NAME

hydro − Hydrographic water property calculations

SYNOPSIS

#include <odflib.h>
#include <math.h>

double BruntVaisalaFreq(int *nObs,* **double** **s,*
int *sInc,* **double** **t,*
int *tInc,* **double** **p,*
int *pInc,* **double** **pPav,*
double **pE***);**

double BunsenO2(double *s,* **double** *t***);**

double CFC11Saturation(double *theta,* **double** *salt***);**

double CFC12Saturation(double *theta,* **double** *salt***);**

void CFCSetAirVals(double *f11AirVal,* **double** *f12AirVal***);**

double CarterCorrectedDepth(double *uncorr,* **int** *area,*
int **pCorr***);**

double CarterUnCorrectedDepth(double *uncorr,* **int** *area,*
int **pCorr***);**

void Depth(int *n,* **double** *g,*
double **p,* **int** *pInc,*
double **rho,* **int** *rInc,*
double **depth,* **int** *dInc***);**

void DynamicHeight(int *n,* **double** **p,*
int *pInc,* **double** **spVolAn,*
int *spVolAnInc,* **double** **dynHt,*
int *dynHtInc***);**

double ExpDelayCalc(ExpDelay *pE,* **int** *missing,*
double *val***);**

ExpDelay ExpDelayInit(double *delayPeriod,* **double** *lagPeriod,*
double *sampPeriod,* **double** *initVal***);**

double ExpDelayReset(ExpDelay *pE,* **double** *initVal***);**

double FOfProb(double *p,* **int** *df1,*
int *df2***);**

int FindCarterArea(double *degLat,* **double** *degLon***);**

double *GaussianWeights(int *n***);**

double Gravity(double *deglat,* **double** *pdb***);**

double GreatCircleDist(double *slt,* **double** *slg,*
double *xlt,* **double** *xlg,*
double **course***);**

void GreatCirclePos(double *dist,* **double** *course,*
double *slt,* **double** *slg,*
double **xlt,* **double** **xlg***);**

double IESRho(double *s,* **double** *t,*
double *rp***);**

double IESSpVolAn(double *s,* **double** *t,*
double *p,* **double** **sigma* **);**

double IPTS68Temp(double *t90* **);**

double ITS90Temp(double *t68* **);**

void Interpolate(int *nPts,* **float** **xp,*
int *incrX,* **float** **yp,*
int *incrY,* **int** *iPts,*
float **xi,* **int** *incrXi,*
float **yi,* **int** *incrYi* **);**

int LMnlFit(int *(*pInitFunc)(),* **int** *(*pCalcFunc)(),*
int *m,* **int** *n,*
double **x,* **double** **fvec,*
int *maxfev,* **int** *printInterval,*
int **nfev* **);**

double O2MllToPartPress(double *o2mlpl,* **double** *s,*
double *t,* **double** *p* **);**

double O2PartPressToMll(double *o2pp,* **double** *s,*
double *t,* **double** *p* **);**

double O2PerKgToPerLiter(double *o2umpkg,* **double** *rho* **);**

double O2PerLiterToPerKg(double *o2mlpl,* **double** *rho* **);**

double O2Saturation(double *t,* **double** *s* **);**

double O2SaturationPartPress(double *s,* **double** *t,*
double *p* **);**

double PSS78Conductivity(double *press,* **double** *temp,*
double *salt* **);**

double PSS78Pressure(double *temp,* **double** *cond,*
double *salt* **);**

double PSS78Salinity(double *press,* **double** *temp,*
double *cond* **);**

double PSS78Temperature(double *press,* **double** *cond,*
double *salt* **);**

int PolynomialFit(int *opcode,* **int** *maxfit,*
int *order,* **double** **fit,*
double **weights,* **double** **r2,*
int *nPts,* **double** **x,*
double **y* **);**

double PotentialTemp(double *s,* **double** *t,*
double *p,* **double** *rp* **);**

double Poynting(double *s,* **double** *t,*
double *p* **);**

double ProbOfF(double *f,* **int** *df1,*
int *df2* **);**

double ProbOfR(double *r,* **int** *n* **);**

double ProbOfZ(double *z* **);**

double PtoZ(double *grav,* **double** *dynz,*
double *prsdb* **);**

double RunningMeanCalc(RunningMean *pR,* **double** *val* **);**

RunningMean RunningMeanInit(int *n,* **double** *initVal* **);**

double RunningMeanReset(RunningMean *pR,* **double** *initVal* **);**

double SoundVelCandM(double *s,* **double** *t,*
double *p* **);**

double SoundVelDelGrosso(double *s,* **double** *t,*
double *p* **);**

double SoundVelWilson(double *s,* **double** *t,*
double *p* **);**

double SoundVelocity(int *method,* **double** *s,*
double *t,* **double** *p* **);**

void Transport(int *n,* **double** **z,*
int *zInc,* **double** **dynHt,*
int *dInc,* **double** **transport,*
int *tInc* **);**

double ZOfProb(double *p* **);**

double ZtoP(double *grav,* **double** *dynz,*
double *depth* **);**

double betacf(double *a,* **double** *b,*
double *x* **);**

double betai(double *a,* **double** *b,*
double *x* **);**

double gammln(double *xx* **);**

int isinf(double *d* **);**

double quiet_nan(*void* **);**

MT-LEVEL

MT-unsafe.

DESCRIPTION

hydro – Hydrographic water properties and misc calculations

These functions perform various calculations of hydrographic water properties.

double BruntVaisalaFreq(int *nObs,* **double** **s,* **int** *sInc,* **double** **t,* **int** *tInc,* **double** **p,* **int** *pInc,* **double** **pPav,* **double** **pE* **);**

This function calculates the Brunt-Vaisala (buoyancy) frequency in cycles/hour. **nObs** is the number of data points in the supplied s, t and p series. ***s**, ***t**, and ***p** are the salinity (PSUs), temperature (Celsius) and pressure (decibars) series, respectively. **sInc**, **tInc**, and **pInc** are byte increments to successive array elements of the salinity, temperature and pressure series. ***pPav** is the returned mean pressure for the series interval (decibars). ***pE** is the returned stability (N**2) in radians/second. This function uses a gaussian-weighted filter to smooth the calculated density gradient.

double BunsenO2(double *s,* **double** *t* **);**

This function calculates the Bunsen absorption coefficient for oxygen. **s** is salinity (PSU) and **t** is temperature (Celsius).

double CFC11Saturation(double *theta,* **double** *salt***);**

This function calculates the cfc-11 saturation value (pM/Kg). **theta** is a Potential temperature (Celsius), **salt** is the salinity (PSU).

double CFC12Saturation(double *theta,* **double** *salt***);**

This function calculates the cfc-12 saturation value (pM/Kg). **theta** is a Potential temperature (Celsius), **salt** is the salinity (PSU).

void CFCSetAirVals(double *f11AirVal,* **double** *f12AirVal***);**

This function allows you to set the blank (air) values for cfc-11 and cfc-12 used by **CFC11Saturation**() and **CFC12Saturation**(). **f11AirVal** is the cfc-11 blank value (pM/kg), and **f12AirVal** is the cfc-12 blank value (pM/kg).

double CarterCorrectedDepth(double *uncorr,* **int** *area,* **int** **pCorr***);**

This function returns the Carter Table-corrected depth (Meters) for an uncorrected PDR depth. **uncorr** is the uncorrected depth (nominal meters), **area** is an area index returned by **FindCarterArea()**, and ***pCorr** (if not NULL) will contain the amount of correction applied.

double CarterUnCorrectedDepth(double *corr,* **int** *area,* **int** **pCorr***);**

This function returns the uncorrected PDR depth for a Carter Table-corrected depth (Meters). It is the inverse function of **CarterCorrectedDepth()**. **corr** is the corrected depth (meters), **area** is an area index returned by **FindCarterArea()**, and ***pCorr** (if not NULL) will contain the amount of correction removed.

void Depth(int *n,* **double** *g,* **double** **p,* **int** *pInc,* **double** **rho,* **int** *rInc,* **double** **depth,* **int** *dInc***);**

Note that this function is provided for completeness, but has been superseded by **PtoZ** in normal use. This function computes depth in meters by integration of insitu density with respect to pressure. **n** is the length of the **p, rho** and **depth** series, **g** is the local acceleration of gravity (m/sec**2) @ 0.0 decibars pressure, ***p** is the pressure series in decibars, **pInc** is the byte increment for the pressure series, ***rho** is the insitu density series (kg/m**3), **rInc** is the byte increment for the insitu density series, ***depth** is the output depth series in meters, **dInc** is the byte increment for the depth series. If the initial pressure in the interval is > 15.0 decibars, depth is calculated relative to the initial pressure (the first depth interval is set to 0.0 meters).

void DynamicHeight(int *n,* **double** **p,* **int** *pInc,* **double** **spVolAn,* **int** *spVolAnInc,* **double** **dynHt,* **int** *dynHtInc***);**

This function calculates a dynamic height series (dynamic meters) from pressure and specific-volume anomoly series. **n** is the length of the series, ***p** is the pressure series (decibars), **pInc** is the byte increment of the pressure series, ***spVolAn** is the specific-volume anomoly series (m**3/kg), **spVolAnInc** is the byte increment of the specific-volume anomoly series, ***dynHt** is the output dynamic height series (dynamic meters), **dynHtInc** is the byte increment of the dynamic height series.

ExpDelay ExpDelayInit(double *delayPeriod,* **double** *lagPeriod,* **double** *sampPeriod,* **double** *initVal***);**

double ExpDelayCalc(ExpDelay *pE,* **int** *missing,* **double** *val***);**

double ExpDelayReset(ExpDelay *pE,* **double** *initVal***);**

These functions implement simple exponential response filters with an optional delay. **ExpDelayInit()** initializes and returns an exponential response filter. The result is **malloc**()'d. **delayPeriod** is a delay to be applied before filtering, **lagPeriod** is the time-constant of the filter, **sampPeriod** is interval between successive samples to be filtered and **initVal** is the initial value for the filter.

ExpDelayCalc() calculates a filtered value using a previously-initialized exponential response filter. **pE** is the filter returned by **ExpDelayInit()**, **missing** is the number of samples missing since the previously-filtered value, and **val** is the value to filter.

ExpDelayReset() resets the state of the exponential response filter to its initial state. **pE** is the filter returned by **ExpDelayInit()**, **initVal** is the initial value for the filter.

double FOfProb(double *p,* **int** *df1,* **int** *df2* **);**

This function returns the F ratio for a given probability **p**. **df1** is the degrees of freedom in population 1, **df2** is the degrees of freedom in population 2. It is the inverse function of **ProbOfF()**.

int FindCarterArea(double *degLat,* **double** *degLon* **);**

This function returns the Carter Area for a given latitude and longitude. The retured value is used to correct (**CarterCorrectedDepth()**) or uncorrect (**CarterUnCorrectedDepth()**) PDR depths.

double ∗**GaussianWeights(int** *n* **);**

This function computes a series of **n** Gaussian weights with a mean of 1.0 at W[**n**/2]. The result is **malloc()**'d.

double Gravity(double *deglat,* **double** *pdb* **);**

This function calculates the local acceleration of earth gravity (m/sec∗∗2) for a given latitude and pressure. **deglat** is degrees latitude (decimal degrees), **pdb** is pressure (decibars).

double GreatCircleDist(double *slt,* **double** *slg,* **double** *xlt,* **double** *xlg,* **double** ∗*course* **);**

This function computes great-circle distance in kilometers (and optionally an initial course) from a starting and ending position. It is the inverse function of **GreatCirclePos()**. **slt** is the starting latitude in decimal degrees, **slg** is the starting longitude in decimal degrees (west negative), **xlt** is the ending latitude in decimal degrees, **xlg** is the ending longitude in decimal degrees (west negative). if ∗**course** is not NULL, the initial course in degrees is returned.

void GreatCirclePos(double *dist,* **double** *course,* **double** *slt,* **double** *slg,* **double** ∗*xlt,* **double** ∗*xlg* **);**

This function computes an ending position from an initial position, a great-circle distance (kilometers) and an initial course (degrees). It is the inverse function of **GreatCircleDist()**.

double IESRho(double *s,* **double** *t,* **double** *rp* **);**

This function calculates potential density (kg/m∗∗3) using the International Equation of State for seawater (1980). **s** is salinity (PSS78), **t** is Potential Temperature (IPTS68 degrees Celsius), and **p** is pressure (decibars).

double IESSpVolAn(double *s,* **double** *t,* **double** *p,* **double** ∗*sigma* **);**

This function calculates the specific-volume anomoly (steric anomoly) (m∗∗3/kg) and potential density anomoly (kg/m∗∗3) using the International Equation of State for seawater (1980). **s** is salinity (PSS78), **t** is Potential Temperature (IPTS68 degrees Celsius), **p** is the reference pressure (decibars). ∗**sigma** (if not NULL) is the returned potential density anomoly (kg/m∗∗3).

double IPTS68Temp(double *t90* **);**

This function converts an ITS90 temperature (ITS90 degrees Celsius) to an IPTS68 temperature (IPTS68 degrees Celsius) over the oceanographic temperature range. It is the inverse function of **ITS90Temp()**.

double ITS90Temp(double *t68* **);**

This function converts an IPTS68 temperature (IPTS68 degrees Celsius) to an ITS90 temperature (ITS90 degrees Celsius) over the oceanographic temperature range. It is the inverse function of **IPTS68Temp()**.

void Interpolate(int *nPts,* **float** **xp,* **int** *incrX,* **float** **yp,* **int** *incrY,* **int** *iPts,* **float** **xi,* **int** *incrXi,* **float** **yi,* **int** *incrYi* **);**

This function interpolates a series using linear, quadratic or double quadratic interpolation. **nPts** is the number of points in ***xp** and ***yp**, ***xp** is the dependent variable series (ordered ascending), **incrX** is the second dimension of ***xp**, ***yp** is the independent variable series, **incrY** is the second dimension of ***yp**, **iPts** is the number of points to interpolate, ***xi** is a series of dependent variable values to interpolate, **incrXi** is the second dimension of ***xi**, ***yi** is the output series of interpolated independent variable values, and **incrYi** is the second dimension of ***yi**.

int LMnlFit(int *(*pInitFunc)(),* **int** *(*pCalcFunc)(),* **int** *m,* **int** *n,* **double** **x,* **double** **fvec,* **int** *maxfev,* **int** *printInterval,* **int** **nfev* **);**

This function performs non-linear least squares curve fitting for arbitrary user-supplied non-linear equations. **LMnlFit** minimizes the sum of the squares of m nonlinear functions in n variables by a modification of the Levenberg-Marquardt algorithm. This code is a combination of the Argonne Laboratory MINPACK subroutines lmder, lmdif, and lmstr.

LMnlFit() returns an integer competion code:

<0: The supplied **(*pCalcFunc)()** returned a negative value;

0: Improper input parameter(s).

1: Both actual and predicted relative reductions in the sum of squares are at most fTol.

2 Relative error between two consecutive iterations is at most xTol.

3 Conditions for both fTol and xTol have been met.

4 The cosine of the angle between fvec and any column of the Jacobian is at most gTol in absolute value.

5 The maximum number of function evaluations has been exceeded.

6 fTol is too small. No further reduction in the sum of squares is possible.

7 xTol is too small. No further improvement in the solution is possible.

8 gTol is too small. Fvec is orthogonal to the columns of the Jacobian to machine precision.

9 Out of memory.

Return codes 1-4 indicate normal completion. **LMPrintMsg()** can be used to print a message corresponding to the return code on *stderr*.

Parameters:

(***pInitFunc**)() is a supplied function to provide additional options. If not NULL, It must match the prototype: **void InitFunc(int** **jacOpt,* **int** **scaleOpt,* **double** **fTol,* **double** **xTol,* **double** **gTol,* **double** **epsFcn,* **double** **stepBound* **);**

Where **jacOpt** specifies how to calculate the Jacobean matrix:

1: Approximate by forward differencing;

2: ***pCalcFunc**() calculates the entire Jacobian;

3: ***pCalcFunc()** calculates the Jacobian a row at-a-time.

scaleOpt specifies how to scale the intermediate results:

0: Let **LMnlFit**() do it;

1: ***pCalcFunc()** calculates a scaling vector.

fTol is the Function-convergence tolerance, **xTol** is the Relative-error tolerance, **gTol** is the Geometric-convergence tolerance, **epsFcn** is the epsilon for the forward-difference approximation if **jacOpt** is 1, **stepBound** is a factor used to determine the initial

Levenberg-Marquardt step bound. 100 is a good value.

(*__pCalcFunc__)() is a supplied function that evaluates the functions to solve, prints intermediate results and optionally supplies other information. It must match the prototype: **int CalcFunc(int** *op,* **int** *m,* **int** *n,* **double** *x,* **double** **fVec,* **double** **fjac,* **int** *ldfjac*);

Where **op** specifies the operation to perform:

0: Print intermediate values;

1: Evaluate the function;

2: Calculate and supply the full Jacobean matrix;

3: Calculate and supply the Jacobean matrix one row at-a-time.

4: Calculate and return a scaling vector;

(*__pCalcFunc__)() should return 0 if ok, -1 if error. **m** is the number of non-linear functions to solve, **n** is the number of variables to solve for, ***x** contains the current solution estimate (variables), ***fVec** contains the current solution (function evaluations), **fjac** is only used when **op** is 2-4 and contains the Jacobean matrix (2-3) or the scaling vector (4), **ldfjac** is the number of elements in a row of the Jacobean matrix if **op** is 3, otherwise it is set to **m**. If fBop is 4, this is also the length of the scaling vector.

m is the number of non-linear functions to solve, **n** is the number of variables to solve for (**n** <= **m**), ***x** is a vector of length **n** containing an initial solution estimate, ***fvec** is a vector of length **m** returning the function evaluations at ***x**, **maxfev** limits the maximum number of supplied function evaluations by (*__pCalcFunc__)(), **printInterval** specifies how frequently (number of evaluations) fB(*pCalcFunc)() is to be called with **op** = 0 to print intermediate results. If **printInterval** <= 0, printing is disabled. ***nfev** is the returned total number of supplied function evaluations.

double FPMachineConst(int *what*); **double EuclidianNorm(int** *n,* **double** **x*); **int LMPrintMsg(int** *num*);

These functions are a part of **LMnlFit()**. **FPMachineConst()** returns machine-dependent double constants, determined by the argument. Given a floating point mantissa base *b* (usually 2), the number of mantissa bits *t*, the minimum integer exponent value (base *b*) *eMin* and the maximum integer exponent value (base *b*) *eMax* then:

what==0:
Returns *b***(*eMin*-1), the smallest positive magnitude;

what==1:
Returns *b***eMax**(1-*b***-*t*), the largest magnitude;

what==2:
Returns *b***-*t*, the smallest relative spacing;

what==3:
Returns *b***(1-*t*), the largest relative spacing;

what==4:
Returns log10(*b*).

EuclidianNorm() returns the euclidian norm of an **n** element vector ***x**. **LMPrintMsg()** prints a message corresponding to the **LMnlFit()** return code on *stderr*.

double O2MllToPartPress(double *o2mlpl,* **double** *s,* **double** *t,* **double** *p*);

Returns the oxygen partial-pressure (micro-atm) for the oxygen concentration **o2mlpl** (ml/l), salinity **s** (PSS78), insitu temperature **t** (degrees Celsius) and pressure **p** (decibars). This is the inverse function of **O2PartPressToMll()**.

double O2PartPressToMll(double *o2pp,* **double** *s,* **double** *t,* **double** *p* **);**

Returns the oxygen concentration (ml/l) for the oxygen partial-pressure **o2pp** (micro-atm), salinity **s** (PSS78), insitu temperature **t** (degrees Celsius) and pressure **p** (decibars). This is the inverse function of **O2MllToPartPress()**.

double O2PerKgToPerLiter(double *o2umpkg,* **double** *rho* **);**

Returns dissolved oxygen concentration (ml/l) for the dissolved oxygen concentration **o2umpkg** (micro-moles/Kg), and density **rho** (rho(s, t, 0.0) (Kg/M**3)). This is the inverse function of **O2PerLiterToPerKg()**.

double O2PerLiterToPerKg(double *o2mlpl,* **double** *rho* **);**

Returns dissolved oxygen concentration (micro-moles/Kg) for the dissolved oxygen concentration **o2mlpl** (ml/l), and density **rho** (rho(s, t, 0.0) (Kg/M**3)). This is the inverse function of **O2PerKgToPerLiter()**.

double O2Saturation(double *t,* **double** *s* **);**

Returns the oxygen saturation value (ml/l) for a potential temperature **t** (degrees Celsius) and salinity **s** (pss78).

double O2SaturationPartPress(double *s,* **double** *t,* **double** *p* **);**

Returns the oxygen saturation value (micro-atm) for a potential temperature **t** (degrees Celsius), salinity **s** (pss78) and pressure **p** (decibars).

double PSS78Conductivity(double *press,* **double** *temp,* **double** *salt* **);**

Returns the conductivity (milli-siemens) for pressure **press** (decibars), insitu temperature **temp** (degrees Celsius) and salinity **salt** (PSS78). This is the inverse function to **PSS78Salinity()**.

double PSS78Pressure(double *temp,* **double** *cond,* **double** *salt* **);**

Returns the pressure (decibars) for insitu temperature **temp** (degrees Celsius), conductivity **cond** (milli-siemens) and salinity **salt** (PSS78).

double PSS78Salinity(double *press,* **double** *temp,* **double** *cond* **);**

Returns the salinity (PSS78) for pressure **press** (decibars), insitu temperature **temp** (degrees Celsius) and conductivity **cond** (milli-siemens). This is the inverse function to **PSS78Conductivity()**.

double PSS78Temperature(double *press,* **double** *cond,* **double** *salt* **);**

Returns the insitu temperature (degrees Celsius) for pressure **press** (decibars), conductivity **cond** (milli-siemens) and salinity **salt** (PSS78).

int PolynomialFit(int *opcode,* **int** *maxfit,* **int** *order,* **double** **fit,* **double** **weights,* **double** **r2,* **int** *nPts,* **double** **x,* **double** **y* **);**

This function computes least-squares polynomial fits of specified order for arbitrary 2D data. **opcode** determines what is done:

opcode==0:
Load data points (x,y)

opcode==1:
calculate the polynomial fit

opcode==2:
calculate the fit coefficients

opcode==3:
calculate y for given x

opcode==4:
free internally-allocated fit matrix.

PolynomialFit() is called once with **opcode**==0 to load the data points. **maxfit** specifies the maximum order of polynomial to be fit, **order** specifies an order of fit (<=**maxfit**), ∗**fit** is the array of polynomial coefficients calculated when **opcode**==2, used when **opcode**==3. It must be dimensioned at least **maxfit**+2. ∗**weights** (if not NULL) is a vector of **nPts** weights (1.0==no change) to apply to **x** values when **opcode**==0. ∗**r2** is the returned correlation coefficient when **opcode**==2. **nPts** is the number of points in ∗**x** and ∗**y**. ∗**x** contains the independent variable. ∗**y** contains the dependent variable to fit.

PolynomialFit() is next called once with **opcode**==1 to compute the polynomial fit, then called one or more times with **opcode**==2 to calculate fit coefficients for the specified order polynomial. The fit coefficients are returned in ascending order. For a given set of fit coefficients, **PolynomialFit()** can be called any number of times with **opcode**==3 to calculate x values from y values. **nPts** determines the number of points calculated. Finally **PolynomialFit()** should be called once with **opcode**==4 to free-up internally allocated resources.

double PotentialTemp(double *s,* **double** *t,* **double** *p,* **double** *rp***);**

This function returns potential temperature (IPTS68 degrees Celsius) for salinity **s** (PSS78), pressure **p** (decibars) and reference pressure **rp** (decibars). To calculate *theta*, 0.0 is used for the reference pressure.

double Poynting(double *s,* **double** *t,* **double** *p***);**

This function calculates the Poynting pressure correction for oxygen partial-pressure. **s**, **t**, and **p** are salinity (PSU), temperature (Celsius) and pressure (decibars).

double ProbOfF(double *f,* **int** *df1,* **int** *df2***);**

This function returns the probability (0:1) of an F ratio **f**. **df1** is the degrees of freedom in population 1, **df2** is the degrees of freedom in population 2. It is the inverse function of **FOfProb()**.

double ProbOfR(double *r,* **int** *n***);**

This function returns the probability (-1:1) of a Pearson correlation coefficient **r**. **n** is the sample size.

double ProbOfZ(double *z***);**

This function returns the probability (0:1) of a normal distribution Z value **z**. It is the inverse function of **ZOfProb()**.

double PtoZ(double *grav,* **double** *dynz,* **double** *prsdb***);**

This function computes depth in meters using the Saunders-Mantyla (1981) method. It is the inverse function to **ZtoP**. **grav** is the local acceleration of gravity (m/sec∗∗2) @ 0.0 decibars pressure (which can be calculated with **Gravity()**), **dynz** is the dynamic height (dynamic meters), and **prsdb** is the pressure (decibars).

double RunningMeanCalc(RunningMean *pR,* **double** *val***);**

RunningMean RunningMeanInit(int *n,* **double** *initVal***);**

double RunningMeanReset(RunningMean *pR,* **double** *initVal***);**

These functions implement running means. **RunningMeanInit()** is called initially to set up the running mean. **n** is the number of samples in the running mean, **initVal** is the initial value. **RunningMeanCalc()** is called to calculate a new running mean value for a value **val**. **RunningMeanReset()** resets a running mean to a new initial value **initVal**.

double SoundVelCandM(double *s,* **double** *t,* **double** *p***);**

double SoundVelDelGrosso(double *s,* **double** *t,* **double** *p***);**

double SoundVelWilson(double *s,* **double** *t,* **double** *p***);**

double SoundVelocity(int *method,* **double** *s,* **double** *t,* **double** *p* **);**

These function calculate the velocity of sound in seawater using various formulations. In all cases, **s** is salinity (PSS78), **t** is the insitu temperature (degrees Celsius) and **p** is pressure (decibars). **SoundVelocity()** allows you to select one of the three calculation methods by specifying **method** (1==DelGrosso, 2==Wilson, 0 or default == Chen & Millero).

void Transport(int *n,* **double** **z,* **int** *zInc,* **double** **dynHt,* **int** *dInc,* **double** **transport,* **int** *tInc* **);**

This function calculates the vertical transport function (m**3/(10*sec**2)). **n** is the length of the supplied ***z**, ***dynHt** and ***transport** series. ***z** is a depth series (meters). **zInc** is the byte increment of ***z**. ***dynHt** is a dynamic height series (dynamic meters). **dInc** is the byte increment of ***dynHt**. ***transport** is the output vertical transport series (m**3/(10*sec**2)). **tInc** is the byte increment of fB*transport.

double ZOfProb(double *p* **);**

This function returns the normal distribution Z value for a probability **p**. It is the inverse function of **ProbOfZ**().

double ZtoP(double *grav,* **double** *dynz,* **double** *depth* **);**

int isinf(double *d* **);**

This function returns 1 if the argument **d** is an IEEE754 double "Infinite" value, otherwise 0.

double quiet_nan(*void* **);**

This function returns an IEEE754 double "NaN" (not-a-number) value which can be assigned to a double variable.

SEE ALSO

AUTHOR

Frank Delahoyde, STS/ODF

BUGS