UNIVERSITY of CALIFORNIA
Santa Barbara

Fast Algorithms with Applications to PDEs

A Dissertation submitted in partial satisfaction of the requirements for the degree

Doctor of Philosophy
in
Mathematics

by

William Lyons

June 2005

The dissertation of William Lyons is approved

Hector D. Ceniceros, Committee Co-Chairman

Shivkumar Chandrasekaran, Committee Co-Chairman

Bjorn Birnir

Ming Gu

June 2005

Fast Algorithms with Applications to PDEs

Acknowledgements

I would like to thank, first and foremost, my two co-chairs: Shiv Chandrasekaran and Hector Ceniceros. I can safely say that none of this work would exist without them. When Shiv first proposed to me that I determine an algorithm for multiplying together two "matrices in hierarchically semiseparable form" and stipulated that the algorithm should run in linear time, I refused to believe such a thing was possible. Eventually he convinced me otherwise. A few months later there was an algorithm, a proof and working code to carry it out. At every stage my research has advanced in this manner: Shiv would pick a problem that was difficult, but tractable. He would put in quite a bit of time explaining to me why the problem was interesting and discussing approaches. And eventually, usually much later, the work would get done. I can't imagine what more a candidate can ask from an advisor. Hector, as well as motivating the current applications to PDEs, introduced me to the dark art of actually getting a manuscript published. Without his consistent proofreading, editing and corrections (not to mention the mathematical advice) the first "rebus" paper would never have seen the light of day. Above all else, both Shiv and Hector were always a pleasure to work with: both are brilliant minds in their respective fields and succeeded in communicating their interests and enthusiasm to me.

There are also many people, at UCSB and elsewhere, who I would like to thank for their help and support over the years. Xu-Dong Liu, Ming Gu and Bjorn Birnir took an interest in this work and contributed to its current form. And without the support of Austa, Gunnar, Medina and my parents Tom and Arline, it is very unlikely I would have completed my doctorate at all.

Vita of William Lyons

Education

Ph.D. Mathematics, emphasis in Computational Science and Engineering	
University of California, Santa Barbara.	2005
M.A. Mathematics, University of California, Santa Barbara.	2002
B.A. (Hons) Theoretical Physics, Trinity College Dublin, Ireland.	1998

Professional Employment

Teaching Assistant, UCSB.	2000–2005
Analyst, Santa Barbara Market Neutral Fund.	2001–2002
Executive, Investment Banking Division, Deutsche Bank.	1998–1999

Publications

Fast LU decomposition for operators with hierarchically semiseparable structure, (with S. Chandrasekaran and M. Gu), Preprint UCSB Math 2005-9.

Fast algorithms for spectral collocation with non-periodic boundary conditions, (with H. D. Ceniceros, S. Chandrasekaran and M. Gu), Journal of Computational Physics, Volume 207 (1), Pages 173-191, July 2005.

A fast and stable adaptive solver for hierarchically semi-separable representations, (with S. Chandrasekaran and M. Gu), Preprint UCSB Math 2004-20.

Awards

UCSB Math Department Fee Fellowship	2002–2005
UC Regents Special Fellowship.	2001–2002

Abstract

Fast Algorithms with Applications to PDEs

by

William Lyons

In the first part of this thesis we describe the *rebus* representation of a linear operator and present algorithms for working with linear systems in this form. The main contributions in the area of algorithms are: a rebus-rebus multiplication, an LU factorization and forward and back substitution algorithms. All of these are described in detail. Implementations of the algorithms are tested and shown to be accurate and stable. The algorithms scale linearly in the size of the rebus and break even with high-performance dense routines for matrix sizes of $O(100)$. These are, to our knowledge, the first algorithms proposed for these operations on a rebus representation.

In the second part of the thesis we apply rebus-based methods to a recurring problem in the numerical solution of partial differential equations. Problems discretized by collocation on the Chebyshev nodes give rise to large non-sparse matrices. Conventionally, a transform to a spectral domain is used to make the differential operator sparse. However, this imposes periodic boundary conditions on the problem, which may not be desirable. The main contribution in the area of numerical solution of PDEs is a procedure for using fast operations to solve problems of this kind with nonperiodic boundary conditions. We implement a fast solver for linearly implicit methods for stiff equations. We describe in detail how different boundary conditions can be imposed and apply the techniques to the Allen-Cahn equation and the diffusion equation.

Contents

Chapter 1

Introduction

1.1 Overview

In this thesis we will explore algorithms and applications relating to a novel representation of linear operators: the *rebus*. The rebus structure originates with Chandrasekaran and Gu [18] and is related to the fast multipole methods (FMM) of Greengard and Rokhlin [26]. At the time of its introduction, the only algorithms available for working with an operator or linear system in this form were a "matrix-vector" product (to apply a rebus to a vector) and a direct solver, based on the ULV decomposition.

In Chapter 3 we introduce new algorithms that add to the utility of this representation. An algorithm to compose two operators in rebus form (analogous to matrix-matrix multiplication) is described and proven to be correct. An implementation of the algorithms is also shown to be accurate, stable and fast. This algorithm is described in Section 3.2.

This rebus-rebus multiplication, as well as providing a standard tool of linear algebra, helps us avoid one of the dominant costs of rebus methods. The construction of the rebus from a description of the operator involves repeated singular value decompositions (SVDs). These factorizations quickly become the dominant cost for large problems of any kind involving rebus methods. Provided we can compose operators, add and subtract and multiply by scalars we have enough algebraic structure to generate most needed operators from a pre-computed set of “building blocks”. For example, if we have the derivative operator available in rebus form, all the operators needed for a Runge-Kutta timestep for a second order differential equation may be generated using the fast algorithms without any SVDs being used.

The second major algorithm described in this thesis is an LU decomposition. Given a rebus R, we wish to generate upper and lower triangular rebuses such that $L \cdot U = R$. This is important as it greatly increases the speed for solving against multiple right hand sides via fast forward and back substitution. The LU decomposition is described in Section 3.3 and the forward and back substitutions are described in Section 3.4.

All of these algorithms are fast, in the sense that the time taken to execute them grows linearly with the size of the system. A conventional dense matrix LU or matrix multiplication scales as N^3, where N is the size of the discretization. For large systems rebus methods are orders of magnitude faster than dense matrix methods. The break-even point for the experiments performed was found to be for discretizations where N is a few hundred. The overhead of rebus methods is not worth paying for small or medium sized systems. However, the methods provide a new approach for large problems that would previously have been intractable.

In the second part of the thesis, we apply these fast algorithms to construct solvers for partial differential equations (PDEs). Unlike conventional fast solvers which rely on transformation to a spectral domain, the fast solvers based on the rebus structure keep the representation of the operator in the spatial domain. Thus, unlike fast Fourier methods which force us to work with periodic boundary conditions, these methods allow us to use Dirichlet, Neumann or mixed boundary conditions.

In Chapter 4 we demonstrate the utility of such solvers in the context of nonlinear stiff PDEs. For such problems, it is in our interest to use an implicit scheme for timestepping, so as to avoid severe stability restrictions on the allowed step size. However, the nonlinear term is very difficult and costly to invert, and in general is better treated explicitly. A number of methods have resulted from these observations and the most popular is the linearly implicit (also called implicit-explicit [2] family of methods [11]. We demonstrate a fast solver for this kind of scheme and demonstrate its effectiveness by solving the diffusion equation and the Allen-Cahn equation with various nonperiodic boundary conditions. The solver is shown to be accurate, stable and fast.

In Chapter 5 we consider some important auxiliary topics: the conditioning of the Chebyshev derivative operator, the ramifications of this for numerical methods and the extension of the methods presented to higher-dimensional problems. Finally, in Chapter 6 we discuss some fruitful topics for further research. In particular we are able to demonstrate applications to the calculation of matrix exponentials. Amongst other applications, this can be used to construct fourth-order accurate timestepping schemes using exponential time differencing methods.

In the remainder of this chapter, we cover some preliminary topics. In Section 1.2 we define semiseparable structure and consider the relationship between matrices with low-rank blocks and their inverses. In Section 1.4 we demonstrate that by block partitioning matrices we can reveal low-rank structure that is not usually accessible. This motivates a representation for the operators where we can take advantage of this by factoring certain blocks. This is one of the ideas that will lead to the introduction of the rebus in Chapter 2. In Section 1.5, we survey the history of the rebus, comparing it to ideas from which it has evolved.

1.2 Low-Rank Blocks and Semiseparable Structure

As noted by Golub and Van Loan [25], good matrix algorithms rely on exploiting structure. The techniques employed in this thesis rely on the fact that there may be exploitable structure in matrices even when none is apparent. Consider the tridiagonal matrix

$$T = \begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ c_2 & a_2 & b_2 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & c_n & a_n \end{pmatrix}. \tag{1.1}$$

Such a matrix is specified by $3n-2$ parameters. Thus, if T is invertible, the matrix T^{-1} is also completely determined by $3n-2$ parameters. However, the matrix T^{-1} is not sparse and must have full rank (since it is invertible). Thus a different kind of structure must be exploited to represent it in terms of these parameters.

In fact, T^{-1} will have the following structure:

$$(T^{-1})_{i,j} = \begin{cases} u_i v_j \text{ if } j \geq i \\ s_i t_j \text{ if } j \leq i \end{cases}, \tag{1.2}$$

where the factors must obey the constraints that $u_i v_i = s_i t_i$ and $u_1 = s_1 = 1$. Thus we have $n+2$ constraints on our $4n$ parameters u, v, s and t, leaving us with $3n-2$ free parameters.

This kind of structure arises in many different fields, such as integral equations and statistics. It is a specific case of “semiseparable” structure, which is more generally defined as follows.

Definition A matrix S is called a lower (or upper) *semiseparable* matrix of semiseparability rank r if all submatrices that can be taken out of the lower (upper) triangular part of the matrix S have rank $\leq r$ and there exists at least one submatrix having rank r.

Strang and Nguyen give good survey of the connections between low-rank blocks of matrices and their inverses in [40]. They also prove the following extension of the result that the inverse of a tridiagonal matrix is semiseparable of semiseparability rank 1. We state it here in terms of the submatrices *above* a certain diagonal. We will refer to the pth band parallel to, and above, the diagonal as the pth superdiagonal. The analogous results for submatrices below the diagonal, and subdiagonal bands also holds.

Theorem 1.2.1 (Strang and Nguyen) *For invertible T, all submatrices B above the pth superdiagonal of T have* $\operatorname{rank} B < k$ *if and only if all submatrices C above the pth subdiagonal of T^{-1} have* $\operatorname{rank} C < p + k$.

The meaning of this theorem becomes more clear if we consider a few simple examples. The case $p = 0$ and $k = 1$ gives: all submatrices above the diagonal of T have rank zero if and only if all submatrices of T^{-1} above the diagonal have rank zero. This is just the well known fact that the inverse of a lower triangular matrix is itself lower triangular. The case $p = 1$ and $k = 1$ corresponds to our earlier tridiagonal matrix (in the two-sided case). For this, the result is that all submatrices of T above the first diagonal have rank zero if and only if all submatrices of T^{-1} above the first subdiagonal have ranks strictly less than 2.

If we keep $k = 1$, we get the general result for banded matrices where all entries after the pth off–diagonal are zero. The inverse of a banded matrix of bandwidth p is a matrix such that all submatrices above the pth subdiagonal (and also all submatrices below the pth superdiagonal) have rank less than or equal to p. This connection was apparently first discovered by Asplund in 1959 [4].

The whole theorem may be understood as a combination of this banded case with the Woodbury-Morrison formula for the effect of a low-rank update of a matrix on its inverse [25].

1.3 Extensions of Semiseparability

1.3.1 Semiseparable Plus Structure

Theorem 1.2.1 relates the semiseparable structure of a matrix and of its inverse. The idea of semiseparable structure has been generalized in a number of papers. Frequently the structures studied take the form of "semiseparable plus" structure, where the matrix may be expressed as the sum of a semiseparable part and a second part that might be banded, block-diagonal, Toeplitz or some other structure. Example of this kind include treatments of banded plus semiseparable matrices [15] and symmetric block-diagonal plus semiseparable matrices [16].

A more recent development was the introduction of "sequentially semiseparable" structure, which is a direct precursor of the hierarchically semiseparable structure explored in this thesis [14].

1.3.2 Continuous and Block Matrix Analogues

There are two important generalizations of these results that must be made before they can be applied to most practical problems.

1. Extend scalar entries to block matrices.
2. Extend discrete matrices to continuous kernels.

Strang and Nguyen mention both of these points in their survey [40]. In fact, the proof they present for Theorem 1.2.1 extends directly to the important block matrix case.

The extension to (discretizations of) continuous kernels arises in equations such as

$$u(x) = \int K(x,y) f(y)\,\mathrm{d}y \quad , \tag{1.3}$$

where $K(x,y)$ is the kernel corresponding to an underlying differential equation (the Green's function). For such kernels, we have the following correspondences:

- A kernel which decays rapidly away from the line $x = y$ corresponds to a matrix with low bandwidth.
- A kernel that is smooth (in the sense of not having high frequency oscillations) corresponds to a matrix with low-rank off-diagonals.

As these are approximate results, the interpretation needs a little more refinement. In fact, the discretization of a smooth kernel will in general not have low-rank off-diagonal blocks. However, it will be accurately approximated by a matrix that does. That is, it will have low *numerical rank*.

Recall that where the rank of a matrix is given by the number of non-zero singular values, the numerical rank, to a given tolerance ε , is the number of singular values strictly greater than ε . The magnitude of the singular values scales with the norm of the matrix, so in order for the measure to be meaningful it is usual to adopt a convention whereby the matrix is scaled so as to have a largest singular value of order 1. We will use the term "low numerical rank" qualitatively to mean that most of the singular values are many orders of magnitude smaller than the norm of the matrix.

Similarly, a rapidly decaying kernel will generally not yield a discretization that is banded. However, it will be well approximated by such a matrix. Clearly, the lower the tolerance allowed in the approximation, the higher the resulting bandwidth will be.

1.4 Low-Rank Structure

The above sections deal with special classes of matrices. However, the real utility of the methods to be developed comes from the observation that exploitable structure of a semiseparable kind occurs far more frequently than might be expected. In particular, most differential operators will demonstrate a high degree of this structure.

This is not to say that the differential operators themselves are semiseparable.

They are not. However, large blocks of the matrices representing these operators do have low numerical rank. As we will show in Section 2.2, this is one of the properties which is exploited by the rebus structure, though it is not the whole story.

To show the extent to which we may expect to realize savings in storage and number of operations, we will look at the rank structure of various matrices. In particular we will recursively block partition some matrices and look at the ranks of submatrices of various sizes.

1.4.1 Examples of Low-Rank Structure

Extreme cases

To set the baseline for looking at rank structures we will first look at the two extreme cases: a matrix where every block is full rank and a matrix where every block has rank 1.

For an example of a matrix where every block is full-rank, we will use a matrix where each matrix element is independently randomly generated. Such a matrix is shown in Figure 1.1 where the level of grey represents the magnitude of the matrix element. Every block of such a matrix should have full rank.

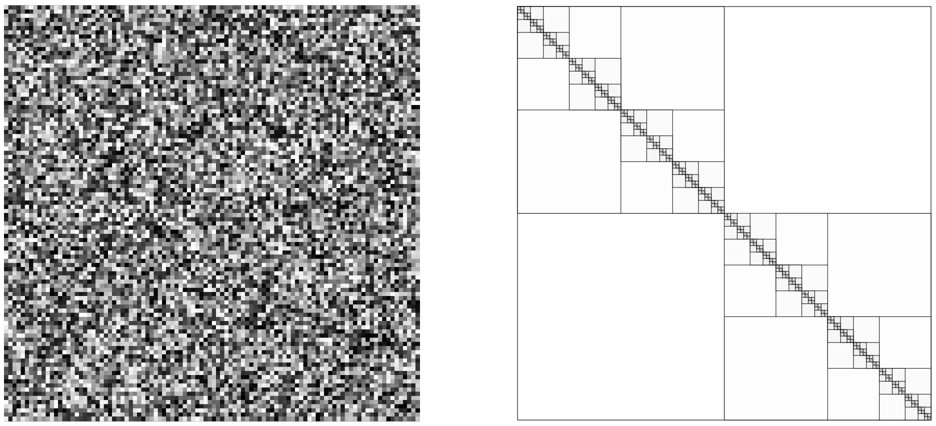

Figure 1.1. A matrix of random entries and a recursive partitioning.

Discretization of a $1/r$ kernel

An interesting example of the low-rank off-diagonal blocks we wish to exploit is found in the context of particle interaction. Consider a matrix which is a discretization of the kernel

$$f(x, y) = \frac{1}{\|x - y\| + \varepsilon}.$$

Where ε is a small parameter added to ensure that we do not run into problems with singularities.

A discretization of such a kernel in one dimension will lead to a matrix with elements

$$a_{i,j} = \frac{1}{\|x_i - x_j\| + \varepsilon}.$$

Such a matrix will have entries that are smooth with respect to the indices i and j. Additionally, away from the diagonal, the entries will fall quickly towards zero.

Figure 1.2 shows two different pictures of such a discretization. The picture to the left shows the value of each matrix element (in a 100×100 example). The white diagonal corresponds to the maximal value the elements attain. The grey which dominates the majority of the picture represents the near-zero elements

The right hand picture illustrates the data in Table 1.1. This shows the ranks in successive off-diagonal blocks of a 1024×1024 discretization of the same kernel. For each N which corresponds to a block size in our recursive partitioning, the table lists the actual rank of the blocks of that size. Thus, the two blocks of size 512×512 each have rank 7 (to machine tolerance). These could be represented and stored in a factored form to achieve large saving over storing all 2^18 matrix elements. This is pictured by the shading of each block corresponding to the relative rank of that block. That is, a full-rank block (of whatever size) is pictured as black and a block that is 1% of full rank is pictured as 1% grey.

Table 1.1 clearly shows that the ranks grow much more slowly than the block size.

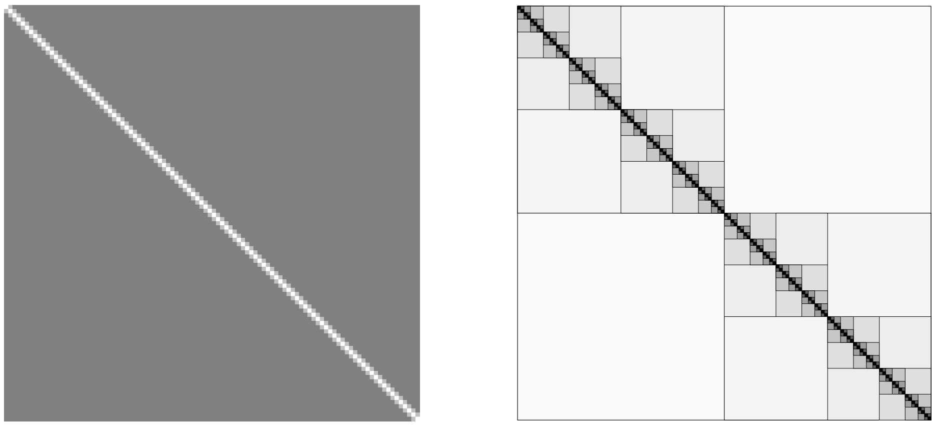

Figure 1.2. A discretization of $1/r$ and a recursive partitioning with shading of each block proportional to the rank of that block as a percentage of the size of the block.

N	Rank of $N \times N$ block	% of full rank
512	7	1.4
256	7	2.7
128	6	4.7
64	5	7.8
32	5	16
16	4	25
8	4	50
4	3	75

Table 1.1. Rank structure of the 1024×1024 discretization of the $1/r$ kernel.

Chebyshev derivative matrix

Consider the hierarchical partitioning of a 1024 by 1024 Chebyshev derivative matrix illustrated in Figure 1.3. It is a special case of the partitioning described in Section 2.2. In this example each partition takes exactly half the rows or columns.

In Figure 1.3 and Table 1.2 we demonstrate this by showing the ranks of various off-diagonal blocks of the first derivative matrix for a Chebyshev pseudospectral method. Note that the ranks increase very slowly as the block size increases.

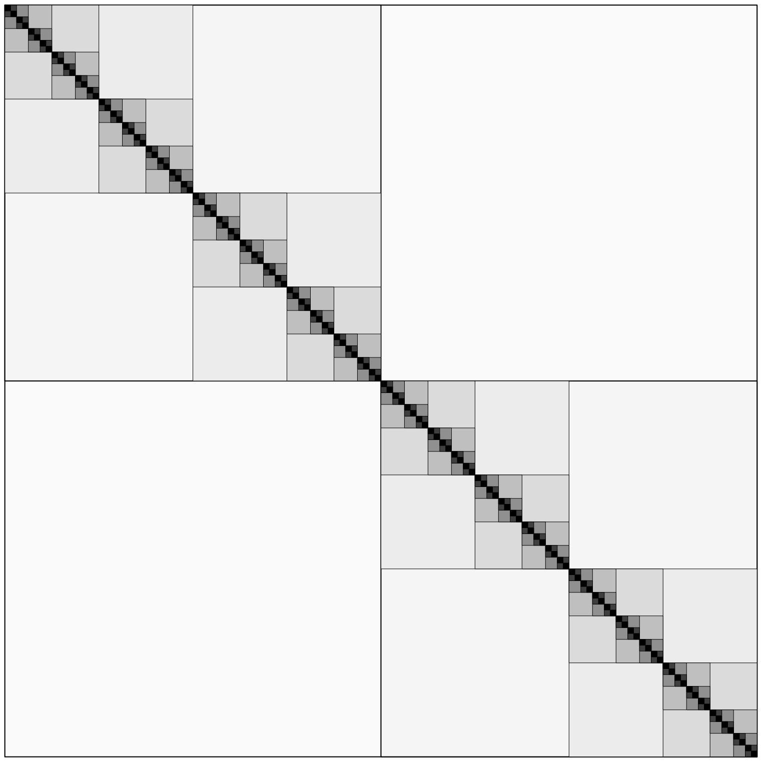

Figure 1.3. Rank structure of the Chebyshev derivative matrix with shading proportional to rank.

N	Rank of $N \times N$ block	% of full rank
512	11	2.1
256	10	3.9
128	10	7.8
64	9	14
32	8	25
16	7	44
8	6	75
4	4	100

Table 1.2. Rank structure of the 1024×1024 Chebyshev derivative matrix.

In Chapter 5, we will demonstrate that this same structure persists not just in the derivative operator itself, but in all of the operators required to numerically solve PDEs.

1.5 Connections to Earlier Work

Although the rebus methods we will develop in this thesis are new, the structure they exploit has been noted before. A number of representations and algorithms have arisen to take advantage of the same properties.

We will briefly compare and contrast the current approach with others which rely on the same principles. We will try to emphasize the differences between each approach and the one advocated here.

1.5.1 Tree Codes

For N-body problems of the type encountered in astrophysics, computational chemistry, plasma physics and other fields relying on particle simulations, the cost of evaluating the mutual interactions grows as N^2 and realistic problems are frequently intractable.

In response to this, a number of procedures were created to reduce the burden of these calculations. Notable amongst these were the Barnes-Hut tree code [6], Appel's method [1] and particle-in-cell methods and their descendants, the particle-particle, particle-mesh methods [29].

The underpinning of these methods was the observation that although short range interactions can be arbitrarily complex, the long range effect of a cluster of particles will be relatively smooth. An alternative viewpoint is that the matrix of the interaction will have low-rank away from the diagonal or will contain off-diagonal blocks that can be accurately approximated by low-rank matrices.

Structured low-rank representations lie behind the tree codes and are rediscovered periodically. They are methods for fast summation and all of the applications rely on the fast evaluation of matrix-vector products. In N-body particle interaction, we are summing the contributions to the potential from N sources and the application is direct.

An extension of fast summation is the solution of linear systems. By using conjugate gradient or other iterative methods, the rate-determining step in the solution of a linear system becomes the rapid evaluation of matrix-vector products. Thus tree codes may be used to solve linear systems, notably those arising from partial differential equations.

1.5.2 Fast Multipole Methods

The fast multipole method (FMM) was originally introduced to solve integral equations [36]. A different formulation was used as a fast summation method [26] and it is this algorithm that had the most impact and is generally known as the FMM.

The work of Rokhlin and collaborators on FMM structure resulted in a rich complex of ideas dealing with much more than fast summation. After the initial application to integral equations, it was applied to evaluating conformal mappings [35], two-point boundary value problems [39], ordinary differential equations [38], 2-dimensional integral equations in scattering theory [37], the wave equation [19] and Laplace's equation [30].

Of particular interest to us here is Rokhlin and Starr's work presented in [38] and [39]. Here the authors use a recursive partitioning to solve an integral equation. The apparatus developed is analytic in nature and specific to integral equations. The development demonstrates that integral equations can be solved efficiently using a recursive partitioning. The off-diagonal blocks in this partitioning are shown to have low rank. However, unlike the partitioned low-rank representations in Section 1.5.1, the representations in each block are not independent, but are related in a multilevel framework, where information may be projected or interpolated between fine and coarse scales.

The twin ideas of a partitioned SVD representation and a full FMM interaction tree arise again in Rokhlin and Yarvin's work [43].

These ideas are further developed by Beylkin, Coult and Mohlenkamp in [9], where the authors use a recursive block decomposition to enable them to work efficiently with spectral projection operators. This is the same block decomposition used in Rokhlin's work that also arises in rebus methods. They prove spectral projection operators to have a compact representation in terms of low-rank blocks. Further, they develop an efficient algorithm to multiply together two matrices stored in this form. The algorithm used is similar to that employed for multiplying in non-standard wavelet form [24].

1.6 Rebus Methods

Our motivation in developing rebus-based methods was that all of these applications of the FMM were exploiting the same underlying structure. Although the expansions and representations used in the FMM papers cited above were problem specific, the underlying ideas were consistent. By confining ourselves to a specific recursive block partitioning we further develop the ideas of the FMM to allow an efficient representation of differential and integral operators in general, as opposed to treating specific operators.

By this restriction to a specific FMM tree (which is what a rebus is), we lose the flexibility of the full FMM but it becomes possible to develop a full algebra. Thus we can implement a fast rebus-vector multiplication, rebus-rebus multiplication, perform LU factorization and code direct solvers. The rebus structure is a hybrid, retaining enough FMM structure to design fast algorithms but simplifying matters enough to make algorithms algebraically tractable.

All the algorithms developed to operate on this structure share some desirable features. They are intrinsically parallel since calculations happen locally and are propagated in discrete stages through the tree. The algorithms are also intrinsically multiresolution. All have recognizable "upsweep" and "downsweep" recursions corresponding to interpolating or projecting information to a different scale. All are easily formulated as adaptive algorithms, where refinement is performed only locally and as needed. An example of this in the case of the solution of linear systems is found in Chandrasekaran [17].

Chapter 2

Hierarchically Semiseparable Structure and the Rebus Representation

2.1 Introduction to HSS Structure

In this chapter we will outline the fundamental operations on the representation of dense, structured matrices with what is known as *hierarchically semiseparable* (HSS) structure [18]. Matrices which have low numerical rank off-diagonal blocks may be efficiently represented by a binary tree of low-rank matrices, related by translation operators. This idea in its current form was introduced in [18], where the algorithms for matrix-vector multiplication and solving a linear system using an implicit ULV factorization were also described.

The data structure used to exploit the HSS structure of a matrix is also known as a *rebus*. The terms are used somewhat interchangeably, but we will try to refer to "HSS structure" when referring to the underlying property of the operator that allows efficient representation and use "HSS representation", "rebus representation" or simply "rebus" to mean the representation of the operator in terms of low-rank products and translation operators. Thus we can talk about the rebus-rebus product corresponding to the composition of two operators having HSS structure. The same composition could be represented by a matrix-matrix product.

Using the rebus representation of an operator, we are able to use tree-based algorithms to exploit structure that is not accessible to us under ordinary circumstances. By designing algorithms with reference to the underlying tree structure, large speed improvements are realized for all standard matrix operations.

It should be noted that although a general matrix will not possess the low-rank off-diagonal blocks that these algorithms exploit, dense matrices arising from physical or statistical problems in general will. In particular, discretizations of integral and differential equations will exhibit HSS structure. All sparse matrices also have a simple HSS representation.

2.2 Fundamentals of the Rebus Representation

2.2.1 Description of the Rebus

A rebus representation is based on a hierarchical block structure and may be described in terms of block partitioning. Given any dense matrix we consider it as a block 2×2 matrix where the blocks are of arbitrary size and in particular need not all be the same size.

We use the notational device of preceding the usual positional subscripts with the "level" of splitting, so the original dense matrix A can be thought of as $A_{0;11}$. That is, the $(1,1)$ block of the matrix partitioned zero times. Using this notation and block partitioning we get

$$A_{0;11} = \begin{pmatrix} A_{1;11} & A_{1;12} \\ A_{1;21} & A_{1;22} \end{pmatrix}.$$

The off-diagonal blocks are factored and stored as

$$A_{1;ij} = U_{1;i} B_{1;ij} V_{1;j}^H \quad \text{for } (i,j) = (1,2), (2,1),$$

where V^H denotes the Hermitian conjugate of V.

The notation here should be reminiscent of that conventionally used for the singular value decomposition (SVD), $A = U \Sigma V^H$. The purpose of the factorization is to allow us to take advantage of rank deficient blocks. However, it is in fact *not* a simple SVD, as the global structure places constraints on which factors are allowable.

We repeat the above process for each of the diagonal blocks. That is, we subdivide $A_{1;11}$ and $A_{1;22}$ to get

$$A_{1;11} = \begin{pmatrix} A_{2;11} & A_{2;12} \\ A_{2;21} & A_{2;22} \end{pmatrix}$$

and

$$A_{1;22} = \begin{pmatrix} A_{2;33} & A_{2;34} \\ A_{2;43} & A_{2;44} \end{pmatrix}.$$

The off-diagonal blocks of these matrices are again factored in the form

$$A_{2;ij} = U_{2;i} B_{2;ij} V_{2;j}^H \quad \text{for } (i,j) = (1,2), (2,1), (3,4), (4,3).$$

The new diagonal blocks are again split, and this process continues down to some lowest level where they are simply stored as dense matrices.

For an appropriate factorization, this will enable us to use a low-rank representation of the off-diagonal blocks. The resulting block structure is illustrated in Figure 2.1. The benefit of this structure for representing differential operators is demonstrated in Section 1.4.

Together with the (low-rank) representations of the off-diagonal blocks, there is a tree structure as follows: we do not store the factors $U_{k;i}$ and $V_{k;i}$ at every level. Instead, we store them at some lowest level only, and then store intermediate quantities $R_{k,i}$ and $W_{k,i}$ which satisfy the relationships

$$U_{k-1;i} = \begin{pmatrix} U_{k;m} R_{k;m} \\ U_{k;n} R_{k;n} \end{pmatrix}, \qquad V_{k-1;i}^H = \begin{pmatrix} W_{k;m}^H V_{k;m}^H & W_{k;n}^H V_{k;n}^H \end{pmatrix}, \qquad (2.1)$$

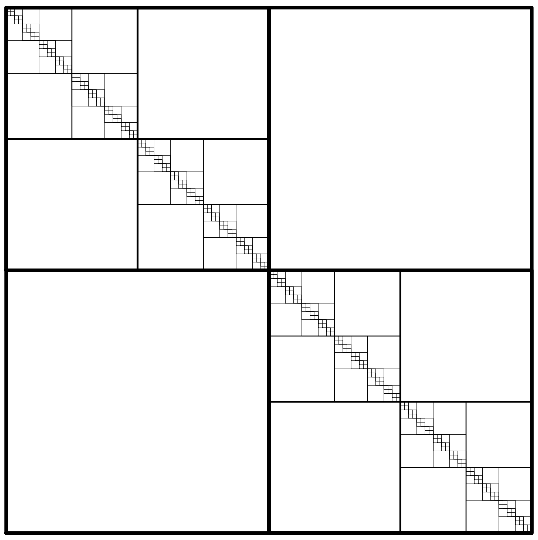

Figure 2.1. Block structure used to generate rebus representation.

where level $k-1$ is the parent of level k. This allows us to store only the "lowest level" factorizations and small transformations which relate them to the factors at the next highest level.

To ensure that this is possible, we must relate the factorizations at each level of the rebus. It is for this reason that we cannot simply take a singular value decomposition of each block at each level. The $U_{k;n}$ at the lowest level must not only provide a basis for the column space of $A_{k;nm}$, it must provide a basis for the column space of the whole row n of A_k (excluding the diagonal block).

Once we have performed this recursive splitting and factorization we obtain the components of the rebus: the diagonal blocks D_k, the lowest level factors U_k and V_k and a binary tree of low-rank factors R, W and B. This representation allows us to efficiently recover the original matrix, while exposing any low-rank structure.

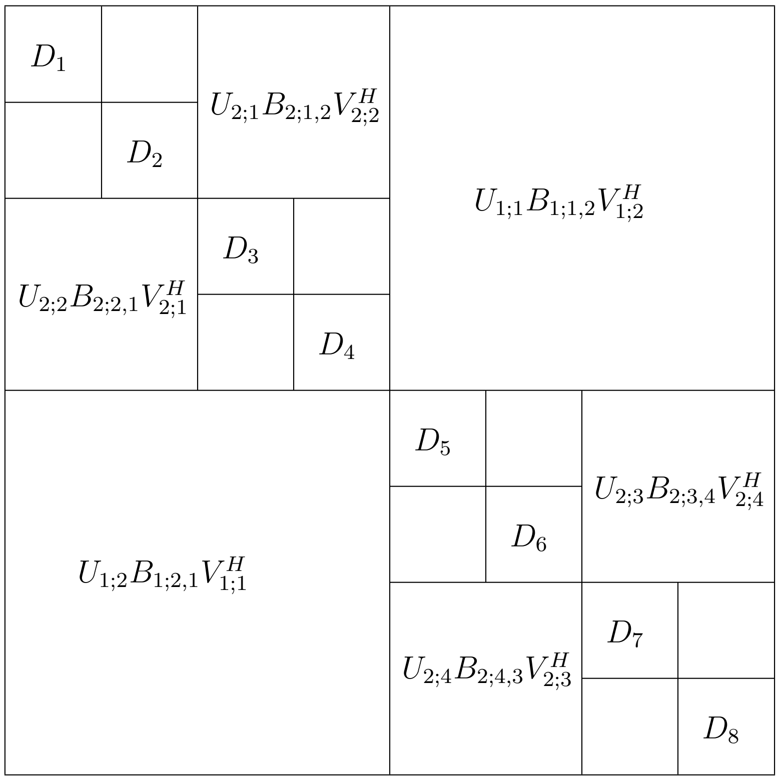
D_1
$U_{2;1}B_{2;1,2}V_{2;2}^H$
D_2
$U_{1;1}B_{1;1,2}V_{1;2}^H$
D_3
$U_{2;2}B_{2;2,1}V_{2;1}^H$
D_4
D_5
$U_{2;3}B_{2;3,4}V_{2;4}^H$
D_6
$U_{1;2}B_{1;2,1}V_{1;1}^H$
D_7
$U_{2;4}B_{2;4,3}V_{2;3}^H$
D_8

Figure 2.2. How the factors of the rebus correspond to matrix blocks. The matrices $U_{k;i}$ and $V_{k;i}$ are not explicitly stored as part of the rebus. They are related to $U_{K;i}$ and $V_{K;i}$ via Equation 2.1.

2.3 Lemmas Concerning Rebus Structure

Given a rebus, it may be desirable to find a different rebus for the same underlying operator. In particular, given a block $2^k \times 2^k$ rebus we would like formulas for block $2^{k-1} \times 2^{k-1}$ and block $2^{k+1} \times 2^{k+1}$ representations. It is worth bearing in mind that these have a natural interpretation in terms of the tree: the block $2^{k-1} \times 2^{k-1}$ corresponds to merging the lowest level of the tree into the second lowest and the block $2^{k+1} \times 2^{k+1}$ corresponding to splitting the leaves of the tree and so adding another level to our binary tree.

These transformations of the tree are also useful in deriving rebus algorithms, as we have a family of transformations of the tree that are known to represent the same operator.

2.3.1 Merging

Lemma 2.3.1 *If a rebus is block partitioned as a $2^K \times 2^K$ we can generate the block $2^{K-1} \times 2^{K-1}$ rebus as follows.*

$$\begin{aligned} D_{K-1;i} &= \begin{pmatrix} D_{K;m} & U_{K;m} B_{K;m,n} V_{K;n}^H \\ U_{K;n} B_{K;n,m} V_{K;m}^H & D_{K;n} \end{pmatrix}, \\ U_{K-1;i} &= \begin{pmatrix} U_{K;m} R_{K;m} \\ U_{K;n} R_{K;n} \end{pmatrix}, \\ V_{K-1;i}^H &= \begin{pmatrix} W_{K;m}^H V_{K;m}^H & W_{K;n}^H V_{K;n}^H \end{pmatrix}, \end{aligned}$$

and $B_{k;i,j}$, $R_{k;i}$ and $W_{k,i}$ remain unchanged for $k \neq K$ and are no longer explicitly needed for $k = K$.

Proof Viewed in terms of the quantities

$$\begin{aligned} U_{k-1;i} &= \begin{pmatrix} U_{k;m}R_{k;m} \\ U_{k;n}R_{k;n} \end{pmatrix} \qquad, \\ V_{k-1;i}^H &= \begin{pmatrix} W_{k;m}^H V_{k;m}^H & W_{k;n}^H V_{k;n}^H \end{pmatrix} \qquad, \end{aligned}$$

for $k = 1 \ldots K$, the only change to the matrix is that the diagonal blocks absorb the first off-diagonal, leaving the rest of the matrix unchanged. By inspection of the original HSS structure, as shown in Figure 2.2, the new diagonals are as claimed. However, the matrices $U_{k;i}$ and $V_{k;i}$ are not stored as part of the HSS structure, except at the lowest level, $U_{K,i}, V_{K;i}$ for $i = 1 \ldots 2^K$. As such we need to use this stored lowest level to generate the matrices at the next level and store these instead. By construction of the rebus, these neighboring levels are related by Equation 2.1, which immediately gives us the result. ∎

2.3.2 Splitting

We have determined how to merge the lowest two levels of our rebus to get a more coarsely blocked rebus. We now turn to the question of obtaining a finer rebus structure. That is, splitting our diagonal and our lowest level leaves in a $2^{K-1} \times 2^{K-1}$ rebus to give a block $2^K \times 2^K$ representation.

Lemma 2.3.2 *If a rebus is block partitioned as a $2^{K-1} \times 2^{K-1}$ we can generate the block $2^K \times 2^K$ rebus as follows. Let*

$$\begin{aligned} D_{K-1;i} &= \begin{pmatrix} D_{K-1;i}^{1,1} & D_{K-1;i}^{1,2} \\ D_{K-1;i}^{2,1} & D_{K-1;i}^{2,2} \end{pmatrix}, \\ U_{K-1;i} &= \begin{pmatrix} U_{K-1;i}^{1} \\ U_{K-1;i}^{2} \end{pmatrix}, \\ V_{K-1;i} &= \begin{pmatrix} V_{K-1;i}^{1} \\ V_{K-1;i}^{2} \end{pmatrix}. \end{aligned} \tag{2.2}$$

Then the rebus structure at the Kth level is:

$$D_{K;m} = D^{1,1}_{K-1;i}, \qquad D_{K;n} = D^{2,2}_{K-1;i},$$

with the remaining components being given by the factorizations:

$$\begin{pmatrix} U^1_{K-1;i} & D^{1,2}_{K-1;i} \\ 0 & V^{2H}_{K-1;i} \end{pmatrix} = \begin{pmatrix} U_{K;m} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} R_{K;m} & B_{K;m,n} \\ 0 & W^H_{K;n} \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & V^H_{K;n} \end{pmatrix}$$
$$\begin{pmatrix} V^{1H}_{K-1;i} & 0 \\ D^{2,1}_{K-1;i} & U^2_{K-1;i} \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & U_{K;n} \end{pmatrix} \begin{pmatrix} W^H_{K;m} & 0 \\ B_{K;n,m} & R_{K;n} \end{pmatrix} \begin{pmatrix} V^H_{K;m} & 0 \\ 0 & I \end{pmatrix}.$$

Proof Having determined the merging formulas in Lemma 2.3.1 we will solve the merging formulas to give Kth level in terms of the larger blocks. We do not expect there to be a unique way to split the blocks, but we will obtain the relationships that any splitting must satisfy and thus identify appropriate factorizations. In order to do this we write the lowest level D, U and V matrices as in Equation 2.2. Combining these formulas with the merge formulas from Lemma 2.3.1 we obtain the relations:

$$\begin{aligned} D_{K-1;i} &= \begin{pmatrix} D^{1,1}_{K-1;i} & D^{1,2}_{K-1;i} \\ D^{2,1}_{K-1;i} & D^{2,2}_{K-1;i} \end{pmatrix} = \begin{pmatrix} D_{K;m} & U_{K;m}B_{K;m,n}V_{K;n} \\ U_{K;n}B_{K;n,m}V_{K;m} & D_{K;n} \end{pmatrix}, \\ U_{K-1;i} &= \begin{pmatrix} U^1_{K-1;i} \\ U^2_{K-1;i} \end{pmatrix} = \begin{pmatrix} U_{K;m}R_{K;m} \\ U_{K;n}R_{K;n} \end{pmatrix}, \\ V_{K-1;i} &= \begin{pmatrix} V^1_{K-1;i} \\ V^2_{K-1;i} \end{pmatrix} = \begin{pmatrix} V_{K;m}W_{K;m} \\ V_{K;n}W_{K;n} \end{pmatrix}. \end{aligned}$$

Where the extreme right hand expressions consist of unknowns and the other terms are known. The first equation allows us to read off

$$D_{K;m} = D^{1,1}_{K-1;i}, \qquad D_{K;n} = D^{2,2}_{K-1;i}.$$

and we are left with six equations in 10 unknowns. This six equations can be compactly written in the form:

$$\begin{pmatrix} U^1_{K-1;i} & D^{1,2}_{K-1;i} \\ 0 & V^{2H}_{K-1;i} \end{pmatrix} = \begin{pmatrix} U_{K;m} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} R_{K;m} & B_{K;m,n} \\ 0 & W^H_{K;n} \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & V^H_{K;n} \end{pmatrix}$$

$$\begin{pmatrix} V^{1H}_{K-1;i} & 0 \\ D^{2,1}_{K-1;i} & U^2_{K-1;i} \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & U_{K;n} \end{pmatrix} \begin{pmatrix} W^H_{K;m} & 0 \\ B_{K;n,m} & R_{K;n} \end{pmatrix} \begin{pmatrix} V^H_{K;m} & 0 \\ 0 & I \end{pmatrix}.$$

where again the terms on the right are unknown and those on the left are known. Thus, a matrix factorization of the form shown allows us to calculate the $2^K \times 2^K$ rebus, given the $2^{K-1} \times 2^{K-1}$ rebus. ∎

Lemma 2.3.3 *It is possible to (nontrivially) factor a block upper triangular matrix into the form*

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ 0 & a_{2,2} \end{pmatrix} = \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} \\ 0 & b_{2,2} \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & Q_2 \end{pmatrix}.$$

Proof First consider a QR factorization of the top row:

$$\begin{pmatrix} a_{1,1} & a_{1,2} \end{pmatrix} = Q_1 \begin{pmatrix} R_{1,1} & R_{1,2} \end{pmatrix}$$

followed by an LQ factorization of

$$\begin{pmatrix} R_{1,2} \\ a_{2,2} \end{pmatrix} = \begin{pmatrix} L_{1,2} \\ L_{2,2} \end{pmatrix} Q_2.$$

Then it follows that

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ 0 & a_{2,2} \end{pmatrix} = \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} R_{1,1} & L_{1,2} \\ 0 & L_{2,2} \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & Q_2 \end{pmatrix}$$

and our result follows with $b_{1,1} = R_{1,1}$, $b_{i,2} = L_{i,2}$. ∎

The factorization for the block lower triangular case follows in the same manner. By using the rank reducing SVD to carry out the component factorizations we can assure that the low-rank blocks will be suitably compressed.

2.4 Block Sparse Notation

A convenient and useful alternative to the triple-indexed notation above can also be employed while working with rebuses. This involves embedding each group of factors of the rebus in a block sparse matrix. In this way, we refer to the entire set of diagonal entries at the kth level, $D_{k;1}, D_{k;2}, \ldots, D_{k;2^k}$, by defining the single-index quantity

$$D_k = diag(D_{k;1}, D_{k;2}, \ldots, D_{k;2^k}) \quad ,$$

as a block sparse matrix. Similarly we define

$$U_k = diag(U_{k;1}, U_{k;2}, \ldots, U_{k;2^k}) \quad \text{and} \quad V_k = diag(V_{k;1}^H, V_{k;2}^H, \ldots, V_{k;2^k}^H)$$

giving us all of the "leaf" variable in this form. Note that we have absorbed the Hermitian conjugate into the block sparse embedding.

A K-level rebus contains the factors $B_{k;ij}$ for pairs of indices

$$(i, j) = (1, 2), (2, 1), (3, 4), (4, 3), \ldots, (2^k, 2^k - 1)$$

for each level $k = 1 \ldots K$. We embed each level k separately, so that the (i, j) block of B_k contains $B_{k;i,j}$ if that matrix is defined and is empty otherwise. This is equivalent to defining

$$B_k = diag(B_{k;1,2}, B_{k;2,1}, \ldots, B_{k;2^k,2^k-1}) \cdot P \quad ,$$

where P is the permutation which is empty except for identity matrices in the first off-diagonal blocks at the kth level. Figure 2.3 illustrates the block pattern.

To complete our new representation of the rebus, we also have block sparse representations of the R and W^H matrices at each level. These are the vertices if the partition tree that connect each level to the levels one higher and one lower. The block structure reflects this. In a full tree, R_k would have 2^k block-rows and 2^{k-1} block-columns. The rows contained in these blocks are determined by the partition tree. The extra partition between rows marks the change between indices which belong to the right child and indices which belong to the left child.

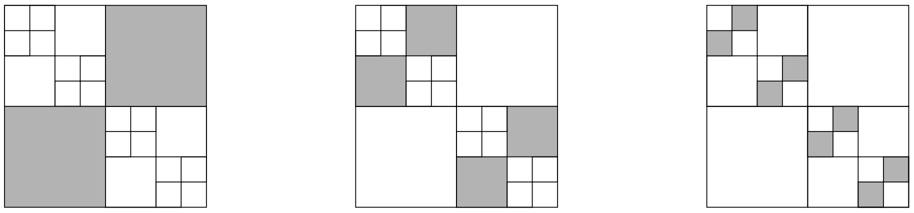

Figure 2.3. The block structures of B_1, B_2 and B_3, all of which are components of a 3-level rebus.

Figure 2.4 illustrates this for a perfectly uniform partitioning. The R_3 matrices shown connect the 2^3 nodes of the third level with the 2^2 nodes of the second level.

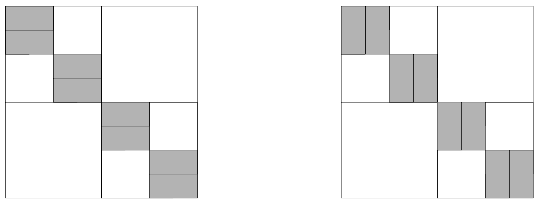

Figure 2.4. The block structure of R_3 and W_3. The shaded areas contain $R_{3;1}$ through $R_{3;8}$ and $W_{3;1}^H$ through $W_{3;8}^H$.

2.4.1 Fundamental Block Sparse Relationships

The advantage of the block sparse notation is that the fundamental operations of merging and splitting and relating the higher-level U and V matrices to those stored on the leaves may be expressed much more succinctly. In addition, the separation of scales it affords gives better insight into the role played by the upsweep and downsweep operations.

Recall from Section 2.2 that the fundamental operations we consider in dealing with the recursive structure of the rebus are:

1. splitting and merging, and
2. relating the off-diagonal factors U and V to the leaf values.

2.4.2 Splitting and Merging

In our new notation, the splitting formula 2.3.2 is expressed as

$$D_k = D_{k+1} + U_{k+1}B_{k+1}V_{k+1} \quad .$$

This may be used recursively, starting with D_0 (the original matrix representation of an operator), to obtain the representation

$$D_0 = D_K + U_K B_K V_K + U_{K-1}B_{K-1}V_{K-1} + \ldots + U_2 B_2 V_2 + U_1 B_1 V_1$$

for an operator as a K-level rebus. Figure 2.5 shows the block patterns of these terms. Note that the term $U_i B_i V_i$ has the same block structure as B_i itself, as U and V are block-diagonal.

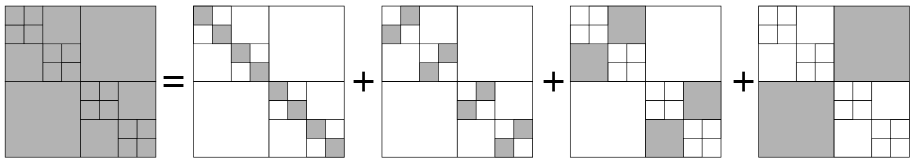

Figure 2.5. The equation $D_0 = D_3 + U_3B_3V_3 + U_2B_2V_2 + U_1B_1V_1$ in pictures.

2.4.3 Projection and Interpolation

The second relationship we rely on expresses U_i in terms of U_K and $\{R_j : j = K, K-1, \ldots, i-1\}$. This is the connection between the different levels of the rebus and allows us to store only the lowest-level U_K.

Using the definition of R_k and W_k above, we see that the factors obey the relationships

$$U_k = U_{k-1}R_{k-1}$$

and

$$V_k = W_{k-1}V_{k-1}.$$

2.4.4 Recursive Representations of a Matrix

Using the equations above, we can derive the following rebus representation for a matrix D_0.

$$\begin{aligned}
D_0 &= D_K + U_K B_K V_K + U_{K-1} B_{K-1} V_{K-1} + \ldots + U_1 B_1 V_1 && (2.3)\\
&= D_K + U_K B_K V_K + U_K R_K B_{K-1} W_K V_K \\
&\qquad + U_K R_K R_{K-1} B_{K-2} W_{K-1} W_K V_K \\
&\qquad + \ldots + U_K R_K R_{K-1} R_{K-2} \cdots R_2 B_1 W_2 \cdots W_{K-1} W_K V_K && (2.4)\\
&= D_K + U_K (B_K + R_K (B_{K-1} + R_{K-1} (B_{K-2} \\
&\qquad + R_{K-2} (\cdots (B_2 + R_2 B_1 W_2) \cdots) W_{K-2}) W_{K-1}) W_K) V_K && (2.5)\\
&= D_K + U_K (B_K V_K + R_K (B_{K-1} W_K V_K \\
&\qquad + R_{K-1} (B_{K-2} W_{K-1} W_K V_K + \cdots + R_2 (B_1 W_2 W_3 \cdots W_{K-1} W_K V_K)))) \\
& && (2.6)
\end{aligned}$$

Now, if we recursively define the quantities

$$G_K = V_K$$

and

$$G_{k-1} = W_k G_k \text{ for } 1 \le k < K,$$

we obtain

$$D_0 = D_K + U_K (B_K G_K + R_K (B_{K-1} G_{K-1} + R_{K-1} (B_{K-2} G_{K-2} + \cdots + R_2 (B_1 G_1)))).$$

These are the "upsweep" recursions, taking information from the leaves, up the tree. Note that, because of the way the low-rank factors are generated, each G_k is low-rank.

We next define the "downsweep" recursion by:

$$F_k = B_k G_k + R_k F_{k-1}$$

and

$$F_1 = B_1 G_1 \quad ,$$

giving

$$D_0 = D_K + U_K F_K \tag{2.7}$$

as an expression of our original matrix, in terms of a low-rank upsweep and downsweep.

It is also useful to introduce another recursive representation. In this version, the quantities on the tree are not generated in such a way as to preserve low-rank. thus, it is not a representation we use in performing actual calculations. However, it is useful in deriving such algorithms.

We start from Equation 2.5:

$$\begin{aligned} D_0 &= D_K + U_K(B_K + R_K(B_{K-1} + R_{K-1}(B_{K-2} \\ &\qquad + R_{K-2}(\cdots(B_2 + R_2 B_1 W_2)\cdots)W_{K-2})W_{K-1})W_K)V_K \qquad (2.8) \\ &= D_K + U_K X_K V_K \qquad (2.9) \end{aligned}$$

where

$$X_k = B_k + R_k X_{k-1} W_k \quad \text{and} \quad X_1 = B_1. \tag{2.10}$$

We will use this form in Section 3.1 to compactly derive the rebus-vector multiplication originally presented in [18].

2.5 Tree-based Structure

In understanding rebus algorithms, it is frequently better to think in terms of the binary tree of factors than to think of the recursive block structure. The tree structure of rebuses is well explained in [18]. In this picture, the rebus is a binary tree of low rank operators, with their position in the tree reflecting whether they influence the operator on a finer or coarser scale. For example, the leaves of the binary tree contain the D_i which are the diagonal, short-range, contributions to the operator.

Figure 2.6 shows the role of a leaf in a rebus algorithm. The leaf's parent passes it an intermediate variable f. The leaf then performs a calculation depending only on f and the leaf's own contents: a single D, u and v from the rebus. The leaf may then produce some output and it also returns its own intermediate variable g back to its parent.

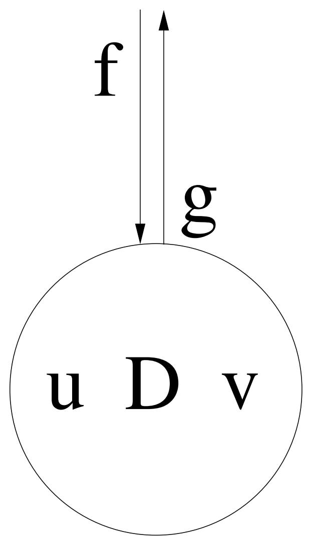

Figure 2.6. Schematic of data flow in a leaf of the rebus.

At a node of the rebus, the process is similar. As before, the node receives an intermediate f and returns its own variable g to its parent. In general, however, this g will depend on the gs of its children. So, using its own internal variables and the f, the node constructs appropriate values f to pass down to its children who will then do likewise until a leaf is reached. When all the calculations "beneath" the node have terminated it can use the returned values of g and its own internal structure to calculate its own g to return to its parent. This is illustrated schematically in Figure 2.7.

The upward propagating gs are what we have been referring to as the "upsweep recursion". Similarly, the downward propagating fs constitute the "downsweep recursion".

These recursions are fundamental to exploiting the tree structure of the rebus in all of the algorithms. Each of these recursions produces a tree of values having the same number K of levels as the rebuses being considered.

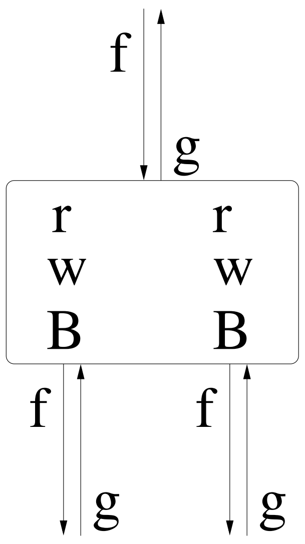

Figure 2.7. Schematic of data flow at a node of the rebus.

The speed and memory efficiency of these rebus algorithms is due to the data locality of the structure. No knowledge of the rest of the computation is needed at any stage other than the f from the parent and the g's of the child nodes.

The order in which these operations happen on the tree depends on the specific algorithm. If the calculation of g does not depend of the parent's f (as is the case in rebus-vector and rebus-rebus multiplication) then the computation can start at the leaves of the tree. In these multiplication algorithms the recursions take the form of one recursion starting at the leaves and propagating information towards the root (the upsweep) and a second which starts at the root and propagates information in the direction of the leaves (the downsweep).

In the LU factorization and forward and back substitution algorithms, however, the computation must start with the root of the tree and travel through the tree in either a left-to-right or right-to-left preorder depth first traversal.

Chapter 3

Algorithms for the Rebus Representation

In this chapter we will describe the fundamental operations in the rebus algebra. These will constitute the foundation for the more complicated applications in Chapter 4. The operations that we will describe here are rebus-vector multiplication (originally described in [18]), rebus-rebus multiplication, LU decomposition and forward and back substitution.

Along with some smaller and more specialized algorithms, the rebus-rebus multiplication allows us to quickly construct complicated rebuses from some basic “building blocks” that can be calculated in advance. Once the linear system is constructed in this form our LU factorization allows its solution.

3.1 Rebus-Vector Multiplication

3.1.1 Algorithm

We wish to evaluate $D_0 \cdot b$, the product of a dense matrix with a vector. Expressing the matrix as a K-level rebus, using Equation 2.10, we obtain the following equations:

$$D_0 \cdot b = D_K \cdot b + U_K X_K V_K \cdot b \tag{3.1}$$

$$\begin{aligned} &= D_K \cdot b + U_K(B_K + R_K(B_{K-1} + R_{K-1}(B_{K-2} \\ &+R_{K-2}(\cdots(B_2 + R_2 B_1 W_2)\cdots)W_{K-2})W_{K-1})W_K)V_K \cdot b \end{aligned} \tag{3.2}$$

$$= D_K \cdot b + U_K F_K. \tag{3.3}$$

Where F is defined via the upsweep and downsweep recursions:

$$F_k = B_k G_k + R_k F_{k-1} \quad \text{and} \quad F_1 = B_1 G_1 \tag{3.4}$$

where

$$G_K = V_K \cdot b \quad \text{and} \quad G_{k-1} = W_k G_k. \tag{3.5}$$

3.1.2 Comments

The simplest operation that we can implement using the rebus is the application of a linear transformation to a vector. This is a fundamental operation and has more applications than the immediately obvious. For example, in an iterative solve of conjugate-gradient type, the actual calculation of the solution consists of a sequence of linear transformations of vectors. Thus, even with this first algorithm, we have the necessary equipment to tackle an iterative solver of linear systems.

3.2 Rebus-Rebus Multiplication

In practical implementations of rebus-based methods, the first issue that will arise is how to get the operator you want to work with into rebus form. One option is to form the dense matrix representation of the operator and then use a construction algorithm (such as in the paper [18]) to generate the corresponding rebus. For very large matrices that may not fit in memory, we can generate the dense representation of various sub-blocks sequentially, and build the rebus from these.

The construction algorithms are based on singular value decompositions and are costly to execute. For many problems, the construction of the rebus representation will be the dominant cost in the entire solution. It is therefore in our interest to precompute a number of useful rebuses to use as building blocks for generating a much larger set of rebuses using fast operations. In this context, being able to compose operators in rebus form is critical to the overall usefulness of the rebus in practical computations.

In this section we will describe the rebus multiplication algorithm, provide a proof of the formulae presented, and report on numerical experiments for rebuses of different sizes and rank structures.

3.2.1 Algorithm

We begin by defining the recursions for the intermediate variables used in the computation. We then proceed to describe an algorithm for computing the product and prove that the rebus so generated corresponds to the product we seek.

Definition For a $2^K \times 2^K$ rebus multiplication, we define the *upsweep* recursion as:

$$g_{K;i} = V_{K;i}^H(A)U_{K;i}(B)$$

for $i = 1, \ldots, 2^K$ and

$$g_{k-1;i} = W^H_{k;n}(A) g_{k;n} R_{k;n}(B) + W^H_{k;n+1}(A) g_{k;n+1} R_{k;n+1}(B) \tag{3.6}$$

for $i = 1, \ldots, 2^k$ and $k = K, \ldots, 1$.

It is worth noting that in terms of the matrices $U_{k,i}$ and $V_{k;i}$ this definition may be written as:

$$g_{k;i} = V^H_{k;i}(A) U_{k;i}(B).$$

However since the matrices $U_{k,i}$ and $V_{k;i}$ are not stored as part of the rebus, this does not represent what is actually calculated as accurately as Equations 3.6.

Definition For a $2^K \times 2^K$ rebus multiplication, we define the *downsweep* recursion as:

$$f_{1;i} = B_{1;i,j}(A) g_{1;j} B_{1;j,i}(B)$$

for $(i, j) = (1, 2), (2, 1)$ and

$$f_{k;i} = B_{k;i,j}(A) g_{k;j} B_{k;j,i}(B) + R_{k;i}(A) f_{k-1;\frac{i}{2}} W^H_{k;i}(B)$$

for $i = 1, \ldots, 2^k$, $j = i + 1$ (for i odd) or $i - 1$ (for i even) and $k = 2, \ldots, K$.

Theorem 3.2.1 *The product $C = A \cdot B$, where A and B are rebuses, can be computed through the following relations:*

$$\begin{aligned}
D_i(C) &= D_i(A) D_i(B) + U_i(A) f_{K;i} V_i^H(B), \\
U_i(C) &= \begin{pmatrix} U_i(A) & D_i(A) U_i(B) \end{pmatrix}, \\
V_i(C) &= \begin{pmatrix} D^H_i(B) V_i(A) & V_i(B) \end{pmatrix},
\end{aligned}$$

for $i = 1, \ldots, 2^K$ *and*

$$\begin{aligned}
B_{k;i,j}(C) &= \begin{pmatrix} B_{k;i,j}(A) & R_{k;i}(A) f_{k-1;\frac{i}{2}} W_{k;j}^H(B) \\ 0 & B_{k;i,j}(B) \end{pmatrix}, \\
R_{k;i}(C) &= \begin{pmatrix} R_{k;i}(A) & B_{k;i,j}(A) g_{k;j} R_{k;j}(B) \\ 0 & R_{k;i}(B) \end{pmatrix}, \\
W_{k;i}^H(C) &= \begin{pmatrix} W_{k;i}^H(A) & W_{k;j}^H(A) g_{k;j} B_{k;j,i}(B) \\ 0 & W_{k;i}^H(B) \end{pmatrix},
\end{aligned}$$

for $k = 1, \ldots, K$ *and* $i = 1, \ldots, 2^k$.

Proof Let us proceed by induction. We will assume that the formulas given give a rebus for the product for all block $2^{k-1} \times 2^{k-1}$ rebuses. Then we will use the block merging and splitting formulas to deduce that the relations will also hold for the block $2^k \times 2^k$ rebuses. The base case will be established by direct computation.

Let $A \cdot B = C$ where A, B and C are matrices and assume that the above formulas give the block $2^{k-1} \times 2^{k-1}$ rebus of C in terms of the block $2^{k-1} \times 2^{k-1}$ representations of A and B.

By this induction assumption, we know that the diagonals of the $2^{k-1} \times 2^{k-1}$ product are given by

$$D_{k-1;i}(C) = D_{k-1;i}(A) D_{k-1;i}(B) + U_{k-1;i}(A) f_{k-1;i} V_{k-1;i}^H(B).$$

which we can expand out in terms of the kth level to give

$$\begin{pmatrix} D_{k;m} & U_{k;m} B_{k;m,n} V_{k;n} \\ U_{k;n} B_{k;n,m} V_{k;m} & D_{k;n} \end{pmatrix}_A \begin{pmatrix} D_{k;m} & U_{k;m} B_{k;m,n} V_{k;n} \\ U_{k;n} B_{k;n,m} V_{k;m} & D_{k;n} \end{pmatrix}_B$$
$$+ \begin{pmatrix} U_{k;m} R_{k;m} \\ U_{k;n} R_{k;n} \end{pmatrix}_A f_{k-1;i} \begin{pmatrix} W_{k;m}^H V_{k;m}^H & W_{k;n}^H V_{k;n}^H \end{pmatrix}_B \quad (3.7)$$

By splitting the blocks in product matrix C we get

$$D_{k-1;i}(C) = \begin{pmatrix} D_{k;m} & U_{k;m} B_{k;m,n} V_{k;n} \\ U_{k;n} B_{k;n,m} V_{k;m} & D_{k;n} \end{pmatrix}_C \quad (3.8)$$

Now, by equating Equation 3.7 and Equation 3.8 and reading off the diagonal for $i = 1, \ldots, 2^k$ (letting $n \to i$) we have

$$\begin{aligned}
D_{k;n}(C) &= D_{k;n}(A)D_{k;n}(B) \\
&\quad + U_{k;n}(A)B_{k;nm}(A)V^H_{k;m}(A)U_{k;m}(B)B_{k;mn}(B)V^H_{k;n}(B) \\
&\quad + U_{k;n}(A)R_{k;n}(A)f_{k-1;i}W^H_{k;n}(B)V^H_{k;n}(B) \\
&= D_{k;n}(A)D_{k;n}(B) \\
&\quad + U_{k;n}(A)(B_{k;nm}(A)g_{k;m}B_{k;mn}(B) \\
&\quad + R_{k;n}(A)f_{k-1;i}W^H_{k;n}(B))V^H_{k;n}(B) \\
&= D_{k;n}(A)D_{k;n}(B) + U_{k;n}(A)f_{k;n}V^H_{k;n}(B).
\end{aligned} \tag{3.9}$$

Our base case is a 2×2 block matrix multiplication. where we immediately have

$$\begin{aligned}
D_{1;i} &= D_{1;i}(A)D_{1;i}(B) + U_{1;i}(A)B_{1;ij}(A)V^H_{1;j}(A)U_{1;j}(B)B_{1;ji}(B)V^H_{1;i}(B) \\
&= D_{1;i}(A)D_{1;i}(B) + U_{1;i}(A)B_{1;ij}(A)g_{1;j}B_{1;ji}(B)V^H_{1;i}(B) \\
&= D_{1;i}(A)D_{1;i}(B) + U_{1;i}(A)f_{1,i}V^H_{1;i}(B)
\end{aligned}$$

for $(i, j) = (1, 2)$ and $(2, 1)$.

For the other relations, consider the off-diagonal block at the $(k-1)$ level:

$$U_{k-1;i}B_{k-1;i,j}V^H_{k-1;j}. \tag{3.10}$$

As before we prove the formulas are valid by induction. Assuming that the multiplication rules hold for a $2^{k-1} \times 2^{k-1}$ matrix, we can expand Expression 3.10 in two ways and then equate these two equivalent terms. Using the splitting formulas (Equation 2.3.2) we see that

$$(U_{k-1;i}B_{k-1;i,j}V^H_{k-1;j})_C = \begin{pmatrix} U_{k;m}R_{k;m} \\ U_{k;n}R_{k;n} \end{pmatrix}_C B_{k-1;i,j} \begin{pmatrix} V^H_{k;\mu}W^H_{k;\mu} & V^H_{k;\nu}W^H_{k;\nu} \end{pmatrix}_C \tag{3.11}$$

and by our induction hypothesis at the $(k-1)$ level we know that if $C = A \cdot B$, the expression in Equation 3.10 also equals the product $\hat{U}\hat{B}\hat{V}$ where

$$\begin{aligned}
\hat{U} &= \begin{pmatrix} U_{k-1;i}(A) & D_{k-1;i}(A)U_{k-1;i}(B) \end{pmatrix} \\
\hat{B} &= \begin{pmatrix} B_{k-1;i,j}(A) & R_{k-1;i}(A)f_{k-2;\frac{i}{2}}W^H_{k-1;j}(B) \\ 0 & B_{k-1;i,j}(B) \end{pmatrix} \\
\hat{V} &= \begin{pmatrix} V^H_{k-1;j}(A)D_{k-1;j}(B) \\ V^H_{k-1;j}(B) \end{pmatrix}
\end{aligned} \tag{3.12}$$

It follows from the equality of Equation 3.11 and the product of the three terms of Equation 3.12 that

$$\begin{aligned}
\begin{pmatrix} U_{k;m}R_{k;m} \\ U_{k;n}R_{k;n} \end{pmatrix}_C &= \begin{pmatrix} U_{k-1;i}(A) & D_{k-1;i}(A)U_{k-1;i}(B) \end{pmatrix} \\
&= \begin{pmatrix} U_{k;m}(A)R_{k;m}(A) & X_m \\ U_{k;n}(A)R_{k;n}(A) & X_n \end{pmatrix}
\end{aligned} \tag{3.13}$$

where

$$X_n = D_{k;n}(A)U_{k;n}(B)R_{k;n}(B) + U_{k;n}(A)B_{k;n,m}(A)g_{k;m}R_{k;m}(B)$$

where we have now applied Equation 2.3.2 to the right hand side. Looking at the formulas for node i we see that at level k we can factorize as:

$$U_{k;i}R_{k;i} = \begin{pmatrix} U_{k;i}(A) & D_{k;i}(A)U_{k;i}(B) \end{pmatrix} \begin{pmatrix} R_{k;i}(A) & B_{k;i,n}(A)g_{k;n}R_{k;n}(B) \\ & R_{k;i}(B) \end{pmatrix} \tag{3.14}$$

Corresponding results hold at node n and analogous calculations go through for V and W. Thus the validity of the expression at level k is established from that at level $k-1$. The base case, again, is a 2×2 block matrix multiplication.

It remains to show the formula given for B is valid.

We have expressions for U and V and we return to the equality of Equation 3.7 and Equation 3.8 and equate the off-diagonal blocks. This gives the equation

$$\begin{aligned}
(U_{k;m}B_{k;m,n}V^H_{k;n})_C &= D_{k;m}(A)U_{k;m}(B)B_{k;m,n}(B)V^H_{k;n}(B) \\
&+ U_{k;m}(A)B_{k;m,n}(A)V^H_{k;n}(A)D_{k;n}(B) \\
&+ U_{k;m}(A)R_{k;m}(A)f_{k-1;i}W^H_{k;n}(B)V^H_{k;n}(B)
\end{aligned} \tag{3.15}$$

and factorizing this in terms of the U and V we have already determined gives us

$$B_{k;m,n}(C) = \begin{pmatrix} B_{k;m,n}(A) & R_{k;m}(A) f_{k-1;i} W_{k;n}^H(B) \\ 0 & B_{k;m,n}(B) \end{pmatrix} \tag{3.16}$$

∎

3.2.2 Numerical Experiments

The cost of computing the rebus-rebus product depends primarily on two factors. One is the size of the equivalent matrix representation. Everything else being equal, if one rebus represents a matrix twice as large as another rebus, we would expect multiplication by the smaller one to be faster. This follows since either each node contains larger matrices, or we have more nodes in the tree. In either case extra computations will be required.

As shown in Section 1.4, there is a typical rank structure for a rebus which represents a differential or integral operator. This consists of full rank blocks up to some point, followed by larger and larger blocks of approximately constant rank.

For these numerical experiments we generated random rebuses with known rank structure. Table 3.1 lists the size of the rebus N, the maximal rank of the off-diagonal blocks p, and time for a rebus-rebus multiplication. For comparison, the time taken for the same multiplication by the BLAS routine `dgemm`.

	Size of rebus or matrix									
p	128	256	512	1024	2048	4096	8192	16384	32768	65536
4	0.01	0.05	0.03	0.07	0.14	0.3	0.61	1.38	2.59	5.41
8	0.02	0.03	0.06	0.13	0.26	0.56	1.21	1.72	3.46	7.36
16	0.02	0.05	0.1	0.2	0.41	0.84	1.82	2.70	5.44	11.8
32	0.03	0.07	0.17	0.36	0.76	1.63	3.74	5.48	11.6	25.3
64	0.08	0.23	0.52	1.16	2.47	4.05	7.08	15.3	37.6	106
128	0.17	0.74	1.92	4.26	3.85	8.11	17.0	36.0	86.7	
`dgemm`	0	0.04	0.28	2.01	15.8	123	1070			

Table 3.1. Timings for $N \times N$ rebus-rebus multiplication.

For differential operators in one dimension, our off-diagonal ranks will be order of 10. thus we will break even with the dense matrix multiply at matrix sizes of about 500. We also see that for problems of size about 10,000, the dense matrix multiply was no longer possible in the 1.5 GB of memory the testing computer had available. Again, the memory efficiency of the rebus methods allows these much larger systems to be tackled.

Figure 3.1 shows the scaling of the multiplication algorithm for different values of p. In each case the scaling is approximately linear as expected. For cases when the ratio of p to N is large, the rebus is approaching a full matrix, and so the timings for larger p are seen to depart from linearity for small matrix sizes. The `dgemm` times are too large to be usefully shown on the same graph.

We can estimate the scaling behavior of the algorithm by considering the log-log plot of the time taken against N, the system size. If we assume that the cost scales as N^{α}, for some α, a simple linear regression can be used to estimate the scaling exponent α. For $p = 128$, the highest-rank case tested, the linear regression estimate of α was found to be 1.00, with $R^2 = 0.97$. For the $p = 8$ case, where we expect more benefit from the rebus structure, a linear regression gives us the estimate $\alpha = 0.95$ with $R^2 = 0.99$. Thus, in practice the algorithm scales linearly in the system size.

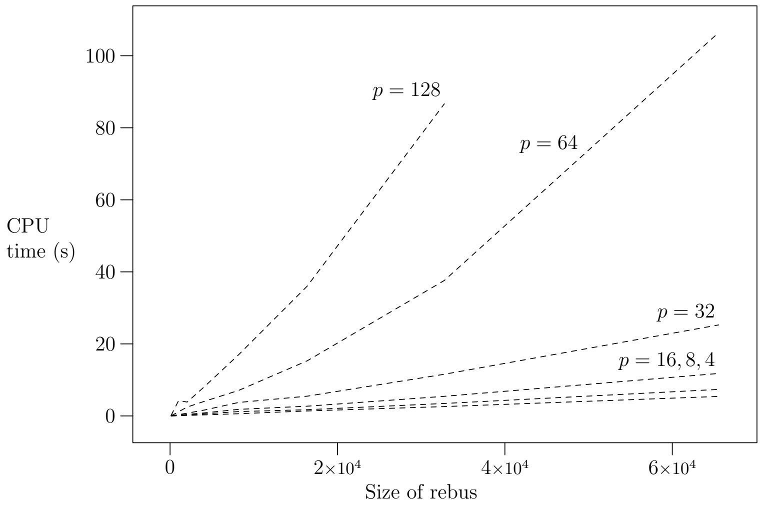

Figure 3.1. Scaling of rebus-rebus multiplication at different values of p.

3.3 LU Factorization

The LU factorization is a fundamental tool in matrix algebra. It is invaluable in applications, where we frequently encounter the problem of solving a matrix system

$$Ax = b$$

for multiple right hand sides b.

If we wish to represent a linear operator by anything other than its matrix, we would clearly benefit from having access to an LU decomposition for our representation. For example, in order to perform certain calculations efficiently in a wavelet basis, it is convenient to represent the operator in "nonstandard form" [8]. This representation of the linear operator requires new algorithms in order to achieve the composition of operators or the solution of linear systems. In order to extend the applicability of this nonstandard form, Gines, Beylkin and Dunn have derived an LU factorization and used it to directly solve systems of linear equations [24].

We are interested in using rebus algorithms for the solution of partial differential equations discretized by collocation on the Chebyshev nodes (see Chapter 4). In this case the solution is arrived at by stepping the initial condition forward in time by repeatedly solving against a time evolution operator. The availability of a rebus LU factorization makes it possible to further reduce the overall computation by simply factorizing the evolution operator.

As part of a larger algebra of operations that we can quickly execute on rebuses, we hope to realize a linear-time LU factorization. In Section 3.3.1 we describe such an algorithm. Then, in Section 3.3.2 we present numerical results for the algorithm.

3.3.1 Description of Algorithm

Let A be a k-level rebus, consisting of the factors

$$D, B, u, v, r, w$$

Let f and g be local quantities calculated at each node and leaf. Each node and leaf is able to return a g when provided with its parent's f. At each node, the calculations of f and g are as described below, with the subscripts r and l referring to the right and left child node, respectively. At a leaf, we calculate

$$g = v^H (D - u\, f\, v^H)^{-1} u$$

and at a node, in terms of the quantities $B'_{ij} = B_{ij} - r_i f w_j^H$, we define

$$g = \begin{pmatrix} w_l & w_r \end{pmatrix} \begin{pmatrix} g_l + g_l\, B'_{lr}\, g_r\, B'_{rl}\, g_l & -g_l\, B'_{lr}\, g_r \\ -g_r\, B'_{rl}\, g_l & g_r \end{pmatrix} \begin{pmatrix} r_l \\ r_r \end{pmatrix}.$$

Each of these calculations depends on an f from the parent node. At the root $f = \phi$ and at a node, the f sent to the left is

$$f_l = r_l\, f\, w_l^H.$$

When the left branch has returned g_l we calculate

$$f_r = \begin{pmatrix} r_r & B_{rl} \end{pmatrix} \begin{pmatrix} f + f\, w_l^H\, g_l\, r_l\, f & -f\, w_l\, g_l \\ -g_l\, r_l\, f & g_l \end{pmatrix} \begin{pmatrix} w_r^H \\ B_{lr} \end{pmatrix},$$

and this is sent to the right branch.

Once the calculation of the above quantities has terminated, we have all the information needed for L and U. The diagonal blocks of L and U are given by the LU-factors of the $D - ufv^H$, which we will denote $\mathcal{L}$ and $\mathcal{U}$.

The components of L are:

$$\begin{pmatrix} D \leftarrow \mathcal{L} & r \leftarrow r & w_l \leftarrow w_l & w_r \leftarrow w_r - w_l\, g_l\, B'_{lr} \\ B_{rl} \leftarrow B'_{rl} & B_{lr} \leftarrow 0 & u \leftarrow u & v^H \leftarrow v^H\, \mathcal{U}^{-1} \end{pmatrix},$$

and the factors of U are:

$$\begin{pmatrix} D \leftarrow \mathcal{U} & r_l \leftarrow r_l & r_r \leftarrow r_r - B'_{rl}\, g_l\, R_l & w \leftarrow w \\ B_{rl} \leftarrow 0 & B_{lr} \leftarrow B'_{lr} & u \leftarrow \mathcal{L}^{-1}\, u & v \leftarrow v \end{pmatrix} .$$

We note that as L shares the row space of the original matrix A, we can set u and r of L to be those of A, and similarly with the column space and U.

3.3.2 Numerical Examples

The purpose of the above algorithm is to allow us to produce LU-factorizations much more rapidly than via conventional methods. Using dense matrices, the operation count for LU decomposition is $\frac{1}{3}N^3$ for an $N \times N$ matrix.

Our operation count is expected to be linear in the size of the discretization. We provide two sets of experimental times to demonstrate the effectiveness of the algorithm. First we will look at the LU factorization of a standard kernel on the Chebyshev nodes. Second we will look at the LU factorization of random rebuses of varying size and off-diagonal rank.

LU of kernel discretized on Chebyshev nodes

The first set of experimental runtimes come from the discretization of the kernel

$$k(x, y) = |x - y| \sin(|x - y|)$$

which demonstrates rank structure typical of operators that have HSS structure. We discretize this kernel on the n zeros of the nth Chebyshev polynomial $T_n(x) = \cos(n \cos^{-1} x)$. These points cluster quadratically at -1 and 1. We partition these points by recursively dividing the interval $[-1, 1]$ into equal halves and halting this process when the number of nodes in the interval reaches a preset limit. Thus, in the case recorded here, the partition tree will be highly non-uniform, with a much deeply nested structure near the endpoints of the interval.

N	Rebus LU	`dgetrf`
200	0.01	0.12
400	0.02	0.45
800	0.04	2.03
1600	0.11	9.11
3200	0.22	46.2
6400	0.42	263
12800	0.84	-
25600	1.52	-
51200	2.99	-
102400	5.84	-

Table 3.2. CPU time (s) for LU factorization of an $N \times N$ rebus or matrix.

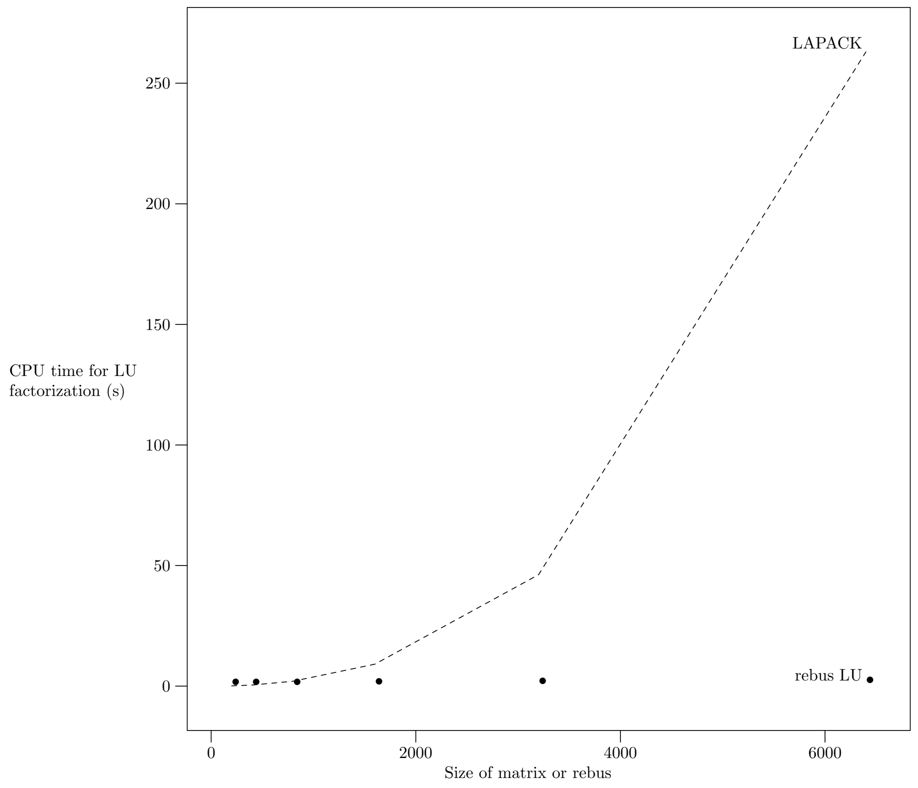

Figure 3.2. Timings for LU factorization of $N \times N$ matrix and rebus.

Table 3.2 and Figures 3.2 and 3.3 show the timings for the rebus LU, and compare them to LAPACK's LU factorization routine, `dgetrf`. From the log-log plot in Figure 3.3, we can estimate the complexity of the algorithm. Assume that the algorithm scales as N^{α}. Then a linear regression on the log times estimates

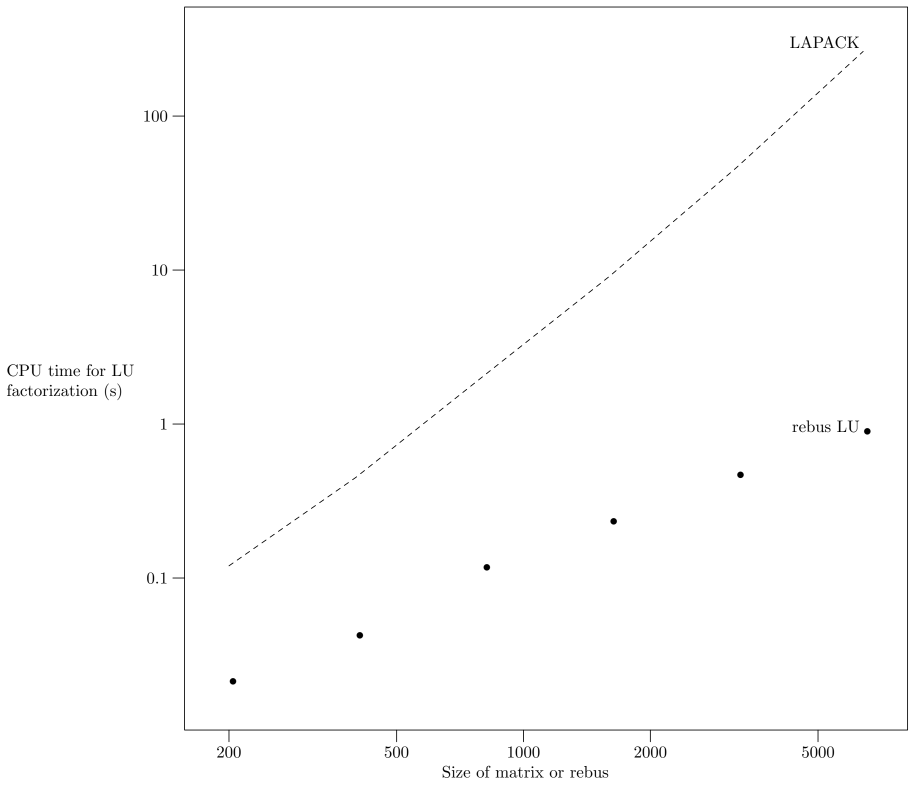

Figure 3.3. Timings for LU factorization of $N \times N$ matrix and rebus (log scale).

$\alpha = 1.03$, with $R^2 = 1.00$. Thus, the algorithm presented is of linear complexity: it runs in time proportional to the size of the system.

LU of random rebuses

For these numerical experiments we generated random rebuses with known rank structure. Table 3.3 lists the size of the rebus N, the maximal rank of the off-diagonal blocks p, and time for a rebus LU factorization. For comparison, the time taken for the same multiplication by the BLAS routine `dgetrf`.

It is also worth noting the benefit of being more efficient in terms of memory, as well as time. In some cases time is not critical, but we would want to solve the problem at hand on a machine we have available. These tests were carried out on an Apple dual 1GHz PowerPC G4 machine with 1.5GB of RAM. For matrix sizes of greater than about 10,000 the dense matrix computations could no longer

	Size of rebus or matrix									
p	128	256	512	1024	2048	4096	8192	16384	32768	65536
4	0.02	0.03	0.09	0.11	0.23	0.49	0.99	2.08	4.15	8.65
8	0.02	0.05	0.10	0.25	0.54	1.07	2.70	3.01	6.33	13.1
16	0.03	0.07	0.15	0.42	0.83	1.96	2.44	4.90	10.2	21.0
32	0.09	0.20	0.34	0.73	1.74	2.93	4.93	10.2	20.5	42.1
64	0.10	0.27	0.51	1.31	2.74	5.65	11.8	24.7	52.8	137
128	0.10	0.47	1.34	3.11	6.82	14.2	29.3	61.1	136	331
`getrf`	0.01	0.02	0.19	0.98	6.52	46.7	393			

Table 3.3. Time (s) for $N \times N$ rebus LU factorization.

be completed at all. Due to the compressed nature of the rebus representation, much larger rebuses may be manipulated.

3.4 Forward (and Back) Substitution

Given a lower (or upper) triangular rebus, we would like to be able to solve systems by forward (and back) substitutions. In this section we describe algorithms to accomplish this. Similar to the LU factorization, these take the form of a depth-first sweep through the tree, with the intermediate upsweep and downsweep variables being created as needed. Corresponding to the dense form of the algorithm, the forward substitutions hit the nodes in left-to-right order and the back substitutions hit them in reverse order.

3.4.1 Forward Substitution

Let us consider solving the system

$$L\,x = b,$$

where L is a lower triangular rebus and x and b are taken to be dense vectors, block-partitioned conformally with L. We will use the same notation as in Section 3.3.1.

The upsweep variable g is calculated at each leaf as:

$$g = v^H D^{-1}(b - u\, f).$$

Each of the variables in this expression is associated with the leaf at which we are performing the calculation, except f, which is passed to the leaf by its parent.

Note that if our operator L is truly lower triangular, then in particular all of its diagonal blocks will be lower triangular. Thus the calculation at the leaves of $D^{-1}(b - u\, f)$ can itself be accomplished via conventional, dense matrix, forward substitution.

At a node, once the children's g are known, we calculate

$$g = (w\, g)_l + (w\, g)_r$$

and the downsweep variable f is defined to be an empty matrix at the root, and at a node

$$f = R\, f_p + B\, g_s,$$

where the sibling subscript refers to the left sibling (if present).

Given the lower triangular rebus L, we may solve the system $Lx = b$ by partitioning x and b conformally with L and solving

$$\mathcal{L}x = b - u\, f$$

at each node. The influence of all the non-diagonal blocks is mediated by the presence of the f.

3.4.2 Back Substitution

The complementary problem of solving the system

$$U\, x = b$$

where U is an upper triangular rebus is equivalent to forward substitution with left and right reversed. Thus the calculation at each leaf depend on the calculations at every leaf to the right.

At a node, the variable sent down to the child to the left is

$$f = R\,f_p + B\,g_s$$

where the sibling subscript now refers to the right sibling (if present). To the right, we simply send

$$f = R\,f_p.$$

The calculation is initiated with an empty matrix being sent to the root node as its f. This is then propagated to the right hand branch successively until a leaf is reached, the first g is calculated and returned up to that leaf's parent node. Once a left subtree has returned, the calculation then proceeds through the right subtree. This is a preorder depth-first traversal of the tree.

3.4.3 Numerical Examples

We consider a family of matrices A of the form

$$A_{i,j} = \sqrt{|x_i - x_j|},$$

where for an $n \times n$ matrix A the points x_i are chosen to be the n zeros of the nth Chebyshev polynomial $T_n(x) = \cos(n \cos^{-1} x)$. These points cluster quadratically at -1 and 1. We partition these points by recursively dividing the interval $[-1, 1]$ into equal halves. We stop partitioning an interval as soon as the number of points in that interval decreases below a pre-set threshold p. This partitioning is then used as the HSS partitioning of A. We did a series of experiments with such matrices A ranging in size from 256 to 131072. In each case we report the values of p, the cpu run-time in seconds, and the approximate backward error,

$$\frac{\|A\hat{x} - b\|_1}{n\|\hat{x}\|_1 + \|b\|_1},$$

where n is an approximation of the 1-norm of A. The results can be found in Table 3.4.

For comparison we also provide the timings for the LAPACK solver `dgels` using the same BLAS on the same machine in Table 3.4. In some cases there was insufficient memory to do the timings. Those entries are left blank.

N	p	Factor (s)	Solve (s)	Total time (s)	Error	`dgels` (s)
256	13	0.04	0	0.04	9.7e-18	0.06
512	14	0.08	0.02	0.10	1.2e-17	0.28
1024	15	0.19	0.03	0.22	2.1e-18	1.89
2048	16	0.36	0.07	0.43	6.3e-18	13.7
4096	17	0.81	0.13	0.94	8.4e-18	107
8192	18	1.63	0.26	1.89	1.8e-18	1429
16384	19	3.05	0.50	3.55	7.6e-18	
32768	20	5.77	0.98	6.75	2.7e-18	
65536	21	11.9	2.14	14.0	3.1e-18	
131072	22	23.1	4.24	27.3	2.6e-18	

Table 3.4. Time takes to factor and solve an $N \times N$ rebus system

The algorithms described in this section provide a practical LU factorization and forward and back substitution for use in solving systems of linear equations in rebus form.

In standard linear algebra, having the LU factorization available reduces the complexity of solving a linear system from $O(N^3)$ to $O(N^2)$. The rebus algorithms presented here are designed to scale linearly with the number of unknowns. In the rebus case, once the LU factorization has been computed, the solution of the linear system by forward and back substitution accounts for only 15% of the overall solution time.

In any application where the solution for many right hand sides is needed these algorithms should provide significant improvements over current methods. In particular, for the solution of PDEs by the methods described in [32] where a rebus system had to be solved sequentially against 100 distinct right hand sides, we would expect to see large increases in speed. We discuss the impact of such LU accelerated solvers in Section 6.1.

Assuming the complexity to be proportional to N^α, we can estimate the exponent α using a linear regression of the timing data in a log-log plot. For the data in Table 3.4, doing this tells us that for the LU factorization, $\alpha = 1.02$ with $R^2 = 1.00$. Similarly, for the solution via forward and back substitution we find that $\alpha = 0.98$ with $R^2 = 1.00$. Thus, both the factorization and the solution of the system can be performed in an amount of CPU time proportional to the size of the system.

Chapter 4

Numerical Implementations of PDE Solvers

4.1 Introduction

An important factor in the rise in popularity of spectral methods is the availability of fast transform methods. Due to the global nature of the basis functions employed, matrix representations of differential operators in the spatial domain are not sparse and are costly to work with. Working in a transformed domain returns us to a sparse representation.

In a number of situations, staying in the spatial domain may be highly desirable. The most common reason for favoring a transform-free method is to facilitate enforcing boundary conditions. If we wish to work on a non-periodic problem and specify Dirichlet or Neumann boundary conditions, it is natural to work in physical space. However, if we do so our Chebyshev derivative operator will be a full matrix and the size of problem we can tackle will be severely reduced.

Using the rebus algorithms introduced in Chapter 3, we may construct methods for efficiently solving spectral discretizations of partial differential equations while staying in the spatial domain. We will develop our ideas in the context of linearly implicit (LI) methods. This approach is currently the most popular choice for PDEs with low order nonlinear terms and higher order linear terms.

A recent paper by Kassam and Trefethen [11] includes a survey of methods being applied to treat problems of this kind and indicates that LI methods may not always be the best choice. As such, we also consider how the ideas we propose here can be used to similar advantage in integrating factor (IF) and exponential time differencing (ETD) methods. These methods are of recent vintage but show great promise.

Linearly implicit methods, also known as implicit-explicit (IMEX) methods, are widely used and have a history going back at least to 1980. In these early papers we find a full description of the method [20] and some results on stability [42]. More recent treatments include [3] and [2]. The defining feature of LI methods is that they employ an implicit discretization of leading order *linear* terms and an explicit discretization of the remaining, typically *nonlinear* terms. Thus, if the inversion of the resulting linear system can be accomplished at a low cost then one obtains an efficient method in which the leading order timestepping stability constraint has been eliminated. Unfortunately, the solution to such linear systems is costly for Chebyshev collocation and quite generally for spectral discretizations when remaining in the spatial domain.

Here, by employing a rebus representation of the differential operator on the spatial domain, we can use an implicit time discretization of the leading order linear operators and apply direct, fast, non-iterative methods to solve the resulting linear system at each timestep. This allows us to remove the highest order timestepping constraint while retaining nearly linear scaling in the number of collocation points.

We illustrate this strategy in the particular case of functions on a finite interval, discretized by collocation on the Gauss-Lobatto points.

4.2 Approach to PDEs

The basis of our approach is the rebus representation of differential operators. In transform methods no explicit representation of the derivative operator is needed, as the coefficients of the derivative are generated through recursion. In the spatial domain, the derivative has a matrix representation and classically this is what is used in physical space numerical methods.

However, the matrix representation of the derivative is not sparse, symmetric or even normal (see [22]). Thus it is computationally expensive to work with. It is also ill-conditioned, so even a brute force solution of a discretized differential equation using iterative methods will be very expensive.

Our approach to solving the PDEs is as follows. Starting from the governing equations, we discretize the time dependence using an implicit discretization of the leading order linear terms. This avoids high stability restrictions on the allowable time step. Then we introduce the rebus representation of the spectral derivative operator and use fast algorithms to generate the needed higher order linear operator for a timestep. After this, the linear system that results from the implicit or semi-implicit time discretization and the application of the boundary conditions is solved in rebus form.

The goal of the rebus representation is to keep the derivative operator, the timestepping operator and all related quantities in rebus form at all stages of the numerical method. This allows us to use the fast algorithms of Chapter 3 to achieve large computational savings.

4.3 Description of the PDE

We consider a general PDE that can be written in the form

$$u_t = \mathbf{L}(\mathbf{x}, t, \mathbf{u}) + \mathbf{N}(\mathbf{x}, t, \mathbf{u}), \quad t > 0, \quad \mathbf{x} \in \Omega, \tag{4.1}$$

where $\mathbf{L}$ and $\mathbf{N}$ represent a linear and a nonlinear differential operator, respectively. Inhomogeneous terms, if present, are also included in $\mathbf{N}$. The domain Ω is bounded and the equation is supplemented with appropriate initial and boundary conditions. We also assume that the leading order terms at small scales (e.g. terms with the highest order derivatives) are linear and thus contained in $\mathbf{L}$.

With a collocation method the computational cost of incorporating boundary conditions of Dirichlet, Neumann or mixed type is low. It is most convenient, in this approach, to use boundary bordering as described by Boyd in [13].

To illustrate ideas let us consider the particular case where the right hand side of the differential equation can be written as the sum of a linear elliptic operator and a nonlinear operator, possibly including an inhomogeneous forcing term:

$$u_t = \nabla \cdot (a \nabla \mathbf{u}) + \mathbf{N}(\mathbf{x}, t, \mathbf{u}), \tag{4.2}$$

where $a > 0$. This type of equation arises routinely as diffusion-convection equations in computational fluid dynamics or reaction-diffusion problems in chemistry. As explained in Section 4.1, a popular approach in this situation is to treat the elliptic part implicitly and the other terms explicitly. The reason for this linearly implicit approach is that the elliptic term is the stiffest and gives rise to severe timestep constraints if treated explicitly. The remaining nonlinear terms are treated explicitly as their implicit discretization would result in a nonlinear system which would be difficult and costly to invert.

Terms that we are treating explicitly need to be evaluated before we proceed to solve the system. Since the rebus representation of the derivative is already being computed, it may be applied cheaply to a vector by using a rebus-vector multiply as described in Section 3.1 to efficiently calculate any derivatives in the explicit term. This would be needed for example in the common case of an advective term. Here we focus on the problem of integrating implicitly the stiff elliptic term without leaving the physical space.

4.4 Underlying Elliptic Problem

The timestepping solution of PDEs of this type reduces to solving an elliptic equation at each timestep. Since we are in the spatial domain, imposing the boundary conditions at each timestep introduces no extra complications.

In most spectral methods, a finite difference discretization is used in time. As the most simple example, consider a first-order (backward Euler) discretization. Higher order schemes in time can be easily implemented with exactly the same approach. The time discretization is

$$\frac{1}{\Delta t}(\mathbf{u}(\mathbf{x}, t+\Delta t) - \mathbf{u}(\mathbf{x}, t)) = \mathbf{L}(\mathbf{x}, t+\Delta t, \mathbf{u}) + \mathbf{N}(\mathbf{x}, t, \mathbf{u}).$$

To step our solution forward in time with given, time-dependent Dirichlet boundary conditions we solve

$$\mathcal{L}_{\Delta t}\mathbf{u}(\mathbf{x}, t+\Delta t) = \mathbf{u}(\mathbf{x}, t) + \Delta t \mathbf{N}(\mathbf{x}, t, \mathbf{u}), \tag{4.3}$$

where $\mathcal{L}_{\Delta t} = I - \Delta t \cdot \mathbf{L}$.

The solution at each timestep consists of the following steps.

1. Generate the rebus representation of D, the Chebyshev derivative operator.

2. Use the fast rebus-rebus multiplication algorithm of Section 3.2 to generate the relevant D^n operator.

3. Use scaling and diagonal updates to generate $\mathcal{L}_{\Delta t}$. Algorithms for this are presented in Sections 4.4.1 and 4.4.2.

4. Evaluate the nonlinear term $\mathbf{N}(\mathbf{x}, t, \mathbf{u})$ using $\mathbf{u}$ and possibly the rebus representation of D.

5. Add the right hand side terms to obtain a single, vector-valued, right hand side.

6. Apply boundary conditions via an efficient leaf-update of $\mathcal{L}_{\Delta t}$, described in Section 4.5.

7. Solve the system using the algorithm presented in [17].

Step 1 may of course be done only once for a given set of nodes and may then be stored. In nearly all problems of interest, step 2 may be done as a preprocessing step and need not be repeated. For a constant coefficient equation with uniform timesteps, step 3 may also be taken out of the loop. In this special case we need only evaluate the nonlinear term, update boundary conditions and solve.

4.4.1 Rebus Scaling

Consider the operation $A \to c \cdot A$. For a matrix representation of A we clearly update the matrix elements $a_{i,j} \to c \cdot a_{i,j}$. Now consider the equivalent rebus operation. We will use the notation of Section 2.2.

The interaction of each subdomain with each other subdomain is represented either by $D_{K;i}$ or the product $U_{k;i} B_{k;i,j} V_{k;j}^H$, for some k. U and V themselves satisfy Equation 2.1.

Thus, the structure can correctly be scaled by

$$D_{K;i} \to c \cdot D_{K;i} \quad \text{for } i = 1 \dots 2^K$$

and

$$B_{k;i,j} \to c \cdot B_{k;i,j} \quad \text{for } k = 1 \dots K \text{ and } (i,j) = (1,2),(2,1),\dots,(2^k-1,2^k).$$

All other factors of the rebus remain unchanged.

4.4.2 Diagonal Updates

The backward Euler timestepping discretization of the PDE is given by

$$(I - \Delta t \cdot \mathbf{L})\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u}(\mathbf{x}, t) + \Delta t \mathbf{N}(\mathbf{x}, t, \mathbf{u})$$

and in this case the operator we must represent as a rebus is

$$\mathcal{L} = I - \Delta t \cdot \mathbf{L}.$$

Given the rebus representation of $\mathbf{L}$, forming this operator requires a scaling by $-\Delta t$, followed by a diagonal update.

Every off-diagonal block of $I - \Delta t \cdot \mathbf{L}$ is equal to the corresponding block of $-\Delta t \cdot \mathbf{L}$. Thus the update need operate only on the blocks $D_{K;i}$. But these are exactly the blocks which are stored explicitly as normal matrices in our rebus representation.

Thus, the structure can be correctly updated by

$$D_{K;i} \to I - D_{K;i} \quad \text{for } i = 1 \ldots 2^K,$$

where $D_{K;i}$ are the diagonal blocks of our scaled rebus.

4.5 Boundary Conditions by Boundary Bordering

The main reason for pursuing these methods is to efficiently treat PDEs with nonperiodic boundary condition. It is appropriate then, to consider how to enforce various boundary conditions in the rebus formulation.

As discussed by Boyd in [13], the most robust and flexible method of imposing physical boundary conditions for a matrix formulation of Chebyshev collocation is boundary bordering. The principle of this method is to allocate m rows of the matrix to explicitly imposing the m boundary conditions, of whatever type. Thus, to impose Dirichlet boundary conditions on a discretization of a second order equation in one dimension we collocate at $N - 2$ interior points of the interval and use two rows of the matrix to impose boundary conditions.

The equivalent process for a rebus proceeds as follows. Instead of using the first two rows of the matrix we need to respect the spatial structure and operate on the first and last rows, corresponding to the boundary positions. We can then accomplish the bordering. However, the rows that need to be replaced are not readily available, as they are split between a number of blocks, which are themselves factored.

The process is less straightforward than bordering a matrix, but is not difficult. The process may be broken down as follows.

- Initialize by setting the m required rows to zero within the rebus.
- Form the required boundary conditions as dense matrix rows.
- Construct a low-rank product expressing the desired boundary conditions in matrix form.
- Use Algorithm 4.5.4 to add this low-rank product to the rebus.

Thus the procedure amounts to zeroing out m rows of the rebus and performing one rank-m addition.

4.5.1 Initialization

We may modify the first and last diagonal block, $D_{K;1}$ and $D_{K;N}$ directly, as we would the corresponding matrix. However, the remaining factors contributing to the border rows still need to be set to zero. (We tacitly assume here that all the border rows can be contained in the rows covered by these blocks. The procedure below extends to the more general case).

We use the fact that each other element of the first row is the first row of a block $U_{k;i}B_{k;i,j}V_{k;j}^H$, for some k, and that all of the $U_{k;i}$ are generated from the lowest level $U_{K;i}$ via the relation

$$U_{k-1;i} = \begin{pmatrix} U_{k;m}R_{k;m} \\ U_{k;n}R_{k;n} \end{pmatrix}. \tag{4.4}$$

Our requirement is that

$$\sum_{\mu=1}^{p}\sum_{\nu=1}^{q}(U_{k;i})_{1,\mu}(B_{k;i,j})_{\mu,\nu}(V_{k;j}^{H})_{\nu,\kappa} = 0$$

for all κ for each block. Thus it is sufficient to require that

$$(U_{k;1})_{1,\mu} = 0 \quad \text{for all } \mu \quad \text{for all } k.$$

And from the recursion relation 4.4 it is sufficient to enforce

$$(U_{K;1})_{1,\mu} = 0 \quad \text{for all } \mu$$

to achieve boundary bordering of the first row.

Similarly, we impose analogous conditions on all other rows that are to be used for boundary conditions. For example, we would use

$$(U_{K;N})_{N_K,\mu} = 0 \quad \text{for all } \mu$$

for boundary bordering on the final row.

4.5.2 Boundary Conditions in Matrix Form

We now generate the boundary rows that we would border with in a matrix method.

For Dirichlet boundary conditions or conditions on any linear combination of endpoint or interior values in the form

$$\sum_{j=1}^{N}\omega_j u(x_j) = \alpha,$$

where x_j are our Chebyshev nodes, we simply border with the vector of weights ω. In the Dirichlet case this amounts to a single 1 in the first or last position.

Alternatively, we may be presented with Neumann conditions or conditions on some linear combination of derivatives at the endpoints or interior points. Our boundary condition then would have the form

$$\sum_{i=1}^{N}\omega_i u'(x_i) = \alpha.$$

Let D be the Pseudospectral derivative operator. Using the matrix representation of the derivative, our condition is equivalent to

$$\sum_{j=1}^{N} \left(\sum_{i=1}^{N} \omega_i D_{ij} \right) u(x_j) = \alpha. \tag{4.5}$$

Thus, we border our matrix with a linear combination of rows from the derivative matrix D as determined by the weights ω. In the case of a simple Neumann condition at $x = -1$, we would border with the first row of D.

For more exotic boundary conditions or constraints the treatment is equally straightforward. An integral condition may be represented by any suitable Chebyshev quadrature scheme, for example the Gaussian or Clenshaw-Curtis scheme. Since quadratures are expressed in terms of weights as

$$\sum_{j=1}^{N} w_j u(x_j) = \alpha,$$

the correct boundary row is again simply the vector of weights w.

4.5.3 Formulation as a Low-Rank Product

Given our m boundary conditions, we now need to position them appropriately in our rebus. For Dirichlet conditions in one dimension, for example, we should respect the geometry of the problem and border on the first and last rows. Similarly for Neumann conditions. The treatment of more exotic conditions is less obvious but should be guided by attempts to preserve locality in the rebus structure.

Putting our m boundary conditions together into an m by n matrix B, we then write a simple n by m permutation matrix Π so that the product ΠB contains the rows in their chosen position relative to the rows in the rebus.

These are the same rows whose initialization we described in Section 4.5.1 and we should also have the corresponding value of the condition α on this row on the right hand side of our rebus equation.

4.5.4 Low-Rank Addition

It remains to combine the initialized rebus with the boundary conditions. We can do this using an efficient algorithm for a low-rank update of a rebus.

Consider the problem of adding the product AC^T to a rebus. As in Section 2.2 we will name the components of the rebus as follows: diagonal blocks D_k, the lowest level factors U_k and V_k and a binary tree of low-rank factors R, W and B.

We first partition the columns of A and C commensurately with the rebus.

$$A = A_0 = \begin{pmatrix} A_{1;1} \\ A_{1;2} \end{pmatrix} = \begin{pmatrix} A_{2;1} \\ A_{2;2} \\ A_{2;3} \\ A_{2;4} \end{pmatrix} = \cdots = \begin{pmatrix} A_{k;1} \\ \vdots \\ A_{k;2^k} \end{pmatrix},$$

and similarly for C.

At the kth level we need to represent $D_{k;i} + A_{k;i}C_{k;i}^T$ for $j = 1 \ldots 2^k$ and $U_{k;i}B_{k;i,j}V_{k;j}^T + A_{k;i}C_{k;j}^T$ for $(i,j) = (2l, 2l-1), (2l-1, 2l)$, for $l = 1 \ldots 2^{k-1}$.

The off-diagonal blocks can be updated by assigning

$$U_{k;i} \leftarrow \begin{pmatrix} U_{k;i} & A_{k;i} \end{pmatrix}, \quad V_{k;j} \leftarrow \begin{pmatrix} V_{k;j} & C_{k;j} \end{pmatrix}, \quad B_{k;ij} \leftarrow \begin{pmatrix} B_{k;ij} & 0 \\ 0 & I_m \end{pmatrix},$$

where m is the rank of the product AC^T.

It remains to treat the diagonal blocks. These are either dense matrices or each is a rebus with one fewer levels than the case just treated. If a block is a dense matrix (a zero-level rebus) we simply perform the addition

$$D_{k;i} \leftarrow D_{k;i} + A_{k;i}C_{k;i}^T.$$

If the block is a rebus with one fewer levels, we recursively apply the original procedure until all the diagonals have terminated with a dense block.

So using a straightforward recursive algorithm we can efficiently add a low-rank matrix to a rebus.

4.6 Numerical Examples

The following section reports the results of applying these methods to a number of test problems. Since conventional methods find the combination of non-periodic boundary conditions with many collocation points the most problematic, we focus on such problems.

The implicit treatment of the diffusion terms corresponds to the conventional use of linearly implicit methods (see [3]) for the convection-diffusion or reaction-diffusion equations arising in chemical simulation. Such simulations typically use a spectral discretization in space and face exactly the problem we address: efficiently solving the equations arising from an implicit discretization in time.

4.6.1 Backward Euler Discretization of a Diffusion Equation with Dirichlet Boundary Conditions

Consider the test problem of a diffusion equation applied to a Gaussian function.

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial^2 u}{\partial x^2} \qquad -1 \le x \le 1, t \ge 0, \\ u(x,0) &= \exp(x^2) \qquad t = 0, \\ u(\pm 1, t) &= \sqrt{\frac{1}{t+1}} \exp\left(\frac{-1}{4(t+1)}\right) \qquad t > 0. \end{aligned} \tag{4.6}$$

This mixed initial-boundary value problem has the exact solution

$$u(x,t) = \sqrt{\frac{1}{t+1}} \exp\left(\frac{-x^2}{4(t+1)}\right). \tag{4.7}$$

Using a first order accurate implicit Euler method, we wish to integrate this from $t = 0$ to $t = 1$ while maintaining our error $||u(x,1) - \hat{u}(x,1)||_\infty \le 10^{-3}$. Thus we report timings and uniform errors after 1000 steps.

As stated earlier, we use collocation on the Gauss-Lobatto points. The number of collocation points ranges from 200 to 6,400. The CPU time taken for the

N	CPU Time (s)	error (e-5)
200	16	4.85
400	41	4.74
800	105	4.68
1600	251	4.66
3200	610	4.62
6400	1424	4.48

Table 4.1. Timings for 1000 rebus timesteps on N Chebyshev nodes.

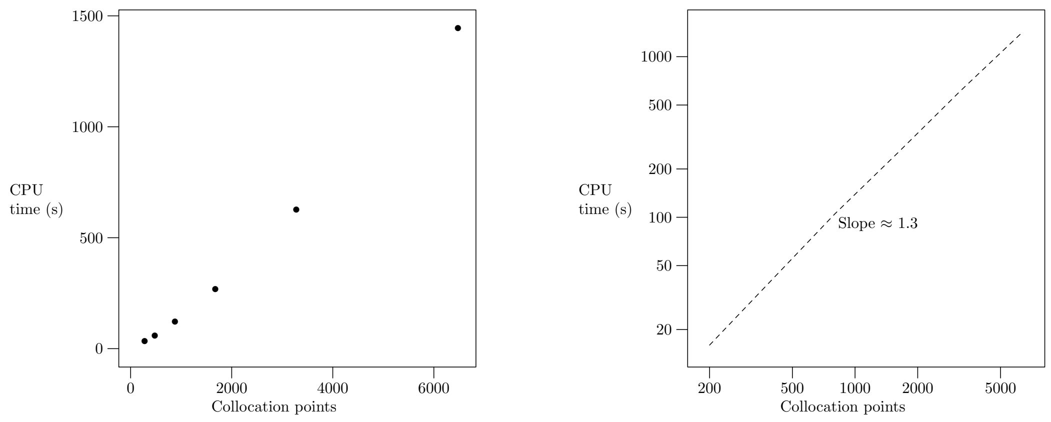

Figure 4.1. Timings for 1000 rebus timesteps on N Chebyshev nodes with linear and log scales.

1000 steps is seen to scale approximately as $N^{1.3}$ which, compares favorably to the N^3 or N^2 scaling of other non-iterative methods of solution of the system. Since we are using a direct method of solution, no preconditioning is necessary.

The scaling observed in Table 4.1 and Figure 4.1 falls short of the theoretical performance of rebus-based methods. The theory and underlying solvers would lead us to expect linear scaling of the solution time with the number of collocation points. As the solvers are themselves linear in time (see [17]), the extra cost associated with the PDE solver may be attributed to non-optimality of the rebus representation with respect to the underlying rank structure. This can hopefully be improved, but is already competitive with $n \cdot \log n$ methods.

4.6.2 Crank-Nicolson Discretization of a Diffusion Equation with Time-Varying Non-Homogenous Dirichlet Conditions

We return to the test problem of a diffusion equation applied to a Gaussian function.

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial^2 u}{\partial x^2} \qquad -1 \le x \le 1, t \ge 0, \\ u(x,0) &= \exp(x^2) \qquad t = 0, \\ u(\pm 1, t) &= \sqrt{\frac{1}{t+1}} \exp\left(\frac{-1}{4(t+1)}\right) \qquad t > 0. \end{aligned} \tag{4.8}$$

Note that the Dirichlet boundary conditions are non-homogenous and time varying. This is the same kind of boundary condition we would need to apply for a Dirichlet boundary control problem.

This mixed initial-boundary value problem has the exact solution

$$u(x,t) = \sqrt{\frac{1}{t+1}} \exp\left(\frac{-x^2}{4(t+1)}\right). \tag{4.9}$$

A second-order Crank-Nicolson scheme was employed for the time integration of the linear term and this example does not have a nonlinear term. To demonstrate that the current approach retains the second order convergence expected of a Crank-Nicolson method we first refine our timestep on a fixed grid. Table 4.2 demonstrates the effect of taking 2^n steps for $n = 3 \dots 14$ on a collocation grid of 64 Gauss-Lobatto points.

Assuming the error of the scheme is of the form $||u(x,1) - \hat{u}(x,1)||_\infty = k \cdot N^{-\alpha}$, a regression of the data determines $\alpha = 2.0$ with $R^2 = 1.00$. As expected, our implementation of the Crank-Nicolson scheme is second-order.

To demonstrate the scaling properties of the method, now take a set number of timesteps and refine our collocation grid. As the spatial accuracy is already exponentially convergent, our accuracy is limited by the fixed timestep size. We are refining the grid in order to show the desirable scaling properties of the method as N grows.

$log_2(N)$	N	error
3	8	3.29E-05
4	16	1.07E-05
5	32	1.93E-06
6	64	4.87E-07
7	128	1.22E-07
8	256	3.04E-08
9	512	7.61E-09
10	1024	1.90E-09
11	2048	4.75E-10
12	4096	1.24E-10
13	8192	2.82E-11
14	16384	7.05E-12

Table 4.2. Convergence after N rebus timesteps on 64 Chebyshev nodes.

We solve Equation 4.8 from $t = 0$ to $t = 1$ for a changing number of collocation points while maintaining our error $||u(x,1)-\hat{u}(x,1)||_\infty \leq 10^{-5}$. We report timings and uniform errors after 100 steps.

N	CPU Time (s)	error (e-6)
200	2.13	1.20
400	5.23	1.19
800	14.1	1.19
1600	38.5	1.20
3200	87.4	1.22
6400	213	1.44

Table 4.3. Timings for 100 rebus timesteps on N Chebyshev nodes.

A regression analysis of the data in Table 4.2 shows the cost to scale as $N^{1.3}$ with $R^2 = 1.00$. This exponent is not optimal, as linear scaling can theoretically be achieved [18]. The performance is competitive with the $N\log(N)$ currently possible for transform-based, diagonalizable problems. It is clearly superior to the N^3 cost expected for non-diagonalizable problems.

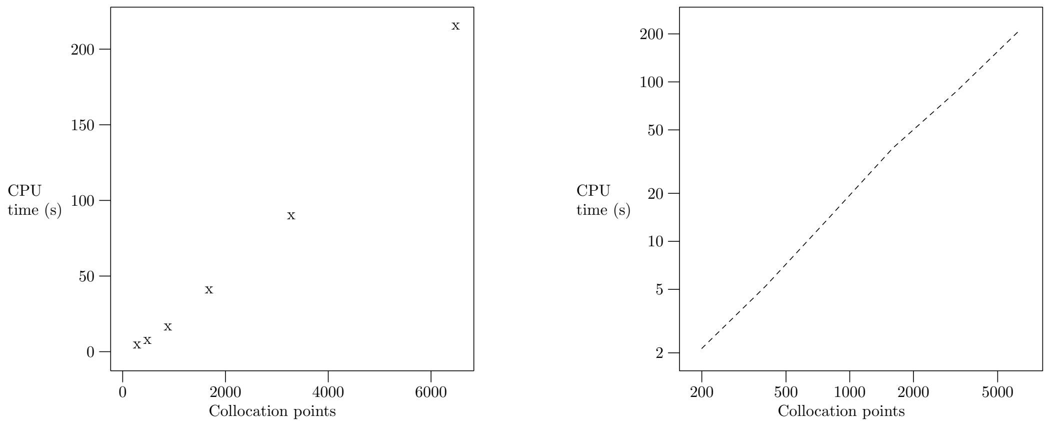

Figure 4.2. Timings for 100 rebus timesteps on N Chebyshev nodes with linear and log scales.

4.6.3 Crank-Nicolson / Runge-Kutta IMEX Discretization of the Allen-Cahn Equation with Homogenous Neumann Boundary Conditions

Having demonstrated the second order convergence of the method and the scaling of computational cost with grid size, we now apply the method to a more challenging simulation. We timestep a reaction-diffusion equation with a nonlinear term and Dirichlet boundary conditions.

Consider the initial value problem for an Allen-Cahn equation of the form

$$\begin{aligned} u_t &= \epsilon u_{xx} + u - u^3 \qquad -1 \leq x \leq 1, t \geq 0, \\ u(x,0) &= \sin(5\pi x/2) \qquad t = 0, \\ u_x(\pm 1, t) &= 0 \qquad t > 0. \end{aligned} \tag{4.10}$$

Equation 4.10 has stable equilibria at $u(x) = \pm 1$ and an unstable equilibrium at $u(x) = 0$. It also demonstrates slow dynamics, whereby metastable states may persist for relatively long periods before undergoing a rapid transition to a lower energy state [44].

In this case the linear term was integrated via a Crank-Nicolson scheme, the nonlinear term via an explicit Runge-Kutta scheme and the Neumann boundary conditions were enforced as described in Section 4.5. Table 4.4 shows the computational time taken for different grid sizes.

N	CPU Time (s)
200	2.06
400	5.12
800	13.9
1600	35.5
3200	86.7
6400	206

Table 4.4. Timings for 100 rebus timesteps on N Chebyshev nodes.

A regression of the data in Table 4.4 shows that the scaling exponent has remained 1.3, as in Section 4.6.2, again with $R^2 = 1.00$. The addition of Dirichlet boundary conditions and a nonlinear term has not altered the scaling behavior of the method.

The initial and final states are shown in Figure 4.3 and the evolution of the state is shown in Figure 4.4. Around the midpoint of this evolution we observe the transition out of the metastable state. High order methods are desirable for tracking transitions such as this. As the solution is stable over relatively long periods, an adaptive step size would also be appropriate for this kind of problem. This is straightforward to implement in a rebus scheme. Whatever timestep is needed, the relevant operator is still generated via fast operations from the derivative rebus, which is already known.

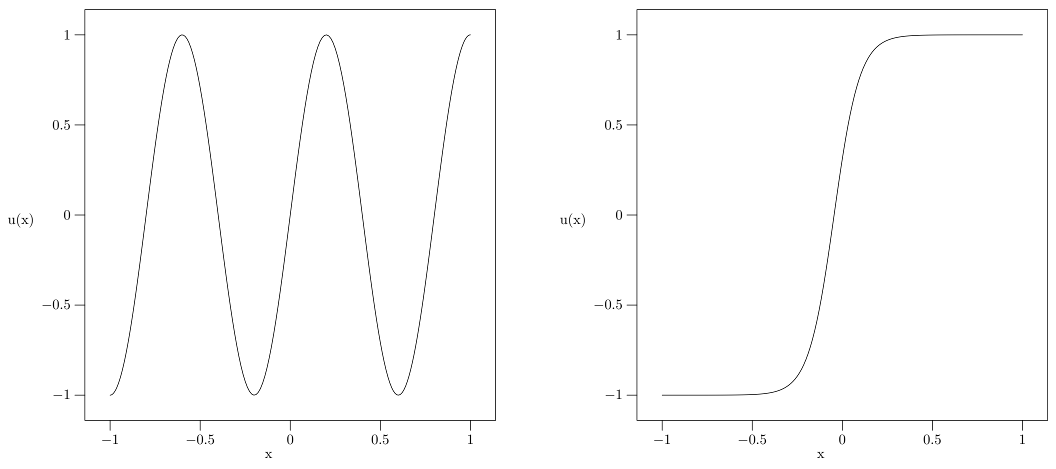

Figure 4.3. Initial condition and final ($t = 127$) state for the Allen-Cahn equation.

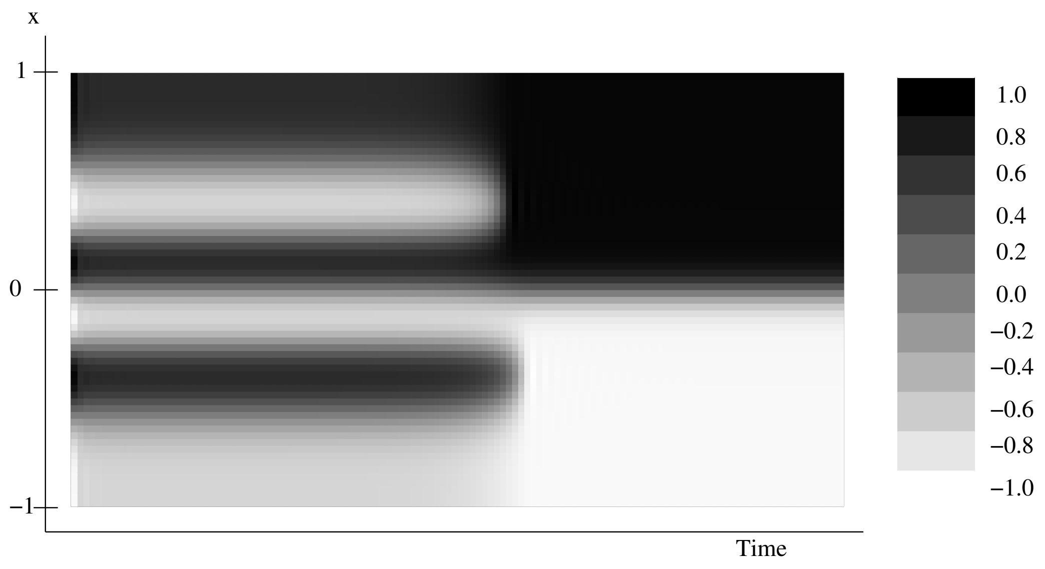

Figure 4.4. Time evolution between the states of Figure 4.3

Chapter 5

Further Considerations for Rebus-Based Solution of PDEs by Spectral Collocation

5.1 Introduction

In applying rebus-based methods to the numerical solution of PDEs, there are many auxiliary issues that need to be treated. by far the most important of these is the question of conditioning. The Chebyshev derivative operator has very high condition number, and all numerical calculations are constrained by this. The problem is not critical for small systems, but as the size of the discretization grows, the numerical concerns also do. Since the rebus methods of Chapter 4 assume that we are dealing with large systems, careful consideration of the effects of ill-conditioning are called for.

In Section 5.2 we discuss these issues and in Section 5.3 we cover how best to deal with them. Then in Section 5.4 we consider the application of these methods in higher dimensional problems.

5.2 Conditioning of Chebyshev Derivative Operators

Since we wish to work in double precision with large numbers of collocation points, we need to be especially careful to avoid being swamped by numerical errors. The pth order derivative sends the first p polynomials to zero and so will have p singular values which are zero. On a computer this will result in singular values of roughly machine precision and maximally ill-conditioned matrices.

This is not surprising, as without p boundary conditions we should not expect to be able to solve $g^{(p)}(x) = f(x)$. However, imposing boundary conditions does not give us a well conditioned system. Figure 5.1 gives an indication of the magnitude of this problem, showing the largest and smallest nonzero singular values for the second derivative.

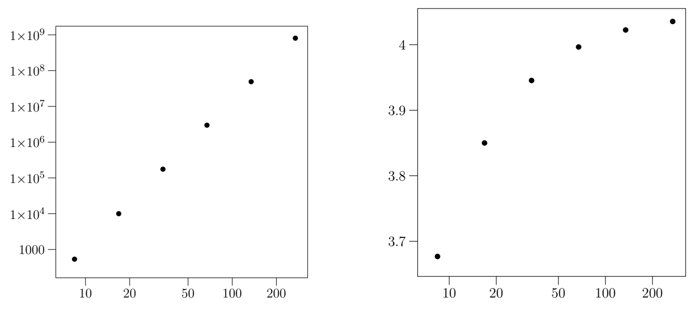

Figure 5.1. Largest and smallest nonzero singular values of D^2_{ps} against grid size (note scales).

So, even if we manage to maintain our lowest singular value at order unity, the condition number will grow as shown here, at a rate of p^4 and reaching a value of 10^9 by the time we have 512 collocation points. This leaves us only 8 digits of accuracy for an exact calculation.

Working with Chebyshev collocation methods with many collocation points, high order derivatives and multiple dimensions is a numerically dangerous proposition. The methods presented in this thesis aim to make certain types of calculation much more efficient. However, the numerical considerations of roundoff error and ill conditioning remain.

It is well known that the discretization of the derivative operator on Chebyshev nodes gives rise to an ill-conditioned matrix. As N, the number of discretization points, increases, the condition number of the derivative matrix D grows with $O(N^2)$. Similarly, the condition number of the second derivative operator grows with $O(N^4)$. As the order of the linear term or the dimension increases, this can quickly become unmanageable.

For example in a Kuramoto-Sivashinsky equation [44] we would have an $O(N^8)$ condition number in 1d or $O(N^{16})$ in 2d. If we are working with 16 digits of accuracy then at a grid size around $N = 10^2$ we can expect our answer to contain no meaningful digits. In a two dimensional simulation we cannot expect accuracy on any grid at all.

Dealing with this issue is part of using pseudospectral Chebyshev methods and there is a considerable literature dealing with it.

5.3 Strategies for Better Conditioning

In the literature, there are a number of well known strategies to improve the condition number and accuracy of pseudospectral derivative matrices.

5.3.1 Scaling of Boundary Conditions

We would expect the singular values of the differentiation matrix to vary smoothly from the highest to the lowest. However, we observe jumps up to the largest two singular values (see Figure 5.2). One way to try to overcome this (without changing our basis polynomials) is to scale our boundary values.

Figure 5.3 shows the singular values for the backward Euler timestep matrix on 256 collocation points with both unscaled boundary values and with boundary values being scaled in proportion to the grid size. Surprisingly, the optimal scaling

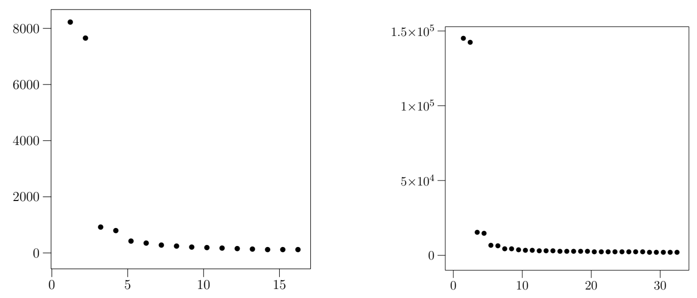
Figure 5.2. Spectrum of singular values for D_{ps}^2 for grid sizes 16 and 32.

factor was found to be the first power of p, not the second or fourth as might be expected from the scaling of the matrix norm or condition number.

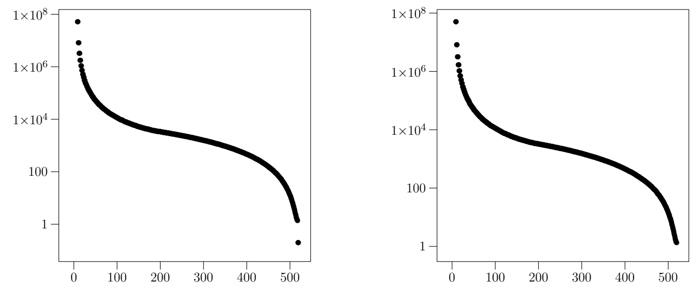
Figure 5.3. Singular values for 512×512 timestep matrix with (right) and without (left) scaled boundary values.

We see that, for large numbers of collocation points, the highest singular values are not the only problem. In fact the lowest singular values are an order of magnitude off the smooth curve. As hoped, scaling the boundary values puts them back on the curve, and reduces the condition number. However, this only buys us one order of magnitude. The condition number still scales as p^4.

5.3.2 Numerical Tricks and Basis Recombination

Numerical evaluation of the matrix representation of the derivative is subject to serious roundoff errors, especially near the boundary points. Bayliss shows how a simple adjustment to the matrix, chosen to ensure that constant functions lie in the null space of the matrix, can improve this [7]. It is also possible to use alternate formulae to generate the matrix elements, as in Tang and Trummer [41] in order to avoid the cancellations leading to numerical errors. Each of these methods can improve the accuracy of the matrix by a few orders of magnitude. The scaling of the condition number is unfortunately not improved.

In a series of papers Heinrichs [27] described a specific basis recombination strategy to improve the condition number of D^k to order N^k. This strategy is also treated by Boyd in [13]. There is also the mapped Chebyshev approach introduced by Tal-Ezer [31] and further studied in [21]. This also reduces the condition number of the kth order derivative to $O(N^k)$.

5.3.3 Extended Precision

The most straightforward way to keep ill-conditioning from interfering with calculations which require high precision is to take advantage of extended precision floating point numbers. These are now available on a number of architectures and supported by many compilers [5].

Using quadruple precision allows us to, for example, calculate on a 100 by 100 grid using Chebyshev collocation and still retain 16 digits of accuracy. In a less extreme case, if 8 digits sufficed, we could use 1000 Chebyshev nodes in each direction.

Calculating at extended precision incurs its own costs and does nothing to mitigate the poor numerical behavior of the underlying problem. However, it does keep numerical errors at bay without requiring any extra algorithmic or analytic complexity. Thus for practical problems that are otherwise intractable it offers a realistic approach.

5.3.4 Alternate Basis Sets

It is well known that other global basis sets, while lacking the optimality properties of the Chebyshev polynomials, lead to much better conditioned derivative operators. Depending on the specifics of the problem it may be advisable to work in a basis of Legendre or Jacobi polynomials.

We note that the methods developed here for Chebyshev collocation methods are equally applicable in any other basis. The low-rank structure we are taking advantage of stems from the smoothness of the operator itself, not the basis in which it is represented.

5.3.5 Iterative Refinement

If we are only able to obtain relatively low accuracy, but have our system in factored form it may be desirable to employ a few cycles of iterative refinement. By calculating our backward error and performing another fast solve we may extend the accuracy of the solution.

5.3.6 Exploiting Limited Precision

We can also use the fact that our final accuracy is known to be limited to our advantage. Whether it is due to the order of our timestepping scheme or due to the conditioning of the differential operator, we often know that we are working to less than machine precision.

Similar to a wavelet representation, the rebus is a thresholded representation and captures the operator to arbitrary but finite precision. The thresholding is similar to that of an economy SVD, where basis vectors corresponding to small singular values are discarded.

If we know that our solution will not have more than, say, 10 digits of accuracy we can threshold more aggressively, discarding factors of the off-diagonal blocks corresponding to singular values smaller than 10^{-10}. This results in lower rank representations of the off-diagonal blocks and hence faster computation speed.

By remaining aware of the limitations imposed on the accuracy of our solution in timestepping methods we may at least dispense with unnecessary computations and work with the relevant components of our problem. If only very low accuracy is required, as would be the case for a preconditioner or an iterative refinement step, we may work with a coarse approximation to the operator and proceed through the calculations very rapidly.

5.4 Higher Dimensional Problems

5.4.1 Introduction

Using the rebus structure to solve linear systems arising from pseudospectral discretizations of PDEs has the greatest impact on the time and memory required when the problem is large. If the problem of interest consists of timestepping an initial condition on 10 Chebyshev nodes, then our operator will be a 10 by 10 matrix, and the overhead of using a rebus will not be worth bearing.

In a timestepping setting, the accuracy of the solution will almost certainly be determined by the temporal step size. This is due to the fact that the solutions at each step are spectrally accurate in the number of discretization points. If the solution we are seeking is smooth, there will likely be no gain from refining the grid in space. Of course, if the solution contains shocks or boundary layers that we wish to track precisely, the spatial convergence will not be spectral and in this case large numbers of collocation points are justified.

The other important scenario where large matrices will be needed is higher dimensional problems. In fluid or plasma dynamics, we will likely want to track interfaces and boundary layers and also, if possible, work in three dimensions. In quantum mechanical simulations, even higher dimensions may be needed. Using conventional methods, pseudospectral methods are prohibitive unless the problem can be diagonalized (that is, periodic boundary conditions imposed). It would seem that rebus methods should have the greatest impact in this high dimensional arena.

We saw in Section 3.3.2 that a modern computer (in 2005) with 1.5GB of RAM could not call the standard LAPACK routine `dgetrf` on a matrix larger than about 10^5 by 10^5 without running out of memory. For a one dimensional problem, this is an ample number of collocation points. However, in a two-dimensional problem, a matrix this size would correspond to around 300 collocation points in each dimension. For a three-dimensional problem, it would correspond to only 40 collocation points in each direction.

5.4.2 Intrinsic One-Dimensionality of the Rebus

The rebus "automatically" captures HSS structure in one dimension, but this is not necessarily true in higher dimensions. It may seem that if a discretization of $|x_i - x_j|^2$ demonstrates the requisite smoothness and decay properties in one dimension, then it should still possess them in two and three dimensions and the rebus representation should remain equally effective in capturing them.

We will consider two pictures of the rebus, the hierarchical block matrix and the binary tree structure, and see why further work is required for higher dimensional problems. First consider the block structure in Figure 2.1. As discussed in Section 1.2, the prototype for a hierarchically semiseparable matrix might be a banded matrix or its inverse. In either case, we capture and full blocks near the diagonal, and then efficiently represent the low-rank outer blocks by their low-rank factors.

A banded matrix is however unlikely to arise in practice unless it is from an inherently one-dimensional problem. A finite difference discretization, for example, will give rise to a banded matrix in one dimension but to a block-banded matrix in two dimensions. This stems from the fact that the discretization points have a natural order in one dimension and so the regularity of the interactions reveals itself in a predictable way. The diagonal of the matrix always corresponds to a nodes interaction with itself, and as we move away from the diagonal the distance between the nodes corresponding to our row and column increases.

In a two-dimensional problem, there is no natural order of the nodes, and so we cannot assume that the smoothness of the interactions will be adequately captured. Numbering the nodes by counting across nodes in rows has the problem that it fails to account for proximity between say the nodes exactly halfway along adjacent rows. Similar remark hold for numbering along columns.

The same difficulty can be noticed if we consider the binary tree representation of the rebus. The leaves capture local interactions, and the branches, representing the translation operators, mediate between these localities on different scales. However, the binary tree structure assumes that one unit of our domain on a coarser scale can be split into exactly two neighboring units on a finer scale. Again, it is clear what this means in a one-dimensional space, but it requires some interpretation in the higher dimensional case.

5.4.3 Nested Dissection Ordering

As long ago as the early 1970s, similar issues arose in the ordering of sparse matrices. It is clear in the sparse case that how we order our unknowns can have a large effect on the time required for a direct solution of a set of equations. Computationally, this can be thought of as *fill-in*. This term refers to the observation that as rows of a sparse matrix are eliminated, new non-zeros appear in the remaining rows. It also has a deeper interpretation in terms of the adjacency graph of the sparse matrix. This allows many tools from graph theory to be used to determine optimal orderings.

In 1973, papers by Alan George [23] and Birkhoff and George [12] introduced the idea of nested dissection ordering. For the sparse matrices arising in two-dimensional finite element methods, this ordering of the nodes was shown to reduce the computational cost to $O(N^3)$ for an $N \times N$ grid of elements. In a usual row by row ordering the cost of solving the system would be $O(N^4)$.

A sparse matrix usually arises when we restrict ourselves to *local* interactions as in the case of finite difference and finite element discretizations. Whenever we are looking at adjacencies, we will have local interactions of this kind.

If we generalize this to the case where the elements do not have to be adjacent to interact, but interact less with separation, we can still realize considerable benefits. Consider a model problem as follows. Form a mesh on the unit square $(0, 1) \times (0, 1)$ by dividing it into N^2 squares each of side length $1/N$ Each smaller square interacts with the others with a force

$$f_{ij} = \frac{1}{d_{ij}} = \frac{1}{|x_i - x_j|}$$

where i and j run over all elements from 1 to N^2. Now consider the matrix A, where $a_{i,j} = f_{ij}$.

Figure 5.4 shows the interaction of elements in an $N \times N$ grid, ordered by nested dissection. Each element of the $N \times N$ grid exerts a force proportional to f_{ij} as above. The darker the cell at $a_{i,j}$, the stronger the interaction between the cell numbered i and the cell numbered j. The diagonals are the darkest, representing the action of a cell on itself.

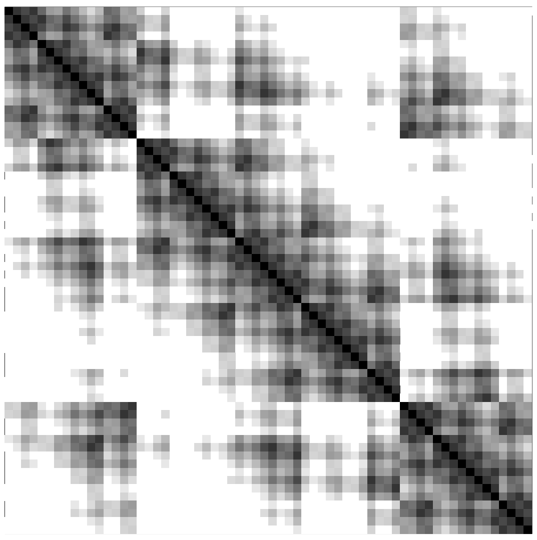

Figure 5.4. A two-dimensional interaction in nested dissection ordering.

The two important properties restored to the interaction matrix are the smoothness of the interaction and the concentration of strong interactions around the diagonal. This resembles the situation for the interaction in one dimension.

So far nested dissection orderings have been used in rebus applications in two and three dimensional problems. More sophisticated treatments of multi-dimensional problems will probably rely on some extensions of the rebus such as a block-rebus structure or a rebus whose components are not matrices but themselves rebuses. Without these extensions (and their associated algorithms) the nested dissection ordering seems to lead to the most efficient representation of operators in higher dimensions.

Chapter 6

Conclusions and Topics for Further Research

In the thesis so far we have described and demonstrated a number of rebus algorithms and applied them to the solution of PDEs. Now that these basic results are in place, and have a full code-base to support them, we are in a good position to extend these applications to even faster and more accurate PDE solvers.

The two most immediate applications are LU accelerated timestepping and implementing the fourth-order exponential time differencing method. Both of these are treated in this section. Leveraging the existing results and code, each of these should be relatively easy to implement.

6.1 LU Accelerated Timestepping

In Chapter 4, we demonstrated fast timestepping solutions for PDEs with a range of boundary conditions. Each timestep required, among other operations, the solution of a linear system. At the time these experiments were done, there was no way to factorize the rebus to allow for easy solution against multiple right hand sides. In particular, there was no LU factorization.

Given that now this algorithm has been determined, implemented and tested, an appropriate first application would be a new implementation of the PDE solvers that take advantage of the factorization to greatly improve their performance. As noted in Section 3.3.2, the forward and back solving only takes 15% of the solution time. It is natural to ask what the overhead of the LU factorization is, compared to an alternative direct solver that does not explicitly factorize the rebus.

Compared to the solution times presented by Chandrasekaran, Gu and Lyons in [17], the solution times in Table 3.4 (comprising an LU decomposition, a forward solve and a back solve) are slower by a factor of two. Let the time for solving a given linear system in rebus form, using the methods presented in [17], be c. The cost of solving against M right hand sides will be $c \cdot M$. Using LU decomposition once, and then forward and back substitution, the cost will be $1.7 \cdot c + 0.3 \cdot c \cdot M$. The break-even point, at which we are better off using the factorization, arrives before $M = 3$. Thus, in any timestepping problem we will gain by using the LU factorization.

The only complication in implementing this is the application of boundary bordering. If we no longer have the original rebus, and wish to work only with the LU factors then we need to determine the correct way to impose boundary conditions on the factors.

In the case of constant boundary conditions of either Dirichlet or Neumann type, we can simple apply the boundary bordering before factorization. Even in the case of time-varying Dirichlet conditions, we can use a fast leaf-update on the L factor without problems. However, the case of time-varying Neumann conditions may require some extra work.

6.2 Alternate Timestepping Schemes for Stiff Nonlinear PDEs

Linearly implicit methods are a popular choice for solving PDEs with a nonlinear contribution and a stiff linear term. However, there are other methods available. Kassam and Trefethen compare the available schemes in [11], and suggest that the integrating factor (IF) and exponential time differencing (ETD) schemes may be superior choices. Methods of the ETD type appear to have been introduced by Beylkin, Keiser and Vozovoi in [10].

These alternate methods rely on the observation that the linear part of the equation can be solved exactly by a matrix exponentiation. As ever, a suitable explicit step is sought for the nonlinear terms so as to avoid an iterative solution of a nonlinear system.

To take a timestep using this type of method, the linear term must be inverted, exponentiated or raised to a power. In the scheme favored in [11], (ETDRK4, an exponential time differencing technique based on the fourth order Runge-Kutta scheme) all of these must be done. This poses no particular problem if we have periodic boundary conditions in one dimension. However, if this is not the case and we cannot diagonalize our operator we must be able to efficiently invert, exponentiate and multiply a full matrix.

In calculating the timestep in the IF and ETD schemes, matrix exponentiation plays a central role. The integrating factor which multiplies both sides of the PDE is of the form $e^{\mathcal{L}h}$, and needs to be calculated. If the linear operator is constant, it only needs to be calculated once. If we are working in Fourier space, the operator can be rendered diagonal and again, the calculation is straightforward. Note that the exponential will not itself be sparse.

The more difficult case is that of a function $\mathcal{L}$ on a finite domain with physical boundary conditions. If a pseudospectral Chebyshev discretization is used, the matrix exponential becomes very expensive to evaluate. It is an $O(N^3)$ operation and even for relatively modest N may be the rate determining step.

The matrix exponential arises frequently in physics and mathematics due to its special role in the solution of linear differential equations. However, like the matrix inverse, it is usually avoided in numerics. The matrix exponential is well-studied numerically (see for example [28]), but has properties that have prevented it from being a practical choice in numerical applications.

Foremost among these undesirable properties is that calculating and applying exponentials of a general matrix requires working with dense matrices. Even if the original matrix is itself sparse, the exponential will not be. Thus, even for finite difference or finite element methods, solution via matrix exponentiation is not practical. An exception arises when the matrix to be exponentiated is diagonal or circulant (and so diagonalizable by Fourier transform). These can arise in applications with periodic boundary conditions. However, the need for the Fourier transform further limits it to constant coefficient problems.

Beylkin has shown that in a wavelet basis, general matrix exponentiations may be represented with sparse matrices (to finite but arbitrary precision). He then proceeded to apply this insight in solving the linear terms of PDEs [10]. We suggest a similar approach, but using the rebus to capture the structure in a conventional basis.

Consider the case of the standard finite difference discretization of the second derivative operator.

$$D_2 = \frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} = \frac{1}{h^2}\cdot A. \tag{6.1}$$

We will consider matrix A without the scaling factor $\frac{1}{h^2}$, as this cannot affect the rank structure. If we evaluate the matrix e^A we find it has the rank structure shown in Figure 6.1 and Table 6.1. These numerical ranks correspond to singular values of magnitude greater than 10^{-12}. The ranks of the off-diagonal blocks stay constant, even as the block size grows exponentially. Decreasing the tolerance to 10^{-6} has the effect of causing the off-diagonal ranks to become constant at rank 4. This means that although the matrix e^A is dense, the off-diagonal blocks have singular values that fall off very rapidly towards zero and may be very efficiently compressed by the rebus structure.

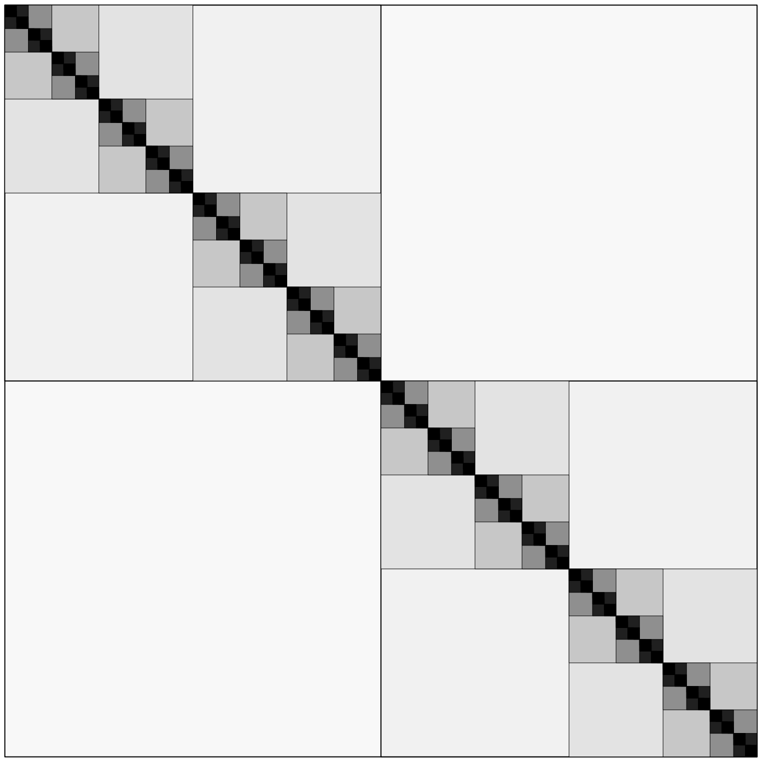

Figure 6.1. Rank structure of e^A with shading proportional to rank.

N	Rank of $N \times N$ block	% of full rank
256	7	2.73
128	7	5.47
64	7	10.9
32	7	21.9
16	7	43.8
8	7	87.5
4	4	100
2	2	100

Table 6.1. Rank structure of the 512×512 e^A with tolerance 10^{-12}.

So, we see that if we need to apply the matrix exponential of derivative operators in a finite difference basis, a rebus representation will be a valuable tool.

Although we lose the sparsity of the original operator, the overall complexity of the operator is seen to be quite low.

Now let us consider the pseudospectral derivative operator. In this case, the Chebyshev derivative matrix D already has nontrivial off-diagonal structure.

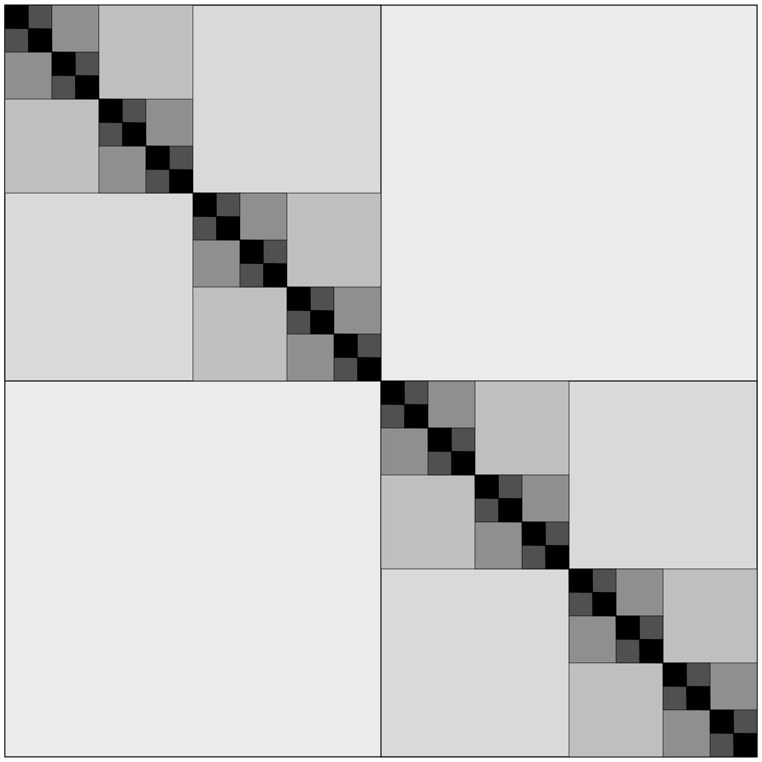

Figure 6.2. Rank structure of e^D in pseudospectral Chebyshev basis with shading proportional to rank.

In Figure 6.2 and Table 6.2 we see that the exponential of the pseudospectral derivative operator also has low-rank off-diagonal structure. In this case, we see the ranks are larger, but still grow very slowly once we get away from the diagonal.

Thus, in applications of the matrix exponential, we expect to benefit from the application of the rebus structure whether using a finite difference or pseudospectral discretization.

The matrix exponential is most often computed using the scaling and squaring

N	Rank of $N \times N$ block	% of full rank
256	20	7.8
128	19	14.8
64	16	25.0
32	14	43.8
16	11	68.8
8	8	100
4	4	100

Table 6.2. Rank structure of the 512×512 matrix of e^D in the pseudospectral Chebyshev basis with tolerance 10^{-12}.

algorithm favored by Golub and Van Loan in [25] and in the review papers [33], [34]. The calculation is accomplished with matrix multiplications and a matrix solve and is therefore easily implemented for the rebus structure. With an $O(N)$ matrix exponentiation, IF and ETD schemes could be applied to problems with a time-varying linear operator and to large problems with non-periodic boundary conditions.

In [10], the authors demonstrated a sparse matrix exponentiation for wavelet representations of strictly elliptic operators. We should be able to do something similar in a standard basis by:

1. using fast operators to construct the matrix exponential via scaling and squaring, and

2. implementing an ETDRK4 timestepping scheme which takes advantage of the fast rebus algebra.

A concern in implementing this scheme is that the exponential of an operator with hierarchically semiseparable structure might not itself contain the same structure. Just as the exponential of a sparse matrix need not be sparse, it could be that the exponential of a rebus will not have any of the rank structure or smoothness that the rebus is meant to take advantage of. The preliminary results above show that the exponentials do in fact retain the hierarchically semiseparable structure that we need. They are perfect candidates for rebus representation.

Bibliography

[1] A. W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, January 1985.

[2] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit–explicit Runge-Kutta methods for time-dependent partial differential equations. *Appl. Numer. Math.*, 25(2–3):151–167, 1997.

[3] U. M. Ascher, S. J. Ruuth, and B. Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM J. Num. Anal.*, 32:797–823, 1995.

[4] E. Asplund. Inverse of matrices $\{a_{ij}\}$ which satisfy $a_{ij} = 0$, $j > i + p$. *Math. Scand.*, 7:57–60, 1959.

[5] D. H. Bailey. A portable high performance multiprecision package. Technical Report RNR-90-022, NASA Ames Research Center, Moffett Field, CA 94035, 1992.

[6] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(6270):446–449, 1986.

[7] A. Bayliss, A. Class, and B. J. Matkowsky. Roundoff error in computing derivatives using the Chebyshev differentiation matrix. *J. Comput. Phys.*, 116:380–383, 1994.

[8] G. Beylkin, R. Coifman, and V. Rokhlin. Fast wavelet transform and numerical algorithm I. *Commun. Pure & Appl. Math.*, 44:141–183, 1991.

[9] G. Beylkin, N. Coult, and M. J. Mohlenkamp. Fast spectral projection algorithms for density-matrix computations. *J. Comput. Phys.*, 152(1):32–54, 1999.

[10] G. Beylkin, J. M. Keiser, and L. Vozovoi. A new class of time discretization schemes for the solution of nonlinear PDEs. *J. Comput. Phys.*, 147(2):362–387, 1998.

[11] G. Beylkin and K. Sandberg. Fourth-order time stepping for stiff PDEs. Technical Report NA-03/14, Oxford University, 2003.

[12] G. Birkhoff and A. George. Elimination by nested dissection. In J. F. Traub, editor, *Complexity of Sequential and Parallel Algorithms*, pages 221–269. Academic Press, New York, 1973.

[13] J. P. Boyd. *Chebyshev and Fourier Spectral Methods.* Springer-Verlag, New York, 1989.

[14] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, and A.-J. van der Veen. Fast stable solver for sequentially semi-separable linear systems of equations. *Lecture Notes in Computer Science*, 2552:545–554, 2002.

[15] S. Chandrasekaran and M. Gu. Fast and stable algorithms for banded plus semiseparable systems of linear equations. *SIAM Journal on Matrix Analysis and its Applications*, 5(2):373–384, 2003.

[16] S. Chandrasekaran and M. Gu. A divide and conquer algorithm for the eigendecomposition of symmetric block-diagonal plus semi-separable matrices. *Numerische Mathematik*, 96(4):723–731, 2004.

[17] S. Chandrasekaran, M. Gu, and W. Lyons. A fast and stable adaptive solver for hierarchically semi-separable representations. Technical Report UCSB Math 2004-20, U.C. Santa Barbara, 2004.

[18] S. Chandrasekaran, M. Gu, and T. Pals. Fast and stable algorithms for hierarchically semi-separable representations. *Submitted for publication*, 2004.

[19] R. Coifman, V. Rokhlin, and S. Wandzura. The fast multipole method for the wave equation: a pedestrian prescription. *IEEE Antennas Propag. Mag.*, 35(3):7–12, June 1993.

[20] M. Crouzeix. Une mèthode multipas implicite-explicite pour l'approximation des èquations d'èvolution paraboliques. *Numer. Math*, 35:257–276, 1980.

[21] W. S. Don and A. Solomonoff. Accuracy enhancement for higher derivatives using Chebyshev collocation and a mapping technique. *SIAM J. Sci. Comput.*, 18(4):1040–1055, 1997.

[22] B. Fornberg. *A Practical Guide to Pseudospectral Methods.* Cambridge University Press, Cambridge, UK, 1996.

[23] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[24] D. Gines, G. Beylkin, and J. Dunn. LU factorization of non-standard forms and direct multiresolution solvers. *Appl. Comput. Harmon. Anal.*, 5(2):156–201, 1998.

[25] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MA, 1996.

[26] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[27] W. Heinrichs. Improved condition number for spectral methods. *Math. Comp.*, 53:103–119, 1989.

[28] M. Hochbruck and Ch. Lubich. On Krylov subspace approximations to the matrix exponential operator. *SIAM J. Numer. Anal.*, 34(5), 1997.

[29] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles.* Taylor & Francis, Inc., 1988.

[30] P. Jones, J. Ma, and V. Rokhlin. A fast direct algorithm for the solution of the laplace equation on regions with fractal boundaries. *J. Comput. Phys.*, 113(1):35–51, 1994.

[31] D. Kosloff and H. Tal-Ezer. Modified chebyshev pseudospectral method with $O(N^{-1})$ time step restriction. *J. Comput. Phys.*, 104:457–469, 1993.

[32] W. Lyons, H. D. Ceniceros, S. Chandrasekaran, and M. Gu. Fast algorithms for spectral collocation with non-periodic boundary conditions. *J. Comput. Phys.*, 207(1):173–191, 2005.

[33] C. B. Moler and C. F. Van Loan. Nineteen dubious ways to compute the exponentional of a matrix. *SIAM Rev.*, 20(4):801–836, 1978.

[34] C. B. Moler and C. F. Van Loan. Nineteen dubious ways to compute the exponentional of a matrix, twenty-five years later. *SIAM Rev.*, 45(1):3–49, 2003.

[35] S. T. O'Donnell and V. Rokhlin. A fast algorithm for the numerical evaluation of conformal mappings. *SIAM J. Sci. Stat. Comput.*, 10(3):475–487, May 1989.

[36] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys.*, 60:187–207, 1985.

[37] V. Rokhlin. Rapid solution of integral equations of scattering theory in two dimensions. *J. Comput. Phys.*, 86(2):414–439, 1990.

[38] P. Starr. On the numerical solution of one-dimensional integral and differential equations. Technical Report YALEU/DCS/RR-888, Yale University, 1991.

[39] P. Starr and V. Rokhlin. On the numerical solution of two-point boundary value problems II. Technical Report YALEU/DCS/RR-802, Yale University, 1990.

[40] G. Strang and T. Nguyen. The interplay of ranks of submatrices. *SIAM Review*, 46(4):637–646, 2005.

[41] T. Tang and M. R. Trummer. Boundary layer resolving pseudospectral methods for singular perturbation problems. *SIAM J. Sci. Comput.*, 17:430–438, 1996.

[42] J. M. Varah. Stability restrictions on second order, three level finite difference schemes for parabolic equations. *SIAM J. Numer. Anal.*, 17:300–309, 1980.

[43] N. Yarvin and V. Rokhlin. A generalized one-dimensional fast multipole method with application to filtering of spherical harmonics. *J. Comput. Phys.*, 147(2):594–609, 1998.

[44] D. Zwillinger (Ed.). *Handbook of Differential Equations.* Academic Press, Boston MA, 3rd edition, 1997.