Runge-Kutta Project Report

Daniel Henderson

July 22, 2021

Introduction

This report investigates various explicit Runge-Kutta methods with an adaptive step size for solving an initial value problem for an autonomous system of ordinary differential equations. The methods will be implemented in Matlab, where our code is displayed in the apendix. First there is a discussion on the assumptions used in controlling the adaptive step and methodology behind using a dynamic memory allocation. Then an there is an attempt to convince the reader that the code is working properly. Next, there is an investigation of the dynamic allocation performance of memory verses a fixed size allocation performance for both $2-3$ Runge-Kutta method and a $4-5$ Runge-Kutta method as well as compare them to the performance of the Matlab solvers of similar form. The performance investigation will be done on a relevant problem - An SIR model compartmental model that shows the spread of an infectious disease.

Adaptive time-stepping methods:

Consider the initial value problem of the form $x' = f(t)$ over $t_0 \leq t \leq t_f$ with an initial condition of $f(a) = x_0$. Recall that a numerical approximation of this problem determines a set of N points that approximates the solution of $x(t_n)$ by x_n, over a chosen grid of the form $t_0 < t_1 < t_2 < \cdots t_N = t_f$. When using a fixed step size the grid is of the form $t_n = \Delta t \cdot n + t_0$ where $\Delta t = \frac{t_f - t_0}{N}$. An inherent problem of using a fixed step size for Δt is that at for some values of n the error in our approximation may be quite large, while at other values of n it can be quite small. Observe the illustrative example below of the function $x(t) = e^{-t}\sin(30t) + \sin(t)$:

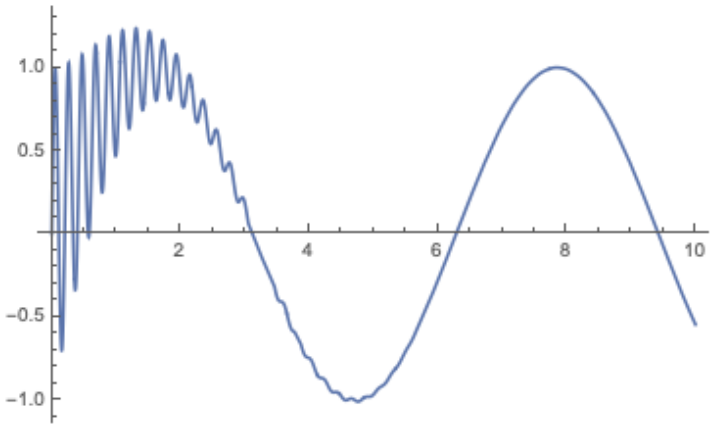

Figure 1: The graph of $x(t)$ is the solution to the IVP $x'(t) = -x + \cos(30e^{-t})\cos(30t) + \cos(t) + \sin(t)$ where $x(0) = 0$.

To solve the problem, presented in the figure above, with an error that is less than ϵ we would need to use a really small step size for $t < 5$ to meet our error criteria. But as the function becomes less rapidly-varying, we could increase our step and still meet our specified error tolerance by doing less work. Problems of this form motivate the need to have an algorithm that changes its step size.

Next we must introduce a few terms. The term *local truncation error*, or LTE for short, refers to the error committed by taking an individual step using a truncated Taylor series approximation. The term *global error* refers to the difference between the exact value of our solution and approximation at any point in the grid, i.e. $|x(t_n) - x_n|$. We will assume that an algorithm that has a local truncation error of $O(\Delta t^q)$ will produce a result that leads to a global error of $O(\Delta t^{q-1})$. Then our adaptive time-stepping procedure had the following properties:

1 At the nth iteration it computes x_n from x_{n-1}, and estimates the LTE in the step.
2 If the LTE is larger than ϵ, then we repeat (1) with a smaller step size.
3 If the LTE is smaller than ϵ, then we accept the step and make estimate of the next step size.

Step Size Control:

As discussed, to control the step size we must have a means of estimating the local truncation error at each step. This can be done by computing two different approximations of different order for $x(t_n)$. Let x_n refer to the approximation that has a local truncation error of $O(\Delta t^{q+1})$ and $\hat{x}_n$ refer to the approximation that has a local truncation error of $O(\Delta t^q)$. We will use $\hat{x}_n$ to estimate the LTE in x_n as $|\hat{x}_n - x_n| = e_{est}$. If $e_{est} > \epsilon$ we can simply half Δt and repeat the iteration. Otherwise, $e_{est} < \epsilon$ and we must determine a value for $\hat{\Delta}t$, the time step at the next iteration. We will simply scale the current step size to do this, so our problem is to solve for α in the equation $\hat{\Delta}t = \alpha\Delta t$ Note that LTE in $\hat{x}_n$ is $\hat{C}_n\Delta t^q$. We will assume that $\hat{C}_n = \hat{C}_{n+1}$ and solve for α such that the LTE error in the x_{n+1} is ϵ, i.e. $\epsilon = \hat{C}_{n+1}\hat{\Delta}t^q = \hat{C}_n(\alpha\Delta t)^q = \alpha^q e_{est}$. Then solving for the scalar gives $\alpha = \left(\frac{\epsilon}{e_{est}}\right)^{\frac{1}{q}}$. In practice it is common to be more conservative in determining $\hat{\Delta}t$ since we want it to be less than ϵ and not equal to ϵ, thus, we say $\hat{\Delta}t = 0.9\alpha\Delta t$. See appendix for specific implementation details.

Dynamic Allocation Methodology:

An inherent problem of implementing an adaptive step size procedure is that we are unaware of the size of our solution prior to our approximation. This is because at each step, we determine a value of Δt that produces the largest error that is less than ϵ. As you have seen from the illustrative example above, this value of Δt can change drastically upon the nature of the problem at hand. One option would be to implement a maximum number of steps and march forward in time until you reach such a point. This would allow you to allocate an appropriate amount of memory prior to performing the approximation based upon the worst case that you reach the maximum number of steps. However, memory allocation can be expensive in terms of time and space when solving a large system of ODE's.

An alternative method would be to use a dynamically growing structure to store your solution. Suppose you make a conservative estimate that your solution will be no more than n elements, which can be based upon your first step value of Δt and the span of time interval. We will perform an amortized analysis. Then for each iteration from 1 to n, the average cost of inserting an element into your array is $\Theta(1)$, which implies that the first n iterations has a cost of $n\Theta(1)$. Then suppose we take another iteration, but before we can insert the $n+1$ step we must double array initial array allocation to $2n$ elements and insert the previous n elements into this new array. So before the $n+1$th iteration we must perform an operation that has an average cost of $\Theta(n)$. So the average cost of insertion $n+1$ elements is $\frac{(n+1)\Theta(1)+\Theta(n)}{n+1} = \frac{(n+1)\Theta(1)+n\Theta(1)}{n+1} = \frac{(n+2)\Theta(1)}{n+1} = \Theta(1)$.

However, it must be noted that the above amortized analysis can be slightly misleading because it doesn't account for the time expense of allocating a new portion of memory when the space is reached. Though, this is the standard approach behind the dynamic structure of an Arraylist object in Java and a vector object in c++[1]. We will be comparing the computation times behind equivalent procedures that perform a dynamic allocation and a fixed step size allocation.

Discussion of Code:

Below is a diagram displaying our code base.

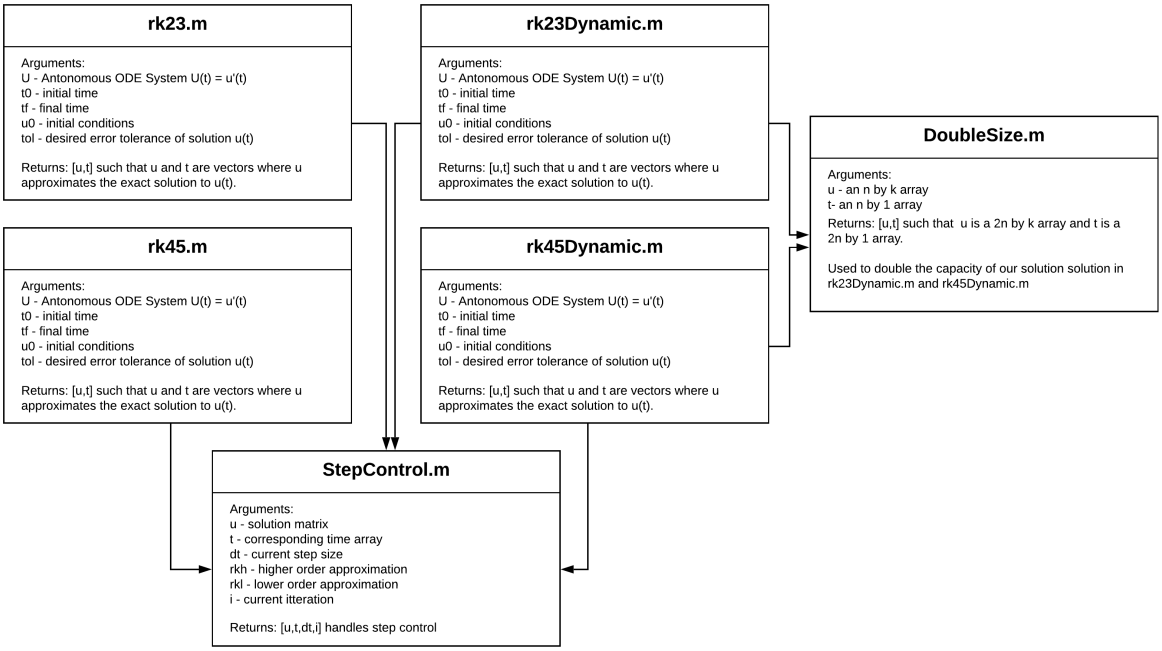

Figure 2: Each box represents a Matlab m-file and an arrow represents a dependency from one file to another. Specifically, our step control for all four of our Runge-Kutta schemes was handled externally in our StepControl.m file. Furthermore, our dynamic implementations call the DoubleSize.m file which allocates a new solution and time array that is double the size and copies over the elements of the previous arrays

We have chosen to implement the Dormand-Prince 4th and 5th order Runge-Kutta Scheme for the rk45.m and rk45Dynamic.m files. For rk23.m and rk23Dynamic.m files we implemented the Bogacki-Shampine scheme. These schemes have the following butcher tables[2]:

0				
1/2	1/2			
3/4	0	3/4		
1	2/9	1/3	4/9	
	2/9	1/3	4/9	0
	7/24	1/4	1/3	1/8

0							
1/5	1/5						
3/10	3/40	9/40					
4/5	44/45	–56/15	32/9				
8/9	19372/6561	–25360/2187	64448/6561	–212/729			
1	9017/3168	–355/33	46732/5247	49/176	–5103/18656		
1	35/384	0	500/1113	125/192	–2187/6784	11/84	
	35/384	0	500/1113	125/192	–2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	–92097/339200	187/2100	1/40

Figure 3: The table on the left is the Bogacki-Shampine scheme where the last row gives a third-order solution and the second to last row gives the second-order solution. The table on the right is the Dormand-Prince scheme where the last row gives a fifth-order accurate solution, and the second to last row gives a fourth-order solution.

Accuracy of Code:

We will return to the illustrative example shown in Figure 1 and start by performing a brief error analysis.

```
Euclidean Norm of Error vecotor for rk45: 6.296666e-08
Infinity Norm of Error vector for rk45: 5.174516e-09

Euclidean Norm of Error vecotor for rk45Dynamic: 6.296666e-08
Infinity Norm of Error vector for rk45Dynamic: 5.174516e-09

Euclidean Norm of Error vector for ode45: 7.642389e-05
Infinity Norm of Error vector for ode45: 1.952047e-05

Euclidean Norm of Error vecotor for rk23: 6.524806e-06
Infinity Norm of Error vector for rk23:9.698895e-08

Euclidean Norm of Error vecotor for rk23Dynamic: 6.524806e-06
Infinity Norm of Error vector for rk23Dynamic: 9.698895e-08

Euclidean Norm of Error vector for ode23: 4.355816e-05
Infinity Norm of Error vector for ode23: 7.668893e-06
```

Figure 4: We ran our methods on the Transient IVP over a time interval of $0 \leq t \leq 15$ with a global tolerance of 10^{-8} and observed the results shown above.

The calculations shown in Figure 3 are evidence that our algorithms are performing an accurate approximation of our solution of roughly the desired order that we specified. However, this is not the case for the Matalab functions ode23 and ode45 - they both fail to meet our specified tolerance goal. This is quite a nasty problem for the 2-3 Runge-Kutta methods to solve, so it makes sense that they are not performing as accurately as we would like. Below is evidence the step control is working properly.

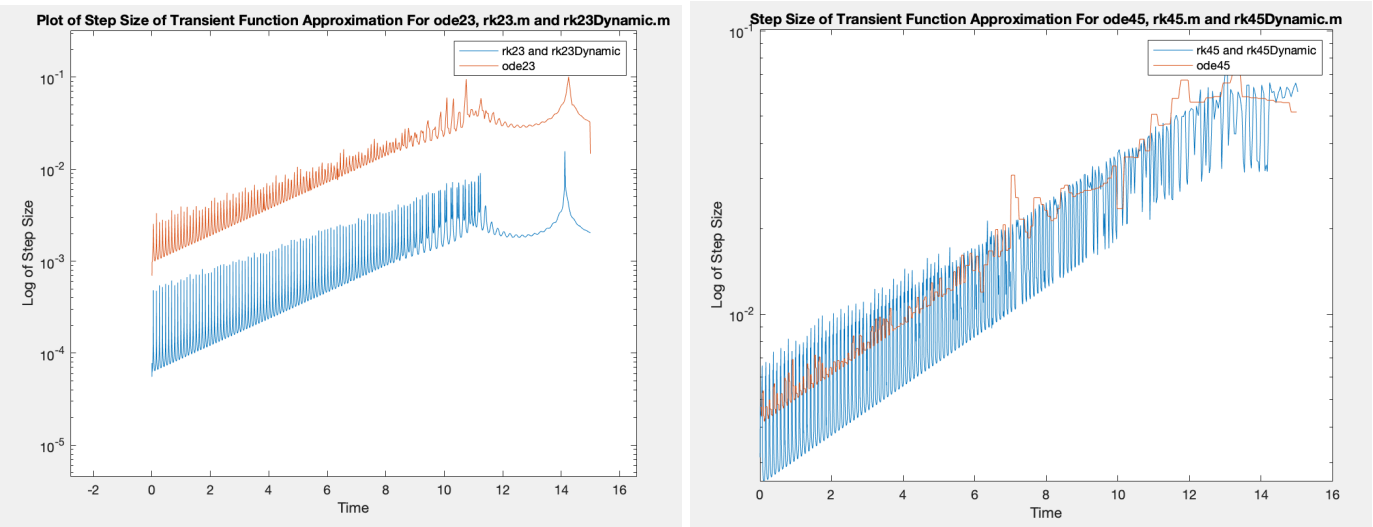

Figure 5: Verification of our Adaptive Time Step Behaving properly, as shown by an increase in step size as time increases

As discussed, when solving a transient problem of this form we would like to see our step size increase as the function starts to vary less as rapidly as time increases, and this is precisely what is shown. The oscillatory nature of these two graphs is a consequence of the algorithm needing to readjust it step size due to our estimated error being greater than our tolerance. This is a result of the transient term e^{-30t} in our solution. This behavior is also observed in the Matlab ode23 and ode45 approximations though, they don't simply have there step when to large of an error is committed - as indicated by the smaller amplitude in the oscillations, which is more clear on the right hand side graph.

Introduction of SIR Model:

The SIR model is a system of ordinary differential equations (ODE), that models the spread of an infectious disease throughout time. The equations are

$$\begin{aligned} S'(t) &= -\alpha S(t)I(t) \\ I'(t) &= \alpha S(t)I(t) - \gamma I(t) \\ R'(t) &= \gamma I(t) \\ \text{Population} &= S(t) + I(t) + R(t) \end{aligned}$$

At any point in time, $S(t)$ is the number of susceptible people in the population, $I(t)$ is the number of infected, and $R(t)$ is the number of recovered people in the population. We assume that everyone behaves the same and wanders around all day, randomly coming into contact with the same amount of people. Additionally, no one is immune from being infected until they are infected and recovered. Then the parameter α is a proportion of the population that each individual comes in contact with each day and the parameter γ refers to the average recovery rate of the disease in days. Below is a solution to the system of equations,

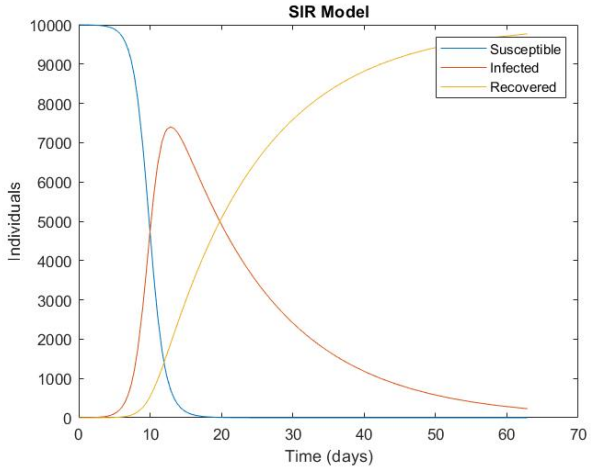

Figure 6: Numerical solution to the basic SIR model. Over time the susceptible population approaches 0, while the recovered population approaches the full population. The mean recovery rate is set at $\frac{1}{14}$, α is set at 0.0001, and the population at 10^4

Timing Test:

To test the computational times of our algorithms we computed the solution to the SIR model, as shown in figure 6, 100 times for all four of our methods and their Matlab equivalents. This was done at a relative error tolerance of 10^{-8} for all 6 of the competitors.

```
Computation of rk45Dynamic: Elapsed time is 0.658189 seconds.
Computation of rk45: Elapsed time is 0.683086 seconds.
Computation of ode45 Elapsed time is 0.333011 seconds.

Computation of rk23Dynamic Elapsed time is 3.759235 seconds.
Computation of rk23 Elapsed time is 3.606205 seconds.
Computation of ode23 Elapsed time is 1.828720 seconds.
```

Figure 7: Timing Test

Timing Test Continued:

As shown in Figure 4, our dynamic and fixed allocation are computing identical results. So we can compare between the two without hesitation. In this problem it appears that the fixed allocation outpaces the dynamic allocation. However, the Matlab equivalents to our schemes are clearly winning the race. As we say in in Figure 4, specifying a tolerance goal for the Matlab algorithms does not guarantee that the solution is to that tolerance. So it might be the case that they are so much quicker becuase they are computing a less accutate approximation. If this is the case, we should see larger step sizes being taken which is shown below

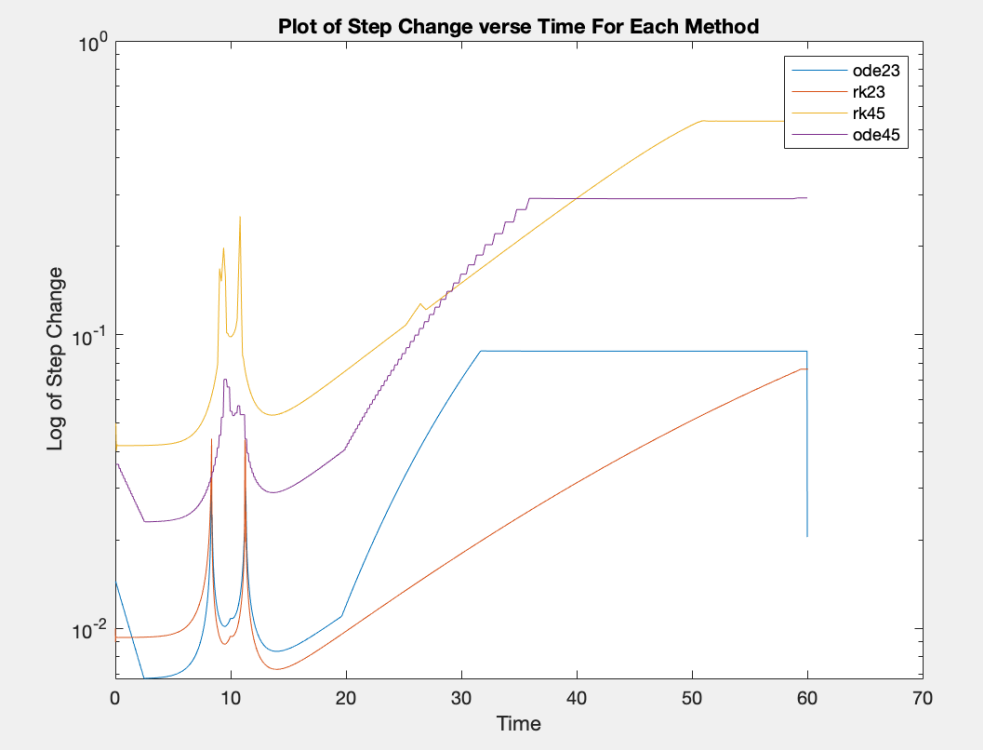

Figure 8: Step Analysis

The ode23 clearly takes much larger step sizes than the rk23 method, but that is not the case for the 4-5 schemes. A more thorough timing test would certainly take in to account the error of each approximation. Nonetheless, the step behavior is working as expected. That is, the step size increases as the action in the ode's decrease, which is shown in the relatively flat solution curves for t greater than 30 days.

One more thing to note on performance - Our algorithms could benefit from an orientation change of storing and accessing our solution. They have been implemented in a row-oriented fashion, that is, we store each solution in its own row of a matrix. This is an inefficient way of accessing matrices in Matlab, which store there arrays in a column-major form.

Appendix:

The following script controls the step size of the numerical solution. It either increases or decreases the size of the step, based on the truncation error estimate of two runge-kutta schemes of different order.

```
function [u,t,dt,i] = StepControl(rkh, rkl, u, t, i, dt, q, tol)
    errEstAbs = max(abs(rkh - rkl));
    errEstRel = max(abs(rkh - rkl)./abs(rkh));
    err = min(errEstAbs, errEstRel);
    if err > tol
        dt = 0.5*dt;
    else
        t(i+1) = t(i)+dt;
        u(:,i+1) = rkh;
        i = i+1;
        dt = 0.9*dt*(tol/err)^(1/q);
    end
end
```

The following script is used to dynamically grow the solution size in the dynamic Runge-Kutta methods, found on the next two pages.

```
function [U,T] = DoubleSize(u,t)
U = zeros(size(u,1),2*size(u,2));
T = zeros(1,2*size(t,2));
U(1:size(u,1), 1:size(u,2)) = u;
T(1,1:size(t,2)) = t;
end
```

The following script uses a dynamically allocated storage to run a RK23 method to solve an initial value problem. It solves the initial value problem, defined by the input function f, where f is an autonomous system of n first-order ODE's having elements of the form $u' = f(u)$, with the initial condition in $u0$. Both f and $u0$ are column vectors of length n. The solution is assumed to exist over the time interval $t0$ to tf. The relative local truncation error is estimated at each step and it must be less than specified *tol* to move forward in time.

```
function [U,t] = rk23Dynamic(f,t0,tf,u0,tol)
% initialize
capacity = (tf-t0)*10;
dt = 0.01;                          % initial step size
i = 1;                              % current solution step
U = zeros(size(u0,1), capacity);    % solution matrix
t = zeros(1, capacity);             % points of approximation
U(:,i) = u0;                        % store initial condition
t(i) = t0;                          % store initial starting time
tol = tol/(tf-t0);
while (t(i) < tf) && (i < 1e7)
    m1 = f(U(:,i))*dt;
    m2 = f(U(:,i)+0.5*m1)*dt;
    m3 = f(U(:,i)+0.75*m2)*dt;
    rk2 = U(:,i)+2/9*m1+1/3*m2+4/9*m3;
    m4 = f(rk2)*dt;
    rk3 = U(:,i)+7/24*m1+1/4*m2+1/3*m3+1/8*m4;
    [U,t,dt,i] = StepControl(rk3,rk2,U,t,i,dt,2,tol);
    if capacity == i
        [U,t] = DoubleSize(U,t);
        capacity = capacity*2;
    end
end
U = U(:,1:i);
t = t(:,1:i);
```

The following script uses dynamically allocated storage to run a RK45 method to solve an initial value problem. It solves the initial value problem, defined by the input function f, where f is an autonomous system of n first-order ODE's having elements of the form $u' = f(u)$, with the initial condition in $u0$. Both f and $u0$ are column vectors of length n. The solution is assumed to exist over the time interval $t0$ to tf. The relative local truncation error is estimated at each step and it must be less than specified *tol* to move forward in time.

```
function [U,t] = rk45Dynamic(f,t0,tf,u0,tol)
% initialize
capacity = (tf-t0)*100;
dt = 0.1;                              % initial step size
i = 1;                                  % current solution step
U = zeros(size(u0,1), capacity);        % solution matrix
t = zeros(1, capacity);                 % points of approximation
U(:,i) = u0;                            % store initial condition
t(i) = t0;                              % store initial starting time

tol = tol/(tf-t0);
while (t(i) < tf) && (i < 1e7)
    k1 = f(U(:,i))*dt;
    k2 = f(U(:,i)+0.2*k1)*dt;
    k3 = f(U(:,i)+1/40*(3*k1+9*k2))*dt;
    k4 = f(U(:,i)+44/45*k1-56/15*k2+32/9*k3)*dt;
    k5 = f(U(:,i)+19372/6561*k1-25360/2187*k2+64448/6561*k3-212/729*k4)*dt;
    k6 = f(U(:,i)+9017/3168*k1-355/33*k2+46732/5247*k3+49/176*k4-5103/18656*k5)*dt;
    rk4 = U(:,i)+35/384*k1+500/1113*k3+125/192*k4-2187/6784*k5+11/84*k6;
    k7 = f(rk4)*dt;
    rk5 = U(:,i)+5179/57600*k1+7571/16695*k3+393/640*k4-92097/339200*k5+187/2100*k6+1/40
    [U,t,dt,i] = StepControl(rk5,rk4,U,t,i,dt,4,tol);
    if capacity == i
        [U,t] = DoubleSize(U,t);
        capacity = capacity*2;
    end
end
U = U(:,1:i);
t = t(:,1:i);
```

The following script solves the initial value problem using an RK23 method. The initial value problem is defined by the input function f, where f is an autonomous system of n first-order ODE's having elements of the form $u' = f(u)$, with the initial condition in $u0$. Both f and $u0$ are column vectors of length n. The solution is assumed to exist over the time interval $t0$ to tf. The relative local truncation error is estimated at each step and it must be less than specified *tol* to move forward in time. This method terminates upon reach of tf or after $10^5 - 1$ steps.

```
function [U,t] = rk23(f,t0,tf,u0,tol)
% initialize
dt = 0.01;                          % initial step size
i = 1;                              % current solution step
U = zeros(size(u0,1), 1e5);         % solution matrix
t = zeros(1, 1e5);                  % points of approximation
U(:,i) = u0;                        % store initial condition
t(i) = t0;                          % store initial starting time

tol = tol/(tf-t0);
while (t(i) < tf) && (i < 1e5)
    m1 = f(U(:,i))*dt;
    m2 = f(U(:,i)+0.5*m1)*dt;
    m3 = f(U(:,i)+0.75*m2)*dt;
    rk2 = U(:,i)+2/9*m1+1/3*m2+4/9*m3;
    m4 = f(rk2)*dt;
    rk3 = U(:,i)+7/24*m1+1/4*m2+1/3*m3+1/8*m4;
    [U,t,dt,i] = StepControl(rk3,rk2,U,t,i,dt,2,tol);
end
U = U(:,1:i);
t = t(:,1:i);
```

The following script solves the initial value problem using an RK45 method. The initial value problem is defined by the input function f, where f is an autonomous system of n first-order ODE's having elements of the form $u' = f(u)$, with the initial condition in $u0$. Both f and $u0$ are column vectors of length n. The solution is assumed to exist over the time interval $t0$ to tf. The relative local truncation error is estimated at each step and it must be less than specified *tol* to move forward in time. This method terminates upon reach of tf or after $10^5 - 1$ steps.

```
function [U,t] = rk45(f,t0,tf,u0,tol)
% initialize
dt = 0.1;                          % initial step size
i = 1;                              % current solution step
U = zeros(size(u0,1), 1e5);         % solution matrix
t = zeros(1, 1e5);                  % points of approximation
U(:,i) = u0;                        % store initial condition
t(i) = t0;                          % store initial starting time

tol = tol/(tf-t0);
while (t(i) < tf) && (i < 1e5)
    k1 = f(U(:,i))*dt;
    k2 = f(U(:,i)+0.2*k1)*dt;
    k3 = f(U(:,i)+1/40*(3*k1+9*k2))*dt;
    k4 = f(U(:,i)+44/45*k1-56/15*k2+32/9*k3)*dt;
    k5 = f(U(:,i)+19372/6561*k1-25360/2187*k2+64448/6561*k3-212/729*k4)*dt;
    k6 = f(U(:,i)+9017/3168*k1-355/33*k2+46732/5247*k3+49/176*k4-5103/18656*k5)*dt;
    rk4 = U(:,i)+35/384*k1+500/1113*k3+125/192*k4-2187/6784*k5+11/84*k6;
    k7 = f(rk4)*dt;
    rk5 = U(:,i)+5179/57600*k1+7571/16695*k3+393/640*k4-92097/339200*k5+187/2100*k6+1/40
    [U,t,dt,i] = StepControl(rk5,rk4,U,t,i,dt,4,tol);
end
U = U(:,1:i);
t = t(:,1:i);
```

The following preforms a race between the dynamically allocated Runge-Kutta scripts, the pre-allocated Runge-Kutta scripts, and the equivalent schemes in the Matlab Suite.

```
a = 0.0001; b = 1/14; Pop = 10^4;
F = @(Y)[-a*Y(1)*Y(2); a*Y(1)*Y(2)-b*Y(2)];
f = @(t,Y)[-a*Y(1)*Y(2); a*Y(1)*Y(2)-b*Y(2)];

tic
fprintf('Computation of rk45Dynamic: ')
for i = 1:100 rk45Dynamic(F, 0, 60, [Pop-1; 1], 1e-4); end
toc

tic
fprintf('Computation of rk45: ')
for i = 1:100 rk45(F, 0, 60, [Pop-1; 1], 1e-4); end
toc

tic
fprintf('Computation of ode45 ')
for i = 1:100 [t1,y1]=ode45(f, [0 60], [Pop-1; 1]); end
toc

fprintf('\n')

tic
fprintf('Computation of rk23Dynamic ')
for i = 1:100 rk23Dynamic(F, 0, 60, [Pop-1; 1], 1e-4); end
toc

tic
fprintf('Computation of rk23 ')
for i = 1:100 rk23(F, 0, 60, [Pop-1; 1], 1e-4); end
toc

tic
fprintf('Computation of ode23 ')
for i = 1:100 [t2,y2]=ode23(f, [0 60], [Pop-1; 1]); end
toc
```

References

[1] Wikipedia. *Amortized analysis*. [Online; accessed 8-October-2020]. 2020. URL: %5Curl%7Bhttps://en.wikipedia.org/wiki/Amortized_analysis#:~:text=In%5C%20computer%5C%20science%5C%2C%5C%20amortized%5C%20analysis,algorithm%5C%2C%5C%20can%5C%20be%5C%20too%5C%20pessimistic%7D.

[2] Wikipedia. *List of Runge–Kutta methods*. [Online; accessed 8-October-2020]. 2020. URL: %5Curl%7Bhttps://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods#Dormand%E2%80%93Prince%7D.