Master's Thesis

Experimenting with deflation-based preconditioning

Bart Dopheide

University of Groningen
Department of Mathematics
P.O. Box 800
9700 AV Groningen

October 2004

Preface

Since childhood, I have been interested in mathematics and computers in general. The latter interest was my hobby and I would not want to turn it into my profession. But I have seen a future in maths, so to study it, seemed the logical next step. I have chosen the university of Groningen ("Rijksuniversiteit Groningen" or R*u*G for short) as it is one with a local and friendly character. A fortunate side effect of my choice was that computational science houses in the same building, so from time to time I was able to attend programming courses to keep my interest in computational science "hot" so to speak.

In the end, my study and my hobby joined in the form of my specialisation: numerical mathematics. The beauty of maths combined with the elegance of high performance computing. This I told my supervisor (F.W. Wubs) who would try to find a project in which I could lay my programming skills combined with my gained knowledge in high performance computing. My supervisor attended a presentation of Jason Frank and Kees Vuik [1] about deflation. Afterwards, he spoke with Vuik about it with respect to using deflation on current research on parallelisation of MRILU at the R*u*G. It was possible to use deflation according to Vuik. Wubs then knew he had a nice project for me.

The goal for us was to implement the method described in that article, to verify the results, to understand the results to some degree and to implement another (similar) method to suit current research done at the R*u*G. In short, chapter 2 is Frank and Vuik redone for symmetric problems and chapter 3 is about deflation for non-symmetric problems.

This report is the result of two years work. I never worked for this long on a project and I couldn't have done this without the support of friends and relatives.

In no particular order I hereby want to thank them

- Wibrich Kooistra. My girlfriend has always stood by my side and has always encouraged me to 'just do it' and 'finish it'.
- Ineke Kruizinga. For chats and cryptographic puzzles during 'lunch hours'.
- Fred Wubs. My supervisor who never gave up on me.
- Arie de Niet. My vice-supervisor who had a refreshing look on the matter.
- My father and mother. They phoned me every Wednesday to get an update on my progress.
- My grandpa. On every family occasion he asked me when he would see the historical R*u*G-building once more.
- Every forgotten person that *should* have been mentioned...

The work could have been done in less time, though, but other activities intervened. My job as student assistant took at least a couple of months. Doing administrative tasks for the local badminton club cost several weeks. I have to build a curriculum vitae, don't

I? I followed a couple of courses which consumes lots of time. And of course the holidays; they took a couple of months. All in all, two years is not so bad a score...!

With this report, my college years will probably end. Time for a few weeks of activities not involving mathematics, i.e. activities that have not gotten enough attention last two years ;-).

Groningen, October 2004 **Bart Dopheide**

Contents

1 Introduction

Assume we want to solve x from the problem

$$Ax = b, \tag{1}$$

where A is symmetric positive definite (SPD). These kinds of systems are encountered when a finite volume/difference/element method is used to discretise an elliptic partial differential equation (PDE). It is well known that the convergence rate of the conjugate gradient method is bounded as a function of the condition number of the system matrix to which it is applied.

When A is the discrete approximation of an elliptic PDE, the condition number can become very large as the grid is refined, thus slowing down convergence. In this case it is advisable to solve, instead, a preconditioned system $K^{-1}Ax = K^{-1}b$, where the symmetric positive definite preconditioner K is chosen such that $K^{-1}A$ has a more clustered spectrum or a smaller condition number than that of A. Furthermore, a system with the matrix K must be cheap to solve relative to the improvement it provides in convergence rate. A final desirable property is that it should parallelise well, especially on distributed memory computers.

Probably the most effective preconditioning strategy in common use is to take $K = LL^T$ to be an incomplete Cholesky (IC) factorisation of A. When there are a few isolated extremal eigenvalues, another preconditioning strategy has proven successful; *deflation*.

This report is about deflation and based on work of Kees Vuik. We implement the deflated CG method described in his article (see [1]). Furthermore, we put the method to the test and try to understand the results to a degree (chapter 2). We also study the parallèlisation of the method. Vuik covers mostly SPD problems, but we dive into non-symmetric problems, too (chapter 3).

2 Deflation for symmetric problems

In this chapter, we will introduce the notion of "projection". Projections play a key role in the deflation technique. We discuss a few interesting properties of projections and the potential of deflation before we start experimenting with the deflation technique. At the end of this chapter, we discuss the parallelisation of the deflated preconditioned CG method (DPCG).

2.1 Theory

The deflation technique tries to remove the smallest (or largest) eigenvalues from an iterative linear system solver. By doing so, the condition number of the problem is reduced which usually makes the solver use fewer iterations than before. Of course, the price we pay for removing eigenvalues has to be paid in the end by making a correction to the solution found.

Assume we want to solve $Ax = b$ where A is SPD. Let us define the projection P by

$$P := I - AZ(Z^TAZ)^{-1}Z^T, \quad Z \in \mathbb{R}^{z \times d} \tag{2}$$

where the columns of Z span the deflation subspace, i.e., the space to be projected out of the residual, and I is the identity matrix of appropriate size. d can be seen as the number of eigenvalues that are to be projected out of the residual. z has to match the number of columns of A.

Later on, we will see that we should not try to eliminate too many eigenvalues as it is counterproductive, so we should assume that $d \ll z$ in practical cases.

We assume that Z has rank d. Then the inverse of Z^TAZ exists and P has the following properties (which are easily proven):

- $P^2 = P$ [Projecting a projection does nothing.] (3)
- $P(I - P) = 0$ [P is perpendicular to $(I - P^T)$.] (4)
- $Z^TP = 0$ [P is perpendicular to Z.] (5)
- $PA = AP^T$ [PA is symmetric.] (6)
- $PAZ = 0$ [Z is in the nullspace of PA.] (7)

Lemma 2.1. *Let A be positive semidefinite and P a projection ($P^2 = P$). If PA is symmetric, it is positive semidefinite.*

Proof. By definition: $0 \leq u^TAu$ for all u. But then it also holds for $u = P^Tv$ where v is an arbitrary vector:

$$\begin{aligned} 0 \leq u^TAu &= (P^Tv)^TA(P^Tv) = v^T(PA)P^Tv = v^T(A^T(P^2)^T)v \\ &= v^T(A^TP^T)v = v^T(PA)v \end{aligned}$$

□

Example. To see that the condition number of PA may be better than that of A, consider the case in which Z is the invariant subspace of A corresponding to the smallest d eigenvalues. PA has d zero-eigenvalues since, by (7) $PAZ = 0$. Furthermore, since PA is symmetric, by the orthogonal diagonalisation theorem the remaining eigenspace, say Y, can be chosen in the orthogonal complement of column space of $\{Z\}$, i.e. $Z^TY = 0$ and thereby the convergence is determined by the condition number of Y^TPAY:

$$\kappa_{\text{eff}}(PA) = \kappa(Y^TPAY) = \frac{\lambda_n(A)}{\lambda_{d+1}(A)}.$$

Since this holds for any A, especially it holds for a preconditioned system, say $A_{\text{prec}}^{-1}A$.

In summary, deflation of an invariant subspace cancels the corresponding eigenvalues, leaving the rest of the spectrum untouched.

2.2 Algorithm DPCG

As d is relatively small, $A_{\text{deflated}} \equiv Z^TAZ$ may be easily computed and factored and is symmetric positive definite. Since $x = (I - P^T)x + P^Tx$ and because

$$(I - P^T)x = ((AZ(Z^TAZ)^{-1})Z^T)^Tx = Z(Z^TAZ)^{-1}Z^TAx = ZA_{\text{deflated}}^{-1}Z^Tb \tag{8}$$

can immediately be computed, we only have to compute P^Tx. Since $AP^Tx = PAx = Pb$, we can solve the deflated system

$$PA\tilde{x} = Pb \tag{9}$$

for $\tilde{x}$ using the conjugate gradient method and premultiply this by P^T. Obviously, (9) is singular and this raises a few questions. First, the solution $\tilde{x}$ may contain arbitrary components in the null space of PA, i.e. in the column space of $\{Z\}$. This is not a problem however, because the projected solution $P^T\tilde{x}$ is unique. Second, what consequences does the singularity of (9) imply for the conjugate gradient method? In theory, a positive semidefinite system can be solved as long as the right-hand side is consistent (i.e., as long as $b = Ax$ for some x). This is certainly true for (9), where the same projection is applied to both sides of the nonsingular system. Furthermore, because the null space never enters the iteration, the corresponding zero-eigenvalues do not influence the convergence.

2.3 Another view on deflation

In order to develop theory on a given subject, it always comes in handy to have multiple looks on the subject. That is why give another view on deflation.

In general, if a square matrix $\begin{bmatrix} V & W \end{bmatrix}$ has full rank, any x can be decomposed into $x = V\hat{x} + W\tilde{x}$. This also means that any given linear problem $Ax = b$ can be written as $A(V\hat{x} + W\tilde{x})$, or equivalently, $A \begin{bmatrix} V & W \end{bmatrix} \begin{bmatrix} \hat{x} \\ \tilde{x} \end{bmatrix} = b$. If we can choose V and W such that

$V^TAW = 0$, then we can solve the following system easier:

$$\begin{bmatrix} V^T \\ W^T \end{bmatrix} A \begin{bmatrix} V & W \end{bmatrix} \begin{bmatrix} \hat{x} \\ \tilde{x} \end{bmatrix} = \begin{bmatrix} V^TAV & V^TAW \\ W^TAV & W^TAW \end{bmatrix} \begin{bmatrix} \hat{x} \\ \tilde{x} \end{bmatrix} = \begin{bmatrix} V^T \\ W^T \end{bmatrix} b.$$

That is, we can first compute $\hat{x}$ from $V^TAV\hat{x} = V^Tb$ and then $\tilde{x}$. In the symmetric case ($A = A^T$), the problem even gets decoupled because $(W^TAV)^T = V^TAW = 0$. This is precisely what happens in the deflation approach if we take $V = Z$ then $(I - P^T)$ projects any vector onto the space spanned by Z: $(I - P^T)x = Z(Z^TAZ)^{-1}Z^TAx$. Then W and $\tilde{x}$ are implicitly, not uniquely defined by $W\tilde{x} = P^Tx$ because $x = (I - P^T)x + W\tilde{x}$. It holds that Z and W are perpendicular in an inner product based on A:

$$PAZ = 0 \Rightarrow x^TPAZ = 0 \ \forall x \Rightarrow Z^TAP^Tx = 0 \ \forall x \Rightarrow Z^TAW\tilde{x} = 0 \ \forall \tilde{x} \Rightarrow Z^TAW = 0,$$

which is favourable.

2.4 Choices for deflation

Deflation of an eigenspace cancels the corresponding eigenvalues without affecting the rest of the spectrum. This has led some authors to try to deflate with "nearly invariant" subspaces (possibly obtained during iteration), and led others to try to choose in advance subspaces which represent the extremal modes. We will investigate both approaches.

We will call the first one *subdomain deflation* as suggested in [1]. The domain is split up into non-overlapping subdomains. Vertically in p and horizontally in q subdomains. For each subdomain, we create a deflation vector which is one on its subdomain and zero on the others. We take the set of all these vectors for Z.

This choice of deflation subspace is related to domain decomposition. The projection $(I - P^T)x$ (see (8)) can be seen as a subspace correction in which each domain is agglomerated into a single cell.

Note that the matrix $A_{\text{deflated}} \equiv Z^TAZ$, the projection of A onto the deflation subspace Z, has sparsity pattern similar to that of A. We will see that the effective condition number of PA (κ_{eff}) improves as the number of subdomains is increased (for a fixed problem size). However, this implies that the dimension of A_{deflated} also increases, making direct solution expensive.

In the second approach we will use exactly computed eigenvectors to investigate the maximal effects possible after which it is possible to try to choose simple(r) vectors that resemble the eigenvectors for extremal modes.

2.4.1 Consequences of other choices than subdomain deflation

Since subdomain deflation is cheap, it is interesting to have some insight in what other choices might cost. Although other choices might lead to better convergence (as they might cancel the smallest or largest eigenvalues better), we only consider the computational cost of PAx, where x is an arbitrary vector.

We note that if Z becomes a full matrix, AZ and Z^TAZ also become full matrices (independent of the problem A). In the main loop of DPCG, Z is used only in computing PAx:

$$\begin{aligned} v &= Ax \\ PAx &= v - (AZ)(Z^TAZ)^{-1}Z^Tv \end{aligned}$$

- AZ is constant and can be computed before the main loop; that makes it an order 1 computation.
- $(Z^TAZ)^{-1}$ is also constant and can be computed (in factored form) before the main loop which makes it an order 1 computation.
- Z^Tv can cost up to pq as much calculations (since all columns of Z are completely filled instead of only $\frac{1}{pq}$).
- The computation of the whole $(AZ)(Z^TAZ)^{-1}Z^Tv$ as parts of the previous temporary results requires not considerably more work.

If $d \ll z$, then Z^Tv is only a small matrix-vector computation it will hardly have any effect on the speed in terms of wall-clock time even if it takes pq as much computations.

2.5 Test set

In order to test the deflation technique, we have to create a test set. We choose to solve the 2D Laplace equation: $\varphi_{xx} + \varphi_{yy} = 0$. The Dirichlet boundaries are chosen such that the solution corresponds to the temperature in a heated room. One side is kept at 25°C while the other sides are kept to 15°C, see figure 1. The discretisation is a standard five-point one.

2.5.1 Grid

We choose a *square*, equidistant grid of $mq \times np$ points, where p is the number of subdomains in the vertical direction and q in the horizontal. See figure 2. Unknowns are numbered in a lexical order (left to right, top to bottom). The resulting matrix A is of size $mqnp \times mqnp$.

2.5.2 Preconditioner

In A, a subdomain is connected to its neighbour(s). When we disconnect all these pq subdomains, we obtain the preconditioner we use in the test set. Observe that the preconditioner consists in fact of pq problems that can be solved independently which allows for parallelisation. We will call this preconditioner A_{prec}.

There is another choice that we will make which is based on the technique called *Gustafsson modification*. All ties between subdomains are not simply dropped, but are added to the main diagonal instead. We will call this preconditioner G_{prec}. Note that this

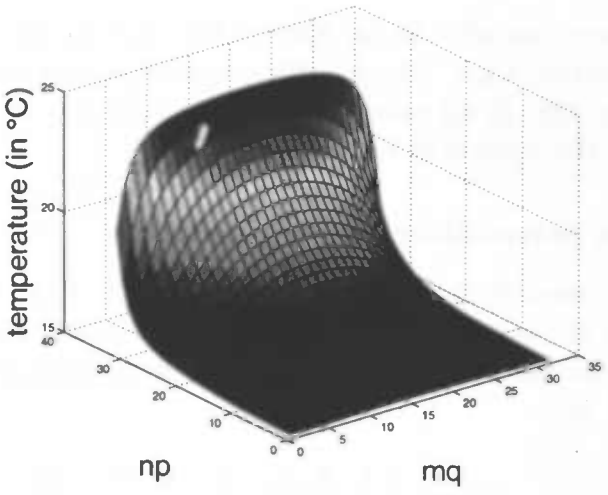


Figure 1: The solution to the stationary heated room problem (32 by 32).

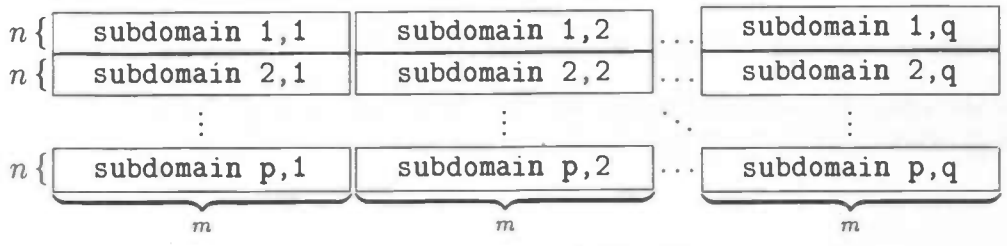


Figure 2: The division of the grid in subdomains.

preconditioner is basically singular if both p and q are greater than 2: the interior subdomains have no ties to the boundary anymore making it Neumann subdomains. Adding one to the last element of each singular subdomain cancels the singularities.

2.5.3 Numerical results

The DPCG-algorithm takes, among others things, a preconditioner which can be given in factored form. We will use no preconditioner, a complete and an incomplete Cholesky factorisation of our preconditioner A_{prec} or G_{prec}. For the incomplete ones, we will use `luinc` from MATLAB ® with `droptol`=0.1.

For the deflation space, we choose between subdomain and no deflation and deflation based upon eigenvectors. Domains can be split up in two directions in various configurations. Currently, DPCG solves $A_{\text{prec}}x = r_n$ directly, but we note that this might as well be solved in an iterative manner, too. That technique is destined to be faster than the direct approach.

With the computers available to us, 128 by 128 was the largest problem we can investigate within reasonable time. Consequently, most tests are based on the heated room problem of size 128 by 128. In all cases, we will use an all zero starting vector and require a precision of 10^{-6} in the 2-norm of the residual.

2.5.3.1 CG without preconditioning nor deflation

First, for comparison, we run the test set on standard CG: no deflation and no preconditioner. Since there is no preconditioner, the choices for p and q are irrelevant. See table 1. As expected, the number of iterations rises by a factor of 2 when the total number of unknown is increased by 4.

$mq = np$	1	2	4	8	16	32	64	128	256	512
Iteration needed to reach precision	1	2	6	21	45	90	176	349	694	1378

Table 1: Total number of iterations with neither a preconditioner nor deflation (standard CG).

2.5.3.2 CG with preconditioning but without deflation

To investigate the effect of preconditioning, we run the test set without deflation, but with preconditioning. See table 2, 3, 4 and 5. Each table covers a different preconditioning technique.

q \ p	1	2	4	8	16	32	64	128
1	1	40	54	72	96	133	188	266
2	31	42	60	75	100	130	186	267
4	53	61	61	83	107	142	190	269
8	72	79	85	86	116	148	198	275
16	95	103	108	117	122	161	208	282
32	133	137	142	148	161	172	227	297
64	187	188	192	198	209	227	243	323
128	266	268	269	274	282	297	324	349

Table 2: Total number of iterations using a complete Cholesky factorisation of A_{prec}. No deflation applied.

If we choose $p = 1$ and $q = 1$, then both A_{prec} and G_{prec} are equal to A. The preconditioned system $A^{-1}Ax = A^{-1}b$ is 'solved', which requires just one iteration.

The tables clearly show that using *square* subdomains is better than stretched ones.

Giving each subdomain a size of 1×1 reduces the preconditioner A_{prec} to a diagonal matrix. This does not help much since $A_{\text{prec}}^{-1}A$ is in essence the same as A; they only differ by a factor of 4. This explains why the number of iterations for $p = 128$ and $q = 128$ in tables 1, 2 and 4 are the same. The same holds for G_{prec} in tables 3 and 5.

q \ p	1	2	4	8	16	32	64	128
1	1	1	12	32	75	169	330	599
2	6	6	17	38	90	186	350	629
4	15	17	88	150	232	380	637	1000
8	34	38	153	162	232	318	491	783
16	79	91	244	235	224	282	352	570
32	170	186	386	324	283	244	314	442
64	333	350	667	503	388	335	295	379
128	614	636	1252	897	613	479	380	349

Table 3: Total number of iterations using a complete Cholesky factorisation of G_{prec}. No deflation applied.

q \ p	1	2	4	8	16	32	64	128
1	122	147	143	149	162	181	205	266
2	146	139	152	158	169	181	216	267
4	141	152	146	155	168	191	217	269
8	149	158	155	156	169	192	223	275
16	161	169	167	170	177	198	231	282
32	180	182	189	192	198	214	249	297
64	205	216	219	224	232	249	263	323
128	266	268	269	274	282	297	324	349

Table 4: Total number of iterations using a incomplete Cholesky factorisation of A_{prec}. No deflation applied.

q \ p	1	2	4	8	16	32	64	128
1	122	200	203	197	202	195	226	599
2	210	231	247	254	260	264	264	630
4	209	241	258	258	275	284	272	1000
8	205	244	256	260	269	278	281	783
16	210	251	253	250	255	266	297	570
32	197	247	242	233	233	248	296	442
64	538	584	668	506	391	338	296	379
128	613	634	1252	897	613	479	380	349

Table 5: Total number of iterations using a incomplete Cholesky factorisation of G_{prec}. No deflation applied.

The smaller the subdomains, the more information is lost, the slower the convergence (in number of iterations).

The idea of preconditioning is to reduce the number of iterations with a bit of extra calculations. This is observed for A_{prec} as the number of iterations is always fewer than 349 (the no-preconditioning case). But G_{prec} seems to have an unfavourable effect: while spending extra calculations it also *increases* the number of iterations sometimes. Since

most subdomains are still almost singular (small eigenvalue), the condition number of $G_{\text{prec}}^{-1}A$ is quite high resulting in extra iterations. Note that (subdomain) deflation will cancel those small eigenvalues. Consequently, the effect noticed here should not be that worrying.

2.5.3.3 CG without preconditioning but with deflation

To investigate the effect of deflation, we run the test set with subdomain deflation, but without preconditioning. See table 6.

q \ p	1	2	4	8	16	32	64	128
1	286	286	318	314	314	313	313	313
2	266	266	262	261	260	260	260	260
4	280	280	196	217	211	207	204	204
8	268	268	212	110	117	115	115	115
16	270	270	206	117	56	60	60	59
32	273	273	203	115	60	29	31	30
64	273	273	203	115	60	31	15	15
128	273	273	202	115	59	30	15	0

Table 6: Total number of iterations using no preconditioner at all. Subdomain deflation applied.

$p = q = 1$ uses only 1 deflation vector, namely an all one vector. This one vector reduces the number of iterations by 20% (compare with table 1).

Better yet, it seems we can solve a problem with 0 iterations! It is true, but the computations have shifted to computing a direct inverse: Z becomes the identity matrix which means that P reduces to an all zero matrix. CG has to solve $0Ax = 0b$. Any vector suffices. The bulk of the work is now done computing $ZA_{\text{deflated}}^{-1}Z^Tb$ which simplifies to $A^{-1}b$.

The more subdomains we have, the more deflation *seems* to help, but more subdomains actually mean that deflation shifts it work to computing $ZA_{\text{deflated}}^{-1}Z^Tb$. It is quite possible that wall clock time is not at its best at minimum nor maximum number of subdomains; there is a trade-off probably. Unfortunately, this trade-off is not visible in the number of iterations. To investigate the trade-off-point would require a timed series, but it depends on the number of processors used and optimisations made et cetera. Nevertheless, in table 7, such a timed series is performed on 1 processor. For simplicity, we assume that $PAx = Pb$ can be solved completely in parallel and computing $ZA_{\text{deflated}}^{-1}Z^Tb$ is restricted to one processor. In our situation (with pq hypothetical processors), the trade-off-point is $p = q = 32$.

2.5.3.4 CG with preconditioning and deflation

To see the full potential of subdomain deflation, we have to use preconditioning and subdomain deflation at the same time. See table 8, 9, 10 and 11.

$p = q$	1	2	4	8	16	32	64	128
number of iterations	286	266	196	110	56	29	15	0
time used for solving $PAx = Pb$	16.6	15.5	11.8	6.9	3.9	3.1	5.1	0
$\frac{\text{time used for solving } PAx=Pb}{pq}$	16.6	3.9	0.74	0.11	0.015	0.0030	0.0013	0
time used for computing $ZA_{\text{deflated}}^{-1}Z^Tb$	0.020	0.019	0.019	0.020	0.023	0.040	0.16	1.2
total time 1 processor	16.6	15.5	11.8	6.9	3.9	3.1	5.2	1.2
total time pq processors	16.6	3.9	0.76	0.13	0.048	0.043	0.16	1.2

Table 7: Timed series. No preconditioning used. Subdomain deflation is applied. Cheapest solution are marked.

q \ p	1	2	4	8	16	32	64	128
1	1	37	53	60	79	110	154	219
2	36	41	52	56	71	96	131	185
4	50	55	42	55	62	79	105	146
8	55	63	55	34	42	51	65	86
16	72	78	63	41	25	30	37	48
32	96	103	79	51	30	17	21	27
64	134	141	106	65	37	21	12	15
128	191	196	146	86	47	26	15	0

Table 8: Total number of iterations using a complete Cholesky factorisation of A_{prec}. Subdomain deflation applied.

q \ p	1	2	4	8	16	32	64	128
1	1	1	33	51	81	140	225	329
2	25	28	41	57	88	148	235	324
4	37	39	37	45	58	88	134	173
8	53	56	45	35	40	53	72	91
16	84	88	58	40	26	32	40	48
32	145	147	87	53	32	19	23	27
64	232	235	133	72	40	23	13	15
128	323	325	172	90	48	27	15	0

Table 9: Total number of iterations using a complete Cholesky factorisation of G_{prec}. Subdomain deflation applied.

The effects of both preconditioning and deflation are clearly visible. The optimal size of a subdomain is the square one. For complete factorisation, the number of iterations 'starts' with 1 and 'ends' 0. Here, too, will the minimum wall clock time not be in one of the endpoints. Towards the highly stretched subdomains the number of iterations rises dramatically.

If we compare table 10 to tables 4 and 6 we see that deflation and preconditioning have synergy; the combination is at least as good as the best of the parts and often even much better. Unfortunately, this cannot be said of Gustafsson preconditioning and deflation.

q \ p	1	2	4	8	16	32	64	128
1	96	114	119	120	125	138	161	219
2	114	103	99	97	103	116	139	185
4	112	103	73	85	86	93	110	146
8	117	100	83	45	51	57	67	86
16	124	109	84	51	27	32	37	48
32	136	123	93	57	32	18	22	27
64	153	147	111	67	37	22	12	15
128	191	196	146	86	47	26	15	0

Table 10: Total number of iterations using a incomplete Cholesky factorisation of A_{prec}. Subdomain deflation applied.

q \ p	1	2	4	8	16	32	64	128
1	96	158	168	164	158	151	178	329
2	168	165	163	160	157	170	167	324
4	170	164	131	137	132	137	137	173
8	166	159	130	79	73	75	81	91
16	161	157	126	76	44	44	45	48
32	149	154	122	72	44	27	25	27
64	274	277	146	76	41	23	13	15
128	323	324	172	90	48	27	15	0

Table 11: Total number of iterations using a incomplete Cholesky factorisation of G_{prec}. Subdomain deflation applied.

They are rivals in a sense as they both try to cancel low frequent components of the error and the high-frequent ones are demped less.

2.6 Deflation based on eigenvectors

A deflation vector corresponding to an eigenvalue of the problem should cancel that eigenvalue. So, in theory, a set of deflation vectors corresponding to the smallest n eigenvalues should cancel those n eigenvalues leaving a better conditioned system to be solved.

Since we don't actually solve the *actual* problem but the *preconditioned* one, we base the eigenvectors on the preconditioned system.

For our Poisson test problem, the condition number (of the preconditioned system) decreases slower when the largest eigenvalues are cancelled, so we use eigenvectors based upon the smallest ones. In order to be able to compare results, we take just as many deflation vectors as before, but note that we can just as easily use fewer of more.

The results in table 12 should be seen as the best results deflation can have.

It is clear that this deflation technique outperforms subdomain deflation in terms of number of iterations (see table 8), but bear in mind that computing many eigenvectors is a costly business. The gain isn't that much, except when the subdomains are (highly)

q \ p	1	2	4	8	16	32	64	128
1	1	30	44	46	51	53	56	60
2	27	33	39	42	41	42	43	44
4	43	38	34	35	33	32	32	32
8	46	41	35	28	29	26	25	24
16	51	42	33	29	21	22	19	
32	53	42	32	26	22	16		
64	56	43	32	25	20			
128	60	45	33	24				

Table 12: Total number of iterations using a complete Cholesky factorisation of A_{prec}. Eigenvector deflation applied.

stretched, which is not very interesting.

From these results we can conclude that subdomain deflation might be a cheap yet powerful choice of deflation.

In practise, we *approximate* the eigenvalues (and thus eigenvectors) rather than use exact ones as they are cheaper to compute. Note that trying to find structure in the eigenvectors is also an option that will save time.

2.7 A parallel view on DPCG

The main loop of DPCG consists of updating vectors, computing scalars, computing inner products and some matrix-vector multiplications.

To parallelise the main loop involves standard techniques, but the computation of PAp deserves extra attention. This computation can be split up in several components:

Action	(Intermediate) result
• Compute $M := AZ$ before the main loop	
• Compute $A^{-1}_{\text{deflated}} = (Z^T M)^{-1}$ before the main loop (or compute a factored form (LU) for back-forward-solving)	$A^{-1}_{\text{deflated}} = (Z^T AZ)^{-1}$
• Compute $w := Ap$	
• Compute $\tilde{w} := Z^T w$	$\tilde{w} = Z^T Ap$
• Compute $\tilde{e} := A^{-1}_{\text{deflated}} \tilde{w}$	$\tilde{e} = (Z^T AZ)^{-1} Z^T Ap$
• Compute $PAp = w - M\tilde{e}$	$PAp = (I - AZ(Z^T AZ)^{-1} Z^T)Ap$

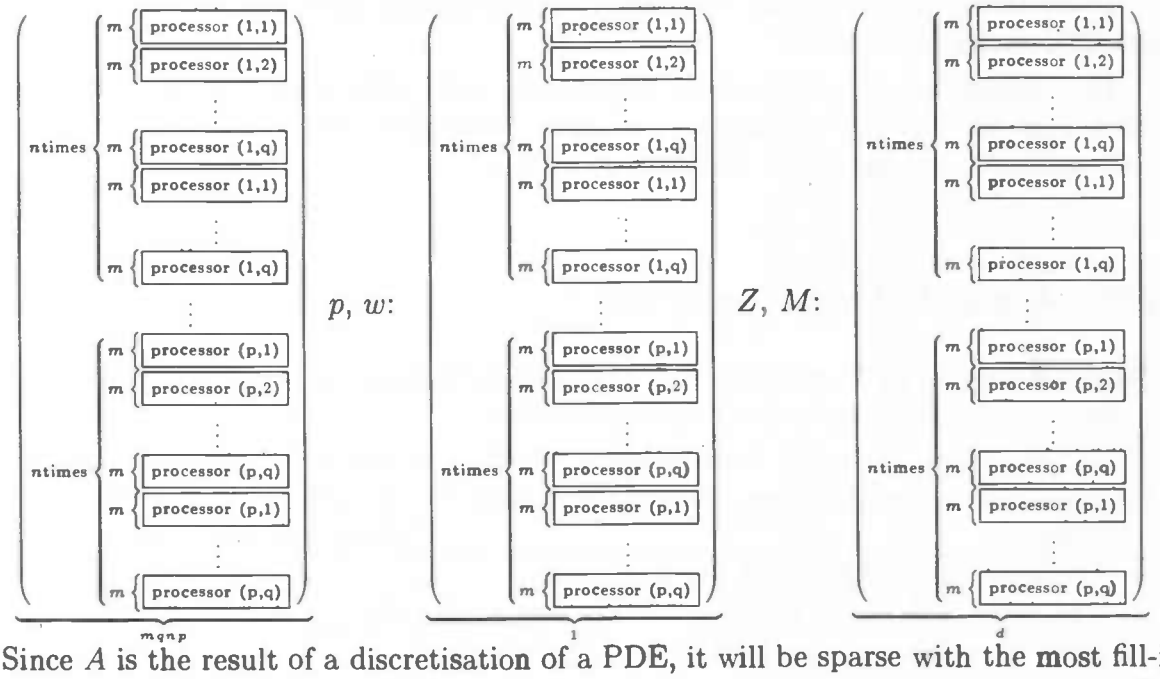
Assume, as before, that the grid consists of qp subdomains of size $m \times n$ and that we have d deflation vectors. Furthermore, assume $d \ll mqnp$ and we have qp processors[1]. We then have the following dimensions of the variables:

[1] For simplicity, we speak of processors while we could also speak of processes depending on the parallelisation method to be used.

Matrices involved				Vectors involved			
A	$\in$	$\mathbb{R}^{mqnp \times mqnp}$	(square matrix)	w	$\in$	$\mathbb{R}^{mqnp}$	
Z	$\in$	$\mathbb{R}^{mqnp \times d}$	(tall matrix)	p	$\in$	$\mathbb{R}^{mqnp}$	
M	$\in$	$\mathbb{R}^{mqnp \times d}$	(tall matrix)	$\tilde{w}$	$\in$	$\mathbb{R}^{d}$	(small vector)
A^{-1}_{deflated}	$\in$	$\mathbb{R}^{d \times d}$	(small matrix)	$\tilde{e}$	$\in$	$\mathbb{R}^{d}$	(small vector)

Since $d \ll mqnp$, $\tilde{w}$ and $\tilde{e}$ can be computed and stored on every processor. Similarly, since $d^2 \ll (mqnp)^2$, we decide to let A^{-1}_{deflated} exist on every processor. The distribution we propose is:

- A^{-1}_{deflated} stored (in factored form) on every processor.
- $\tilde{w}$ stored on every processor.
- $\tilde{e}$ stored on every processor.



Since A is the result of a discretisation of a PDE, it will be sparse with the most fill-in appearing on the block-diagonal. Other fill-in will appear in the off block-diagonals. This means that the bulk of the computation of $w = Ap$ can be done locally and almost no nearest neighbour communication (n.n.c.) is needed. In general, each boundary point of a subdomain requires n.n.c., but large subdomains have relatively few boundary points; in general only in the order of $2(m+n)$ neighbouring points have to be known. In contrast, the order of work involved for one subdomain is proportional to the number of non-zeroes (n.n.z.) of the subdomain which has order mn. It is obvious that for large subdomains $2(m+n) \ll mn$ holds.

The vector p should be divided over the processors similarly to A: each point in subdomain S should be stored on processor S. The same goes for Z and M, too. The computation of $\tilde{w} = Z^T w$ involves a gather-broadcast (g.b.) to compute. The vector $\tilde{e}$

can be computed locally. Finally, $w - M\tilde{e}$ can be computed *without* communication. We now give an estimate of the computational cost based on the discussed distribution over qp processors:

Computation	Order of work per subdomain	
	computations	communication
$w = Ap$	mn	n.n.c. of $\mathcal{O}(2(m+n))$ points
$\tilde{w} = Z^T w$	dmn	g.b. gathering: $\mathcal{O}(qpd)$ points broadcast: $\mathcal{O}(d)$ points
$\tilde{e} = A_{\text{deflated}}^{-1}\tilde{w}$	d^2	0
$PAp = w - M\tilde{e}$	dmn	0

In the subdomain deflation ($d = qp$) case, there is less work to be done, because only a fraction ($\frac{1}{pq}$) of every deflation vector is filled:

Computation	Order of work per subdomain	
	computations	communication
$w = Ap$	mn	n.n.c.
$\tilde{w} = Z^T w$	mn	g.b.
$\tilde{e} = A_{\text{deflated}}^{-1}\tilde{w}$	$(qp)^2$	0
$PAp = w - M\tilde{e}$	mn	0

All in all, we need in the order of $mn+(qp)^2$ computations per iteration (for subdomain deflation). If we denote the total size of A by N ($N = mqnp$) the order can be rewritten to $\frac{N}{qp} + (qp)^2$ and $mn + (\frac{N}{mn})^2$. This analysis shows that neither very small nor very large subdomains won't do any good to the work per iteration.

Unfortunately, we don't know an estimate of the number of iterations in terms of q, p, m, n. If we had such insights, we could multiply this estimate with the above found one. This new order would show the *total computational costs* which is obviously an important estimate.

The choice of preconditioner is also important. The preconditioner A_{prec} (or G_{prec}) is converted to an (in)complete Cholesky factorisation L and is then used in the main loop to update z: $z = (LL^T)^{-1}r$. This is only a process of back solving which can be done *locally* on each processor if the subdomains of A_{prec} are *decoupled*. This reduces the communications to zero for updating z. It also holds that if the preconditioner has decoupled subdomains, the factorisation can be computed locally with no communication (each processor stores its own part of the factorisation).

3 Deflation for non-symmetric or indefinite problems

Up to now, only SPD problems could be solved using deflation, but with only moderate modifications also non-symmetric problems can be tackled.

3.1 Projections

In the non-symmetric case, two projections are needed. Define the following projections:

$$\begin{aligned} P &:= I - AZ(Y^TAZ)^{-1}Y^T \\ Q &:= I - Z(Y^TAZ)^{-1}Y^TA, \end{aligned}$$

where $Z = [z_1 \dots z_d]$, $Y = [y_1 \dots y_d]$ and $z_1, \dots, z_d$ and $y_1, \dots, y_d$ are independent sets of deflation vectors. Observe that this is a generalization of the symmetric case: $P^T = Q$ if $Z = Y$ and A symmetric. P and Q have the following properties:

- $P^2 = P$, $Q^2 = Q$ [compare to (3)] (10)
- $P(I - P) = 0$, $Q(I - Q) = 0$ [compare to (4)] (11)
- $QZ = 0$, $Y^TP = 0$ [compare to (5)] (12)
- $PA = AQ$ [compare to (6)] (13)
- $PAZ = 0$, $Y^TAQ = 0$ [compare to (7)] (14)

Since $u = (I - Q)u + Qu$ and because

$$(I - Q)u = Z(Y^TAZ)^{-1}Y^TAu = Z(Y^TAZ)^{-1}Y^Tf$$

can immediately be computed, we need only to compute Qu. Since $AQu = PAu = Pf$, we can solve the deflated system

$$PA\tilde{u} = Pf$$

for $\tilde{u}$ using GMRES (or any other appropriate Krylov-subspace solver) and premultiply this by Q.

3.1.1 Preconditioning

In this section, we show that providing GMRES with PA, L, U and Pb and using Q to find the contribution $Q\tilde{u}$, where $\tilde{u}$ is the solution obtained by GMRES, is the same as providing it with $\hat{P}$, $\hat{A}$ and $\hat{P}b$ and using $\hat{Q}$ to find the contribution $U\tilde{u}$, where $\hat{A}$ is the preconditioned matrix A. $\hat{P}$ and $\hat{Q}$ are defined below.

Suppose $A \approx LU$ then

$$\begin{aligned}\hat{P} &:= L^{-1}PL \\ &= I - L^{-1}AU^{-1}UZ(Y^TL(L^{-1}AU^{-1})UZ)^{-1}Y^TL \\ &= I - \hat{A}UZ(Y^TL\hat{A}UZ)^{-1}Y^TL, \qquad \hat{A} := L^{-1}AU^{-1} \\ &= I - \hat{A}\hat{Z}(\hat{Y}^T\hat{A}\hat{Z})^{-1}, \qquad \hat{Z} := UZ, \hat{Y} := L^TY\end{aligned}$$

and

$$\begin{aligned}\hat{Q} &:= UQU^{-1} \\ &= U(A^{-1}PA)U^{-1} \\ &= (UA^{-1}L)(L^{-1}PL)(L^{-1}AU^{-1}) \\ &= \hat{A}^{-1}\hat{P}\hat{A}.\end{aligned}$$

This $\hat{P}$ and $\hat{Q}$ have the same properties as P and Q: $\hat{P}^2 = \hat{P}, \hat{Q}^2 = \hat{Q}, \hat{P}\hat{A} = \hat{A}\hat{Q}, \hat{P}\hat{A}\hat{Z} = 0, \hat{Y}^T\hat{A}\hat{Q} = 0$, so anything said about P and Q can also be said about $\hat{P}$ and $\hat{Q}$.

3.2 Driven cavity test set

In order to test the performance of DGMRES we have to create a (non-symmetric) test problem. Since the mathematics department of the University of Groningen is also involved in computational fluid dynamics, it seems logical to solve a relatively easy flow problem: a Stokes problem.

3.2.1 Continuous Stokes equations

We will compute the steady state of a flow of an incompressible viscous fluid in a square cavity. The flow is driven by a constantly moving upper lid that drags the fluid along. We will assume that the motion in terms of the Reynolds number is calm enough that the non-linear (convection) terms of the Navier-Stokes equations may be dropped. All boundaries are considered to have the "no-slip" condition. The equations we want to solve are:

$$\triangle u - \frac{\partial p}{\partial x} = 0 \qquad \text{[Horizontal momemtum equation]} \quad (15)$$

$$\triangle v - \frac{\partial p}{\partial y} = 0 \qquad \text{[Vertical momemtum equation]} \quad (16)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \qquad \text{[Continuity equation]} \quad (17)$$

3.2.2 Discrete Stokes equations

The pressure p is defined in the centre of a control volume and the velocities at the edges. The horizontal velocity u is defined in the centre of the right edge and the vertical velocity v in the centre of the lower edge. This is called a staggered grid. See also [2].

Our discrete versions of the flow equations are:

$$4u_{i,j} - (u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}) + (p_{i-\frac{1}{2},j} - p_{i+\frac{1}{2},j}) = 0 \quad (15')$$

$$4v_{i,j} - (v_{i-1,j} + v_{i,j-1} + v_{i+1,j} + v_{i,j+1}) + (p_{i,j-\frac{1}{2}} - p_{i,j+\frac{1}{2}}) = 0 \quad (16')$$

$$(u_{i+1,j} - u_{i-1,j}) + (v_{i,j+1} - v_{i,j-1}) = 0. \quad (17')$$

The resulting coefficient matrix will be singular as only the pressure *gradient* is computed. Later on, we will see that this has some consequences we will have to deal with. We will include bogus points (see 4) as it turned out it was handier from a programmers point of view.

Solutions typically look like figure 3.

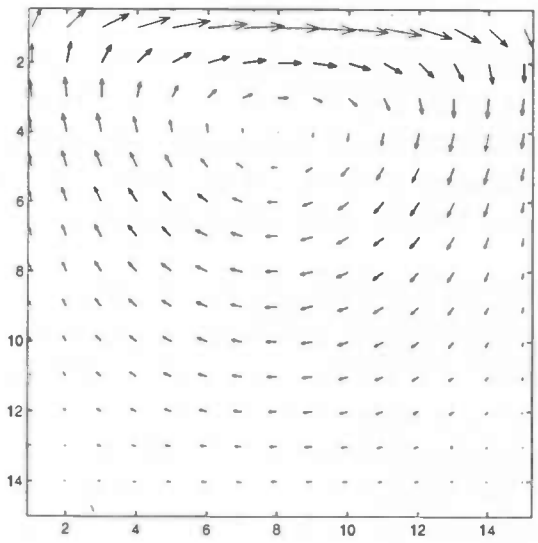


Figure 3: Typical solution of a driven cavity problem.

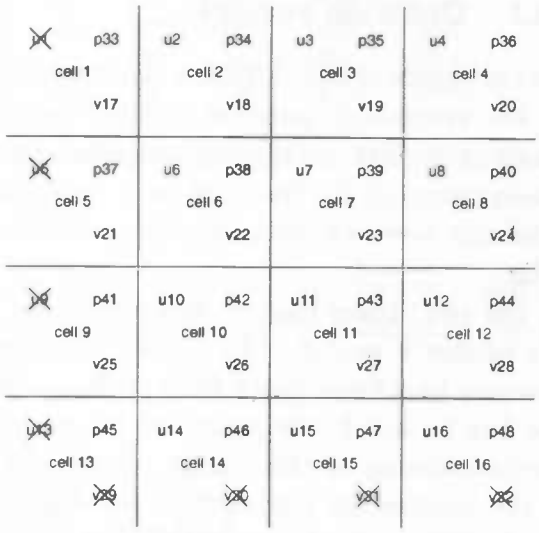


Figure 4: Driven cavity problem made up of 16 cells. Dummy variables are marked.

3.2.3 Grid

Like in the Poisson test set, we choose a square, equidistant grid of $mq \times np$ points, where p is the number of subdomains in the vertical direction and q in the horizontal. See figure 2. (Now, p is used twice, but the context will always be clear about which meaning should be applied.)

3.2.4 Preconditioner

Similar to Poisson, we disconnect all subdomains. This gives us our preconditioner A_{prec}. A small problem arises with the subdomain that includes the lower-right side as this subdomain is essentially a smaller Stokes problem, thus having a singularity.

An incomplete factorisation does not suffer from this as the factorisation will likely be non-singular. Only when dealing with very small subdomains (in the order of 2 by 2), the incomplete one is singular. We then replace any 0 with 1 on the diagonal of the upper triangular matrix.

A complete factorisation will be singular and this negatively influences the convergence, so we "fix" it by adding 0.1 to the diagonal of the coefficient matrix corresponding to the discrete continuity equation of the lower right pressure point.

No Gustafsson modification based preconditioner will be used as the Poisson test set indicated it is counter productive.

3.2.5 Deflation vectors

For the choice of the deflation vectors, the conditioning of the deflated matrix is important.

For symmetric positive definite matrices it is obvious to take $Y = Z$. This choice, which is in fact a Galerkin approach, can still be used if the matrix becomes slightly non-symmetric, for instance in a convection-diffusion problem. If the matrix is (almost) indefinite however, then the deflated matrix may become very ill-conditioned or even singular.

For the Stokes matrix, we could make use of the finite element theory for the selection of a proper Y and Z. The matrix itself can be thought of as a restriction of the continuous operator to a finite space built by finite elements. It is known that for the Stokes equation one has to satisfy the so-called inf-sup condition in order to avoid the above mentioned ill-conditioning of the matrix. Now, the deflated matrix can be viewed as a restriction of the continuous operator to a subspace of the original finite element space. Also for this subspace one has to satisfy the inf-sup condition. This will restrict the choice of the deflation vectors, since these vectors define how linear combinations of the original finite element basis functions are made in order to arrive at the basis functions for the subspace.

3.2.5.1 Subdomain deflation

Although we have theory as mentioned above, we decide to use subdomain deflation once more as it has proven in the heated room problem to have good qualities.

Subdomain deflation is not as straightforward as with Poisson though; the row sum in the coefficient matrix for every discrete continuity equation is zero. Computing a row sum is the same as multiplying the row with an all one vector, which occurs in subdomain deflation. Hence that applying this deflation gives a singularity in Y^TAZ when $Y = Z$. This can be overcome in many different ways. We did this in a rather crude way: set the last element of the last subdomain to 3 and set the first element of the first subdomain to 2. This yields better results than modifying only the first or last. As this choice seemed fair enough, no further research was done in optimising the choice for making Y^TAZ non-singular.

Subdomain deflation for u and v is the same as with Poisson.

3.2.6 Numerical results

A grid of 32×32 was doable within a fair amount of time, but 64×64 not. As a consequence, all testing is performed on a grid of 32 × 32 cells.

We use a restart after 40 iterations, require a relative precision of 10^{-6} in the 2-norm of the residual and an all zero starting vector. For incomplete LU factorisation, we use `droptol=0.1`. Using a restart after 20 iterations caused stagnation or too slow convergence.

$mq = np$	1	2	4	8	16	32	64	128
Iteration needed (restart=5)	0	5	92	300	1387	8665	58500	393769
Iteration needed (restart=10)	0	5	54	158	726	4411	28766	195701
Iteration needed (restart=20)	0	5	21	114	395	2321	14755	97725
Iteration needed (restart=40)	0	5	21	80	248	1170	7456	49227
Iteration needed (restart=80)	0	5	21	54	163	685	3765	23981
Iteration needed (restart=160)	0	5	21	54	126	530	1956	11956
Iteration needed (restart=320)	0	5	21	54	126	297	1205	5695

Table 13: Total number of inner GMRES iterations with neither a preconditioner nor deflation.

As we can see in table 13, the restart value has quite a large influence on the convergence behaviour. For large problems, the number of iterations is reduced by a factor of two when the restart value is increased by a factor of two.

q \ p	1	2	4	8	16	32
1	998	631	637	630	624	628
2	668	371	277	267	262	263
4	639	330	171	142	137	136
8	592	318	147	66	62	62
16	574	313	148	63	35	32
32	558	312	148	62	31	0

Table 14: Total number of iterations using no preconditioner at all. Subdomain deflation applied.

Table 14 shows the bare effects of subdomain deflation. As with Poisson, only 1 deflation vector reduces the number of iterations by 10+%. If we take 16, we only need about 15% of the original number of iterations while computing a direct inverse of 48 × 48 is peanuts. The effects seen in table 6 are essentially the same as table 14.

Tables 15 and 16 are comparable to tables 2 and 8 respectively; square subdomains are favoured over stretched ones, preconditioning is quite effective and preconditioning with subdomain deflation is a killer combination. For example, using 16 subdomains ($p = 4$, $q = 4$) in table 16 leaves a mere 27 iterations instead of the reference value of 1170. The extra work is only an inverse of 16 × 16.

q \ p	1	2	4	8	16	32
1	1	15	23	51	159	556
2	19	25	33	76	191	631
4	25	31	43	101	215	624
8	57	72	95	143	298	745
16	159	180	198	279	432	825
32	446	439	572	638	863	1117

Table 15: Total number of iterations using a complete LU factorisation of A_{prec}. No deflation used.

q \ p	1	2	4	8	16	32
1	1	17	24	48	117	268
2	19	23	29	46	73	144
4	25	29	27	27	43	72
8	51	41	29	20	23	35
16	114	77	43	23	14	18
32	264	146	76	36	18	0

Table 16: Total number of iterations using a complete LU factorisation of A_{prec}. Subdomain deflation applied.

q \ p	1	2	4	8	16	32
1	102	120	145	196	307	739
2	114	129	158	217	332	751
4	134	152	179	234	340	779
8	174	190	227	274	420	924
16	267	274	317	428	615	1265
32	717	742	824	1000	1360	1117

Table 17: Total number of iterations using an incomplete LU factorisation of A_{prec}. No deflation used.

q \ p	1	2	4	8	16	32
1	137	100	98	109	153	363
2	92	66	61	70	94	162
4	97	66	44	42	48	88
8	111	76	49	27	26	38
16	145	96	59	28	16	18
32	320	187	99	43	19	0

Table 18: Total number of iterations using an incomplete LU factorisation of A_{prec}. Subdomain deflation applied.

The behaviour observed in tables 17 and 18 is practically identical to that of tables 4 and 10.

3.3 Conclusion

We have seen that the numerical results are remarkably similar to that of the DPCG-method. We think it is fair to say that deflation can be used on non-symmetric, indefinite problems although there are certain pitfalls that need to be considered or studied.

4 Conclusion

We created two test sets to test the performance of deflation in two common used solvers: `CG` and `GMRES`. We have created preconditioners that divided the problem in uncoupled subdomains. This is useful for both *subdomain deflation* and high performance computing.

The numerical results show that preconditioning works well with large subdomains while subdomain deflation is more suited for small subdomains. Combined, they make a strong team. This team is "powerful" in the sense that it reduces the number of iterations needed by a fair amount. A drawback is that the optimal configuration of the size of the subdomains cannot be given a priori. This is related to the problem that fewer iterations does not always mean less computing time.

We tried to further reduce the number of iterations by means of Gustafsson's modification. It seems that Gustafsson's modification and deflation are rivals in a sense as they both try to cancel the low-frequent components in the error; the number of iterations did not go down, but went up instead. This can be understood from the fact that high-frequent components are demped less if Gustafsson's modification is applied.

A few difficulties arose when working with singular problems and a singular preconditioner; the deflation vectors had to be adapted and the preconditioner had to be made non-singular. Also, we saw that the restart value `DGMRES` seems to be of great influence on the convergence rate; too small a value resulted in stagnation.

The performed experiments show that deflation is promising.

5 Recommendations for further study

It would be highly interesting to see what performance can be reached when (subdomain) deflation is used in multiple processor / vector computers. We have done only theoretic experiments, yet they indicate that subdomain deflation will only then show its full potential.

To be true, we have no theory *why* deflation works for the Stokes equation; we only know that it works. It would be nice to have a theoretic foundation for deflation, which is likely to be based on finite element theory.

We have no theory about what the optimal size of subdomains is a priori. To see whether theory can be developed to estimate this size will be a challenge.

Only two test problems were tackled, but in quite some detail. It would be pleasing to see whether deflation in general produces good results.

A Code fragments

All our testing is done with MATLAB ® (version 6.5.1.199709 Release 13 (Service Pack 1) for GNU/Linux). We give the source of most functions used.

A.1 Heated room problem

The 1D Poisson coefficient matrix:

```
function A = polsson1D(n)
%POISSON1D    Trldiagonal matrix from the 1D Poisson equation
%            (sparse).
%   POISSON1D(N) returns the matrix corresponding to the 1D Poisson
%   equation of length N.
e = ones(n, 1);
A = spdiags([-e, 2*e, -e], -1:1, n, n);
```

The 2D Poisson coefficient matrix:

```
function A = poisson2D(m, n)
%POISSON2D    Block tridiagonal matrix from the 2D Polsson equatlon
%            (sparse).
%   A = POISSON2D(M, N) returns the matrix corresponding to the 2D
%   Poisson equation of size MxN.
A = kron(speye(n), poisson1D(m)) + kron(poisson1D(n), speye(m));
```

The simple precondltioner:

```
function [A_prec] = poisson2D_simple_preconditioner(m, n, q, p)
%POISSON2D_SIMPLE_PRECONDITIONER    A simple precondioner for the 2D
%                                   Poisson equation.
%   A_PREC = POISSON2D_SIMPLE_PRECONDITIONER(M, N, Q, P) returns a
%   preconditioned version of the matrix corresponding to the MQxNP
%   2D Poisson equation In which subdomains of size MxN are made.
%   POISSON2D_SIMPLE_PRECONDITIONER(M, N, 1, 1) glves the same
%   result as POISSON2D(M, N).
%
%   See also POISSON2D and POISSON2D_GUSTAFSSON_PRECONDITIONER.
A_prec = kron(speye(n*p*q),  poisson1D(m)) + ...
         kron(speye(p), kron(poisson1D(n), speye(m*q)));
```

The Gustafsson modified preconditioner:

```
function [G_prec] = poisson2D_gustafsson_preconditloner(m, n, q, p)
%POISSON2D_GUSTAFSSON_PRECONDITIONER    A preconditioner for the 2D
%                                       Poisson equation based upon
%                                       Gustafsson modificatlon.
%   G_PREC = POISSON2D_GUSTAFSSON_PRECONDITIONER(M, N, Q, P)
%   returns a preconditioned version of the matrlx correspondlng to
%   the MQxNP 2D Poisson equation in which subdomains of size MxN
%   are made. POISSON2D_GUSTAFSSON_PRECONDITIONER(M, N, 1, 1) gives
%   the same result as POISSON2D(M, N).
%
%   The result whould be singular is Q > 2 and P > 2, but that is
%   fixed by adding one every last element of each singular
%   subdomain.
%
%   See also POISSON2D and POISSON2D_SIMPLE_PRECONDITIONER.

% Preformance loss, but very easy to understand and implement ;-).
A = poisson2D(m * q, n * p);
A_prec = poisson2D_simple_preconditioner(m, n, q, p);
G_prec = A_prec + diag(sum(A, 2) - sum(A_prec, 2));

% Make non-singular.
for row = 1 : p - 2 % Loop over p-2 Interior subdomains.
  for col = 1 : q - 2 % Loop over q-2 interior subdomains.
    beginSubdomain = 1 + col * m + m * n * q * row;
    endSubdomain = beginSubdomain + (m - 1) + m * (n - 1) * q;
    e = endSubdomain;
    G_prec(e, e) = G_prec(e, e) + 1;
  end
end
```

The right hand side to make it a heated room problem:

```
function [b] = rhs_heated_room(m,n)
%RHS_HEATED_ROOM    Right hand side for a 2D heated room problem.
%   B = RHS_HEATED_ROOM(m, n) returns the right hand side for a 2D
%   heated room problem of size MxN.
%
%   See also POISSON2D, POISSON2D_TEST_SUITE.
b = zeros(m, n);
b(:,     1) = b(:,     1) + 15;
b(:, end) = b(:, end) + 15;
b(  1, :) = b(  1, :) + 15;
b(end, :) = b(end, :) + 25;
b = b(:);
```

The subdomain deflation vectors:

```
function [Z] = poisson2D_subdomain_deflation_vectors(m,n,q,p)
%POISSON2D_SUBDOMAIN_DEFLATION_VECTORS    Subdomaln deflation
%                                         vectors for POISSON2D.
%   Z = STOKES2D_SUBDOMAIN_DEFLATION_VECTORS(M,N,Q,P) creates PQ
%   deflation vectors; one for each subdomain. The vectors are
%   Dirac-delta like; each vector is only non-zero on its
%   subdomain. The vectors are constructed for Q horizonal
%   subdomains and P vertical ones of size MxN.
%
%   See also POISSON2D, POISSON2D_SIMPLE_PRECONDITIONER,
%   POISSON2D_GUSTAFSSON_PRECONDITIONER, RHS_HEATED_ROOM.

firstSubdomainFirstRow = sparse(m*q, 1);
firstSubdomaInFirstRow(1:m) = 1;
firstSubdomainOfFirstRowOfSubdomains = ...
  kron(ones(n, 1), firstSubdomainFirstRow);

Z_oneRowOfSubdomains = [];
for shiftSubdomain = 0 : m : m*q - m
  Z_oneRowOfSubdomains ...
  = [Z_oneRowOfSubdomaIns, ...
     circshift(firstSubdomainOfFirstRowOfSubdomains, ...
               shiftSubdomaIn)];

end
Z = kron(speye(p), Z_oneRowOfSubdomains);
```

We adapted the MATLAB ® CG method (pcg.m) to use deflation. As that file is copyrighted and we don't want to infringe copyright law, we give the patch file to generate our dpcg.m from pcg revision 1.18:

```
function [x,flag,relres,iter,resvec] = pcg(A,b,tol,maxIt,M1,M2,x0,var |
%PCG     Preconditioned Conjugate Gradients Method
%   X = PCG(A,B) attempts to solve the system of linear equations A*X |
%   for X.  The N-by-N coefficient matrix A must be symmetric and pos |
%   definite and the right hand side column vector B must have length |
%   A may be a function returning A*X.                                |
                                                                      >
%   PCG(A,B,TOL) specifies the tolerance of the method.  If TOL is [] |
%   then PCG uses the default, 1e-6.                                  |
%   PCG(A,B,TOL,MAXIT) specifies the maximum number of iterations.  I |
%   MAXIT is [] then PCG uses the default, min(N,20).                 |
                                                                      >
%   PCG(A,B,TOL,MAXIT,M) and PCG(A,B,TOL,MAXIT,M1,M2) use symmetric   |
%   positive definite preconditioner M or M=M1*M2 and effectively     |
%   solve the system inv(M)*A*X = inv(M)*B for X.  If M is [] then     |
%   a preconditioner is not applied.  M may be a function returning M |
                                                                      >
%   PCG(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess.  If X0 i |
%   then PCG uses the default, an all zero vector.                    |
%   PCG(AFUN,B,TOL,MAXIT,M1FUN,M2FUN,X0,P1,P2,...) passes parameters  |
%   to functions: AFUN(X,P1,P2,...), M1FUN(X,P1,P2,...), M2FUN(X,P1,P |
%   [X,FLAG] = PCG(A,B,TOL,MAXIT,M1,M2,X0) also returns a convergence |
%    0 PCG converged to the desired tolerance TOL within MAXIT iterat |
%    1 PCG iterated MAXIT times but did not converge.                 |
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
%    3 PCG stagnated (two consecutive iterates were the same).        |
%    4 one of the scalar quantities calculated during PCG became too  |
%      small or too large to continue computing.                       |
%   [X,FLAG,RELRES] = PCG(A,B,TOL,MAXIT,M1,M2,X0) also returns the     |
%   relative residual NORM(B-A*X)/NORM(B).  If FLAG is 0, RELRES <= T |
%   [X,FLAG,RELRES,ITER] = PCG(A,B,TOL,MAXIT,M1,M2,X0) also returns t |
%   iteration number at which X was computed: 0 <= ITER <= MAXIT.     |
                                                                      >
%   [X,FLAG,RELRES,ITER,RESVEC] = PCG(A,B,TOL,MAXIT,M1,M2,X0) also re |
%   vector of the residual norms at each iteration including NORM(B-A |
                                                                      >
%      [x,flag,rr,iter,rv] = pcg(A,b,tol,maxit,M);                    |
%   as inputs to PCG                                                  |
%      [x1,flag1,rr1,iter1,rv1] = pcg(@afun,b,tol,maxIt,@mfun,[],[],2 |
                                                                      >
%   See also BICG, BICGSTAB, CGS, GMRES, LSQR, MINRES, QMR, SYMMLQ, C |
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
                                                                      >
% Check for all zero right hand side vector => all zero solution     <
n2b = norm(b);                      % Norm of rhs vector, b         <
if (n2b == 0)                       % if    rhs vector is all zeros <
```

```
function [x,flag,res,iter,resvec] = ...
  dpcg(A,b,tol,maxit,M1,M2,x0,Z,varargin)
%DPCG    Deflated Preconditioned Conjugate Gradients Method
%   X = DPCG(A,B) attempts to solve the system of linear equations
%   A*X=B for X. The N-by-N coefficient matrix A must be symmetric
%   and positive definite and the right hand side column vector B
%   must have length N. A may be a function returning A*X.
%   DPCG(A,B,TOL) specifies the tolerance of the method. If TOL is
%   [] then DPCG uses the default, 1e-6.
%   DPCG(A,B,TOL,MAXIT) specifies the maximum number of
%   iterations. If MAXIT is [] then DPCG uses the default,
%   min(N,20).
%   DPCG(A,B,TOL,MAXIT,M) and DPCG(A,B,TOL,MAXIT,M1,M2) use
%   symmetric positive definite preconditioner M or M=M1*M2 and
%   effectively solve the system inv(M)*A*X = inv(M)*B for X. If M
%   is [] then a preconditioner is not applied.  M may be a
%   function returning M\X.
%   DPCG(A,B,TOL,MAXIT,M1,M2,X0) specifies the initial guess. If
%   X0 is [] then DPCG uses the default, an all zero vector.
%   DPCG(A,B,TOL,MAXIT,M1,M2,X0,Z) specifies the deflation vectors.
%   If Z is [] then DPCG uses the default, no deflation at all.
%   DPCG(AFUN,B,TOL,MAXIT,M1FUN,M2FUN,X0,Z,P1,P2,...) passes
%   parameters P1,P2,... to functions: AFUN(X,P1,P2,...),
%   M1FUN(X,P1,P2,...), M2FUN(X,P1,P2,...).
%
%   [X,FLAG] = DPCG(A,B,TOL,MAXIT,M1,M2,X0,Z) also returns a
%   convergence FLAG:
%    0 DPCG converged to the desired tolerance TOL within MAXIT
%      iterations
%    1 DPCG iterated MAXIT times but did not converge.
%    3 DPCG stagnated (two consecutive iterates were the same).
%    4 one of the scalar quantities calculated during DPCG became
%      too small or too large to continue computing.
%   [X,FLAG,RES] = DPCG(A,B,TOL,MAXIT,M1,M2,X0,Z) also returns the
%   absolute residual NORM(B-A*X). If FLAG is 0, RES <= TOL.
%   [X,FLAG,RES,ITER] = DPCG(A,B,TOL,MAXIT,M1,M2,X0,Z) also
%   returns the iteration number at which X was computed:
%   0 <= ITER <= MAXIT.
%   [X,FLAG,RES,ITER,RESVEC] = DPCG(A,B,TOL,MAXIT,M1,M2,X0,Z) also
%   returns a vector of the residual norms at each iteration
%   including the zeorth residual.
%      [x,flag,r,iter,rv] = dpcg(A,b,tol,maxit,M);
%   as inputs to DPCG
%      [x1,flag1,r1,iter1,rv1] =
%      dpcg(@afun,b,tol,maxit,@mfun,[],[],21);
%   See also BICG, BICGSTAB, CGS, GMRES, DGMRES, LSQR, MINRES, QMR,
%   SYMMLQ, CHOLINC, @.
%%DPCG: Added.
%   Adapted by Bart Dopheide (dopheide@fmf.nl, 2004) to use
%   deflation technique.
%%ABSOLUTE: Added.
%   Adapted by Bart Dopheide (dopheide@fmf.nl, 2004) to use
%   switch between absolute and relative tolerance. This switch can
%   only be operated from _within_ this function and not in the
%   function call. (Therefore, the help part for the function
%   applies to absolute tolerance only.) This is because the author
%   only needed absolute tolerance.
%
%%ABSOLUTE: Added. A switch for backward compatibility.
use_absolute = 1; % = 0 implies use relative tolerance.
```

```
    x = zeros(n,1);                    % then  solution is all zeros     <
    flag = 0;                          % a valid solution has been obtai <
    relres = 0;                        % the relative residual is actual <
    iter = 0;                          % no iterations need be performed <
    resvec = 0;                        % resvec(1) = norm(b-A*x) = norm( <
    if (nargout < 2)                                                      <
       itermsg('pcg',tol,maxit,0,flag,iter,NaN);                          <
    end                                                                   <
    return                                                                <
end                                                                       <

if ((nargin > 7) & isequal(atype,'matrix') & ...                          <
```

```
tolb = tol * n2b;                  % Relative tolerance
```

```
| %%DPCG: Adapted. Because one parameter is added, nargin shifts by 1.
> if ((nargin > 8) & isequal(atype,'matrix') & ...
> %%DPCG: Added. Check input parameter Z.
> if ((nargin >= 8) & ~isempty(Z))
>   if ~isequal(size(Z,1), n)
>     es = sprintf(['Deflation_matrix_must_be_a_matrix_with' ...
>                   '_%d_rows_to_match_the_problem_size.'],n);
>     error(es);
>   elseif size(Z,2) > n
>     es = sprintf(['Deflation_matrix_has_more_columns_than_A;' ...
>       '_Z_cannot_have_%d_independent_deflation_vectors.'], n + 1);
>     error(es);
>   end
> else
>   % No deflation wanted, so standard pcg will do just fine.
>   if ~existM1 M1 = []; end
>   if ~existM2 M2 = []; end
>   [x,flag,res,iter,resvec] = absolute_pcg(A, b, tol, maxit, ...
>                                           M1, M2, x, varargin{:});
>   return;
> end
>
> %%DPCG: Added. Introducing new variables.
> AZ = A * Z;
> A_deflated = Z' * AZ;
>
> % Check for all zero right hand side vector => all zero solution
> %%DPCG: Changed.
> n2b = norm(b - AZ * (A_deflated \ (Z' * b)));% Norm of rhs vector,P*b
> if (n2b == 0)                      % If     rhs vector is all zeros
>    x = zeros(n,1);                 % then   solution is all zeros
>    flag = 0;                       % a valid solution has been obtained
>    res = 0;                        % the absolute residual is actually 0/0
>    iter = 0;                       % no iterations need be performed
>    resvec = 0;                     % resvec(1) = norm(0)
>    if (nargout < 2)
>       %%DPCG: Cosmetic change. 'pcg' -> 'dpcg'.
>       itermsg('dpcg',tol,maxit,0,flag,iter,NaN);
>    end
>    return
> end
> %%ABSOLUTE: Added. Using a norm of 1 makes it absolute.
> if (use_absolute)
>   n2b = 1;
> end
>
| tolb = tol * n2b;                  % Absolute tolerance
> %%DPCG: Added. We want: r = P * b - P * A * x0.
> %%Rewrite:
> %% r = P * (b - A * x0)
> %%   = (I - A * Z * (Z'T * A * Z)^-1 * Z'T) * r0_nodeflation
> %%   = r0_nodeflation - AZ * (A_deflated \ (Z'T * r0_nodeflation))
> r = r - AZ * (A_deflated \ (Z' * r));
>
```

```
    relres = normr / n2b;
       itermsg('pcg',tol,maxit,0,flag,iter,relres);
```

```
    res = normr / n2b;
       %%DPCG: Cosmetic change. 'pcg' -> 'dpcg'.
       itermsg('dpcg',tol,maxit,0,flag,iter,res);

    %%DPCG: Added. Convert the deflated solution to the real solution.
    %% The solution x we find is not the solution we want as
    %% this is the solution to PAx = Pb.
    %% Compute P'T x
    %%   = x - Z (A_deflated)^-1 Z'T A x.
    %% Compute (I-P'T)u
    %%   =     Z (A_deflated)^-1 Z'T b.
    x = x + Z * (A_deflated \ (Z' * (b - A * x)));

    %%DPCG: Added. We want q = P * A * p.
    q = q - AZ * (A_deflated \ (Z' * q));
```

```
    normr = norm(b - A * x);
```

```
       %%DPCG: Changed. Compute correct norm.
       pcg_r = b - A * x;
       normr = norm(pcg_r - AZ * (A_deflated \ (Z' * pcg_r)));
```

```
    normr = norm(b - iterapp(afun,atype,afcnstr,x,varargin{:}));
```

```
       %%DPCG: Changed. Compute correct norm.
       pcg_r = b - iterapp(afun,atype,afcnstr,x,varargin{:});
       normr = norm(pcg_r - AZ * (A_deflated \ (Z' * pcg_r)));
```

```
    relres = normr / n2b;
    relres = normrmin / n2b;
```

```
    res = normr / n2b;
    res = normrmin / n2b;
%%DPCG: Added. Convert the deflated solution to the real solution.
%%      The solution x we find is not the solution we want as
%%      this is the solution to PAx = Pb.
%%      Compute P'T x
%%        = x - Z (A_deflated)^-1 Z'T A x.
%%      Compute (I-P'T)u
%%        =     Z (A_deflated)^-1 Z'T b.
x = x + Z * (A_deflated \ (Z' * (b - A * x)));
```

```
    itermsg('pcg',tol,maxit,i,flag,iter,relres);
```

```
    %%DPCG: Cosmetic change. 'pcg' -> 'dpcg'.
    itermsg('dpcg',tol,maxit,i,flag,iter,res);
```

For the same reasons, only the patch file for our version of pcg.m that uses absolute instead of relative tolerance:

```
function [x,flag,relres,iter,resvec] = pcg(A,b,tol,maxit,M1,M2,x0,var
```

```
function [x,flag,relres,iter,resvec] = ...
  absolute_pcg(A,b,tol,maxit, M1,M2,x0,varargin)
%   Adapted by Bart Dopheide (dopheide@fmf.nl, 2004) to use
%   absolute tolerance instead of relative.
%     "Bug": Names are not updated. I.e. relres is still named
%      relres while res, residual or absres would be better. In
%      essence, I made it work, nothing more nothing less. (It also
%      saves in the diff/patch file ;-).)
```

```
tolb = tol * n2b;                    % Relative tolerance
```

```
%%ABSOLUTE_PCG: Changed.
tolb = tol;                          % Absolute tolerance
```

Since we have several choices for preconditioning and deflation, we built a small test suite:

```
function [x,flag,res,iter,resvec] = ...
  poisson2D_test_suite(m,n,q,p,preconditioning_type,deflation_type)
%POISSON2D_TEST_SUITE    Function to aid testing of a 2D heated
%                        room problem based on the 2D Poisson
%                        equation.
%   X = POISSON2D_TEST_SUITE(M,N,Q,P) will try to solve a 2D heated
%   room problem of size MQxNP with subdomains of size MxN. To
%   solve with only one subdomain, this function can be
%   abbreviated to POISSON2D_TESTSUITE(M,N).
%
%   POISSON2D_TEST_SUITE(M,N,Q,P,PRECONDITIONING_TYPE) specifies the
%   preconditioning type to use. Options:
%     PRECONDITIONING_TYPE(1):
%       0: no preconditioner (default)
%       1: simple preconditioner without Gustafsson modification
%       2: simple preconditioner with    Gustafsson modification
%     PRECONDITIONING_TYPE(2) (only meaningful for
%     PRECONDITIONING_TYPE(1) ~= 0):
%       0: make no factorisation (default)
%       1: make a     complete Cholesky factorisation
%       2: make an incomplete Cholesky factorisation (droptol = 0.1)
%
%   POISSON2D_TEST_SUITE(M,N,Q,P,PRECONDITIONING_TYPE,DEFLATION_TYPE)
%   specifies the deflation type to use. Options:
%     DEFLATION_TYPE(1):
%       0: no deflation (default)
%       1: subdomain deflation
%       2: eigenvector deflation
%     DEFLATION_TYPE(2) (Only meaningful for DEFLATION_TYPE(1) == 2):
%       0: use smallest eigenvectors (default)
%       1: use largest eigenvectors
%   Note: If a preconditioner is requested, the eigenvectors will
%   be based on the generalized eigenvalue problem, otherwise, the
%   eigenvectors will be based on the coefficient matrix only.
%
%   See also POISSON2D_TEST_COMBINATION, POISSON2D,
%   RHS_HEATED_ROOM, DPCG, PCG.

if (nargin < 2 | nargin == 3)
  error('Not enough input arguments.');
end

if (nargin == 2)
  q = 1;
  p = 1;
end

if (nargin < 5)
  preconditioning_type = [];
end
preconditioning_type = ...
  extract_values(preconditioning_type, [0 0], [0 1 2], [0 1 2]);
preconditioner_type = preconditioning_type(1);
factorisation_type = preconditioning_type(2);
clear preconditioning_type;

if (nargin < 6 | isempty(deflation_type))
  deflation_type = 0;
end
deflation_type = ...
  extract_values(deflation_type, [0 0], [0 1 2], [0 1]);
if (deflation_type(1) == 2)
  eigenvectors_type = deflation_type(2);
end
deflation_type = deflation_type(1);


% Set constants.
droptol = 0.1;
tol = 1e-6;
maxit = 30000;
x0 = [];
%eigenvalues_method = 0; % Use eig().
eigenvalues_method = 1; % Use eigs().


% Create test problem.
A = poisson2D(m * q, n * p);
b = rhs_heated_room(m * q, n * p);


% Create preconditioner.
switch (preconditioner_type)
case 0 % No preconditioning.
  A_prec = [];
case 1 % Simple preconditioner.
  A_prec = poisson2D_simple_preconditioner(m, n, q, p);
case 2 % Gustafsson modification.
  A_prec = poisson2D_gustafsson_preconditioner(m, n, q, p);
end


% Create factorisation of preconditioner.
switch (factorisation_type)
case 0 % No factorisation.
  L = A_prec; U = [];
case 1 % Complete factorisation.
  U = chol(A_prec); L = U';
case 2 % Incomplete factorisation.
  U = cholinc(A_prec, droptol); L = U';
end


% Create deflation vectors.
switch (deflation_type)
case 0 % No deflation.
  Z = [];
case 1 % Subdomain deflation
  Z = poisson2D_subdomain_deflation_vectors(m, n, q, p);
case 2 % Eigenvector deflation
  switch (eigenvectors_type)
  case 0 % Eigenvectors based on smallest eigenvalues.
    if (preconditioner_type == 0)
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, [], q * p, 'SM', ...
                            eigenvalues_method);
    else
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, A_prec, q * p, 'SM', ...
                            eigenvalues_method);
    end
    Z = eigenVec(:, end + 1 - q * p : end);
  case 1 % Eigenvectors based on largest eigenvalues.
    if (preconditioner_type == 0)
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, [], q * p, 'LM', ...
                            eigenvalues_method);
    else
      [eigenVec, eigenVal] = ...
```

```
        compute_eigenvalues(A, A_prec, q * p, 'LM', ...
                            eigenvalues_method);
    end
    Z = eigenVec(:, 1 : q * p);
  end
end

% Everything is setup now. Let's compute!
[x, flag, res, iter, resvec] = dpcg(A, b, tol, maxit, L, U, x0, Z);

if (flag ~= 0)
  fprintf('No converge reached within %d iterations.\n', maxit);
  x = x0;
end
```

To generate the tables in this report, we used a wrapper function that calls the test suite many times:

```
function [iterationsMatrix] = poisson2D_test_combination( ...
  problem_size,preconditioner_type,deflation_type)
%POISSON2D_TEST_COMBINATION    Try all combinations of subdomains.
%
%   ITERATIONSMATRIX = POISSON2D_TEST_COMBINATION(PROBLEM_SIZE,
%   PRECONDITIONER_TYPE,DEFLATION_TYPE) tests combinations of a
%   square PROBLEM_SIZExPROBLEM_SIZE POISSON2D problem. This is
%   wrapper code for POISSON2D_TEST_SUITE. See that function for
%   details.
%
%   See also POISSON2D_TEST_SUITE, POISSON2D.

% Make a fancy header.
if (nargout == 0)
  fprintf('\\_p|\nq_\\|');
  for p = 2 .^ [0 : log2(problem_size)] fprintf('_%4d', p); end
  fprintf('\n---+');
  for p = 2 .^ [0 : log2(problem_size)] fprintf('-----', p); end
  fprintf('\n');
end

for q = 2 .^ [0 : log2(problem_size)]
  m = problem_size / q;
  if (nargout == 0) fprintf('%3d|', q); end
  for p = 2 .^ [0 : log2(problem_size)]
    n = problem_size / p;
    [x,flag,res,iter,resvec] = polsson2D_test_suite(m,n,q,p, ...
      preconditioner_type, deflation_type);
    iter_mat(1 + log2(q), 1 + log2(p)) = iter;
    if (nargout == 0) fprintf('_%4d', iter); end
  end
  if (nargout == 0) fprintf('\n'); end
end

if (nargout > 0) iterationsMatrix = iter; end
```

This is the input needed to generate most tables in chapter 2:

```
problem_size = 16;
no_p = 0; simp = 1; gust = 2;
no_f = 0; comp = 1; incp = 2;
no_d = 0; subd = 1; eigd = 2;
smal = 0; larg = 1;

% CG without preconditioning nor deflation.
for i = 2 .^ [ 0 : log2(128) ]
  [x,f,r,iter,rv] = poisson2D_test_suite(i,i,1,1);
  fprintf('%d_', iter)
end
fprintf('\n');

% CG with preconditioning but without deflation.
poisson2D_test_combination(problem_size, [simp, comp], no_d);
poisson2D_test_combination(problem_size, [gust, comp], no_d);
poisson2D_test_combination(problem_size, [simp, incp], no_d);
poisson2D_test_combination(problem_size, [gust, incp], no_d);

% CG without preconditioning but with deflation.
poisson2D_test_combination(problem_size, no_p, subd);

% CG with preconditioning and deflation.
poisson2D_test_combination(problem_size, [simp, comp], subd);
poisson2D_test_combination(problem_size, [gust, comp], subd);
poisson2D_test_combination(problem_size, [simp, incp], subd);
poisson2D_test_combination(problem_size, [gust, incp], subd);

for i = 2 .^ [ 0 : log2(16384) ]
  [x,f,r,iter,rv] = poisson2D_test_suite(i,i,1,1);
  fprintf('%d_', iter)
end
fprintf('\n');

% Deflation based upon eigenvectors.
poisson2D_test_combination(problem_size, [simp, comp], [eigd]);
```

A.2 Driven cavity problem

The 2D Poisson coefficient matrix:

```
function [A] = stokes2D(m,n)
%STOKES2D   Stokes Matrix from the 2D Stokes equation (sparse).
%
%   A = STOKES2D(M,N) returns the matrix corresponding to the 2D
%   Stokes equation of size MxN. Note that the right side as well
%   as the lower side are bogus points to ease up constructional
%   work for preconditioned versions of this matrix.
%
%   See also RHS_DRIVEN_CAVITY.

e = ones(max(m, n), 1);
no_dependence = sparse(m*n, m*n);
part = append_matrix(1, poisson1D(m-1));
almostSpeyeM = append_matrix(0, speye(m - 1));
e_adapted = [0; e(1:end-1)];

dependence_u_on_u = kron(speye(n), part) + ...
                    kron(poisson1D(n), almostSpeyeM);
dependence_u_on_p = kron(speye(n), ...
                         spdiags([e, -e_adapted], -1:0, m, m));

dependence_v_on_v = append_matrix(poisson2D(m, n-1), speye(m));
dependence_v_on_p = kron(spdiags([e -e], 0:1, n, n), speye(m));
dependence_v_on_p(end-m+1:end, end-m+1:end) = 0;

A = [dependence_u_on_u , no_dependence      , dependence_u_on_p; ...
     no_dependence     , dependence_v_on_v  , dependence_v_on_p; ...
     dependence_u_on_p', dependence_v_on_p', no_dependence     ];
```

The simple preconditioner:

```
function [A_prec] = ...
  stokes2D_simple_preconditioner(m,n,q,p,make_non_singular)
%STOKES2D_SIMPLE_PRECONDITIONER   Preconditioner for STOKES2D.
%
%   A_PREC =
%   STOKES2D_SIMPLE_PRECONDITIONER(M,N,Q,P,MAKE_NON_SINGULAR)
%   returns a preconditioner for the 2D Stokes matrix of
%   STOKES2D. The preconditioner is divided horizontally in Q
%   subdomains and vertically in P. The size of the subdomains is
%   MxN. If MAKENONSINGULAR is true (~0) then the preconditioner
%   is made non-singular.
%
%   A_PREC = STOKES2D_SIMPLE_PRECONDITIONER(M,N,Q,P) is an
%   abbreviation of STOKES2D_SIMPLE_PRECONDITIONER(M,N,Q,P,0).
%
%   A_PREC = STOKES2D_SIMPLE_PRECONDITIONER(M,N) is an
%   abbreviation of STOKES2D_SIMPLE_PRECONDITIONER(M,N,1,1).
%
%   See also STOKES2D, RHS_DRIVEN_CAVITY.

if (nargin < 5)
  make_non_singular = 0;
end
if (nargin < 3)
  q = 1;
  p = 1;
end

e = ones(max(m,n), 1);
noDependence = sparse(m*n*q*p, m*n*q*p);
almostSpeyeMQ = append_matrix(0, speye(m*q-1));
firstLineDependence_u_on_u = kron(speye(q), poisson1D(m));
firstLineDependence_u_on_p = kron(speye(q), ...
                                  spdiags([e -e], -1:0, m, m));

% Make a correction for bogus u's on right side.
firstLineDependence_u_on_u(1,1) = 1;
firstLineDependence_u_on_p(1,1) = 0;
if (max(size(firstLineDependence_u_on_u)) > 1)
  firstLineDependence_u_on_u(1,2) = 0;
  firstLineDependence_u_on_u(2,1) = 0;
end
neighboursHorizontal = kron(speye(n*p), ...
                            firstLineDependence_u_on_u);
neighboursVertical   = kron(speye(p), ...
                            kron(poisson1D(n), almostSpeyeMQ));
dependence_u_on_u = neighboursHorizontal + neighboursVertical;
dependence_u_on_p = kron(speye(n*p), firstLineDependence_u_on_p);

dependence_v_on_v = poisson2D_simple_preconditioner(m, n, q, p);
dependence_v_on_p = kron(speye(p), ...
                         kron(spdiags([e -e],0:1,n,n),speye(m*q)));

% Make a correction for bogus v's at the bottom.
dependence_v_on_v = dependence_v_on_v(1:m*n*q*p - m*q, ...
                                      1:m*n*q*p - m*q);
dependence_v_on_v = append_matrix(dependence_v_on_v, speye(m*q));
dependence_v_on_p(end-m*q+1:end, 1:end) = 0;

A_prec=[dependence_u_on_u , noDependence      , dependence_u_on_p; ...
        noDependence     , dependence_v_on_v  , dependence_v_on_p; ...
        dependence_u_on_p', dependence_v_on_p', noDependence     ];

if (make_non_singular)
  cell = (m*q - 1)*n*p + 1;
  point = cell + 2*m*q*n*p;
  A_prec(point, point) = A_prec(point, point) + 0.1;
end
```

The right hand side to make it a heated room problem:

```
function [b] = rhs_driven_cavity(m,n)
%RHS_DRIVEN_CAVITY   Right hand side for a 2D driving cavity
%                    problem.
%   B = RHS_DRIVEN_CAVITY(M,N) returns the right hand side for a
%   2D driving cavity problem of size MxN.
%
%   See also STOKES2D.
b = sparse(3 * m * n, 1);
b(1:m) = 1;

% Make upper bogus points also zero.
b(1) = 0;
```

The subdomain deflation vectors:

```
function Z = stokes2D_subdomain_deflation_vectors(m,n,q,p)
%STOKES2D_SUBDOMAIN_DEFLATION_VECTORS   Subdomain deflation vectors
%                                       for STOKES2D.
%
%   Z = STOKES2D_SUBDOMAIN_DEFLATION_VECTORS(M,N,Q,P) creates
%   3*PQ deflation vectors. Three for each subdomain being one for
%   u, one for v and one for p. The vectors are Dirac-delta like;
%   each vector is only non-zero on its subdomain for its
%   variable. The vectors are constructed for Q horizontal
```

```
%   subdomains and P vertical ones of size MxN. The vectors for P
%   are slighty different as to prevent singularities in the
%   deflated matrix Z'*STOKES2D(M*Q,N*P)*Z.
%
%   See also STOKES2D, STOKES2D_PRECONDITIONER, RHS_DRIVEN_CAVITY,
%   POISSON2D_SUBDOMAIN_DEFLATION_VECTORS.
```

```
P = poisson2D_subdomain_deflation_vectors(m, n, q, p);
V = P;
U = P;
P(end,end) = 3; % Prevent singularity.
P(1,1) = 2;     % Prevent singularity.
Z = append_matrix(U, V, P);
```

We adapted the MATLAB ® GMRES method (gmres.m) to use deflation. As that file is copyrighted and we don't want to infringe copyright law, we give the patch file to generate our dgmres.m from gmres revision 1.21:

```
function [x,flag,relres,iter,resvec] = gmres(A,b,restart,tol,maxit,M1
%GMRES  Generalized Minimum Residual Method.
%   X = GMRES(A,B) attempts to solve the system of linear equations A
%   X.  The N-by-N coefficient matrix A must be square and the right
%   column vector B must have length N.  A may be a function returnin
%   uses the unrestarted method with MIN(N,10) total iterations.

%   GMRES(A,B,RESTART) restarts the method every RESTART iterations.
%   is N or [] then GMRES uses the unrestarted method as above.

%   GMRES(A,B,RESTART,TOL) specifies the tolerance of the method.  If
%   then GMRES uses the default, 1e-6.
%   GMRES(A,B,RESTART,TOL,MAXIT) specifies the maximum number of oute
%   iterations. Note: the total number of iterations is RESTART*MAXIT
%   is [] then GMRES uses the default, MIN(N/RESTART,10). If RESTART
%   then the total number of iterations is MAXIT.

%   GMRES(A,B,RESTART,TOL,MAXIT,M) and GMRES(A,B,RESTART,TOL,MAXIT,M1
%   use preconditioner M or M=M1*M2 and effectively solve the
%   system inv(M)*A*X = inv(M)*B for X.  If M is [] then a preconditi
%   not applied.  M may be a function returning M\X.

%   GMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0) specifies the first initial
%   guess.  If X0 is [] then GMRES uses the default, an all zero vect

%   GMRES(AFUN,B,RESTART,TOL,MAXIT,M1FUN,M2FUN,X0,P1,P2,...) passes p
%   to functions: AFUN(X,P1,P2,...), M1FUN(X,P1,P2,...), M2FUN(X,P1,P

%   [X,FLAG] = GMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0) also returns a c
%   FLAG:
%    0 GMRES converged to the desired tolerance TOL within MAXIT iter
%    1 GMRES iterated MAXIT times but did not converge.

%    3 GMRES stagnated (two consecutive iterates were the same).
%   [X,FLAG,RELRES] = GMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0) also retu
%   the relative residual NORM(B-A*X)/NORM(B).  If FLAG is 0, RELRES

%   [X,FLAG,RELRES,ITER] = GMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0) also
%   returns both the outer and inner iteration numbers at which X was
%   computed: 0 <= ITER(1) <= MAXIT and 0 <= ITER(2) <= RESTART.
%   [X,FLAG,RELRES,ITER,RESVEC] = GMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X
%   returns a vector of the residual norms at each inner iteration, i
%   NORM(B-A*X0).

%      x = gmres(A,b,10,tol,maxit,M1,[],[]);
%   as inputs to GMRES
%      x1 = gmres(@afun,b,10,tol,maxit,@mfun,[],[],21);
%   See also BICG, BICGSTAB, CGS, LSQR, MINRES, PCG, QMR, SYMMLQ, LUI
```

```
function [x,flag,relres,iter,resvec] = ...
  dgmres(A,b,restart,tol,maxit,M1,M2,x0,Z,Y,varargin)
%DGMRES   Deflated Generalized Minimum Residual Method.
%   X = DGMRES(A,B) attempts to solve the system of linear
%   equations A*X = B for X. The N-by-N coefficient matrix A must
%   be square and the right hand side column vector B must have
%   length N. A may be a function returning A*X. This uses the
%   unrestarted method with MIN(N,10) total iterations.
%   DGMRES(A,B,RESTART) restarts the method every RESTART
%   iterations. If RESTART is N or [] then DGMRES uses the
%   unrestarted method as above.
%   DGMRES(A,B,RESTART,TOL) specifies the tolerance of the method.
%   If TOL is [] then DGMRES uses the default, 1e-6.
%   DGMRES(A,B,RESTART,TOL,MAXIT) specifies the maximum number of
%   outer iterations. Note: the total number of iterations is
%   RESTART*MAXIT. If MAXIT is [] then DGMRES uses the default,
%   MIN(N/RESTART,10). If RESTART is N or [] then the total number
%   of iterations is MAXIT.
%   DGMRES(A,B,RESTART,TOL,MAXIT,M) and
%   DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2) use preconditioner M or
%   M=M1*M2 and effectively solve the system inv(M)*A*X = inv(M)*B
%   for X.  If M is [] then a preconditioner is not applied. M
%   may be a function returning M\X.
%   DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0) specifies the initial
%   guess. If X0 is [] then DGMRES uses the default, an all zero
%   vector.
%   DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0,Z,Y) specifies the
%   deflation vectors. If Y is  [] then DGMRES uses Y = Z. If Z is
%   [] then DGMRES uses the default, no deflation at all.
%
%   DGMRES(AFUN,B,RESTART,TOL,MAXIT,M1FUN,M2FUN,X0,Z,Y,P1,P2,...)
%   passes parameters to functions: AFUN(X,P1,P2,...),
%   M1FUN(X,P1,P2,...), M2FUN(X,P1,P2,...).
%   [X,FLAG] = DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0,Z,Y) also return
%   a convergence FLAG:
%    0 DGMRES converged to the desired tolerance TOL within MAXIT
%      iterations.
%    1 DGMRES iterated MAXIT times but did not converge.
%    3 DGMRES stagnated (two consecutive iterates were the same).
%   [X,FLAG,RELRES] = DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0,Z,Y) also
%   returns the relative residual NORM(B-A*X)/NORM(B).  If FLAG is
%   0, RELRES <= TOL.
%   [X,FLAG,RELRES,ITER] = DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0,Z,Y)
%   also returns both the outer and inner iteration numbers at
%   which X was computed: 0 <= ITER(1) <= MAXIT and 0 <= ITER(2)<= RE
%   [X,FLAG,RELRES,ITER,RESVEC] =
%   DGMRES(A,B,RESTART,TOL,MAXIT,M1,M2,X0,Z,Y) also returns a vector
%   of the residual norms at each inner iterationm including the
%   zeroth norm.
%      x = dgmres(A,b,10,tol,maxit,M1,[],[]);
%   as inputs to DGMRES
%      x1 = dgmres(@afun,b,10,tol,maxit,@mfun,[],[],21);
%   See also BICG, BICGSTAB, CGS, LSQR, MINRES, PCG, QMR, SYMMLQ,
```

```
% Check for all zero right hand side vector => all zero solution
n2b = norm(b);                          % Norm of rhs vector, b
if (n2b == 0)                           % if    rhs vector is all zeros
   x = zeros(n,1);                      % then  solution is all zeros
   flag = 0;                            % a valid solution has been obtai
   relres = 0;                          % the relative residual is actual
   iter = [0 0];                        % no iterations need be performed
   resvec = 0;                          % resvec(1) = norm(b-A*x) = norm(
   if (nargout < 2)
      itermsg('gmres',tol,maxit,0,flag,iter,NaN);
   end
   return
end

if ((nargin > 8) & isequal(atype,'matrix') & ...
```

```
>  %   LUINC, @.
>  %%DGMRES: Added.
>  %   Adapted by Bart Dopheide (dopheide@fmf.nl, 2004) to use
>  %   deflation technique.
<
<
<
<
<
<
<
<
<
<
<
<
<
<
<
| %%DGMRES: Added. Check input parameter Z and Y.
>  if ((nargin >= 10) & ~isempty(Y))
>    if (size(Z) ~= size(Y))
>      error('Z and Y must have same dimension.');
>    end
>    if size(Z,1) ~= n
>      es = sprintf(['Deflation matrix must be a matrix with' ...
>                    ' %d rows to match the problem size.'],n);
>      error(es);
>    elseif size(Z,2) > n
>      es = sprintf(['Deflation matrix has more columns than A;' ...
>        ' Z cannot have %d independent deflation vectors.'], n + 1);
>      error(es);
>    end
>  else % Y not given or Y empty.
>    Y = [];
>    if (~isempty(Z))
>      Y = Z;        % Use symmetric deflation.
>    else % Neither Z nor Y given.
>      % No deflation wanted, so gmres() will do just fine.
>      if ~exist('M1') M1 = []; end
>      if ~exist('M2') M2 = []; end
>      [x,flag,relres,iter,resvec] = gmres(A, b, restart, tol, maxit, M1
>      return;
>    end
>  end
>
>  %%DGMRES: Changed. To allow Y and Z as extra arguments.
>  if ((nargin > 10) & isequal(atype,'matrix') & ...
>  %%DGMRES: Added. Introducing new variables.
>  AZ = A * Z;
>  A_deflated = Y' * AZ;
>
>  %%DGMRES: Moved.
>  % Check for all zero right hand side vector => all zero solution
>  n2b = norm(b - AZ * (A_deflated \ (Y' * b))); % Norm of rhs vector, P
>  if (n2b == 0)                                 % if    rhs vector is all zeros
>     x = zeros(n,1);                            % then  solution is all zeros
>     flag = 0;                                  % a valid solution has been obtai
>     relres = 0;                                % the relative residual is actual
>     iter = [0 0];                              % no iterations need be performed
>     resvec = 0;                                % resvec(1) = norm(b-A*x) = norm(
>     if (nargout < 2)
>        itermsg('gmres',tol,maxit,0,flag,iter,NaN);
>     end
>     %%DGMRES: Added. Convert the deflation solution to the real
>     %%       solution.
```

```
    x = Z * (A_deflated \ (Y' * (b - A * x))) + x;

return
end

%%DGMRES: Added. We want: r = P * b - P * A * x0.
%%        Rewrite: r = P * (b - A * x0)
%%                   = (I - A * Z * (Y'T * A * Z)^-1 * Y'T) * r0_node
%%                   = r0_nodeflation -
%%                     AZ * (A_deflated \ (Y'T * r0_nodeflation))
r = r - AZ * (A_deflated \ (Y' * r));

%%DGMRES: Cosmetic change. 'gmres' -> 'dgmres'.
itermsg('dgmres',tol,maxit,[0 0],flag,iter,relres);
%%DGMRES: Added. We want u2 = P * A * V(:,j).
u2 = u2 - AZ * (A_deflated \ (Y' * u2));

    %%DGMRES: Changed. Compute correct norm.
    gmres_r = b - A * x;
    normr = norm(gmres_r - AZ * (A_deflated \ (Y' * gmres_r)));
%%DGMRES: Changed. Compute correct norm.
gmres_r = b - iterapp(afun,atype,afcnstr,x,varargin{:});
normr = norm(gmres_r - AZ * (A_deflated \ (Y' * gmres_r)));
    %%DGMRES: Added. We want: r = P*b - P*A*x.
        r = r - AZ * (A_deflated \ (Y' * r));

%%DGMRES: Added. Compute correct residual.
r = r - AZ * (A_deflated \ (Y' * r));
%%DGMRES: Added. Convert the deflation solution to the real
%%        solution.
x = Z * (A_deflated \ (Y' * (b - A * x))) + x;

%%DGMRES: Cosmetic changes. 'gmres' -> 'dgmres' 2x.
itermsg('dgmres',tol,maxit,[i j],flag,
    itermsg(sprintf(['dgmres(%d)'],restart),tol,maxit,[i j],flag,
itermsg(sprintf(['gmres(%d)'],restart),tol,maxit,[i j],flag,
itermsg('gmres',tol,maxit,[0 0],flag,iter,relres);

normr = norm(b - A * x);

    normr = norm(b - iterapp(afun,atype,afcnstr,x,varargin{:}));

itermsg('gmres',tol,maxit,[0 0],flag,iter,relres);
```

Since we have several choices for preconditioning and deflation, we built a small test suite:

```
function [x,flag,res,iter,resvec] = ...
  stokes2D_test_suite(m,n,q,p,preconditioning_type, ...
                      deflation_type,restart)
%STOKES2D_TEST_SUITE    Function to aid testing of a 2D heated
%                       room problem based on the 2D Poisson
%                       equation.
%   X = STOKES2D_TEST_SUITE(M,N,Q,P) will try to solve a 2D
%   driven cavity problem of size MQxNP with subdomains of size
%   MxN. To solve with only one subdomain, this function can be
%   abbreviated to STOKES2D_TESTSUITE(M,N).
%
%   STOKES2D_TEST_SUITE(M,N,Q,P,PRECONDITIONING_TYPE) specifies
%   the preconditioning type to use. Options:
%     PRECONDITIONING_TYPE(1):
%       0: no preconditioner (default)
%       1: simple preconditioner without Gustafsson modification
%     PRECONDITIONING_TYPE(2) (only meaningful for
%     PRECONDITIONING_TYPE(1) ~= 0):
%       0: make no factorisation (default)
%       1: make a    complete Cholesky factorisation
%       2: make an incomplete Cholesky factorisation (droptol = 0.1)
%
%   STOKES2D_TEST_SUITE(M,N,Q,P,PRECONDITIONING_TYPE,DEFLATION_TYPE)
%   specifies the deflation type to use. Options:
%     DEFLATION_TYPE(1):
%       0: no deflation (default)
%       1: subdomain deflation
%       2: eigenvector deflation
%     DEFLATION_TYPE(2) (Only meaningful for DEFLATION_TYPE(1) ==
%     2):
%       0: use smallest eigenvectors (default)
%       1: use largest eigenvectors
%   Note: If a preconditioner is requested, the eigenvectors will
%   be based on the generalized eigenvalue problem, otherwise, the
%   eigenvectors will be based on the coefficient matrix only.
%
%   STOKES2D_TEST_SUITE(M,N,Q,P,PRECONDITIONING_TYPE,...
%   DEFLATION_TYPE,RESTART) specifies the restart value to use in
%   (D)GMRES. Default value is 40.
%
%   See also STOKES2D_TEST_COMBINATION, STOKES2D,
%   RHS_DRIVEN_CAVITY, DGMRES, GMRES.

if (nargin < 2 | nargin == 3)
  error('Not enough input arguments.');
end

if (nargin == 2)
  q = 1;
  p = 1;
end

if (nargin < 5)
  preconditioning_type = [];
end
preconditioning_type = ...
  extract_values(preconditioning_type, [0 0], [0 1], [0 1 2]);
preconditioner_type = preconditioning_type(1);
factorisation_type = preconditioning_type(2);
clear preconditioning_type;

if (nargin < 6 | isempty(deflation_type))
  deflation_type = 0;
end
deflation_type = ...
  extract_values(deflation_type, [0 0], [0 1 2], [0 1 2]);
if (deflation_type(1) == 2)
  eigenvectors_type = deflation_type(2);
end
deflation_type = deflation_type(1);

if (nargin < 7)
  restart = 40;
end

% Set constants.
droptol = 0.1;
tol = 1e-6;
maxit = 30000;
x0 = [];
%eigenvalues_method = 0; % Use eig().
eigenvalues_method = 1; % Use eigs().


% Create test problem.
A = stokes2D(m * q, n * p);
b = rhs_driven_cavity(m * q, n * p);

% Create preconditioner.
switch (preconditioner_type)
case 0 % No preconditioning.
  A_prec = [];
case 1 % Simple preconditioner.
  if (factorisation_type == 0)
    % Make preconditioner non-singular.
    A_prec = stokes2D_simple_preconditioner(m, n, q, p, 1);
  else
    % Leave it singular. luinc() will solve that problem.
    A_prec = stokes2D_simple_preconditioner(m, n, q, p, 1);
  end
end

% Create factorisation of preconditioner.
switch (factorisation_type)
case 0 % No factorisation.
  L = A_prec; U = [];
case 1 % Complete factorisation.
  [L, U] = lu(A_prec);
case 2 % Incomplete factorisation.
  [L, U] = luinc(A_prec, droptol);
  % Fix singularity when dealing with small subdomains.
  index = find(diag(U) == 0);
  U(index, index) = 1;
end

% Create deflation vectors.
switch (deflation_type)
case 0 % No deflation.
  Z = [];
case 1 % Subdomain deflation
  Z = stokes2D_subdomain_deflation_vectors(m, n, q, p);
case 2 % Eigenvector deflation
  switch (eigenvectors_type)
  case 0 % Eigenvectors based on smallest eigenvalues.
    if (preconditioner_type == 0)
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, [], 3 * q * p, 'SM', ...
                            eigenvalues_method);
    else
```

```
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, A_prec, 3 * q * p, 'SM', ...
                            eigenvalues_method);
    end
  case 1 % Eigenvectors based on largest eigenvalues.
    if (preconditioner_type == 0)
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, [], 3 * q * p, 'LM', ...
                            eigenvalues_method);
    else
      [eigenVec, eigenVal] = ...
        compute_eigenvalues(A, A_prec, 3 * q * p, 'LM', ...
                            eigenvalues_method);
    end
  end
  Z = eigenVec(:, q * p);
end

% Everything is setup now. Let's compute!
[x, flag, res, iter, resvec] = ...
  dgmres(A, b, restart, tol, maxit/restart, L, U, x0, Z);

if (flag ~= 0)
  fprintf('No converge reached within %d iterations.\n', maxit);
  x = x0;
end
```

To generate the tables in this report, we used a wrapper function that calls the test suite many times:

```
function [iterationsMatrix] = ...
  stokes2D_test_combination(problem_size, preconditioner_type, ...
                            deflation_type, restart)
%STOKES2D_TEST_COMBINATION    Try all combinations of subdomains.
%
%   ITERATIONSMATRIX = STOKES_TEST_COMBINATION(PROBLEM_SIZE,
%   PRECONDITIONER_TYPE,DEFLATION_TYPE,RESTART) tests combinations
%   of a square PROBLEM_SIZExPROBLEM_SIZE STOKES2D problem. This
%   is wrapper code for STOKES2D_TEST_SUITE. See that function for
%   details.
%
%   See also STOKES2D_TEST_SUITE, STOKES2D.

% Make a fancy header.
if (nargout == 0)
  fprintf('\\_p|\nq_\\|');
  for p = 2 .^ [0 : log2(problem_size)] fprintf('_%4d', p); end
  fprintf('\n---+');
  for p = 2 .^ [0 : log2(problem_size)] fprintf('-----', p); end
  fprintf('\n');
end

for q = 2 .^ [0 : log2(problem_size)]
  m = problem_size / q;
  if (nargout == 0) fprintf('%3d|', q); end
  for p = 2 .^ [0 : log2(problem_size)]
    n = problem_size / p;
    [x,flag,res,iter,resvec] = stokes2D_test_suite(m,n,q,p, ...
      preconditioner_type, deflation_type, restart);
    iter = inner_gmres_iterations(iter, restart);
    iter_mat(1 + log2(q), 1 + log2(p)) = iter;
    if (nargout == 0) fprintf('_%4d', iter); end
  end
  if (nargout == 0) fprintf('\n'); end
end
if (nargout > 0) iterationsMatrix = iter; end
```

This is the input needed to generate most tables in chapter 3:

```
% GMRES without preconditioning nor deflation.
for i = 2 .^ [ 0 : log2(128) ]
  for restart = [5 10 20 40 80 160 320]
    [x,f,r,iter,rv] = stokes2D_test_suite(i,i,1,1,[],[],restart);
    fprintf('%6d_', inner_gmres_iterations(iter, restart))
  end
  fprintf('\n');
end

problem_size = 32;
no_p = 0; simp = 1; gust = 2;
no_f = 0; comp = 1; incp = 2;
no_d = 0; subd = 1; eigd = 2;
smal = 0; larg = 1;
restart = 40;


% GMRES without preconditioning but with deflation.
stokes2D_test_combination(problem_size, no_p, subd, restart);

% GMRES with complete factorisation.
stokes2D_test_combination(problem_size, [simp, comp], no_d, restart);
stokes2D_test_combination(problem_size, [simp, comp], subd, restart);

% GMRES with incomplete factorisation.
stokes2D_test_combination(problem_size, [simp, incp], no_d, restart);
stokes2D_test_combination(problem_size, [simp, incp], subd, restart);

for i = 2 .^ [ 0 : log2(16384) ]
  [x,f,r,iter,rv] = stokes2D_test_suite(i,i,1,1.[],[],restart);
  fprintf('%d_', iter.)
end
fprintf('\n');

% Deflation based upon eigenvectors.
warning off MATLAB:nearlySingularMatrix
warning off MATLAB:eigs:SigmaNearExactEig
stokes2D_test_combination(problem_size,[simp,comp],[eigd,0],restart);
stokes2D_test_combination(problem_size,[simp,comp],[eigd,1],restart);
```

A.3 Other functions used

We give several small functions that don't belong specifically to either test problem, but are not standard MATLAB ® files.

```
function [M] = append_matrix(varargin)
%APPEND_MATRIX    Append matrices.
%   M = APPEND_MATRIX(A, B) will append matrix A to B resulting in
%     [A 0]
%     [0 B].
%   APPEND_MATRIX can be used with any number of input arguments
%   (including zero and one).

M = sparse([]);
for i = 1 : nargin
  [x(i), y(i)] = size(varargin{i});
  M(end + 1 : end + x(i), end + 1 : end + y(i)) = varargin{i};
end
```

```
function [sorted_eigen_vec,sorted_eigen_val] = ...
  compute_eigenvalues(A,A_prec,n,type,method)
%COMPUTE_EIGENVALUES    Compute eigenvalues.
%
%   [SORTED_EIGEN_VEC,SORTED_EIGEN_VAL] =
%   COMPUTE_EIGENVALUES(A,A_PREC,N,TYPE,METHOD) computes the
%   (generalized) eigensystem of (A,A_PREC). A_PREC may be empty,
%   in that case the eigenvalues of A are computed. N specifies
%   the number of eigenvectors/eigenvalues wanted. N is ignored
%   when METHOD == 0. METHOD == 0 corresponds to computing the
%   eigenvalues with an exact method while METHOD == 1 corresponds
%   to an inexact method. TYPE indicates the style to use when
%   METHOD == 1. TYPE == 'SM' for smallest eigenvalues and TYPE ==
%   'LM' for largest eigenvalues.
%
%   The output is sorted from largest absolute eigenvalues first
%   to smallest absolute eigenvalues last.
%
%   See also EIG, EIGS.

%FIXME: This function does not to error checking etc. In fact, it
%needs a rebuild.

clear opts
opts = struct('disp', 0, 'maxit', 1000);

if (method == 0)
  if (isempty(A_prec))
    [eigenVec, eigenVal] = eig(full(A));
  else
    [eigenVec, eigenVal] = eig(full(A), full(A_prec));
  end
else
  % Matlab 6.?
  warn = warning('query', ...
                 'MATLAB:eigs:TooManyRequestedEigsForRealSym');
  warning('off', 'MATLAB:eigs:TooManyRequestedEigsForRealSym');
  [eigenVec, eigenVal, flag] = eigs(A, A_prec, n, type, opts);
  if (flag ~= 0)
    es = sprintf(['Eigenvalues computation did not converge ' ...
                  'within %d iterations.'], opts.maxit);
    error(es);
  end
  warning(warn.state, ...
          ['MATLAB:eigs:TooManyRequestedEigsForRealSym']);
end

eigenVal = diag(eigenVal);
[sorted_eigen_val, index] = sort(-abs(eigenVal));
sorted_eigen_val = eigenVal(index);
sorted_eigen_vec = eigenVec(:, index);
```

```
function [outValues]=extract_values(inValues,defaultValues,varargin)
%

n = length(defaultValues);
outValues = zeros(n, 1);

if (isempty(inValues))
  inValues = defaultValues;
end
if (min(size(inValues)) ~= 1 | max(size(inValues)) > n)
  es = sprintf('inValues must be a vector of at most %d long.', n);
  error(es);
end
inValues = inValues(:);
m = length(inValues);

outValues(1:m) = inValues;
if (m < n)
  outValues(m + 1 : n) = defaultValues(m + 1 : n);
end

for i = 1 : nargin - 2
  if (~isempty(varargin{i}) & prod(outValues(i) - varargin{i}) ~= 0)
    es = sprintf(['element no. %d of inValues with value %f is ' ...
                  'not allowed. Allowed values are:'], i, ...
                 outValues(i));
    es = [es sprintf(' %f', varargin{i})];
    error(es);
  end
end
```

```
function [iter] = inner_gmres_iterations(Iters,restart)
%INNER_GMRES_ITERATIONS    Convert (D)GMRES iterations to inner
%                          iterations.
%
%   ITER = INNER_GMRES_ITERATIONS(ITERS,RESTART) converts the
%   outer/inner style iterations from (D)GMRES to total number of
%   inner iterations used.

if (iters(:) == [0; 0])
  iter = 0;
else
  iter = (iters(1) - 1) * restart + iters(2);
end
```

B The GNU General Public License

Version 2, June 1991

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the

same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM

(INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

References

[1] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. *SIAM Journal on Scientific Computing*, 23:442–462, 2001.

[2] A.E.P. Veldman. Computational fluid dynamics, 2001. R*u*G lecture notes.

[3] E.F.F. Botta. De methode der eindige elementen, 1999. R*u*G lecture notes (in Dutch).

[4] R. Nabben and C. Vuik. A comparison of deflation and coarse grid correction applied to porous media flow. Report 03-10, Delft University of Technology, Department of Applied Mathematical Analysis, Delft, 2003.

[5] C. Vuik, A. Segal, L. El Yaakoubi, and E. Dufour. A comparison of various deflation vectors applied to elliptic problems with discontinuous coefficients. *Applied Numerical Mathematics*, 41:219–233, 2002.

[6] F. Vermolen and C. Vuik. The influence of deflation vectors at interfaces on the deflated Conjugate Gradient method. Report 01-13, Delft University of Technology, Department of Applied Mathematical Analysis, Delft, 2001.

[7] C. Vuik, A. Segal, L. El Yaakoubi, and E. Dufour. A comparison of various deflation vectors applied to elliptic problems with discontinuous coefficients. Report 01-03, Delft University of Technology, Department of Applied Mathematical Analysis, Delft, 2001.

[8] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. MAS-R 0009, CWI, Amsterdam, 2000.