AIP Conference Proceedings

RESEARCH ARTICLE | JULY 10 2014

GPU acceleration of Runge Kutta-Fehlberg and its comparison with Dormand-Prince method FREE

Wo Mei Seen; R. U. Gobithaasan; Kenjiro T. Miura

AIP Conf. Proc. 1605, 16–21 (2014)
https://doi.org/10.1063/1.4887558

Articles You May Be Interested In

A comparative study of Taylor method, fourth order Runge-Kutta method and Runge-Kutta Fehlberg method to solve ordinary differential equations

AIP Conf. Proc. (March 2024)

Asymptotic solution for heat convection-radiation equation

AIP Conference Proceedings (July 2014)

Variational iteration method for solving sea-air oscillator of the ENSO model

AIP Conference Proceedings (October 2015)

02 December 2024 11:16:42

AIP Publishing

GPU Acceleration of Runge Kutta-Fehlberg and Its Comparison with Dormand-Prince Method

Wo Mei Seen[a], R.U. Gobithaasan[a] and Kenjiro T. Miura[b]

[a]*School of Informatics & Applied Mathematics, University Malaysia Terengganu, 20130 Kuala Terengganu, Terengganu Malaysia*
[b]*Graduate School of Science & Technology, Shizuoka University, Shizuoka Prefecture, Japan*

Abstract. There is a significant reduction of processing time and speedup of performance in computer graphics with the emergence of Graphic Processing Units (GPUs). GPUs have been developed to surpass Central Processing Unit (CPU) in terms of performance and processing speed. This evolution has opened up a new area in computing and researches where highly parallel GPU has been used for non-graphical algorithms. Physical or phenomenal simulations and modelling can be accelerated through General Purpose Graphic Processing Units (GPGPU) and Compute Unified Device Architecture (CUDA) implementations. These phenomena can be represented with mathematical models in the form of Ordinary Differential Equations (ODEs) which encompasses the gist of change rate between independent and dependent variables. ODEs are numerically integrated over time in order to simulate these behaviours. The classical Runge-Kutta (RK) scheme is the common method used to numerically solve ODEs. The Runge Kutta Fehlberg (RKF) scheme has been specially developed to provide an estimate of the principal local truncation error at each step, known as embedding estimate technique. This paper delves into the implementation of RKF scheme for GPU devices and compares its result with Dorman Prince method. A pseudo code is developed to show the implementation in detail. Hence, practitioners will be able to understand the data allocation in GPU, formation of RKF kernels and the flow of data to/from GPU-CPU upon RKF kernel evaluation. The pseudo code is then written in C Language and two ODE models are executed to show the achievable speedup as compared to CPU implementation. The accuracy and efficiency of the proposed implementation method is discussed in the final section of this paper.

Keywords: Parallel Processing, CUDA, Adaptive Runge Kutta Method, Log-aesthetic Curves, Curve Synthesis.
PACS: 43.28.Js, 45.10.Na.

02 December 2024 11:16:42

INTRODUCTION

Since the emergence of Graphic Processing Units (GPUs), there is a significant reduction of processing time and speedup of performance in computer graphics' and rendering. Over the past few years, GPUs have been developed to such a degree that they have surpassed CPU in terms of performance and processing speed. This evolution has opened up a new area in computing and researches where GPU has been used for non-graphical applications especially those that are highly parallel. Similar program executed on many data element parallel, are computationally intensive and have high regular memory-access patterns. We call this as general purpose computing on GPUs (GPGPU) [1].

Thanks to the highly parallel (many core) architecture of modern GPUs [2], the GPU now offers a powerful platform for various scientific and mathematical applications such as high-order numerical integrations [3] and physical based simulations [4]. For this particular purpose, various GPU programming tools have been developed in order to implement and accelerate the mentioned applications. One of this is the Common Unified Device Architecture (CUDA™) which was introduced by NVIDIA Corporation in November 2006. It is a C based programming language which allows users to design programs around the GPU architecture [5].

We have mentioned that either physical or phenomenal simulations or modelling can be accelerated through GPGPU and CUDA implementations. For example, problems such as N-body planetary orbital trajectories [3], Particle Simulation [6], and various other classic Newtonian physics. In order to simulate these behaviours, ordinary differential equations (ODEs) are numerically integrated over time via platforms such as MATLAB™ and Mathematica™ or even programming languages such as C/C++ which provide such implementations [7].

Among many available numerical integrating algorithms, the Runge-Kutta family is perhaps the most widely available method – with a wide variation of orders and schemes. The main objective of this study is to tailor a Runge-Kutta method with adaptive step-size for the GPU device, in this case – NVIDIA GeForce GT540M, simulating multiple trajectories from a system of ODEs while attempting to reduce its overall computation time [7].

Proceedings of the 21st National Symposium on Mathematical Sciences (SKSM21)
AIP Conf. Proc. 1605, 16-21 (2014); doi: 10.1063/1.4887558
© 2014 AIP Publishing LLC 978-0-7354-1241-5/$30.00

CUDA AND GPU'S STRUCTURE

According to [1], GPU is a digital device composed of a co-processor and tightly coupled high speed memory, graphics RAM (GRAM) that is tailored for rasterizing images through a graphics pipeline or in other words, image processing and rendering. As CPUs are not dedicated for highly paralleled computing, image rendering is a mathematically intensive task that, if assigned solely to CPU, it would give a negative impact on the performance. This is due to the architectural difference of the GPU and CPU which result in different nature of their programming models.

While CPU is dedicated for data caching and flow control, the GPU is specialised for compute intensive and highly parallel processes. The same operation is executed on a set of data in parallel which also indicates that there is less memory related operations needed. To achieve that, a GPU is built in such a way that there are more transistors devoted to data processing compared to that of a CPU. The GPU is devoted to data operations with high arithmetic intensity while the CPU is devoted for memory operations [2]. Due to the GPU's architecture and potential for highly parallel operations, many GPU programming models have been developed to harness its computational power.

CUDA™ is a parallel computing platform and programming model invented by NVIDIA Corporation which allows GPUs to execute programs written in various languages including C and C++ [8]. It is an extension for the existing programming languages to allow users to work in a familiar developing environment without having to learn a new programming language and thus, lowering the learning curve of users. It is also designed to support various Application Programming Interfaces, APIs such as MATLAB, OpenGL, and Mathematica 8 [2].

CUDA programming model consists of three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization [2]. To further clarify its architecture, a thread block in the CUDA programming model is actually a set of threads executed in a concurrent manner. A group of 32 threads is known as a warp. Each thread block owns a block ID whereas a grid is an array of thread blocks that execute the same kernel. The C functions are defined by programmers, who read inputs from and write its results to global memory, and synchronize them between dependent kernel calls [8]. Through this unique structure, the programmers are guided to partition a complex problem to coarse sub-problem and further to finer parts that can be executed con-currently by the GPU.

There are a few types of memories in the CUDA structure which are global memory, local memory, shared memory, constant memory, as well as texture and surface memory. The global memory is a read and write memory, the main memory of the GPU device. It is uncached and needs to be aligned naturally (32-, 64-, or 128-byte segments of device memory that are aligned to their size – memory coalescing). Local memory is generally used when the registers do not have enough space to store large structures or arrays or when register spilling happens. Similar to the global memory, it needs to be aligned naturally and is slow. Shared memory is an on-chip memory that is shared between threads and with a bandwidth higher than both global and local memories. Constant memory is the storage for constants and kernel arguments. It is a cached method to access global memory. The texture and surface memory is a read-only memory that resides in GPUs, and cached optimally for 2D spatial access pattern [2].

While there are various GPU programming models such as Brook GPU and Brahma [1] developed by other parties, CUDA holds an advantage for its threading technology that allows higher memory bandwidth and its ability to interact with a wide range of APIs and SDKs [2].

NOTATION OF RUNGE-KUTTA METHODS

Given an initial value problem, some state x at some time t,

$$\frac{dx}{dt} = \mathrm{x}'(t) = f(t, \mathrm{x}(t)), \quad \mathrm{x}(t_0) = \mathrm{x}_0 \,. \tag{1}$$

The numerical solution to the problem via Runge-Kutta Method takes the form:

$$\mathrm{x}_{r+1} = \mathrm{x}_i + h\sum_{i=1}^{S} b_i k_i \,, \quad \mathrm{x}(t_r) = \mathrm{x}_r \,. \tag{2}$$

$$\begin{aligned} k_1 &= f(t, \mathrm{x}(t)) \\ k_i &= f(t_n + c_i h, \mathrm{x} + h\sum_{j=1}^{i-1} a_{i,j} k_j), \quad i = 2,3,4 \end{aligned} \tag{3}$$

Here, r, h and S represent the iteration number, stepsize and the number of stages respectively. The coefficients $a_{i,j}$ and c_i are arranged in a tableau known as the Butcher Tableau (Figure 1).

02 December 2024 11:16:42

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{2,1} & & & & \\
c_3 & a_{3,1} & a_{3,2} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_S & a_{S,1} & a_{S,2} & \ldots & a_{S,S-1} & \\
\hline
 & b_1 & b_2 & \ldots & b_{S-1} & b_S \\
 & \hat{b}_1 & \hat{b}_2 & \ldots & \hat{b}_{S-1} & \hat{b}_S
\end{array}
$$

FIGURE 1. Butcher Tableau: $\hat{b}_i$ represents the coefficient b_i in (2) for the extra stage found in embedded schemes for adaptive stepsize Runge-Kutta methods.

ADAPTIVE STEP-SIZE AND ERROR CONTROL

This research will only focus on two schemes which are the explicit Runge-Kutta Fehlberg of order 4 (RKF (4)5) and Dormand-Prince six-stage 5(4) (DOPRI5) methods. The standard Runge-Kutta equation is provided in the previous section as equations (2) and (3).

While DOPRI5 works the same way as any other standard adaptive Runge-Kutta methods, RKF (4) differs slightly in the number of stages and has less accuracy compared to DOPRI5.

For step size control and error control, which is the core of adaptive step size Runge-Kutta method, we consider the equation:

$$h_{r+1} = 0.84\, h_r \left(\frac{\delta}{\|E_{r+1}\|} \right)^{1/q} \tag{4}$$

In (4), δ is the user-defined maximum allowable local error (tolerance), q is the order of the Runge-Kutta method and $E_{r+1} = \mathrm{x}_{r+1} - \widetilde{\mathrm{x}}_{r+1}$ [9] at r-th iteration. To adjust the step size for each stage, we first compute the values for each stage and corresponding scheme. Continuing from there, we compute the local truncation error, E_{r+1}. If $E_{r+1} > \delta$, then stage calculation for $r+1$ is re-executed with the new h computed using the above equation (4). Otherwise, we continue with the next stage.

CONCEPT OF IMPLEMENTATION

For the general implementation in GPU, the concept is rather similar for all CUDA programs. In this study, an ODE will be considered to be the prototype for experimenting with this implementation. A sequential one threaded CPU code will be generated as a benchmark for comparison in both runtime and accuracy with the GPU implementation. For accuracy comparisons, one of the ways to determine the errors is, by evaluating the root mean squared error, RMSE [7] for the outputs $x_{n,d}, n = 1,2,\ldots,N, d = 1,2,\ldots, X$ generated by each implementation. We will assume that outputs of a double precision run are the canonical "correct" values, $\hat{x}_{n,d}, n = 1,2,\ldots,N, d = 1,2,\ldots, X$, where X is the total number of values of x_n generated. The RMSE is given by:

$$RMSE = \sqrt{\frac{1}{X}\sum_{d=1}^{X}(\hat{x}_{n,d} - x_{n,d})^2} \tag{5}$$

However for systems whose exact solutions are available, we will take the exact solutions' values as $\hat{x}_{n,d}$ to find the *RMSE* for each implementation.

To determine the possible parallelisation, we first need to determine the processes which are independent to each other. In the case of RK integrators, the computation of each argument in the grid functions (3) themselves are independent of each other.

The basic concept of implementation in CUDA is, start with allocating memory space in both CPU and GPU, initialise values in CPU and copy them to GPU. The kernels for stage calculations will first be executed in parallel. Since all the grid functions k_i and truncation errors will be computed concurrently, we will have a total of $N \times X$ E_r at the moment. A maximum reduction is done by first, syncing all the threads to retrieve the all E_r from different threads and "striding" through all the threads in search for the maximum E_r. The x values will be updated if $E_r \le \delta$. If not, no changes will be made to x at the particular iteration. A step size adjustment will be made

02 December 2024 11:16:42

accordingly to prepare the method for recalculating a more accurate x or continue the iteration by calculating the next stage. The steps are depicted Figure 2.

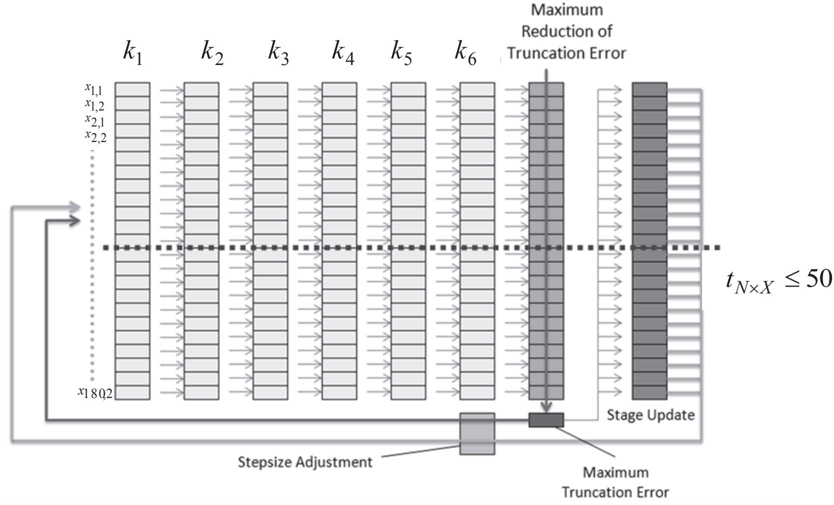

FIGURE 2. GPU implementation of Runge-Kutta Methods.

NUMERICAL EXAMPLE

Simple System of ODE

$$\begin{aligned} \frac{dx_1}{dt} &= -4x_1 + 3x_2 + 6 \\ \frac{dx_2}{dt} &= -2.4x_1 + 1.6x_2 + 3.6 \end{aligned} \tag{6}$$

The ODE (6) is a two dimensional ODE system where x_1 behaves in such a way that it will always converge to the value of 1.5 units regardless of the initial value assigned. The experiment is carried out with h=0.05 and δ=0.0001 as well. The total number of elements generated, $X_{total} = N \times X = 2(90)$ =360. To initialise the variables, $x_n(0) = \beta$ where $n=1,2$, values are taken in an interval of 0.025 starting from 0. The exact solution of (6) is given as below.

$$\begin{aligned} x_1 &= 1.5 + (3.375\beta - 0.375)\mathrm{e}^{-2t} + (1.875 + 0.625\beta)\mathrm{e}^{-4t} \\ x_2 &= (-2.25 + 0.25\,\beta)\,\mathrm{e}^{-2t} + (2.25 + 0.75\,\beta)\,\mathrm{e}^{-0.4t} \end{aligned} \tag{7}$$

Lorenz ’96 Model

The ODE of Lorenz ’96 Model [10] is given below:

$$\frac{dx_n}{dt} = x_{n-1}(x_{n+1} - x_{n-2}) - x_n + F \tag{8}$$

In (8), x_n is the nth component of x and its boundary condition $x_{n-N} \equiv x_n \equiv x_{n+N}$ is cyclic. This is a one dimensional model which studies the chaotic processes in the atmosphere where F is a parameter which induces various behaviors such as decay and periodicity [7].

In the experiment, the dimension of the system, N is set to be 4 and 90 systems, X=90 are simulated concurrently in the experiment, with $F = 1$, $h = 0.05$ and $\delta = 0.0001$. Thus, the total number of elements generated, $X_{total} = 360$. To initialise the variables, $x_n(0)$ where $n = 1,2,3,4$, values are taken in an interval of 0.00001 starting from 0.

Implementations are done on both CPU and GPU. All simulations are processed in double precision. The results are discussed in the following sections.

02 December 2024 11:16:42

RESULTS

Accuracy of implementation for system (6) analysed by the RMSE is given in Table 1.

TABLE (1). RMSE comparison between the implementations for system (6) and the exact solution.

Benchmark	**CPU**		**GPU**	
	x_1	x_2	x_1	x_2
RKF	0	3.57847×10^{-12}	0	6.90017×10^{-11}
DOPRI5	0	2.28177×10^{-13}	0	2.28258×10^{-13}

Based on the result above, the GPU implementation of RKF solver for system (6) performed quite well in terms of accuracy. It can be note that the DOPRI5 GPU Implementation in the mentioned system performed nearly as good as its CPU implementation.

For the implementation of Lorenz'96 Model (8), the result is very accurate where the RMSE computed were all 0. This shows that accuracy does not put GPGPU into disadvantage.

This section discusses the performance enhancement in terms of program execution speed ups. Figure 3 and Figure 4 illustrate the speed ups achieved by different implementations.

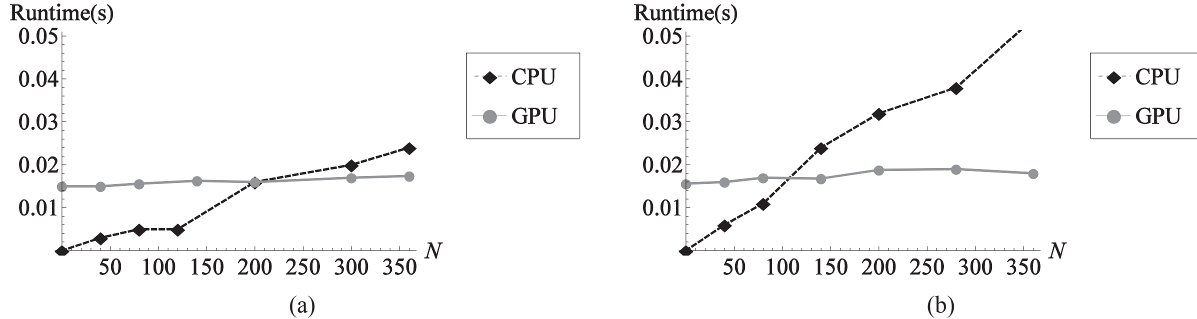

FIGURE 3. RKF Runtime Comparison: (a) ODE system (6). (b) RKF-Lorenz'96 Model (7).

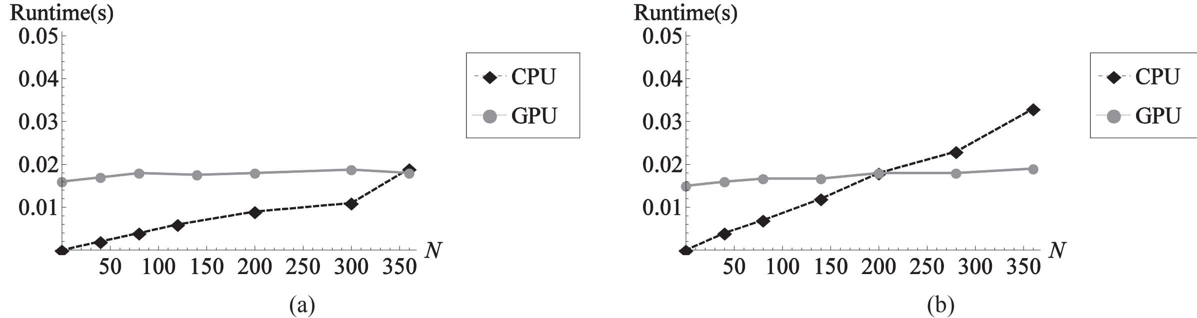

FIGURE 4. DOPRI5 Runtime Comparison: (a) ODE system (6). (b) RKF-Lorenz'96 Model (7).

All of the above figures proof that GPU implementation does improve performance in terms of optimising runtime. All the GPU implementations execute nearly as fast as each other. However, it can be note that DOPRI5 has a lower computation time in CPU compared to RKF. This is due to the fact that DOPRI5 has FSAL (first same as last) property in which the last stage is evaluated at the same point as the first stage of the next step. This is due to the efficiency of DOPRI5 being able to compute global truncation error of $O(h^5)$ compared to RKF with global truncation error of $O(h^4)$. The performance of GPU is able to overcome that of CPU at some point when the problem dimension or size, *N* is large enough.

It is observed that different ODE systems will produce different results in terms of computation speed up. It is due to the difference in truncation error produced by the ODE systems in each stage. From the result produced in the stated experiment, it can be seen that the ODE system (6) has lower computation speed compared to Lorenz'96 model. This implies that ODE system (6) produces lower truncation error at each stage which leads to faster

02 December 2024 11:16:42

computation time. However, it can be influenced by other factors such as algorithm efficiency and CPU's compute efficiency.

Due to the limitation of memory space and cores of GPU used in this research, the maximum dimension of the problem to fully occupy the GPU is 360. For sizes greater than that, the kernels will not be executed as it will be detected as an error by the GPU for exceeding the overall memory capacity. This can be solved by using RK methods which is optimised for minimum memory storage such as Kennedy's RK4 (3)5[2R+] scheme.

CONCLUSION & FUTURE WORK

This research is conducted with the purpose of studying the concept and implementing RK integrators in the GPU using CUDA programming model. This research was completed entirely using C programming language of Microsoft Visual Studio 2010. The speed up gained is noticeable for each implementation. The accuracy achieved by both the ODE systems is highly desirable. Thus, it can be concluded that accuracy is not a problem in GPGPU. However, result may vary as ODE systems vary.

The next feasible step is to apply the proposed method to render Log-Aesthetic curve (LAC) after representing it in the form of ODE [11]. LAC is the emerging curve segment to render aesthetic shapes and Gaussian-Konrod method was used to render in [12] which proven to be slow. Hence, the representation of LAC in the form of ODE and RK method to render may lead to a tremendous speedup. A noticeable impact can be experienced when one renders a piecewise LAC similar to techniques stated in [13, 14], using LAC as a fairing method [15] and extending LACs to form aesthetic surfaces [16].

ACKNOWLEDGMENTS

The authors acknowledge University Malaysia Terengganu and Ministry of Education Malaysia (FRGS:59265) for providing financial aid to carry out this research. We would like to acknowledge the reviewer of this paper for his/her helpful comments.

02 December 2024 11:16:42

REFERENCES

1. S. K. Boggan and D. M. Pressel, *GPUs: An Emerging Platform for General-Purpose Computation No. ARL-SR-154,* ARMY RESEARCH LAB ABERDEEN PROVING GROUND MD COMPUTATIONAL AND INFORMATION SCIENCES DIR, 2007.
2. NVIDIA Corporation, *CUDA C Programming Guide Version 4.2*, 2012. {http://www.math.umass.edu/~johnston/CUDA_WG _2012/ CUDA _C_Programming_Guide.pdf}
3. K. Hawick, D. Playne and M. Johnson, *Proc. Int. Conf. on Computer Design No. CDE4469*, 2011.
4. T. Oberhuber, A. Suzuki, J. Vacata and J. V. Žabka, *J. Math-for-Industry* **3**, 73-79 (2011).
5. I. Buck, High Performance Computing with CUDA, 2008. {http://mc.stanford.edu /cgi-bin/images/b/ba/M02_2.pdf}
6. S. Green, "Particle Simulation using CUDA" in *NVIDIA Whitepaper*, 2010.
7. L. Murray, *IEEE Transactions on Parallel and Distributed Systems* **23**, 94-101 (2012).
8. NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Fermi" in *NVIDIA Whitepaper*, 2009. {http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf}
9. J. R. Dormand and P. J. Prince, *J. Computational and Appl. Maths.* **6**, 19-26 (1980).
10. E. N. Lorenz, *Proc. Seminar on predictability* **1**, 40-58 (1996).
11. Y. M. Teh, R. U. Gobithaasan, A. R. M. Piah and K. T. Miura, "Rendering Log Aesthetic Curves via Runge-Kutta Method" in *AIP Proc.* (in press).
12. N. Yoshida and T. Saito, *Visual Computers* **22**, 896-905 (2006).
13. R. U. Gobithaasan and J. M. Ali, *Int. Conf. on Computer Graphics, Imaging and Visualization*, 2004, 109.
14. K. T. Miura, D. Shibuya, R. U. Gobithaasan, and S. Usuki, *Computer-Aided Design and Applications* **10**, 1021-1032 (2013).
15. K. T. Miura, R. Shirahata, S. Agari, S. Usuki and R. U. Gobithaasan, *Computer-Aided Design and Applications* **9**, 901-914 (2012).
16. K.T. Miura and R.U. Gobithaasan, "Aesthetic curves and surfaces in Computer Aided Geometric Design" in *Int. J. Automation Technology* (in press).