16 SPACE-MARCHING AND DEACTIVATION

For several important classes of problems, the propagation behaviour inherent in the PDEs being solved can be exploited, leading to considerable savings in CPU requirements. Examples where this propagation behaviour can lead to faster algorithms include:

- **detonation**: no change to the flowfield occurs ahead of the denotation wave;
- **supersonic flows**: a change of the flowfield can only be influenced by upstream events, but never by downstream disturbances; and
- **scalar transport**: a change of the transported variable can only occur in the downstream region, and only if a gradient in the transported variable or a source is present.

The present chapter shows how to combine physics and data structures to arrive at faster solutions. Heavy emphasis is placed on space-marching, where these techniques have reached considerable maturity. However, the concepts covered are generally applicable.

16.1. Space-marching

One of the most efficient ways of computing supersonic flowfields is via so-called space-marching techniques. These techniques make use of the fact that in a supersonic flowfield no information can travel upstream. Starting from the upstream boundary, the solution is obtained by marching in the downstream direction, obtaining the solution for the next downstream plane (for structured (Kutler (1973), Schiff and Steger (1979), Chakravarthy and Szema (1987), Matus and Bender (1990), Lawrence et al. (1991)) or semi-structured (McGrory et al. (1991), Soltani et al. (1993)) grids), subregion (Soltani et al. (1993), Nakahashi and Saitoh (1996), Morino and Nakahashi (1999)) or block. In the following, we will denote as a **subregion** a narrow band of elements, and by a **block** a larger region of elements (e.g. one-fifth of the mesh). The updating procedure is repeated until the whole field has been covered, yielding the desired solution.

In order to estimate the possible savings in CPU requirements, let us consider a steady-state run. Using local timesteps, it will take an explicit scheme approximately $O(n_s)$ steps to converge, where $n_s$ is the number of points in the streamwise direction. The total number of operations will therefore be $O(n_t \cdot n_s^2)$, where $n_t$ is the average number of points in the transverse planes. Using space-marching, we have, ideally, $O(1)$ steps per active domain, implying a total work of $O(n_t \cdot n_s)$. The gain in performance could therefore approach $O(1 : n_s)$ for large $n_s$. Such gains are seldomly realized in practice, but it is not uncommon to see gains in excess of 1:10.
Of the many possible variants, the space-marching procedure proposed by Nakahashi and Saitoh (1996) appears as the most general, and is treated here in detail. The method can be used with any explicit time-marching procedure, it allows for embedded subsonic regions and is well suited for unstructured grids, enabling a maximum of geometrical flexibility. The method works with a subregion concept (see Figure 16.1). The flowfield is only updated in the so-called active domain. Once the residual has fallen below a preset tolerance, the active domain is shifted. Should subsonic pockets appear in the flowfield, the active domain is changed appropriately.

![Figure 16.1. Masking of points](image)

In the following, we consider computational aspects of Nakahashi and Saitoh’s space-marching scheme and a blocking scheme in order to make them as robust and efficient as possible without a major change in existing codes. The techniques are considered in the following order: masking of edges and points, renumbering of points and edges, grouping to avoid memory contention, extrapolation of the solution for new active points, treatment of subsonic pockets, proper measures for convergence, the use of space-marching within implicit, time-accurate solvers for supersonic flows and macro-blocking.

16.1.1. MASKING OF POINTS AND EDGES

As seen in the previous chapters, any timestepping scheme requires the evaluation of fluxes, residuals, etc. These operations typically fall into two categories:

(a) point Loops, which are of the form

```fortran
      do ipoin=1,npoin
         do work on the point level
      enddo
```
(b) *edge loops*, which are of the form

```plaintext
do iedge=1,nedge
    gather point information
    do work on the edge level
    scatter-add edge results to points
endo
```

The first loop is typical of unknown updates in multistage Runge–Kutta schemes, initialization of residuals or other point sums, pressure, speed of sound evaluations, etc. The second loop is typical of flux summations, artificial viscosity contributions, gradient calculations and the evaluation of the allowable timestep. For cell-based schemes, point loops are replaced by cell loops and edge loops are replaced by face loops. However, the nature of these loops remains the same. The bulk of the computational effort of any scheme is usually carried out in loops of the second type.

In order to decide where to update the solution, points and edges need to be classified or ‘masked’. Many options are possible here, and we follow the notation proposed by Nakahashi and Saitoh (1996) (see Figure 16.1):

- `maskp=0`: point in downstream, uncomputed field;
- `maskp=1`: point in downstream, uncomputed field, connected to active domain;
- `maskp=2`: point in active domain;
- `maskp=3`: point of `maskp=2`, with connection to points of `maskp=4`;
- `maskp=4`: point in the residual-monitor subregion of the active domain;
- `maskp=5`: point in the upstream computed field, with connection to active domain;
- `maskp=6`: point in the upstream computed field.

The edges for which work has to be carried out then comprise all those for which at least one of the endpoints satisfies `0 < maskp < 6`. These active edges are marked as `maske=1`, while all others are marked as `maske=0`.

The easiest way to convert a time-marching code into a space- or domain-marching code is by rewriting the point- and edge loops as follows.

**Loop 1a:**

```plaintext
do ipoin=1,npoin
    if(maskp(ipoin).gt.0 .and .maskp(ipoin).lt.6) then
        do work on the point level
    endif
endo
```

**Loop 2a:**

```plaintext
do iedge=1,nedge
    if(maske(iedge).eq.1) then
        gather point information
        do work on the edge level
        scatter-add edge results to points
    endif
endo
```
For typical aerodynamic configurations, resolution of geometrical detail and flow features will dictate the regions with smaller elements. In order to be as efficient as possible, the region being updated at any given time should be chosen as small as possible. This implies that, in regions of large elements, there may exist edges that connect points marked as maskp=4 to points marked as maskp=0. In order to leave at least one layer of points in the safety region, a pass over the edges is performed, setting the downstream point to maskp=4 for edges with point markings maskp=2, 0.

16.1.2. RENUMBERING OF POINTS AND EDGES

For a typical space-marching problem, a large percentage of points in Loop 1a will not satisfy the if-statement, leading to unnecessary work. Renumbering the points according to the marching direction has the twofold advantage of a reduction in cache-misses, and the possibility to bound the active point region locally. Defining

npami: the minimum point number in the active region,
npamx: the maximum point number in the active region,
npdmi: the minimum point number touched by active edges,
npdmx: the maximum point number touched by active edges,

Loop 1a may now be rewritten as follows.

Loop 1b:

do ipoin=npami,npamx
   if(maskp(ipoin).gt.0. and .maskp(ipoin).lt.6) then
      do work on the point level
   endif
endo

For the initialization of residuals, the range would become npdmi,npdmx. In this way, the number of unnecessary if-statements is reduced significantly, leading to considerable gains in performance.

As was the case with points, a large number of redundant if-tests may be avoided by renumbering the edges according to the minimum point number. Such a renumbering also reduces cache-misses, a major consideration for RISC-based machines. Defining

neami: The minimum active edge number;
neamx: The maximum active edge number;

Loop 2a may now be rewritten as follows.

Loop 2b:

do iedge=neami,neamx
   if(maske(iedge).eq.1) then
      gather point information
      do work on the edge level
      scatter-add edge results to points
   endif
endo
16.1.3. GROUPING TO AVOID MEMORY CONTENTION

In order to achieve pipelining or vectorization, memory contention must be avoided. The enforcement of pipelining or vectorization is carried out using a compiler directive, as Loop 2b, which becomes an inner loop, and still offers the possibility of memory contention. In this case, we have the following:

\textit{Loop 2c:}

\begin{verbatim}
  do ipass=1, npass
    nedg0 = edpas(ipass) + 1
    nedg1 = edpas(ipass+1)
    c$dir ivdep ! Pipelining directive
    do iedge=nedg0, nedg1
      if(maske(iedge).eq.1) then
        gather point information
        do work on the edge level
        scatter-add edge results to points
      endif
    enddo
  enddo
\end{verbatim}

It is clear that in order to avoid memory contention, for each of the groups of edges (inner loop), none of the corresponding points may be accessed more than once. Given that in order to achieve good pipelining performance on current RISC chips a relatively short vector length of 16 is sufficient, one can simply start from the edge-renumbering obtained before, and renumber the edges further into groups of 16, while avoiding memory contention (see Chapter 15). For CRAYs and NECs, the vector length chosen ranges from 64 to 256.

![Figure 16.2. Near-optimal point-range access of edge groups](image)

The loop structure is shown schematically in Figure 16.2. One is now in a position to remove the \texttt{if}-statement from the innermost loop, situating it outside. The inactive edge groups are marked, e.g. \texttt{edpas(ipass)<0}. This results in the following.
Loop 2d:

do  ipass=1,npass  
nedg0=abs(edpas(ipass))+1  
nedg1= edpas(ipass+1)  
if(nedg1.gt.0) then  
c$dir ivdep  
  do iedge=nedg0,nedg1  
     gather point information  
     do work on the edge level  
     scatter-add edge results to points  
  enddo  
endif  
enddo  

Observe that the innermost loop is the same as that for the original time-marching scheme. The change has occurred at the outer loop level, leading to a considerable reduction of unnecessary if-tests, at the expense of a slightly larger number of active edges, as well as a larger bandwidth of active points.

16.1.4. EXTRAPOLATION OF THE SOLUTION

As the solution progresses downstream, a new set of points becomes active, implying that the unknowns are allowed to change there. The simplest way to proceed for these points is to start from whatever values were set at the beginning and iterate onwards. In many cases, a better way to proceed is to extrapolate the solution from the closest point that was active during the previous timestep. This extrapolation is carried out by looping over the new active edges, identifying those that have one point with known solution and one with unknown solution, and setting the values of the latter from the former. This procedure may be refined by keeping track of the alignment of the edges with the flow direction and extrapolating from the point that is most aligned with the flow direction (see Figure 16.3). Given that more than one layer of points may be added when a new region is updated, an open loop over the new edges is performed, until no new active points with unknown solution are left. This extrapolation of the unknowns can significantly reduce the number of iterations required for convergence, making it well worth the effort.

![Figure 16.3. Extrapolation of the solution](image-url)
16.1.5. TREATMENT OF SUBSONIC POCKETS

The appearance of subsonic pockets in a flowfield implies that the active region must be extended properly to encompass it completely. Only then can the ‘upstream-only’ argument be applied.

In this case, the planes are simply shifted upstream and downstream in order to satisfy this criterion. For small subsonic pockets, which are typical of hypersonic airplanes, a more expedient way to proceed is shown in Figure 16.4. The spatial extent of subsonic points upstream and downstream of the active region is obtained, leading to the ‘conical’ regions $C_u, C_d$. All edges and points in these regions are then marked as $l\text{poin}(\text{ipoin})=2, 4$, respectively. All other steps are kept as before. Subsonic pockets tend to change during the initial formation and subsequent iterations. In order to avoid the repeated marking of points and edges, the conical regions are extended somewhat. Typical values of this ‘safety zone’ are $s = 0.1–0.2 \, dx_{saf}$.

![Figure 16.4. Treatment of subsonic pockets](image)

16.1.6. MEASURING CONVERGENCE

Any iterative procedure requires a criterion to decide when the solution has converged. If we write an explicit time-marching scheme as

$$
M_t \Delta u^n = R^n, \tag{16.1}
$$

where $R^n$ and $\Delta u^n$ denote the residual and change of unknowns for the $n$th timestep, respectively, and $M_t$ is the lumped mass matrix, the convergence criterion most commonly used is some global integral of the form

$$
r^n = \int_{\Omega} |\Delta u^n| \, d\Omega \approx \sum_i M'_t |\Delta \hat{u}_i^n|. \tag{16.2}
$$
is compared to \( r^n \) and, if the ratio of these numbers is sufficiently small, the solution is assumed converged. Given that for the present space-marching procedure the residual is only measured in a small but varying region, this criterion is unsatisfactory. One must therefore attempt to derive different criteria to measure convergence. Clearly the solution may be assumed to be converged if the maximum change in the unknowns over the points of the mesh has decreased sufficiently:

\[
\max_i (|\Delta \hat{u}_i^n|) < \epsilon_0. \quad (16.3)
\]

In order to take away the dimensionality of this criterion, one should divide by an average or maximum of the unknowns over the domain:

\[
\frac{\max_i (|\Delta \hat{u}_i^n|)}{\max_i (\hat{u}_i^n)} < \epsilon_1. \quad (16.4)
\]

The quantity \( \Delta u \) depends directly on the timestep, which is influenced by the Courant number selected. This dependence may be removed by dividing by the Courant number \( CFL \) as follows:

\[
\frac{\max_i (|\Delta \hat{u}_i^n|)}{CFL \max_i (\hat{u}_i^n)} < \epsilon_2. \quad (16.5)
\]

This convergence criterion has been found to be quite reliable, and has been used for the examples shown below. When shocks are present in the flowfield, some of the limiters will tend to switch back and forth for points close to the shocks. This implies that, after the residual has dropped to a certain level, no further decrease is possible. This ‘floating’ or ‘hanging up’ of the residuals has been observed and documented extensively. In order not to iterate \textit{ad infinitum}, the residuals are monitored over several steps. If no meaningful decrease or increase is discerned, the spatial domain is updated once a preset number of iterations in the current domain has been exceeded.

### 16.1.7. APPLICATION TO TRANSIENT PROBLEMS

The simulation of vehicles manoeuvring in supersonic and hypersonic flows, or aeroelastic problems in this flight regime, require a time-accurate flow solver. If an implicit scheme of the form (e.g. Alonso et al. (1995))

\[
\frac{3}{2} M_{l}^{n+1} u^{n+1} - 2 M_{l}^{n} u^{n} + \frac{1}{2} M_{l}^{n-1} u^{n-1} = \Delta t R^{n+1} \quad (16.6)
\]

is solved using a pseudo-timestep approach as

\[
\frac{d}{d\tau} u + R^* = 0, \quad (16.7)
\]

where

\[
R^* = \frac{3}{2} M_{l}^{n+1} u^{n+1} - 2 M_{l}^{n} u^{n} + \frac{1}{2} M_{l}^{n-1} u^{n-1} - \Delta t R^{n+1}, \quad (16.8)
\]

an efficient method of solving this pseudo-timestep system for supersonic and hypersonic flow problems is via space-marching.
16.1.8. MACRO-BLOCKING

The ability of the space-marching technique described to treat possible subsonic pockets requires the availability of the whole mesh during the solution process. This may present a problem for applications requiring very large meshes, where machine memory constraints can easily be reached. If the extent of possible subsonic pockets is known – a situation that is quite common – the computational domain can be subdivided into subregions, and each can be updated in turn. In order to minimize user intervention, the mesh is first generated for the complete domain. This has the advantage that the CAD data does not need to be modified. This large mesh is then subdivided into blocks. Since memory overhead associated with splitting programs is almost an order of magnitude less than that of a flow code, even large meshes can be split without reaching the memory constraints the flow code would have for the smaller sub-domain grids.

Once the solution is converged in an upstream domain, the solution is extrapolated to the next downstream domain. It is obvious that boundary conditions for the points in the upstream ‘plane’ have to be assigned the ‘no allowed change’ boundary condition of supersonic inflow. For the limiting procedures embedded in most supersonic flow solvers, gradient information is required at points. In order to preserve full second-order accuracy across the blocks, and to minimize the differences between the uni-domain and blocking solutions, the second layer of upstream points is also assigned a ‘no allowed change’ boundary condition (see Figure 16.5). At the same time, the overlap region between blocks is extended by one layer. Figure 16.6 shows the solutions obtained for a 15° ramp and $Ma_\infty = 3$ employing the usual uni-domain scheme, and two blocking solutions with one and two layers of overlap respectively. As one can see, the differences between the solutions are small, and are barely discernable for two layers of overlap.

![Figure 16.5. Macro-blocking with two layers of overlap](image)

Within each sub-domain, space-marching may be employed. In this way, the solution is obtained in an almost optimal way, minimizing both CPU and memory requirements.
16.1.9. EXAMPLES FOR SPACE-MARCHING AND BLOCKING

The use of space-marching and macro-blocking is exemplified on several examples. In all of these examples the Euler equations are solved, i.e. no viscous effects are considered.

16.1.9.1. Supersonic inlet flow

This internal supersonic flow case, taken from Nakahashi and Saitoh (1996), represents part of a scramjet intake. The total length of the device is \( l = 8.0 \), and the element size was set uniformly throughout the domain to \( \delta = 0.03 \). The cross-section definition is shown in Figure 16.7(a). Although this is a 2-D problem, it is run using a 3-D code. The inlet Mach number was set to \( Ma = 3.0 \). The mesh (not shown, since it would blacken the domain) consisted of 540 000 elements and 106 000 points, of which 30 000 were boundary points. The flow solver is a second-order Roe solver that uses MUSCL reconstruction with pointwise gradients and a vanAlbada limiter on conserved quantities. A three-stage scheme with a Courant number of \( CFL = 1.0 \) and three residual smoothing passes were employed. The convergence criterion was set to \( \epsilon_2 = 10^{-3} \).

The Mach numbers obtained for the space-marching and the usual time-marching procedure are superimposed in Figure 16.7(a). As one can see, these contours are almost indistinguishable, indicating that the convergence criterion used is proper. The solution was also obtained using blocking. The individual blocking domains are shown for clarity in Figure 16.7(b). The five blocks consisted of 109 000, 103 000, 126 000, 113 000 and 119 000 elements, respectively. The convergence history for all three cases – usual timestepping, space-marching and blocking – is summarized in Figure 16.7(c). Table 16.1 summarizes the CPU requirements on an SGI R10000 processor for different marching and safety-zone sizes, as well as for usual time-marching and blocking.

The first observation is that although this represents an ideal case for space-marching, the speedup observed is not spectacular, but worth the effort. The second observation is that the
Figure 16.7. Mach number: (a) usual versus space-marching, min = 0.825, max = 3.000, incr = 0.05; (b) usual versus blocking min = 0.825, max = 3.000, incr = 0.05; (c) convergence history for inlet

Table 16.1. Timings for inlet (540 000 elements)

<table>
<thead>
<tr>
<th>dxmar</th>
<th>dxsaf</th>
<th>CPU (min)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usual</td>
<td></td>
<td>400</td>
<td>1.00</td>
</tr>
<tr>
<td>0.05</td>
<td>0.20</td>
<td>160</td>
<td>2.50</td>
</tr>
<tr>
<td>0.10</td>
<td>0.40</td>
<td>88</td>
<td>4.54</td>
</tr>
<tr>
<td>0.10</td>
<td>0.60</td>
<td>66</td>
<td>6.06</td>
</tr>
<tr>
<td>Block</td>
<td></td>
<td>140</td>
<td>2.85</td>
</tr>
</tbody>
</table>

speedup is sensitive to the safety zone ahead of the converged solution. This is a user-defined parameter, and a convincing way of choosing automatically this distance has so far remained elusive.

16.1.9.2. F117

As a second case, we consider the external supersonic flow at $Ma = 4.0$ and $\alpha = 4.0^\circ$ angles of attack over an F117-like geometry. The total length of the airplane is $l = 200.0$. 

The unstructured surface mesh is shown in Figure 16.8(a). Smaller elements were placed close to the airplane in order to account for flow gradients. The mesh consisted of 2,056,000 elements and 367,000 points, of which 35,000 were boundary points. As in the previous case, the flow solver is a second-order Roe solver that uses MUSCL reconstruction with pointwise gradients and a vanAlbada limiter on conserved quantities. A three-stage scheme with a Courant number of $\text{CFL}=1.0$ and three residual smoothing passes were employed.
The convergence criterion was set to $\epsilon_2 = 10^{-4}$. The Mach numbers obtained for the space-marching, usual time-marching and blocking procedures are superimposed in Figures 16.8(b) and (c). The individual blocking domains are shown for clarity in Figure 16.8(d). The seven blocks consisted of 357,000, 323,000, 296,000, 348,000, 361,000, 386,000 and 477,000 elements, respectively. As before, these contours are almost indistinguishable, indicating a proper level of convergence. Table 16.2 summarizes the CPU requirements on an SGI R10000 processor for different marching and safety-zone sizes, macro-blocking, as well as for usual time-marching and grid sequencing. The coarser meshes consisted of 281,000 and
Table 16.2. Timings for F117 (543 000 tetrahedra, 106 000 points)

<table>
<thead>
<tr>
<th>dxmax</th>
<th>dxsaf</th>
<th>CPU (min)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usual</td>
<td></td>
<td>611</td>
<td>1.00</td>
</tr>
<tr>
<td>Seque</td>
<td></td>
<td>518</td>
<td>1.17</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>227</td>
<td>2.69</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>218</td>
<td>2.80</td>
</tr>
<tr>
<td>Block</td>
<td></td>
<td>260</td>
<td>2.35</td>
</tr>
</tbody>
</table>

42 000 elements respectively. The residual curves for the three different cases are compared in Figure 16.8(e). As one can see, grid sequencing only provides a marginal performance improvement for this case. Space-marching is faster than blocking, although not by a large margin.

16.1.9.3. Supersonic duct flow with moving parts

This case simulates the same geometry and inflow conditions as the first case. The center-piece, however, is allowed to move in a periodic way as follows:

\[ x_c = x_c^0 + a \cdot \sin(\omega t). \]

For the case considered here, \( x_c^0 = 4.2, a = 0.2 \) and \( \omega = 0.05 \). The mesh employed is the same as that of the first example, and the same applies to the spatial discretization part of the flow solver used. The implicit timestepping scheme given by (16.6) is used to advance the solution in time, and the pseudo-timestepping of the residuals, given by (16.7), is carried out using space-marching, with the convergence criterion set to \( \epsilon_2 = 10^{-3} \). Each period was discretized by 40 timesteps, yielding a timestep \( \Delta t = \pi \) and a Courant number of
approximately 300. The number of space-marching steps required for each implicit timestep was approximately 600, i.e. similar to one steady-state run. Figure 16.9 shows the Mach number distribution at different times during the third cycle.

Figure 16.9. Inlet flowfield with oscillating inner part, Mach number: min = 0.875, max = 3.000 incr = 0.05

16.2. Deactivation

The space-marching procedure described above achieved CPU gains by working only on a subset of the complete mesh. The same idea can be used advantageously in other situations, leading to the general concept of deactivation. Two classes of problems where deactivation has been used extensively are point detonation simulations (Löhner et al. (1999b)) and scalar transport (Löhner and Camelli (2004)). In order to mask points and edges (faces, elements) in an optimal way, and to avoid any unnecessary if-statements, the points are renumbered according to the distance from the origin of the explosion, or in the streamline direction. This idea can be extended to multiple explosion origins and to recirculation zones, although in these cases sub-optimal performance is to be expected. For the case of explosions, only the points and edges that can have been reached by the explosion are updated. Similarly, for scalar transport problems described by the classic advection-diffusion equation

$$c_{,t} + \mathbf{v} \cdot \nabla c = \nabla k \nabla c + S,$$

(16.10)

where \(c\), \(\mathbf{v}\), \(k\) and \(S\) denote the concentration, velocity, diffusivity of the medium and source term, respectively, a change in \(c\) can only occur in those regions where

$$|S| > 0, \quad |\nabla c| > 0.$$

(16.11)

For the typical contaminant transport problem, the extent of the regions where \(|S| > 0\) is very small. In most of the regions that lie upwind of a source, \(|\nabla c| = 0\). This implies that in a
considerable portion of the computational domain no contaminant will be present, i.e. $c = 0$. As stated before, the basic idea of deactivation is to identify the regions where no change in $c$ can occur, and to avoid unnecessary work in them.

The marking of deactive regions is accomplished in two loops over the elements. The first loop identifies in which elements sources are active, i.e. where $|S| > 0$. The second loop identifies in which elements/edges a change in the values of the unknowns occurs, i.e. where $\max(c_e) - \min(c_e) > \epsilon_u$, with $\epsilon_u$ a preset, very small tolerance. Once these active elements/edges have been identified, they are surrounded by additional layers of elements which are also marked as active. This ‘safety’ ring is added so that changes in neighbouring elements can occur, and so that the test for deactivation does not have to be performed at every timestep. Typically, four to five layers of elements/edges are added. From the list of active elements, the list of active points is obtained. The addition of elements to form the ‘safety’ ring can be done in a variety of ways. If the list of elements surrounding elements or elements surrounding points is available, only local operations are required to add new elements. If these lists are not present, one can simply perform loops over the edges, marking points, until the number of ‘safety layers’ has been reached. In either case, it is found that the cost of these marking operations is small compared to the advancement of the transport equation.

16.2.1. EXAMPLES OF DYNAMIC DEACTIVATION

The use of dynamic deactivation is exemplified on several examples.

16.2.1.1. Generic weapon fragmentation

The first case considered is a generic weapon fragmentation, and forms part of a fully coupled CFD/CSD run (Baum et al. (1999)). The structural elements are assumed to fail once the average strain in an element exceeds 60%. At the beginning, the fluid domain consists of two separate regions. These regions connect as soon as fragmentation starts. In order to handle narrow gaps during the break-up process, the failed structural elements are shrunk by a fraction of their size. This alleviates the timestep constraints imposed by small elements without affecting the overall accuracy. The final breakup leads to approximately 1200 objects in the flowfield. Figure 16.10 shows the fluid pressure, the mesh velocity and the surface velocity of the structure at three different times during the simulation. The edges and points are checked every 5 to 10 timesteps and activated accordingly. The deactivation technique leads to considerable savings in CPU at the beginning of a run, where the timestep is very small and the zone affected by the explosion only comprises a small percentage of the mesh. Typical meshes for this simulation were of the order of 8.0 million tetrahedra, and the simulations required of the order of 50 hours on the SGI Origin2000 running on 32 processors.

16.2.1.2. Subway station

The second example considers the dispersion of an instantaneous release in the side platform of a generic subway station, and is taken from Löhner and Camelli (2004). The geometry is shown in Figure 16.11(a).
Figure 16.10. Pressure, mesh and fragment velocities at three different times

A time-dependent inflow is applied on one of the end sides:

\[ v(t) = b(t - 60)^3 e^{-a(t-60)} + v_0 \]
Figure 16.11. (a) Problem definition; (b), (c) iso-surface of concentration $c = 0.0001$; (d), (e) surface velocities

where $b = 0.46\; \text{m/s}, a = 0.5\; \text{l/s},$ and $\nu_0 = 0.4\; \text{m/s}$. This inflow velocity corresponds approximately to the velocities measured at a New York City subway station (Pflistch et al. (2000)). The Smagorinsky model with the Law of the Wall was used for this example. The volume grid has 730 000 elements and 144 000 points. The dispersion simulation was
performed using a three-stage Runge–Kutta scheme with a Courant number of $C = 0.6$. The dispersion calculation was run for 485 s of real time (corresponding to the time of a train entering, halting, exiting the station and the time for the next train to arrive) on a workstation with the following characteristics: Dec Alpha chip running at 0.67 GHz, 4 Gbyte of RAM, Linux operating system, Compaq compiler. Figures 16.11(b)–(e) show the resulting iso-surface of concentration level $c = 0.0001$, as well as the surface velocities for time $t = 485$ s. Note the transient nature of the flowfield, which is reflected in the presence of many vortices.

Deactivation checks were performed every 5 timesteps. The tolerance for deactivation was set to $\epsilon_u = 10^{-3}$. The usual run (i.e. without deactivation) took $T = 5,296$ sec, whereas the run with deactivation took $T = 526$ sec, i.e. a saving in excess of 1:10. This large reduction in computing time was due to two factors: the elements with the most constraining timestep are located at the entry and exit sections, and the concentration cloud only reaches this zone very late in the run (or not at all); furthermore, as this is an instantaneous release, the region of elements where concentration is present in meaningful values is always very small as compared to the overall domain. Speedups of this magnitude have enabled the use of 3-D CFD runs to assess the maximum possible damage of contaminant release events (Camelli (2004)) and the best possible placement of sensors (Löhner (2005)).