VLUGR2:
A Vectorized Local Uniform Grid Refinement Code for PDEs in 2D

J.G. Blom, J.G. Verwer
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

This paper describes an ANSI FORTRAN 77 code, VLUGR2, vectorized for the Cray YMP, that is based on an adaptive-grid finite-difference method to solve time-dependent two-dimensional systems of partial differential equations.

1991 Mathematics Subject Classification: Primary: 65-04. Secondary: 65M20, 65M50, 65F10.

1991 CR Categories: G1.8.

Keywords & Phrases: software, partial differential equations, method of lines, adaptive grid methods, nonsymmetric sparse linear systems, iterative solvers, vectorization.

Note: This work was supported by Cray Research, Inc., under grant CRG 92.05, via the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF).

1. Introduction

In [16, 14, 15, 18, 17, 19] an adaptive-grid finite-difference method is studied to solve time-dependent two-dimensional systems of partial differential equations (PDEs). Among others, a code, MOORKOP[13], has been developed which uses an implicit time-stepping method. In this paper we describe an ANSI FORTRAN 77 code, VLUGR2, that is based on MOORKOP but with the intention to use it on a vector processor. To that aim the datastructure of MOORKOP has been adapted and much attention has been paid to the solution of the large systems of nonlinear equations that arise in each time step.

The datastructure used in MOORKOP to store the grids and the solutions already is Fortran friendly and based on the way sparse matrices are usually stored in Fortran (cf. [8]). We changed it only to facilitate the vector processing of the most time consuming parts of the code, and to make the use of memory as compact and as small as possible.

Originally modified Newton in combination with a direct sparse matrix solver was used to solve the large systems of nonlinear equations (cf. [16, 14, 18]). In [3], it was shown that Krylov-type iterative solvers combined with standard Incomplete LU preconditioning were much faster. There we examined the ILU preconditioner and the iterative solvers from the Sparse Linear Algebra Package (SLAP, a public domain code written by Greenbaum and Seager (with contributions of several other authors) that is available from Netlib[7]). The most effective combination for the application in [3] turned out to be BiCGStab[20] + ILU preconditioning. As a result, this combination has been implemented in the code MOORKOP[13] to solve the linear systems.

The greater part of the iterative solvers is highly vectorizable (triads, dot products, etc.). In the SLAP code, which is intended for general sparse matrices, the required matrix operations, however, are much less amenable to vectorize. Since no matrix structure is assumed, the ILU factorization is more a searching process than a computational process. The ILU backsolve and the matrix-vector multiply do vectorize, but the resulting vector lengths are relatively small. In [2] we developed a matrix-vector multiply and the ILU preconditioning routines written especially for our type of problem: systems of PDEs discretized on a 9-point stencil and on a nonrectangular grid, by which we mean a grid

Report NM-R93??
ISSN 0169-0388
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

that is bounded by arbitrary right-angled polygons (cf. Fig. 3). A significant speedup is reached in comparison with the general purpose SLAP codes, in spite of the fact that only the matrix-vector multiply reaches its optimal vector performance for this type of problem. For the ILU preconditioning the vector lengths are in general too small to reach a satisfactory vector speed.

Therefore we also incorporated in our code a second linear system solver, viz., GMRESR[21], which is a variant of the iterative solver of Eirola and Nevanlinna[9]. This method was able to solve linear systems arising from a certain class of groundwater-flow problems with a simple diagonal scaling as preconditioner and vectorized extremely well, although the total CPU time was, for this type of problems, often larger than when BiCGStab + ILU preconditioning was used. The use of a basic polynomial preconditioner in GMRESR reduces the number of iterations needed to solve the linear systems, but in most cases not the total amount of time.

In Section 2 we define the problem class of `VLUGR2`. Section 3 is devoted to an outline of the local-uniform-grid-refinement method as implemented in `VLUGR2` and the choices made in the code with respect to the refinement strategy and the integration strategy (including the solution of the arising (non)linear systems). In Section 4 we discuss the implementation itself, viz., the for a user relevant part of the datastructure. There we also describe how to use `VLUGR2`, enlightened by an elaborated example problem. In Section 5 we compare the performance of `VLUGR2` and `MOORKOP`[13] on the same groundwater-flow problems as were used in [17]. The performance evaluations were done on a Cray YMP in scalar and vector mode.

2. PDE definition

`VLUGR2` has been designed to solve initial boundary value problems for systems of partial differential equations that fit in the following master equation [13]

$$\mathcal{F}(t, x, y, \mathbf{u}, \mathbf{u}_t, \mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_{xx}, \mathbf{u}_{xy}, \mathbf{u}_{yy}) = 0, \quad (x, y) \in \Omega, \quad t > t_0, \tag{2.1}$$

where the solution $\mathbf{u}$ may be a vector and the domain Ω an arbitrary domain that can be described by right-angled polygons (see, e.g., Fig. 3). The boundary conditions belonging to system (2.1) are formulated as

$$\mathcal{B}(t, x, y, \mathbf{u}, \mathbf{u}_t, \mathbf{u}_x, \mathbf{u}_y) = 0, \quad (x, y) \in \partial\Omega, \quad t > t_0, \tag{2.2}$$

and the initial conditions satisfy

$$\mathbf{u}(t_0, x, y) = \mathbf{u}_0(x, y), \quad (x, y) \in \Omega \cup \partial\Omega. \tag{2.3}$$

3. Outline of the LUGR algorithm

The concept of local uniform grid refinement is very simple. The domain is covered by a, uniform, coarse base grid and nested finer uniform subgrids are recursively created in regions with high spatial activity. So all grids consist of one or more disjunct sets of interconnected grid cells, all having the same size. When a grid of a specific refinement level has been created the corresponding initial boundary value problem is solved on the current time interval.

For the time integration we use the second-order two-step implicit BDF method with variable step sizes. The resulting system of nonlinear equations is solved with modified Newton and (preconditioned) iterative linear solvers. For the space discretization standard second-order finite differences are used, central on the internal domain and one-sided at the boundaries. At 'internal' corners, i.e., with an angle of 270^0, we use also one-sided differencing to prevent difficulties with the ILU decomposition of the Jacobian in the Newton process. Where grid refinement is required we divide a grid cell in 4 equal parts, so we do not apply semi-refinement, i.e., refinement in only one direction parallel to an axis.

In short the implementation is given by

0. Start with the coarse base grid, the initial solution and an initial time step

1. Solve the IBVP on the current grid with the current time step
2. If the required resolution in space is not yet reached:
   (a) Determine at forward time level a new, embedded, grid of next finer grid level
   (b) Get solution values at previous time level(s) on the new grid
   (c) Interpolate internal boundary values from old grid at forward time
   (d) Get initial values for the Newton process at forward time
   (e) Goto 1.
3. Inject fine grid solution values in coinciding coarser grid nodes
4. Estimate error in time-integration. If time error is acceptable
   - Advance time level
5. Determine new step size, goto 1 with coarse base grid as current grid.

Where interpolation is needed to get solution values, linear interpolation is used.

3.1. Refinement strategy

The virtue of a LUGR method lies in the fact that one reaches the accuracy of a fine mesh width with considerably less computational effort and memory requirements, since the fine subgrids cover only part of the domain. A necessity for the good functioning of a LUGR method is that the refinement strategy takes care that fine subgrids are timely and at the correct places created. In [14, 15, 18, 19] the refinement strategy is based on a comprehensive error analysis of both the space discretization and the interpolation. Here we use a much more simple strategy because class (2.1)-(2.3) is too general for the error analysis to be valid. Another point is that although this strategy is mathematically more pleasing than the one used here it has the disadvantage that error estimates are numerically correct on a coarse grid, only if the solution is sufficiently smooth, which is, unfortunately, not the case in practical situations where one applies a LUGR method. We therefore use here the same refinement strategy as in [16, 17, 13], i.e., based on a curvature monitor.

For each grid point (i, j) the space monitor is determined by

$$\mathtt{SPCMON}(i,j) := \max_{ic=1,\mathtt{NPDE}} \mathtt{SPCTOL}(ic).(|\Delta x^2.u_{xx}^{ic}(i,j)| + |\Delta y^2.u_{yy}^{ic}(i,j)|), \tag{3.1}$$

where Δx, respectively, Δy is the grid width in the x-, respectively, the y-direction and

$$\mathtt{SPCTOL}(ic) := \frac{\mathtt{SPCWGT}(ic)}{\mathtt{UMAX}(ic).\mathtt{TOLS}}. \tag{3.2}$$

The variables on the right-hand side of (3.2) are user specified quantities, $0 \leq \mathtt{SPCWGT} \leq 1$ a weighting factor for the relative importance of a PDE component on the space monitor, UMAX the, approximate, maximum absolute value for each component, and TOLS the space tolerance. The second-order derivatives in (3.1) are approximated by second-order finite differences at the internal domain and first order at the boundaries.

A next level of refinement is created if

$$\max_{(i,j)} \mathtt{SPCMON}(i,j) > \mathtt{TOLWGT}. \tag{3.3}$$

TOLWGT acts as a bar against fluctuations of the number of grid levels in subsequent time steps: if the next level of refinement existed at the previous time level $\mathtt{TOLWGT} = 0.9$, otherwise $\mathtt{TOLWGT} = 1.0$.

If a next level of refinement is required, then all grid points with

$$\texttt{SPCMON}(i,j) > 1/4 \tag{3.4}$$

are flagged together with their 8 direct neighbors. A cell with at least one flagged corner is quartered. The next grid level will contain all these refined cells. For a more elaborate explanation of the refinement strategy we refer to [16].

Finally, VLUGR2 offers the user the possibility to enforce grid refinement in a priori selected regions by adapting the SPCMON values (see the description of the CHSPCM subroutine in Section 4.2).

3.2. Integration in time

As time integrator we use the second-order two-step implicit BDF method with variable step sizes

$$\mathbf{U}_t = a_0\mathbf{U}^{n+1} + a_1\mathbf{U}^n + a_2\mathbf{U}^{n-1}, \tag{3.5a}$$

with

$$a_0 = \frac{1+2\alpha}{1+\alpha}\frac{1}{\Delta t},\ a_1 = -\frac{(1+\alpha)^2}{1+\alpha}\frac{1}{\Delta t},\ a_2 = \frac{\alpha^2}{1+\alpha}\frac{1}{\Delta t},\ \text{ and } \alpha = \frac{\Delta t}{\Delta t_{old}}. \tag{3.5b}$$

In the first time step we apply as usual Backward Euler ($\alpha = 0$ in (3.5)). The time integration is controlled by the solution monitor value in time which is computed at each existing grid level

$$\texttt{TIMMON}(level) := \max_{ic=1,\texttt{NPDE}} \max_{\{(i,j)|(x_i,y_j)\in\Omega_{level}\}} \texttt{TIMTOL}(ic).|\Delta t.u_t^{ic}(i,j)|, \tag{3.6}$$

where Δt is the current time step size, u_t is approximated by first-order finite differences and

$$\texttt{TIMTOL}(ic) := \frac{\texttt{TIMWGT}(ic)}{\texttt{UMAX}(ic).\texttt{TOLT}}. \tag{3.7}$$

Note that the time monitor is first order although the used time integration method is second order. The reason is that an estimator based on, for one time level, interpolated solution values, and for another computed, leads readily to unnecessarily small time steps (cf. [16]). This is also the reason to exclude the boundary points in (3.6). The variables on the right-hand side of (3.7) are the, user specified, analogues of the variables in (3.2).

An integration step is rejected and redone at all grid levels if

$$\max_{level} \texttt{TIMMON}(level) > 1.0. \tag{3.8}$$

A new step size is computed such that the prediction of the monitor at the next time point is 0.5, i.e.,

$$\Delta t_{new} := \frac{0.5}{\max_{level}\texttt{TIMMON}(level)}.\Delta t. \tag{3.9}$$

If the step was accepted the increase in step size is restricted to a factor 2 and if the time step was rejected the decrease is restricted to a factor 4. Finally, the step size is restricted to a user specified minimum and maximum value and adjusted such that the rest of the integration interval is an integer number times Δt.

3.2.1. Solving the nonlinear system

Since we use an implicit time integration method each time step a large system of nonlinear algebraic equations has to be solved. This is done using the modified Newton method in combination with an iterative linear system solver. The latter is described in Section 3.2.2. Here we discuss some of the strategies implemented in the Newton procedure. Globally spoken we have followed the approach as used in DASSL[5, pages 123–124]. To solve the nonlinear system

$$\mathbf{F}(t,x,y,\mathbf{U},\mathbf{U}_t,\mathbf{U}_x,\mathbf{U}_y,\mathbf{U}_{xx},\mathbf{U}_{xy},\mathbf{U}_{yy}) = 0,\quad t > t_0, \tag{3.10}$$

which is the fully discretized form of (2.1) and (2.2), we compute, numerically, a Jacobian

$$G = \frac{\partial \mathbf{F}}{\partial \mathbf{U}} \tag{3.11}$$

at the beginning of each time step. With this Jacobian linear systems

$$G.\mathbf{c}^{(k)} = -\mathbf{F}(\mathbf{U}^{(k)}) \tag{3.12}$$

are solved and the solution is updated

$$\mathbf{U}^{(k+1)} = \mathbf{U}^{(k)} + \mathbf{c}^{(k)}. \tag{3.13}$$

The Newton process is continued until the iteration error $\|\mathbf{U} - \mathbf{U}^{(k)}\|_w$ is sufficiently small, where the user-set tolerances `TOLS`, `TOLT` are buried in the norm

$$\|v\|_w = \max_{ipt=1,N;ic=1,\mathtt{NPDE}} \frac{|v_{ipt,ic}|}{w_{ipt,ic}}, \tag{3.14}$$

with

$$w_{ipt,ic} = \mathtt{ATOL}(ic) + |\mathbf{U}^{(k)}_{ipt,ic}|.\mathtt{RTOL}(ic). \tag{3.15}$$

Assuming convergence of the Newton process, the inequality

$$\|\mathbf{U} - \mathbf{U}^{(k)}\|_w \leq \frac{\rho}{1-\rho}\|\mathbf{U}^{(k)} - \mathbf{U}^{(k-1)}\|_w, \tag{3.16}$$

holds, where ρ is the convergence rate which is in actual computation approximated by

$$\rho \approx \rho^{(k)} = \left(\frac{\|\mathbf{U}^{(k)} - \mathbf{U}^{(k-1)}\|_w}{\|\mathbf{U}^{(1)} - \mathbf{U}^{(0)}\|_w}\right)^{\frac{1}{k-1}}. \tag{3.17}$$

This leads to the stopping criterion for the Newton iteration

$$\frac{\rho^{(k)}}{1-\rho^{(k)}}\|\mathbf{U}^{(k)} - \mathbf{U}^{(k-1)}\|_w < \mathtt{TOLNEW}, \tag{3.18}$$

where we have set $\mathtt{TOLNEW} = 1.0$. For the variables `ATOL` and `RTOL` in the weighting vector (3.15) we have made the choice

$$\mathtt{ATOL}(ic) = 0.01.\mathtt{TOL}.\mathtt{UMAX}(ic) \text{ and } \mathtt{RTOL}(ic) = \mathtt{TOL}, \tag{3.19}$$

with

$$\mathtt{TOL} = 0.1\min(\mathtt{TOLT}^2, \mathtt{TOLS}), \tag{3.20}$$

and `UMAX`, `TOLT`, and `TOLS` are the user specified values described above. Note that we use the square of the user-defined time tolerance, `TOLT`, because the BDF formula (3.5) is second order and the time monitor is first order.

If during the Newton iteration $\rho^{(k)} > 1$ or if the maximum number of iterations ($\mathtt{MAXNIT} = 10$) is exceeded, a new Jacobian is computed, once, and the iteration is restarted. If the Newton process does not converge with the new Jacobian the time step is redone with $\Delta t = \Delta t/4$.

Numerical approximation of the Jacobian

The Jacobian

$$G = \frac{\partial \mathbf{F}(t, x, y, \mathbf{U}, \mathbf{U}_t, \mathbf{U}_x, \mathbf{U}_y, \mathbf{U}_{xx}, \mathbf{U}_{xy}, \mathbf{U}_{yy})}{\partial \mathbf{U}} \tag{3.21}$$

is approximated by numerical differencing. To save residual evaluations we have made use of the identity

$$\begin{aligned}\frac{\partial \mathbf{F}}{\partial \mathbf{U}} &= \frac{\partial \mathbf{F}(.,\mathbf{U},\mathbf{U}_t,.)}{\partial \mathbf{U}} + \\ &\frac{\partial \mathbf{F}(.,\mathbf{U}_x,.)}{\partial \mathbf{U}_x}.\frac{\partial \mathbf{U}_x}{\partial \mathbf{U}} + \frac{\partial \mathbf{F}(.,\mathbf{U}_y,.)}{\partial \mathbf{U}_y}.\frac{\partial \mathbf{U}_y}{\partial \mathbf{U}} + \\ &\frac{\partial \mathbf{F}(.,\mathbf{U}_{xx},.)}{\partial \mathbf{U}_{xx}}.\frac{\partial \mathbf{U}_{xx}}{\partial \mathbf{U}} + \frac{\partial \mathbf{F}(.,\mathbf{U}_{xy},.)}{\partial \mathbf{U}_{xy}}.\frac{\partial \mathbf{U}_{xy}}{\partial \mathbf{U}} + \frac{\partial \mathbf{F}(.,\mathbf{U}_{yy},.)}{\partial \mathbf{U}_{yy}}.\frac{\partial \mathbf{U}_{yy}}{\partial \mathbf{U}}.\end{aligned} \tag{3.22}$$

This way of approximating the Jacobian makes it also easier to change the space discretizations. The implementation of (3.22) is very simple and vectorizes extremely well. If we approximate the partial derivatives of **F** in the right-hand side of (3.22) by numerical differencing, the perturbation is only local to a grid point and therefore the Jacobian can be obtained by 6 residual evaluations. Different space or time discretization only leads to different multiplying factors in (3.22). The way to compute the partial derivative of **F** with respect to **U** is copied from DASSL[5, page 124]

$$\frac{\partial \mathbf{F}(.,\mathbf{U},\mathbf{U}_t,.)}{\partial \mathbf{U}} \approx \frac{\mathbf{F}(,.\mathbf{U}+\boldsymbol{\Delta},\mathbf{U}_t+a_0\boldsymbol{\Delta},.)-\mathbf{F}(.,\mathbf{U},\mathbf{U}_t,.)}{\boldsymbol{\Delta}}, \tag{3.23}$$

where, since we use second-order central discretization,

$$\Delta_{ipt,ic} = \sqrt{uround}.\mathrm{sign}(\Delta t U_{t_{ipt,ic}}).\max(|U_{ipt,ic}|, |\Delta t U_{t_{ipt,ic}}|, \mathtt{ATOL}(ic)). \tag{3.24}$$

For the other partial derivatives we use

$$\frac{\partial \mathbf{F}(.,\mathbf{U}_p,.)}{\partial \mathbf{U}_p} \approx \frac{\mathbf{F}(.,\mathbf{U}_p+\boldsymbol{\Delta}_p,.)-\mathbf{F}(.,\mathbf{U}_p,.)}{\boldsymbol{\Delta}_p} \quad \text{for } p = x, y, xx, xy, yy. \tag{3.25}$$

The main difficulty in the numerical computation of the partial derivatives is the choice of the perturbation vector $\boldsymbol{\Delta}_p$, and especially to decide when the value to be perturbed should be considered zero. We chose the perturbation

$$\Delta_{p_{ipt,ic}} = \sqrt{uround}.\max(|U_{p_{ipt,ic}}|, \mathtt{ATOL}(ic).fac_p), \tag{3.26}$$

where

$$fac_x = \frac{1}{2\Delta x},\ fac_y = \frac{1}{2\Delta y},\ fac_{xx} = \frac{1}{\Delta x^2},\ fac_{yy} = \frac{1}{\Delta y^2},\ \text{ and } fac_{xy} = \frac{1}{4\Delta x \Delta y}. \tag{3.27}$$

So $\mathtt{ATOL}/2\Delta x$ is considered to be the noise value for U_x if `ATOL` is the noise value for U. Finally, the 'magic tricks'

$$\Delta_{p_{ipt,ic}} = \mathrm{sign}(U_{p_{ipt,ic}}).\Delta_{p_{ipt,ic}} \tag{3.28}$$

and

$$\boldsymbol{\Delta} = (\mathbf{U}+\boldsymbol{\Delta}) - \mathbf{U} \tag{3.29}$$

have been applied to ensure that the perturbed value has the same sign as the original one and that the perturbed value is a true machine number. Note that in (3.24) the former is not necessarily true. If the PDE is, e.g., undefined for negative values of U this can occasionally be a source of difficulties.

For the problems we solved with the code this way of computing a Jacobian was sufficiently accurate. However, if for a specific problem Newton failures would occur much more often than time step failures, it could be worthwhile to store the exact partial derivatives instead of the approximated ones (see Section 4.2).

One extra difficulty when solving nonlinear systems arising from the LUGR method is to provide an initial solution. If one employs the method of lines on a single grid fixed in time the solution of the previous time level is in general a good initial estimate. In any case, if $\Delta t \to 0$ it is *the* solution. This is, however, not the case with LUGR methods. The solution values injected from a finer grid are in general *not* a solution of the PDE system discretized on a coarser grid. So, even when $\Delta t \to 0$ it is still possible that the injected values on the previous time level are not a good initial estimate. For the nonlinear groundwater-flow problems we indeed experienced some serious problems with the convergence of the Newton process. So, for the sake of robustness, we also save the not-injected solution values, i.e., the solution values actually computed at the coarser grid and use these as initial estimate, with interpolated values where a grid point at the previous time did not exist.

3.2.2. Solving the linear system

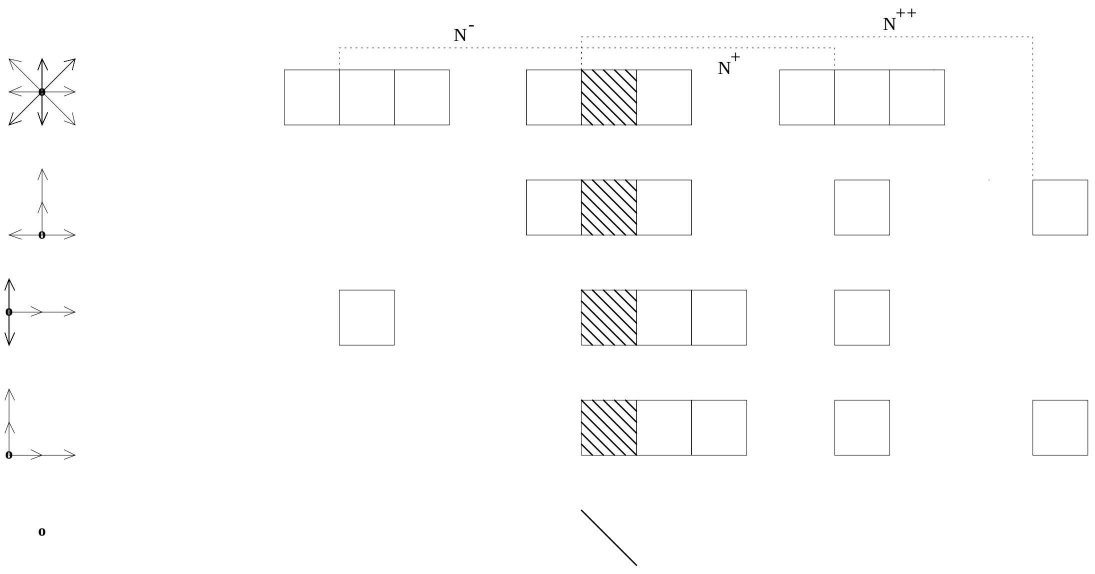


FIGURE 1. Structure of block-rows: internal point,
lower boundary point, left boundary point, lower left corner, Dirichlet boundary point

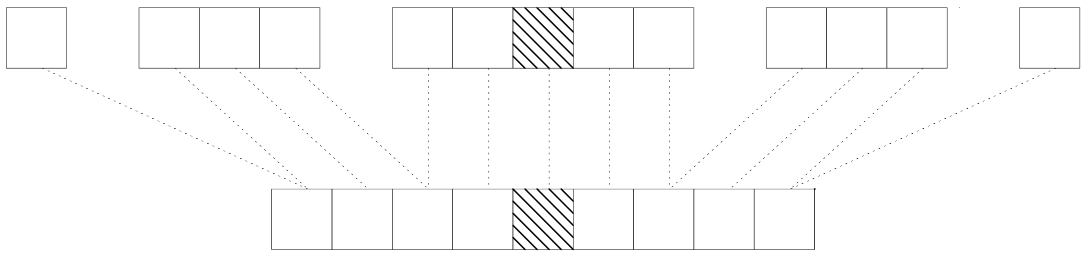


FIGURE 2. Block diagonal storage

VLUGR2 contains two iterative linear solvers to solve the linear systems (3.12). In both cases the matrix G, resulting from a second-order central discretization on the internal domain and second-order one-sided differences on the boundaries consists of block-rows as depicted in Fig. 1 and is stored in block 9-diagonal storage mode, i.e., in an array `G(NPTS,NPDE,NPDE,-4:4)` (cf. Fig. 2). The actual placing of the grid points corresponding with the 3 lower diagonals is given in the array `LLDG(NPTS,-4:-2)` and for the 3 upper diagonals in `LUDG(NPTS,2:4)`.

It is not necessary to solve the linear system (3.12) up to machine precision (cf. [3, 17]). The stopping criterion we use in our code is

$$\|K^{-1}r^{(l)}\|_w < \texttt{TOLLSS} \tag{3.30}$$

where K^{-1} is the preconditioner in use and `TOLLSS` = `TOLNEW`/(10.`MAXNIT`). Recall that, as decided in (3.18), `TOLNEW` = 1.

The first linear solver is BiCGStab[20], a variant of the well-known CGS method [12], in combination with ILU preconditioning. The latter is vectorized using a variant of the hyperplane method[1], which is described in [2]. Although this combination is in our experience robust and rather efficient with respect to the number of linear-solver iterations needed, the vector performance of the ILU preconditioning is not very high, due to the rather small vector lengths that result from the refined grids.

Therefore we also added to our code a second iterative linear system solver, viz., GMRESR[21], which is a variant of the iterative solver of Eirola and Nevanlinna[9]. This method adapts the initial preconditioner each step with rank-one updates. In GMRESR this initial preconditioner is based on the latest residual and is computed with a number of GMRES[11] iterations. For the GMRES solver we used a stripped version of the SLAP implementation. The linear system is, explicitly, diagonally preconditioned. This combination has a much higher Mflop rate, but it is less robust and its behavior is less 'smooth' than the combination BiCGStab + ILU. In our experiments a stagnation of the iteration process occurs in a number of occasions.

A very simple polynomial preconditioner has also been added, i.e.,

$$K^{-1}Ax = K^{-1}b \tag{3.31}$$

is solved, where

$$K^{-1} = I + (I - A) + \cdots + (I - A)^M. \tag{3.32}$$

Although this polynomial preconditioner in general diminishes the number of iterations, in our experiments it mostly not results in a decrease of computer time. Therefore we chose not to use it by default.

The GMRESR default parameters we use are the following

- `NRRMAX` = 1, maximum number of restarts outer loop
- `MAXLR` = 20, maximum loop length outer loop, dimension solution space
- `MAXL` = 10, maximum loop length inner loop, dimension preconditioner space
- `M` = 0, maximum power in the polynomial preconditioner.

4. The code

We have deliberately chosen to keep the use of `VLUGR2` as simple as possible, i.e., all method parameters are set in parameter statements in the code itself rather than letting the user specify them. The user has to specify only the problem parameters and routines. If one wishes to change one of the method parameters the corresponding parameter statements should be changed in the code.

In the simplest case, a well-scaled problem on a rectangular domain, one has to specify

- the number of PDEs,
- the initial and final time, and the initial step size,

- the lower-left and the upper-right corner of the domain, and the initial grid width in the x-, respectively, y-direction,
- the space and time tolerance `TOLS` and `TOLT`, and
- the subroutines `PDEIV`, `PDEF`, and `PDEBC`, to specify respectively, the initial solution (2.3), the PDE system at the internal domain (2.1) and the boundary conditions (2.2).

If the initial domain is not a true rectangle, a virtual rectangle is to be placed upon the irregular domain (cf. Fig. 3). In this case the user should also provide a routine that defines the initial grid. Furthermore, one can supply a routine to enforce grid refinement and a monitor routine for user purposes that will be called after each successful time step (cf. Section 4.2).

4.1. Datastructure

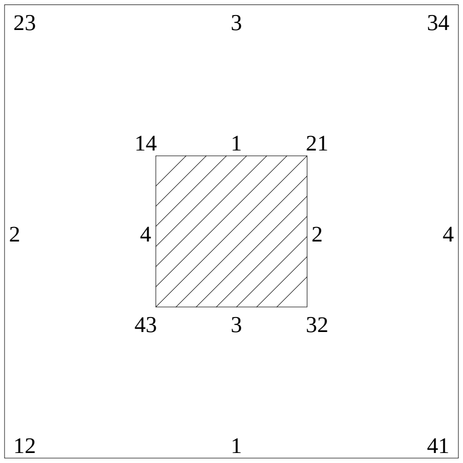
The datastructure used in `MOORKOP`[13] to store the grids and the solutions is changed only to facilitate the vector processing of the most time consuming parts of the code, and to make the use of memory as compact and as small as possible. To achieve the former, pointers to the boundary points in the grid structure are added so that, e.g., the computation of the PDE can be performed in two 'sweeps', one, using direct addressing, over the whole domain for the PDE system on the internal domain (2.1) and one, using indirect addressing, for the boundary conditions (2.2). For an efficient use of memory all information with respect to the solutions and the grids is stored consecutively in one real and one integer work array.

The solution at a specific grid level is stored row-wise, one component vector after the other. Solutions from 3 different time levels have to be saved. For the oldest time level only the injected solution values for the computation of $\mathbf{U}_t$ with (3.5). For the previous time level also the original, not-injected, solution is needed to serve as an initial solution estimate in the Newton process.

A grid at a specific grid level is stored in the following datastructure

- `LROW`: the actual number of rows in the grid, the pointers to the start of a row in the grid, and the number of grid points + 1
- `IROW`: the row number of a row in the virtual rectangle
- `ICOL`: the column number of a grid point in the virtual rectangle
- `LLBND`: the total number of physical boundaries and corners in the actual domain, the pointers to the start of a specific boundary or corner in `LBND`, and the number of boundary points + 1
- `ILBND`: the type of the boundaries

1:	lower boundary
2:	left boundary
3:	upper boundary
4:	right boundary
12:	lower left corner (90^0)
23:	left upper corner (90^0)
34:	upper right corner (90^0)
41:	right lower corner (90^0)
21:	left lower corner (270^0)
32:	upper left corner (270^0)



43: right upper corner (270^0)

14: lower right corner (270^0)

- `LBND`: the pointers to the boundary points in the actual grid
- `LBELOW`: pointer to node below in the actual grid or 0, if index node is lower boundary point
- `LABOVE`: pointer to node above in the actual grid or 0, if index node is upper boundary point

All grids from 3 different time levels have to be saved. For the base grid all information is saved, i.e., the arrays containing the grid information and the pointer arrays needed for the Jacobian and, if used, the pointer arrays for the hyperplane ordering. For the higher level grids only the first 3 arrays (`LROW,IROW,ICOL`) are saved.

The above mentioned arrays and workspace for the Jacobian, ILU, and linear system solver are stored consecutively in the real and integer work arrays. Pointers for each time and grid level indicate the start in a work array of a solution or grid at a specific time and at a specific grid level. For a more complete description of the contents of the work arrays we refer to the documentation of the enveloping routine `VLUGR2`.

4.2. How to use

The calling sequence of `VLUGR2` is

```
      CALL  VLUGR2 (NPDE, T, TOUT, DT, XL, YL, XR, YU, DX, DY,
     +   TOLS, TOLT, INFO, RINFO, RWK, LENRWK, IWK, LENIWK, LWK, LENLWK,
     +   MNTR)
```

where in the simplest case, using all default values the meaning of the parameters is

`NPDE`: the number of PDE components

`T`: the initial time

`TOUT`: the final time

`DT`: the initial time step size

`(XL,YL)`: coordinate of lower left corner of domain

`(XR,YU)`: coordinate of upper right corner of domain

`DX`: cell width in x-direction of base grid

`DY`: cell width in y-direction of base grid

`TOLS`: space tolerance as used in (3.2)

`TOLT`: time tolerance as used in (3.7)

`INFO`: `INFO(1)` $= 0$, which implies that default parameters are used

`RINFO`: dummy array

`RWK(LENRWK)`: real workspace, `LENRWK` $\approx$ (5.`MAXLEV`+ 18.`NPDE`).`NPTS`.`NPDE`, with `MAXLEV` the maximum number of levels allowed and `NPTS` the, expected, average number of points on a grid

`IWK(LENIWK)`: integer workspace, `LENIWK` $\approx$ (7.`MAXLEV`+ 20).`NPTS`

`LWK(LENLWK)`: logical workspace, `LENLWK` $\approx$ 2.`NPTS`

MNTR: MNTR $= 0$, first call of VLUGR2 for this problem

Furthermore one should supply the routines

PDEIV to specify the initial solution (2.3),

PDEF to specify the PDE system at the internal domain (2.1), and

PDEBC to specify the boundary conditions (2.2).

Note, that PDEBC will be called after PDEF, which implies that in PDEF all grid points can be handled as if they were internal grid points.

If one wants to continue the integration after returning from VLUGR2, one should set MNTR to 1. All parameters can be changed although T, DT, XL, YL, XR, YU, DX, and DY will be overwritten with their old values. The contents of the work arrays RWK and IWK should not be altered.

A more sophisticated use can be made with the aid of the INFO and RINFO arrays and by overloading some subroutines. If INFO(1) $\neq 0$ a number of parameters can be specified in INFO and RINFO. The value between brackets is the default value used when INFO(1) $= 0$.

INFO(2): MAXLEV (3), the maximum number of grid levels allowed

INFO(3): RCTDOM (0), if RCTDOM $= 0$ the initial domain is a rectangle, otherwise the user should specify the subroutine INIDOM to define the initial grid (see below)

INFO(4): LINSYS (0), if LINSYS $= 0$ the linear system solver will be the combination BiCGStab + ILU, otherwise it will be GMRESR (+ polynomial preconditioning)

INFO(5): LUNPDS (0), the logical unit number of the file where information on the integration history will be written. If LUNPDS $= 0$ no information will be written

INFO(6): LUNNLS (0), the logical unit number of the file where information on the Newton process will be written. If LUNNLS $= 0$ no information will be written

INFO(7): LUNLSS (0), the logical unit number of the file where information on the linear system solver will be written. If LUNLSS $= 0$ no information will be written

RINFO(1): DTMIN (0.0), the minimum time step size allowed

RINFO(2): DTMAX (TOUT-T), the maximum time step size allowed

RINFO(3): UMAX ((1.0)), the approximate maximum values of the PDE solution components. These values are used for scaling purposes

RINFO(3+NPDE): SPCWGT ((1.0)), the weighting factors used in the space monitor to indicate the relative importance of a PDE component on the space monitor

RINFO(3+2.NPDE): TIMWGT ((1.0)), the weighting factors used in the time monitor to indicate the relative importance of a PDE component on the time monitor.

The subroutines that are candidates for overloading by the user are

MONITR to monitor the solution or the grids; will be called after each successful time step

CHSPCM to enforce grid refinement at a specific point in space and time and on a specific level

INIDOM to specify the initial grid for a domain that is not rectangular

DERIVF to store the exact partial derivatives of the residual **F** with respect to (the derivatives of) **U**.

Finally, the package also contains a number of routines that facilitate the use of the datastructure

`PRDOM` to print the domain one has defined with `INIDOM`

`SETXY` to get the x- and y-coordinates corresponding with the grid points

`PRSOL` to print the solution and the corresponding coordinate values at all grid levels

`DUMP` to dump all necessary information for a restart on file

`RDDUMP` to read all necessary information for a restart from the dump file

For the details of these routines we refer to the documentation in `VLUGR2`.

4.3. Example problem

Our example problem is the two-dimensional Burgers' system

$$u_t = -uu_x - vu_y + \varepsilon(u_{xx} + u_{yy}) \tag{4.1a}$$

$$v_t = -uv_x - vv_y + \varepsilon(v_{xx} + v_{yy}) \tag{4.1b}$$

on the domain Ω as given in Fig. 3. On the boundaries $\partial\Omega$ we prescribe Dirichlet conditions. An exact solution is given by (cf. [4])

$$u = \frac{3}{4} - \frac{1}{4}\frac{1}{1+\exp((-4x+4y-t)/(32.\varepsilon))} \tag{4.2a}$$

$$v = \frac{3}{4} + \frac{1}{4}\frac{1}{1+\exp((-4x+4y-t)/(32.\varepsilon))}. \tag{4.2b}$$

The solution represents a wave front at $y = x + 0.25t$. The speed of propagation is $\sqrt{2}/8$ and is perpendicular to the wave front. We solve this problem at the time interval [0.0,3.0] and with $\varepsilon = 10^{-3}$.

For this example we will describe shortly how to write the user routines. The complete text of the user program is enclosed with the package.

In the first call of `VLUGR2` the initial grid is formulated in the `INIDOM` routine, a refinement is enforced up to level 3 in the lower right corner of the physical domain, and the monitor routine is used to print the error after each time step and at each grid level. After $t = 1.0$ we will stop program 1, write all information to file and restart the computation with a different program reading the data from file. In that run we overload `DERIVF` with our own version which stores the exact partial derivatives $\partial\mathbf{F}/\partial\mathbf{U}_p$.

We cover the domain by a virtual rectangle ((0.0,0.0),(1.0,1.0)) and make a virtual base grid of 11×11 grid points (cf. Fig. 3). For the first part of the time interval [0,1.0] the user program will contain

```
      MNTR = 0
      NPDE = 2
      T    = 0.0
      TOUT = 1.0
      DT   = 0.001
C Since domain is not a rectangle the grid parameters need not to be
C specified
      TOLS = 0.1
      TOLT = 0.1
      INFO(1) = 1
```

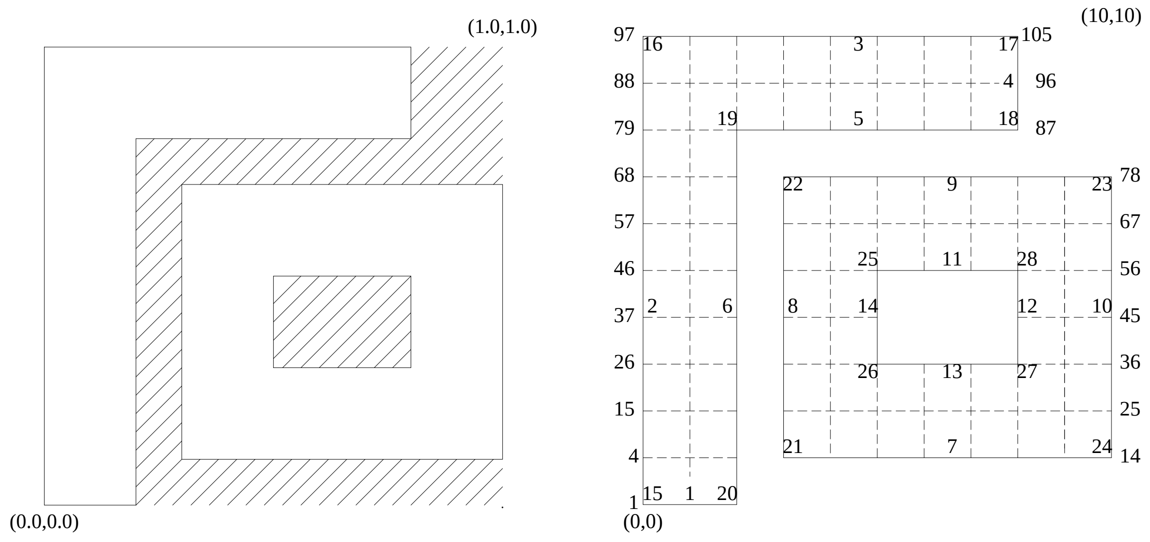


FIGURE 3. Example domain
Left the physical domain, right the base grid on the virtual rectangle.
Numbers at the left and the right are the node numbers of the grid points
Numbers inside the domain give the boundary or corner order as stored in the boundary structure

```
C MAXLEV
      INFO(2) = 5
C Domain not a rectangle
      INFO(3) = 1
C GMRESR
      INFO(4) = 1
C
      OPEN (UNIT=61,FILE='RunInfo')
C Write integration history to unit # 61
      INFO(5) = 61
C Write Newton info to unit # 61
      INFO(6) = 61
C Write GMRESR info to unit # 61
      INFO(7) = 61
C DTMIN = 1E-7
      RINFO(1) = 1.0E-7
C DTMAX = 1.0
      RINFO(2) = 1.0
C UMAX = 1.0
      RINFO(3) = 1.0
      RINFO(4) = 1.0
C SPCWGT = 1.0
      RINFO(5) = 1.0
      RINFO(6) = 1.0
C TIMWGT = 1.0
      RINFO(7) = 1.0
      RINFO(8) = 1.0
```

```
C
C Call main routine
      CALL  VLUGR2 (NPDE, T, TOUT, DT, XL, YL, XR, YU, DX, DY,
     +   TOLS, TOLT, INFO, RINFO, RWK, LENRWK, IWK, LENIWK, LWK, LENLWK,
     +   MNTR)
```

Since we want to restart the computation we write an, unformatted, file with all the necessary information

```
      OPEN(UNIT=LUNDMP,FILE='DUMP',FORM='UNFORMATTED')
      CALL DUMP (LUNDMP, RWK, IWK)
      CLOSE(LUNDMP)
```

Next the PDE defining routines

```
      SUBROUTINE PDEIV
         . . .
      DO 10 I = 1, NPTS
         U(I,1) = 0.75 - 0.25/(1+EXP((-4*X(I)+4*Y(I)-T)/(32*EPS)))
         U(I,2) = 0.75 + 0.25/(1+EXP((-4*X(I)+4*Y(I)-T)/(32*EPS)))
   10 CONTINUE

      SUBROUTINE PDEF
         . . .
      DO 10 I = 2, NPTS-1
         RES(I,1) = UT(I,1) -
     +      (-U(I,1)*UX(I,1) - U(I,2)*UY(I,1) + EPS*(UXX(I,1)+UYY(I,1)))
         RES(I,2) = UT(I,2) -
     +      (-U(I,1)*UX(I,2) - U(I,2)*UY(I,2) + EPS*(UXX(I,2)+UYY(I,2)))
   10 CONTINUE

      SUBROUTINE PDEBC
         . . .
      NBNDS = LLBND(0)
      DO 10 K = LLBND(1), LLBND(NBNDS+1)-1
         I = LBND(K)
         RES(I,1) = U(I,1) -
     +            (0.75 - 0.25/(1+EXP((-4*X(I)+4*Y(I)-T)/(32*EPS))))
         RES(I,2) = U(I,2) -
     +            (0.75 + 0.25/(1+EXP((-4*X(I)+4*Y(I)-T)/(32*EPS))))
   10 CONTINUE
```

To define the initial grid in INIDOM one should store

```
XL = 0.0
YL = 0.0
XR = 1.0
YU = 1.0
DX = 0.1
DY = 0.1

LROW(0:12)  = (11, 1, 4, 15, 26, 37, 46, 57, 68, 79, 88, 97, 106)
IROW(1:11)  = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
ICOL(1:105) = (0, 1, 2,
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               0, 1, 2, 3, 4, 5, 6, 7, 8,
               0, 1, 2, 3, 4, 5, 6, 7, 8,
               0, 1, 2, 3, 4, 5, 6, 7, 8)
```

For the boundaries, when numbered as in Fig. 3 the following should be stored in `LLBND`, `ILBND`, and `LBND`

```
LLBND(0:29) = ( 28,
                 1,  2, 11, 18, 19, 24, 31, 37, 42, 48, 53, 55, 56, 58,
                59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73)
ILBND(1:28) = (  1,  2,  3,  4,  1,  4,
                 1,  2,  3,  4,
                 1,  2,  3,  4,
                12, 23, 34, 41, 14, 41,
                12, 23, 34, 41,
                14, 43, 32, 21)
LBND(1: 72) = (  2,
                 4, 15, 26, 37, 46, 57, 68, 79, 88,
                98, 99,100,101,102,103,104,
                96,
                86, 85, 84, 83, 82,
                70, 59, 48, 39, 28, 17, 6,
                 8,  9, 10, 11, 12, 13,
                18, 29, 40, 49, 60,
                72, 73, 74, 75, 76, 77,
                67, 56, 45, 36, 25,
                52, 53,
                43,
                33, 32
                42,
                 1, 97, 105, 87, 81, 3, 7, 14, 78, 71, 51, 31, 34, 54)
```

To check if we defined the domain correct we print it out with

```
      INTEGER IDOM((NX+1)*(NY+1))
C
      LLBND(30) = LLBND(29)
      CALL PRDOM (LROW, IROW, ICOL, LLBND, ILBND, LBND,
     +   IDOM, NX, NY)
```

which prints internal points of the domain as `..`, external points as `XX` and for physical boundary points their `ILBND` value. The results are shown in Fig. 4.

To restart the computation we have to read the information from file and call `VLUGR2` with $\texttt{MNTR} = 1$. In this run we use the default parameters. Note that the old values for `T`, `DT`, `XL`, `YL`, `XR`, `YU`, `DX`, and `DY` will be taken.

```
23  3  3  3  3  3  3  3 34 XX XX
 2 .. .. .. .. .. .. ..  4 XX XX
 2 .. 14  1  1  1  1  1 41 XX XX
 2 ..  4 23  3  3  3  3  3  3 34
 2 ..  4  2 .. .. .. .. .. ..  4
 2 ..  4  2 .. 14  1  1 21 ..  4
 2 ..  4  2 ..  4 XX XX  2 ..  4
 2 ..  4  2 .. 43  3  3 32 ..  4
 2 ..  4  2 .. .. .. .. .. ..  4
 2 ..  4 12  1  1  1  1  1  1 41
12  1 41 XX XX XX XX XX XX XX XX
```

FIGURE 4. Print out by the subroutine PRDOM of the example domain

```
C Continuation call of VLUGR2
      TOUT = 3.0
      TOLS = 0.1
      TOLT = 0.1
      INFO(1) = 0
      MNTR = 1
C
      OPEN(UNIT=LUNDMP,FILE='DUMP',FORM='UNFORMATTED')
      CALL RDDUMP (LUNDMP, RWK, LENRWK, IWK, LENIWK)
      CLOSE(LUNDMP)
C
C call main routine
      CALL  VLUGR2 (NPDE, T, TOUT, DT, XL, YL, XR, YU, DX, DY,
     +   TOLS, TOLT, INFO, RINFO, RWK, LENRWK, IWK, LENIWK, LWK, LENLWK,
     +   MNTR)
```

For this run we want to use the exact partial derivatives $\partial\mathbf{F}/\partial\mathbf{U}_p$ and therefore we have overloaded DERIVF with our own subroutine of which contents we give a survey below

```
      SUBROUTINE DERIVF
         . . .
C A0: coefficient of U_n+1 in time derivative (cf. (3.5))
C
Ccc Loop over the components of the (derivatives of) U
      IC = 1
C
C dF(U,Ut)/dU_ic
         DO 20 IPT = 1, NPTS
            FU(IPT,1,IC) = A0 - (-UX(IPT,1))
            FU(IPT,2,IC) = - (-UX(IPT,2))
   20    CONTINUE
C
C dF(Ux)/dUx_ic
         DO 40 IPT = 1, NPTS
```

```
            FUX(IPT,1,IC) = - (-U(IPT,1))
            FUX(IPT,2,IC) = 0.0
   40    CONTINUE

         . . .

      IC = 2
C
C dF(U,Ut)/dU_ic
         DO 120 IPT = 1, NPTS
            FU(IPT,1,IC) = - (-UY(IPT,1))
            FU(IPT,2,IC) = A0 - (-UY(IPT,2))
  120    CONTINUE

         . . .
C
C dF(Uyy)/dUyy_ic
         DO 180 IPT = 1, NPTS
            FUYY(IPT,1,IC) = 0.0
            FUYY(IPT,2,IC) = - (EPS)
  180    CONTINUE

C
C Correct boundaries (incl. the internal); all Dirichlet so FUp = 0.0
      NBNDS = LLBND(0)
      DO 100 LB = LLBND(1), LLBND(NBNDS+2)-1
         IPT = LBND(LB)
         FU(IPT,1,1) = 1.0
         FU(IPT,1,2) = 0.0
         FU(IPT,2,1) = 0.0
         FU(IPT,2,1) = 0.0
         FU(IPT,2,2) = 1.0
         FUX(IPT,1,1) = 0.0

         . . .

         FUYY(IPT,2,2) = 0.0
  100 CONTINUE
```

In Fig. 5 the generated grids after the final times of the first and the second program are shown. The corresponding accuracy is, measured in the maximum norm, 0.01 at $t = 1.0$ and 0.08 at $t = 3.0$. Note that in the second run the number of grid levels is restricted to 3 which affects of course the accuracy.

5. Performance

In this section we will compare the performance of VLUGR2 and MOORKOP on the groundwater-flow problems that were also used as examples in [17].

5.1. Problem description: the 2D fluid-flow / salt-transport problem

In [17] we consider a model for a non-isothermal, single-phase, two-component saturated flow problem which consists of 3 PDEs basic to groundwater flow: the continuity equation, the transport equation

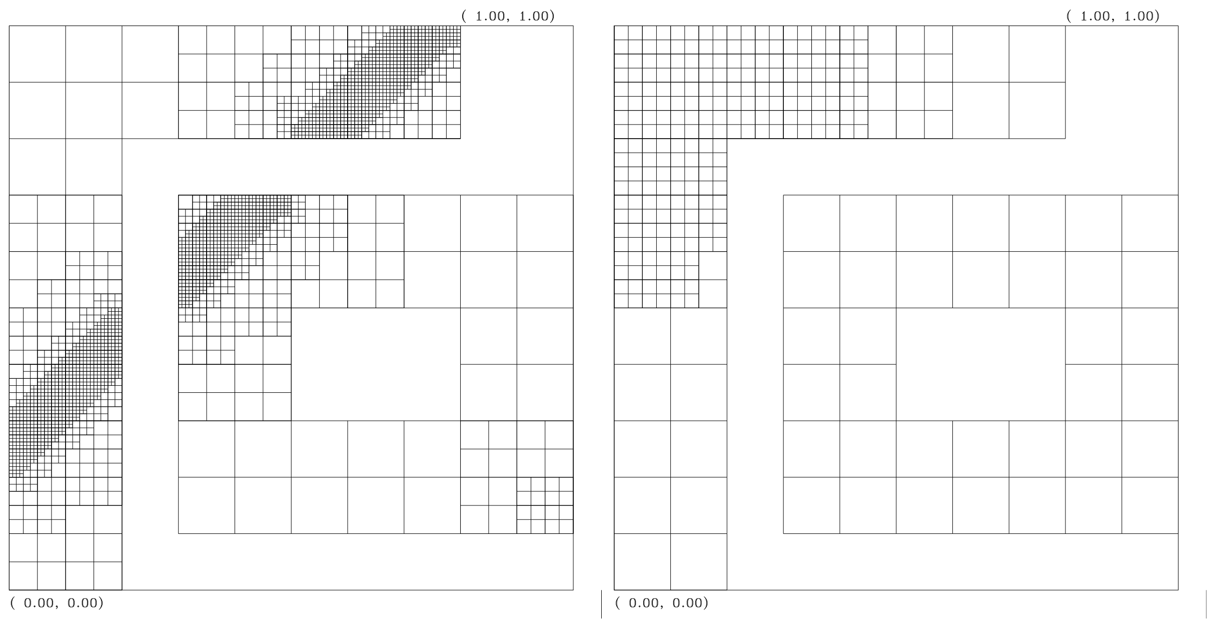


FIGURE 5. Grids for the example problem at $t = 1.0$ and at $t = 3.0$.

and the temperature equation. For the background of these equations we refer to [17]. We here present the model in non-conservative form. As independent variables we have the pressure p, the salt mass fraction ω, and the temperature T. The continuity equation for the fluid, the salt transport equation, and the temperature equation are given by

$$n\rho(\beta\frac{\partial p}{\partial t} + \gamma\frac{\partial \omega}{\partial t} + \alpha\frac{\partial T}{\partial t}) + \nabla \cdot (\rho\mathbf{q}) = 0, \tag{5.1a}$$

$$n\rho\frac{\partial \omega}{\partial t} + \rho\mathbf{q} \cdot \nabla\omega + \nabla \cdot (\rho\mathbf{J}^{\omega}) = 0, \tag{5.1b}$$

$$c^m\rho^m\frac{\partial T}{\partial t} + \rho c\mathbf{q} \cdot \nabla T + \nabla \cdot (\mathbf{J}^T) = 0, \tag{5.1c}$$

where n is the porosity parameter of the porous medium, β a compressibility coefficient, γ a salt coefficient, α a temperature coefficient, $c^m\rho^m$ a density expression satisfying $c^m\rho^m = nc\rho{+}(1{-}n)c^s\rho^s$, and c the specific heat capacity of the porous medium. Darcy's law gives the equation for the fluid velocity $\mathbf{q} = (q_1, q_2)^T$

$$\mathbf{q} = -\frac{k}{\mu}(\nabla p - \rho\mathbf{g}), \tag{5.2}$$

with $\mathbf{g}$ the acceleration-of-gravity vector and k the permeability coefficient of the porous medium. The density ρ and the viscosity μ obey the state equations

$$\rho = \rho_0 \exp[\alpha(T - T_0) + \beta(p - p_0) + \gamma\omega], \tag{5.3}$$

$$\mu = \mu_0(T).m(\omega),\ m(\omega) = 1 + 1.85\omega - 4.0\omega^2, \tag{5.4}$$

where ρ_0 is the reference density of fresh water, p_0 a reference pressure, T_0 a reference temperature, and $\mu_0(T)$ a possibly temperature-dependent reference viscosity. The equation for the salt-dispersion flux vector is given by Fick's law

$$\mathbf{J}^{\omega} = -n\mathbf{D}\nabla\omega, \tag{5.5}$$

with the dispersion tensor $\mathbf{D}$ for the solute salt defined as

$$n\mathbf{D} = (nD_{mol} + \alpha_T|\mathbf{q}|)I + \frac{\alpha_L - \alpha_T}{|\mathbf{q}|}\mathbf{q}\mathbf{q}^T, \ |\mathbf{q}| = \sqrt{\mathbf{q}^T\mathbf{q}}. \tag{5.6}$$

The coefficients D_{mol}, α_T and α_L correspond, respectively, with the molecular diffusion and the transversal and longitudinal dispersion. Finally, the heat-flux vector $\mathbf{J}^T$ is given by

$$\mathbf{J}^T = -\mathbf{H}\nabla T, \tag{5.7}$$

where $\mathbf{H}$, the heat conductivity tensor of the porous medium, is defined as

$$\mathbf{H} = (\kappa + \lambda_T|\mathbf{q}|)I + \frac{\lambda_L - \lambda_T}{|\mathbf{q}|}\mathbf{q}\mathbf{q}^T. \tag{5.8}$$

The parameter κ is the coefficient of heat conductivity and λ_T and λ_L are, respectively, the transversal and longitudinal heat conductivity coefficients.

Our examples are connected with Intraval test case 13[10]. This laboratory experiment deals with the displacement of fresh water by brine in a thin vertical column filled with a porous medium. Salt water of a high concentration is injected through gates at the bottom of the column giving rise to a fresh-salt water front moving in all directions into the column.

5.1.1. Data for Problem I

In [10] the experiment was carried out under isothermal conditions. We assume non-isothermal conditions and suppose that warm brine is injected. Because the column is very thin, the flow can be considered as being two-dimensional. In our numerical experiment it is supposed that two gates are used for salt water injection so that initially two disjunct salt / fresh water fronts exist which later interact and merge into one front. Due to dispersion the fronts smooth out with time in all directions. For t sufficiently large the fronts disappear completely which means that the whole medium is filled with the high-salt-concentration fluid.

In Problem I the model is considered on the time-space domain $[0, t_{end}] \times \Omega$ where the flow domain Ω representing the vertical column is the unit square $\Omega = \{(x,y)|0 < x, y < 1\}$. Since Ω represents a vertical cross section, the independent variable y stands for a vertical variable with unit vector pointing upward. Hence the acceleration of gravity vector takes the form $\mathbf{g} = (0, -g)^T$, where g is the gravity constant. The initial values at $t = 0$ at $\Omega \cup \partial\Omega$ are taken as

$$p(x,y,0) = p_0 + (1-y)\rho_0 g, \quad \omega(x,y,0) = 0, \quad \text{and} \quad T(x,y,0) = T_0, \tag{5.9}$$

which correspond, respectively, to hydrostatic pressure, fresh water and a non-heated medium. For $0 < t \leq t_{end}$ the following boundary conditions are imposed

$$\begin{array}{lll} x = 0, 1, & 0 \leq y \leq 1: & q_1 = 0, \ \omega_x = 0, \ T_x = 0, \\ y = 1, & 0 < x < 1: & p = p_0, \ \omega_y = 0, \ T_y = 0, \\ y = 0, & \frac{1}{11} \leq x \leq \frac{2}{11}, \ \frac{9}{11} \leq x \leq \frac{10}{11}: & q_2 = q_c, \ \omega = \omega_0, \ T = T_c, \\ y = 0, & 0 < x < \frac{1}{11}, & \\ & \frac{2}{11} < x < \frac{9}{11}, \ \frac{10}{11} < x < 1: & q_2 = 0, \ \omega_y = 0, \ T_y = 0. \end{array} \tag{5.10}$$

The third line is connected with the two gates where the warm brine is injected with a prescribed velocity and concentration. The other conditions are self-evident. All remaining problem data are contained in Table 1.

We have run this example using the following set of numerical parameters

$$\Delta t_0 = 0.1 \quad \text{and} \quad \Delta x = 0.05, \quad \Delta y = 0.05,$$

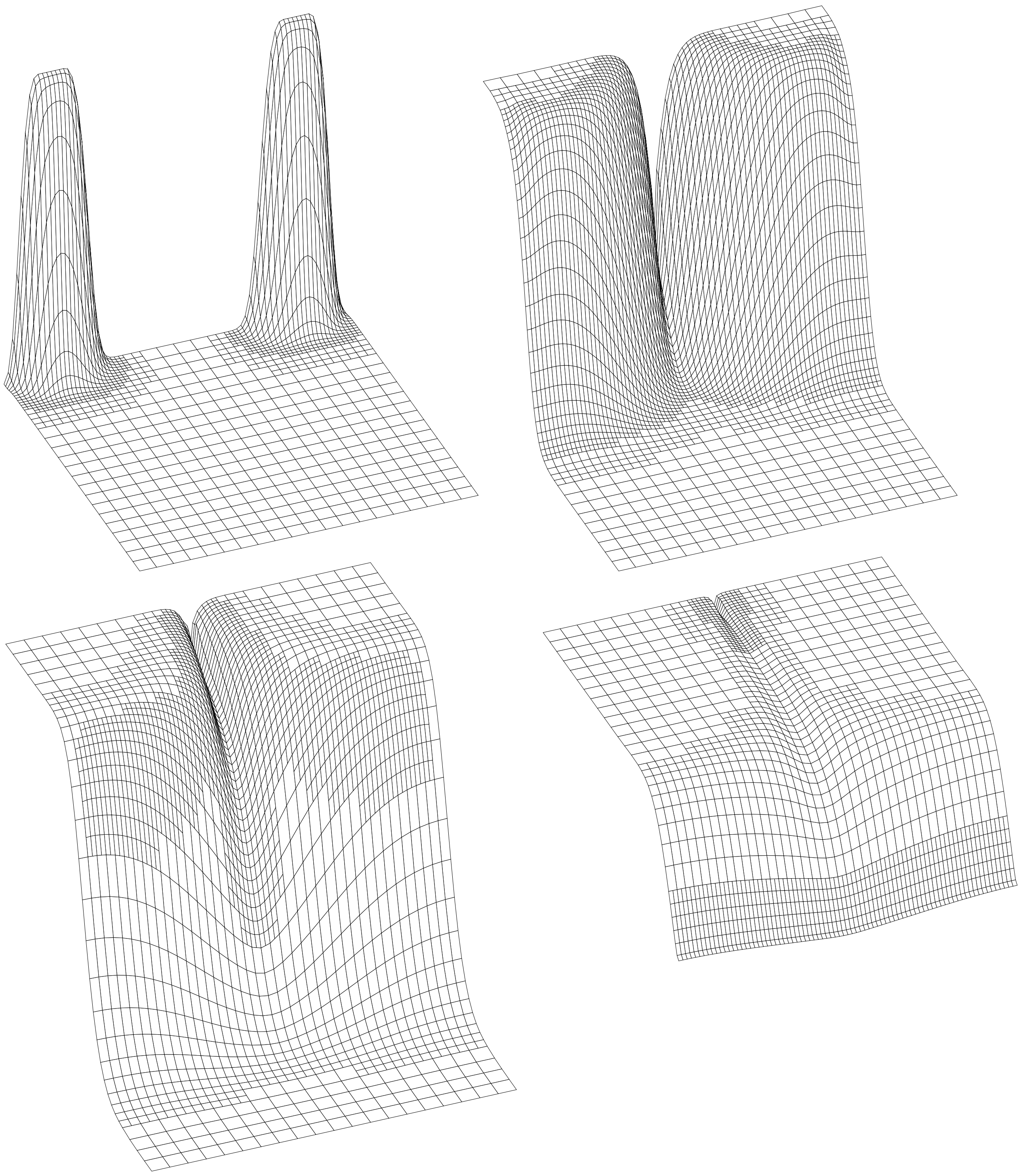


FIGURE 6: Distribution of salt concentration at $t = 500, 5000, 10000$ and 20000 with the corresponding grid configuration

n	0.4	λ_T	0.001	β	$4.45\,10^{-10}$
k	10^{-10}	λ_L	0.01	γ	ln(1.2)
g	9.81	c^s	840	μ_0	10^{-3}
D_{mol}	0.0	ρ^s	2500	ω_0	0.25
α_T	0.002	ρ_0	1000	q_c	10^{-4}
α_L	0.01	T_0	290	T_c	292
c	4182	p_0	10^5	t_{end}	$2\,10^4$
κ	4.0	α	$-3\,10^{-4}$		

TABLE 1. Data for Problem I.

$$\texttt{TOLS} = 0.1 \quad \text{and} \quad \texttt{TOLT} = 0.1,$$
$$\texttt{UMAX} = (1.1\text{E}+5, 0.25, 292).$$

For all other parameters we used the default choice. Since Δx and Δy are the cell widths of the coarse base grid and MAXLEV = 3 this results in a finest grid size allowed of 1/80. Although this is small enough to avoid wiggles, at most time intervals the refinement is restricted by MAXLEV and not by TOLS.

In Fig. 6 we show the distribution of salt concentration at various time levels with the corresponding grid configuration. The temperature distribution shows a comparable behavior, but less steep and slower in time. The grid configurations in [17] are slightly different, since those results were obtained with a different refinement strategy. The results for the current MOORKOP code are comparable to Fig. 6.

5.1.2. Data for Problem II

In the second example we consider the case of total impermeability for part of the flow domain, viz., the region $\{(x, y)|0 \leq x \leq 0.5,\ 0.4 \leq y \leq 0.6\}$. We also consider the flow now to be isothermal and incompressible ($\alpha = \beta = 0$).

The boundary conditions for the impermeability region are

$$\begin{array}{lll} 0 \leq x \leq 0.5, & y = 0.4, 0.6: & p_y = -\rho g, \quad \omega_y = 0, \\ x = 0.5 & 0.4 < y < 0.6: & p_x = 0, \quad \omega_x = 0. \end{array} \tag{5.11}$$

We have closed the right gate so that salt water is injected only through the left gate. Hence we deal with a single salt / fresh water front which first collides with the region of impermeability and then must flow around it. For this problem we run until steady state is reached ($\omega \equiv \omega_0$), i.e., $t_{end} = 10^5$. All other parameters, if required, are chosen as in Problem I.

The distribution of the salt concentration at various time levels with the corresponding grid configuration is shown in Fig. 7.

5.2. Performance

Our performance evaluation was done on a Cray YMP with the CF77 compiling system. Scalar results were obtained using `cf77 -Wf" -o novector"`, and vector results with `cf77 -Zv -Wf"-o aggress"`. We also used the performance measurement package `perfview` (compiler flags `-F`, and loader flags `-F -lperf`).

5.2.1. Integration history

The codes MOORKOP and VLUGR2 use different strategy parameters (the description of the strategy choices made in MOORKOP can be found in [13] and of VLUGR2 in Section 3). Therefore the integration history is not equivalent and to appreciate the performance comparison one should take these differences into account. For both problems the integration history is given in Table 2.

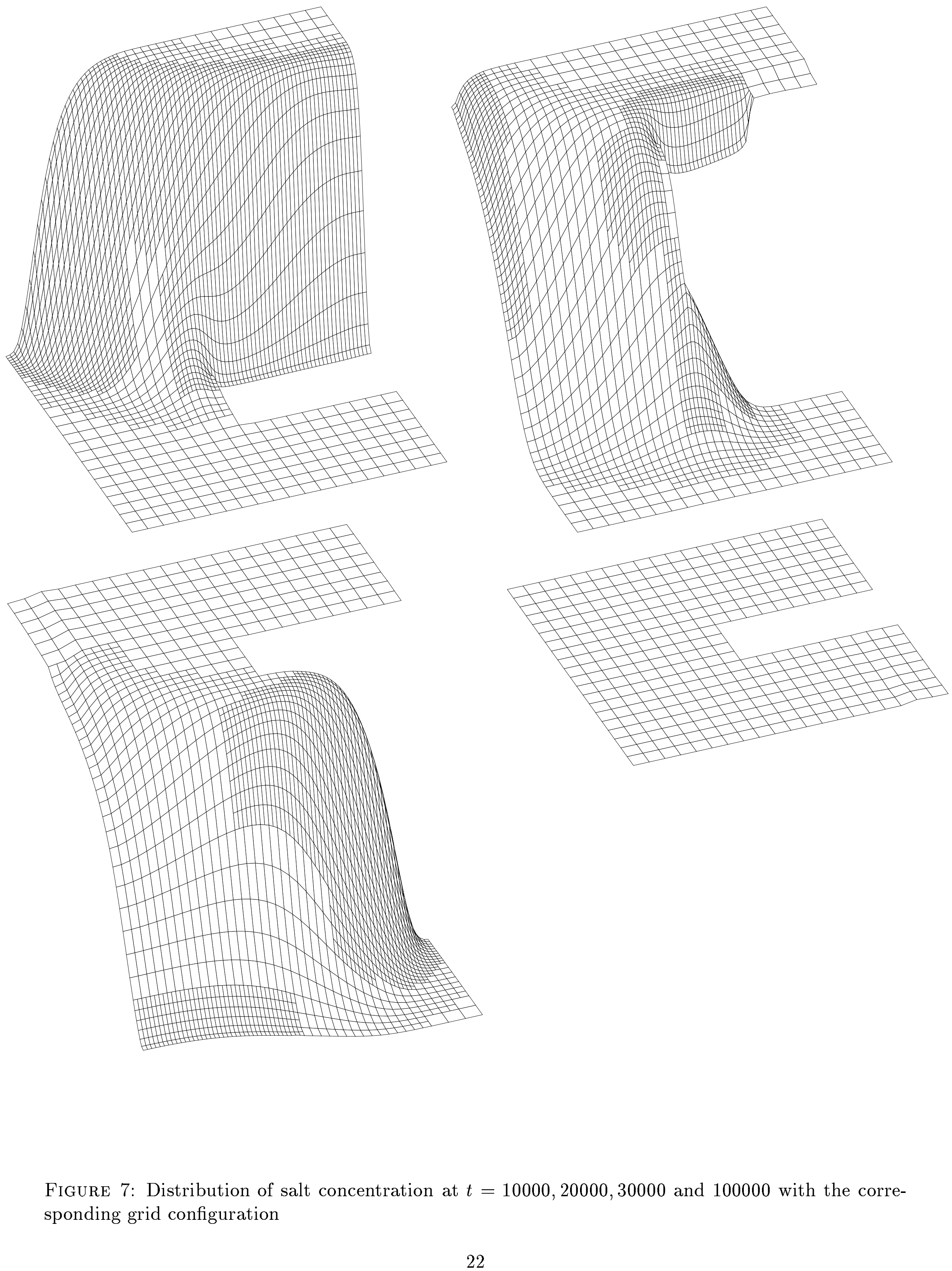


FIGURE 7: Distribution of salt concentration at $t = 10000, 20000, 30000$ and 100000 with the corresponding grid configuration

		Problem I			Problem II		
		`MOORKOP` BiCGStab	`VLUGR2` BiCGStab	`VLUGR2` GMRESR	`MOORKOP` BiCGStab	`VLUGR2` BiCGStab	`VLUGR2` GMRESR
# time steps		178	172	172	268	273	303
# rejected steps		0	0	0	0	0	17
# Newton it./level	1	579	344	344	889	546	623
	2	595	344	344	954	524	528
	3	616	344	344	1018	459	463
# Lin. sys. it. per							
Newton it., level	1, 1	2180	1501	16075	3467	2863	55335
	1, 2	2123	1212	13902	2988	2091	24680
	1, 3	2318	1299	16623	3550	2057	25993
	2, 1	1418	543	4519	2344	1090	7888
	2, 2	1384	541	6033	1895	1027	9709
	2, 3	1387	600	5587	2073	953	9409
	3, 1	482			988	0	0
	3, 2	501			795	0	0
	3, 3	529			871	7	15
	4, 1	55			265		
	4, 2	55			300		
	4, 3	172			386		
	5, 1	8			3		
	5, 2	3			1		
	5, 3	2			42		
	6, 1	6			4		
	6, 2	2			0		
	6, 3	0			5		
	7, 1	2			2		
	7, 2	0			0		
	7, 3	0			0		

TABLE 2. Integration history

The number of linear system iterations given under GMRESR is the sum of the outer-loop iterations (the Eirola-Nevanlinna process) and the inner-loop iterations (GMRES 'preconditioner'). The time step rejections are due to GMRESR stagnations. The number of these stagnations is dependent on the method parameters. If we run Problem II, e.g., with `MAXL` $= 15$ the number of rejected steps is only 2 and the total number of iterations is decreased by approximately 10%. However, since the decrease is mainly due to base grid iterations the total computer time is approximately equal. If the polynomial preconditioner is used with `M` $= 2$ the number of iterations is approximately decreased by a factor of 2, but the CPU time used is roughly the same. It is however possible that with a more sophisticated (polynomial) preconditioner the GMRESR version will use less iterations.

The number of Newton iterations used by `MOORKOP` is larger because the tolerance on convergence of the Newton process is more strict by a factor 100, although occasionally also the computation of the Jacobian can have its influence. We emphasize however that the results of both codes are comparable.

Note that in the second example `MOORKOP` also solves a PDE on the impermeability region (cf. [17]). The resulting inefficiency is not so much the extra number of grid points in the base grid, but the fact that even at steady state a refinement up to the maximum number of grid levels allowed will remain around the region of impermeability, because of the jump in the salt concentration values.

5.2.2. *Global performance*

		MOORKOP BiCGStab		VLUGR2 BiCGStab		VLUGR2 GMRESR	
		CP sec	Mflop	CP sec	Mflop	CP sec	Mflop
Problem I	scalar	1744	6	449	14		
	vector	1163	11	100	65	135	152
Problem II	scalar	1210	7	280	14		
	vector	822	12	62	67	126	140

TABLE 3. Global performance.

We first give a global idea of the performance of the 3 different codes, i.e., `MOORKOP`, `VLUGR2` + BiCGStab, and `VLUGR2` + GMRESR. In Table 3 the CPU time and the Mflop rate is shown for both example problems.

It is clear that although the GMRESR solver reaches an almost optimal vector speed, considering that we use indirect addressing, the CP time is (much) larger than when we use BiCGStab + ILU. For the example problem in Section 4.3 `VLUGR2` + GMRESR is faster than using `VLUGR2` + BiCGStab, but not much. So for the moment we do not advocate this method, the more so because it is less robust and polynomial preconditioning of different powers `M` gives quite various results. Perhaps GMRESR can be a better competitor with a better but equally vectorizable (polynomial) preconditioner.

With respect to both BiCGStab codes it should be mentioned that, even if we look at the ratio CPsec / lin.sys.it., `VLUGR2` is faster. For Problem I, e.g., this ratio is for `MOORKOP` 1744/12627 = 0.138, respectively, 1163/12627 = 0.092 in scalar, respectively, vector mode, whereas the figures for `VLUGR2` are 449/5696 = 0.080 for the scalar run, respectively, 100/5696 = 0.018 for the vector run.

5.2.3. *Vector results*

In this section we will compare the vector performance of the three combinations. The timings were done on 1 processor of a Cray Y-MP, which has a clock cycle time of 6ns. This gives a theoretical peak performance on 1 processor of 167 Mflops and 333 when chaining an add and a multiply. Since during one cycle time 1 store and 2 loads can be performed, indirect addressing of (one of) the vector operands of a triad will reduce the performance at least with a factor of 2, bank conflicts left out of consideration. To measure the Megaflop rate and the CPU time of a routine we used the Cray utility Perftrace[6], that gives the hardware performance by program unit.

As can be seen from the global performance in Table 3, `MOORKOP` does not vectorize very well, except for the linear solver BiCGStab which consists mainly of BLAS calls. `MOORKOP` could be sped up by almost 30% by storing the Jacobian directly in SLAP column format. However, one has to pay a price for the easy use of an 'off-the-shelf' code. As long as no use is made of the fact that the Jacobian arises from a 9-point discretization, the ILU decomposition will always take a significant part of the computation and is essentially sequential.

	# calls	Avg time	ACM %	Mflop
ILU dec	541	7.1E-1	33.1	1
conversion	541	5.9E-1	60.5	0
ILU backs	27044	7.7E-3	78.4	23
PDE	14233	8.6E-3	89.0	21
MATVEC	27044	2.7E-3	95.2	47

TABLE 4. Vector performance of top 5 routines of `MOORKOP` for Problem I.
ACM %: Accumulated percentage of CPU-time spent

	# calls	Avg time	ACM %	Mflop
ILU backs	12424	3.2E-3	40.4	39
ILU dec	516	5.1E-2	67.4	12
MATVEC	11392	9.6E-4	78.6	143
PDEF	10320	1.0E-3	89.5	213
BiCGStab	1032	1.8E-3	91.4	154

TABLE 5. Vector performance of top 5 routines of `VLUGR2` + BiCGStab for Problem I.

	# calls	Avg time	ACM %	Mflop
MATVEC	62739	1.0E-3	50.0	141
GMRES	7058	5.3E-3	78.8	183
PDEF	10320	1.0E-3	87.0	212
GMRESR	1032	4.1E-3	90.2	180
WMXNRM	72377	5.7E-5	93.4	97

TABLE 6. Vector performance of top 5 routines of `VLUGR2` + GMRESR for Problem I.

	# calls	Avg time	ACM %	Mflop
ILU dec	806	2.2E-1	22.1	1
conversion	806	2.2E-1	44.2	0
ILU backs	42819	5.0E-3	70.5	16
PDE	15757	6.9E-3	83.8	20
MATVEC	42819	1.6E-3	92.4	34

TABLE 7. Vector performance of top 5 routines of `MOORKOP` for Problem II.
ACM %: Accumulated percentage of CPU-time spent

	# calls	Avg time	ACM %	Mflop
ILU backs	21705	1.4E-3	50.0	32
MATVEC	20176	3.5E-4	61.9	139
PDEF	10697	6.0E-4	72.6	222
ILU dec	764	7.4E-3	82.1	20
BiCGStab	1529	1.5E-3	86.0	115

TABLE 8. Vector performance of top 5 routines of `VLUGR2` + BiCGStab for Problem II.

	# calls	Avg time	ACM %	Mflop
GMRES	14574	3.2E-3	39.4	153
MATVEC	133029	3.4E-4	77.8	139
PDEF	11394	5.9E-4	83.5	222
GMRESR	1614	3.9E-3	88.9	143
WMXNRM	151629	2.9E-5	92.7	92

TABLE 9. Vector performance of top 5 routines of `VLUGR2` + GMRESR for Problem II.

The tables on the vector performance of `VLUGR2` show that the PDE definition on the internal domain (`PDEF`) vectorizes nicely, as expected since no exceptions have to be made. The definition of the boundary conditions (`PDEBC`) run at a lower Mflop rate of about 65, but this routine takes only 0.5% of the total CPU time. All in all it appears that, with the exception of the preconditioning, `VLUGR2` + BiCGStab is an efficiently vectorized code. The routines needed for the irregular grid structure and refinement take together less than 5% of the CPU time (in vectorized mode). So the overhead for the grid refinement is negligible as long as we need to solve our problems with an implicit time integrator.

The very large number of calls in `VLUGR2` + GMRESR of the routine that computes the weighted maximum norm is due to the fact that we check in every iteration, both of the inner and the outer loop, if the weighted maximum norm of the residual is small enough to stop the iteration process. If we use GMRESR with the weight vector as scaling vector for the residual vector and apply the natural stopping criterion, i.e., the Euclidean norm of the (weighted) residual which is built up during the GMRES process, more stagnations of the iteration process occur, most likely because the condition of the matrix is worsened by the scaling with the weight vector.

5.2.4. Data storage requirements

The Cray YMP system has not much memory hierarchy, viz., registers, main memory and disks. This means that for normal use one has not to think much about memory requirements as long as the program + data fits into the main memory. On most other architectures the hierarchy is extended, e.g., with virtual memory, with the consequence that the main memory is much smaller. On those systems the real-time performance is often not determined by the floating-point operations but by memory operations. Therefore we consider it also of importance to keep the data storage small in amount and compact.

The amount of work storage required by `MOORKOP` is approximately

$$\texttt{INTEGER}: \quad \texttt{nptspl}.(4\frac{2}{5}.\texttt{maxlev} + 3.\texttt{npde}.(8 + 9.\texttt{npde}))$$

$$\texttt{REAL}: \quad \texttt{nptspl}.\texttt{npde}.(5.\texttt{maxlev} + 25.\texttt{npde} + 31).$$

`VLUGR2` when using BiCGStab as linear system solver needs

$$\texttt{INTEGER}: \quad \texttt{NPTS}.(7.\texttt{MAXLEV} + 20)$$

$$\texttt{REAL}: \quad \texttt{NPTS}.\texttt{NPDE}.(5.\texttt{MAXLEV} + 18.\texttt{NPDE} + 9)$$

and when using GMRESR

$$\texttt{INTEGER}: \quad \texttt{NPTS}.(7.\texttt{MAXLEV} + 17)$$

$$\texttt{REAL}: \quad \texttt{NPTS}.\texttt{NPDE}.(5.\texttt{MAXLEV} + 9.\texttt{NPDE} + 9 + 2.\texttt{MAXLR} + \texttt{MAXL} + 6).$$

So `VLUGR2` requires in most cases less data storage space than `MOORKOP`. Moreover, in `MOORKOP` the arrays are fixed on forehand, so `nptspl` should be the maximum number of points encountered at any level and at any time, and `maxlev` the maximum number of levels at any time, whereas the work arrays in `VLUGR2` are dynamically stored although at the cost of shifting data at the end of a time step.

References

[1] C. C. Ashcraft and R.G. Grimes. On vectorizing incomplete factorization and SSOR preconditioners. *SIAM J. Sci. Stat. Comput.*, 9(1):122–151, 1988.

[2] J.G. Blom and J.G. Verwer. Vectorizing matrix operations arising from PDE discretization on 9-point stencils. Report NM-R9221, CWI, Amsterdam, 1992. (submitted to J. Supercomputing).

[3] J.G. Blom, J.G. Verwer, and R.A. Trompert. A comparison between direct and iterative methods to solve the linear systems arising from a time-dependent 2D groundwater flow model. Report NM-R9205, CWI, Amsterdam, 1992. (to appear in Int. J. Comp. Fluid Dynamics, 1993, Vol. 1).

[4] J.H.M. ten Thije Boonkkamp. *The Numerical Computation of Time-Dependent, Incompressible Fluid Flow.* PhD thesis, University of Amsterdam, the Netherlands, 1988.

[5] K.E. Brenan, S.L. Campbell, and L.R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North-Holland, New-York, 1989.

[6] Cray Research, Inc. *UNICOS Performance Utilities Reference Manual*, SR-2040 6.0 edition.

[7] J.J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Commun. ACM*, 30:403–407, 1987. (netlib@research.att.com).

[8] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[9] T. Eirola and O. Nevanlinna. Accelerating with rank-one updates. *Lin. Alg. and its Appl.*, 121:511–520, 1989.

[10] S.M. Hassanizadeh, A. Leijnse, W.J. de Vries, and R.A.M. Stapper. Experimental study of brine transport in porous media, Intraval test case 13. Report 725206003, National Institute of Public Health and Environmental Protection, Bilthoven, the Netherlands, 1990.

[11] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[12] P. Sonneveld. CGS: A fast Lanczos-type solver for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[13] R.A. Trompert. Moorkop, an adaptive grid code for initial-boundary value problems in two space dimensions. Report NM-N9201, CWI, Amsterdam, 1992.

[14] R.A. Trompert and J.G. Verwer. Analysis of the implicit Euler local uniform grid refinement method. Report NM-R9011, CWI, Amsterdam, 1990. (to appear in SIAM J. Sci. Stat. Comput. Vol. 14, #2, 1993).

[15] R.A. Trompert and J.G. Verwer. Runge-Kutta methods and local uniform grid refinement. Report NM-R9022, CWI, Amsterdam, 1990. (to appear in Math. Comp.).

[16] R.A. Trompert and J.G. Verwer. A static-regridding method for two-dimensional parabolic partial differential equations. *Appl. Numer. Math.*, 8:65–90, 1991.

[17] R.A. Trompert, J.G. Verwer, and J.G. Blom. Computing brine transport in porous media with an adaptive-grid method. *Int. J. Numer. Meth. in Fluids*, 16:43–63, 1993.

[18] J.G. Verwer and R.A. Trompert. An adaptive-grid finite-difference method for time-dependent partial differential equations. In D.F. Griffiths and G.A. Watson, editors, *Proc. 14-th Biennial Dundee Conference on Numerical Analysis*, pages 267–284. Pitman Research Notes in Mathematics Series 260, 1992.

[19] J.G. Verwer and R.A. Trompert. Analysis of local uniform grid refinement. Report NM-R9211, CWI, Amsterdam, 1992. (to appear in Appl. Numer. Math.).

[20] H.A. van der Vorst. BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2), 1992.

[21] H.A. van der Vorst and C. Vuik. GMRESR: A family of nested GMRES methods. Report 91-80, Faculty of Technical Mathematics and Informatics, TU Delft, the Netherlands, 1991.