2950 Niles Road, St. Joseph, MI 49085-9659, USA
269.429.0300 fax 269.429.3852 hq@asabe.org www.asabe.org

An ASABE Meeting Presentation

DOI: https://doi.org/10.13031/aim.201901371

Paper Number: 1901371

Compact Codes for Advanced Computations

Hubert Montas[1] and Jonathan Resop[2]

[1]Fischell Department of Bioengineering, University of Maryland at College Park, College Park, MD, 20742

[2]Center for Geospatial Information Science, University of Maryland at College Park, College Park, MD, 20742

Written for presentation at the
2019 ASABE Annual International Meeting
Sponsored by ASABE
Boston, Massachusetts
July 7–10, 2019

ABSTRACT. ***Compact examples of vectored codes are presented for various advanced computational techniques, including Runge-Kutta solution of systems of ODEs, symplectic integration of Hamiltonian systems, Finite Difference solution of transport PDEs, nonlinear optimization, and error back-propagation in Artificial Neural Networks (ANNs). Most examples are shorter than 25 lines of code, implemented in either MATLAB or GNU Octave, and demonstrate the substantial efficiency resulting from the vectored approach to code development. These short code examples help to clarify some of the more nebulous aspects of the advanced computational methods which they implement, and can find uses both in educational settings and as starting points for targeted, vectored, research codes.***

Keywords. Calculus, Linear Algebra, Differential Equations, Vectored Code

Introduction

The combination of calculus and linear algebra produces important mathematical tools for engineering analysis. Calculus provides the formulations for describing the dynamics of phenomena of interest and for identifying optimal parameters to describe them, and linear algebra provides a way to concisely express the relationships between systems of several variables and to describe computations performed on them (Strang, 1988). For example, while solving a single differential equation analytically is very much a problem in scalar calculus, solving a system of coupled differential equations is most clearly expressed by wrapping calculus techniques around the matrix-vector form of the system. Similarly, numerical approaches for solving differential equations, that involve discrete approximations of calculus operators, generally result in systems of simultaneous algebraic equations, which involve approximate solution values at a discrete set of spatio-temporal locations, and are most conveniently expressed in matrix-vector form. Optimization techniques, where the values of several parameters are sought, simultaneously, to minimize the value of an objective function that itself involves a set of dependent variable values, lead most naturally to a calculus formulation expressed in matrix-vector terms.

The engineering problems addressed by combining calculus and linear algebra are often quite complex and their solution is commonly approached computationally, using pre-made software or via code development. Pre-made, purpose-built software is the best choice in professional engineering practice, providing a combination of robustness and efficiency. In education however, where the goal is for students to become knowledgeable in those mathematical and computational techniques that underpin modern engineering analysis, it is most beneficial if clear and concise code examples of these key techniques can be provided, for the students to study, comprehend, and experiment with. Pre-made software is useful for students to practice applied design, in capstone courses for example, but code development is required for them to also develop skills in logical thinking and to deeply understand the fundamental concepts on which the field is built. In engineering research, the needed software may not exist as specific investigations are often one-offs, and one may have to resort to code development by necessity. The development of clear and concise code, in this context, can be most valuable in enhancing the prospects for publication, review and especially reproduction of results by others (Baker, 2016).

This manuscript presents five computational techniques that are used in advanced engineering analyses and associated compact code examples that can be useful for education and research. The five techniques are: 1) the Runge-Kutta method; 2) symplectic integration; 3) the Finite Difference Method; 4) the Levenberg-Marquardt nonlinear optimization technique, and; 5) Artificial Neural Networks. In each case, the mathematical underpinnings of the technique are described, combining calculus and linear algebra, an example application is provided, and a compact code example is listed and described briefly. The code examples span from approximately 10 lines to 50 lines which should make them relatively easy to analyze and comprehend. In turn, this should help to clarify the exact nature of the mathematical techniques which they implement.

The code examples presented in this work are short in large part because they are written in a high-level language that natively supports linear algebraic computations. The implementation language is that of the Mathworks MATLAB and GNU Octave softwares. As this computer language was designed from the onset for matrix computations, iterative constructs (such as for-loops, do-loops or while-loops) are not needed to multiply matrices with vectors, and the solution of a system of linear algebraic equations (expressed in matrix-vector form) is expressed as just one line of code, just as it would form a single line in a mathematical description of an advanced technique. Accordingly, the code is frequently closer to the mathematical description of the process it implements, which should enhance its comprehensibility. As a particular example, several of the code samples use the left-division operator, originally defined in 1928 (Hensel, 1928, p.249, right above equation 3) to represent the solution of linear-algebraic systems (for which multiplication is not commutative) that is available in the target language:

$$A\ X = B \quad \Leftrightarrow \quad X = A \setminus B$$

Matrix-vector computations are performed very efficiently in the Mathworks MATLAB where the multiple cores available in contemporary CPUs are automatically used to parallelize these operations. In GNU Octave, automated parallelization is possible with the installation and linking of an appropriate linear algebra library, such as OpenBLAS or ATLAS.

The Runge Kutta Method

The Runge-Kutta method is a popular numerical technique for solving linear and nonlinear Ordinary Differential Equations (ODEs) (Boyce and DiPrima, 1986, Chapter 8). It is more accurate than simpler approaches, like the Euler or improved Euler method, while being nearly as computationally efficient. Consider a system of ODEs that describe the rate at which a vector of n dependent variables, X, changes with a dependent variable, t, as a vector function with internal parameter, F:

$$\frac{dX}{dt} = F(t, X)$$

where:

$$X(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix} \quad , \quad F(X,t) = \begin{bmatrix} f_1(t, x_{1,} x_{2,} \ldots, x_n) \\ f_2(t, x_{1,} x_{2,} \ldots, x_n) \\ \vdots \\ f_n(t, x_{1,} x_{2,} \ldots, x_n) \end{bmatrix}$$

or, if each ODE is written out individually:

$$\begin{gathered} \frac{dx_1}{dt} = f_1(t, x_{1,} x_{2,} \ldots, x_n) \\ \frac{dx_2}{dt} = f_2(t, x_{1,} x_{2,} \ldots, x_n) \\ \vdots \\ \frac{dx_n}{dt} = f_n(t, x_{1,} x_{2,} \ldots, x_n) \end{gathered}$$

The objective of the Runge-Kutta method is to compute an accurate approximate solution for X over values of the independent variable ranging from t_0 to t_{max}, given initial values of X_0 for X at t_0. The process starts by defining a vector of m points (which we will assume equidistant) along the line form t_0 to t_{max} and proceeds to iteratively compute an approximate solution for X, for points k + 1 = 2 to m, from the value of the solution at the previous point (k), and four estimates, computed from F, of the rate of change of the solution over each subinterval. The spacing between the points, the value of the independent variable at each point, k, and the value of the approximate solution vector at these points are:

$$\Delta t = \frac{t_{max} - t_0}{m-1} \quad , \quad t_k = (k-1)\Delta t + t_0 \quad , \quad X_k \approx X(t_k)$$

and the Runge-Kutta formula is:

$$X_{k+1} = X_k + \frac{1}{6}\left[dx_1 + 2\,dx_2 + 2\,dx_3 + dx_4\right]$$

with:

$$\begin{aligned} dx_1 &= F\left(t_k \, , \, X_k\right) \\ dx_2 &= F\left(t_k + \frac{\Delta t}{2} \, , \, X_k + dx_1 \frac{\Delta t}{2}\right) \\ dx_3 &= F\left(t_k + \frac{\Delta t}{2} \, , \, X_k + dx_2 \frac{\Delta t}{2}\right) \\ dx_4 &= F\left(t_k + \Delta t \, , \, X_k + dx_3 \Delta t\right) \end{aligned}$$

The MATLAB software provides a function named ode45() which computes the Runge-Kutta calculations with adaptive subinterval spacings. The function takes 3 inputs: 1) the vector function F, that takes the current value of t and a (column) vector of current values of the dependent variable as input and returns a (column) vector of the rate at which X changes with t; 2) a vector of the m points, from t_0 to t_{max}, at which solution values are sought, and; 3) the (column) vector X_0 of initial values of the dependent variables. The function produces two output: 1) a (column) vector of the values of the independent variable, t, at which solutions were computed, and; 2) a matrix of solution values, X, where the 1st column contains X_1 at all values of t, the second column contains X_2, and so forth.

A compact function, named ode45rk(), that implements the Runge-Kutta solution process with the same input-output interface as described above is listed further below. To examplify its use, consider the solution of the nonlinear Lotka-Volterra predator-prey system of ODEs (where P is predator population and H is herbivore population):

$$\frac{dH}{dt} = 0.5H - 0.02HP$$
$$\frac{dP}{dt} = -0.4P + 0.001HP$$

for times, t, of 0 to 100 years, with initial populations of H_0 = 600 herbivores and P_0 = 10 predators. The derivative function (coded as an anonymous function), time vector (with 500 subintervals) and initial values are specified as:

```
F  = @( t , X) [ 0.5*X(1) - 0.01*prod(X) ; -0.4*X(2) + 0.001*prod(X) ];
tv = linspace(0,100,501);
X0 = [ 600 ; 10 ] ;
```

where H and P are the 2 components of the vector of dependent variables: H=X_1 and P =X_2. The solution times and matrix are obtained using:

```
[t , XM] = ode45rk( F , tv , X0 );
```

The results are shown in Figure 1 and display the expected pattern of lagged population oscillations, coupled with an egg-shaped closed-loop phase portrait.

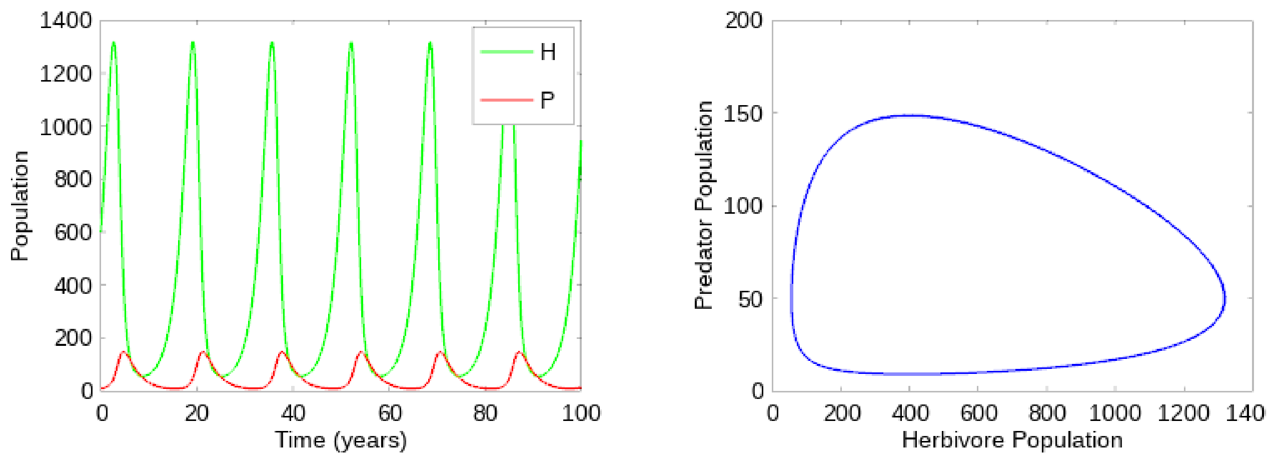

Figure 1. Runge-Kutta Solution of a Nonlinear System of Predator-Prey Dynamics ODEs

The implementation of the ode45rk() function takes advantage of vectored operations to result in code that is compact, efficient, and readily applicable to systems of ODEs of arbitrary size. It is a rather direct implementation of the mathematical formulas presented above and consists of less than 20 lines of code, including comments:

```
function [t,XM] = ode45rk(F, tv, X0)
% Runge-Kutta Solution of a System of ODEs
%   F  = derivative function
%   tv = time vector
%   X0 = initial values
```

```
t       = tv(:);                    % independent var output vector
dt      = t(2)-t(1);                % subinterval spacing
XM      = zeros(length(t),length(X0)); % initial solution matrix
XM(1,:) = X0;                       % set initial values into matrix
% loop to compute solution over independent variable values
for k = 1:length(t)-1
  tk   = t(k);                      % current value of indep var
  Xk   = XM(k,:)';                  % current dependent var vector
  dx1  = F(tk,      Xk);            % Runge-Kutta estimate 1
  dx2  = F(tk+dt/2, Xk+dx1*dt/2);   % Runge-Kutta estimate 2
  dx3  = F(tk+dt/2, Xk+dx2*dt/2);   % Runge-Kutta estimate 3
  dx4  = F(tk+dt,   Xk+dx3*dt);     % Runge-Kutta estimate 4
  XM(k+1,:) = XM(k,:) + (dx1+2*dx2+2*dx3+dx4)'*dt/6; % next solution val
end
```

Symplectic Integration

Symplectic integrators are mathematical tools used to solve Hamiltonian systems of equations that typically describe the temporal dynamics of rigid bodies, under the influence of various forces whose magnitude depend on the position of these bodies. Examples of such systems arise in both planetary dynamics, where forces emanate from the gravitational potential, and molecular dynamics where Lennar-Jones interactions are effective. Consider a system of p bodies in d-dimensional space, with masses represented by a vector of p values (M), and positions, velocity vectors and acceleration vectors represented by p-by-d matrices (X, V and A, respectively), as follows (shown here in 3-D):

$$M = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_p \end{bmatrix} \;,\quad X = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_p & y_p & z_p \end{bmatrix} \;,\quad V = \begin{bmatrix} v_{x[1]} & v_{y[1]} & v_{z[1]} \\ v_{x[2]} & v_{y[2]} & v_{z[2]} \\ \vdots & \vdots & \vdots \\ v_{x[p]} & v_{y[p]} & v_{z[p]} \end{bmatrix} \;,\quad A = \begin{bmatrix} a_{x[1]} & a_{y[1]} & a_{z[1]} \\ a_{x[2]} & a_{y[2]} & a_{z[2]} \\ \vdots & \vdots & \vdots \\ a_{x[p]} & a_{y[p]} & a_{z[p]} \end{bmatrix}$$

Provided that these bodies obey Newtonian physics, their dynamics are described by the following equations:

$$\frac{dX}{dt} = V \quad , \quad \frac{dV}{dt} = A \quad , \quad A = F_{tot}(M, X) \;./\; M_d$$

where F_{tot} is the p-by-d matrix of force vectors acting on each mass (which are typically position-dependent), the symbol ./ is the itemwise division operator (item-by-item division of one vector by another, resulting in a vector of quotients), and the p-by-d matrix M_d consists of d replicates of vector M. To keep matters simple, yet relevant, we will restrict ourselves here to interaction forces exerted by each particle onto the other particles and whose magnitude and direction depends on the magnitude and direction of the separation vector between those particles, and possibly on the product of their masses. The total force acting on body i is given by:

$$F_{tot}^{(i)} = \sum_{j=1,\, j\neq i}^{p} F\left(m_i m_j, L^{(i,j)}\right) U^{(i,j)}$$

where F is a functions that returns the magnitude of the force between bodies i and j as a function of the product of their masses and of the distance, $L^{(i,j)}$ between them, and $U^{(i,j)}$ is the unit vector from body j to body i, which determines the direction of the force (shown here in 3-D):

$$L^{(i,j)} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \quad , \quad U^{(i,j)} = \frac{(x_i - x_j,\, y_i - y_j,\, z_i - z_j)}{L^{(i,j)}}$$

Given initial positions, X_0, and velocities, V_0, of the p bodies, and a formula for the magnitude of the interaction force, F, it is possible, using a numerical technique for example, to compute their positions at a pre-selected set of future times, from 0 to t_{max}. The Verlet method is a numerical symplectic integrator that can be used for this purpose (Verlet, 1967). In this method, the time domain is discretized into m equidistant time points, and approximate values of X, V and A are computed at each of these times based on their values (and force) at the previous time:

$$\Delta t = \frac{t_{max}}{m-1}\ ,\ \ t_k = (k-1)\Delta t\ ,\ \ X_k \approx X(t_k)\ ,\ \ V_k \approx V(t_k)\ ,\ \ A_k \approx A(t_k)\ ,\ \ F_{tot[k]} \approx F_{tot}(M, X_k)$$

The Verlet update formulas are:

$$X_k = X_{k-1} + V_{k-1}\,\Delta t + \frac{1}{2}\,A_{k-1}\,(\Delta t)^2\ ,\ A_k = F_{tot}(M, X_k)\ ./\ M_d\ ,\ V_k = V_{k-1} + \frac{1}{2}(A_k + A_{k-1})\Delta t$$

and applying them iteratively, for k = 2 to m, produces the successive positions of the system of rigid bodies.

A function named verlet(), that performs the above computations for an arbitrary number of particles, in an arbitrary number of spatial dimensions, and with and arbitrary force function, is presented further below. Its five inputs are: 1) the force function; 2) the vector of solution times; 3) the vector of p masses; 4) the matrix of initial positions, and; 5) the matrix of initial velocities. Its two first outputs are the vector of times and a 3-D array of computed positions where rows correspond to successive times, columns correspond to spatial coordinates (x, y, z in 3-D) and planes correspond to the different bodies (1 to p). To illustrate its application, the example below computes the simultaneous motions of the earth and moon, as a response to each other's gravitational force, in two dimensions, for one month. The time vector, mass vector, position matrix, velocity matrix (for a stable moon orbit), and gravitational force function are defined as:

```
tv = linspace(0,2.592e6,101);     % solution times (30 days, in seconds)
M  = [5.97e24 ; 7.35e22];         % masses of earth and moon (kg)
X0 = [ 0 0 ; 378000e3 0];         % initial positions (m)
V0 = [ 0 0 ; 0   1.02e3];         % initial velocities (m/s)
F  = @(MM,dist) -6.674e-11*MM./dist.^2; % gravitational force (function)
```

Next, to keep the system within a fixed area of space, we correct the initial earth velocity so that the system has zero total momentum (otherwise it will drift continually into outer space):

```
V0(1,:) = -sum(repmat(M,1,2).*V0)/M(1); % earth veloc adj for zero momentum
```

The final step is to solve for the motion of the two bodies:

```
[t,XM] = verlet( F, tv, M, X0, V0);
```

The result is shown in Figure 2. Within the 30 days of the simulation the moon performs a full circular orbit of the earth. The earth also performs a small circular motion (wobble) that is caused by the gravitational pull of the moon.

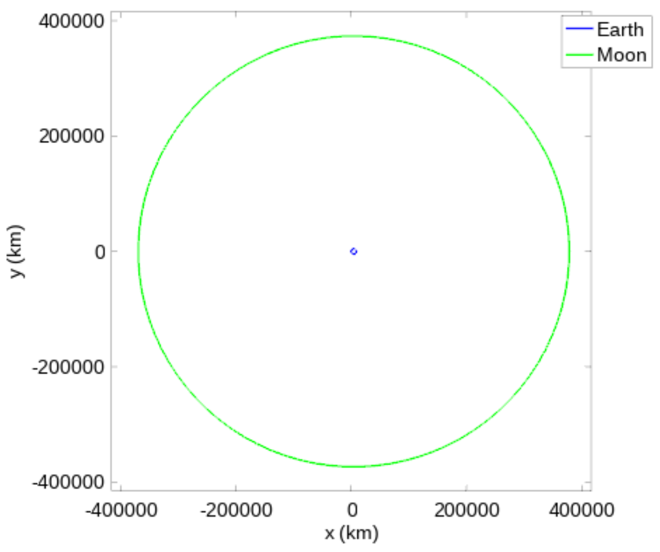

Figure 2. Verlet Solution of the Motion of the Earth-Moon Planetary System

The code of the verlet() function is listed below. It takes advantage of vectored computations for compactness and speed. It uses a nested function, named Ftot(), to compute the total force acting on all of the simulated bodies and comes to a total of 30 lines, including comments.

```
function [t,XM] = verlet(F,tv,M,X,V)
% Verlet Symplectic Integrator (order 2)
%F=force function, tv=time vector, M=mass vector
%X=initial position matrix, V=initial velocity matrix
% preliminary computations
p  = size(X,1);                      % number of bodies
d  = size(X,2);                      % number of spatial dimensions (2 or 3)
Md = repmat(M,1,d);                  % replicated mass matrix
t  = tv(:);                          % output time-vector
dt = t(2) - t(1);                    % time step
A  = Ftot(F,M,X)./Md;                % initial acceleration
XM = zeros(length(t),d,p);           % initial solution matrix
XM(1,:,:) = X';                      % store initial positions in matrix
% solution for successive times
for k = 2:length(t)
  X  = X + V*dt + 0.5*A*dt*dt;       % new positions
  XM(k,:,:) = X';                    % store new positions in matrix
  An = Ftot(F,M,X)./Md;              % new accelerations
  V  = V + (An+A)*dt/2;              % new velocities
  A  = An;                           % accelerations for next iter
end
% nested function to compute the force matrix
  function fout = Ftot(F,M,X)
    R    = repmat(reshape(X,p,1,d),[1,p,1])-repmat(reshape(X,1,p,d),[p,1,1]);
    dist = sqrt(sum(R.^2,3));         % distances between all bodies
    Fod  = F(M*M',dist)./dist;        % forces between all bodies
    Fod(1:p+1:p*p) = 0;               % remove "self"-force
    fout = squeeze(sum(R.*repmat(Fod,[1,1,d]),2)); % total force on each body
  end
end
```

The Finite Difference Method

The Finite Difference Method (FDM) is a technique for computing approximate numerical solutions of Partial Differential Equations (PDEs) (Skaggs and Khaleel, 1982). It is simpler than the Finite Element Method (FEM) (Reddyand.Gartling, 2001) and thus potentially more adequate for the development of compact illustrative codes. An example application of the method would be the solution of a 1-D time-dependent transport equation of the form:

$$\frac{\partial u}{\partial t}=k\,u-v\frac{\partial u}{\partial x}+D\frac{\partial^2 u}{\partial x^2}$$

over a spatial domain where the spatial coordinate x ranges from x_{min} to x_{max}, and a temporal domain where time, t, ranges from 0 to t_{max}, with Dirac Initial Conditions (IC s) and Neumann Boundary Conditions (BCs):

$$\text{IC:}\quad u(x,0)=M_0\,\delta(x-x_0)$$
$$\text{BCs:}\quad \left.\frac{\partial u}{\partial x}\right|_{x=x_{min}}=0 \quad , \quad \left.\frac{\partial u}{\partial x}\right|_{x=x_{max}}=0$$

In the above PDE, the dependent variable, u, may represent the concentration of an entity transported by a combination of advection, with speed v, diffusion, with coefficient D, and first-order reaction, at rate k. The FDM solution process starts by discretizing the spatial and temporal domains over which a solution is sought. Here, we will use n_x equidistant nodes in space, indexed by the variable i, and n_t equidistant nodes in time, indexed by the variable q. The spacing between nodes, and the position of the nodes are then given by:

$$\Delta x = \frac{x_{max} - x_{min}}{n_x - 1} \quad , \quad x_i = (i-1)\Delta x + x_{min} \quad , \quad \Delta t = \frac{t_{max}}{n_t - 1} \quad , \quad t_q = (q-1)\Delta t$$

The FDM proceeds by defining a discrete approximation to the governing PDE, where solution values are obtained at the spatial nodes, at a given time step q, from their known values at the earlier time step q-1 (starting from the initial conditions at q = 0) by solving a system of linear algebraic equations of the form:

$$LHS\ U^q = RHS\ U^{q-1} \ \Leftrightarrow\ U^q = LHS^{-1} RHS\ U^{q-1}$$

where LHS and RHS are n_x-by-n_x discretization matrices and U^q is the vector of solution values at all spatial nodes, at time step q:

$$U^q = \begin{bmatrix} u_1^q \\ u_2^q \\ \vdots \\ u_{n_x}^q \end{bmatrix}$$

The spatial component of the PDE is discretized by replacing the spatial derivatives with the 2nd-order accurate finite difference approximations:

$$\left.\frac{\partial u}{\partial x}\right|_{x=x_i} \approx \frac{\partial u_i}{\partial x} = \frac{u_{i+1} - u_{i-1}}{2\Delta x} \quad , \quad \left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i} \approx \frac{\partial^2 u_i}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}$$

which, upon substitution into the PDE, for an arbitrary interior node i (i.e. i ranges from 2 to n_x-1) produces the semi-discrete equation:

$$\frac{\partial u_i}{\partial t} = \begin{bmatrix} \underbrace{\frac{v}{2\Delta x} + \frac{D}{(\Delta x)^2}}_{\text{lsd}} & \underbrace{k - \frac{2D}{(\Delta x)^2}}_{\text{csd}} & \underbrace{-\frac{v}{2\Delta x} + \frac{D}{(\Delta x)^2}}_{\text{rsd}} \end{bmatrix} \begin{bmatrix} u_{i-1} \\ u_i \\ u_{i+1} \end{bmatrix}$$

At the boundaries of the spatial domain, phantom nodal solutions for i= -1 and i = n_x+1 would appear in this formulation. However, because of the Neumann BCs, solution values right outside of the domain equal those in the first inside nodes, such that the full set of semi-discrete equations can be assembled into:

$$\frac{\partial}{\partial t}\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_x-1} \\ u_{n_x} \end{bmatrix} = \begin{bmatrix} \text{csd} & \text{rsd}+\text{lsd} & 0 & \cdots & 0 \\ \text{lsd} & \text{csd} & \text{rsd} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \text{lsd} & \text{csd} & \text{rsd} \\ 0 & \cdots & 0 & \text{lsd}+\text{rsd} & \text{csd} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_x-1} \\ u_{n_x} \end{bmatrix}$$

or, more compactly, introducing SMAT as the above tri-diagonal spatial discretization matrx:

$$\frac{\partial U}{\partial t} = \text{SMAT}\ \ U$$

We choose to discretize the PDE in time using the Crank-Nicolson 2nd-order approach which uses the approximation:

$$\frac{U^q - U^{q-1}}{\Delta t} = SMAT \quad \frac{U^q + U^{q-1}}{2}$$

which upon re-arranging produces the LHS/RHS form presented earlier:

$$\left[I - \frac{\Delta t}{2} SMAT\right] U^q = \left[I + \frac{\Delta t}{2} SMAT\right] U^{q-1}$$

Once the SMAT has been built and Δt has been determined, all that remains to obtain a solution is to insert the ICs into the initial solution vector, U^0, and iterate over time values with index k going from 2 to n_t. The function FDM_sol() listed further below builds the required matrices and performs the iteration process needed to solve a PDE of the above form using the FDM. The function takes 4 inputs: 1) a vector of the 3 transport parameters: k, v and D; 2) a vector of the equidistant spatial locations at which a solution is sought; 3) a vector of the equidistant times at which solutions are sought, and; 4) a vector of the initial solution, U^0. The vectors of spatial and temporal node locations are not strictly necessary (only the spacings and node counts are important) but are included for potential future extensions where parameters may depend on spatio-temporal coordinates. The function returns 3 outputs: 1) a column vector of spatial locations of the nodes; 2) a row vector of the times at which solutions were computed, and 3) a matrix of solution values with spatial position increasing as one goes down the rows and time increasing from one column to the next.

To illustrate the use of FDM_sol() we consider the numerical solution of the above PDE with parameters:

$$k = -0.01 \ , \ v = 0.3 \ , \ D = 0.7 \ , \ x_{min} = 0 \ , \ x_{max} = 10 \ , \ t_{max} = 8 \ , \ M_0 = 3 \ , \ x_0 = 4$$

and n_x = 101 nodes in space and n_t = 201 time steps. The following sequence of commands defines and solves the problem, and also produces a plot of the solution, which is shown on Figure 3:

```
kvD = [-0.01 0.3 0.7];
nx  = 101;
nt  = 201;
xv  = linspace(0,10,nx);
tv  = linspace(0,8,nt);
U0  = zeros(nx,1);
dx  = xv(2)-xv(1);
U0(round(1+4/dx)) = 3/dx;
[x,t,U] = FDM_sol(kvD,xv,tv,U0);
plot(x,U(:,1:40:end))
xlabel('x (units)'); ylabel('Concentration (units)')
axis([0 10 0 2])
legend([repmat('t = ',6,1) num2str(tv(1:40:end)',3)])
```

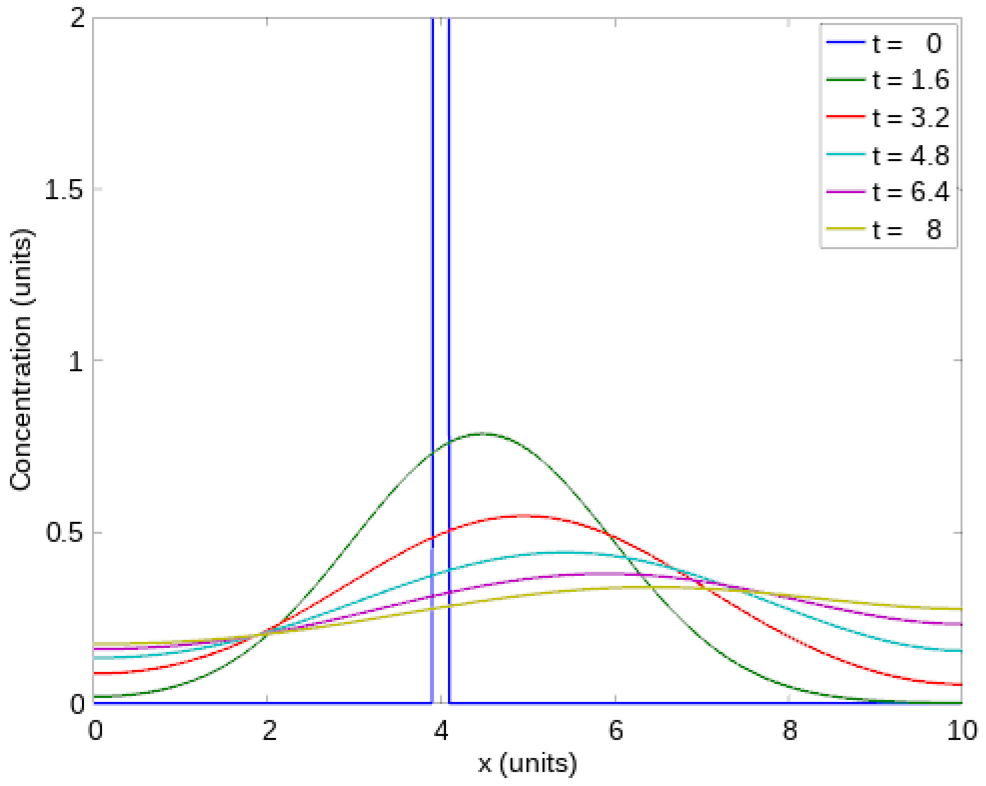

Figure 3. FDM Solution of a Linear Advection-Diffusion-Reaction Transport PDE

The function FDM_sol itself is listed below. At 20 lines of code (shown without comments) it is very compact, thanks to the use of vectored operations (there is a single for-loop). The code uses sparse matrix functions speye() and spdiags() to construct the SMAT, LHS and RHS matrices, which saves on memory and further improves computational efficiency.

```
function [x,t,U] = FDM_sol(kvD,xv,tv,U0)
x    = xv(:);
t    = tv(:)';
nx   = length(x);
nt   = length(t);
dx   = x(2)-x(1);
dt   = t(2)-t(1);
lsd  =  kvD(2)/(2*dx) + kvD(3)/dx^2;
csd  =  kvD(1) - 2*kvD(3)/dx^2;
rsd  = -kvD(2)/(2*dx) + kvD(3)/dx^2;
SMAT = spdiags(repmat([lsd csd rsd],nx,1),[-1 0 1],nx,nx);
SMAT(1,2)       = SMAT(1,2)+lsd;
SMAT(end,end-1) = SMAT(end,end-1)+rsd;
LHS  = speye(nx) - SMAT*dt/2;
RHS  = speye(nx) + SMAT*dt/2;
U    = zeros(nx,nt);
U(:,1) = U0;
for k = 2:nt
  U(:,k) = LHS\(RHS*U(:,k-1));
end
```

Nonlinear Optimization

Nonlinear optimization is a process aimed at identifying a set of parameter values that minimize the value of a pre-selected objective function. It is used, for example, to find parameter values for a nonlinear model, such that model predictions provide a good fit to observed data. Consider for example a model described by a nonlinear function, G, that takes vectors of independent variables (T, X, Y, ...) as input, each with n items, and is parameterized by a vector of m parameters, U. The model's output is a vector of n predicted values, P, which we seek to be a good fit to a vector of n corresponding observations, O. Accordingly:

$$P = G(T, X, Y, \ldots; U)$$

where (including the vector E of deviations between model predictions and observations, aka errors or residuals):

$$P = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \;,\quad O = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_n \end{bmatrix} \;,\quad T = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} \;,\quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \;,\quad \cdots \;,\quad U = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} \;,\quad E = P - O = \begin{bmatrix} p_1 - o_1 \\ p_2 - o_2 \\ \vdots \\ p_n - o_n \end{bmatrix}$$

We consider the special but common case where the objective of the optimization is to obtain a set of parameters U that minimizes the sum of squared errors, SSE. Combining calculus and linear algebra, we seek U such that:

$$\frac{d\,SSE}{d\,U} = \frac{d}{d\,U} E^T E = 2\frac{d\,E^T}{d\,U} E = 2\frac{d\,(P-O)^T}{d\,U} E = 2\frac{d\,P^T}{d\,U} E = 2\,J^T E = 0$$

where the Jacobian, J, is the matrix (m x n) of partial derivatives of model predictions with respect to model parameters:

$$J = \frac{d\,P}{d\,U^T} = \begin{bmatrix} \frac{d\,P}{d\,u_1} & \frac{d\,P}{d\,u_2} & \cdots & \frac{d\,P}{d\,u_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial p_1}{\partial u_1} & \frac{\partial p_1}{\partial u_2} & \cdots & \frac{\partial p_1}{\partial u_n} \\ \frac{\partial p_2}{\partial u_1} & \frac{\partial p_2}{\partial u_2} & \cdots & \frac{\partial p_2}{\partial u_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial p_m}{\partial u_1} & \frac{\partial p_m}{\partial u_2} & \cdots & \frac{\partial p_m}{\partial u_n} \end{bmatrix}$$

If an arbitrary parameter vector U is used to compute P, E, J and dSSE/dU from it (as $2\,J^T E$), dSSE/dU will likely not be zero, indicating that this arbitrary U does not minimize the SSE of model predictions. If this U is changed by a small amount, ΔU, such that it becomes U + ΔU, model predictions and errors will change approximately to (to 1st order):

$$P_{new} = P + J\,\Delta U \quad , \quad E_{new} = P_{new} - O = P + J\,\Delta U - O = J\,\Delta U + E$$

We can choose ΔU such that dSSE/dU is approximately zero at U + ΔU by substituting E_{new} for E in the equation for dSSE/dU and setting the result to zero. This produces the normal (Gauss-Newton) equation for iterative parameter update:

$$\frac{d\,SSE}{d\,U} = 2\,J^T E_{new} = 0 \;\Leftrightarrow\; J^T (J\,\Delta U + E) = 0 \;\Leftrightarrow\; J^T J\,\Delta U = -J^T E \;\Rightarrow\; \Delta U = -(J^T J)^{-1} (J^T E)$$

The convergence of successive updates of U, for cases where the distance between the initial parameter guess and the optimal one is large, has been improved by Levenberg and Marquardt (Transtrum and Sethna, 2012). Their modification introduces a damping parameter, λ (positive or null), to the above update formula:

$$\Delta U = -\left(J^T J + \lambda\,I\right)^{-1} (J^T E)$$

When λ = 0, the Gauss-Newton equation is recovered and higher values act as a divisor (damping) for ΔU. The damping parameter is commonly updated during iteration, decreasing it at each time step (moving towards Gauss-Newton), unless the new SSE is larger than the previous one, in which case it is increased (moving away from Gauss-Newton).

Application of the Leveberg-Marquardt nonlinear optimization process typically proceeds as follows:

```
1. choose a starting parameter vector U
2. choose a starting value for $\lambda$
3. compute P, E and SSE
4. if SSE is small enough, stop (done)
5. compute J
6. compute $\Delta U = - (J^T J + \lambda I) \backslash ( J^T E)$
7. compute $U_{new} = U + \Delta U$
8. compute P, E and $SSE_{new}$ using $U_{new}$
9. if $SSE_{new}$ is larger than SSE, increase $\lambda$ and go to step 6
10. set $U = U_{new}$, decrease $\lambda$, and go to step 3
```

The MATLAB function lsqnonlin() implements the Levenberg-Marquardt optimization algorithm, along with other techniques (it chooses the technique to apply automatically). Its two inputs consist of: 1) a user-defined function that takes a vector of model parameters (U) as input and returns the vector of model prediction errors (E) as output, and; 2) a user-defined vector of the initial guess for model parameter values (U). The lsqnonlin() function returns as output a vector of optimized model parameters (U) obtained by its selected optimization technique.

In the following, a function named lsqlm(), which has the same input-output interface, but uses only the Levenberg-Marquardt method, is presented as an example of compact code implementation. As an example of its use, we consider fitting a Langmuir isotherm to observations of phosphorus adsorption in an Australian soil. The Langmuir isotherm model, expressed in terms of original variables and parameters, and in the present notation, is:

$$x = \alpha \frac{K c}{1 + K c} \quad \equiv \quad P = u_2 \frac{u_1 X}{1 + u_1 X} = G(X ; U)$$

where x (eg. μg/g) is the adsorbed concentration of a compound (mapping to P and O in our notation), c (eg. μg/ml) is its concentration in the fluid (mapping to the independent variable X), and K and α are model parameters (mapping to u_1 and u_2). For this example, the (column) vectors of observations (x) and independent variable (c) are (specified as code):

```
c = [ 0.162 0.324 0.649 1.297 1.622 2.757 3.730 7.297 ]' ;
x = [ 158 200 265 366 417 500 583 692 ]' ;
```

and an anonymous function that computes prediction error is given by:

```
E = @(U) U(2)*U(1)*c ./ (1+U(1)*c)  -  x;
```

such that the optimized isotherm parameters, starting wit the guess K = 126 and $\alpha = 1$, are obtained using:

```
U = lsqlm( E , [ 126 ; 1 ] );
K = U(1)
alpha = U(2)
```

which produces K = 0.8024 and $\alpha = 772.3$. The fit is illustrated in the figure below.

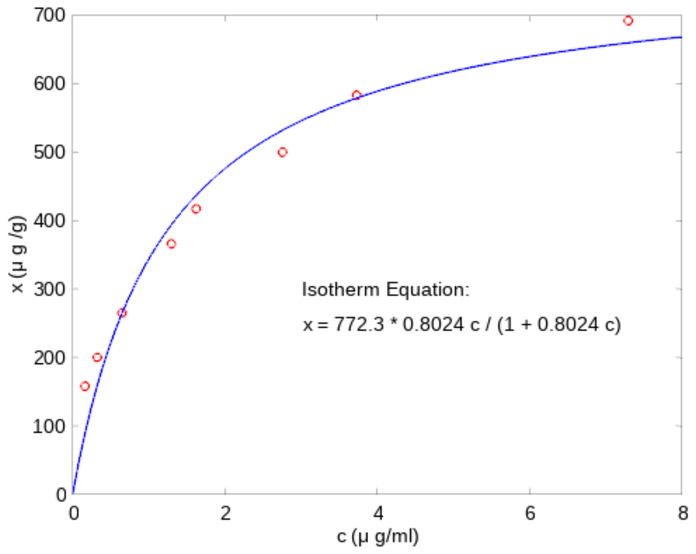

Figure 5. Nonlinear Optimization of Langmuir Isotherm Parameters

The code of the function lsqlm() is essentially a direct implementation of the 10-step algorithm described above. It takes advantage of vector operations for compactness and speed, and uses a finite difference approach to compute the Jacobian. The code is 30 lines long, including comments and diagnostics, and is listed below.

```
function  [U, diagnostic] = lsqlm(fres, U)
% Levenberg-Marquardt nonlinear Least Squares
% fres = handle to model residuals function
% U    = initial vector of model parameters
% (c) 2015, H.J. Montas (University of Maryland)

% constant problem parameters and termination criteria
n          = length(U);% number of model parameters
rtol       = 1e-6;                % termination criterion on sqrt(SSE)
dUtol      = 1e-20;               % termination criterion on dU if U is zero
dUUtol     = 1e-8;                % termination criterion on dU/U
relstep    = 1e-7;                % relative step for finite differences
iter       = 250;                 % max iterations to perform
% initial values for iterations
R          = fres(U);             % initial residuals
rsse       = norm(R);             % sqrt(SSE) for initial residuals
dU         = U;                   % initial update vector (pseudo)
lambda     = 1;                   % initial Levenberg-Marquardt parameter
% iterations
while rsse > rtol & norm(dU) > max(dUtol,dUUtol*norm(U)) & iter
  iter     = iter - 1;            % remaining number of iterations
  % Jacobian calculation (first-order finite differences)
  dUfd     = diag( relstep * max( relstep, abs(U) ) );
  for  k = 1:n
    J(:,k) = (fres(U+dUfd(:,k)) - R)/dUfd(k,k);
  end
  % update of model parameter vector and lambda
  JJ       = J'*J;                % Gauss-Newton matrix
  JR       = J'*R;                % descent vector
  orsse = rsse;                   % previous sqrt(SSE)
  while rsse >= orsse & norm(dU) > max(dUtol,dUUtol*norm(U))
    dU     = (JJ+lambda*eye(n)) \ -JR; % new parameter update vector
    R      = fres(U+dU);          % updated residuals
```

```
      rsse   = norm(R);              % updated sqrt(SSE) of residuals
      lambda = 5*lambda;             % move away from Gauss-Newton
    end
    lambda = lambda/7.5;             % move back towards Gauss-Newton
    U      = U + dU;                 % update parameter vector
  end
  % diagnostic: iterated out = 0, converged in SSE = 1, in dU = 2.
  diagnostic = (rsse <= rtol) + 2*(norm(dU) <= max(dUtol,dUUtol*norm(U)));
```

Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational tools developed from the biomimetic approach to Artificial Intelligence (AI) that process input signals according to simple rules to produced outputs that are aimed to be informative. Examples of their application in the biological resources field, including comparisons to more traditional statistical regression approaches, are detailed in Resop (2006). ANNs commonly rely on a simplified computational model of biological neurons as depicted in Figure 6. The neuron receives synaptic input signals, x, from a set of n upstream neurons, wieghs each by a corresponding weight, W, adds the results, along with a bias, and passes that through a neuronal activation function, f, to produce its output signal, y. Considering the neurons weight vector as one row of a matrix and representing its set of input signals as a column vector of size n+1 (to include the bias) enables us to write the signal-processing computation performed by the model neuron as:

$$y = f\left(\begin{bmatrix} w_1 & w_2 & w_3 & \cdots & w_n & w_{n+1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \right)$$

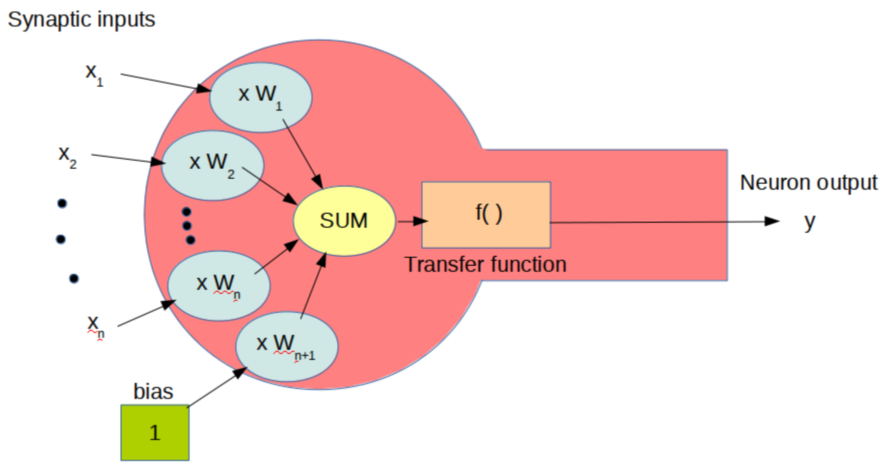

Figure 6. Diagram of a Typical Model Neuron used for ANN Computations

In analogy with biological tissues, ANNs rely on an arrangement of several neurons to perform their computational tasks. The acyclic feed-forward ANN is one of the simplest geometries for this type of system and is depicted in Figure 7. In this type of ANN, neurons are arranged in layers, with full synaptic interconnections between one layer and the next, but no layer bypass and no backwards interconnections. These feed-forward ANNs are referred-to by their number of layers and by the number of neurons in each layer. The ANN depicted in Figure 7, for example, is a 4-layer 3-2-3-2 ANN.

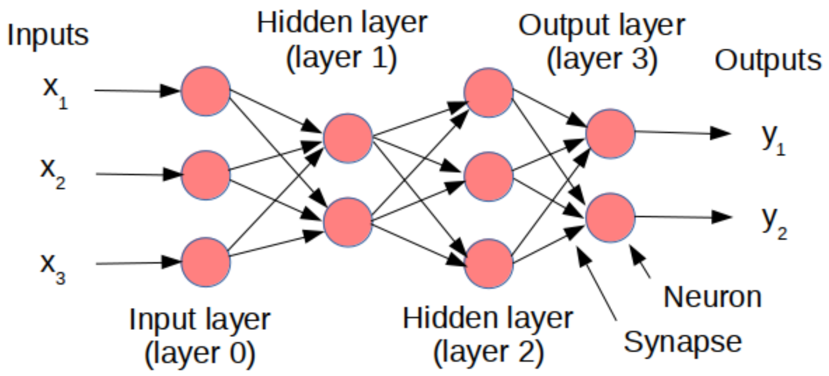

Figure 7. Example of Acyclic Feed-Forward ANN

It is convenient to express the n+1 inputs (including bias) and m+1 outputs (including bias pass-through) of a layer with m neurons in the form of vectors, and to use a m+1-by-n+1 matrix to represent the weights of all neurons in the layer (including the bias weight and pass-through value):

$$Y=\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \\ 1 \end{bmatrix} \quad , \quad X=\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \quad , \quad W=\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,n} & w_{1,n+1} \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,n} & w_{2,n+1} \\ w_{3,1} & w_{3,2} & w_{3,3} & \cdots & w_{3,n} & w_{3,n+1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ w_{m,1} & w_{m,2} & w_{m,3} & \cdots & w_{m,n} & w_{m,n+1} \\ 0 & 0 & 0 & 0 & 0 & \gamma \end{bmatrix}$$

where the pass-through factor, ɣ, is a value such that f(ɣ) = 1. The output of a single layer can then be written:

$$Y=\mathrm{f}(W\ X)$$

and the output of an ANN with q layers can be calculated by iteration, starting with the first layer, as (taking advantage of the fact that the output of one layer is the input to the next):

$$\begin{gathered} X^1=\mathrm{f}\left(W^1\ X^0 \right) \\ X^2=\mathrm{f}\left(W^2\ X^1 \right) \\ \vdots \\ X^k=\mathrm{f}\left(W^k\ X^{k-1} \right) \\ \vdots \\ X^q=\mathrm{f}\left(W^q\ X^{q-1} \right) \end{gathered}$$

The simplicity of the feed-forward ANN makes computations of its output a very efficient process. In order to use this type of tool in practice, one needs to select an appropriate geometry for the network, select appropriate activation functions for each layer and identify an appropriate set of weights. Commonly used activation functions, with corresponding values of the pass-through factor, are presented in Table 1. It is common in practice to use sigmoidal activation functions for the internal layers of the network and to use a linear function for the output layer.

Table 1. Commonly used Activation Functions for ANN Applications

Name	Formula	gamma	df/dx in terms of x	df/dx in terms of f(x)
linear	$f(x)=x$	1	1	N/A
exponential sigmoid	$f(x)=\frac{1}{1+e^{-x}}$	100	$\frac{e^{-x}}{\left(1+e^{-x}\right)^2}$	$f(x)\left(1-f(x)\right)$
hyperbolic tangent sigmoid	$f(x) = \tanh(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}}$	100	$\frac{4}{\left(e^x+e^{-x}\right)^2}$	$1-f(x)^2$

The most challenging aspect of ANN application is the determination of neural weights. In similarity to biological organisms, ANN weights are commonly determined by training the network over a set of p pre-selected input-output data items which we want the network to internalize, and hopefully "comprehend" and generalize as well. During this training process, weights are adjusted iteratively with the aim of reducing the sum of squared errors (SSE) between the ANN's outputs and the reference data, computed over all p input-output training samples and all output nodes. To facilitate these computations, the network is fed with all input samples, simultaneously, as a matrix with p columns, during each forward phase of the training process, producing a matrix of predictions (also with p columns) at the output of the ANN. The weight adjustment formula used during is similar to the parameter update formula derived earlier for Levenberg-Marquardt optimization. For the kth layer of the network, it takes the form:

$$\Delta W^k = -\alpha \left[E^k \ .* \ f'\left(W^k \ TX^{k-1}\right)\right] \left(TX^{k-1}\right)^T$$

where α is a user-selected training speed coefficient (with value between 0 and 1), TX^k is the matrix of outputs of layer k, E^k is the matrix of prediction errors at the output of layer k, and f'() is the derivative of the activation function (as given in the last 2 columns of Table 1). Prediction errors are readily computed at the output of the ANN, by comparison with the training data, but for hidden layers the error is unknown and must be estimated based on output error and network characteristics. The tool used for doing this is the error back-propagation formula (Rumelhart et al., 1986) which produces an estimate of the network's prediction error at the output of layer k from the value of its error at the next downstream layer, k+1:

$$E^k = \left(W^{k+1}\right)^T \left[f'\left(W^{k+1} \ TX^k\right) .* \ E^{k+1}\right]$$

where (here and above) the symbol .* represents the item-wise vector multiplication operator.

The algorithm for training an ANN against an input-output dataset combines all of the above parts as follows:

0) Initialize the ANN's weights using random values
1) Compute the ANN's output by passing the input training set, TX0, through all the layers (sequentially)
2) Compute the residuals and the SSE at the ANN's output (the SSE is a vector with one value per training sample):
3) If the SSEs are below the desired threshold, then stop -- the ANN is trained.
4) Back-Propagate residuals to the previous layer (initially, k = q).
5) Adjust weights in the current layer (initially, k = q)
6) Repeat steps 4 and 5, decreasing k by 1 each time, until k = 1 (1st hidden layer) has been updated
7) go to step 1

A set of compact functions for acyclic feed-forward ANN training and application, named ANN_train() and ANN_apply(), are listed further below. The codes use the hyperbolic tangent sigmoid activation function and the training code considers a RMSE of $1e^{-6}$ or 10,000 iterations (whichever comes first) as termination criteria. The training function takes the training parameter α, network geometry, input set and corresponding target output set as inputs, and returns a cell array of layer weight matrices and error convergence results as output. The application function takes takes a cell array of trained weight matrices and a vector (or matrix) of input data as input and returns the output of the trained ANN for this input data and weights. To examplify their use we consider training a 5-4-5-1 feed-forward ANN to interpolation of a non-smooth dataset relating an observed variable, y, to an independent variable, x, given by:

```
x = [-1.0  -0.8  -0.6  -0.4  -0.2  0.0  0.2  0.4  0.6  0.8  1.0 ];
y = [-0.95 -0.92 -0.85 -0.77 -0.51 0.01 0.09 0.19 0.26 0.35 0.42];
```

The training factor, network geometry and training input and output datasets are defined as:

```
alpha = 0.05;
netgeom = [4 5 1];
X =[x ; x.^2 ; x.^3 ; x.^4 ; x.^5];
Y = y;
```

where the independent variable has been expanded into a set of 5 polynomial values to form the ANN's input (the ANN has 5 input nodes) and the network geometry (netgeom) specifies only the number of neurons in the layers that follow the input layer (the number of neurons in the input layer is inferred from the size of the input dataset). The ANN is trained using:

```
[W,rmse] = ANN_train(alpha,netgeom,X,Y);
```

Figure 8 presents the target dataset and the evolution of the root mean square error (RMSE) of ANN predictions during the training process. The target data has a sudden change of slope, near x = 0, which is challenging to accurately predict with polynomial and trigonometric models. The ANN went through 10,000 training iterations without reaching the target error of 1e^{-6}, but may still have resulted in a usable model. The satisfactoriness of the trained model is to be determined by applying it to an input set that covers a range of interest of values of the independent variable, x.

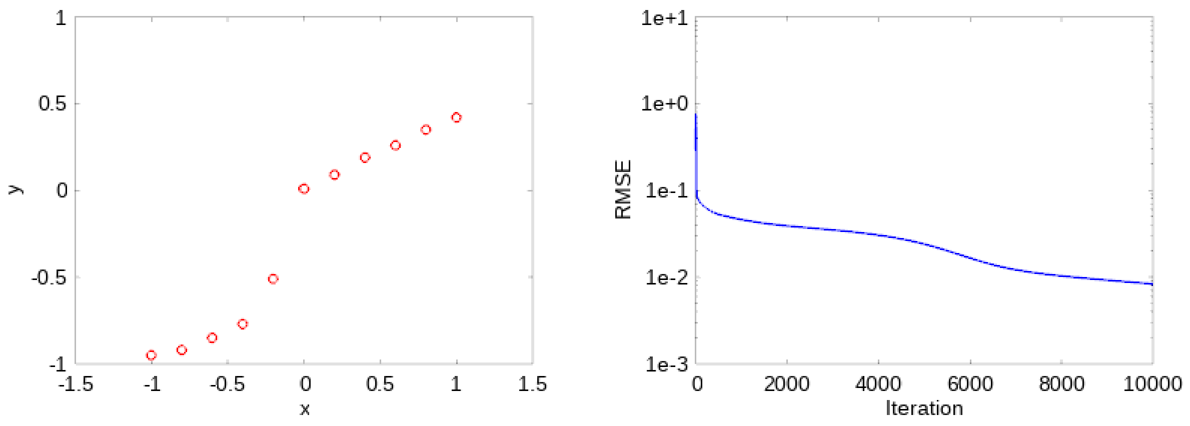

Figure 8. Target Dataset (left) and Evolution of RMSE during a 5-4-5-1 Tanh ANN Predictive Model

The adequacy of the trained ANN model is evaluated by generating a vector of equally spaced values of the independent variable, x, forming the appropriate input matrix, X, from it and using the ANN_apply() function, with trained weights and X as inputs, to predict values of the dependent variable y:

```
xa = linspace(-1.5,1.5,101);
Xa =[xa ; xa.^2 ; xa.^3 ; xa.^4 ; xa.^5];
Ya = ANN_apply(W,Xa);
```

Results are presented in Figure 9. They indicate that the trained ANN does a very good job of interpolating the target dataset for values of the independent variable that are within the range of that dataset. ANN extrapolations for values of x that are outside of the the dataset's range are however less satisfactory, especially for low values of x.

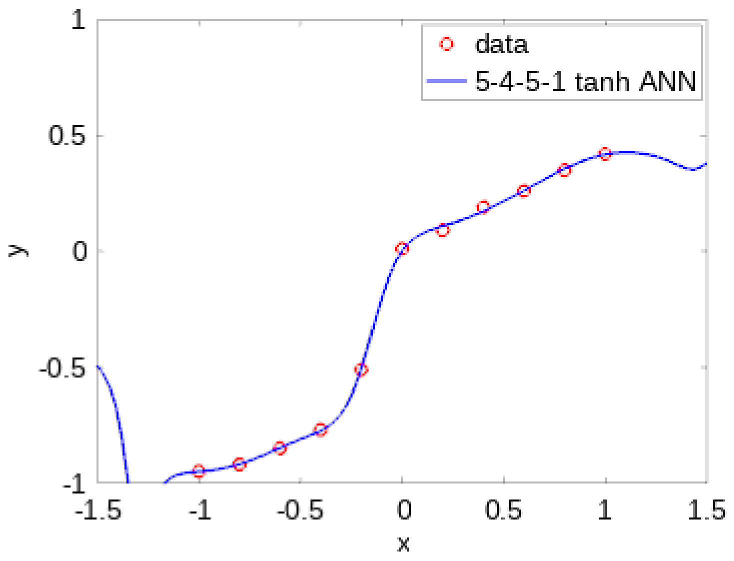

Figure 9. Comparison of the Predictions of the Trained ANN to the Training Dataset

The code of the ANN_train() function is listed below. Despite the relative complexity of the process, it takes just 55 lines (with comments) as it takes advantage of vectored computations and flexible data structures, such as the cell arrrays. A substantial fraction of the code is in the preparation of variables and parameters which give it its flexibility in handling feed-forward ANNs of arbitrary geometry.

```
function [W, rmse] = ANN_train(alph,nnl,X,Y)
%  Multi-layer acyclic feed-forward ANN training
%        alph = training speed, nnl = network geometry
%        X = trainng set inputs,  Y = training set outputs
%        W = cell array of trained layer weights
%        rmse = matrix of root mean square error training history
% initialize variables and parameters
fn   = @tanh;                          % neuron tranfer function (tanh)
dfndx= @(fx) 1 - fx.^2;                % derivative of tranfer function
maxiter       = 10000;                 % max number of training iterations
maxerr         = 1e-6;                 % max allowable rmse for each output
nino = size(X,1);                      % number of input  layer nodes
nono = size(Y,1);                      % number of output layer nodes
ntrs = size(X,2);                      % number of training samples
q    = length(nnl);                    % number of layer (excluding input)
TX0  = [X ; ones(1,ntrs)];             % expanded input  samples (for bias)
TY   = [Y ; ones(1,ntrs)];             % expanded output samples (for bias)
W    = cell(q,1);                      % layer weights, cell array (empty)
TX   = cell(q,1);                      % layer outputs, cell array (empty)
rmse = maxerr*ones(maxiter,nono); % training error array (for output)
% initialize layer weights to random values (range = -1 to 1)
W{1} = [2*rand(nnl(1),nino+1)-1 ; zeros(1,nino) 100];
for k = 2:q
  W{k} = [2*rand(nnl(k),nnl(k-1)+1)-1 ; zeros(1,nnl(k-1)) 100];
end
W{k}(end,end) = 1;                     % gamma corrected for output layer
% perform training iterations
iter  = 0;                             % current iteration
irmse = 1+maxerr;                      % initial error (greater than allowable)
while iter < maxiter & any(irmse > maxerr)
  iter = iter + 1;                     % update current iteration
  % forward computation of layer outputs with new weights
```

```
    TX{1} = fn(W{1}*TX0);                  % layer 1 output
    for k = 2:q-1
      TX{k} = fn(W{k}*TX{k-1});            % layer k output
    end
    TX{q} = W{q}*TX{q-1};                  % layer q output (ANN output)
    % residuals and rmse
    R     = TX{q} - TY;                    % residuals at ANN output
    irmse = sqrt(mean(R(1:end-1,:).^2,2))'; % rmse for this iteration
    rmse(iter,:) = irmse;                  % store rmse (for output)
    % back-propagation of residuals and weight update
    if iter < maxiter & any(irmse > maxerr)
      fpR  = R;                            % common update factor df/dx*R
      R    = W{q}'*fpR;                    % resids back-prop to layer q-1
      W{q} = W{q} - alph*fpR*TX{q-1}';     % updated weights of layer q (output)
      for k = q-1:-1:2
        fpR  = dfndx(TX{k}).*R;            % common update factor df/dx*R
        R    = W{k}'*fpR;                  % resids back-prop to layer k-1
        W{k} = W{k} - alph*fpR*TX{k-1}';% updated weights of layer k
      end
      W{1} = W{1}-alph*(dfndx(TX{1}).*R)*TX0';% updated weights of layer 1
    end
  end
```

By contrast, the ANN_apply() function is extremely short at less than 10 lines with comments, reflecting the simplicity of the process of applying trained ANNs to arbitrary input datasets, using vectored computations:

```
  function Y = ANN_apply(W,X0)
  % ANN application (tanh activation function)
  %     W  = cell array of trained layer weights (matrices)
  %     X0 = ANN input (vector or matrix)
  X = tanh(W{1}*[X0 ; ones(1,size(X0,2))]); % layer 1 output
  for k = 2:length(W)-1
    X = tanh(W{k}*X);                      % layer k output(s)
  end
  Y = W{length(W)}(1:end-1,:)*X;           % ANN output
```

Conclusions

The combination of calculus and linear algebra results in a powerful framework for the analysis and solution of complex engineering problems. Compact computational codes can help to clarify the related methods and also provide efficient implementations for learning and research. As demonstrated here, methods ranging from Runge-Kutta to Artificial Neural Networks can be implemented in 10 to 50 lines of high-level language code. The compactness of the codes enhances their communicability and comprehensibility. The codes listed in this manuscript have been used successfully in educational processes related to courses on Biocomputation Methods (BIOE241) and Spatial Control of Biological Agents (BIOE463) at the University of Maryland at College Park, as well as in research activities. It is hoped that these code sample, or similar ones inspired by this work, characterized by conciseness and clarity, will be useful also to other educators and researchers.

References

Baker, M, 2016. Is There a Reproducibility Crisis? Nature, 533(26 May 2016):452-454.

Boyce, W.E. and R.C. DiPrima, 1986. Elementary Differential Equations and Boundary Value Problems, 4th ed. Wiley & Sons, New York, NY, 654pp.

Hensel, K., 1928. Über den Zusammenhang zwischen den Systemen und ihren Determinanten. Journal für die reine und angewandte Mathematik, 159(9):246-254.

Reddy, J.N. and D.K. Gartling, 2001. The Finite Element Method in Heat Transfer and Fluid Dynamics, 2nd ed. CRC Press, New York, NY, 469pp.

Resop, J.P., 2006. A comparison of Artificial Neural Networks and Statistical Regression with Biological Resources Applications. M.Sc. Thesis, University of Maryland at College Park, College Park, MD, USA.

Rumelhart, D.E., G.E. Hinton and R.J. Williams, 1986. Learning Representations by Back-Propagating Errors. Nature, 323(9):533-536.

Skaggs, R.W. and R. Khaleel, 1982. Infiltration, Chapter 4 in Haan, C.T., Johnson, H.P. and Brakensiek, D.L. (Eds) Hydrologic modeling of small watersheds, ASAE Monograph No. 5, American Society of Agricultural Engineers, Michigan, 1982, 533 pp.

Strang, G., 1988. Linear Algebra and its Applications, third edition. Harcourt Brace Jovanovich Inc., New York, NY, 505pp.

Transtrum, M.K. and J.P. Sethna, 2012. Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization. arXiv preprint arXiv:1201.5885, Cornell University, NY, USA.

Verlet, L, 1967. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. Physical Review, 159(1):98-103.