt“ (Zitat eines Mathematikers, Februar 1999). In der Tat funktioniert's so, üblicherweise mit 25 bzw. 30 Bildern pro Sekunde (Videobildfrequenz), oft reichen aber auch weniger. Allerdings sind einige Dinge zusätzlich zu beachten. Die Folge der Einzelbilder muss natürlich so gewählt werden, dass weder ruckartige Übergänge noch zu langsames Fortschreiten den Eindruck stören. Solche Rucke können auch durch das Aufeinandertreffen unterschiedlicher Farbgebungen auf zwei aufeinanderfolgenden Einzelbildern entstehen. Da die Übernahme der Skala des ersten Bildes für alle folgenden Bilder aber ebenfalls Risiken birgt (werden die Unterschiede im Lauf des Films kleiner, ist später nichts mehr zu erkennen, werden sie größer, fallen immer mehr Werte aus dem darstellbaren Bereich), müssen im Vorfeld über alle Bilder globale Minima und Maxima ermittelt werden, um eine einheitliche und aussagekräftige Einfärbung sicherzustellen.

Für Filme geeignet sind prinzipiell alle Einzelbild-Visualisierungstechniken, die die Verhältnisse zu einem festen Zeitpunkt beschreiben. Welche Darstellungsmöglichkeit die Dynamik am besten wiedergibt, hängt vom speziellen Problem ab.

Was aber tun, wenn die Erzeugung einer solchen Vielzahl von Einzelbildern zu aufwendig ist? Bei dreidimensionalen Strömungssimulationen liegt die Rechenzeit für einen Zeitschritt leicht im Stundenbereich. Für einen einminütigen Film mit 1800 Bildern bedeutet dies somit einen nicht unerheblichen Rechenaufwand. Da die Zeitschrittweite bei der Simulation numerischen Zwängen unterworfen ist (Stabilitätsbedingungen), fällt eine gewisse Anzahl von einzelnen Bildern (Frames) ohnehin an. Man kann aber u. U. auf die visualisierungsbedingte Verkleinerung der Schrittweite verzichten und stattdessen weitere Bilder (Frames) durch Interpolation erzeugen (*Inbetweening*). Man muss sich aber der Tatsache bewusst sein, dass das Produkt dann keine Visualisierung von Simulationsdaten im engeren Sinne mehr ist (siehe auch Regel 6 im nachfolgenden Abschnitt).

6.5 Bunte Bildchen über alles

Dass die Visualisierung wichtig, ja für das moderne wissenschaftliche Rechnen unverzichtbar ist, haben wir jetzt gelernt. Etwas Skepsis kann aber dennoch nie schaden.

Ein wichtiger Aspekt ist zunächst, dass Visualisierungen nicht „besser“ als die ihnen zugrunde liegenden Daten sein können. Ein Bild sagt beispielsweise per se nichts über die Qualität des verwendeten Modells aus, auch wenn Falsifizierungen manchmal möglich sind. Außerdem führt auch der denkbar langsamste numerische Algorithmus (so er denn korrekt ist) zum richtigen Ergebnis und somit zum hübschen Bild, egal, wie lange der Rechner dafür schwitzen musste.

Ein zweiter Problempunkt sind *optische Artefakte*: Spiegelt das, was ich sehe, tatsächlich die Wirklichkeit wider, oder handelt es sich nur um eine optische Täuschung?

Drittens muss das Problem der fehlerhaften Interpretation beachtet werden. Deute ich die Abbildungen richtig (ein Problem, das vor allem in der Frühphase der Computertomographie allgegenwärtig war, als Tumoren oftmals schlichtweg übersehen wurden), handelt es sich bei dem interpretierten Phänomen wirklich um eine Eigenschaft der Daten, oder liegt es vielleicht nur an der Wahl der Parameter (Farbskala, Intensität, Kontrast etc.)? Trotz aller Suggestivkraft guter Visualisierungen sollte man von der Disziplin bzw. dem Phänomen, welches visualisiert wurde, ein gewisses Maß an Ahnung haben, bevor man sich auf Interpretationen einlässt.

Schließlich sind die Manipulationsmöglichkeiten bei der Visualisierung nahezu unbegrenzt. In diesem Zusammenhang sei die Arbeit *13 Ways to Say Nothing with Scientific Visualization* (A. Globus, E. Raible, NASA Ames Research Center, Report RNR-92-006) erwähnt, in der dem angehenden Visualisierer folgende Ratschläge erteilt werden:

1. *Never include a colour legend*:
   Auf die Optik kommt's an, Legenden lenken den Betrachter nur ab.
2. *Avoid annotation*:
   Erklärungen unterminieren bloß die visuelle Verwirrung.
3. *Never mention error characteristics*:
   Wenn ein Bild gut aussieht, muss es einfach auch korrekt sein.
4. *When in doubt, smooth*:
   Störende Peaks immer herausglätten, denn: Schönheit und rasche Publizierbarkeit sind allemal wichtiger als Genauigkeit.
5. *Avoid providing performance data*:
   Nur Kleingeister scheren sich darum, wieviel Zeit die Visualisierung in Anspruch nimmt.
6. *Quietly use stop-frame video techniques*:
   Warum für eine Animation unnötig viele Einzelbilder (Frames) produzieren, wenn einzelne wiederholt werden können oder zwischen einzelnen interpoliert werden kann?
7. *Never learn anything about the data or scientific discipline*:
   Wer nicht weiß, was er im Bild erwartet, muss sich ggfs. auch nicht auf Fehlersuche begeben – es lebe die Unvoreingenommenheit.
8. *Never compare your results with other visualization techniques*:
   Die Bilder der anderen könnten besser aussehen.
9. *Avoid visualization systems*:
   Warum den Mist von anderen benutzen (vgl. Regel 8), wenn man selbst hübsche Sachen erfinden kann?
10. *Never cite references for the data*:
    Das wäre nur eine Einladung zum Nachprüfen und Nörgeln.
11. *Claim generality, but show results from a single data set*:
    Wozu gibt's 'o. B. d. A.'?
12. *Use viewing angle to hide blemishes*:
    Warum 'Hidden Surface Removal' nicht zum 'Disturbing Detail Removal' benutzen?
13. *'This is easily extended to 3D'*:
    siehe Regel 11.

Damit die Visualisierung aber nicht alles abbekommt, sei auch auf die *Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers* (D. Bailey, Supercomputer Review, Aug. 1991, pp. 54–55) hingewiesen: Getrickst wird schließlich überall.

Kapitel 7

Validation

Kapitel 8

Case Study: Computational Fluid Dynamics

8.1 Das mathematische Modell

8.2 Die Diskretisierung

8.3 Die Implementierung

8.4 Die Visualisierung