# UC Irvine
## ICS Technical Reports

### Title
The effect of FPU architecture on a dynamic precision algorithm for the solution of differential equations

### Permalink
https://escholarship.org/uc/item/0qv5h4nb

### Authors
Kramer, David
Scherson, Isaac D.

### Publication Date
1991-11-05

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 91-73

# The Effect of FPU Architecture on a Dynamic Precision Algorithm for the Solution of Differential Equations

David Kramer
Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08544

Isaac D. Scherson
Department of Information and Computer Science
University of California
Irvine, California 92717

**Technical Report #91–73**

November 5, 1991

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

# The Effect of FPU Architecture on a Dynamic Precision Algorithm for the Solution of Differential Equations

David Kramer
Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08544
(714) 865-7713
kramer@ics.uci.edu

Isaac D. Scherson
Department of Information and Computer Science
University of California
Irvine, California 92717
(714) 856-8144
isaac@ics.uci.edu

**Abstract**

Solution of Initial Value Problems (IVPs) is an important application in scientific computing. Methods for solving these problems use techniques for reducing the error and increasing the speed of the computation. This paper introduces a class of algorithms which dynamically reconfigure their operating parameters to reduce the computation time. By dynamically varying the precision of the arithmetic being performed, it is possible to obtain dramatic speedups on certain architectures when solving IVPs. This paper illustrates how various architectures impact on a dynamic precision version of the Runge-Kutta-Fehlberg algorithm. It is shown that a speedup of over 30 percent is possible for both massively parallel processors and vector supercomputers.

**Keywords:** Computer Arithmetic, Floating Point, ALU architecture, Initial Value Problems, Precision, ODE Solver.

*This research was supported in part by the National Science Foundation under grant number MIP 9106949

# 1 Introduction

The effect of architecture dependent parameters on the solution of numerically intensive codes is often overlooked. Algorithms which are theoretically sound can be dramatically affected by their practical implementation on physical machines. Kahan [12] has shown that even apparently innocuous code can produce widely diverging results depending on the nature of the machine on which it is run. Architectural features which can affect the result of computations include: the precision of the operands, the nature of the arithmetic being performed, and the way the data is stored.

Theoretical analysis of the effect of these parameters on complex computations is often difficult, if not impossible. The traditional response to this whenever possible has been to implement codes which display an inherent stability to spurious errors introduced by the computation. Furthermore, it is often prudent to leave a large margin of safety in the implementation of the code. For example, one might use the largest precision available on the machine, in order to minimize the effect of truncation or rounding errors. This conservative approach improves the confidence of the user in the results obtained. However, there is a penalty paid in terms of machine resources for using this approach. Conservative codes require more storage space, as well as greater execution time. This penalty may be significant, as is shown below.

Many Arithmetic Logic Units (ALUs) are implemented such that all floating point operations are executed in extended precision arithmetic, irrespective of the precision of the operands. The time taken to execute a floating point operation on such ALUs is relatively independent of the precision of the operands (see figures 1a and 1b). Examples of machines of this type include workstations and some mainframes. For conciseness we refer to these architectures as *fixed range* architectures. Other machines implement their ALUs such that they perform the arithmetic on the precision of the operands, with perhaps one or more guard bits and a sticky bit (see figures 1c and 1d). In these architectures the time taken to implement floating point operations may be proportional to the size of the operands, or even proportional to the square of size of the operands. Examples of machines of this type include supercomputers of both the vector processing and massively parallel type. We refer to machines of this type as *multiple range* architectures. It is interesting to note that we classify massively parallel and vector supercomputers together, in spite of their being seemingly competing paradigms. These architectures are all optimized for high performance. Every

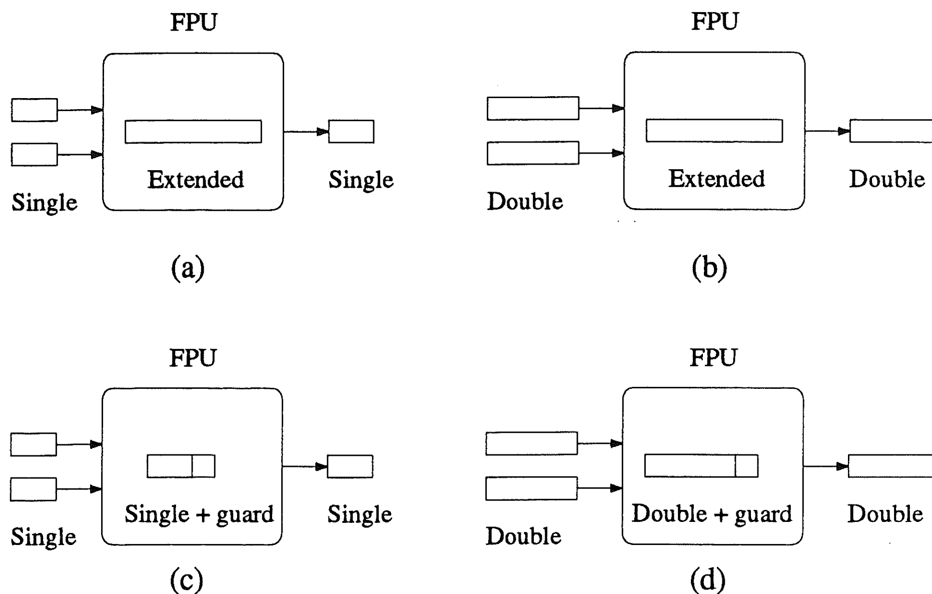


Figure 1: FPU precision characteristics, for fixed range architectures (a and b) and multiple range architectures (c and d)

technological advantage is pressed as possible. For this reason, the precision of the Floating Point Unit (FPU) is kept to the minimum required. They have widely different arithmetic execution times for different precisions. We show how it is possible to exploit this differential in order to gain a significant speedup in executing several important algorithms.

This paper introduces a class of algorithms dubbed *dynamic precision* algorithms. Dynamic precision algorithms monitor their own performance and modify the precision, or number of bits of the operands, as they proceed. In this way it is possible to retain the inherent conservatism of longer precision arithmetic when necessary. However, they use smaller precision arithmetic when possible. The benefit being a speedup due to reduced arithmetic computation time for the reduced precision These dynamic precision algorithms will be introduced here by means of an example.

One of the most complex applications that users expect to solve with application library codes is the solution of Ordinary Differential Equations. Moreover, monitoring of the frequency of use of numerical libraries has shown that it is also one of the most commonly used techniques [15]. We believe that dynamic precision techniques can be incorporated into a wide range of ODE solvers. In order to illustrate this point we present an example of a dynamic precision ODE solver.

A common method for solving non-stiff ODEs is the class of algorithms known as the Runge-Kutta (RK) methods. These methods solve a system of first order ODEs of the form

$\overline{Y(t)} = f'(t, \overline{Y})$;  $\alpha \leq t \leq \beta$.

The vector $\overline{Y(t)}$ is known as the *state* of the system.

If the initial values of $\overline{Y}$ are available; $Y(\alpha) = \overline{K}$, then the problem is known as an Initial Value Problem (IVP). If the values of $\overline{Y}$ are known for some other time $\gamma$, then the method is known as a Boundary Value Problem. Fehlberg [5] introduced a variation of RK methods which moderate the local error introduced at each step by modifying the steplength. These are known as RKF methods. RKF methods keep the error introduced at each step to within a certain tolerance with the intention of bounding the global error. While this does not necessarily occur, the method generally yields good results.

This paper introduces an algorithm, dubbed the Runge-Kutta-Fehlberg-Kramer (RKFK) algorithm, which uses the estimate of the error produced at each step, to not only determine the size of the next step, but also the precision of the arithmetic operations taken on that step. At each step the code is then run on the smallest precision that will yield a result within the given error tolerance. Because on many architectures, the time taken to implement floating point operations is proportional to the precision of the operands, by using the minimal precision without impacting the error, the execution time is minimized.

An analysis of the source of errors in the solution of RK methods is presented in Section 2. The dynamic precision RKFK algorithm is presented in Section 3. The effect of precision on performance in fixed and multiple range architectures is contrasted in Section 4. The performance of the RKFK algorithm is benchmarked against RKF algorithms for a system of ODEs of scientific interest, namely the N-body problem. Given a user specified error tolerance, it is shown that the RKFK algorithm offers performance comparable to the RKF method with the shortest precision necessary to solve the ODE. The speedup of the RKFK algorithm over single precision RKF in fixed range architectures is typically low, and is shown to be approximately 5 to 10 percent. In multiple range architectures the speedup is shown to lie in the 30 per cent range.

It must be stressed that while the RKFK algorithm is an instance of a dynamic precision algorithm, it is by no means the only such application. The ideas presented in this paper can be applied to a wide range of algorithms, both for solving ODEs, as well as other numerically intensive

applications where numerical errors are significant.

## 2 Solution of ODEs and Error Propagation

This section introduces the Runge-Kutta-Fehlberg algorithm and illustrates the potential source of errors in the solution of ODEs using this method.

The RKF class of algorithms can be presented as follows. Given a system of first order ODEs:

$$\overline{Y(t)} = f'(t, \overline{Y})$$

compute the value of $\overline{Y(t+h)}$. This value is computed using two different orders of the Runge-Kutta method:

$$\overline{Y(t+h)} = \overline{Y(t)} + \sum_{i=1}^{p} a_i k_i$$

where the $a_i$s are constants, $h$ is the steplength and $k_i = hf(t, h, k_1, \ldots, k_{i-1})$. $p$ is known as the *order* of the method.

The Fehlberg embedding of the fifth order has the advantage that with only six evaluations of the function $f$, both a fourth and a fifth order solution for $\overline{Y}$ can be computed. The error in the step is computed as the difference in the solutions of the two orders. It is the magnitude of this local error that is used to determine the steplength of the following step. In regions where the solution is varying rapidly this error will be large and small steps will be required. In regions where the solution is fairly stable larger steps can be taken.

This brings us to the source of errors in this type of computation. There are two sources of errors, namely the *discretization error* and the *roundoff error*.

The discretization error is a property of the method used to solve the ODEs. It has been shown that the magnitude of this error is $O(h^{p+1})$ [6]. This error can be reduced by reducing the steplength $h$, or increasing the order of the method $p$. The disadvantage of both of these techniques is that more function evaluations are required and the execution time per step increases.

Roundoff error is introduced by the actual computation, rather than the algorithm. It consists of several components, including the errors in quantizing the data, and the errors introduced by performing the arithmetic. A thorough analysis of the behavior of roundoff errors in the solution

of ODEs has been presented by Henrici [8, 9]. He shows that the size of this error can grow exponentially, and is a function of the precision used. Roundoff error increases with the number of steps taken. Hence a decrease in the steplength can actually result in an increase in the error, because more steps are required.

RKF methods do not discriminate between these two types of errors introduced into the solution of the ODEs. On the other hand, the algorithm introduced below performs a trade-off between these two errors in order to improve performance.

# 3 The Runge-Kutta-Fehlberg-Kramer Algorithm

Typical RKF codes for the solution of IVPs require several parameters [2, 11]. One of these is the error tolerance which specifies the maximum allowable local error at each step. Generally as this tolerance becomes tighter, smaller steplengths are required to reach a solution. Another parameter that is required is the maximum steplength. This parameter may be required for several reasons. Firstly the solution of the problem may have some periodic frequencies. Shannons sampling theorem dictates that the solution must be sampled at more than twice the fastest frequency if an aliasing type error is to be avoided. This phenomenon is illustrated by Shampine [15] where he cites a case in which a biological system had a periodic component with a period of one day. The solver sampled the solution one day after the initial conditions and found that it had not changed. It increased the steplength by a multiple of the first, i.e. several days, and the same phenomenon occured. It did not detect the periodic nature of the solution at all. In this case the maximum steplength should have been limited to less than half a day. Other reasons for specifying a maximum steplength include the so called *dense output* [7] case. The user may require output at certain intervals and if these intervals are shorter than the steplength required by the algorithm, they constitute a *de-facto* maximum steplength. Finally, Lenferink and Spijker [13] show that by limiting the steplength, the rate of error growth in the global solution can be controlled.

At a specific step, the error introduced into the solution will be a sum of the discretization and roundoff errors. The discretization error is dependent on the order of the method used. Several dynamic order algorithms have emerged recently [4], which exploit different orders of algorithms in different regions. These algorithms increase the order of the method in regions of instability,

enabling reasonably large steps to be taken in these regions. These techniques do not impact on the algorithm presented here. Both dynamic precision and dynamic order algorithms could be included into a single ODE solver.

The size of the roundoff error is not necessarily a constant overhead, as can be seen from Henrici's analysis [9]. Let us assume that the system is using a precision $\rho$. The size of the roundoff error introduced is proportional to the magnitude of the least significant bit of $\rho$, and hence is inversely proportional to $2^{\phi(\rho)}$, where $\phi(\rho)$ is the number of bits in the mantissa of $\rho$. We can decrease the roundoff error introduced at any step by increasing the precision of the arithmetic being performed. The converse is also true. Increasing the precision results in larger steps and presumably faster code. On the other hand, the advantage of a decrease in precision is that the execution time of the individual function evaluations will be reduced. It seems reasonable to assume that there is a precision $\rho$ which minimizes the execution time by combining function evaluation time with number of function evaluations. Unfortunately most contemporary architectures and compilers do not give the user a continuous range of possible precisions. The user typically has two or three floating point precisions available. It is desirable to select the one which will provide the fastest running time for a given error tolerance. This leads us to the RKFK algorithm.

The largest value of $\phi(\rho)$ available will generally yield the largest steplengths. If the steplength demanded by the algorithm running at this precision is greater than the user specified maximum steplength, we say that the precision is *steplength saturated.* It is reasonable to assume that the discretization error will be less than the tolerance if the system enters this state when using a certain precision. We can then tolerate a somewhat larger roundoff error and still maintain the maximum steplength. A somewhat smaller precision $\rho'$ is used to do the computation. This process can be repeated for the range of precisions available on the architecture. In regions where the solution is varying rapidly the solver should use a large precision, and vice-versa for regions of stability. Naturally the significance of the final results produced by the solver is the significance of the smallest value of $\phi(\rho)$ used by the algorithm. Pseudocode for the RKFK algorithm is presented in Appendix A.

# 4 Experimental Results

This section presents the results of a performance evaluation of an implementation of the RKFK algorithm. This implementation uses two different values of $\phi(\rho)$. Firstly single precision (32 bit) arithmetic is performed, with $\phi(\rho_1) = 24$. The second value of $\rho$ used is double precision floating point which has $\phi(\rho_2) = 54$ bits. The performance of the dynamic precision code *RKFK* is compared with two other static precision codes, called *RS* and *RD*, with precisions of $\rho_1$ and $\rho_2$ respectively. In all three cases the underlying algorithm implemented was the 4-5 embedding of RKF presented in [2]. It should be emphasized that although only two different values of $\rho$ are used in this implementation, more values of $\rho$ can be used if the architecture permits. Furthermore, other techniques for speeding up IVP solvers, such as Nystrom embeddings [1] or block solvers [3] can be incorporated without affecting the dynamic precision nature of the algorithm.

## 4.1 Experimental Methodology

The specific problem investigated was the N-body problem. This problem specifies the motion of N bodies under their mutual gravitational attraction. The state equations for this system are given in Appendix B. In order to illustrate different aspects of performance of the three solvers, two cases are considered. The first problem considered here is the simulation of the orbits of the four gas giant planets of the solar system and the sun. This case was chosen because it is of scientific interest and presents a realistic application of reasonable computational intensity.

We firstly wish to illustrate the performance of the codes on a system whose state variables vary at a relatively constant rate. We dub a system which has a relatively constant rate of change, such as this one, a *steady system.* The rate of variation can range from very slow to very fast. A slowly varying system is defined as one whose steplength is large relative to the maximum steplength, for a given tolerance. The converse is true for a slowly varying system. We evaluate the performance of the solvers over the full range of systems. For purposes of illustration we have set the maximum steplength at an interval of 604800 seconds or one week. The tolerance is then varied from the point where both single and double precisions are steplength saturated to the point where neither are.

Another type of system which can arise is one which sometimes varies slowly and sometimes

varies rapidly. This system is investigated by introducing a short period comet into the solar system described above. This comet will travel rapidly at times, exciting the system, but at other times will travel slowly and its effect will not be noticeable. We dub such systems, which have a varying rate of change of the system state variables, an *unsteady system.*

The performance of the three codes was evaluated on two separate architectures, one of each of the fixed and multiple ranges described above. These results are presented below.

## 4.2 Fixed Range Architectures

We first consider the class of architectures which utilize a monolithic ALU and perform floating point arithmetic on a single, typically extended, precision. If a lower precision is required, the extended precision result is chopped or rounded to the required precision. The time taken to implement floating point operations is thus independent of the precision of the operands. Architectures which fall into this class are microprocessor-based engineering workstations and personal computers as well as certain mainframes. This class also includes some parallel processors which utilize a microprocessor-based architecture for each processing element, such as the Intel Touchstone. The experiments reported in this section were implemented on a SPARCstation 2 workstation produced by SUN Microsystems.

The results presented in table 1 illustrate the ratios of the running times for the three algorithms on a steady system. In table 1 $t_k$ is the running time of the RKFK algorithm, $t_d$ is the running time of RD and $t_s$ is the running time of RS. The units of the tolerance are $m/s$ or $m$, depending on whether the equation of motion it is applied to is one specifying position or velocity. It should be noted that the scale of the solar system is of the order $10^{12}$ meters. The relative scale of the tolerance to the data in this case is $O(10^{12}/tolerance)$. The algorithms are evaluated over five or more orders of magnitude in the tolerance. The number of function evaluations required by each of the solvers is also presented.

Examination of the data in table 1 reveals several trends. Firstly, it is immediately apparent that the performance of the single precision solver, RS, degenerates rapidly when the tolerance is less than a certain level (a *tight* tolerance). This is an example of a system which varies rapidly. In these cases the RKFK algorithm chooses to do arithmetic in double precision. The resulting execution time is approximately equal to that of the double precision solver, RD. There is a small

Table 1: Relative run times of RKFK, RD and RS in a Steady System, Fixed Range Architecture

| | Relative Run Times | | | Function Evaluations | | |
|---|---|---|---|---|---|---|
| Tol | $t_d/t_k$ | $t_s/t_k$ | $t_s/t_d$ | RKFK | RD | RS |
| 0.1 | 1.00 | $> 15$ | $> 15$ | 35772 | 35772 | - |
| 1 | 1.01 | $> 15$ | $> 15$ | 31332 | 31308 | - |
| 5 | 1.01 | 12.83 | 12.76 | 31332 | 31308 | 428586 |
| 10 | 1.01 | 6.37 | 6.35 | 31326 | 31308 | 212454 |
| 50 | 1.00 | 1.25 | 1.25 | 31380 | 31308 | 41970 |
| 100 | 1.05 | 0.99 | 0.95 | 31512 | 31308 | 31626 |
| 500 | 1.06 | 0.99 | 0.99 | 31308 | 31308 | 31308 |
| 1000 | 1.07 | 0.99 | 0.93 | 31308 | 31308 | 31308 |

overhead incurred by RKFK in choosing the appropriate precision. This overhead typically amounts to less than one percent of the total time. Towards the bottom of table 1 it can be seen that the relative running times of RS and RD are reversed when the tolerance is *loose*. It is apparent that RS runs approximately 7% faster than RD. Furthermore we see that the RKFK algorithm is now running in single precision and benefits from the resultant speedup. Between these two regions is a region of transition, with $Tolerance \approx 10$, where RKFK switches between the two precisions. Over the total range of tolerances, the number of function evaluations required by the double precision solver is less than that required by the single, as is expected. The three solvers all require the same number of evaluations when they are steplength saturated at a tolerance of approximately 500.

It is noticeable that for relatively loose tolerances RS is about 7% faster than RD. As we have stated, this can not be due to reduced arithmetic computation time. We believe that this speedup is due to reduced bus and memory cycles required by single precision operands.

Next we consider the results of solving an unsteady system, shown in table 2. We see similar trends as above, with the single precision solver performing poorly for tight tolerances, and outperforming the double precision solver by about 7% for the loose tolerances.

In conclusion it can be seen that for fixed range architectures the performance of the single precision algorithm may be significantly worse than for the double. On the other hand, when the tolerance is loose enough (or the maximum steplength small enough) the single precision algorithm may outperform the double by 5 to 10 percent. One may well decide that the speedup of 5 to 10 percent may not outweigh the risk associated with using single precision code. It can then be

Table 2: Relative run times of RKFK, RD and RS in an Unsteady System, Fixed Range Architecture

| | Relative Run Times | | | Function Evaluations | | |
|---|---|---|---|---|---|---|
| Tol | $t_d/t_k$ | $t_s/t_k$ | $t_s/t_d$ | RKFK | RD | RS |
| 0.1 | 1.00 | $> 10$ | $> 10$ | 163806 | 163806 | - |
| 1 | 1.00 | $> 10$ | $> 10$ | 104148 | 104100 | - |
| 5 | 1.00 | 7.18 | 7.17 | 77934 | 77904 | 593514 |
| 10 | 1.00 | 4.06 | 4.06 | 69672 | 69648 | 299940 |
| 50 | 1.00 | 1.26 | 1.26 | 55596 | 55560 | 74592 |
| 100 | 1.01 | 1.03 | 1.01 | 51348 | 51162 | 55002 |
| 500 | 1.03 | 0.97 | 0.94 | 43686 | 43680 | 43800 |
| 1000 | 1.03 | 0.97 | 0.94 | 41358 | 41352 | 41412 |
| 10000 | 1.04 | 0.98 | 0.94 | 36180 | 36180 | 36186 |

concluded that on fixed range architectures , it is advisable to use the largest precision available. However by utilizing the dynamic precision code, one never does any worse than the double precision case. In the region of transition from double to single precision, the RKFK code performs marginally worse than the single precision code, but still better than the double precision code. Without a-priori knowledge of the behavior of the state of the system, the user may well not want to use the single precision code and hence the dynamic precision code should be applied.

### 4.3 Multiple Range Systems

We define multiple range architectures to be those which internally perform arithmetic only with the precision of the operands. Multiple Range architectures include most pipelined vector type supercomputers [10] as well as many massively parallel machines. They have the property that the time taken to perform arithmetic operations is proportional to the precision (or possibly the square of the precision, for bit-serial ALUs). In order to examine the performance of RKFK on multiple range architectures, the above experiments were implemented on a Maspar MP-1 massively parallel computer. It should be noted that all the processing elements in the array were operating on identical data and produced the same results, due to the SIMD nature of the machine. However it was not the results that were of interest (outside of verifying the correctness of the solution) but the relative computation times of the three algorithms on this architecture. In this context it is not meaningful to compare the relative computation times of the above fixed range architecture with

Table 3: Timing of Selected Floating Point Operations on Maspar MP-1 (Clock Cycles)

| Precision | Add/Sub | Multiply | Divide |
|---|---|---|---|
| single | 120 | 240 | 300 |
| double | 180 | 530 | 1000 |

Table 4: Relative run times of RKFK, RD and RS in an Steady System, Multiple Range Architecture

| | Relative Run Times | | | Function Evaluations | | |
|---|---|---|---|---|---|---|
| Tol | $t_d/t_k$ | $t_s/t_k$ | $t_s/t_d$ | RKFK | RD | RS |
| 0.1 | 1.00 | $> 15$ | $> 15$ | 1044 | 1044 | - |
| 1 | 0.98 | $> 15$ | $> 15$ | 930 | 912 | - |
| 5 | 0.97 | 10.7 | 11.0 | 948 | 912 | 15102 |
| 10 | 0.98 | 5.42 | 5.53 | 936 | 912 | 7590 |
| 50 | 1.00 | 1.05 | 1.05 | 936 | 912 | 1440 |
| 100 | 1.27 | 0.86 | 0.68 | 930 | 912 | 924 |
| 500 | 1.47 | 0.98 | 0.67 | 912 | 912 | 912 |
| 1000 | 1.47 | 0.98 | 0.67 | 912 | 912 | 912 |

this multiple range architecture.

The timings of some operations on the Maspar MP-1 array are shown in table 3 [14]. It is apparent that single precision operations are 30 to 60 percent faster than double precision operations. We would expect to see this speedup in the relative performance of the three ODE solvers. It should also be noted that in an architecture like the MP-1 interprocessor communication is performed bit-serially. The time to perform communication is proportional to the precision of the operands. While this particular application used no interprocessor communications, we would expect a commensurate speedup in applications that did require communication.

Table 4 lists the relative timings of the three ODE solvers for the steady system described above. Once again we see the performance of the single precision solver degrade when the tolerance is below a certain threshold. However, the most striking feature of this table is that when the system is varying slowly, relative to the maximum steplength, the speedup of RS over RD is approximately 33 percent. Table 5 presents the similar data for the unsteady system.

In concluding this section we note that the execution times of RKFK are again comparable to the better of the two precisions from which it had to choose, and sometimes better than both.

Table 5: Relative run times of RKFK, RD and RS in an Unsteady System, Multiple Range Architecture

| | Relative Run Times | | | Function Evaluations | | |
|---|---|---|---|---|---|---|
| Tol | $t_d/t_k$ | $t_s/t_k$ | $t_s/t_d$ | RKFK | RD | RS |
| 1 | 0.99 | $> 5$ | $> 5$ | 3228 | 3198 | - |
| 10 | 1.00 | $> 5$ | $> 5$ | 2130 | 2118 | - |
| 30 | 0.99 | 1.45 | 1.46 | 1806 | 1794 | 3906 |
| 50 | 1.00 | 0.97 | 0.97 | 1680 | 1674 | 2412 |
| 100 | 1.10 | 0.81 | 0.74 | 1548 | 1536 | 1680 |
| 500 | 1.21 | 0.82 | 0.68 | 1302 | 1302 | 1308 |
| 1000 | 1.24 | 0.83 | 0.67 | 1230 | 1230 | 1230 |
| 5000 | 1.30 | 0.87 | 0.67 | 1104 | 1104 | 1104 |
| 10000 | 1.42 | 0.89 | 0.68 | 1062 | 1062 | 1068 |
| 100000 | 1.36 | 0.92 | 0.67 | 1020 | 1020 | 1020 |

Indeed there is effectively no advantage to using a double precision code if a dynamic precision code is available. There may be some advantage to using a single precision code if it can be determined *a-priori* that the system will remain in the region where the performance of the single precision solver is the best of the three. In general the user may not be able to determine *a-priori* the behavior of his system. In this case it would be unwise to use a single precision algorithm because of the penalty associated with a tight tolerance. A dynamic precision algorithm insures that the efficiency of the solver will be high, regardless of the nature of the system being solved. Furthermore users of numerical libraries are often unfamiliar with the details of the implementation of the code. It is difficult for them to pick the most efficient precision. A dynamic precision algorithm *adapts* itself to the problem, producing close to optimal execution times over a wide range of problem parameters.

## 5 Conclusions

We have shown that in numerical applications such as IVP solvers, there is an essential trade-off associated with choosing a precision. A smaller precision results in a less accurate computation, but may be faster. The converse is true for a larger precision. A dynamic precision algorithm is able to tailor itself such that it uses larger precision when it has to, and smaller precision when possible. We have examined an instance of a dynamic precision algorithm and its performance on

two different types of architectures. It became apparent that for fixed range architectures, where performance is often not critical, the speedup available from using smaller precision was small. Nevertheless on such architectures, by using a dynamic precision algorithm one can guarantee that the performance will be no worse than that of the the larger precision solver.

Multiple range architectures tend to be extremely expensive and every is usually made to reduce the execution time. It is on these architectures that a dynamic precision algorithm, like the RKFK is most useful. The penalty associated with using too small a precision is high, but the benefit is a significant reduction in the total execution time. In the case of the N-Body problem, this reduction was as large as one third. The former penalty can be averted and the speedup gained, by using a dynamic precision algorithm. We believe that the applicability of dynamic range algorithms to this broad base of high performance architectures will lead to their widespread acceptance.

Finally it should be noted that while this paper presented the results of only one implementation of a dynamic precision algorithm, we feel that they potentially have widespread application. The Runge-Kutta method was implemented here as an illustration, but other solvers could have been used. There is potential for application of these methods to other fields of numerical computation, particularly for those in which error feedback is available.

# References

[1] Brankin R.W. et al. *Algorithm 670 A Runge-Kutta-Nystrom Code*, ACM Transactions on Mathematical Software, Vol. 15, No 1, March 1989, pp 31 - 40.

[2] Burden R.L. and Faires A.C., *Numerical Analysis*, Prindle, Weber and Schmidt, 1985.

[3] Cash J.R. A Block 6(4) Runge-Kutta Formula for Nonstiff Initial Value Problems , ACM Transactions on Mathematical Software, Vol 15, No. 1, March 1989, pp 15 - 28.

[4] Cash J.R. and Karp A.H. , *A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right Hand Sides*, ACM Transactions on Mathematical Software, Vol 16, No. 3, September 1990, pp 201 - 222.

[5] Fehlberg E., *Classical Eighth and Lower Order R-K-N Formulas with Step Size Control for Special Second Order Differential Equations*, NASA Technical report R-381, 1972, Washington D.C.

[6] Forsythe G.E., Malcolm M.A. and Moler C.B., *Computer Methods for Mathematical Computations*, Prentice Hall, 1972 .

[7] Hairer E., Norsett S.P., Wanner G, *Solving Ordinary Differential Equations I, Nonstiff Problems*, Springer-Verlag, 1980.

[8] Henrici P., *Discrete Variable Methods in Ordinary Differential Equations*, John Wiley and Sons Inc. New York, 1962.

[9] Henrici P., *Error Propagation for Difference Methods*, John Wiley and Sons Inc. New York, 1963.

[10] Hwang K. and Briggs F.A. *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[11] ..., *IMSL Users Manual, Version 1.0*, IMSL Problem Solving Software Systems, April 1987.

[12] Kahan W.M., Turing Award acceptance speech, ACM Computer Science Conference, Washington D.C., November 1990.

[13] Lenferink H.W.J. and Spijker M.N., *On the Use of Stability Regions in the Numerical Analysis of Initial Value Problems*, Mathematics of Computation, V57, No. 195, July 1991, pp 221 - 237.

[14] MasPar Computer Corporation, *MasPar Parallel Application Language (MPL) User Guide, Version 2.0*, MasPar Computer Corporation, Sunnyvale CA.

[15] Shampine L.F., *What Everyone Solving Differential Equations Should Know*, pp 2 - 17, in Gladwell I., Sayers D.K., *Computational Techniques for Ordinary Differential Equations*, Academic Press, 1980 .

# Appendix A The RKFK Algorithm

The RKFK algorithm is a modified version of the RKF algorithm presented in [2]. It is presented in pseudocode form in table 6 below. In table 6 the normal type is the original RKF algorithm, while the boldface shows the additional steps due to RKFK.

Table 6: Pseudocode for the RKFK algorithm

*time* $= \alpha$

$\overline{Y} = \overline{K}$

$h = h_{max}$

**precision = double**

*while time* $< t_{end}$

    *for i = 1, ..., 6*

        $k_i = hf'(time, k_{i-1}, \ldots, k_1)$

    $error = g(k_1, \ldots, k_6)$

    *if error < tolerance*

        *time = time + h*

        *update* $\overline{Y(time)}$

    $h = q(h, error)$

    *if* $h \geq h_{max}$

        $h = h_{max}$

        **precision = single**

        **else precision = double**

# Appendix B The N-Body Problem

The N-Body Problem is described by Newtons law of gravitational attraction, namely

$$F = \frac{m_1 m_2 G}{r^2}$$

Where F is the force between two bodies, $m_1$ and $m_2$ are the masses of the bodies, G is the constant of gravitational attraction, and $r^2$ is the distance between the bodies. For the purposes of this example we simulated the motion of the bodies in two dimensions. Each body in the system has four equations of motion associated with it. These are $x_i$ and $y_i$ the components of position in space, and $v_{xi}$ and $v_{yi}$ the components of velocity. The derivatives of these parameters $f'(x_i, y_i, v_{xi}, v_{yi})$ can be evaluated as follows:

$$x_i' = v_{xi}$$

$$y_i' = v_{yi}$$

$$v_{xi}' = G \sum_{j=1, j \neq i}^{n} \frac{m_j(x_j - x_i)}{r^3}$$

$$v_{yi}' = G \sum_{j=1, j \neq i}^{n} \frac{m_j(y_j - y_i)}{r^3}$$