Crash course in basic Numerical Methods

Dr. M.F. Somers, May 2020.
Theoretical Chemistry group, Leiden University.
m.somers@chem.leidenuniv.nl

Table of content

1) Introduction.
2) Numbers and computers.
3) Solving Linear equations.
4) Interpolating and least square fitting.
5) Differentiating.
6) Integrating.

Introduction

In this course you will be gaining some practical experience in basic numerical methods. You will start out with some basic theory on how numbers are represented on computers. The next chapter will be on solving linear equations because many problems encountered in science turn out to be linear or can be fairly well approximated linearly. Solving a linear equation is of key element in the subsequent chapter on interpolating and least square fitting. The final two chapters of this course are about numerically differentiating and integrating.

This is course is by no means a complete overview of all the possible numerical methods used. It will only cover the most basic principles and most often used tricks. Although the course is rather short, the experience obtained by practically dealing with the algorithms should get you on your way in more advanced courses or books. Students that would like to get further knowledge in the field of numerical methods are very much encouraged to do so by studying, for example, the book 'Numerical Recipes' (http://numerical.recipes/).

Numbers and computers

Computers are often used to calculate things and for therefore the computer must be able to represent numbers in memory. Computers store numbers in bits and thus use a binary system. We humans use a decimal system.

Briefly, considering the number 12345, 12345 actually means $1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$ in the decimal system. Similarly, the number 12345 can also be expressed in the binary system as 11000000111001 which means $1 \times 2^{13} + 1 \times 2^{12} + 0 \times 2^{11} + ... + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. Essentially, when expressing an integer number in any system, you express it in powers of the base 'b' of the system:

$$N=\sum_{i=0}^{n} c_i . b^i \quad \text{with } 0 \leq c_i < b$$

In order to represent not only integer numbers, but also fractional numbers, the powers of the base in the formula above are allowed to take negative values too. So, with b=2, any real number x can be expressed as:

$$x=\sum_{i=-m}^{n} c_i . 2^i \quad \text{with } c_i \text{ either being 0 or 1}$$

This works if both the integers n and m are allowed to go to infinity. The number 3.25 in decimal notation would get the notation 11.01 in the binary system and one would, naively assume that one needs at least 4 bits to represent that number in memory. Moreover, when considering the decimal number 32 which is in binary written as 100000, one would think one needs at least 6 bits to represent that number. Considering an even bigger number, say 12345678901234567890.123456, for which you would need at least 61 bits, one might think, the bigger the number, the more memory or bits one needs to represent it.

However, in the decimal system we are used to use a scientific notation and express 12345 as 0.12345×10^5. In the binary system one can do that equally well: 11.01 can be written as 0.1101×2^2. With this, a need trick can be applied to these so-called 'normalized' binary numbers. Consider any real number in this 'binary scientific notation'; they will all look like $0.1bbbbb \times 2^E$, if the exponential number of 2 (E) is chosen wisely. One can even leave out the bit that is always 1 in the representation! With this, it is possible to even significantly represent very big numbers with a limited number of bits. Clearly, in order to also store and represent negative numbers, an extra bit is needed and thus apart from the the bits needed to store the “b's” above, some bits are needed to store the E exponent and the sign of the number too.

Exercises and Questions:

1) Given the 32 bit single precision IEEE floating point representation details at http://en.wikipedia.org/wiki/IEEE_floating-point_standard;

- *what is the biggest positive number that can be represented?*
- *what is the smallest positive number that can be represented?*
- *what is the smallest number that can be added to 1 such that the correct result of the addition can still be represented? to how many significant figures does this correspond roughly in the decimal system?*

2) Try the following C code;

```
#include <stdio.h>

int main( int *argc, char *argv[] )
{
 double A, B;

 A = 1.0;
 B = 1000;

 while( ( B - A ) < B )
 {
  printf( "%30.26lE %30.26lE\n", B, A );
  A = A / 2.0;
 }
}
```

- *why does the loop stop after a while?*
- *what happens if B is set to 1.0 instead of 1000?*
- *would you be able to determine the 'epsilon' (http://en.wikipedia.org/wiki/Machine_epsilon) of a computer?*

3) What is wrong in the following algorithm that iteratively tries to approach the value of the square root of C using the 'Newton-Raphson' method (http://en.wikipedia.org/wiki/Newton-Raphson_method)?

```
#include <stdio.h>

int main( int *argc, char *argv[] )
{
 double X, dX, C, epsilon;

 epsilon = 1E-35;
 C = 3.0;

 /* try to find the sqrt(C) by finding the root of
    the function Y=X*X-C. the root is found by using
    a newton-raphson method starting with a guess value
    of 1.0  */

 X = 1.0;
 do
 {
  /* calculate dX = dY / f'(X) = -f(X) / f'(X) = -(X*X-C) / 2X */
  dX = - 0.5 * ( X * X - C ) / X;

  X += dX;

  printf( "%30.26lE  %30.26lE\n", dX, X );
 }
 while( fabs( dX / X ) > epsilon );

}
```

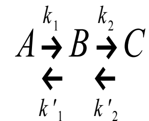
4) According to https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

- *what is a 'guard digit' and why is it used?*
- *what are so-called 'catastrophic cancellation' errors and how are they caused?*
- *which step of the 'Newton-Raphson' method above, at first glance, could potentially introduce such a 'catastrophic cancellation' error?*
- *which operations on the IEEE floating point numbers should be 'exactly*

rounded'?
- does the 'Newton-Raphson' method return stable results when 'epsilon' is set correctly to the machines precision? does the 'catastrophic cancellation' occur?
- what could be done to improve the stability of the method?

Solving Linear Equations

In science a lot of problems can be reduced to a linear system of equations (http://en.wikipedia.org/wiki/System_of_linear_equations). When, for example, considering the following chain of first order reactions between species A, B and C:



the change in concentrations of these species can be expressed as:

$$\frac{d[A]}{dt} = -k_1[A] + k'_1[B]$$

$$\frac{d[B]}{dt} = k_1[A] - (k'_1 + k_2)[B] + k'_2[C]$$

$$\frac{d[C]}{dt} = k_2[B] - k'_2[C]$$

which can be rewritten in matrix vector form like:

$$\frac{d\begin{pmatrix}[A]\\ [B]\\ [C]\end{pmatrix}}{dt} = \begin{bmatrix} -k_1 & k'_1 & 0 \\ k_1 & -(k'_1 + k_2) & k'_2 \\ 0 & k_2 & -k'_2 \end{bmatrix} \begin{pmatrix}[A]\\ [B]\\ [C]\end{pmatrix}$$

When one desires to get the concentrations corresponding to a given change in concentration or reaction speed, one should set the derivative vector and solve for the vector ([A] [B] [C])^T.

You probably have been taught in your basic Linear Algebra classes that solving such a system boils down to reducing the augmented matrix into a 'row reduced echelon' form by elementary row operations:

$$\begin{bmatrix} -k_1 & k'_1 & 0 & \frac{d[A]}{dt} \\ k_1 & -(k'_1+k_2) & k'_2 & \frac{d[B]}{dt} \\ 0 & k_2 & -k'_2 & \frac{d[C]}{dt} \end{bmatrix} \xrightarrow{\text{elementary row operations}} \begin{bmatrix} 1 & 0 & 0 & [A] \\ 0 & 1 & 0 & [B] \\ 0 & 0 & 1 & [C] \end{bmatrix}$$

As can be read at http://en.wikipedia.org/wiki/Reduced_row_echelon_form and http://en.wikipedia.org/wiki/Gauss-Jordan_elimination, the Gauss-Jordan algorithm can be used to perform elementary row operations on the augmented matrix in such a way that the matrix is obtained in the 'row reduced echelon' form. Once the matrix is in that form, the solution vector is easily identified as the augmented vector $([A]\ [B]\ [C])^T$. Although there are faster algorithms to solve linear equations, the Gauss-Jordan algorithm is particularly handy because it allows for an easy calculation of the inverse of a square matrix.

Exercises and Questions:

Read the web-pages http://en.wikipedia.org/wiki/Pivot_element and http://en.wikipedia.org/wiki/Reduced_row_echelon_form and try to implement the Gauss-Jordan algorithm, given the pseudo-code on one of these pages, into the partial C code below. Also try to implement 'partial pivoting' in the method (note that swapping two rows in a matrix is an elementary matrix operation too). Finally test your code by setting up a random square matrix, for which you calculate the inverse by using the Gauss-Jordan algorithm and check the inverse matrix by matrix multiplying it with the original matrix.

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>

/* ----------------------------------------------- */

typedef struct { int R, C;
                 double **Data;
               } Matrix;

/* ----------------------------------------------- */

Matrix AllocateMatrix( int R, int C )
{
 Matrix M;
 int i;

 M.Data = NULL;
 M.R = M.C = 0;

 if( ( R <= 0 ) || ( C <= 0 ) ) return( M );

 M.R = R;
 M.C = C;
 M.Data = calloc( R, sizeof( double * ) );

 for( i = 0; i < R; ++i )
  M.Data[ i ] = calloc( C, sizeof( double ) );

 return( M );
}
```

```
/* ------------------------------------------------ */

void FreeMatrix( Matrix M )
{
 int i;

 if( ( !M.R ) || ( !M.C ) || ( M.Data == NULL ) ) return;

 for( i = 0; i < M.R; i++ )
  free( M.Data[ i ] );

 free( M.Data );
}

/* ------------------------------------------------ */

void CopyMatrix( Matrix A, Matrix B )
{
 int i, j;

 if( ( A.Data == NULL ) || ( B.Data == NULL ) ) return;

 if( ( A.R != B.R ) || ( A.C != B.C ) ) return;

 for( i = 0; i < A.R; i++ )
  for( j = 0; j < A.C; j++ )
   B.Data[ i ][ j ] = A.Data[ i ][ j ];
}

/* ------------------------------------------------ */

void PrintMatrix( Matrix M )
{
 int i, j;

 if( M.Data == NULL ) return;

 for( i = 0; i < M.R; i++ )
 {
  for( j = 0; j < M.C; j++ )
   printf( "%20.12lE", M.Data[ i ][ j ] );
  printf( "\n" );
 }
 printf( "\n" );

}

/* ------------------------------------------------ */

void AddMatrices( Matrix A, Matrix B, Matrix Result )
{
 int i, j;

 if( ( A.Data == NULL ) || ( B.Data == NULL ) || ( Result.Data == NULL ) ) return;
 if( ( A.R != B.R ) || ( A.R != Result.R ) ) return;
 if( ( A.C != B.C ) || ( A.C != Result.C ) ) return;

 for( i = 0; i < A.R; i++ )
  for( j = 0; j < A.C; j++ )
   Result.Data[ i ][ j ] = A.Data[ i ][ j ] + B.Data[ i ][ j ];
}

/* ------------------------------------------------ */

void MultiplyMatrices( Matrix A, Matrix B, Matrix Result )
{
 int i, j, k;

 if( ( A.Data == NULL ) || ( B.Data == NULL ) || ( Result.Data == NULL ) ) return;
 if( ( B.R != A.C ) || ( Result.C != B.C ) || ( Result.R != A.R ) ) return;

 for( i = 0; i < Result.R; ++i )
  for( j = 0; j < Result.C; ++j )
   for( Result.Data[ i ][ j ] = 0.0, k = 0; k < B.R; ++k )
    Result.Data[ i ][ j ] += A.Data[ i ][ k ] * B.Data[ k ][ j ];

}

/* ------------------------------------------------ */

void DoGaussJordan( Matrix A, Matrix B )
{

 if( ( A.Data == NULL ) || ( B.Data == NULL ) ) return;
 if( A.R != B.R ) return;

 /* do gauss jordan algorithm on matrix A using pivoting and apply
    all elementary row operations performed on A on B too. */

}

/* ------------------------------------------------ */

int main( int *argc, char *argv[] )
{
```

```
 /* write your test code here to fill in a random matrix, calculate the
    inverse of it and then multiply the original matrix with its inverse
    to test your code. make use of the functions already implemented
    for you above. */

}
```

Interpolating and least square fitting

Quite often a scientist has a set of data only for specific values and he or she needs to have data for intermediate values too. A simple solution would be to draw a graph of the data to 'fiddle' a line through the points and use that line to estimate the intermediate values. A more advanced numerical solution would be to get the two nearest data points and connect them with a straight line. One could then figure out the equation of that line and use the actual equation to get the intermediate values.

Clearly, an even better method would be to get three or four points close to the wanted intermediate value, try to find the polynomial that exactly connects the data points and then use the found polynomial to calculate the intermediate values. One could even go a step further and decide that polynomials aren't all that a good an approximation, despite the well-known Taylor series expansions being convergent and all, and use a general linear combination of basis functions:

$$Y(x) \approx Y_{int}(x) = \sum_{i=0}^{N-1} C_i \cdot F_i(x)$$

Here the value of the function Y at the point x is being approximated by a linear combination of basis functions F_i with C_i being the expansion coefficients. Now for each actual known data-pair (X_j, Y_j), $Y_j = Y_{int}(X_j)$ applies and the above equation can be rewritten to the matrix vector equation:

$$\begin{pmatrix} F_{N-1}(X_{N-1}) & F_{N-2}(X_{N-1}) & \dots & F_0(X_{N-1}) \\ F_{N-1}(X_{N-2}) & F_{N-2}(X_{N-2}) & \dots & F_0(X_{N-2}) \\ \vdots & \vdots & \dots & \vdots \\ F_{N-1}(X_0) & F_{N-2}(X_0) & \dots & F_0(X_0) \end{pmatrix} . \begin{pmatrix} C_{N-1} \\ C_{N-2} \\ \vdots \\ C_0 \end{pmatrix} = \begin{pmatrix} Y_{N-1} \\ Y_{N-2} \\ \vdots \\ Y_0 \end{pmatrix}$$

Finding the linear expansion coefficients C_i is again equivalent to solving a set of linear equations. Just by calculating the correct 'collocation' matrix and by augmenting it, the Gauss-Jordan algorithm can again be used to get the so-called 'interpolating linear basis set expansion'.

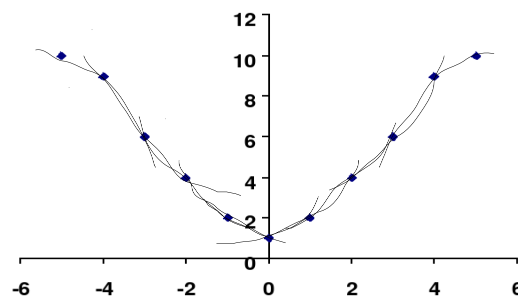
Exercises and Questions:

Write down the collocation matrix for a polynomial basis set and write a C function that computes the expansion coefficients for a given set of data pairs. Use the code and algorithms from the previous chapter and blue-print given below for your function. Test your code by interpolating analytical polynomials and with the interpolation of the function f(x)=tan(x) by using 10 data points in the x range of [0,pi/2).

```
double PolynomialValue( Matrix P, double X )
{
 double pX, Y;
 int i;

 for( i = 0, pX = 1.0, Y = 0.0; i < P.R; i++ )
 {
  Y += P.Data[ i ][ 0 ] * pX;
  pX *= X;
 }

 return( Y );
}

/* ------------------------------------------------ */

Matrix InterpolatingPolynomial( int N, double X[], double Y[] )
{
 Matrix M = AllocateMatrix( N, N );
 Matrix P = AllocateMatrix( N, 1 );

 /* build up and fill in the M matrix here
    look at the PolynomialValue routine to
    see how to fill in the matrix and the
    P vector exactly */

 DoGaussJordan( M, P );
 FreeMatrix( M );
 return( P );
}
```

In the previous exercise you probably have seen that interpolating the tan(x) function didn't work that well. At http://en.wikipedia.org/wiki/Runge's_phenomenon you can read some more on the problems associated with using a straightforward 'global' polynomial interpolation scheme. The polynomial always goes exactly through the data points on which is was based, but does not necessarily approximate the function very well in between. It is often found to oscillate heavily between the data points.

Despite the failure of using higher order polynomials on the whole interval for which data is available, one might try and 'locally' approximate the data by a 2nd or 3rd order polynomials:



Exercises and Questions:

Add a routine to your code that, given a set of N (X_j,Y_j) data pairs, searches the (m+1) closest data points in X_j to X, calculates the mth order interpolating polynomial going exactly through those (m+1) data points and finally uses that polynomial to get an interpolated value for Y(X) for the intermediate X. Base your code on the routines you already have and the blue-print below. Test your routine by using the same 10 data points of the f(x)=tan(x) function of the previous exercise by using a 3rd order local polynomial approximation. Calculate the mean-square deviation S for at least 100 values of x within the range [0,pi/2):

$$S=\frac{\sum_{i=0}^{N-1}\left(Y_{int}(x_i)-Y(x_i)\right)^2}{N}$$

Also compute the deviation for higher and lower orders of local approximations and see which order does best in general for several analytical functions like sin(x), ln(x), sqrt(x) and exp(x).

```
double LocalPolynomialInterpolation( int Order, int N, double X[], double Y[], double AtX )
{
 Matrix M = AllocateMatrix( Order + 1, Order + 1 );
 Matrix P = AllocateMatrix( Order + 1, 1 );
 int IndexInX[ Order + 1 ];
 double Result;

 /* put algorithm here that finds the Order+1 closest points to AtX in X
    and store the indices of these points into the IndexInX array. then
    build the Order + 1 square collocation matrix, just like you did in
    the subroutine InterpolatingPolynomial. make use of the IndexInX array
    to construct that matrix and fill in the P vector. keep in mind the
    precise ordering of the elements so that the PolynomialValue routine
    can be used as below */

 DoGaussJordan( M, P );

 FreeMatrix( M );

 Result = PolynomialValue( P, AtX );

 FreeMatrix( P );

 return( Result );
}
```

In the previous exercise you probably found out that in general the 3rd order local polynomial approximation in interpolating does reasonably well in interpolating smoothly and continuously between the data points. However, the method does not guarantee that the first-derivative is continues nor can it be used to extrapolate. This is easily understood when considering an increasing X value: as soon as an approximation is computed for some larger X value, causing different data pairs to be used to compute the interpolating polynomial, a different polynomial will be obtained. There is no guarantee that the new polynomial and the previous one have the same derivative at the X value being crossed. If one would like to have the guaranteed continuity of the first and higher derivatives too, a 'spline' algorithm could be used: http://en.wikipedia.org/wiki/Spline_interpolation. Often the 'cubic-spline' method already suffices.

Another approach, to also be able to extrapolate, often employed to represent data, is by 'fitting' the data pairs (http://en.wikipedia.org/wiki/Curve_fitting). The difference between interpolating and fitting is that with interpolating it is guaranteed that a

method is used that exactly reproduces the initial data pairs. With fitting, there is no such guarantee. None the less, often it is known beforehand, because of other scientific reasons, that the data pairs should follow a certain, perhaps non-linear, analytical expression, with some undetermined parameters. The challenge is then to find the best set of parameters so that the 'fit' is best. Once found, the expression can easily be used to extrapolate the data.

When considering 'linear fitting', a linear expression is used, but this time with many more data points than coefficients in the expression:

$$Y_{fit}(x) = \sum_{i=0}^{m<N} C_i \cdot F_i(x)$$

Finding the 'best' coefficients C_i can done by minimizing the 'error' made by the fit:

$$LSQ = \sum_{i=0}^{m<N} \left(Y_i - Y_{fit}(X_i)\right)^2$$

Simply by setting the partial derivatives of the above LSQ error expression with respect to the coefficients C_i to zero leads to yet again a linear system to be solved.

Exercises and Questions:

Express the partial derivative of LSQ with respect to C_n in terms of Y_i and $F_i(x)$ by using the chain rule. Set this derivative to zero and rewrite the expression in such a way that only the terms involving Y_i are on the right-hand side of the equation. Now show that the matrix and the augmented vector to be solved with the Gauss-Jordan algorithm to find the best fitting coefficients, have the following elements:

$$M_{r,c} = \sum_{j=0}^{N-1} F_r(X_j) F_c(X_j) \quad and \quad P_r = \sum_{j=0}^{N-1} Y_j F_r(X_j)$$

Have a look at http://en.wikipedia.org/wiki/Linear_least_squares_%28mathematics%29 and find expressions for the M matrix and P vector when $F_n(X)=X^n$. Notice which powers are needed for the matrix elements and implement a routine that can fit a given data set to a polynomial of given order. Use the blue-print below as a starting point and test the routine by fitting a data set generated from a few random polynomials:

```
Matrix LeastSquareFitPolynomial( int Order, double X[], double Y[], int N )
{
 Matrix M, P;
 double tX;
 int n, i;

 M.Data = NULL;
 M.C = M.R = 0;

 if( Order + 1 >= N ) return( M );

 /* build the P vector first with all powers of data needed */

 P = AllocateMatrix( 2 * Order + 1, 1 );

 for( i = 0; i < 2 * Order + 1; i++ )     /* set vector to zero first */
  P.Data[ i ][ 0 ] = 0.0;

 for( i = 0; i < N; ++i )            /* loop over all X values */
 {
  /* fill from 0 to 2 * Order + 1 by adding the needed powers into the P vector */

  for( tX = 1.0, n = 0; n < 2 * Order + 1; n++ )
  {
   P.Data[ n ][ 0 ] += tX;
   tX *= X[ i ];
  }
 }

 /* start filling the M matrix now with the powers of the data
    we computed already in the P vector above taking care that
    we want the result vector to be usable by the PolynomialValue
    routine at end end */

 M = AllocateMatrix( Order + 1, Order + 1 );

 for( i = 0; i < Order + 1; ++i )
  for( n = 0; n < Order + 1; ++n )
   M.Data[ i ][ n ] = P.Data[ ... ][ 0 ];

 /* free temp stuff not needed anymore and setup true P vector */

 FreeMatrix( P );
 P = AllocateMatrix( Order + 1, 1 );

 for( i = 0; i < Order + 1; i++ )     /* set vector to zero first */
  P.Data[ i ][ 0 ] = 0.0;

 for( i = 0; i < N; ++i )
 {
  for( tX = Y[ i ], n = 0; n < Order + 1; ++n )
  {
   P.Data[ ... ][ 0 ] += tX;
   tX *= X[ i ];
  }
 }

 /* matrices are filled now... so solve and return poly */

 DoGaussJordan( M, P );
 return( P );
}
```

The least square fitting method can easily be extended to fit functions with multiple variables (V(x,y,z,...)) because the basis functions used in the linear expression can be chosen to be functions of those several variables x,y,z,... One can even decide to use a set of basis functions based on a 'direct product' of basis functions for each variable: $F_{klm}(x,y,z,...)=f_k(x)g_l(y)h_m(z)...$ One just needs to fill the M and the P matrices according to the given expressions and solve for the best coefficients.

Differentiating

Problems often encountered in science relate to differentiating and integrating multi-dimensional functions; when calculating the force or gradient of a multi-dimensional potential energy surface in a molecular dynamics simulation, or perhaps even a hessian (http://en.wikipedia.org/wiki/Hessian_matrix) to do a normal mode analysis (http://en.wikipedia.org/wiki/Normal_mode) of a big molecule, or by solving a differential equation like the Newton equation of motion for a molecular system.

When trying to differentiate a function f(x) numerically, the simplest method one could try is the 'two-point finite difference method' (http://en.wikipedia.org/wiki/Finite_difference). In this method, the function f(x) is evaluated at two very close by points ($x=x_0-\triangle x/2$ and $x=x_0+\triangle x/2$) of the point of interest (x_0) and the difference of the function ($\triangle f=f(x_0+\triangle x/2)-f(x_0-\triangle x/2)$) is then divided by the difference between the two points ($\triangle x$): $df/dx \approx \triangle f/\triangle x$. Clearly, the smaller $\triangle x$, the better the approximation.

As it turns out, if you carefully choose the points around the point of interest x_0 like $x=x_0-\triangle x/2$ and $x=x_0+\triangle x/2$, this 'two-point finite difference' method, based on the so-called 'central difference', is accurate to second order $O((\triangle x)^2)$. This means that the error made by the approximation, would only appear in the second order term of the Taylor expansion of the function f(x) around x_0, but with linear and quadratic functions being differentiated exactly by this method. The latter is easily understood by using the Taylor expansion of f(x) around $x=x_0$:

$$f(x_0+n\triangle x/2) = f(x_0) + (n\triangle x/2)f'(x_0)/1! + (n\triangle x/2)^2 f''(x_0)/2! + (n\triangle x/2)^3 f'''(x_0)/3! + ...$$

By taking n=1 and n=-1, the two resulting equations can be subtracted from each other to obtain:

$$f(x_0+\triangle x/2)-f(x_0-\triangle x/2) = (\triangle x)f'(x_0)/1! + (\triangle x)^3 f'''(x_0)/3! + ...$$

Now by simply dividing this equation by ($\triangle x$), you see that you get an expression for $f'(x_0)$ with an error term of order $(\triangle x)^2$ with third or higher order derivatives of f(x), which for linear and quadratic functions are always zero. When taking the limit ($\triangle x \rightarrow 0$), the derivative of f(x) at $x=x_0$ is computed with a quadratic converging method for other functions.

Exercises and Questions:

Use the previously given expression for $f(x_0+n\triangle x/2)$ with n=2,1,-1 and -2 to derive an $O((\triangle x)^4)$ numerical approximation to $f'(x_0)$ using four function evaluations. Make a little test program that computes f'(x) using the two-point and the four-point methods for $f(x)=x^2+x-1$, $f(x)=x^3-x^2+x-1$ and $f(x)=x^4-x^3+x^2-x+1$. Let your program use $\triangle x=1$, $\triangle x=0.1$, $\triangle x=0.01$, $\triangle x=0.001$ ... up to $\triangle x=0.0000001$.Which approximation converges faster when $\triangle x \rightarrow 0$ and which polynomials are differentiated exactly by what method?

In the previous exercise you learned that by using equally spaced function values, the derivative of the function at a single point can be computed rather easily. The method

given can also be applied to deriving expressions for higher order derivatives of functions and when dealing with analytical functions, or with non-discretized functions, these 'finite difference methods' work rather well. The method can also be extended to functions of several variables. However, when dealing with a data set or a discretized function, there might be a problem. It can happen that you do not have all the correct points you need. To solve this, one could use the 'local polynomial approximation' again.

Exercises and Questions:

Rewrite the function ' LocalPolynomialInterpolation' to compute the derivative of the found local interpolating polynomial exactly and use this to get an approximation to f'(x) at the point of interest. Hint: write down the coefficients vector of a general polynomial, then analytically differentiate the polynomial and see how the coefficients vector of the derivative polynomial can be computed. Give this new routine the name 'LocalPolynomialDerivativeApproximation' and try it on a data set obtained from the functions $f(x)=\sin(x)$, $f(x)=\tan(x)$ *and* $f(x)=x^3+x^2-x-1$.

The above demonstrated method can be easily extended to several dimensions. By approximating the many-variable function as a linear combination of powers of each of the variables the function has, a set of linear equations can be setup and solved. With the solution, the (partial) derivative can now be easily approximated.

Integrating

In molecular dynamics, one not only needs to differentiate numerically to obtain forces on particles, one also needs to use an equation of motion to simulate the dynamics. Running such a trajectory involves integrating a differential equation and it is clear that numerical techniques to approximate integrals or solutions to differential equations are needed. This final chapter deals with that and it will show some often used techniques to approximate the general integral (http://en.wikipedia.org/wiki/Integral):

$$I=\int_a^b f(x)\,dx$$

The most trivial method to approximating this integral is by using the very definition of an integral (http://en.wikipedia.org/wiki/Riemann_integral). With the Riemann definition, the integral can be approximated by a summation with N→∞:

$$I\approx\frac{(b-a)}{N}\sum_{n=0}^{N-1} f\left(\frac{(b-a)n}{N}+a\right)$$

Exercises and Questions:

Write a function that computes the integral of an arbitrary function for given a, b and N using the Riemann-sum. Compare the approximated integral with the analytical integrals of the functions f(x)=x-1, f(x)=x^2-x+1, f(x)=sin(x) and f(x)=tan(x) from x=-1 to x=2.Which integrals are computed exactly for N=1? Which values of N are needed to approximate the integrals within $1.0x10^{-6}$?

As can be read at http://en.wikipedia.org/wiki/Numerical_integration, the Riemann-sum approach to the integral can be improved upon by considering small 'trapezoids' instead of simple bars for the intervals. One can easily prove that by weighting the f(a) and f(b) points of the normal Riemann-sum with ½ instead of 1, the accuracy is improved (http://en.wikipedia.org/wiki/Trapezoidal_rule).

Exercises and Questions:

Adapt the previously written function to include the correct weighting of ½ for f(a) and f(b) so that the 'Trapezoidal' method is used. Re-try the integrals of the functions and what do you notice? Which integrals are now computed exactly for N=1? Which values of N are needed to approximate the integrals within $1.0x10^{-6}$?

When adopting the 'Trapezoidal rule', f(x) is approximated by a line between each of the points of a small interval. This linearity is then used to compute the integral of each of the intervals and all contribution are added together.

Clearly, when using a quadratic approximation of the function in each small interval (http://en.wikipedia.org/wiki/Simpson's_rule), an even better approximation might be obtained. In general, several of these 'Newton-Cotes' rules (http://en.wikipedia.org/wiki/Newton-Cotes_formulas) have been constructed in

which higher order polynomials are used to approximate f(x) within each interval. Although these higher-order rules sometimes do provide better accuracy, they also involve more function evaluations. This can be computationally costly, similarly to simply increasing N in 'composite' rules. Often these higher-order rules also suffer from the 'Runge' effect (http://en.wikipedia.org/wiki/Runge's_phenomenon) and different methods are used to get better approximations to the integral.

When inspecting all the above mentioned 'rules', a different approach could be to approximate the integral by a weighted summation of function values at specific points:

$$I \approx \sum_{n=0}^{N-1} w_n f(x_n)$$

When considering these types of 'quadratures', one can now freely, and thus wisely, choose the weights of the summation and the points at which f(x) is evaluated. Depending on the specific function f(x) and integration interval, optimized weights w_n and points x_n can be found: http://en.wikipedia.org/wiki/Gaussian_quadrature.

Exercises and Questions:

Implement a function that computes the integral of f(x) from [-1,2] using only three function evaluations using a Gaussian Quadrature. Try to integrate the functions of the previous exercises and determine to which order polynomials this routine is still exact. Adapt the function to allow for the integration from [a,b] instead of [-1,2].

The 'quadrature' principle can be extended to multi-dimensional integrals and different 'Monte-Carlo' methods have been adopted (http://en.wikipedia.org/wiki/Monte_Carlo_integration).

Another approach to calculating the general integral I is by reformulating the integral into a differential equation (http://en.wikipedia.org/wiki/Initial_value_problem):

$$I = \int_a^b f(x)\,dx \;\rightarrow\; \frac{\partial I}{\partial x} = f(x) \quad with \quad I(a) = 0$$

By solving the differential equation for I and computing I(b), the original integral I is computed. The 'Euler' method (http://en.wikipedia.org/wiki/Euler_method) is one of the simplest method to solving such a differential equation.

Exercises and Questions:

Write a function that implements the 'Euler' method and computes the integral of f(x) for given interval [a,b] and number of intervals N. Compute the integrals of the previously inspected functions for the range [-1,2].Which functions are integrated exactly? What values of N are needed to obtain an accuracy of at least $1.0x10^{-6}$?

Although the 'Euler' method is a very basic method to solve differential equations, it

can be improved upon and several 'Runge-Kutta' methods are available (http://en.wikipedia.org/wiki/Runge-Kutta_method and http://en.wikipedia.org/wiki/List_of_Runge-Kutta_methods). The 'RK4' method essentially uses the 'Euler' method in a more iterative scheme to achieve faster convergence.

Exercises and Questions:

Write a function that implements the 'RK4' method to compute the integrals studied so far. Which functions are integrate analytically and what values of N are now needed to achieve an accuracy of at least $1.0x10^{-6}$*?*

The 'Runge-Kutta' methods are often used to integrate an equation of motion. Other often used integrators are the 'Verlet' (http://en.wikipedia.org/wiki/Verlet_integration) methods.