International Journal of Automation and Computing, **3**, (3), pp. 252–262, 2006

Real-time Design Constraints in Implementing Active Vibration Control Algorithms

M. A. Hossain and M. O. Tokhi*

Department of Computing, School of Informatics,
University of Bradford, Bradford, BD7 1DP, UK
Email: m.a.hossain1@bradford.ac.uk

*Department of Automatic Control and Systems Engineering,
The University of Sheffield, Sheffield, S1 3JD, UK
Email: O.Tokhi@sheffield.ac.uk

ABSTRACT
Although computer architectures incorporate fast processing hardware resources, high performance real-time implementation of a complex control algorithm requires an efficient design and software coding of the algorithm so as to exploit special features of the hardware and avoid associated shortcomings of the architecture. This paper presents an investigation into the analysis and design mechanisms that will lead to reduction of the execution time in implementing real-time control algorithms. The proposed mechanisms are exemplified by means of one algorithm, which demonstrates the applicability of these mechanisms to real-time applications. An active vibration control (AVC) algorithm for a flexible beam system simulated using the finite difference (FD) method is considered to demonstrate the effectiveness of the proposed methods. A comparative performance evaluation of the proposed design mechanisms is presented and discussed through a set of experiments.

Keywords: Algorithm analysis and design, active vibration control, flexible beam system, real-time control, memory management.

1 Introduction

Analysis and design of algorithms are currently subjects of widespread interest among researchers and scientists. Accordingly a new scientific subject has emerged during the 1960s and has quickly been established as one of the most active fields of study and an important topic in computer and systems engineering. The reason for this sudden interest in the study of algorithms is not difficult to trace as a fast and successful development of digital computers and their uses in many different areas of human activity, which have led to the construction of a great variety of computer algorithms. In many cases, analysis of algorithms leads to the revelation of completely new algorithms that are even faster than all available algorithms. In general, a goal of algorithmic analysis is to obtain sufficient understanding of the relative merits of complicated algorithms so as to provide useful information to someone undertaking an actual computation.

In practice, more than one algorithm exists for solving a specific problem. Depending on the formulation, each can be evaluated numerically in different ways. As computer arithmetic is of finite accuracy, different results can evolve, depending on the algorithm used and the way it is evaluated. On the other hand, the same computing domain could offer different performances due to variation in the algorithm design and in turn, source code implementation. The choice of the best algorithm for a given problem and for a specific computer is a difficult task and depends on many factors, for instance, data and control dependencies of the algorithm, regularity and granularity of the algorithm and architectural features of the computer domain [1], [2].

The ideal performance of a computer system demands a perfect match between machine capability and program behaviour. Program performance is the turnaround time, which includes, disk and memory accesses, input and output activities, compilation time, operating system overhead, and central processing unit (CPU) time. In order to shorten the turnaround time, one can reduce all these time factors. Minimising the run-time memory management, efficient partitioning and mapping of the program, and selecting an efficient compiler for specific computational demands, could enhance the performance. Compilers have a significant impact on the performance of the system. This means that some high-level languages have advantages in certain computational domains, and some have advantages in other domains. The compiler itself is critical to the performance of the system as the mechanism and efficiency of taking a high-level description of the application and transforming it into a hardware dependent implementation differs from compiler to compiler [3], [4].

Performance is also related to program optimisation facility of the compiler, which may be machine dependent. The goal of program optimisation is, in general, to maximise the speed of code execution. This involves several factors such as minimisation of code length and memory accesses, exploitation of parallelism, elimination of dead code, in-line function expansion, loop unrolling and maximum utilisation of registers. The optimisation techniques include vectorization using pipelined hardware and parallelization using multiprocessors simultaneously [5].

The performance demand in modern real-time signal processing and control applications has motivated the development of advanced special-purpose and general-purpose hardware architectures. However, the developments within the software domain have not been at the same pace and/or level as within the hardware domain. Thus, although advanced computing hardware with significant levels of capability is available in the market, these capabilities are not fully utilised and exploited at the software level. Efficient software coding is essential in order to exploit the special hardware features and avoid associated shortcomings of the architecture. There has been a substantial amount of effort devoted to this area of research over the last decade [6], [7], [8].

It is essential for enhanced performance of a computing domain that a characteristic matching between the computing requirements of an algorithm and computing capabilities of the computing domain is made. Moreover, source code and corresponding memory management facility of the computing domain play an important role in its overall performance in implementing an algorithm. This further includes the memory access time required during execution of a program code. Some special-purpose digital signal processing (DSP) devices, for example the Texas Instruments TMS320 devices, incorporate instructions, at the assembly language level, that allow executing commonly occurring operations in digital filtering applications, such as multiply, add and shift together. Such facilities attempt to minimise the memory access time and hence enhance the performance of the processor [9], [10].

This paper addresses the issue of algorithm analysis, design and software coding for real-time active control systems. A number of design methodologies are proposed for the real-time implementation of an AVC algorithm. The proposed methodologies are exemplified and demonstrated with FD simulation algorithm of a flexible beam system within the framework of AVC. Finally, a comparative performance of the proposed design mechanisms is presented and discussed through a set of experimental investigations.

2 Active Vibration Control Algorithm

Consider a cantilever beam system with a force $U(x,t)$ applied at a distance x from its fixed (clamped) end at time t. This will result in a deflection $y(x,t)$ of the beam from its stationery position at the point where the force has been applied. In this manner, the governing dynamic equation of the beam is given by

$$\mu^2 \frac{\partial^4 y(x,t)}{\partial x^4} + \frac{\partial^2 y(x,t)}{\partial t^2} = \frac{1}{m} U(x,t) \qquad (1)$$

where, μ is a beam constant and m is the mass of the beam. Discretising the beam in time and length using central FD methods, a discrete approximation to equation (1) can be obtained as [11], [12]:

$$Y_{k+1} = -Y_{k-1} - \lambda^2 S Y_k + \frac{(\Delta t)^2}{m} U(x,t) \qquad (2)$$

where, $\lambda^2 = \left[(\Delta t)^2 / (\Delta x)^4\right]\mu^2$ with Δt and Δx representing the step sizes in time and along the beam respectively,

$$Y_{k+1} = \begin{bmatrix} y_{1,k+1} \\ y_{2,k+1} \\ \vdots \\ y_{n,k+1} \end{bmatrix}, \quad Y_j = \begin{bmatrix} y_{1,k} \\ y_{2,k} \\ \vdots \\ y_{n,k} \end{bmatrix}, \quad Y_{j-1} = \begin{bmatrix} y_{1,k-1} \\ y_{2,k-1} \\ \vdots \\ y_{n,k-1} \end{bmatrix},$$

and S is a penta-diagonal matrix, given (for $n = 20$, say) as:

$$S = \begin{bmatrix} a & -4 & 1 & 0 & 0 & 0 & \cdots & \cdots & 0 \\ -4 & b & -4 & 1 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & -4 & b & -4 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & -4 & b & -4 & 1 & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & 1 & -4 & b & -4 & 1 \\ \cdots & \cdots & \cdots & \cdots & 0 & 1 & -4 & c & -2 \\ \cdots & \cdots & \cdots & \cdots & 0 & 0 & 2 & -4 & d \end{bmatrix}$$

where, $a = 7 - \frac{7}{\lambda^2}$, $b = 6 - \frac{2}{\lambda^2}$, $c = 5 - \frac{2}{\lambda^2}$ and $d = 2 - \frac{2}{\lambda^2}$. Equation (2) is the required relation for the simulation algorithm, characterising the behaviour of the cantilever beam system, which can be implemented on a digital computer easily. For the algorithm to be stable it is required that the iterative scheme described in equation (2), for each grid point, converges to a solution. It has been shown that a necessary and sufficient condition for stability satisfying this convergence requirement is given by $0 < \lambda^2 \leq 0.25$ [12].

A schematic diagram of an AVC structure is shown in Fig. 1. A detection sensor detects the unwanted (primary) disturbance. This is processed by a controller to generate a cancelling (secondary, control) signal so that to achieve cancellation at the observation point. The objective in Fig. 1 is to achieve total (optimum) vibration suppression at the observation point. Synthesising the controller on the basis of this objective yields [13],

$$C = \left[1 - \frac{Q_1}{Q_0} \right]^{-1} \qquad (3)$$

where, Q_0 and Q_1 represent the equivalent transfer functions of the system (with input at the detector and output at the observer) when the secondary source is *off* and *on* respectively.

To investigate the nature and real-time processing requirements of the AVC algorithm, it is divided into two parts, namely control and identification. The control part is tightly coupled with the simulation algorithm, and both will be described in an integral manner as the control algorithm. The simulation algorithm will also be explored as a distinct algorithm. Both of these algorithms are predominately matrix based. The identification algorithm consists of parameter estimation of the models Q_0 and Q_1 and calculation of the required controller parameters according to equation (3). However, the nature of identification algorithm is completely different as compared with the simulation and control algorithms [10]. Thus, for reasons of consistency only the simulation and control algorithms are considered in this investigation.

3 Algorithm Design

3.1 Beam Simulation Algorithm

The beam simulation algorithm is of regular iterative type. In implementing this algorithm on a sequential vector processor a performance better than with any other processor can be expected. The algorithm processes floating-point data, which is computed within a small iterative loop. Accordingly, the performance is further enhanced if the processor has internal/external data cache and instruction cache, built-in maths co-processor etc.

The simulation algorithm in equation (2) can be expressed for exchange of information, for computing the deflection of segments 8 and 16, as in Fig. 2, assuming no external force is applied at these points.

It follows from the above that computation of deflection of a segment at time step *t* can be described as in Fig. 3. It is also noted that computation of deflection of a particular segment is dependent on the deflection of six other segments. These heavy dependencies could be major causes of performance degradation in real-time sequential computing, due to memory access time. On the other hand, these dependencies might cause significant performance degradation in real-time parallel computing due to inter-processor communication overheads.

To explore this issue, a number of design mechanisms for the beam simulation algorithm were developed in a real-time performance context. Seven designs of the simulation algorithm were developed and tested through a set of experiments [14], [5]. These designs are considered here for further investigation in the AVC framework. The algorithms for different designs are described through Fig.(s) 3 to 13.

3.1.1 Beam Algorithm–1: Shifting of data array

Algorithm–1 was adopted from a previously reported work [5]. The algorithm is listed in Fig. 4. It is noted that complex matrix calculations are performed within an array of three elements each representing information about the beam position at different instants of time. Following these calculations, the memory pointer is shifted to the previous pointer time step before the next iteration. This technique of shifting the pointer does not contribute to the calculation efforts and is thus a program overhead. Other algorithms were deployed to address this issue further.

3.1.2 Beam Algorithm–2: Array rotation

Algorithm–2 incorporates design suggestions made by [14]. A listing of Algorithm–2 is given in Fig. 5. In this case, each loop calculates three sets of data. Instead of shifting the data of the memory pointer (that contains results) at the end of each loop, the most current data is directly recalculated and written into the memory pointer that contains the older set of data. Therefore, re-ordering of array in Algorithm–1 is replaced by recalculation. The main objective of the design effort is to achieve better performance by reducing the dynamic memory allocation and, in turn, memory pointer shift operation. Thus, instead of using a single code block and data-shifting portion, as in Algorithm–1, to calculate the deflection, three code blocks, are used with the modified approach in Algorithm–2. It is worth noting that in Algorithm–2, the overhead of Algorithm 1 due to memory pointer shift operation is eliminated and every line of code is directed towards the simulation effort.

3.1.3 Beam Algorithm–3: Large array and less frequent shifting

In Algorithm–1 shifting of memory pointers was required in each iteration. Algorithm–3 was developed as an attempt to reduce the number of memory pointer shifting instructions and thereby to decrease program overhead. An array of 1000 elements was considered for each beam segment. This array size was chosen rather arbitrarily, but small enough to allow easy allocation of these monolithic memory blocks within typical hardware boundaries. Fig. 6 shows how the array is utilised in Algorithm–3. Shifting occurs at the end of every thousandth iteration, rendering the overhead produced at this stage negligible. However, array positions are indirectly referenced through a variable, accessed at run-time, which, in turn, lead to an overhead. Of far greater concern to program performance is the fact that large data structures need to be dealt with. Therefore, the internal data cache struggles to handle large amount of data.

3.1.4 Beam Algorithm–4: Nested loops and shifting

Algorithm–4 incorporates merely a minor modification of Algorithm–1, as shown in Fig. 7. The aim in this algorithm is to contain the number of instructions inside the main loop, and thus, reduce the instruction size of the program. This was accomplished by nesting secondary loops inside the main iterations. Complex substitutions need to be carried out to determine which matrix elements need to be referred to for performing the ongoing calculations. A moderate amount of overhead resulting from these necessary substitutions was anticipated. The benefits of this algorithm include quicker compilation, greater flexibility in respect of the number of segments (possibly changes at run-time) and a fixed number of program instructions in the main loop as segment sizes are increased. The likelihood of cache misses in the instruction cache was significantly reduced.

3.1.5 Beam Algorithm–5: Nested loops and array rotation

Fig. 8 shows a listing of Algorithm–5, in which the new methods of Algorithm–4 were applied with the concepts of Algorithm–2. Three distinct calculation runs are performed during each iteration, but instead of listing the instructions for each segment separately, nested loops are used to limit the number of instructions (source code lines) in the main program loop. The benefits of employing this technique are identical with those listed in the description of Algorithm–4. However, it possesses the same disadvantage of overhead produced by the complex substitutions required.

3.1.6 Beam Algorithm–6: Two-element array rotation

Algorithm–6 is shown in Fig. 9. This makes use of the fact that access to the oldest time segment is only necessary during re-calculation of the same longitudinal beam segment. Hence, it can directly be overwritten with the new value as shown in Fig. 10.

Fig.(s) 11 and 12 show simplified flow diagrams of Algorithm–2 and Algorithm–6, respectively. The conventional re-calculation algorithm in Fig. 4 requires three memory segments in the time domain. In contrast, Algorithm–6 is optimised for the particular discrete mathematical approximation of the governing physical formula, exploiting the previously observed features.

It is noted that this particular algorithm is not suitable for applications for which the previous assumption does not hold. This technique gives a major performance advantage over the conventional rotation method, in particular when the number of beam segments is increased.

3.1.7 Beam Algorithm–7: Nested loops two-element array and rotation

Algorithm–7, as shown in Fig. 13, is based on improvements achieved with Algorithm–6. Additionally, the notion of nested loops was incorporated. The advantages and disadvantages of this approach were identified earlier and remain true for this particular algorithm.

3.2 Control Algorithm

As mentioned earlier, the AVC algorithm consists of the beam simulation algorithm and control algorithm. For simplicity the control algorithm in equation (3) can be rewritten as a difference equation as in Fig. 14 (Hossain, 1995), where, b0, …, b4, and a0, …, a3 represent controller parameters. The arrays y12 and yc denote input and controller output, respectively. It is noted that the control algorithm shown in Fig. 14 has similar design and computational complexity as one of the beam segments described and discussed in the beam simulation Algorithm-1.

4 Implementation and results

The AVC algorithms based on seven different methods of the beam simulation and the control algorithms were implemented with similar specification using the C programming language on a uniprocessor computing domain for similar specification [7]. It is worth mentioning that seven different forms of the AVC algorithm were implemented based on the seven different forms of the beam simulation algorithm. Thus, the AVC algorithm Alg-1 design consists of the beam simulation Algorithm-1 and control algorithm in Fig. 14. Similarly, AVC algorithm Alg-2 is formed by combining the beam simulation Algorithm-2 and the control algorithm in Fig. 14 and so on. Thus, the seven different forms of AVC algorithm were implemented, tested and verified. It is worth noting that a fixed number of iterations (250,000) was considered in implementing all the algorithms for reasons of consistency.

To explore the controller performance of the design mechanisms, all the seven forms of the AVC algorithm were implemented for 20 segments. Although the AVC algorithm is designed in different forms, the resultant outcomes of all these forms are maintained the same. Fig.(s) 15, 16 and 17 show the performance of the AVC algorithm using Alg-1. Fig. 15 shows the beam fluctuation before cancellation and Fig. 16 shows the corresponding fluctuation after cancellation. These diagrams demonstrate the capabilities and dynamic behaviour of the resultant controller. This is further demonstrated in Fig. 17, which shows the auto-power spectral density at the end point of the beam before and after cancellation. As mentioned earlier, the main objective of this investigation is to maintain the same processing output with different forms of the algorithm so as to demonstrate the comparative real-time computing performance in implementing the AVC algorithm. Therefore, performances of the other forms of the AVC system are not included here to avoid duplication.

To explore the comparative real-time computing performance of the design mechanisms, all the seven forms of the AVC algorithm were implemented for 20 segments. The execution time performance of the algorithms relative to Alg-1 is shown in Table I. It is observed that Alg-3 was the slowest among all the algorithms. On the other hand, Alg-2 performs best among all the design mechanisms. Alg-6 performed better than Alg-1 but was slower than Alg-2. It is also observed that Alg-4 is almost 2.5 times slower than Alg-1. This is further demonstrated in Fig. 18, where Alg-3 has not been incorporated due to its poor performance as compared to other designs of the algorithm. It is noted that Alg-4 performed worst among the six design mechanisms of the algorithm shown in Fig. 18.

To explore performance of the design mechanisms further, all designs of the algorithm, except Alg-3, were implemented with different number of segments. Fig. 19 depicts comparative performance of Alg–1 and Alg–2 for 20 to 200 segments. It is noted that execution time for both algorithms increases almost linearly with increasing the number of segments. It is also noted that Alg-2 performs better throughout except for the 100 segments case.

Fig. 20 shows the comparative real-time performance in implementing Alg-6 and Alg-7. It is noted that Alg-6 performs better throughout. It is also noted that the performance variation of Alg-6 as compared to the Alg-7 is not linear and it performs best for the 80 segments case. Table II presents further details to demonstrate the performance of all the different designs of the AVC algorithm relative to Alg-1.

Table II shows the performance ratio of the different forms of the algorithm relative to Alg-1. It is noted that Alg-4 performed worst throughout. It is also noted that the transition towards weaker performance occurred in AVC Alg–6 halfway between the transitions of Alg–1 and Alg–2. In spite of being outperformed by Alg–1 in a narrow band of around 100 segments, Alg–6 offers the best performance overall. Thus, the design mechanism employed in Alg–3 can offer potential advantages in real-time control applications.

5 Concluding Remarks

An investigation into the analysis, design, software coding and implementation of algorithms so as to reduce the execution time and, in turn, enhance the real-time performance of the algorithm, has been presented within the framework of real-time implementation of an active control algorithm. A number of design approaches have been proposed and demonstrated with the control algorithm of a flexible beam. The same resultant outcomes with the different forms in implementing the AVC algorithm have been maintained so as to demonstrate the comparative real-time computing performances. It has been observed that the execution time and in turn, performance of an algorithm varies with different approaches in a real-time implementation context. It is also noted from the investigations that a design based on reduced instructions provides linear performance, although in most cases these are slower. On the other hand, designs leading to large number of instructions cause non-linear transitions at certain stages where internal built-in instruction cache is unable to handle the load. It is worth mentioning that such transitions with the control algorithms considered occur with computation of different number of segments. Therefore, identification of the suitability of source code design and implementation mechanism for best performance is a challenge. As a whole, the proposed approaches can have a significant impact on the design and real-time implementation of real-time control algorithms.

6 References

[1] A. U. Thoeni,. Programming real-time multicomputers for signal processing. Prentice-Hall, Hertfordshire, 1994.

[2] M. O. Tokhi and M. A. Hossain. CISC, RISC and DSP processors in real-time signal processing and control, *Journal of Microprocessors and Microsystems*, vol. 19, no. 5, UK. pp. 291-300, 1995.

[3] G. Bader and E. Gehrke. On the performance of transputer networks for solving linear systems of equation. *Parallel Computing*, vol. 17, no. 12, pp. 1397-1407, 1991.

[4] M. O. Tokhi, M. A. Hossain, M. J. Baxter and P. J. Fleming. Heterogeneous and homogeneous parallel architectures for real-time active vibration control. *IEE Proceedings-D: Control Theory and Applications*, vol. 142, no. 6, pp. 1-8, 1995.

[5] M. O. Tokhi, M. A. Hossain, M. J. Baxter and P. J. Fleming. Performance evaluation issues in real-time parallel signal processing and control. *Journal of Parallel and Distributed Computing*, vol. 42, pp. 67-74, 1997.

[6] B. N. Bershad, D. Lee, T. H. Romer and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, San Jose, California, ACM Press, pp. 158-170, 1994.

[7] B. Clader, C. Krintz, S. John and T. Austin. Cache-concious data placement. *Proceedings of Eighth Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, San Jose, California, ACM SIGARCH SIGOPS SIGPLAN and the IEEE Computer Society, pp. 139-149, 1998.

[8] K. Hwang. Advanced computer architecture – Parallelism, scalability, programmability. McGraw-Hill, USA, 1993.

[9] S. Mcfalring. Program optimization for instruction caches. Proceedings of third Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89), Boston, MA, ACM Press, pp. 183-191, 1989.

[10] M. A. Hossain. Digital signal processing and parallel processing for real-time adaptive noise and vibration control. *Ph.D. thesis*, Dept. of Automatic Control and Sys. Eng., The University of Sheffield, UK, 1995.

[11] P. K. Kourmoulis. Parallel processing in the simulation and control of flexible beam structure system. *PhD. Thesis*. Department of Automatic Control and Systems Engineering, The University of Sheffield, UK. 1990.

[12] G. S. Virk. and P. K. Kourmoulis. On the simulation of systems governed by partial differential equations. *Proceedings of IEE Control-88 Conference*, PP. 318-321, 1988.

[13] M. O. Tokhi and M. A. Hossain. Self-tuning active control of noise and vibration. *Proceedings IEE - Control Conference-94*, vol. 1, 21-24 March, UK, pp. 263-278, 1994.

[14] U. Kabir, M. A. Hossain and M. O. Tokhi. Reducing memory access time in real-time implementation of signal processing and control algorithms. *Pro. of AARTC00: IFAC Workshop on Algorithms and Architectures for Real-time Control*. Palma de Mallorca (Spain), 15-17 May, pp. 15-18, 2000.

Biographies

Alamgir Hossain received his MSc from University of Dhaka (Bangladesh) in 1984 and PhD from University of Sheffield (UK) in 1995. He has worked at several academic institutions and is currently employed as Lecturer at the Department of Computing, The University of Bradford (UK). His main research interests include intelligent control, high-performance computing for real-time signal processing and control (HPC) and network congestion control.

Osman Tokhi obtained his BSc (Electrical Engineering) from Kabul University (Afghanistan) in 1978 and PhD from Heriot-Watt University (UK) in 1988. He has worked at several academic and industrial establishments and is currently employed as Reader at the Department of Automatic Control and Systems Engineering, The University of Sheffield (UK). His main research interests include adaptive/intelligent and soft computing techniques for modelling and control of dynamic systems, high-performance computing for real-time signal processing and control (HPC), and biomedical applications of robotics and control.

Figures and Tables:

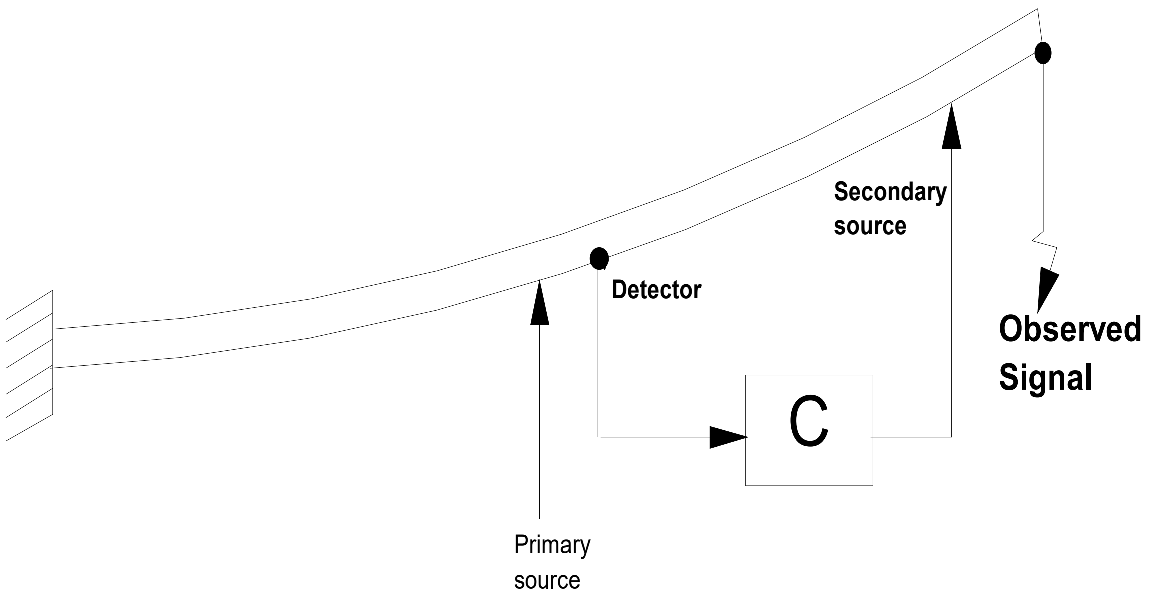


Fig. 1: Active vibration control structure

```
y[8][8] ← –y[8][6] – lumsq*(y[6][7]– 4*y[7][7] +  b*y[8][7] –4*y[9][7]+y[10][7]);
y[16][16]←–y[16][14]–lumsq*(y[14][15]–4*y[15][15]+b*y[16][15]–4*y[17][15]+y[18][15]);
```

Fig. 2: Calculation of deflection of segments 8 and 6 (where, lumsq is lambda square)

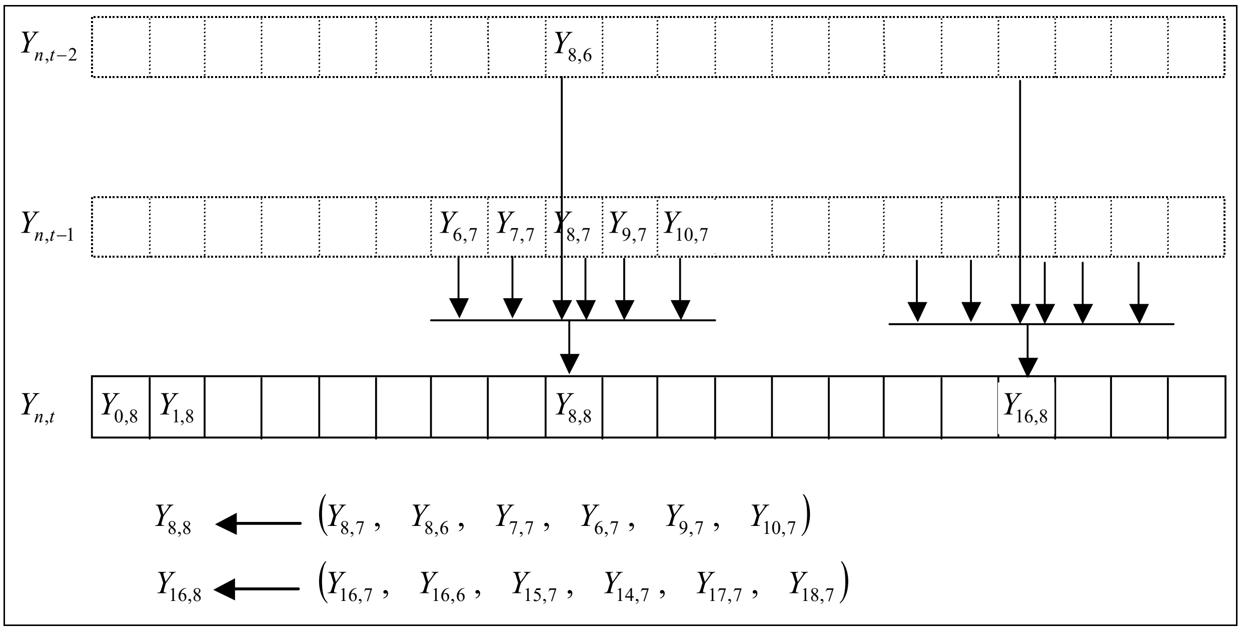

$Y_{n,t-2}$
$Y_{8,6}$
$Y_{n,t-1}$
$Y_{6,7}$ $Y_{7,7}$ $Y_{8,7}$ $Y_{9,7}$ $Y_{10,7}$
$Y_{n,t}$
$Y_{0,8}$ $Y_{1,8}$ $Y_{8,8}$ $Y_{16,8}$
$Y_{8,8} \longleftarrow (Y_{8,7}, Y_{8,6}, Y_{7,7}, Y_{6,7}, Y_{9,7}, Y_{10,7})$
$Y_{16,8} \longleftarrow (Y_{16,7}, Y_{16,6}, Y_{15,7}, Y_{14,7}, Y_{17,7}, Y_{18,7})$

Fig. 3: Data dependencies for computation of deflection of each segment

```
Loop {
//Step 1
  y0[2]=-y0[0]-lamsq*(a*y0[1]-4*y1[1]+y2[1]);
  y1[2]=-y1[0]-lamsq*(-4*y0[1]+b*y1[1]-4*y2[1]+y3[1]);
          :
  y18[2]=-y18[0]-lamsq*(y16[1]-4*y17[1]+c*y18[1]-2*y19[1]);
  y19[2]=-y19[0]-lamsq*(2*y17[1]-4*y18[1]+d*y19[1]);
//Step 2 :  Shifting memory locations
  y0[0]=y0[1]; y0[1]=y0[2]; y1[0]=y1[1]; y1[1]=y1[2];
          :
  y18[0]=y18[1]; y18[1]=y18[2];  y19[0]=y19[1]; y19[1]=y19[2]; }
```

Fig. 4: Design of Algorithm–1

```
Loop {
//Step 1
   y0[2]=-y0[0]-lamsq*(a*y0[1]-4*y1[1]+y2[1]);
   y1[2]=-y1[0]-lamsq*(-4*y0[1]+b*y1[1]-4*y2[1]+y3[1]);
         :
   y18[2]=-y18[0]-lamsq*(y16[1]-4*y17[1]+c*y18[1]-2*y19[1]);
   y19[2]=-y19[0]-lamsq*(2*y17[1]-4*y18[1]+d*y19[1]);
//Step 2
   y0[0]=-y0[1]-lamsq*(a*y0[2]-4*y1[2]+y2[2]);
   y1[0]=-y1[1]-lamsq*(-4*y0[2]+b*y1[2]-4*y2[2]+y3[2]);
         :
   y18[0]=-y18[1]-lamsq*(y16[2]-4*y17[2]+c*y18[2]-2*y19[2]);
   y19[0]=-y19[1]-lamsq*(2*y17[2]-4*y18[2]+d*y19[2]);
 //Step 3
   y0[1]=-y0[2]-lamsq*(a*y0[0]-4*y1[0]+y2[0]);
   y1[1]=-y1[2]-lamsq*(-4*y0[0]+b*y1[0]-4*y2[0]+y3[0]);
            :
   y18[1]=-y18[2]-lamsq*(y16[0]-4*y17[0]+c*y18[0]-2*y19[0]);
   y19[1]=-y19[2]-lamsq*(2*y17[0]-4*y18[0]+d*y19[0]); }
```

Fig. 5: Design of Algorithm–2

```
Loop {
  for(j=0; j<1000; j++) {
   y0[j]=-y0[pj]-lamsq*(a*y0[ppj]-4*y1[ppj]+y2[ppj]);
   y1[j]=-y1[pj]-lamsq*(-4*y0[ppj]+b*y1[ppj]-4*y2[ppj]+y3[ppj]);
            :
   y18[j]=-y18[pj]-lamsq*(y16[ppj]-4*y17[ppj]+c*y18[ppj]-2*y19[ppj]);
   y19[j]=-y19[pj]-lamsq*(2*y17[ppj]-4*y18[ppj]+d*y19[ppj]);
   pj++; ppj++;
  }
  // Shifting memory locations
  y0[0] = y0[998]; y0[1] = y0[999]; y1[0] = y1[998]; y1[1] = y1[999];
         :
  y18[0] = y18[998]; y18[1] = y18[999]; y19[0] = y19[998]; y19[1] = y19[999];}
```

Fig. 6: Design of Algorithm–3

```
Loop {
  y[0][2]=-y[0][0]-lamsq*(a*y[0][1]-4*y[1][1]+y[2][1]);
  y[1][2]=-y[1][0]-lamsq*(-4*y[0][1]+b*y[1][1]-4*y[2][1]+y[3][1]);
  for (i=2; i<18; i++){
   y[i][2]=-y[i][0]-lamsq*(y[i-2][1]-4*y[i-1][1]+b*y[i][1]-4*y[i+1][1]+y[i+2][1]);
   }
  y[18][2]=-y[18][0]-lamsq*(y[16][1]-4*y[17][1]+c*y[18][1]-2*y[19][1]);
  y[19][2]=-y[19][0]-lamsq*(2*y[17][1]-4*y[18][1]+d*y[19][1]);
  // Shifting memory locations
  for (i=0; i<20; i++) {
   y[i][0]=y[i][1]; y[i][1]=y[i][2];  } }
```

Fig. 7: Design of Algorithm–4

```
Loop {
 // Step 1
 y[0][2]=-y[0][0]-lamsq*(a*y[0][1]-4*y[1][1]+y[2][1]);
 y[1][2]=-y[1][0]-lamsq*(-4*y[0][1]+b*y[1][1]-4*y[2][1]+y[3][1]);
 for (i=2; i<18; i++){
   y[i][2]=-y[i][0]-lamsq*(y[i-2][1]-4*y[i-1][1]+b*y[i][1]-4*y[i+1][1]+y[i+2][1]);
    }
 y[18][2]=-y[18][0]-lamsq*(y[16][1]-4*y[17][1]+c*y[18][1]-2*y[19][1]);
 y[19][2]=-y[19][0]-lamsq*(2*y[17][1]-4*y[18][1]+d*y[19][1]);
 // Step 2
 y[0][0]=-y[0][1]-lamsq*(a*y[0][2]-4*y[1][2]+y[2][2]);
 y[1][0]=-y[1][1]-lamsq*(-4*y[0][2]+b*y[1][2]-4*y[2][2]+y[3][2]);
 for (i=2; i<18; i++){
   y[i][0]=-y[i][1]-lamsq*(y[i-2][2]-4*y[i-1][2]+b*y[i][2]-4*y[i+1][2]+y[i+2][2]);
   }
 y[18][0]=-y[18][1]-lamsq*(y[16][2]-4*y[17][2]+c*y[18][2]-2*y[19][2]);
 y[19][0]=-y[19][1]-lamsq*(2*y[17][2]-4*y[18][2]+d*y[19][2]);
 // Step 3
 y[0][1]=-y[0][2]-lamsq*(a*y[0][0]-4*y[1][0]+y[2][0]);
 y[1][1]=-y[1][2]-lamsq*(-4*y[0][0]+b*y[1][0]-4*y[2][0]+y[3][0]);
 for (i=2; i<18; i++){
   y[i][1]=-y[i][2]-lamsq*(y[i-2][0]-4*y[i-1][0]+b*y[i][0]-4*y[i+1][0]+y[i+2][0]);
    }
 y[18][1]=-y[18][2]-lamsq*(y[16][0]-4*y[17][0]+c*y[18][0]-2*y[19][0]);
 y[19][1]=-y[19][2]-lamsq*(2*y[17][0]-4*y[18][0]+d*y[19][0]);}
```

Fig. 8: Design of Algorithm–5

```
Loop {

   // Step 1
   y0[0]=-y0[0]-lamsq*(a*y0[1]-4*y1[1]+y2[1]);
   y1[0]=-y1[0]-lamsq*(-4*y0[1]+b*y1[1]-4*y2[1]+y3[1]);
          :
   y18[0]=-y18[0]-lamsq*(y16[1]-4*y17[1]+c*y18[1]-2*y19[1]);
   y19[0]=-y19[0]-lamsq*(2*y17[1]-4*y18[1]+d*y19[1]);

   // Step 2
   y0[1]=-y0[1]-lamsq*(a*y0[0]-4*y1[0]+y2[0]);
   y1[1]=-y1[1]-lamsq*(-4*y0[0]+b*y1[0]-4*y2[0]+y3[0]);
          :
   y18[1]=-y18[1]-lamsq*(y16[0]-4*y17[0]+c*y18[0]-2*y19[0]);
   y19[1]=-y19[1]-lamsq*(2*y17[0]-4*y18[0]+d*y19[0]);
}
```

Fig. 9: Design of Algorithm–6

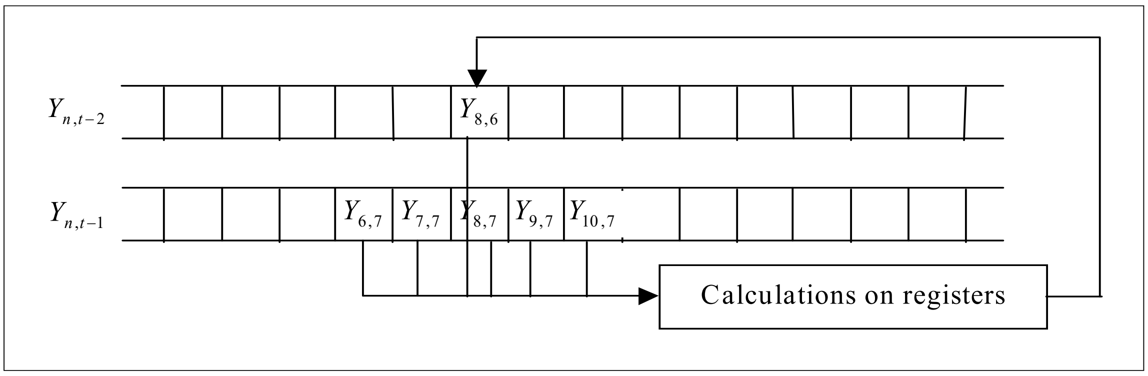
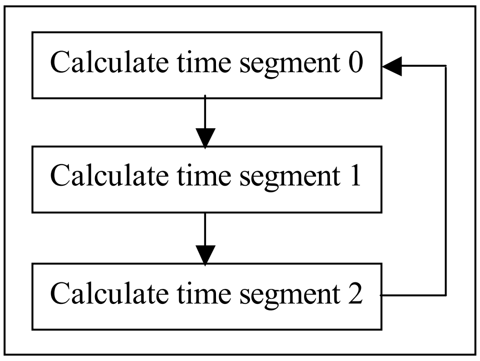


Fig. 10: Re-calculating in 2 time steps



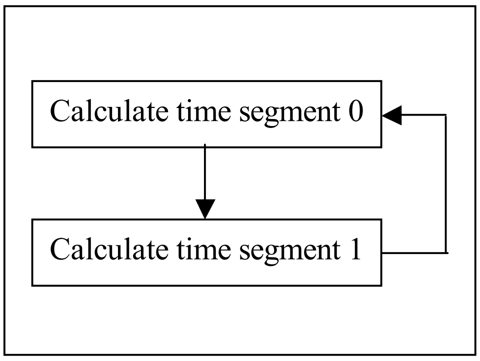


Fig. 11: Block representation of Algorithm–2

Fig. 12: Block representation of Algorithm–6

```
Loop {
   // Step 1
   y[0][0]=-y[0][0]-lamsq*(a*y[0][1]-4*y[1][1]+y[2][1]);
   y[1][0]=-y[1][0]-lamsq*(-4*y[0][1]+b*y[1][1]-4*y[2][1]+y[3][1]);

for (i=2; i<18; i++){
   y[i][0]=-y[i][0]-lamsq*(y[i-2][1]-4*y[i-1][1]+b*y[i][1]-4*y[i+1][1]+y[i+2][1]);
   }
   y[18][0]=-y[18][0]-lamsq*(y[16][1]-4*y[17][1]+c*y[18][1]-2*y[19][1]);
   y[19][0]=-y[19][0]-lamsq*(2*y[17][1]-4*y[18][1]+d*y[19][1]);

   // Step 2
   y[0][1]=-y[0][1]-lamsq*(a*y[0][0]-4*y[1][0]+y[2][0]);
   y[1][1]=-y[1][1]-lamsq*(-4*y[0][0]+b*y[1][0]-4*y[2][0]+y[3][0]);

   for (i=2; i<18; i++){
   y[i][1]=-y[i][1]-lamsq*(y[i-2][0]-4*y[i-1][0]+b*y[i][0]-4*y[i+1][0]+y[i+2][0]);
   }
   y[18][1]=-y[18][1]-lamsq*(y[16][0]-4*y[17][0]+c*y[18][0]-2*y[19][0]);
   y[19][1]=-y[19][1]-lamsq*(2*y[17][0]-4*y[18][0]+d*y[19][0]);
```

Fig. 13: Design of Algorithm–7

```
yc[n]=b0*y12[n] + b1*y12[n-1] + b2*y12[n-2] + b3*y12[n-3]+ b4*y12[n-4]-(a0*yc[n-1]+a1*yc[n-2]
+a2*yc[n-3] +a3*yc[n-4]);

//Shift data array

y12[n-4]=y12[n-3] ; y12[n-3]=y12[n-2] ; y12[n-2]=y12[n-1] ; y12[n-1]=y12[n] ;
yc[n-4]=yc[n-3] ; yc[n-3]=yc[n-2] ; yc[n-2]=yc[n-1] ; yc[n-1]=yc[n] ;
```

Fig. 14: Design outline of the control algorithm (data array shifting method)

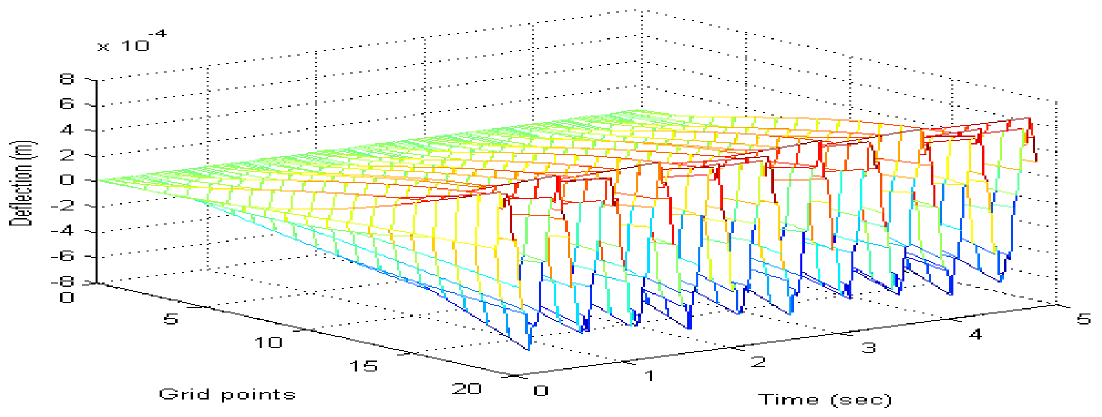


Fig. 15: Fluctuation of the beam along the length before cancellation

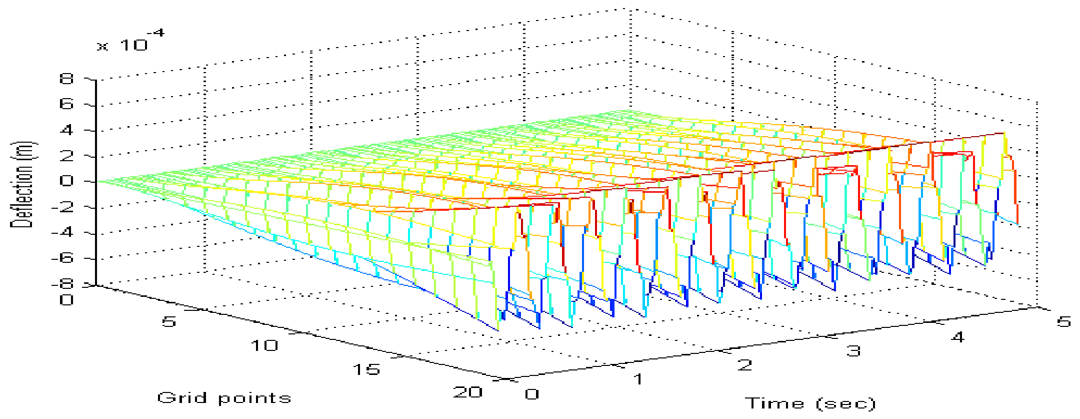


Fig. 16: Fluctuation of the beam along the length after cancellation

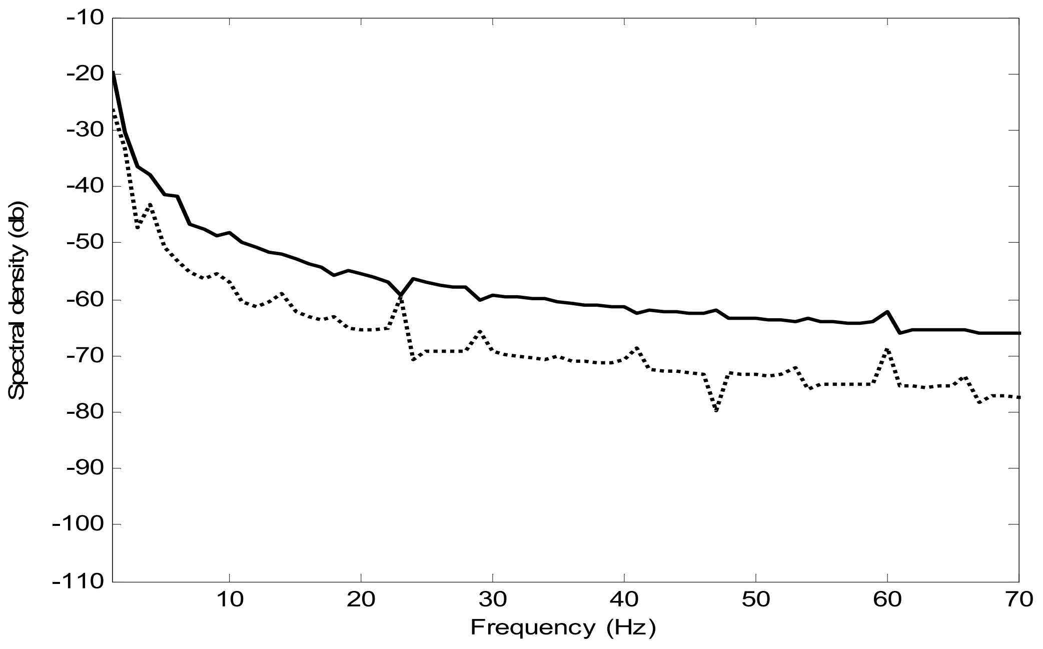
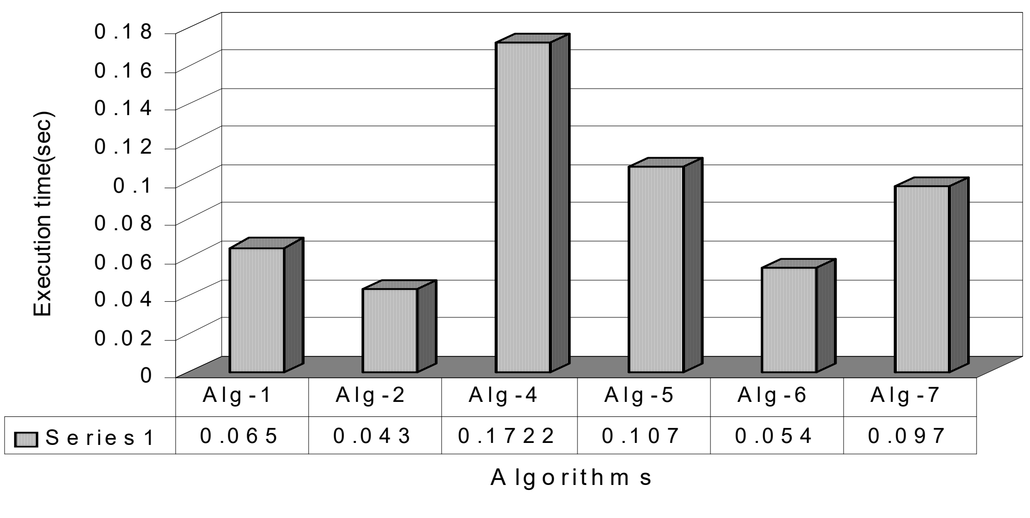


Fig. 17: Auto-power spectral density at the end point before and after cancellation

Table I: Relative performance of the different designs as compared to the Alg-1 ('X' represents 2, 3, 4,---, 7).

Ratio	Alg-1	Alg-2	Alg-3	Alg-4	Alg-5	Alg-6	Alg-7
Alg-X/Alg-1	1	0.67	157	2.46	1.65	0.83	1.48



	Alg-1	Alg-2	Alg-4	Alg-5	Alg-6	Alg-7
▨ Series1	0.065	0.043	0.1722	0.107	0.054	0.097

Fig. 18: Execution time in implementing different algorithms

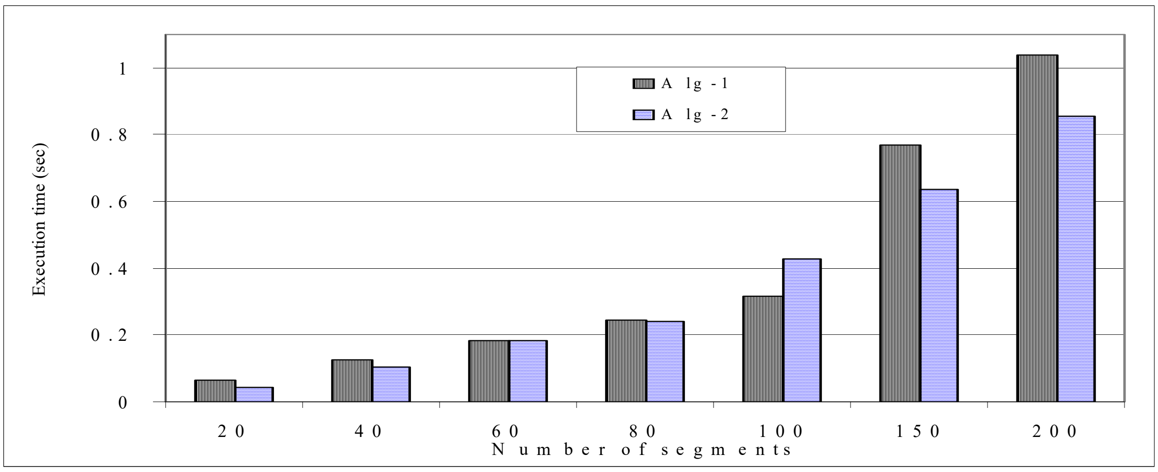


Fig. 19: Performance comparison of Alg–1 and Alg–2

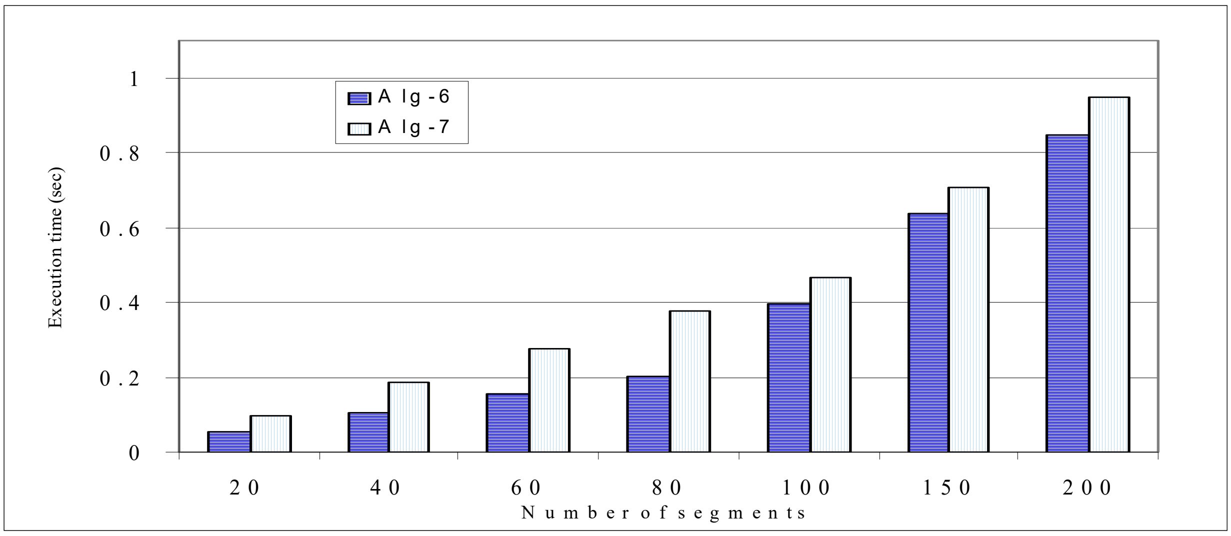


*Fig. 20**:** Performance comparison of Alg–6 and Alg–7*

Table II: Performance of the AVC algorithm designs relative to Alg-1.

No. of Segments	20	40	60	80	100	150	200
A2/A1	0.67	0.83	1.0	1.4	1.6	0.83	0.83
A4/A1	2.46	2.83	2.89	2.92	2.81	1.67	1.63
A5/A1	1.65	1.66	1.77	1.79	1.74	1.10	1.09
A6/A1	0.83	0.83	0.83	0.83	1.3	0.83	0.82
A7/A1	1.48	1.49	1.49	1.54	1.49	0.92	0.91