2008 20th IEEE International Conference on Tools with Artificial Intelligence

The Performance of Approximating Ordinary Differential Equations by Neural Nets

Josef Fojdl and Rüdiger W. Brause
Institute of Informatics
Goethe-University, 60054 Frankfurt, Germany
{Fojdl, R.Brause} @ informatik.uni-frankfurt.de

Abstract

The dynamics of many systems are described by ordinary differential equations (ODE). Solving ODEs with standard methods (i.e. numerical integration) needs a high amount of computing time but only a small amount of storage memory. For some applications, e.g. short time weather forecast or real time robot control, long computation times are prohibitive. Is there a method which uses less computing time (but has drawbacks in other aspects, e.g. memory), so that the computation of ODEs gets faster? We will try to discuss this question for the method of a neural network which was trained on ODE dynamics and compare both methods using the same approximation error.
In many cases, as for physics engines used in computer games, the shape of the approximation curve is important and not the exact values of the approximation. Therefore, we introduce as error measure the subjective error based on the Total Least Square Error (TLSE) which gives more consistent results than the standard error.
Finally, we derive a method to evaluate where neural nets are advantageous over numerical ODE integration and where this is not the case.

1 Introduction

The control and prediction of the dynamics of systems is a important issue in technical, medical, biological and economical applications. In the standard approach, these systems or their control are modeled by ordinary differential equations and implemented either by hardware (analog electronic for small problems) or by software based simulation. For most real world applications, there does not exist an analytical solution but simulation is used, based on numerical integration.

There are many applications where excessive simulation computation times hinders or prohibits the application. Examples are short-term weather forecast or the ballistic control of robot movement in real time. In general, for the simulation of physical laws in real time special software modules called "physics engines" are used. A physics engine uses variables such as mass, velocity, friction and wind resistance and can simulate and predict effects under different conditions that would approximate what happens in either real life or a fantasy world. Physics engines are increasingly relevant to more and more to video games, in addition to their use in scientific simulations and animation movie generation. The physics engine gives the ability to reach unprecedented levels of realism in modeling physical rules of the real world, letting you focus on the logic of your application and not on the physical laws effecting the simulated object. Physics engines are made either in software[2] (e.g. Open Dynamics Engine ODE.org, Envy SDK, Havok SDK, PhysX SDK) or migrated in hardware, called a Physics Processing Unit (PPU). Certainly, the latter should be faster, but is more expensive.

The problem of accelerating the computations in physics engines has not been addressed in literature, up to now. Although the performance of standard iterative methods for approximating the solution of ordinary differential equations (ODE) which are the base for the computations done in physics engines is well known [14] the accuracy of physics engines is only sparsely considered (see e.g. Choi et al. [6]).

What do we have as fast alternatives to numerical integration methods? As alternative approach let us consider in this paper artificial neural networks. Coupled linear differential equations can be modeled by feed-back neural networks [5], especially Hopfield networks [12], implemented in hardware [17]. The local and global minima of the network energy provide the stable solutions of the system. It is also well known that feed-forward two-layered neural networks can approximate every function arbitrary well, e.g. [9]. For the approximation of a function we can use simple feed-forward networks combined with piecewise-linear approximation [13] or general nonlinear kernels [11].

1082-3409/08 $25.00 © 2008 IEEE
DOI 10.1109/ICTAI.2008.44

All these approaches show that ODEs can be transfered into the activity of neural networks, but do not answer the question: What should we prefer in practice? ODE simulation or neural networks? There are also some rumors about the performance of neural networks like small computation time and augmented use of memory. How do these features evolve if we use the network to approximate ODEs? Can we compare the two methods by resource requirements like computation or storage complexity? In a previous work, Brause [3] has shown that the error of a two-layer neural network approximating a robot manipulator position can be optimized using only a restricted amount of storage. This corresponds to the error-bounded descriptive complexity of a neural network approximation [4]. There, the neural network approximation can be done very fast by using linear splines set up at their grid points which corresponds to a fast table lookup procedure.

Now, comparing ODEs and neural networks, what should we take? Before we can answer this question, we first have to calculate the error of both the ODEs and the neural network. Then, assuming the same error for both methods, a comparison is made between the necessary resources of the ODE approximation and the storage-optimized approximation network. The answer is given by a decision procedure presented in this paper and depends on all available resources.

2 Computing ODEs

For the sake of clarity in our investigation let us regard only a very restricted class of differential equations: that of ordinary equations of order one.

2.1 Dynamic modeling by ODEs

An ordinary differential equation contains the variable $y(t)$ and its derivatives $y'(t), y''(t), y'''(t), ...$ Let us concentrate on differential equations of order one, i.e. equations with $y(t)$ and the first derivative $y'(t)$.

$$\frac{dy(t)}{dt} = f(t, y(t)) \tag{1}$$

with the arbitrary function $f(t, y(t))$ of t and $y(t)$.
As solution, we have in the interval $[t_0, T]$

$$y(t) = y(t; t_0, y(t_0)) \tag{2}$$

with the starting values t_0 and $y(t_0)$.

2.2 Example: The frictional rigid body motion

Let us introduce now a simple example for a physical ODE, the rigid body movement with friction (decoupled for each direction) for a body of mass m. Within the context of gravity acceleration g, the coefficient of the flow resistance c_W, the density of the fluid ρ (in our case: the air density), and the reference area A of the body in the moving direction y we define the constant $k = \frac{1}{2}\rho c_W A$ which we will use later on. Now, the body dynamics is described by (see[1])

a) movement upwards

$$\frac{dy_O(t)}{dt} = v_\infty tan(arctan\frac{v_0}{v_\infty} - \frac{g}{v_\infty}t) \tag{3}$$

with

$$v_\infty = \sqrt{\frac{mg}{k}}, \quad t_U = \frac{v_\infty}{g} arctan\frac{v_0}{v_\infty} \tag{4}$$

where t_U is the point of return of the body.
The solution of the dynamics is also well known for $t \leq t_U$:

$$y_O(t) = \frac{v_\infty}{g}\left(ln\ cos\frac{g(t_U - t)}{v_\infty} - ln\ cos\frac{gt_U}{v_\infty}\right) \tag{5}$$

By this, the movement of the body against gravity is well described. For the movement back to the ground we have

b) movement downwards

$$\frac{y_U(t)}{dt} = -\left(v_\infty - \frac{2v_\infty}{1 + e^{p(t-t_U)}}\right) \quad t \geq t_U \tag{6}$$

with

$$p = 2\frac{k}{m}v_\infty \tag{7}$$

As solution of this ballistic equation we get for $t \geq t_U$

$$y_U(t) = y_O(t_U) - v_\infty(t - t_U) - \frac{m}{k}ln\left(\frac{1}{2}e^{-p(t-t_U)} + \frac{1}{2}\right) \tag{8}$$

Using both solutions in combination we can sketch the picture of the whole movement. *Fig.* 1 shows us the height of a body trajectory, e.g. a bowling ball of mass $7.275kg$, with a diameter of $0.12m$ and an initial speed of $v_0 = 1000\frac{m}{s}$. For the friction we assume that the air temperature was 0^o C with an air pressure of 1013 hPa, i.e. $\rho = 1.293\frac{kg}{m^3}$

Each state of this analytical solution at time t has to be approximated by a simulation. Please note that we have indeed two tasks: first to find approximation values for the analytical solution of the ODE at different time points, and then, independently of this sample rate, resample the approximation function for the simulation task. Both sample and interpolation procedures are independent of each other. In this paper, we will only consider the first one, the approximation of the solution.

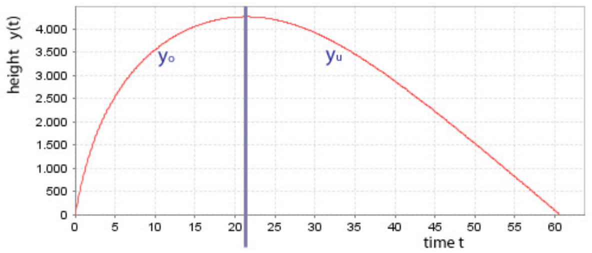

Figure 1. Height of body course with eqs. (3),(6) and $m = 7.275kg$

2.3 The Euler approximation

Standard iterative methods for approximating the solution of ordinary differential equations, like Euler or Runge-Kutta methods, have the handicap of using much computational time when a small error is demanded. The more accurate the method performs, the more computation time is needed. To inspect this in more detail, let us regard the error of the Euler method as simple but representative method. Our differential equation has the form

$$\frac{dy}{dt} = f(t, y), \ y(t_0) = y_0 \tag{9}$$

with y_0 as initial starting value. We define the solution of this differential equation at time t with starting value y_0 by $y(t; t_0, y_0)$. Let the upper time limit of t be time T. Then, the interval $[t_0, T]$ can be partitioned in N equal sized subintervals $[t_n, t_{n+1}]$ and $t_{n+1} - t_n = h$.
The Euler method is based on the idea of approximating the exact solution at time t by one step from the initial starting value and then building the next approximation step on the previous one:

$$\tilde{y}_{n+1}(h) = \tilde{y}_n(h) + f(t_n, \tilde{y}_n(h))h \tag{10}$$

The approximation error corresponds to the non-linear higher terms in a Taylor expansion of the exact solution which we neglect. Here, we have to distinguish between the local and the global error of approximation [15] [10]. The local one is obtained after each step, the global one results by accumulating the local errors of all approximating steps. Let us first focus on the local one.
The local error $L_{n+1}(h)$ is the difference between the approximation and the solution at time t_{n+1}. The maximal local error is given by (see [10])

$$L_{max}(h) = K_T h^2 \tag{11}$$

with constant K_T containing the first derivative. This means, by h^2 the local error is of order two and depends only on step width h.

The global approximation error $G_n(h)$ is the difference between the exact solution of the ODE $y(t_n; t_0, y_0)$ and our step-wise approximated value $\tilde{y}_n(h)$ at time step t_n after n steps with width h

$$G_n(h) = |y(t_n; t_0, y_0) - \tilde{y}_n(h)| \tag{12}$$

The upper limit for the global error $G_n(h)$ becomes with the maximal error $C_T h$ within interval T and step width h (see [10])

$$G_{max}(h) = C_T h \tag{13}$$

This means our global error is of order 1. Nevertheless, the computation load for obtaining the result is linear in n, the number of steps to obtain the last value at the end of the interval which is the approximation $\tilde{y}_n$ at the desired coordinate T.

3 The Neural Network approximation

The most simple form of neural networks is a feed-forward net, like the one shown in (2). For the function once learned by the network, the acyclic feed-forward nets have the big advantage of constant computing time.
Let us use as non-linear function the activation function

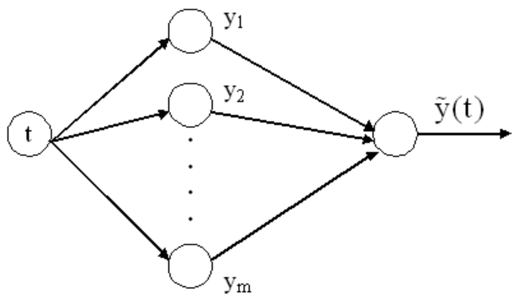

Figure 2. One input feed forward network

$$y_i(t) = \begin{cases} 1 & t \geq t_{1i} \\ t & t_{0i} \leq t < t_{1i} \\ 0 & t < t_{0i} \end{cases} \tag{14}$$

and

$$\tilde{y}(t) = \sum_i w_i y_i(t) \tag{15}$$

This results in the approximation of a non-linear function by splines. The i-th spline is produced by neuron i in the interval $[t_{0i}, t_{1i}]$ by eq.(14). In this case, we can interpret a weight of the network as the gradient of a spline between two interpolation points $y(t_n)$ and $y(t_{n+1})$. We might compute every approximation between two sample points by a simple linear interpolation

$$\tilde{y}_n(t) = y(t_n) + \frac{y(t_{n+1}) - y(t_n)}{t_{n+1} - t_n}(t - t_n) \qquad (16)$$

if the weights have been set by the interpolation points and the associated solution (2) of the ODE.

The network of one input unit is sufficient for a ordinary differential equation of one variable. The appropriate grid points t_{0i}, t_{1i} and the network weights w_i can be set by using a very good approximation of the ODE solution created with the Runge-Kutta method or learned iteratively using some learning rules [8] and stored into a table. When the input for the network is given, the output can be computed very fast in one computing step by determining the proper neuron, i.e. interval, and compute the output by using the proper weight. Therefore, the neural network works like a function lookup-table where its accuracy is determined by the number of table entries.
For the rest of the paper, we will assume a perfectly converged network. This means that errors are only due to the setup of parameters like number of neurons, but not due to a not perfect convergence.

3.1 The Euclidean approximation error

For the approximation with neural networks the maximal error in the whole interval $[t_0, T]$ we get after some computations, dropped because of space restrictions:

$$E_{max}(h) = K_T h^2 \qquad (17)$$

The analytic expression for the error only depends on h. How realistic is this estimation? Let us validate the analytic expression by the observed error $E^{la}(h)$ of linear approximation in a simulation:

$$E^{la}(h) = max\ _{t_0 \leq t \leq T} |y(t) - \tilde{y}_n(t)| \qquad (18)$$

The computation takes the solution of the differential equation which we already got by a Runge-Kutta numerical integration and tests for an interval width h all approximation values within the interval for very small steps $\Delta t_F = 0.0001$. On each step we encounter an error $|y(t) - \tilde{y}_n(t)|$. The maximum of all error values is $E^{la}(h)$.

Let us do this for our example of section 2.2. The height of the ball flight, given by the ODE of (3) and (6), and performed for different intervals $0 < h < 3$, is plotted in *fig.*3. In contrast to this, the maximal observed error $E^{la}(h)$ for a ball with the smaller mass $m = 0.275$ in *fig.*4 shows us that both behaviors are completely different. There is a huge difference between the analytically obtained maximal error and the observed error: The analytically obtained expression for $E_{max}(h)$ in eq. (17) and its actual behavior for h in *fig.*4 are not compatible.

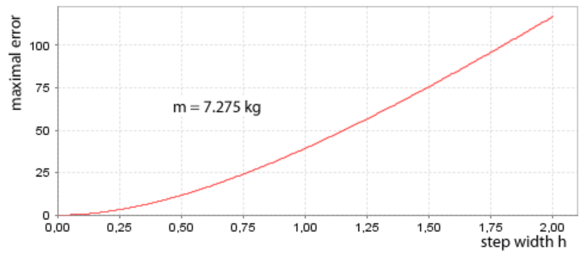

Figure 3. $E^{la}(h)$ for the height of the ball course, eqs.(3),(6) and $m = 7.275kg$

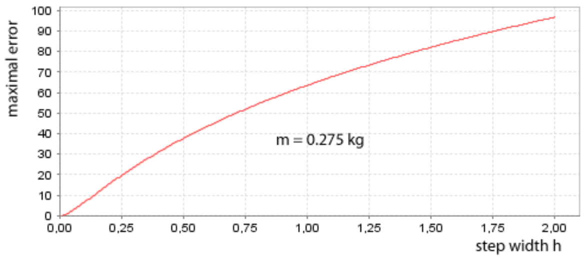

Figure 4. $E^{la}(h)$ for the height of the ball course, eqs.(3), (6) and $m = 0.275kg$

Why? How can we explain this difference? Comparing the actual error $E^{la}(h)$ of a lightweight ball with mass $m = 0.275kg$ to that of a heavy one with $m = 7.275kg$ related to the maximal error $max_{t_0 \leq t \leq T}\ |y_t - 0|$, we remark that the maximal absolute error E^{abs} is obtained already for the lightweight ball for very small step width. The error has nearly obtained its maximum and can not grow exponentially by h any more. In other words, the maximal error $E^{la}(h)$ does not become bigger, independently of all sampling points. The behavior of the observed error changes and is difficult to estimate. Therefore, we have to introduce another way to describe the error of the approximation by neuronal nets.

3.2 The subjective approximation error

In many applications of ODE's, like physics engines for computer games, the shape of the approximation curve perceived by the player is important and not the exactness of the approximation values. This is reflected by introducing another error, let us call it "subjective error", which is the distance between the original curve and the approximation taken perpendicular to the approximation spline, see *fig.* 6. It is build upon the total least mean squared error (TLMSE)[16], also called *eigenvector line fitting* [7]. By this, the distance between a point $\vec{z}$ and our approximation curve can be described as follows:

$$g(\vec{z}) = \vec{z}^T \frac{\vec{u}}{|\vec{u}|} - c = d \quad \textit{Hesse normal form} \tag{19}$$

The vector $\vec{u}$ in our case is the normal vector of our approximation curve with

$$\vec{u} = \begin{pmatrix} 1 \\ -\frac{y(t_{n+1})-y(t_n)}{t_{n+1}-t_n} \end{pmatrix} \tag{20}$$

For the distance c between the approximation curve and its origin we get

$$c = \vec{S}_{n+1}^{\,T} \frac{\vec{u}}{|\vec{u}|} \tag{21}$$

with $\vec{S}_{n+1} = \begin{pmatrix} t_{n+1} \\ y(t_{n+1}) \end{pmatrix}$ as interpolation point vector.

Using eqs.(19) and (21) we can describe the difference d between a point $\vec{z}$ and the approximation as

$$d = \vec{z}^{\,T} \frac{\vec{u}}{|\vec{u}|} - \vec{S}_{n+1}^{\,T} \frac{\vec{u}}{|\vec{u}|} \tag{22}$$

Now we know to calculate the difference between a point and our approximation. With these results we can define the maximal error between two interpolation points as

$$E_{SE}(\vec{S}_n, \vec{S}_{n+1}) = \max |d| \quad \textit{subjective error} \tag{23}$$

over all

$$\vec{z}^{\,T} \in \left[\begin{pmatrix} t \\ y(t) \end{pmatrix} \mid t_{n,1} \le t \le t_{n+1,1} \right] \tag{24}$$

and for the whole interval T as

$$E_{SE}^{la}(h) = \max \; E_{SE}(\vec{S}_n, \vec{S}_{n+1}) \tag{25}$$

Contrary to the standard mean error computed in the previous section in *fig.* 4 this time the error behavior for low masses is consistent to those with higher mass, see *fig.* 5. It should be noted that both axes had to be normalized before computing the error. Without this precaution, the Eigenvector depends on the scaling and small changes makes the total error oscillate.

After computing the subjective error for the neural network we might also use this idea for the case of Euler approximation. In this case, the maximal subjective error is computed over all samples and corresponds to the global error of eq.(13)

$$E_{SE}^{Euler}(h) = \max \; E_{SE}(\vec{S}_n^{Euler}, \vec{S}_{n+1}^{Euler}) \tag{26}$$

using

$$\vec{u} = \begin{pmatrix} 1 \\ -\frac{\tilde{y}_{n+1}(t_{n+1}-t_n)-\tilde{y}_n(t_{n+1}-t_n)}{t_{n+1}-t_n} \end{pmatrix}$$

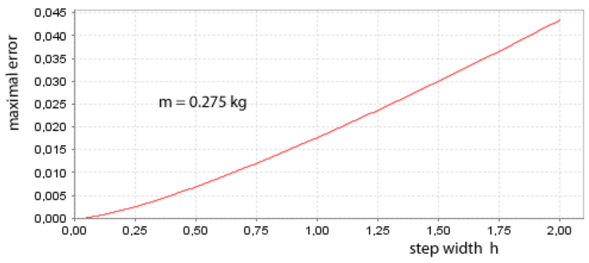

Figure 5. $E_{SE}^{la}(h)$ for the height of the ball course, eqs.(3),(6) with $m = 0.275kg$

and

$$\vec{S}_{n+1}^{Euler} = \begin{pmatrix} t_{n+1} \\ \tilde{y}_{n+1}(t_{n+1} - t_n) \end{pmatrix}$$

3.3 The storage-optimal neural network

Up to now, we assumed that the table of approximation values for the neural network provide numbers of unlimited precision. This is not realistic. If we choose all numbers to be of low resolution, we can enlarge the function lookup table and get more interpolation points with the same amount of storage. This might also decrease the approximation error, depending on the resolution. Therefore, we have to reflect that the error E^{max} does not only depend on the exactness of the approximation, but also on the numerical resolution of the interpolation points which contributes a resolution error E^{res} to the general error

$$E^{max} = E_{SE}^{la}(h) + E^{res} \tag{27}$$

The resolution error depends on the storage which is used for representing the numbers. Therefore, before comparing the ODE performance with that of the neural network, we have to optimize the storage requirements of the network. For this, we will first calculate the maximal error for our example ODE and then find the optimal number and the resolution of the interpolation points for a given amount of storage. There is also one important point to mention. The subjective error is computed from the correct value perpendicular to the approximation, but the approximated value is computed along the ordinate axis. Thus, the resolution error is not computed in the same direction as the subjective error, see *fig.* 6. Nevertheless, we might take the upper bound of this error which is, according to the inequality of Schwarz for distance measures, just the sum of both errors presented in eq.(27).

Let us first evaluate the first term in eq.(27), the approximation error. For simplicity, let us take the resolution of the interpolation values $\tilde{y}(t_n)$ in the direction of the standard error, not in the changing direction of the subjective error.

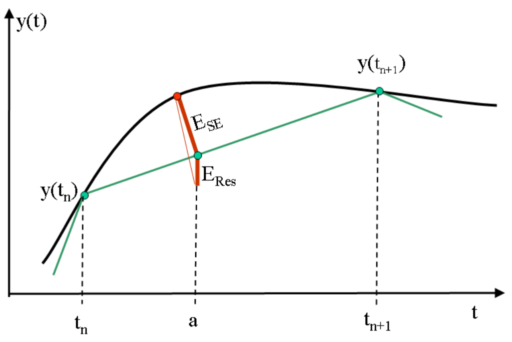

Figure 6. The maximal error is smaller than the sum of subjective and resolution error

Then, if we logarithmize the observed error of the simulation, our example ODE with eqs. (3, 6), we remark in *fig.* 7 that it depends approximately linearly on the logarithm of h:

$$ln(E^{la}_{SE}(h)) \sim ln(h) \tag{28}$$

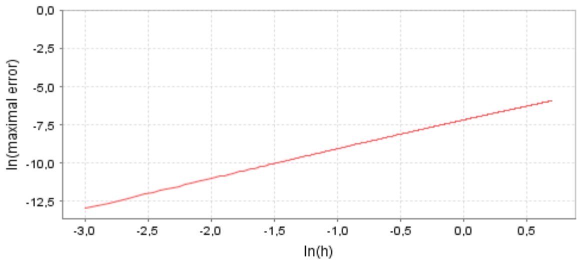

Figure 7. The logarithm of the maximal error vs. the logarithm of the step width

Additionally, in *fig.* 7 we can see that the proportional constant is approximately two and the line has a bias of roughly -7.5. Using this information, we render eq. (28) more precisely

$$ln(E^{la}_{SE}(h)) = 2ln(h) - 7.5 \Longrightarrow E^{la}_{SE}(h) = h^2 e^{-7.5} \tag{29}$$

Now, let us evaluate the resolution error E^{res}. Let us assume that the resolution error gets halved with every additional bit. For a given resolution of b bits, this means

$$E^{res} \sim 2^{-b} \tag{30}$$

The whole value range of our approximation is divided into 2^b intervals. Let the range be represented by an original resolution of, say, 16 bits. So, we have $2^{16}/2^b$ intervals to represent a number. The average error for uniformly distributed values is just half of one resolution interval. So, we get as resolution error from eq.(30)

$$E^{res} = \frac{2^{16}}{2^b 2} = 2^{-(b-15)} \tag{31}$$

Using equations (29) and (31) in eq.(27) results in

$$E^{max} = h^2 e^{-7.5} + 2^{-(b-15)} \tag{32}$$

What are the associated storage requirements of the neural network approximation for this error? The storage usage M of a linear approximation depends within the whole approximation interval T by the step width h or the number of interpolation points $S = T/h$ and the resolution of b bits for a number representation of an interpolation point: $M = S \cdot b$ *[Bit]*. By reformulating this for b we get

$$b = \frac{M \cdot h}{T} \tag{33}$$

and therefore

$$E^{max} = h^2 e^{-7.5} + 2^{-(\frac{M\,h}{T} - 15)} \tag{34}$$

The smallest error is obtained by a network which distributes its storage between resolution and number of interpolation points in such a way that no change in neither direction will decrease the error any more (*principle of optimal information distribution*, see [4]). The optimal information distribution is reached, when

$$\frac{\partial E^{max}}{\partial h} = \frac{\partial(h^2 e^{-7.5} + 2^{-(\frac{Mh}{T} - 15)})}{\partial h} = 0 \tag{35}$$

$$\begin{aligned} \frac{\partial E^{max}}{\partial h} &= 2he^{-7.5} \\ &+ \left(-\frac{M}{T} ln(2) - (\frac{Mh}{T} - 15)\right) 2^{-(\frac{Mh}{T} - 15)}) = 0 \end{aligned} \tag{36}$$

Solving eqs. (35) and (36) for $M = 30$ kBit and $T = 60$ seconds by a symbol manipulation program (e.g. Maple) gives us the optimal positive inter-point distance and resolution $h^{opt} = 0.07419$ and $b^{opt} = 37.0988$.

The optimal storage distribution for a given amount of storage determines the error of the network approximation. Now, we have all elements to compare the performance of a standard numerical ODE approximation with that of neural network approximation.

4 Comparing the resources

In section 2 we have evaluated the error made by a simple numerical integration method, the Euler approximation, which depends on the integration step width h. Afterward, the evaluation of a simple neural network approximation scheme also gives us an approximation error, depending not only on the interval length (step width) h for different kind of errors, but also on the numerical resolution of the network interpolation points (or weights) b. The storage optimization in section 3 gave us for each error an optimal value for the necessary storage needed.

4.1 The resources of the Euler approximation

The computational needs of the Euler approximation are high: for each step of eq.(10) with width h within the interval to the desired time point T, we have one computation of the function (9) in constant time τ and one multiplication and one addition. The overall computing time is characterized by the number of computation steps and becomes with S interpolation points

$$R_{CPU}^{Euler} = S(2+\tau) = \frac{T}{h}(2+\tau) \tag{37}$$

The smaller we choose the step width h for a small error, the more computing time we have to spend: They are inversely proportional. It should be noted that for our example eqs.(3) and (6) the constant τ is much bigger than one computational time step: the transcendental functions take a long time to be computed since they are approximated by truncated Taylor series themselves.

Additionally, each step width h determines an error of the approximation. Following the approach of eq.(28) we plot the dependency

$$ln(E_{SE}^{Euler}(h)) \sim ln(h) \tag{38}$$

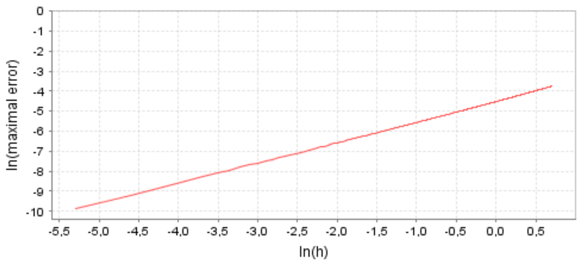

Figure 8. The logarithm of the maximal error vs. the logarithm of the step width

Also, in *fig.* 8 we can see that the proportional constant is approximately one and the line has a bias of roughly -4.5. Using this information, we render eq. (38) more precisely

$$ln(E_{SE}^{la}(h)) = ln(h) - 4.5 \Longrightarrow E_{SE}^{la}(h) = he^{-4.5} \tag{39}$$

Since the subjective error is proportional to step width h, the computing resources R_{CPU}^{Euler} are also inversely proportional to the subjective error. Contrary to this, the storage requirements are only modest: Since we store the approximated value of one approximation step as starting value of the next one, all what we have to store is just this value. Additionally, the constants of the formula (9) (in our case: v_0, v_∞, g) and the step width h. The number of memory units (e.g.floating point numbers) is independent of the step width h

$$R_M^{Euler} = const = 5 \tag{40}$$

It should be remarked that the Runge-Kutta method does not change the scene much: both qualitative arguments of high computational resources and low memory needs are valid here, too.

4.2 The resources of the neural network approximation

In contrast to the Euler approximation, the computational needs of the neural network approximation are much more decent. The approximation by eq.(16) is made by three subtractions, one addition and one multiplication beside the fast lookup of the values of $y(t)$ in the table.

$$R_{CPU}^{Net} = const = 7 \tag{41}$$

According to eqs.(33) and (34) the storage resource $R_M = M$ and the optimal step width h^{opt} determine the optimal resolution b^{opt} and the resulting error E^{max}. Therefore, we can compute how the storage resource R_M depends on the maximal subjective error.
If we combine storage R_M^{Net} and computation resources R_{CPU}^{Euler} of both methods for the same error in one plot, we get *fig.*9. Each configuration point $(R_M^{Net}, R_{CPU}^{Euler})$ is associated with the same error level $E^{max}(R_M^{Net})$ and $E^{max}(R_{CPU}^{Euler})$ where the network approximation uses constant computations and the Euler approximation uses constant memory. The set of all configuration points is shown as curve in the figure: The lower the resources, the higher the error E^{max}.

Now, which method should we choose if we have a computer with the computation resource R_{CPU}, the storage R_M and an acceptable error E^{max}? The resource data are drawn as dotted lines within the figure, one for the storage in parallel to the CPU axis and one in parallel to the storage axis. The crossing of each line with the curve of the configuration set determines the error using the associated resource.
The decision tries to minimize the error obtained by the approximation method. So, we have to take the approximation method which has a crossing with the highest distance (the lowest error) to the origin. In the example, the configuration of the Euler method has an advantage over the neural

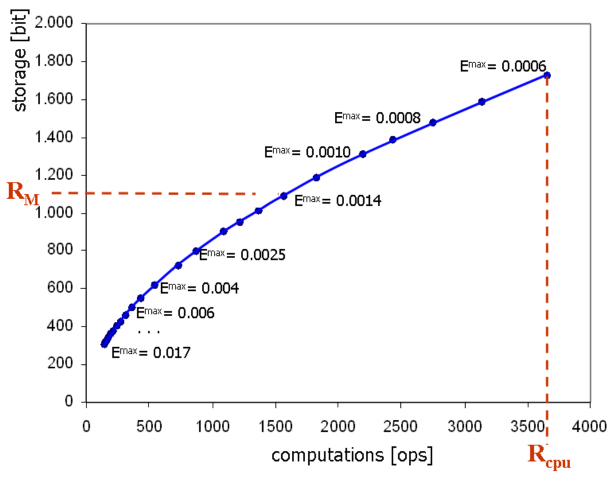

Figure 9. The resources R_M^{Net} vs. R_{CPU}^{Euler}.

network method, but when the storage resources are augmented, the neural network configuration may produce a smaller error. Thus, the decision for one of the two methods depend on the given resources and may even change in time.

5 Discussion and conclusion

In this paper we compared characterized the performance of approximating ordinary differential equations by neural networks implemented by fast lookup-tables. Here, cheap memory resources may be advantageous over high computational needs of the standard approach.

Although our comparison is made just for one representative example of rigid body movement, we can see some typical problems and features of both methods, the standard numerical integration for ODE, and neural networks. Certainly, there are more sophisticated approximation schemes for ODE like Runge-Kutta which uses three interpolation samples instead of two like Euler, or even more accurate ones using more than three points, but the main characteristics remain: the solutions depend highly on the interpolation distance and, with small distances h, are very computational intensive.

Additionally, for the neural network there are many more complicated approximation schemes known in the literature. The more general they are (e.g. by using nonlinear splines or kernel functions) the more complicated the analysis becomes. Even in the simple linear scheme, we have seen that the analytical expression are only valid for some cases and yield good results only in the limit.

Nevertheless, our work has shown that it is possible to compare the two approaches of standard numerical integration of ODE and the alternative approach of optimized, information balanced neural networks and find a strategy of taking the best system according to the actual machine resources.

References

[1] V. Arnold. *Mathematical Methods of Classical Mechanics*. Springer Verlag, Berlin, Germany, 1989.

[2] D. Bourg. *Physics for Game Developers*. O'Reilly, Sebastopol CA 95472 USA, 2001.

[3] R. Brause. Performance and storage requirements of topology-conserving maps for robot manipulator control. *Internal report, JWG-Universitt Frankfurt a.M.*, 1989.

[4] R. Brause. The error-bounded descriptional complexity of approximation networks. *Neural Networks*, 6:177–187, 1993.

[5] R. Brause. Adaptive modeling of biochemical pathways. *IEEE 15th Int. Conf on Tools with Art. Intell. ICTAI*, IEEE Press:62–68, 2003.

[6] J. Choi, D. Shin, W. Heo, and D. Shin. Performance evaluation of numerical integration methods in the physics engine. *Proc. First International Symposium*, LNCS 3314, Springer-Verlag:238–244, 2004.

[7] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, New York, 1973.

[8] S. Haykin. *Neural networks, a comprehensive foundation*. Prentice Hall, New York, 1998.

[9] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks, Perg. Press*, 2:359–366, 1989.

[10] P. Kloeden. Numerical methods for differential equations. see http://www.math.uni-frankfurt.de/˜numerik/lehre/Vorlesungen/NMDE/, Accessed June 1st 2008.

[11] I. Lagaris, A. Likas, and D. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987 – 1000, 1998.

[12] H. Lee and I. Kang. Neural algorithms for solving differential equations. *J. Computational Physics*, 91:110–117, 1990.

[13] A. Meade and A. Fernadez. Solution of nonlinear ordinary differential equations by feedforward neural networks. *Math. Comput. Modelling*, 20(9):19–44, 1994.

[14] A. Polyanin, V. Zaitsev, and A. Moussiaux. *Handbook of First Order Partial Differential Equations*. Taylor & Francis, London, 2002.

[15] H. Schwarz and J. Waldvogel. *Numerical Analysis*. John Wiley, London, 1989.

[16] L. Xu, E. Oja, and C. Suen. Modified hebbian learning for curve and surface fitting. *Neural Networks*, 5:441–457, 1992.

[17] R. Yentis and M. Zaghoul. VLSI implementation of locally connected neural networks for solving partial differential equations. *IEEE Trans. Circuits and Systems-I*, 43(8):687–690, 1996.