Modelling flow-induced vibrations of gates in hydraulic structures

Erdbrink, C.D.

Publication date
2014

9 Inferring numerical algorithms[8]

9.1 Introduction

An obvious drawback of the derivation of ODEs in Section 8.4 is that candidate models need to be solved numerically in order to be evaluated. It was chosen to compute errors with the original target data series $y(t)$ and the predicted time series $\hat{y}(t)$. For a first-order ODE $\dot{y} = f(t, y)$ one could evolve the right-hand side and compute errors by comparison with the derivative of the target series. Reducing a n-th-order ODEs to a system of first-order ODEs, the n-th derivative of the target data is needed. Differentiation of the original time series – especially when done more than once– is not a trivial task due to the noise present in real-life data (see Kronberger 2011). In the previous chapter's study the velocity was required only at the time of the first training point as initial value.

Other approaches assume structures that are easily computable, such as the discrete map used by Howard and Oakley (1994), but these often provide less insight in the system because the resulting expression is unfamiliar and therefore hard to interpret. In an attempt to have the best of both worlds, one interesting alternative[9] is to perform SR on discrete equations and afterwards derive the continuous-form ODE from the discrete expression. Continuing with the example of finding a numerical solution to a second-order ODE, this corresponds to the search for a discretised version of an ODE that lives in the solver domain as in Figure 9.1.

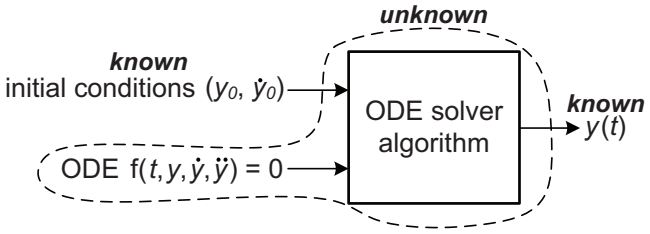

Figure 9.1. Inferring ODEs at the discrete level. The part enclosed by the dashed line represents unknown discrete formulae which are solved by evolutionary computing and afterwards converted to the ODE form.

The central idea that makes this work is that the continuous form, i.e. the right-hand side of $\dot{y} = f(t, y)$, is always derivable from the recurrence relation form by differentiation with respect to the timestep h and taking the limit $h \to 0$. Basic integration methods for solving

[8] Parts of this chapter are based on the manuscript "Evolutionary design of numerical methods: generating finite difference and integration schemes by differential evolution" by C.D. Erdbrink, V.V. Krzhizhanovskaya and P.M.A. Sloot.

[9] This idea was put forward in a presentation by Maarten Keijzer during the symbolic regression workshop of GECCO 2013, the main conference on evolutionary computing. At that time (July 2013) it had not appeared in publications and so far no publications were found that elaborate on this approach.

initial value problems (Euler, Heun, etc.) all have the form $y_n = y_{n-1} + F(h, t_n, y_n)$, for a certain function *F*. Suppose an expression like this is derived through evolutionary computing, then $f(t, y)$ is found from $\lim_{h \to o} dF/dh$. This could provide new computational possibilities, provided that both the variation operations and fitness evaluation of the discrete form can be programmed efficiently.

9.2 Inferring solver algorithms

9.2.1 Introduction

The ODE solution process can have yet another unknown: the solver algorithm itself. See Figure 9.2. The new idea introduced in this chapter is to use evolutionary computing to find formulae or 'schemes' for solving numerical problems. The concept of contriving equations without human intervention is thus applied to numerical mathematics itself. This is a reverse modelling problem where the ODE, the initial conditions and the solution are assumed to be known.

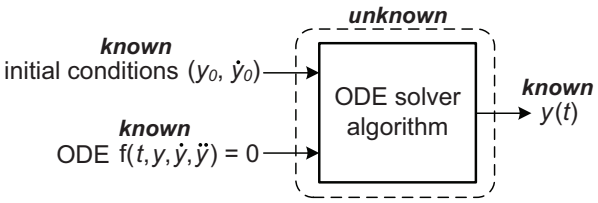

Figure 9.2. Finding a solver algorithm for an ODE with a known solution. This is a modelling problem, where the structure and/or the coefficients of the numerical solver scheme may be unknown.

In this section, coefficients of classical numerical schemes for differentiation and integration are generated by an evolutionary algorithm (EA). EAs generally become interesting when approximate solutions are acceptable and small improvements over existing solutions are extremely valuable. In addition, it is required that testing of the quality of candidate solutions should be unambiguous. The problem at hand, finding discrete computational schemes for estimating derivatives and integrals, meets these conditions: inexact schemes are acceptable as the schemes themselves represent approximations, and the performance of a scheme is easily checked by applying it to a function with known properties. The widespread use and therefore importance of finite difference methods and Runge-Kutta methods is self-evident. Basic finite difference schemes have straightforward analytical derivations, but complex computational science problems often require the design of more sophisticated methods lacking standard approaches.

Runge-Kutta methods comprise a family of integration schemes for solving ODE initial value problems, the most famous being the fourth-order (RK4) Runge-Kutta method. For orders past two the systems of order condition equations are underdetermined, this hampers analytical derivation. Tsitouras and Famelis (2012) note that optimization methods based on computing Jacobians are less suitable for finding Runge-Kutta schemes than EAs, since determining the derivatives of the variables as they appear in the order conditions is cumbersome.

For the modelling task of generating both a scheme's structure as well as its coefficients, the parse tree method of GP provides an appropriate approach. However, if the equation structure is given and the goal is to find only the coefficients, it becomes an optimization problem and other EAs can be used. Here, again differential evolution (DE) will be used. The power of DE is reflected by outperforming a number of stochastic optimization algorithms such as Adaptive Simulated Annealing (Storn and Price, 1997) and Particle Swarm Optimization (Vesterstrøm and Thomsen, 2004); it beats Genetic Algorithms (GA) on many benchmarks as well (Tušar and Filipič, 2007; Hegerty et al., 2009).

Previously, the application of EAs to evolve computational algorithms was mostly aimed at solving domain-specific problems. For example, Spector et al. (1998) used GP to find quantum computing algorithms with superior performance. A number of past studies utilize EAs in more general contexts to solve systems of equations or somehow tune traditional numerical methods to improve performance. For example, He et al. (2000) propose a hybrid algorithm that combines the classical Successive Over-Relaxation method (SOR) for solving linear systems of equations with an evolutionary technique to evolve the relaxation factor.

A study by BaniHani (2007) uses GA to determine points and weights for an integration procedure within a meshfree method for solving boundary value problems. To the best of our knowledge, Martino and Nicosia (2012), Tsitouras and Famelis (2012) and Nanayakkara et al. (1999) are the only efforts so far to apply an evolutionary algorithm to derive Runge-Kutta-related methods. The first study advocates the use of EAs in numerical analysis from the viewpoint of algebraic geometry, but does not present concrete values of coefficients, nor does it discuss accuracy. The second study considers neural networks and DE to find Runge-Kutta-Nyström pairs for solving a second order ODE problem occurring in astronomy. The third study uses an EA with the purpose of finding shape parameters of Runge-Kutta-Gill neural networks, through radial basis function networks, for the identification of robot arm dynamics.

In the field of finite difference methods, an effort by Haeri and Kim (2013) uses GA to optimize boundary characteristics of compact finite difference equations. These recent studies illustrate the relevance of applying EAs to find involved numerical recipes and provide the motivation for our research, where we apply DE for deriving general, i.e. domain-independent numerical schemes.

The aim of this chapter's study is to apply differential evolution to find coefficient sets of classical numerical schemes for approximating derivatives and integrals. This will not only produce new coefficients for schemes where fully analytical derivation of coefficients is unattainable, but the reverse-engineering process will also yield insight into how to apply the evolutionary heuristic and what the resulting accuracy is. This paves the way for the application of EAs to more complex numerical stencils of general use in applied mathematics and engineering.

The next subsection treats related work. Then a background on the studied numerical schemes is given and it is explained how the coefficients will be represented. This is followed by a description of the computational method used in this study. Then the modelling results are presented; this subsection consists of a sensitivity analysis and presentation and

discussion of training and validation results. In Section 9.3, conclusions are drawn and 9.4 gives a reflection on this chapter and the previous one with an outlook on future work.

9.2.2 Background

1. Finite difference derivative estimates

Common finite difference formulae are of the type central, forward or backward. Table 9.1 gives two examples, viz. the second-order central formula and the second-order forward formula for estimating the first-order derivative.

Table 9.1. Representation of finite difference schemes.

finite difference formula	vector representation
$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$	$(1/2 \quad 0 \quad -1/2)^{\mathrm{T}}$
$f'(x) \approx \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h}$	$(-3/2 \quad 2 \quad -1/2)^{\mathrm{T}}$

Here h is the step size. By putting all coefficients in the numerator (thus leaving only h in the denominator) each approximation scheme is defined by its coefficients $m_i \in \mathbb{Q}$ for all terms $m_i \cdot f(x + n_i h)$ with $n_i \in \mathbb{Z}$. The number of terms grows with increasing order of accuracy and the coefficients can, among other ways, be determined via the Taylor series. Note additionally that the same coefficients used in a forward scheme can also be used in an equivalent backward scheme with symmetric terms $f(x \pm h)$. For even order derivatives, the coefficients are equal for corresponding terms, but for odd order derivatives the coefficients are multiplied by -1.

A central scheme of order p contains p nonzero terms, where p is even. The p coefficients of a central scheme will be searched for by evolving vectors of sizes p and $p+1$. The formula's skeleton assumed for runs with vectors of size p consists of terms $f(x \pm ph)$ with q = 1, 2,.., $p/2$ and vectors of size $p+1$ contain the extra term $f(x)$. A forward scheme of order p contains $p+1$ nonzero terms and will be represented only by vectors of size $p+1$. For the forward scheme the skeleton consists of the terms $f(x + qh)$ with q = 0, 1,.., p.

2. Runge-Kutta schemes

Runge-Kutta schemes for solving initial value problems are summarized in Butcher tableaus or arrays (Butcher 2008). Table 9.2 shows this notation and also gives the corresponding vector representation that is used in our study.

Table 9.2. Representation of explicit Runge-Kutta schemes.

Butcher tableau						**vector representation of explicit scheme**
c_1	a_{11}	a_{12}	…	…	a_{1s}	
c_2	a_{21}	a_{22}	…	…	a_{2s}	
⋮	⋮	⋮	⋮	⋮	⋮	$(a_{21}\ a_{31}\ a_{32}\ a_{41}\ \dots\ a_{s1}..a_{ss-1}\ w_1 \dots w_s)^T$
c_s	a_{s1}	a_{s2}	…	a_{ss-1}	a_{ss}	
	w_1	w_2	…	…	w_s	

Where c_i are called the nodes, w_i are the weights and s is the stage. The Butcher tableau holds the coefficients for the formula

$$y_n = y_{n-1} + \sum_{i=1}^{s} w_i k_i$$

$$\text{with } k_i = hf\left(t_n + c_i h; y_n + \sum_{j=1}^{s} a_{ij} k_j\right). \tag{9.1}$$

For explicit schemes the matrix A containing entries a_{ij} is lower triangular, i.e. $a_{ij} = 0$ for $i \leq j$. Furthermore, the consistency condition requires that the sum of the row elements in A is equal to the node at that level: $a_{i1} + \cdots + a_{is} = c_i,\ \forall i$ (Hairer et al. 1993). Assuming consistency therefore implies that explicit schemes can be encoded by the vector given in Table 9.2. For instance, a 4-stage scheme is represented by the vector $(a_{21}\ a_{31}\ a_{32}\ a_{41}\ a_{42}\ a_{43}\ w_1\ w_2\ w_3\ w_4)^T$.

The stage s of a scheme fixes its maximum order of accuracy. Order condition equations determine if an order is actually achieved (Butcher 2008). For example, a 2-stage scheme (s = 2) has second order accuracy if $w_1 + w_2 = 1$ and $w_2 a_{21} = ½$. The order conditions up to order five are included in Appendix I. Note that the 2-stage scheme has easy closure: choose a value for a_{21} and the other coefficients are fixed. By contrast, higher order schemes are far more demanding; 6-stage schemes need to meet 17 order conditions to attain order 5. This study considers 3-, 4- and 6-stage schemes, reaching a maximum order of 5.

3. Adams-Bashforth schemes

The Adams-Bashforth method is an explicit linear multi-step integration method (Butcher 2008). Depending on the target order of accuracy, a certain number of preceding prediction values are used in the new estimate. The general formula for a k-term method reads

$$y_n = y_{n-1} + h \sum_{i=1}^{k} \beta_i f(t_{n-i}, y_{n-i}). \tag{9.2}$$

The maximum achievable accuracy order of a k-term method is k. The order conditions can be used to find analytical values of the coefficients. The coefficients representation is straightforward for the Adams-Bashforth schemes: each vector contains all β_i for $1 \leq i \leq k$, for computing a scheme of intended order k.

9.2.3 Method for generating computational schemes by an evolutionary algorithm

The original DE algorithm, called DE/rand/1/bin (Storn and Price, 1997), assumes constant strategy or 'control' parameters CR for crossover and scaling factor F for mutation. With the advent of dynamic parameters adjustment (Eiben et al. 1999), later versions of DE have control parameters that are varied as part of the evolution (Brest et al. 2006). Our study is based on DE/rand/1/bin and adopts the self-adaptive method by Choi et al. (2013) where each individual carries its own pair of control parameters. Updates take place every generation after the selection by taking the average values of the control parameters of individuals that evolved successfully in the past generation and then adding variation to these averages by drawing from a Cauchy distribution, see the pseudo-code in Chapter 8.

The initial population consists of floating-point vectors with uniformly random entries between -1 and 1, but no value constraints are imposed during the evolution. In every generation five randomly chosen individuals (other than the highest scoring individual) are replaced by completely new individuals, a measure intended to ensure a healthy balance between population diversity and exploitation. Admittedly, later tests showed that this measure had marginal effect on diversity because the new individuals quickly adapt to the environment. The algorithm has four key model parameters: CR_0, F_0, population size NP and the number of computed generations, where the zero indices denote the initial values of the control parameters applied to all individuals. Because of the random nature of the self-adaptation, these initial values have negligible influence on the result of the evolution; this study uses $CR_0 = 0.25$ and $F_0 = 0.6$. The population size and number of computed generations will be determined in a sensitivity study.

1. Finite difference derivative estimates

Training consists of comparing first order derivative estimates calculated by the candidate formula with analytical values sampled from a target derivative function. A bell-shaped curve and its derivative are used as target function, see Figure 9.3.

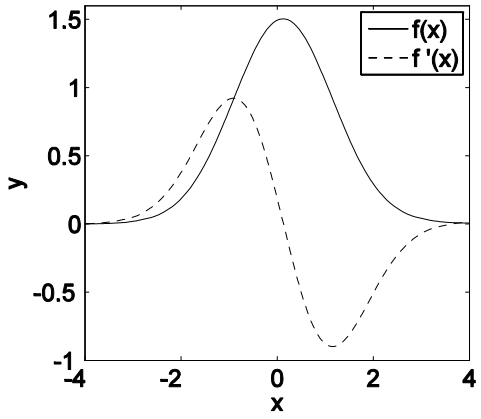

Figure 9.3. Function pair used for training coefficients of finite difference schemes and Adams-Bashforth schemes.

The target function pair of Figure 9.3 is $f(x) = 1.5e^{-0.5x^2}$ and $f'(x) = -1.5xe^{-0.5x^2}$, for -4 < x < 4, with h = 0.01. The fitness function measures the absolute errors between candidate values and analytical coefficient values:

$$F_{FD} = \log \sum_{k=1}^{n} |v_k - v_k^*| \tag{9.3}$$

Where $\boldsymbol{v} \in \mathbb{R}^{\boldsymbol{n}}$ is the vector containing the theoretical values, $\boldsymbol{v^*} \in \mathbb{R}^{\boldsymbol{n}}$ is a candidate vector and $n = \dim_{\mathbb{R}}(\mathbb{R}^{\boldsymbol{n}})$. The logarithm is applied to deal with the large range of values. In this chapter the evolution seeks to minimize the fitness functions.

2. Runge-Kutta schemes

The candidate coefficient vectors are evaluated by substitution into the order condition equations. The fitness of the candidate Runge-Kutta schemes is therefore defined by

$$F_{RK} = \log \sum_{k=1}^{\#C} |C_k - C_k^*| \tag{9.4}$$

where $\boldsymbol{C_k}$ is the analytical value of order condition equation k; these are the right-hand-side values in Appendix I. $\boldsymbol{C_k^*}$ is the candidate order conditions vector computed by inserting all necessary entries of the candidate coefficients vector into the order condition equations. $\#C$ is the number of order conditions.

3. Adams-Bashforth schemes

Fitness evaluation is defined identically to the finite difference schemes, but now the candidate scheme is evaluated by integrating f' (f as defined in Figure 9.3) and the result is compared with the analytical values sampled from f. The necessary starting points are

generated by a Runge-Kutta scheme of the same order of the Adams-Bashforth scheme that is aimed for.

9.2.4 Results and discussion

Sensitivity analysis

Before starting the training process of the finite difference runs the impact of the number of generations and the population size are considered, as well as the characteristics of the target function. The target function pair was chosen in such a way that the derivative has both positive and negative values. Preliminary runs for the sixth order central difference scheme showed that the exact shape (periodicity, number of extremes, symmetry, etc.) of target function f has no notable influence on accuracy, as long as f' is continuous and has a reasonable variation such that the fitness can discriminate between good and bad approximations. Clearly, the function pair $f(x) = 4x + 7, f'(x) = 4$ would be unsuitable.

Varying the range of function f (Figure 9.3) between 0.1 and 1000 for an equal sampling step size h and equal number of training points, giving a variation in the range of f' between 0.4 and 4850, yielded no differences in achieved accuracy. Next, the step size was varied while keeping the total number of training points the same. This also had no noticeable impact on accuracy. Figure 9.4 shows the results when the reverse is done: varying the number of training points while keeping h constant, for three different values of h.

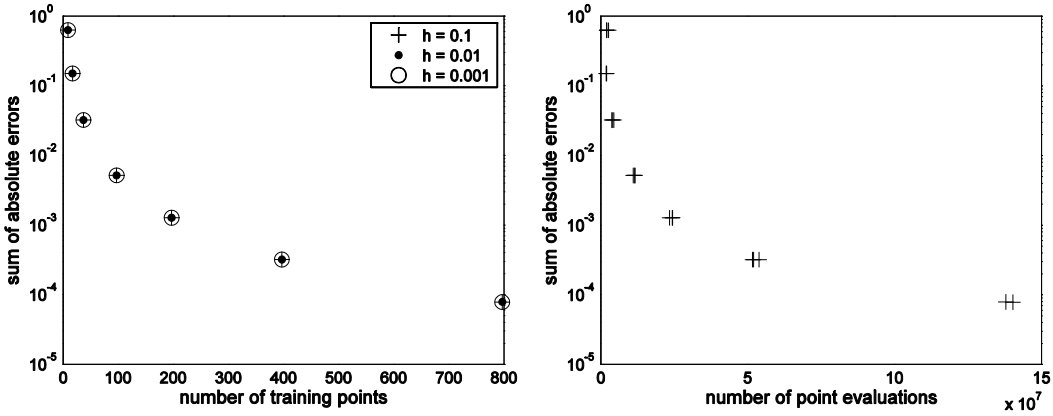

Figure 9.4. Sensitivity to training data for the central sixth order approximation of a first order derivative. Left: varying number of training points in the target function while keeping sample step size h constant, for three different step sizes. Right: sum of absolute errors as function of the number of point evaluations. The two plots are based on the same 21 runs with a population size of 150. On the vertical axes the sum of absolute errors over all coefficients is plotted on a log-scale.

Figure 9.4 (left) shows that the number of training points, i.e. the number of unique points sampled from the target function, is a major factor for accuracy. Additionally, it underlines the fact that the step size at which the target function is sampled has no influence at all. The results also show that the improvements in accuracy are getting smaller as we increase the number of training points (N): the sum of errors gets 1000 times lower with N increasing from 1 to 200; and only 10 times lower with N increasing from 200 to 800.

The number of point evaluations is the product of the number of computed generations, population size and number of calls to training points. The sum of absolute errors as a function of number of point evaluations (Figure 9.4 right) displays a similar relation as for the dependence on the number of training points, which shows that using more training points does not imply significantly more generations needed to attain convergence. Comparing the two plots of Figure 9.4, we conclude that runs with a similar number of training points and a different step size achieve the same accuracy (left plot) and need only a slightly different number of generations for convergence, resulting in small variations in the number of point evaluations per run (right plot).

The computation time of a run depends linearly on the number of point evaluations and therefore also approximately linearly on the number of training points. The plots in Figure 9.4 suggest that improving the accuracy by including more than 800 training points comes with disproportionally higher computational costs. The plotted trial run for the sixth order central scheme with 800 training points consisted of about $1.4 \cdot 10^{8}$ point evaluations and resulted in a sum of absolute errors of $7.8 \cdot 10^{-5}$.

The impact of using different population sizes is investigated by making 25 test runs for the finite difference schemes (based on 200 training points and varying between 50 and 1000 individuals) and for the Runge-Kutta schemes (using the 6-stage scheme and varying between 50 and 500 individuals). The results are plotted in Figure 9.5.

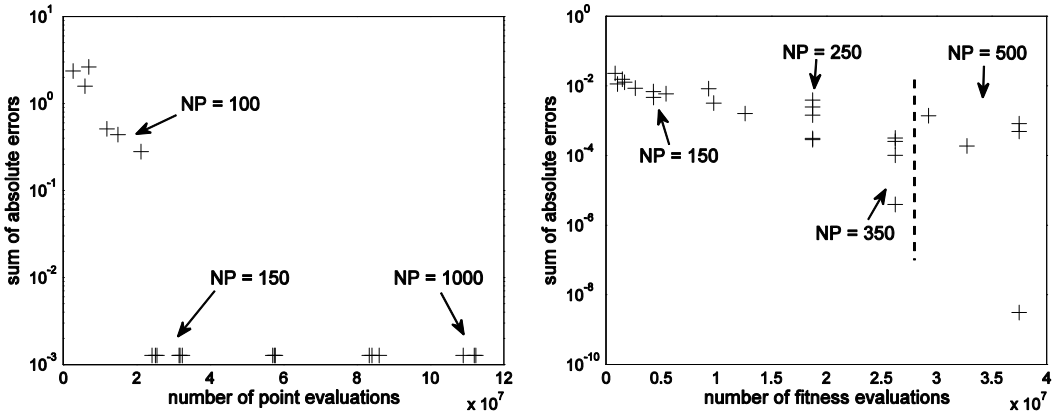

Figure 9.5. Sensitivity to population size. Left: finite difference sixth-order central approximation of first order derivative, changing population size NP from 50 to 1000 and keeping number of training points constant at 200. Right: 6-stage Runge-Kutta runs, varying population size NP between 50 and 500. The sum of absolute errors is plotted vertically on log-scale.

The sensitivity to the population size is quite obvious for the finite difference schemes (Figure 9.5 left): population sizes of 150 or more outperform all smaller population sizes. For population sizes smaller than or equal to 100, the sum of absolute training errors is significant – these schemes never give accurate derivative approximations. Improvements by using populations beyond 150 are marginal compared to the increase in computational costs. Averaged over three runs per population size, the training error for a population of 1000 is

less than 0.1% smaller than for a population of 150, but the computation time is a factor 4.4 higher. Using a population of 150 is therefore optimal for the finite difference runs.

The population size has a different effect on the evolution of Runge-Kutta schemes (Figure 9.5 right). While winning individuals become exponentially better with higher population size, the standard deviation of the attained accuracies also increases with number of fitness evaluations. This means that more runs are required to improve accuracy and at the same time that each run becomes more expensive. Based on this sensitivity, a population size of 350 is chosen for the Runge-Kutta schemes, in combination with a high number of runs.

The number of generations that make up one run is regulated by a termination criterion. The evolutionary algorithm stops if the fitness of the best individual has not changed in a pre-set number of consecutive generations or if a maximum number of generations is reached. Both parameters were determined empirically throughout the sensitivity runs by looking at the convergence of the runs. Figure 9.6 illustrates the working of the termination criterion for the 6-stage Runge-Kutta scheme. The 'example run' never actually converges, as it regularly makes very small improvements but does not escape local optima, it stopped because the maximum number of generations was reached. The 'winning run' experiences a few considerable jumps before converging.

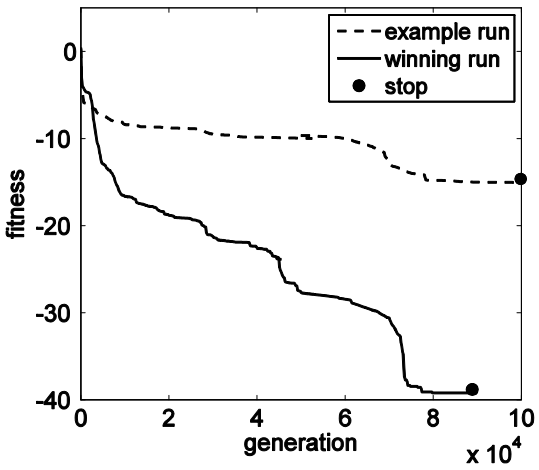

Figure 9.6. Evolution of Runge-Kutta 6-stage scheme, showing the fitnesses as a function of generations for a relatively poor example run and for the winning run (out of 100 runs). Termination points are indicated.

The plot in Figure 9.6 also clarifies that this particular problem demands a large number of generations. The sensitivity runs for Runge-Kutta were done with a maximum of 75,000 generations and a stop after 7,500 generations without improving the best fitness score. This was extended to respectively 100,000 and 10,000 generations to ensure convergence for the more successful runs. However, for the lower order schemes convergence proved to occur much sooner. Table 9.3 summarizes the model parameters chosen after the sensitivity study.

Table 9.3. Chosen model settings.

	population size *NP*	stop after this number of generations without improvement	maximum number of generations	number of training points *N*	number of runs per order
finite difference runs	150	250	2,500	800	10
Adams-Bashforth runs	150	250	2,500	6400	10
Runge-Kutta runs orders ≤ 4	350	500	5,000	-	100
Runge-Kutta runs order 5	350	10,000	100,000	-	100

The sensitivity analysis reveals a clear difference between the computations based on a target function and those based on order conditions. As discussed, for the Runge-Kutta runs the large variation in results and the slow convergence makes it imperative to make many runs with a high number of generations. Conversely, the finite difference and Adams-Bashforth runs have quick convergence and moreover attain very similar optima for different runs, so that a small number of relatively short runs suffice. Compared to the runs of the finite difference schemes, computation of Adams-Bashforth methods requires less point evaluations, so that more training points can be included in a comparable simulation time.

Training results of derivative schemes

The error used in the fitness computation is the difference between candidate derivative values and the analytical values of the target function. However, since the coefficients are known from theory it is more sensible to assess the training results by the absolute errors between computed and analytical coefficient values. These errors are plotted in Figure 9.7 (left). Appendix II contains complete tables with the resulting coefficients of the best runs for all computed configurations.

Figure 9.7 compares the results of schemes of different orders. The general trend is that the mean absolute error of a finite difference scheme increases with the order. The forward scheme exhibits a virtually exponential error growth, whereas the central scheme errors deteriorate only past the sixth order. The central runs where an extra $f(x)$-term was added are only slightly worse than the runs where this was not done; the corresponding coefficients are close to zero (see Appendix II). Figure 9.7 (left) also indicates that the two special schemes attain similar accuracy as the traditional schemes. Also, the results of the computed Adams-Bashforth methods are plotted in the same figure. They are relatively accurate and show an exponential trend up to the fourth order.

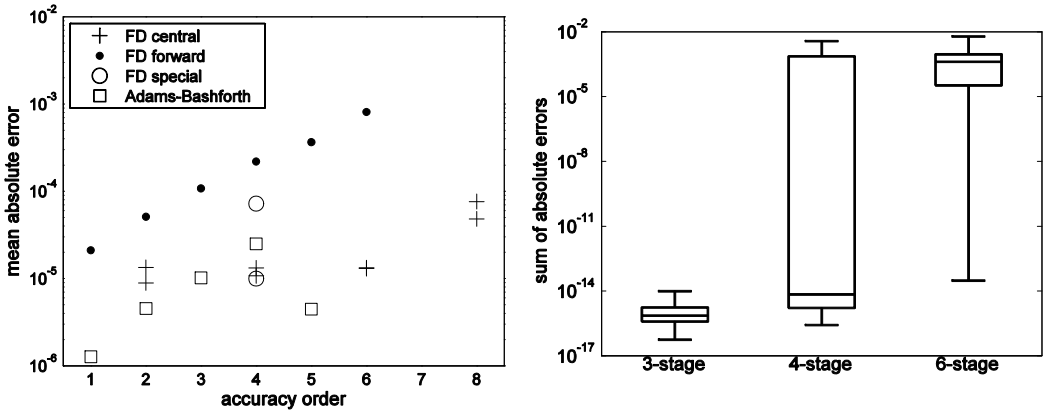

Figure 9.7. Left: mean absolute error of evolved coefficients as function of accuracy order for training based on target points of the function in Figure 9.3. Right: sum of absolute errors for all hundred runs for the computed Runge-Kutta schemes of stages 3, 4 and 6. In the box plots, the central line is the median, the box edges indicate first and third quartiles and the whiskers extend to the most extreme data points.

Training results of integration schemes

The definition of the training error for the Runge-Kutta schemes is based on computed and analytical values of the order conditions. Unlike the analytically tractable finite difference schemes, the computed coefficients for the Runge-Kutta methods do not resemble known values. Box plots are shown in Figure 9.7 (right) for all the runs performed for stages 3, 4 and 6. The whiskers of the box plots extend to the extreme values. Most interestingly, the best evolved schemes have absolute errors, summed over all coefficients, better than 10^{-13} for all stages. The variation in results is largest for the 4-stage scheme, which suggests that in hindsight the termination criterion could have been more conservative. Nevertheless, the accuracy of the evolved schemes for stages 3 and 4 is bounded by rounding-off errors; the used software (MATLAB) allows for 16 digits in the double-precision (64 bit) floating point format. Better results would arguably be possible by using symbolic representation, but this slows down the computations considerably.

Validation

In validation tests we applied the best evolved numerical schemes to the differentiation and integration of unseen functions and compared the errors of these numerical solutions (deviations from the analytical solution) to the errors of the classical (analytically derived) numerical schemes. By varying the step size, the slopes of the resulting graphs reveal the orders of accuracy. The Runge-Kutta schemes are applied to the same non-autonomous initial value problem that was used in Boyce and DiPrima (2001) to compute errors:

$$\dot{y} = 1 - x + 4y,\ y_0 = 1, \text{with solution } y = \frac{1}{4}x - \frac{3}{16} + \frac{19}{16}e^{4x}, \tag{9.5}$$

and all other methods are applied to $f(x) = 2e^{18x}$ and $f'(x) = 36e^{18x}$. The validation runs are plotted in Figure 9.8 for the finite difference schemes and in Figure 9.9 for the integration schemes. In all cases, the normalized local error is considered at the same x-location for all schemes.

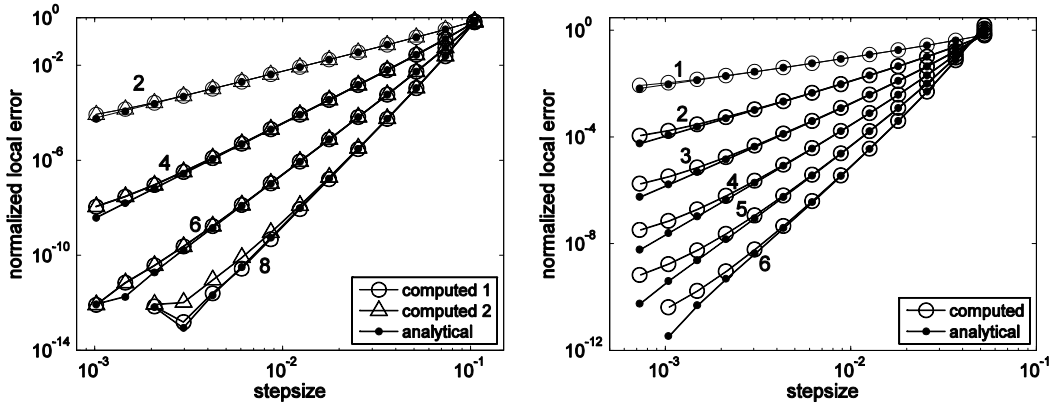

Figure 9.8. Validation errors of computed finite difference schemes for approximating first order derivative. Left: central schemes; Right: forward schemes. The orders of accuracy are indicated in the plots.

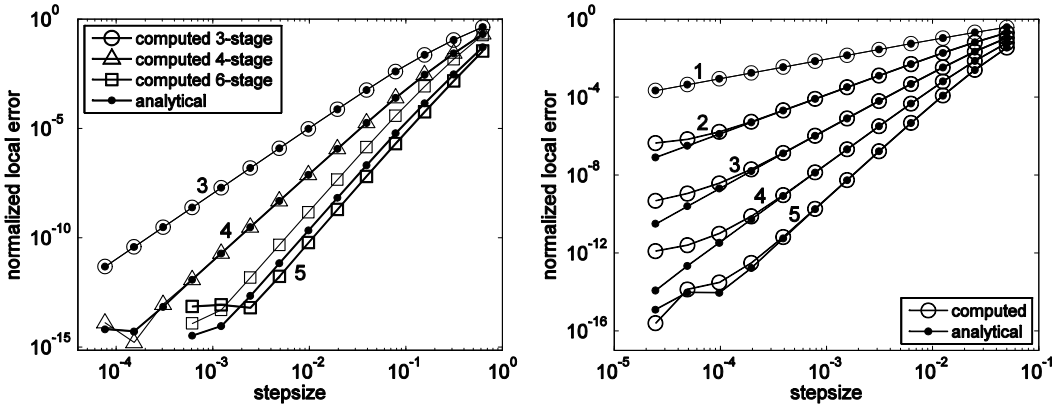

Figure 9.9. Validation errors of computed integration schemes compared to results of analytically derived schemes. Left: Runge-Kutta schemes; Right: Adams-Bashforth schemes. The orders of accuracy are indicated in the plots.

The plots confirm that the intended orders of accuracy are indeed reached. This follows from the fact that there is little deviation between computed and analytical schemes. Also, it was verified that the lines run parallel to the theoretical lines h^n, for order n. The plots of Figure 9.8 and Figure 9.9 exhibit two sources of deviation from the expected straight line: due to rounding-off errors and due to the computed scheme not being as accurate as the analytical scheme. The latter error type is manifested by the bending line ends of the forward scheme (Figure 9.8, right) and the Adams-Bashforth scheme towards smaller step sizes (Figure 9.9, right). The software-dependent machine rounding-off error affects both computed and analytical schemes, as can be seen for example for the fourth-order (4-stage) Runge-Kutta scheme around $h = 10^{-4}$ (Figure 9.9, left). For the Runge-Kutta fifth-order method (6-stage), the results of the *two* best evolved schemes are plotted along with the results of a

theoretically derived scheme (Butcher 2008). Remarkably, one of these schemes achieves a bettter accuracy than the theoretical scheme: the lowest line in Figure 9.9 (left) has the smallest errors. This is a coincidental feature related to the choice of the test function. Several additional tests with different functions (not shown to avoid figure overcrowding) exhibited the same order of accuracy with errors following the same h^5 trend), but the errors were not always lower than those of the analytical scheme.

9.3 Conclusions

This chapter has laid a foundation for automatic derivation of discrete numerical schemes by evolutionary computing. This can become a highly prized approach in situations where analytical solutions are absent and standard numerical approaches fail.

It was shown in this chapter how coefficients of widely used numerical methods can be computed up to practicable accuracies using differential evolution. This was done by representing coefficient sets as floating-point vectors. Two different training procedures were applied: based on a target function pair f and $f\,'$ and based on order condition equations. Finite difference methods for approximating first derivatives were computed up to order 8 (central schemes) and order 6 (forward schemes); also two non-standard schemes of order 4 were derived. The multi-step Adams-Bashforth integration method was computed up to order 5. Accurate training of the finite difference and Adams-Bashforth schemes, which employed samples from the target function pair, was found to depend primarily on the number of training points and not on sample frequency or target function shape. Moreover, there appeared to be a threshold value for the population size required for attaining reasonable accuracies. As a result, the sum of absolute training errors showed a predictable decaying exponential relation with the number of point evaluations.
Explicit Runge-Kutta methods were trained by promoting adherence to the order conditions, yielding schemes of stages 3, 4 and 6, up to order 5 and with absolute errors summed over all order conditions in the order of 10^{-14}. The influence of population size proved to be more diffuse than for the target-function training. This made it necessary to have more and longer runs (up to 100,000 generations) to achieve convergence. Furthermore, schemes of higher order, having more terms and coefficients, required larger population size (for order condition training) or inclusion of more training points (for target function training).

The research underlines the effectiveness of the differential evolution algorithm as an easy-to-implement heuristic with few model parameters. The accurate results of the Runge-Kutta runs showed its persisting capability to avoid local optima, provided that the termination criterion is appropriately tuned so that runs do not end prematurely.

9.4 Reflection on the work in Chapters 8 and 9

As with the work of the previous chapter, the choice for evolutionary computing was initially made with (an extension to) GP in mind. Preliminary GP tests show that evolving recurrence relations goes much in the same way as for standard SR, but fitness evaluation by solving a target initial value problem by the candidate algorithms is a computational hurdle. This can possibly be solved by smart choices in programming implementation. In future work, test problems that vary in difficulty (e.g. stiffness) during the run could be considered or non-

error-based fitness evaluations, such as based on algorithm complexity, or symmetry. An elegant way of reducing the required number of candidate model evaluations worth trying here is the coevolution of fitness predictors (Schmidt and Lipson, 2008). Using differentiation as an example of a numerical problem that we want to find a scheme for, one training point consists of two data sets (a sampled function and corresponding derivative values). Applied to this problem, one fitness predictor consists of a set of such 'training points'. Finding a well-trained fitness predictor then solves the problem of what functions are best to choose for training the numerical schemes.

In short, thinking about the discovery of numerical algorithms in the presented way leads to questions about how and how well they can be optimised. Can we look at certain integration algorithms as local optima in the search space of all solver algorithms? Euler's method is the least complex and fastest to run but provides poor accuracy, Heun is one step up from this, etcetera. Can the associated Pareto front (Figure 8.16) be found and will this produce more efficient solver algorithms? Future general applications include new numerical stencils for solving partial differential equations. Specific examples are algorithms that provide fluid-solid interface coupling, rules for model decomposition in mesh-based CFD, wave-current boundary conditions and again turbulence modelling, to name a few.

Figure 9.10. Weir in the Nederrijn at Driel, The Netherlands (https://beeldbank.rws.nl, Rijkswaterstaat).