INSTITUT FRANCAIS DE RECHERCHE SCIENTIFIQUE
POUR LE DEVELOPPEMENT EN COOPERATION

CENTRE DE BREST

LE TRAITEMENT DES DONNEES D'UN ADCP EMBARQUE A L'AIDE DU LOGICIEL CODAS3

Support de Cours
Brest 14-17 décembre 1993.

Gérard ELDIN

Documentation et mode d'emploi originaux de Eric Firing, Frank Bahr, et leurs collaborateurs de l'Université d'Hawaii, rassemblés, mis en forme et partiellement traduits par Gérard Eldin.

Document Scientifique du Centre ORSTOM de Brest
N° 71, Novembre 1993

Table des matières

Introduction

CODAS3 (Common Oceanographic Data Access System, version 3) est un logiciel de traitement et de base de données, développé au Département d'Océanographie de l'Université d'Hawaii. Il peut s'appliquer à tout type de données d'océanographie physique, XBTs, CTDs, etc... et en particulier aux profils de courants ADCP. Il s'agit à l'origine d'un ensemble de routines en C, portables sur différents systèmes d'exploitation (DOS, UNIX, VAX), avec des interfaces FORTRAN, permettant d'effectuer toutes les opérations habituelles de gestion de données: définition des types de données, création d'une base, entrée/sortie des données, tris (voir doc de CODAS3 page 59). A partir de ces routines, des programmes spécifiques au traitement ADCP ont été conçus, les sorties graphiques et une partie des calculs étant assurés par un logiciel du commerce, MATLAB.

La version actuelle de CODAS3 (06/93) est prévue pour traiter des fichiers ADCP au format PINGDATA enregistrés par le logiciel d'acquisition DAS v. 2.48 de RD Instruments. **Le traitement des données obtenues par le nouveau programme d'acquisition TRANSECT de RDI ne sera pas envisagé ici.**

A l'exclusion de MATLAB, qui doit être fourni par l'utilisateur, l'ensemble du logiciel, incluant ses sources, peut être obtenu librement auprès de Eric Firing, efiring@hokulea.soest.hawaii.edu .

Pour les versions UNIX et DOS, le logiciel se compose d'une arborescence, de racine `CODAS3`, dont les répertoires correspondent chacun à un rôle spécifique: le premier niveau est formé de répertoires qui contiennent les sources de la base et de divers utilitaires, les fichiers "include", les librairies compilées correspondantes, le répertoire `bin` des exécutables, et le répertoire `adcp` qui contient le logiciel de traitement proprement dit. Ce répertoire `adcp` contient à son tour des répertoires de sources, correspondant aux différentes étapes du traitement, et les répertoires correspondant chacun à une campagne (par ex. `surpac14, cither1`...). Ceux-ci sont obtenus par duplication d'un répertoire modèle par un fichier batch. (voir traitement pas-à-pas, page 5).

La suite de ce document présente tout d'abord le plan schématique du traitement d'une campagne (page 4), suivi d'une description pas-à-pas des opérations à effectuer. Les diverses docs originales en anglais réparties dans le logiciel sont ensuite regroupées, à partir de la page 17. Pour ceux qui seraient intéressés par le fonctionnement interne et l'accés direct à une base CODAS3, voir page 59. Une fois familiarisé avec les principes du traitement, il n'est pas inutile d'aller consulter les sources des programmes, qui contiennent souvent des commentaires instructifs (dans `codas3/adcp/*`).

Plan du traitement d'une campagne

☞ Constitution d'une base de données
- ✓ Création d'un répertoire pour la campagne
- ✓ Vérification des fichiers PINGDATA
- ↳ *Correction de dérive d'horloge PC*
- ✓ Chargement dans la base

☞ Validations des profils
- ✓ Calcul de seuils statistiques:
- ↳ *Pics d'intensité, Variance de W, dérivées secondes de U,V,W*
- ✓ Visualisation des profils détectés:
- ↳ *Choix: correction, suppression, pas d'action*

☞ Etalonnage
- ✓ Méthode du "water-tracking"
- ↳ *Rotation des données*

☞ Intégration de la navigation
- ✓ Couche de référence
- ✓ Filtrage
- ↳ *Rejeu de la navigation*
- ↳ *Calcul des courants absolus*

☞ Statistiques sur la base
- ✓ Différences "en route / en station"
- ↳ *Portée moyenne*
- ↳ *Evolution de l'appareil*

☞ Exploitation des données
- ✓ Sélections en fonction du temps
- ✓ Constitutions de grilles espace-temps
- ↳ *Tracés de contours, de vecteurs*
- ↳ *Analyse harmonique*

Traitement pas-à-pas d'une campagne de mesures ADCP

Avant-propos

Ce document n'est destiné à servir que de guide pour les étapes du traitement: une certaine compréhension du fonctionnement de l'ADCP, du système d'exploitation de l'ordinateur, des principes du traitement CODAS3 et du logiciel MATLAB est indispensable avant de se lancer dans le traitement de ces données.

Cet exemple s'appliquera à tous types de machines MS-DOS ou UNIX avec de légères variantes (majuscules ou minuscules, / ou \, et autres). Les noms des programmes à exécuter sont les mêmes pour les deux systèmes, à part bien sur, quelques exceptions, concernant surtout les fichiers de commandes (*.bat pour MS-DOS, shell-scripts pour UNIX) et MATLAB. La plupart des entrées de paramètres pour les programmes se font en éditant des fichiers de contrôle ASCII, auxquels on donne l'extension `.cnt` par convention. Ces fichiers comprennent des commentaires (entre `/*` et `*/`, standard du langage C) et des exemples fort utiles, à consulter fréquemment. Le principe est de naviguer dans une succession de sous-répertoires, correspondants chacun à une des étapes du traitement définies page 4. L'utilisation d'un environnement convivial (par ex. Norton Commander sur PC, X-Windows sur SUN...) est donc fortement recommandée.

On suppose au départ que les logiciels CODAS3 v. 06/93 et MATLAB v3.5 sont installés et fonctionnels, et que les données ont été obtenues par le DAS de RDI, v. 2.48. Le "prompt" de l'ordinateur est symbolisé par: `carl>`, ce que vous devez taper en **`Courier-Bold`**, ses éventuelles réponses en `Courier`. Votre éditeur ASCII favori est symbolisé par: **`Edit`** . Les étapes de traitement optionnelles sont signalées par ☺ .

Traitement pas à pas

1. Création de la structure de sous-répertoires dans laquelle s'effectuera tout le traitement. Ceci se fait par copie d'une série de modèles de fichiers de contrôle, de m-files MATLAB, etc..., à l'aide d'un script. On choisit auparavant le nom de la racine de cette structure, ici `tipe` :

```
carl>cd "le path"/codas3/adcp
carl>adcp tipe
```

2. Copie des fichiers d'acquisition DAS. Si ils sont sur disquettes, il pourra y avoir des fichiers différents de même nom (`PINGDATA.???`). Un programme les copie en les renommant à partir du nom du navire et de l'heure et la date du dernier

enregistrement de chaque fichier. Si ils ont des noms tous différents, une copie simple dans le sous-répertoire `ping` suffit. `XX` est l'abréviation du nom du bateau (2 char.)

☺ `carl>cd ping`
☺ `carl>pingcopy a: XX`

3. Examen des fichiers ("scan") pour détecter les éventuels problèmes d'enregistrement, et produire une liste de tous les "headers" (changement des paramètres d'acquisition) et des entêtes de chaque profil. Si la navigation GPS a été enregistrée avec le programme `UE3.EXE`, on a aussi pour chaque profil la différence temps_PC - temps_GPS. Si cette différence dépasse ≈10 s, il faudra corriger le temps_PC à l'étape suivante (doc des programmes, page 17):

```
carl>cd ../scan
carl>Edit scanping.cnt
```

Dans l'éditeur, modifier si nécessaire les valeurs des paramètres: `OUTPUT_FILE`, `SHORT_FORM`, `UB_OUTPUT_FILE`, `USER_BUFFER_TYPE`, `UB_DEFINITION`. Les valeurs proposées par défaut pour les 4 derniers conviennent en général, et sont conditionnés par la façon dont les données de navigation ont été enregistrées. Donner aussi la liste des fichiers à "scanner" dans le sous répertoire "ping"

```
carl>scanping scanping.cnt
```

En sortie vous obtiendrez le fichier `tipe.scn`. Si la différence temps_PC-temps_GPS y est listée ou si elle a été notée pendant la campagne, il faut calculer la dérive (linéaire) d'horloge à l'instant t, connaissant la différence PC-GPS à t_0, telle que:

$$\text{temps\_GPS}(t) = \textit{derive} * \text{temps\_PC}(t) + (\text{temps\_GPS}(t_0) - \text{temps\_PC}(t_0))$$

On note que *derive* sera <1 si le PC avance, et >1 s'il retarde. Si `tipe.scn` contient les différences PC-GPS, une routine MATLAB calcule la régression, trace le résultat, etc... Cependant il est en général suffisant de disposer d'une calculette, la dérive étant la plupart du temps régulière. Si on y tient:

☺ `carl>Edit tipe.scn`

On ne garde que les colonnes `fix_time` et `PC-fix_time` (MATLAB n'aime pas les commentaires...).

☺ `carl>Edit clkrate.m`

Edition de la routine pour spécifier les noms de fichier entrée/sortie et quelques autres paramètres.

☺
```
carl>matlab
     >>clkrate
          "Graphiques régression"
          clock_rate =
               0.99989289352752
          PC clock reset
```

```
                    date is:91/01/28  21:21:36
      >>quit
carl>
```

Noter les valeurs de `clock_rate`, les moments où il change et les moments de remise à l'heure du PC, ou la correspondance PC/GPS à un instant donné pour chaque valeur de `clock_rate`

4. Création d'une base de données pour cette campagne, et chargement des données dans cette base. Une base de données CODAS3 se compose d'une série de fichiers de données ("blocks") et d'un fichier répértoire ("block directory"). Les noms, types, tailles, taux de répétition des variables de la base sont définis à sa création dans un fichier ASCII `xxxxxx.def` (voir "Producer definition file", page 71). Pour une base de données ADCP "normale" (sans données supplémentaires, CTD ou autres), on peut se contenter de prendre le fichier d'origine `codas3.def`, et de changer peu de choses:

```
carl>cd ../adcpdb
carl>Edit codas3.def
```

Changer `PRODUCER_ID`, en fonction du pays d'origine, du nom du bateau, du type de l'ADCP. Si vous pensez introduire d'autres données que celles de l'ADCP dans la même base (CTD par ex.) alors il faut aussi modifier `VARIABLES`, et les définitions de `STRUCT` (ures). Voir les commentaires dans `codas3.def` .

```
carl>cd ../load
carl>Edit loadping.cnt
```

Modifier `DATABASE_NAME`, les noms de fichiers `DEFINITION_FILE`, `OUTPUT_FILE`, les limites de blocks `NEW_BLOCK_AT_FILE`, `NEW_BLOCK_AT_HEADER`, donner la liste des fichiers de données à charger, `PINGDATA_FILES`, avec éventuellement `time_correction` (la *derive* calculée plus haut), les profils à exclure: `skip_profile_range`, etc. Cette étape est assez délicate, toute la suite des opérations en dépendra. Comme la base est indexée sur le temps, il est indispensable que les données soient entrées dans l'ordre chronologique, sans recouvrement.

```
carl>loadping loadping.cnt
```

Ceci crée la base de données, les blocks sont les fichiers `../adcpdb/tipe???.blk`, et le répertoire `../adcpdb/tipedir.blk`. Un fichier "log" est aussi créé, avec la liste des profils chargés, pour vérification, `tipe.lod`

5. Pour se convaincre qu'on a bien fait ce qu'on voulait, corrections d'horloges, non-chargement de certains profils, on peut examiner la base, avec quelques utilitaires, modifier et même supprimer des blocks, en cas d'erreur. Ne jamais supprimer des blocs par le système d'exploitation de l'ordinateur, sinon il faut reconstruire la base:

☺ `carl>cd ../adcpdb`

☺ `carl>`**`showdb tipe`**
`showdb` permet de parcourir la base et d'accéder à tous les paramètres stockés, profils de vitesse et d'intensité bien sur, mais aussi des paramètres tels que température du capteur, vitesse du son utilisée pour le calcul, qu'il faut pouvoir vérifier dans certains cas.

☺ `carl>`**`lstblock tipe tipeblk.lst`**
Ceci fournit le fichier `tipeblk.lst` qui contient la liste des blocks, le nombre de profils et les limites en temps de chacun.

☺ `carl>`**`depthcng tipe`**
Change les niveaux de profondeur d'une série de blocks, en cas d'erreur dans les paramètres du DAS. **Manier avec précaution.**

☺ `carl>`**`chtime chtime.cnt`**
Change les temps d'une série de blocks. **Manier avec précaution.**

☺ `carl>`**`delblk tipe`**
Programme interactif qui permet de supprimer des données par blocks, par intervalle de temps, etc... **Manier avec précaution.**

☺ `carl>`**`mkblkdir mkblkdir.cnt`**
Permet de reconstruire une base à partir de blocks "orphelins" et/ou de la transférer d'un système d'exploitation à un autre (DOS <-->UNIX<-->VAX).

6. Correction des profils de vitesses (relatives au navire), pour élimination des réflections parasites sur le fond ou sur des objets immergés (CTD, mouillages...). Plusieurs étapes sont nécessaires (docs des programmes page 23).

6.1. On calcule d'abord les moyennes et écarts-types de W, EV et $\delta^2W/\delta z^2$, $\delta^2U/\delta z^2$, $\delta^2V/\delta z^2$:

```
carl>cd ../edit
carl>Edit profstd0.cnt
carl>Edit profstd2.cnt
```
Changer `dbname`, `output`, `step_size` et `time_range` dans les 2 fichiers de commande.

↳ 91/01/03 02:00:00
↳ 91/03/20 04:00:00

```
carl>profstat profstd0.cnt
carl>profstat profstd2.cnt
```
Les statistiques calculées sont stockées dans 2 fichiers ASCII `*.prs` pour inspection visuelle, ainsi que dans deux fichiers MATLAB, `*.mat` pour la suite: une m-file va calculer les seuils statistiques (par défaut 3σ, mais ca peut se changer) qui permettront la détection de profils "hors-normes".

```
carl>Edit editflag.m [threshld.m]
```
Mettre dans cette m-file (dont le nom varie suivant les systèmes) les noms des fichiers statistiques, et lancer MATLAB, qui retourne les valeurs des seuils.

```
carl>matlab
     >>editflag
          w_var_threshold =
          3.6529e+03
          d2w_threshold =
          33.9822
          d2uv_threshold =
          0.6695
     >>quit
carl>
```

On note ces résultats, qu'on va entrer dans le fichier de controle du programme qui fournira la liste des profils "flaggés":

carl>**Edit flag.cnt**
Changer DB_NAME, W_VAR_THRESHOLD, D2W_THRESHOLD, D2U&D2V_THRESHOLD, TIME_RANGE, ainsi que les seuils qui ne sont pas obtenus par statistique mais par l'expérience, AMP_TRESHOLD PGOOD_TRESHOLD...

carl>**flag flag.cnt**
La liste des profils détectés est créée, dans le fichier `tipe.flg` . L'utilisateur doit alors décider de la garder, ou de relancer flag apres avoir "resserré" ou "desserré" les seuils. Par exemple si une campagne se déroule en majorité dans des eaux à faible cisaillement, avec une incursion à l'Equateur, la présence du EUC peut déclencher la détection des "pics" de U, par rapport aux stats de l'ensemble de la campagne. Dans ce cas, il vaut mieux séparer ce traitement en deux parties. On imprime ensuite le ou les fichiers `*.flg` .

6.2. Il s'agit maintenant d'examiner visuellement les profils détectés, pour décider de l'action à effectuer, correction, suppression de "bins" ou de profils entiers, ou... rien du tout. En effet, dans la pratique, un gros pourcentage des détections est dû à des phénomènes naturels ("deep scattering layer", forts cisaillements momentanés). Cependant, il faut pouvoir traiter le petit nombre de profils qui posent réellement problème, en particulier lorsque, à l'acquisition, des reflections sur le fond se sont produites sans que le "bottom-track" soit ON. Dans ce cas rien n'indique que les données sous le fond sont mauvaises, alors que, avec "bottom-track", elles sont "flaggées" à l'origine. La procédure décrite ici correspond à la procédure No 2 de la doc originale (page 28) qui à l'usage se révèle la plus utile. La visualisation se fait dans MATLAB:

carl>**Edit setup.m**
Changer le nom de la base (DBNAME), le nombre de profils à afficher ensemble (DEFAULT_SEQ_FLAG), ... et lancer MATLAB. N.B. On peut créer une m-file pour automatiser la suite des opérations répétitives à effectuer dans MATLAB:

```
carl>matlab
     >> setup
```

```
        >> get(0,10)
        >> init
☺       >> offset(10)  (il y a des valeurs par défaut)
        >> draw(AMP)
        >> draw(PGOOD)
☺       >> offset(.1)
        >> draw(U)
        >> draw(W)
        >> bad([3:7],5)
        >> list
        >> get(0,30)
        >> init
        .
        .
        .
        >> quit
carl>
```

Ici on a, par exemple, visualisé pour le profil 10 du block 0 et ses proches voisins, successivement AMP, PGOOD et les composantes U et W. On a alors décidé de supprimer les "bins" 3 à 7 de ce profil, et des 5 profils suivants. **A ce stade, les profils originaux ne sont pas encore touchés; on a simplement construit un fichier,** badbin.asc, **contenant les bins et profils à supprimer.** Pour supprimer vraiment les bins et profils voulus, il faut lancer le programme badbin:

```
carl>badbin ../adcpdb/tipe badbin.asc
```

6.3. Il faut traiter séparément le cas des dépassements de seuil pour EV. Alors que pour les autres variables le test doit être fait sur l'ensemble du profil, pour EV il faut se restreindre aux bins tels que PGOOD >80%, sous peine d'overdose. On passe donc dans le sous répertoire evpass ou on recommence la même opération. Notons que les profils détectés par le seuil sur EV seront souvent les mêmes que ceux détectés auparavant, mais PAS TOUJOURS...

```
carl>cd evpass
carl>Edit flag.cnt
```

Editer la valeur de EV_TRESHOLD dans ce fichier de controle

```
carl>flag flag.cnt
```

Imprimer la liste des tirs détectés, comparer avec la précédente et visualiser les profils non encore vus:

```
carl>cd ..
carl>matlab
     >> setup
     >> get(1,234)
     >> init
☺    >> offset(.1)
     .
     .
```

```
>> quit
carl>
carl>badbin ../../adcpdb/tipe badbin.asc
```

6.5. Pour être sur d'éliminer totalement l'influence des reflections sur le fond, on "coupe" les derniers 15% des bins sur les profils qui ont été coupés au fond:

```
carl>botmpas3 ../adcpdb/tipe
carl>last_85 ../adcpdb/tipe
```

☺ **N.B.** Il existe des programmes (`logscan`, `blkscan`, `updscan`) qui, acceptant en entrée les fichiers de profils "flaggés" de type `tipe.flg`, agissent directement sur les profils sans passer par la visualisation MATLAB. Pour ceux qui s'y intéressent, leur mode d'emploi est détaillé plus loin (voir page 26, "procédure No 1").

7. Il s'agit maintenant d'étalonner l'ADCP en calculant les erreurs d'angle Φ et d'amplitude A, pour appliquer ensuite aux données une correction par rotation et homothétie du vecteur vitesse mesuré. On ne détaillera ici que la méthode par "water-tracking", celle du "bottom-tracking" est similaire (docs des programmes page 37).

7.1. On obtient tout d'abord un fichier des données de navigation, qu'on supposera GPS et enregistrées avec le programme résident `UE3.EXE` ou similaire.

```
carl>cd ../nav
carl>Edit ubprint.cnt
```

Changer `dbname`, `output`, `year_base`, variables choisies, `time_ranges`.
Comme variable `avg_GPS_summary` convient le plus souvent.

```
carl>ubprint ubprint.cnt
```

Ceci crée le fichier `tipe.ags`, qui contient normalement des informations sur la qualité des fixes, et qui servira aussi plus tard pour l'étape 8. d'intégration de la navigation.

```
carl>Edit tipe.ags
```

On peut ainsi éliminer, par exemple, les fixes pour lesquels HDOP >6.

☺ Si la navigation a été enregistrée indépendemment, il faut construire un fichier comportant environ un fix GPS par ensemble, au format:
jour_decimal longitude_decimale latitude_decimale
Voir `codas3/dbsource/time_.c` pour le calcul du jour_decimal.

7.2. On obtient ensuite un fichier de la vitesse d'une couche de référence (bins 5 à 20, par ex.) par rapport au navire, et on calcule les différences de vitesses absolues à chaque changement important du vecteur vitesse du navire (arrêts, départs, virages)

```
carl>cd ../cal/watertrk
carl>Edit adcpsect.cnt
```

Changer dbname, output, year_base, time range .

carl>**adcpsect adcpsect.cnt**
On obtient tipe.nav, vitesse du navire par rapport à la couche de référence.

carl>**Edit timslip.cnt**
Changer fix_file, reference_file, output_file, year_base et tous les paramètres se rapportant au calcul du courant avant et après les changement de vitesse du bateau, comme le nombre de profils moyennés, ici 5 (voir page 37).

carl>**timslip timslip.cnt**
On obtient tipe.cal, qui contient les résultats du calcul à chaque accélération et décélération, avec A et Φ. On le débarasse de ses commentaires, et on rentre dans MATLAB pour tracer les séries temporelles et histogrammes de A, Φ et le décalage de temps moyen PC-GPS estimé par corrélation ("time-shift").

```
carl>grep -v '%' tipe.cal > tipe.tmp
carl>matlab
     >> load tipe.tmp
     >> adcpcal(tipe, 'TIPE - 5 points')
     " sorties graphiques"
     >> quit
carl>
```

Les valeurs de A et Φ et les statistiques correspondantes sont inscrites automatiquement dans adcpcal.out .

7.3. Il est sage de ne pas appliquer aussitôt la rotation aux données, mais de faire d'autres essais avec des paramètres différents pour timslip, surtout changer le nombre de profils moyennés avant et après les accélérations/décélérations. L'examen des résultats de 8., intégration de la nav, peut aussi être instructif. On y passe donc une première fois, avec les données non étalonnées.

8. Intégration de la navigation: calcul de la vitesse absolue de la couche de référence, lissage, calcul de la vitesse fond et route du navire par estime (l'ADCP est le loch) par rapport à cette couche de référence lissée, entrée dans la base de ces vitesses et positions lissées. On utilise les fichiers tipe.ags (GPS) et tipe.nav (vitesse relative navire) obtenus en 7 (docs des programmes page 51).

carl>**cd ../../nav**
carl>**Edit refabs.cnt**
Changer reference_file, fix_file, output, year_base.

carl>**refabs refabs.cnt**
On obtient tipe.ref, vitesses absolues de la couche de référence.

carl>**Edit smoothr.cnt**
Changer reference_file, refabs_output, output, filter_hwidth.

```
carl>smoothr smoothr.cnt
```
En sortie, deux fichiers `tipe.sm` et `tipe.bin` (version binaire du précédent) qui contiennent essentiellement la vitesse de la couche de référence et les positions après filtrage. On passe dans MATLAB pour tracer ces paramètres en fonction du temps, un ou 2 jours à la fois.

```
carl>Edit callrefp.m
```
Modification des paramètres des tracés, noms des fichiers. Changer `year_base`, `MetName`, `RefInput`, `RefMatName`, `SmBin`, `SmMatName`.

```
carl>matlab
     >> callrefp(31)
     " Graphiques..."
     >> callrefp(33)
          .
          .
     >> quit
carl>gpp -dps tipe031.met
carl>gpp -dps tipe033.met
carl>print tipe031.ps
carl>print tipe033.ps
```
Examen des graphiques. Des "pics" dans la vitesse de la couche de référence indiquent généralement de mauvais fixes GPS. Dans ce cas, il faut éditer `tipe.ags` en commentant ou supprimant les mauvais fixes, et recommencer en 8. Des "sauts" aux accélérations et décélérations du navire confirment ou modifient les conclusions de l'étalonnage (7.), ou peuvent indiquer de façon plus précise à partir de quel moment exactement la rotation doit s'appliquer (problèmes de gyro par exemple). Si aucune correction du GPS n'est nécessaire, et si l'étalonnage est fait (ou inutile) aller en 10.

9. Pour appliquer la rotation/homothétie aux données (doc du programme page 45):

```
carl>cd ../cal/rotate
carl>Edit rotate.cnt
```
Changer `dbname`, `time_range`, `amplitude`, `angle_0`, si nécessaire les autres paramètres: variation de Φ avec le temps, ou avec le cap. **Vérifier plutôt deux fois qu'une, car là on va agir directement sur les données de la base:**

plutôt rotNav.cnt (simulation, n'agit pas sur la base)

⚠ si on effectue plusieurs fois rotNav, effacer avant son fichier de sortie. puis retour à nfabs.

```
carl>rotate rotate.cnt
```
En sortie, un fichier log `tipe.rot` confirme l'action appliquée. Ensuite, recalcul avec les données étalonnées de la vitesse relative de la couche de référence, et rejeu de la navigation (comme en 8., changer éventuellement les noms de fichiers pour garder une trace des traitements avant/après étalonnage):

!! rotate modifie la base !

Si right ade format → \tipe\adcpdb ou lancer convadc2

```
carl>cd ../../nav
carl>adcpsect adcpsect.cnt
carl>refabs refabs.cnt
carl>smoothr.cnt
carl>matlab
     >> callrefp(31)
        .
     >>quit
carl>gpp ....
carl>print ...
```

10. Si satisfait des tracés, il ne reste qu'à rentrer les données de navigation définitives dans la base:

```
carl>Edit putnav.cnt
```
Changer dbname, year_base, position_file

```
carl>putnav putnav.cnt
```
Le traitement proprement dit est terminé, on peut passer à l'extraction et à l'exploitation des données:

11. Extraction et exploitation. Après le traitement, on veut généralement pouvoir extraire des profils le long de sections, pour contouring ou tracés de vecteurs. Comme la base est indexée par le temps, il faut d'abord construire des grilles d'intervalles de temps correspondant aux grilles d'espace sur lesquelles on veut travailler:

```
carl>cd ../grid
carl>Edit llgrid.cnt
```
Changer dbname, output, latitude ou longitude origin et increment, time_ranges

```
carl>llgrid llgrid.cnt
```
En sortie on obtient tipe.grd, liste de "time_ranges" correspondant chacun à un pas de la grille définie ci dessus.

11.1. Pour obtenir un fichier tipe.con de "quadruplettes" (time ou lon ou lat, prof, u,v) pour chaque point de la grille:

```
carl>cd ../contour
carl>type ../grid/tipe.grd >> adcpsect.cnt
carl>Edit adcpsect.cnt
```
Changer dbname, output et les paramètres d'interpolation

```
carl>adcpsect adcpsect.cnt
```
En sortie, tipe.con, à tracer avec un logiciel de contouring. Des fichiers MATLAB sont aussi produits, tipe.muv et tipe.mxy, ainsi qu'un fichier de statistiques tipe.sta.

11.2. Pour obtenir un fichier `tipe.vec` (lon ou lat, couples u,v moyennés par tranches de prof.), même principe:

```
carl>cd ../vector
carl>type ../grid/tipe.grd >> adcpsect.cnt
carl>Edit adcpsect.cnt
```

Changer dbname, `output` , les intervalles de profondeur et les paramètres d'interpolation.

```
carl>adcpsect adcpsect.cnt
```

En sortie, `tipe.vec`, à tracer avec le programme `vector`.

```
carl>Edit vector.cnt
carl>vector vector.cnt
```

Changer les parametres du dessin, en suivant les explications dans `vector.cnt`. Sortie d'un fichier PostScript, avec fond de carte inclus.

☺ 12. Pour controler les performances de l'ADCP et la qualité des données, il est utile d'obtenir des statistiques sur la durée d'une campagne concernant EV, AMP, PGOOD, etc... En particulier, il est intéressant de comparer performances obtenues en stations et en route:

```
carl>cd ../quality
carl>Edit arrdepos.cnt
```

Changer `reference_file`, output: `tipe_os.arr`, range: `whole_station`, etc.

```
carl>arrdep arrdepos.cnt
```

Ceci crée le fichier `tipe_os.arr`, qui contient tous les "time_ranges" correspondant aux stations. Il vaut mieux vérifier son contenu, car la détermination des stations n'est pas toujours évidente. Ensuite, concaténation avec `profstat.cnt`:

```
carl>type tipe_os.arr >> profstos.cnt
carl>Edit profstos.cnt
```

Changer dbname, `output`, liste des `variables`, `labels`, `time_ranges` ...

```
carl>profstat profstos.cnt
```

On obtient les statistiques pour tous les profils en stations, un fichier ASCII `tipe_os.prs`, et un fichier MATLAB, `tipe_os.mat`

Même manip pour les profils en route:

```
carl>Edit arrdepuw.cnt
carl>arrdep arrdepuw.cnt
```

A vérifier aussi.

```
carl>type tipe_uw.arr >> profstuw.cnt
carl>Edit profstuw.cnt
carl>profstat profstuw.cnt
```

Crée `tipe_uw.prs` et `tipe_uw.mat`

On utilise MATLAB pour tracer les profils statistiques:

```
carl>Edit stn_udw.m
```
Changer `metname`, `stn_file`, `udw_file`, etc.

```
carl>matlab
     >> stn_udw
     "Graphiques"
      >> quit
carl>gpp -dps tipe.met
carl>print tipe.ps
```

☺ **13.** Il y a un sous-répertoire `stick`, qui permet de faire de l'analyse harmonique et des "sticks plots", suivant le même principe que pour les contours ou les vecteurs. Pas utilisé par l'auteur jusqu'à présent...

☺ **14.** Il existe aussi plusieurs programmes, plus ou moins documentés, d'intérêt général ou destinés à rechercher les causes de problèmes spécifiques à l'ADCP, qui peuvent se révéler parfois fort utiles. En voici une liste (presque) exhaustive:

A lancer sans arguments, un message explique le mode d'emploi:

atg, bvfreq, depth, grav, press, svel, theta
Calculs de grandeurs physiques diverses
to_day, to_date
Calcul du jour décimal en fonction de la date, et inversement.

A lancer avec des fichiers de controle, dont un modèle se trouve dans `"path"/codas3/cntfiles` :

convadcp, convadc2
Transforment des bases CODAS de formats obsolètes, anciennes versions.
rotnav
Simule l'effet d'une rotation en agissant uniquement sur les fichiers issus de `adcpsect`, sans toucher à la base.
lst_btrk, lst_hdg, lst_prof, lst_temp
Listent la vitesse bottom-track, le cap, la position, la temperature du transducteur, respectivement, en fonction du temps. `lst_prof` permet d'avoir la route du bateau, et `lst_temp` est utile lorsque l'étalonnage indique un problème d'amplitude.
fix_hdg, fix_temp
Permettent de modifier des valeurs de cap ou de température du transducteur. **Manier avec précaution.**
timegrid
Permet de subdiviser une grille d'intervalles de temps en intervalles plus petits.

****

Création d'une base de données ADCP

(d'après E. Firing)

==========

1. scanping

==========

USAGE : scanping [control file name]
INPUT : 1) ping data files
2) user buffer definition file, if needed
OUTPUT: 1) output log file
2) Transit fix file, if present and requested

This program scans binary ping data files and produces an output file listing the header and profile number, recorded profile time and the time interval between two consecutive profiles. It also flags bad header and bad profiles. It is used before loading the ping data files into a CODAS database using `loadping` in order to verify that the ping data files can be read and interpreted correctly, the recorded times are correct, etc. The information in the output file can be used to prepare the control file for `loadping`, where one can indicate any time corrections or which headers and/or profiles must be skipped due to bad data.

If the pingdata files were recorded using the program `MAG1157.EXE` (or later versions, like `UE3.EXE`), the navigation information will have been stored in the `USER_BUFFER` following a specific format. In this case, the user can request fix information by specifying the type of `USER_BUFFER`. Five types of `USER_BUFFER` format may be created by `UE3.EXE`: (a) 128-byte type, (b) 102-byte type, (c) 132-byte type, (d) 72-byte type, and (e) 144-byte type. For the last 3 types, the difference between the PC recorded time and the fix time is recorded in the output file. If the user buffer contains the information in ASCII format, the user can specify the type of `USER_BUFFER` as ascii and thereby obtain an ASCII dump of the user buffer in an output file.

If some other program was used to write user buffer in the format other than the abovementioned formats, the user can still obtain a printout of its contents by specifying the `USER_BUFFER` type as "other" and supplying the appropriate definition in the user buffer definition file. It is recommended that the user also use FORMAT statements in that file to override the default printing format which takes up at least one line per element.

The control file for scanping must have the following structure:

OUTPUT_FILE: <output file name>
SHORT_FORM: <yes | no>
UB_OUTPUT_FILE: <user buffer output file name | none>
USER_BUFFER_TYPE: <none | 720 | 1440 | 1320 | 1280 | 1020 | ascii | other>
UB_DEFINITION: <user buffer definition file name>
PINGDATA_FILES:

<pingdata file 1>
<pingdata file 2>
.
.
.

All uppercase terms in the above are keywords and should never be changed. All items in angular brackets are parameters and must be supplied by the user. Following is a brief explanation of each keyword and parameter.

(1) OUTPUT_FILE:
specifies the output file name.

(2) SHORT_FORM:
yes -- will not print out the data types name recorded in each header;
no -- otherwise.
If a lot of headers are in one pingdata file, but few profiles are under each header, then selecting "yes" will reduce the printing time.

(3) UB_OUTPUT_FILE:
specifies the user buffer output file name. If "none", no user buffer output file will be created. If 102-, 128-, 132-, 72- or 144-byte type of user buffer is used, only the Transit fix information will be printed in the following format:
column
1 — fix_time (decimal day)
2 — longitude (deg)
3 — latitude (deg)
4 — elevation (m)
5 — iterations
6 — dr_dist (m)
7 — used
8 — (pc_time - fix_time) (sec)
For above mentioned user-buffer types, column 8 will also appear in OUTPUT_FILE; the user can easily calculate the PC clock rate to be applied during time correction when loading pingdata files into CODAS database using loadping. There is a Matlab M-file called `clkrate.m` that can be used to read in the file and estimate the clock rate and the last time the PC clock was reset to match the satellite clock. If another type of user buffer is used, all fields of the user buffer will be printed, unless FORMAT statements are used to control printing.

(4) USER_BUFFER_TYPE:
720 -- 72-byte type user buffer is used;
1440 -- 144-byte type user buffer is used;
1320 -- 132-byte type user buffer is used;
1280 -- 128-byte type user buffer is used;
1020 -- 102-byte type user buffer is used;
ascii -- ASCII type user buffer is used;
other -- other type of user buffer.

none -- No user buffer is used or user buffer is unknown. In this case, no user buffer output file will be created.

(5) UB_DEFINITION:
specifies the user buffer definition file name. Typing "none" means there is no user buffer definition file and no user buffer output file will be created. The user buffer definition file contains information about the structure of the user buffer: variable names, types, units, etc. To make your own user buffer definition file, see files *ub_*.def* as examples.

(6) PINGDATA_FILES:
specifies the names of pingdata files to be scanned.

==========
2. clkrate.m
==========

USAGE: clkrate
INPUT: `.fix` file from scanping
OUTPUT: estimates of the `clock_rate` and last PC clock reset time

This Matlab M-file estimates the `clock_rate` correction factor for the loadping control file by fitting a polynomial to the data in the last column of `scanping`'s output file (PCtime - Fixtime). It uses the fitted polynomial to calculate the last time at which the PCtime and Fixtime were equal (that is, when the PC clock was reset to match the satellite clock). Examine the last column (PCtime - Fixtime) of the `scanping` output file for outliers. Set the `abs_max_bad` parameter below to exclude those points. Examine the column for trends. Set the `abs_max_gap` parameter below to catch the breaks in trends, if any. Finally, set the first three noncomment lines to specify the appropriate files and yearbase to use.

```
%---------------- EDIT FOLLOWING LINES ----------------
filename = 'a901scan.fix'; input file (scanping .fix output)
out_file = 'clkrate.out'; output file (diary)
yearbase = 89;
abs_max_gap = 7;          break-in-trend threshold (absolute value, seconds)
abs_max_bad = 1E+2;       outlier threshold (absolute value, seconds)
%---------------- END EDIT LINES ----------------
```

Start Matlab and invoke this M-file. It will yield estimate(s) of the PC clock rate and the time(s) at which the PC clock was reset to jive with the satellite clock for each detected trendline after deleting outliers according to specified thresholds. For the `clock_rate`, either average the different estimates or pick the best one (the one based on a longer time series).

NOTE: It may be necessary to manually edit the fix file and delete specific lines that defy underlying trends but cannot be caught by suitable thresholds.

==========

3. loadping

==========

USAGE : loadping [control file name]
INPUT : 1) producer definition file
2) ping data files
OUTPUT: 1) CODAS database
2) output log file, if requested

This program is used to create a CODAS database from data in ping binary files recorded by the ADCP using RDI DAS, version 2.48.

The control file for loadping must have the following structure:

```
DATABASE_NAME:      <database name>
DEFINITION_FILE:    <producer definition file>
OUTPUT_FILE:        <output file name | none>
MAX_BLOCK_PROFILES: <max prfs. in DB BLK>
NEW_BLOCK_AT_FILE:  <yes | no>
NEW_BLOCK_AT_HEADER: <yes | no>
NEW_BLOCK_TIME_GAP(min): <time gap in min>

PINGDATA_FILES:
      <pingdata file 1>
        [options]
       end
      <pingdata file 2>
        [options]
       end
       .
       .
       .
```

All uppercase terms in the above are keywords and should never be changed. All items in angular brackets are parameters and must be supplied by the user. Items in square brackets are optional. Following is a brief explanation of each keyword, parameter, and option.

(1) DATABASE_NAME:
database name, up to 4 characters plus path name if needed.
(2) DEFINITION_FILE:
producer definition file.
(3) OUTPUT_FILE:
output log file name.
'none' or 'NONE' indicating no output file.

(4) MAX_BLOCK_PROFILES:
max number of profiles to be stored in each database block, up to 400.
(5) NEW_BLOCK_AT_FILE:
yes -- a new database block will be created for each ping file;
no -- otherwise.
(6) NEW_BLOCK_AT_HEADER:
yes -- a new database block will be created when a new header is found
no -- otherwise.
(7) NEW_BLOCK_TIME_GAP(min):
time gap in minutes (any positive number <= 32767)
A new database block will be created if the time interval between two consecutive profiles exceeds this time gap.
(8) PINGDATA_FILES:
list pingdata file names after this line. Each pingdata file name must be followed by the word 'end'. Between the pingdata file name and the word 'end', the user may specify any number of options. There are three kinds of options:
(a) time_correction:
start_header_number: <header number>
correct_time: <yy/mm/dd-hh:mm:ss>
PC_time: <yy/mm/dd-hh:mm:ss>
clock_rate: <PC clock rate>
(b) skip_header_range: <start hdr No.> to <end hdr No.>
(c) skip_profile_range:
hdr= <hdr No.> prof= <start prf No.> to <end prf No.>

All non-bracketed items in the above options are keywords and must appear as is. All items in angular brackets are parameters and must be supplied by the user. Under one pingdata file, all the time_correction options must be given in sequential order; the header number and profile number in all skip-options must also be given in ascending order.

Description of the Options:

(a) `time_correction:`
If `time_correction` is selected, the time of corresponding profiles will be corrected while loading the database. One time correction entry stays in effect until it is changed by the next time correction entry, or until the end of the current pingdata file.
`start_header_number:`
indicates where time correction starts.
`correct_time:`
is the actual time (satellite fix time, t0) when thePC clock was set or observed. It must be in the form of yy/mm/dd-hh:mm:ss.
`PC_time:`
is the time displayed on the PC clock at time t0. It must be in the form of yymm/dd-hh:mm:ss.

`clock_rate`:
is clock rate correction
= 1.0 if clock keeps perfect time;
> 1.0 if clock is slow;
< 1.0 if clock is fast.
loadping applies the following equation to correct the time:
t_correct = t0 + clock_rate * (t_recorded - pc_time)
where t_recorded is the time recorded for a profile in ping data file.

(b) `skip_header_range`:
If `skip_header_range` is specified, header(s) in the pingdata file within the indicated header range will not be loaded into the database.

(c) `skip_profile_range`:
If `skip_profile_range` is specified, the profile(s) within the indicated profile range for the indicated header will not be loaded into the database.

A control file is shown in `codas3/adcp/demo/load/loadping.cnt`. An example of producer def. file is given in `codas3/adcp/demo/adcpdb/codas3.def`.

****

Correction des profils de vitesse

(d'après E. Firing et F. Bahr)

This file provides documentation on a group of executable programs and MATLAB M-files that are used to edit an existing CODAS database. The editing process consists of a profile-by-profile scan of the database for bad data arising from bottom interference or some other disturbances; and setting appropriate flags provided in the database structures to indicate the presence of such bad data.

==========
PROGRAM: flag
==========

USAGE: flag [control file name]
INPUT: CODAS database, control file
OUTPUT: log file containing list of profiles flagged as bad

Given an existing CODAS database and a control file, this program performs the profile-by-profile scan through the database. The control file must specify the following:

1. The name of the CODAS database to be scanned

2. The bin at which to start the *W* variance calculation for the *W* variance algorithm described below (Sometimes, the *W* velocity at the topmost bins of profiles contain unexplained irregularities even if the *U* and *V* velocity data look alright; these may cause profiles to be flagged unnecessarily. This can be avoided by setting the start bin at some value greater than one.)

3. The threshold values for the following variables (using the units in which they are stored in the CODAS database):
   a. the rise in amplitude
   b. the percent good

4. The length unit expressed as a fraction of a meter in which the succeeding velocity thresholds are expressed. For example, if the thresholds are already expressed as meters, then specify a length unit of 1. For mm, specify 1E-3 or .001.

5. The threshold values for the following variables (using the units indicated by the length unit parameter above):
   a. the variance of the *W* velocity
   b. the second difference of the *W* velocity
   c. the second difference of the *U* and *V* velocities
   d. the error velocity

6. The time ranges to scan over specified as:

y1/m1/d1 h1:m1:s1 to y2/m2/d2 h2:m2:s2

See the sample control file named `flag.cnt` in the `codas3/adcp/demo/edit` subdirectory for an example of how to specify these parameters.

The program uses four algorithms to detect bad data:

1. The amplitude criterion ("A" flag)

This consists of scanning the amplitude data for any increases that exceed the specified threshold (a). If such an increase is detected, the bin at which the amplitude reaches the local maximum is recorded as the max_amp_bin. This is taken to indicate the bin at which the ocean bottom was encountered. The profile is flagged by logging its block and profile number in the output log file, as well as the max_amp_bin.

2. The *W* variance criterion ("V" flag)

The routine calculates the variance of the *W* data for bins at which the percent good variable is greater than or equal to the percent good threshold (b). If the computed *W* variance is greater than the *W* variance threshold (c) specified in the control file, the profile is flagged by logging its block and profile number in the output log file, together with the computed variance.

3. The second difference criterion ("D" flag)

The routine calculates the second difference of *W* at only those bins for which the percent good variable is greater than or equal to the percent good threshold (b). If the second difference at any such bin exceeds the threshold (d) for the second difference of *W* in the control file, the second difference of the *U* velocity at that same bin is examined. If this likewise exceeds the threshold (e) specified in the control file for the second difference of *U* and *V*, then the profile is concluded to be contaminated. Otherwise, the second difference of the *V* velocity is examined. If it exceeds the (e) threshold, then the profile is concluded to be contaminated. Otherwise, the profile is left unflagged. The block and profile numbers of contaminated profiles are recorded in the log file, together with the second difference values.

4. The fourth algorithm checks the absolute value of the error velocities, if present, against some threshold for those bins that satisfy the percent good threshold. If any of those error velocities exceed the error velocity threshold, the profile is logged as bad, and the largest absolute error velocity is recorded in the log file together with the bin number. An auxiliary file (`.ev`) is also created as a separate record of the time and block-profile number of each flagged profile, together with the total number of bins found to exceed the threshold and a listing of those bin numbers. This file may be used directly with the badbin program to flag those bins as bad in the database. By recording the computed values together with the block and profile numbers of profiles that have been flagged and comparing

these with plots of the data, one can determine whether the threshold values provided in the control file have been appropriately set.

The output log file consists of three types of lines:

1. Header/comment lines preceded by a single '%' symbol, indicating the threshold values and time ranges read from the control file, as well as the column headings for the subsequent data.
2. Error and warning lines preceded by '%%', indicating any problems encountered during execution. Use your editor's search capability to check the output log file for any such problems.
3. The data itself, comprised of one line each for every profile flagged. Some sample lines from such a file are provided below, following the column headings:

```
% TIME_RANGE: 1989/04/20 00:48:00  to  1989/04/20 19:24:00
% T         ---Profile Time--- ---- ----Amplitude Pass----- ---W---- ---Glitch Pass--- --EV Pass--
% A Blk Prf                    Flag   Min   Max Dep Min Max Variance Glitch              Max    At
% G No. No. yy/mm/dd  hh:mm:ss ----   Bin   Bin  th Amp Amp --Pass--   Bin   d2W  d2UV |EV|   Bin
%                                                (m)          (*1e-3)           (*1e-3)  (*1e-3)
  0   0  33 89/04/20  03:33:22    V 32767 32767  -1 255 255   2203.3  32767    0     0     0 32767
  0   0  34 89/04/20  03:38:21    D 32767 32767  -1 255 255     -1.0      9  -47  -203     0 32767
  0   0  35 89/04/20  03:43:21   AD     1     2  29 222 250     -1.0      9  -47  -200     0 32767
  0   0  36 89/04/20  03:48:21   AV     1     3  37 204 249   1069.3  32767    0     0     0 32767
  0   0 120 89/04/20  10:48:21    A    19    23 197 100 197     -1.0  32767    0     0     0 32767
  0   0 121 89/04/20  10:53:20 AVDE    20    24 205 102 192    781.1     27  -94   510   510     4
```

Following is a brief explanation of each column entry in a line:

a. The "tag" field which is always set to 0 to be used for the manual editing procedures described later
b. The block number of the flagged profile
c. The profile number of the flagged profile
d. The profile time in yy/mm/dd hh:mm:ss format
e. The flag string consisting of any one or combination of the letters "A", "V", "D", and "E" to indicate which criteria caused the profile to be flagged
f. The bin at which the amplitude reached a local minimum prior to increasing (a default setting of 32767 is used if the profile is not flagged by the amplitude criterion)
g. The max_amp_bin or the bin at which the amplitude reached the local maximum (a default setting of 32767 is used if the profile is not flagged by the amplitude criterion)
h. The depth at *max_amp_bin* (default value = -1)
i. The value of amplitude at the *min_amp_bin* (default = 255)
j. The value of amplitude at the *max_amp_bin* (default = 255)
k. The computed *W* variance if the profile was flagged by the *W* variance criterion (defaults to -1.00 otherwise)
l. The bin at which the second difference exceeded the established thresholds if the profile was flagged by the second difference criterion (defaults to 32767 otherwise)
m. The computed second difference of *W* at the flagged bin (defaults to 0 otherwise)

n. The computed second difference of *U* or *V*, whichever first exceeded the *U-V* second difference threshold (defaults to 0 otherwise).
o. The largest error velocity found to exceed the specified threshold.
p. The bin at which the maximum error velocity was found.

==========
MANUAL EDITING PROCEDURE I
==========

INPUT: log file output from flag program, profile plots
OUTPUT: updated log file from flag program

This step consists of using the output log file of the flag program described above to focus on sections of the database with the noted irregularities. By examining the plots of the profile data for those sections, one can decide on any of the following courses of action: (Note: the plots may be obtained by using the MATLAB routines described below under PROGRAMS: getmat & MATLAB M-files)

1. That the flagged profile does not really contain bad data at all (just noise) and the data for that profile in the CODAS database should be left as is. The entry for that profile should be commented out from the flag log file using any text file editor by inserting the '%' symbol at the beginning of the line.

2. That the bad data occur at isolated bins within a profile, probably as a result of instrument or random glitches (internal glitches) rather than bottom interference. Consequently, the velocity data at glitchy bins should be set to bad values using getmat, badbin and the MATLAB M-files described below. The entries for such profiles should also be commented out from the flag log file, using a text file editor to insert '%M' at the beginning of the line, thereby marking them for later reference when using the MATLAB routines.

3. That there has indeed been some bottom interference. In addition, it may be determined that certain other unflagged profiles in the vicinity of a flagged region must also be flagged, perhaps because the thresholds were not set tightly enough or the algorithm simply failed, in which case additional entries must be made in the flag log file using the updscan program described below. Preparatory to using this program, the flag log file should be edited to indicate the type of augmentation that must be done:

a. if unflagged profiles prior to a flagged entry must also be flagged, one must insert a line prior to the flagged entry with a tag set to 1 and the time of the profile at which to start additional flagging

b. if unflagged profiles following a flagged entry must also be flagged, one must set the tag of the flagged entry to a 2 and insert a subsequent line indicating the time of the profile at which to stop additional flagging.

There are two ways to flag additional profiles :

a. using time key
b. using block-profile index key

Example 1: Using time key:

```
0   4 411 86/01/19  01:30:00    A    38    57   470   89 121    -1.00 32767       0          0
1         86/01/19  02:00:00
```

```
0   4 423 86/01/19  02:30:00   VD 32767 32767    -1 255 255  1566.36     1    -61       -204
2   4 424 86/01/19  02:35:00   AV    60    61   502 138 162  2761.30 32767      0          0
          86/01/19  02:40:00
0   4 426 86/01/19  02:45:00    D 32767 32767    -1 255 255    -1.00     1    -78       -238
```

Example 2: Using block-profile index key:

```
0   4 411 86/01/19  01:30:00    A    38    57   470  89 121    -1.00 32767      0          0
1   4 417
0   4 423 86/01/19  02:30:00   VD 32767 32767    -1 255 255  1566.36     1    -61       -204
2   4 424 86/01/19  02:35:00   AV    60    61   502 138 162  2761.30 32767      0          0
    4 425
0   4 426 86/01/19  02:45:00    D 32767 32767    -1 255 255    -1.00     1    -78       -238
```

The tag of 1 in the second line of Example 1 means that additional entries should be made for profiles from time 86/01/19 02:00 (inclusive) to 86/01/19 02:30:00 (exclusive --already there); the second line of Example 2 indicates that additional entries should be made from profile 417 (inclusive) to profile 423 (exclusive) of block 4.

The tag of 2 in the fourth line of Example 1 means that additional entries should be made for profiles from time 86/01/19 02:35:00 (exclusive) to 86/01/19 02:40:00 (inclusive); the fourth line of Example 2 indicates that additional entries should be made from profile 424 (exclusive) to profile 425 (inclusive) of block 4.

The output of this step is an updated flag log file for use as input to the logscan program if using time as the key, or to the blkscan program if using *BLOCK/PROFILE* as the key (both described below). In addition, the entries commented out with a '%M' will be used as a reference for the *getmat* & MATLAB M-files described below.

==========

PROGRAM: logscan

==========

USAGE: logscan <CODAS database name> <input file name> <output file name>

INPUT: the manually edited flag log file and the CODAS database

OUTPUT: augmented flag log file--gaps indicated by tags have been filled in

This program reads each data line in the edited flag log file. The input file must use time as the key. If the tag is set to 0, that line is merely copied over to the output log file.

If the tag is set to 1, the adjacent time is read and the CODAS database is searched for this time. Data for each profile with time equal to or greater than this time and less than the time indicated on the subsequent log line are logged to the output file. The flag string for these added data is set to 'N' and the other fields are set to their default values. The tag field is reset to 0 for further use in the second set of manual editing procedures described below.

If the tag is set to 2, the line is copied over to the output log file. The CODAS database is searched for the adjacent time. Data for each profile with time greater than this time and less than or equal to the time indicated on the subsequent log line are logged to the output file. The flag string for added data is set to 'N' and the other fields are set to their default values. The tag field is reset to 0 for further use in the second set of manual editing procedures described below.

The output of this step is a log file with a complete list of profiles that have been determined to be contaminated by bottom interference. It is used as an input file to the second set of manual editing procedures described below.

==========
PROGRAM: blkscan
==========

USAGE: blkscan <CODAS database name> <input file name> <output file name>

INPUT: the manually edited flag log file and the CODAS database

OUTPUT: augmented flag log file--gaps indicated by tags have been filled in

This is a *BLOCK/PROFILE* search version of logscan. The input filemust use *BLOCK/PROFILE* as the key.

==========
MANUAL EDITING PROCEDURE II
==========

INPUT: flag log file last updated by the logscan or blkscan program

OUTPUT: updated flag log file

Using a text file editor, the flag log file last updated by the *logscan* or *blkscan* program should be edited to set the tag field for each line. If a flagged profile has good data at the upper bins, then the tag is left as 0 to indicate that the value for the *max_amp_bin* field in the *ANCILLARY_2* structure of each profile in the CODAS database should be read from the log file.

NOTE: If a profile in the flag file was added using either the *logscan* or *blkscan* programs (indicated by the flag 'N'), then it would have only the default value of *MAXSHORT* (32767) for the *max_amp_bin* field. The user MUST edit the *max_amp_bin* field in this case to plug in his own reasonable estimate based on the profile plots or the behavior of adjacent profiles.

If a flagged profile has bad data for all bins, then the tag is set to 1 to indicate that the value for the *last_good_bin* field in the *ANCILLARY_2* structure should be set to -1.

This step produces an updated log file for use with the updscan program described below.

==========
PROGRAMS: getmat & MATLAB M-files
==========

USAGE: edit the setup.m file as indicated below to supply the appropriate parameters; invoke MATLAB; and call up the M-files as MATLAB functions.

INPUT: CODAS database and the flag log file (in particular, those lines commented out by '%M')

OUTPUT: badbin.asc file containing time, block and profile numbers of profiles that have bad data for certain bins, a count of the number of bad bins followed by the numbers of the bins with bad data

These programs allow the MATLAB user to retrieve *U, V, W,* error velocity, amplitude, and percent good data for a given block and profile number from the CODAS database into *.mat* files. The *.mat* files are loaded into the MATLAB workspace. The *U, V, W,* error velocity, amplitude and percent good data can then be plotted against the bin numbers. The MATLAB user can then determine from the plots the bin numbers for which the *U, V, W,* and error velocity data ought to be set to bad due to internal glitches. A MATLAB routine can then be invoked to record these bin numbers in the output file *badbin.asc.*

The '%M' lines from the flag log file can be used to identify which profiles should be plotted for examination. It is a good idea to also examine unflagged profiles immediately adjacent to those already flagged, just in case they show some marginal glitchiness that may have slipped through the detecting algorithms or insufficiently stringent thresholds.

Following is a listing of the M-files and their respective functions:

---

1. setup.m

USAGE: setup

OUTPUT:default valuesfor the parameters described below; globalized variables.

This M-file file globalizes and initializes key variables. The user must edit this file to provide the desired default values for the following parameters used by get.m (see get.m below):

a.*DBNAME* = CODAS database name (may include path)

b.*DEFAULT_NPRF* = default number of profiles (*NPRF*) to retrieve in addition to the key profile

c.*DEFAULT_NBINS* = default number of bins to retrieve for each profile

d.*DEFAULT_SEQ_FLAG* = 0 if retrieving *NPRF* profiles before & after key profile; 1 if retrieving *NPRF* profiles after key profile .

It also declares important variables– *U, V, W, E, AMP, PGOOD, KEY_BP, UBAK, VBAK, WBAK, EBAK, YAXIS, DISP, NBINS, NPRFS, TIME, LISTED,* and the *DEFAULT_** --to be global for access by subsequent M-files.

---

2. get.m

USAGE: get(<block number>, <profile number>)

get(<block number>, <profile number>, <no. of add'l profiles>)

get(<block number>, <profile number>, <no. of add'l profiles>, <maximum number of bins>)

INPUT: the block and profile number of the "key" profile to view; optionally, the number of adjacent profiles to retrieve in addition (*NPRFS*) and the maximum number of bins per profile

OUTPUT: .mat files containing *U, V, W,* error velocity, amplitude, percent good, and other data for the key and adjacent profiles .

This M-file uses the external program *getmat* to retrieve *U, V, W,* error velocity, amplitude, and percent good data for selected profiles in a CODAS database into *.mat* formatted files. The .mat files are named after the variables they contain (e.g., *'u.mat'* for *U* and *U_MEAN* data). They contain the following arrays:

Data Arrays:

a. *U* = a 2-dimensional array (number of bins by number of profiles retrieved) of de-meaned *U* velocity data

b. *U_MEAN* = a 1-dimensional array of the mean *U* value over bins 5 to 20 (or *NBINS*, if smaller) for each profile

c. V = a 2-dimensional array of V velocity data

d. V_MEAN = a 1-dimensional array of the mean *V* value over bins 5 to 20 (or NBINS, if smaller) for each profile

e. *W* = a 2-dimensional array of *W* velocity data

f. *W_MEAN* = a 1-dimensional array of the mean *W* value over bins 5 to 20 (or *NBINS*, if smaller) for each profile

g. *E* = a 2-dimensional array of error velocity data

h. *AMP* = a 2-dimensional array of amplitude data

i. *PGOOD* = a 2-dimensional array of percent good data

Auxiliary Arrays/Data in other.mat:

f. *TIME* = a 2-dimensional (number of profiles by 6) array containing the time (year, month, day, hour, minute, second) of each of the profiles retrieved

g. *KEY_BP* = a 2-element vector containing the block number and the profile number of the key profile

h. *IBLKPRF* = a 2-dimensional array (number of profiles by 2) containing the block and profile numbers of each of the profiles represented in the data arrays

i. *NBINS* = no. of bins retrieved

j. *NPRFS* = no. of profiles retrieved

The program getmat requires the following parameters:

a. the name of the CODAS database

b. the block number of the key profile

c. the profile number of the key profile

d. the number of adjacent profiles (NAP) to retrieve in addition to the key–an optional argument which defaults to 2 if not provided

e. the maximum number of bins to retrieve for each profile-- an optional argument which defaults to 128 if not provided

f. the setting of the sequential flag (0 means retrieve *NPRFS* profiles before and after the key, 1 means retrieve *NPRFS* profiles after the key) --an optional argument which defaults to 0 if not provided.

The values of the these parameters must be set up in setup.m file. The get.m function makes it easier to call the getmat program by using the simplest form of the get.m invocation, i.e., the one that merely specifies the block and profile number of the key profile.

---

3. init.m

USAGE: init

INPUT: .mat file name (must already exist)

OUTPUT: MATLAB workspace arrays (U, V, W, E, AMP, PGOOD, TIME, KEY_BP, IBLKPRF, UBAK, VBAK, WBAK, EBAK, YAXIS, DISP) and variables (NBINS, NPRFS).

This M-file is used to perform several functions, including:

a. loading a .mat file containing the *U, V, W, E, AMP, PGOOD,*

b. generating the *YAXIS* array containing the bin numbers for use as the y-axis data in plotting

c. making backup copies of the *U, V, W,* and *E* arrays (*UBAK, VBAK, WBAK* and *EBAK*) for later reference by restore.m (described below).

In addition, it checks that the immediately preceding set of profiles loaded and flagged (if any) have been listed (see list.m below); if not, it lists them out prior to loading the *.mat* file.

---

4. offset.m

USAGE: offset(<multiple>)

INPUT: a constant specifying the multiple to be used in generating the displacement array

OUTPUT: DISP = the MATLAB displacement array

In order to neatly plot the *U, V, W, E, AMP,* and *PGOOD* data foreach of the profiles, the draw.m function (described below) causes each profile to be displaced to the right by adding a positive displacement array, *DISP,* to the data arrays just prior to plotting. The offset.m function is used to generate this displacement array. The user may specify any constant as an argument. If the displacements generated for a given constant are too small, the profile plots would overlap. If the displacements generated are too large, the profile plots would be widely spaced apart at the cost of blurring features. The offset.m function can be called several times to experiment with different constants/displacements. Because of differences in scale, offsets for the velocity data are usually <= 1, while good offsets for amplitude and percent good data start at around 50. Reasonable default values are provided.

---

5. draw.m

USAGE: draw(<array name>)
draw(<array name>, <minimum bin>, <maximum bin>)

INPUT: name of data array to be plotted; optionally, the bin range to be plotted (defaults to all bins if not provided)

OUTPUT: plot of data array vs. bin numbers

This function file is used to plot data for the array named as the argument. It takes care of transposing the data array, adding the displacements, selecting the symbols used ('+' is used for the key profile), specifying the *YAXIS* array containing the bin numbers as the y-axis data, putting grid lines, axes labels, title, and time stamp.

---

6. bad.m

USAGE: bad(<bin vector>)
bad(<bin vector>, <block-profile vector>)
bad(<bin vector>, <no. of profiles>)
bad(<bin vector>, <block-profile vector>, <no. of profiles>)

INPUT: vector of bin numbers contaminated by an internal glitch; optionally, block and profile number of the profile containing the bin (defaults to key profile if not provided) optionally, the number of profiles subsequent to the key profile for which to flag the same bins.

OUTPUT: updated U, V, W, and E arrays (value at bad bin(s) set to NaN)

This function is used to flag bins with contaminated data. After examining the data plots, the user determines which bins of which profiles ought to be flagged as bad so that the *U, V, W,* and *E* velocity data for those bins could be set later to bad in the CODAS database.

Flagging is done by setting the *GLITCH_BIT* of the *PROFILE_FLAGS* byte array for the indicated bins if the *PROFILE_FLAGS* variable has been loaded into the database. Otherwise, flagging is done by setting the *U, V, W,* and *E* velocities at the flagged profile bin to *NaN*.

The user invokes the bad.m function for a given set of bins to be flagged, for some key profile and, optionally, a number of profiles subsequent to the key. If the user provides only the bin vector as argument, it assumes that the bin(s) for the key profile (the one drawn with the '+' symbol) is being flagged. If the user also provides the block and profile number, it assumes that the user wants to reset the key profile to that block-profile number, and to flag the bin(s) for the new key. Upon redrawing, the new key will be drawn with the '+' symbol. If the user also provides the number of additional profiles to be flagged, the bins indicated in the <bin vector> will be flagged for the key profile and for the <no. of profiles> following the key.

For example: Suppose the user inputs the following in sequence:

bad(3,[0 4]) -> sets bin 3 of block 0 profile 4 to be bad
bad(4:10) -> sets bins 4 to 10 of block 0 profile 4 to be bad
bad([7 10:20],[0 5],3)-> sets bins 7, 10 to 20 for block 0 profile 5 as well as for the next 3 profiles to be bad
bad(3) -> sets bin 3 of block 0 profile 5 to be bad
bad(1:2,5) -> sets bins 1 and 2 for block 0 profile 5 as well as for the next 5 profiles to be bad

NOTE: If one redraws either *U, V, W,* or *E* after flagging groups of profiles, MATLAB will no longer show the bins for which *U, V, W* and *E* have been set to *NaN*. In the case of the key profile which is drawn using the symbol '+', the *NaNs* will be display as missing points. In the case of the non-key profiles which are drawn using lines, the *NaNs* will be interpolated across. In either case, the glitches will no longer be seen as wild aberrations in the plot.

---
7. restore.m

USAGE: restore(<bin vector>)
restore(<bin vector>, <block-profile vector>)
restore(<bin vector>, <no. of profiles>)
restore(<bin vector>, <block-profile vector>, <no. of profiles>)

INPUT: vector of bin numbers which have been flagged as bad; optionally, block and profile number of the profile containing the bin (defaults to key profile if not provided); optionally, the number of profiles subsequent to the key profile for which to restore the same bins

OUTPUT: updated *U, V, W,* and *E* arrays (value at restored bin(s) reset from *NaN* to the original value stored in *UBAK, VBAK, WBAK,* and *EBAK*)

This function is used to undo the effects of the bad.m function in case the user changes his mind about flagging certain bins. Undoing means restoring the *U, V, W,* and *E* values at those bins from *NaN* to the original values. *restore.m* works in much the same way as bad.m, except that it resets the values back from *NaN* to the original.

For example: To undo all the flagging done in the preceding example, the user inputs the following in sequence:

restore(3,[0 4]) -> unflags bin 3 of block 0 profile 4
restore(4:10) -> unflags bins 4 to 10 of block 0 profile 4
restore([7 10:20],[0 5],3) -> unflags bins 7, 10 to 20 for block 0 profile 5 as well as for the next 3 profiles
restore(3) -> unflags bin 3 of block 0 profile 5
restore(1:2,5) -> unflags bins 1 and 2 for block 0 profile 5 as well as for the next 5 profiles

---
8. list.m

USAGE: list

INPUT: TIME, IBLKPRF, U, UBAK arrays in MATLAB workspace

OUTPUT: updated badbin.asc file (profile time, block-profile number, and bad bins appended to file)

Once the user is satisfied that all bad bins in the current set of profiles have been properly flagged using the bad.m function, the bad bin numbers for each profile (as well as the profile time, for identification) can be permanently recorded in the badbin.asc file by list.m. The ASCII file badbin.asc will later be used as input to the badbin program (described below) for updating the CODAS database.

Following are a few sample lines from badbin.asc:

```
89/ 4/20  0:53:22    0    1    2    4   14
89/ 4/20  0:58:21    0    2    1   15
89/ 4/20  4:58:21    0   50    1    3
89/ 4/20  7:13:21    0   77    2    3    9
```

In the first line above, the profile corresponding to block 0, profile 1 in the CODAS database has 2 bad bins: bin #4 and bin #14.

NOTE: Failing to type the list command prior to getting another set of profiles and initializing causes an automatic list during the init command. Hence, the user can skip the list command for most of the session, except for the very last time prior to exiting MATLAB, because there would be no subsequent init command to automatically execute list. The user must remember to type list before exiting in order to record the last set of flagged bins. As a matter of good practice, however, it is recommended that the list command be typed each time, rather than depending on init.

==========
PROGRAM: badbin
==========

USAGE: badbin <CODAS database name> <badbin.asc>
INPUT: badbin.asc and the CODAS database
OUTPUT: updated CODAS database

This program uses the file badbin.asc generated by the MATLAB M-files in the preceding step to update a CODAS database. It reads a line from badbin.asc for the block and profile number, the bad bin count, and the numbers of the bins that have been flagged as bad. It then retrieves the *PROFILE_FLAGS* variable from the database, sets the *GLITCH_BIT* for the flagged bins, and stores the updates *PROFILE_FLAGS* back into the database. (If the *PROFILE_FLAGS* variable has not been loaded, it instead retrieves the *U*, *V*, *W*, and *E* velocity arrays and sets their values at the indicated bins to bads then stores the updated *U*, *V*, *W* and *E* arrays back in the CODAS database.) It turns on bit 7 of the data processing mask to reflect that the database has been edited for glitch detection.

==========
PROGRAM: updscan
==========

USAGE: updscan <CODAS database name> <updated flag log file name>
INPUT: CODAS database last updated by badbin, and the flag log file last updated by the second set of manual editing procedures
OUTPUT: updated CODAS database

This program is used to set the *max_amp_bin* and *last_good_bin* fields of the *ANCILLARY_2* structure of each profile in the CODAS database. For each profile, the *ANCILLARY_2* structure is retrieved from the CODAS database. If a profile has not been flagged in the input log file, then the *max_amp_bin* is set to *BADSHORT* and the *last_good_bin* is set to the largest bin number to indicate that all bins are considered uncontaminated by bottom interference. (This step should actually be performed at the time of database creation but has been included here because earlier CODAS databases might not have these fields properly set at the time of creation. The program should later be updated to exclude this step, in which case it would merely retrieve and update the *ANCILLARY_2* structures of profiles that have been flagged in the log file.) If a profile has been flagged in the input log file, then the *max_amp_bin* and the *last_good_bin* are set depending on the value of the tag field. If the tag is set to 0, then the value for the *max_amp_bin* field is read from the log file and stored in the *max_amp_bin* field of the *ANCILLARY_2* structure. The *last_good_bin* field of the *ANCILLARY_2* structure is temporarily set to *BADSHORT* for further updating by a subsequent step. (Setting *last_good_bin* to *BADSHORT* would not also be necessary if this field had been appropriately set at the time of database creation.)

NOTE: If a profile in the flag file was added using either the logscan or blkscan programs (indicated by the flag 'N'), then it would have only the default value of *MAXSHORT* (32767) for the *max_amp_bin* field. The user MUST edit the *max_amp_bin* field in this case to plug in his own reasonable estimate based on the profile plots or the behavior of adjacent profiles.

If the tag is set to 1, then the *max_amp_bin* field is set to *BADSHORT* and the *last_good_bin* field is set to -1 to indicate that none of the bins has good data. (Again, setting the *max_amp_bin* field to *BADSHORT* would not also be necessary if this field had been properly initialized at the time of database creation.)

The updated *ANCILLARY_2* structures are stored back in the CODAS database. The output of this step is an updated CODAS database with the *last_good_bin* and *max_amp_bin* fields of the *ANCILLARY_2* structure for each profile appropriately set to indicate the bin-by-bin reliability of data.

==========
PROGRAM: botmpas3
==========

USAGE: botmpas3 <CODAS database name>
INPUT: CODAS database last updated by updscan
OUTPUT: updated CODAS database

This program resets the *last_good_bin* field of the *ANCILLARY_2* structure of each profile in a CODAS database to the lowest *last_good_bin* value that occurs in a group of three consecutive profiles centered on the profile being updated. This provides more conservative cutoff points for the *last_good_bin*. The first and last profiles are updated using data for the key and one adjacent profile only. The program also turns on bit 9 of the data processing mask to reflect that the database has been edited for bottom detection.

==========
PROGRAM: last_85
==========

USAGE: last_85 <CODAS database name>
INPUT: CODAS database last updated by botmpas3
OUTPUT: updated CODAS database

This program cuts off the *last_good_bin* further to 85% of the original for those profiles contaminated by bottom interference. It turns on bit 10 of the data processing mask to indicate that the database has been edited to set the *last_good_bin* to reflect the maximum bin that is unaffected by bottom interference.

****

Etalonnage

(d'après E. Firing)

This document describes a procedure to calibrate the ADCP data for any misalignment between the transducer and the ship's keel. There are essentially two stages:

1) estimating the misalignment angle and amplitude ("calibration"), and
2) applying the correction to the ADCP data ("rotation").

For estimating the angle of misalignment, two methods are described:
1) water track method,
2) bottom track method.

The calibration correction function used is:

$$U_c = A * \exp(i*\alpha) * U_u$$

where U_c and U_u are the corrected and uncorrected velocities, respectively. Velocity here is expressed as a complex number U = u + i*v, where u and v are the u- and v-components of the velocity. *A* is the amplitude or scaling factor, a dimensionless number, and α is the counterclockwise angle in degrees (the misalignment angle) from the gyrocompass forward axis (which should be aligned with the ship's keel) to the transducer forward axis.

Water track method

As described by Pollard and Read (*JAOT, 6, 860-865,* 1989) and Joyce (*JAOT, 6, 169-172,* 1989), in situ calibration of the ADCP's water tracking performance can be done during accelerations by comparing changes in velocity of the ship relative to a reference layer as measured by the ADCP to changes in the velocity over the ground as measured by GPS. The UH ADCP data processing system includes two programs that work together to calculate these calibration factors: TIMSLIP.EXE (UNIX version: timslip), an executable program that calculates calibration factors for all suitable accelerations, and ADCPCAL.M, a Matlab function that edits, plots, and summarizes the output of TIMSLIP. Additional programs that are involved include ADCPSECT to calculate the ship's velocity relative to a reference layer, UBPRINT to extract the GPS fix file, EDFIX (optional) to edit the fixes, a text editor, and, of course, MATLAB.

Procedure

It is assumed that the starting point is an ADCP CODAS3 database, and that the data have been collected with a UH user exit program (MAG1157 or MAG2 or UE3) as interface to one or more GPS receivers.

1) If it has not already been done, run UBPRINT to get a GPS fix file. Depending on the user exit program version and available navigation, one could use either or both of the options GPS_summary or avg_GPS_summary; the former is always available, but the latter is available only if burst averaging of GPS fix messages has been done. If both are available they should be quite similar, but the "avg_" version should be slightly superior in fix accuracy at the possible cost of a reduced number of valid fixes. The GPS_summary output file has extension `.gps` and the avg_GPS_summary output file has extension `.ags`.

2) Optionally use EDFIX to edit the GPS fix file. It is also a good idea to use a text editor to scan the fix file and manually remove (comment out by putting a percent sign at the beginning of the line) any fixes that are obviously wild. EDFIX edits based on quality parameters alone; it knows nothing about what a reasonable fix looks like, or what part of the ocean the ship was in.

3) Run ADCPSECT to get a file containing only the ship's velocity relative to a reference layer as a function of time. A typical reference layer is bins 5 to 20. If the depth range of the ADCP was poor, one might want to use a shallower layer such as 2 to 10 or 2 to 15. The relevant output file will have the extension `.nav`.

4) Using the output of ADCPSECT and UBPRINT (and EDFIX) as input, run TIMSLIP to produce a single output file containing calibration parameters and quality indications for each ship acceleration that meets your specifications and for which GPS fixes are available. The output file name is specified including extension by the user, but the extension `.cal` is customary.

5) Use a text editor to extract the tabular information (that is, everything but the commented out header lines and blank lines) into another file, which can be a temporary file called `cal` (or whatever you like). The reason for this step is that this file will be read into Matlab as an ASCII file, and Matlab does not allow any form of comments or text in such files.

6) Run Matlab, load `cal` (type `load cal.`; you do need the dot if the filename lacks an extension), and run the function `adcpcal`. The only compulsory arguments are the name of the array (which is the filename less extension) created from the loaded file, and a text string that will serve as part of a plot and output label. Typically one would type something like:

```
adcpcal(cal, 'MW9004 cal 1A')
```

The result will be several plots on the screen (hit a key after each to continue), most of which will be saved automatically in a meta file called `cal.met`, and a summary of the calibration statistics which will be appended to a file called `adcpcal.out`. Caution: `cal.met` will be overwritten each time `adcpcal` is run, so if you want to save the graphical output from several runs, rename cal.met to something else immediately after each run is created.

7) When you have analyzed all available calibration information and decided what calibration parameters to use, the program ROTATE is used to perform the calibration on the velocities in the database. The use of ROTATE is explained elsewhere (page 45).

Details

ADCPSECT: Example control file: (with comments deleted)
dbname: ../adcpdb/a011

```
output: a011
step_size: 1
ndepth:   20
time_ranges: separate
year_base= 1990
option_list:
     pg_min= 30
     navigation:
        reference_bins 2 to 15
     end
     statistics:
        mean
        units= 0.01      /* cm/s instead of default m/s */
     end
 end
90/09/13 18:33:29 to 90/09/19 08:37:26
```

Notes: 1) `step_size` must be 1. 2) `ndepth` must equal or exceed the deepest reference layer bin. 3) A `pg_min` threshold usually provides adequate editing for this purpose. Typical values are 30 to 50. Don't go below 25. 4) Include only `reference_bins` in the list of navigation options, as above. The range should be chosen to be the thickest for which percent good is generally high (say 90 or above), except that one may wish to exclude the top few bins if they are likely to have higher variance than deeper bins. Alternatively, the program can be run several times with different bin ranges (and different output file names!) so that any possible change in calibration amplitude factor (scale factor) with depth can be investigated.

```
TIMSLIP: -------  Example control file:
fix_file_type:        { simple or HIG }
fix_file:
reference_file:
output_file:
year_base=          { year base for decimal day calculations }
min_n_fixes=    { minimum number of fixes for calculation of time shift to proceed. }
{ The jump in velocity will be between indices (n_refs-1)/2 and (n_refs-1)/2 +1. For
example, it will be between 6 and 7 if n_refs is 13. Note that all these indices start
from 0, C-style, not from 1. However, the lowest index specified must be >= 1, since
the time of the start of ensemble 1 is (approximately) the time recorded with
ensemble 0.}
n_refs=      { number of ensembles used in calculation of time shift }
i_ref_l0=           { first and last indices of ensembles used }
i_ref_l1=             in calibration calculation }
i_ref_r0=           { "l" is before jump, "r" is after }
i_ref_r1=           { minimum is 1, not zero }
up_thresh=            { m/s, for jump detection }
down_thresh=
dtmax=    360       { max seconds between ensembles, to screen gaps }
tolerance= 5.e-5     { for BRENT, in days }
```

grid: { fix or ensemble; calculate reference layer velocities averaged between fix times, or over ensemble intervals}
use_shifted_times? { yes or no; use the best-fit time shift in the calibration; or, if an integer is given instead of "yes" or "no", a constant shift of that many seconds will be applied. }

NOTES:

1) The original function of TIMSLIP was to detect ship accelerations from the ADCP and compare them to the GPS record, sliding them around in time to find a time offset that provides a best fit. Hence the name TIMeSLIP. This function was needed because for some datasets the ADCP recorded time could be in error by as much as a few minutes, and with navigation data recorded independently, this offset caused substantial errors in the calculation of absolute reference layer velocity.

2) fix_file_type should always be "simple"; "HIG" refers to an archaic format that was the only one available for some early UH data sets.

3) min_n_fixes will normally be the same as n_refs, and both should be an odd number, 5 or larger. They determine the time span, measured in ADCP ensembles, within which accelerations are detected and the "time slip" calculation is done. The only problem with specifying a long span is that it will cause the accelerations to be missed in the case of a brief CTD station, for example. With 5-minute ensembles I normally use 7, but occasionally go to 5 in order to catch the maximum number of accelerations.

4) The i_ref_xx indices should be chosen as (1 1 4 4) or (1 2 3 4) for n_refs of 5; (1 2 5 6) or (1 3 4 6) for n_refs of 7; (1 3 6 8) or (1 4 5 8) for n_refs of 9; and so on. These indices determine the ranges of ensembles that are used in calculating the calibration coefficients. I think the results are generally a little better with the first of the alternatives in each example above, because it excludes the two ensembles nearest the time of the acceleration. GPS and ADCP velocity estimates are averaged over the ranges specified by the first and last pair of indices in the quadruplet.

5) up_thresh and down_thresh should be about half the normal underway speed of the ship. The algorithm for detecting significant accelerations looks for runs of n_refs/2 consecutive ensembles for which the magnitude of the speed difference between the current ensemble and the ensemble n_refs ahead exceeds up_thresh for an acceleration and down_thresh for a deceleration. For a turn, the criterion to be satisfied similarly is that the magnitude of the course difference exceeds turn_thresh and the speed (for each ensemble involved) exceeds turn_speed. For the latter two parameters, values of 60 degrees and 2 m/s are reasonable.

6) "dtmax" should be just a few seconds more than the ensemble length. It is used to detect data gaps, so that only continuous data are used in the calibration.

7) Leave "tolerance" at around 5.e-5 days (about 5 seconds); it specifies the tolerance for the iterative "time slip" calculation.

8) The "grid" variable should normally be "ensemble". It determines whether all interpolations are to the grid of fix times or ensemble times. These times are nearly coincident for GPS fixes recorded with the UH user exit program, so the setting of "grid" is not critical.

9) The variable "use_shifted_times" should initially be set to "no", again assuming the ADCP data have been loaded with time corrections so that ADCP times and fix times are both correct to within a few seconds. The "yes" setting causes

the calibration factors for each acceleration to be calculated after locally adjusting the ADCP ensemble times for a best fit (using the "time slip" calculation). If the ADCP ensemble times and fix times are not actually in error, this adjustment is a response to noise, not signal, and therefore will tend to make the calibration factors less accurate. A third alternative is to set the variable to an integer number of seconds of shift. For GPS fixes collected through the Magnavox port A and listed via the "GPS_summary" option in UBPRINT, the usual value is 5 to 7 seconds. This is the typical amount by which the recorded fixes effectively preceed the end of the ensemble. It is estimated by the mean value in the column labelled "shift" in the TIMSLIP output, and summarized under the label "nav-pc" in the ADCPCAL output. The "time slip" calculation is done and the "nav-pc" column is displayed regardless of the setting of the "use_shifted_times" variable; the latter controls only the application of a shift in the subsequent calibration calculation. It is unlikely that the "yes" setting will be appropriate for any future data sets.

10) The TIMSLIP output will normally include quite a range of values including several obviously wild points. These can be edited out manually if desired, but it is easier to just let ADCPCAL to that automatically.

ADCPCAL: ADCPCAL is configured by editing the m-file directly to change the following section:

```
% parameters for editing:
                    % phase seems more often to contain a trend than amplitude,
                    % so it may be justified to clip phase less stringently
clip_ph  = 3;       % degrees on either side of median
clip_amp = 0.04;    % eliminate anything over x percent from median
clip_var = 0.05;    % maximum variance of ref layer velocity
clip_dt  = 60;      % maximum time slip beyond median
```

NOTES:

1) The values given above are good starting points, and there may be no need to change them at all. The idea of the editing is simply to remove outliers and points that contain no good information about the calibration factors. Poor calibration data, typically because of GPS deficiencies, is indicated by excessive values of "time slip" (set "clip_dt"), and by poor agreement between the velocity from GPS and that from the ADCP (set "clip_var"). Outliers are directly indicated by large deviations of the calibration angle and amplitude from their medians (set "clip_ph" and "clip_amp", respectively). When in doubt, set the editing parameters loosely at first, then look at the ADCPCAL output to see the standard deviations of "phase", "amplitude", and "nav-pc", and the mean and standard deviation of "var". Reasonable clipping points would then be 2-3 standard deviations of the first 3 of these, and the mean plus 2-3 standard deviations of "var".

2) Typical results for standard deviation of phase and amplitude after editing are 0.6-1.2 degree and 1-1.5%, respectively. This level of jitter in the calibration calculation comes from GPS error plus actual variations in the ocean reference layer velocity during each calibration acceleration interval. In addition, phase tends to be associated with larger velocity variance than amplitude because of errors in the gyrocompass, both short-term Schuler oscillations and long-term drift.

3) Ideally, over the course of a month-long cruise, one obtains 50 or so calibration points, and the plot of phase and amplitude versus time suggest a gradual drift in phase of up to about a degree, and perhaps a very small drift in amplitude. One can fit a curve by eye or numerically. For the latter, one method we have used is to divide the calibration points into groups, perhaps weekly, average them separately, and use spline interpolation to run a smooth curve through the averages. In addition, one can incorporate the results of bottom track calibrations, usually at the start and end of the cruise.

Bottom Track Method

This method estimates the misalignment angle and amplitude by comparing the ship's position, as estimated from the bottom tracking information, to that determined by navigation information. It is applicable only when there is a sufficient overlap in the time ranges for which both bottom tracking and satellite navigation information are available.

In general, the bottom track velocity is available only for certain time ranges (coinciding with the ship being in shallow water). It is best to perform the calibration calculations separately for each such time range, resulting in as many estimates of the misalignment angle and amplitude. Based on actual conditions, one can take these estimates and decide on any one of the following courses of action:

1. Average out the estimates.
2. Dtermine that a different correction be made on different sections of the ADCP data.
3. Use the water track method to confirm/reject certain estimates.

In any case, for each such time range, following are the steps to be done to generate an estimate:

1. Run the *lst_btrk* (list bottom track) program to retrieve the bottom velocities from the ADCP database into an output text file.

```
====================
PROGRAM: lst_btrk
====================

USAGE: lst_btrk <control file name>
INPUT: CODAS database, control file
OUTPUT: A file containing profile time, the velocity of the ship relative to the
bottom, and bottom depth.
```

SAMPLE CONTROL FILE: *lst_btrk.cnt* :

```
dbname:   ../adcpdb/ah18
output:   ah18.bt
step_size= 1
year_base= 1990
```

time_ranges:
90/06/11 22:04:06 to 90/06/15 18:22:15

EXAMPLE OF OUTPUT FILE FROM *lst_btrk*:

```
%Bottom Track time, u, v, depth for :
%%,  dbname: ah18,  output: ah18.bt,  step_size= 1,  year_base= 1990
%
              161.919514    2.058   -0.460     12
            %
              161.926458    1.385   -1.975     14
              161.929919   -0.673   -2.297     13
              161.933391   -1.535   -2.917     13
              161.936863   -3.113   -3.599     29
              161.940347   -3.701   -4.494    219
              161.943819   -3.944   -4.720    402
              161.947280   -4.601   -3.976    464
              161.950752   -5.423   -2.270    463
              161.954225   -4.385    0.856    451
              161.957697   -5.742   -0.920    439
              161.961192   -6.095   -2.080    432
              161.964653   -6.379    0.034    414
              161.968125   -5.414   -2.976    393
              161.971620   -5.991   -1.289    362
              161.975069   -6.231   -0.966    347
              161.978542   -6.343   -0.983    414
            %
```

2. Obtain the output file of navigation fixes for the same time range as the bottom track velocity output. This is usually already done during the navigation calculations.

3. Run the *refabsbt* program to automatically match up the bottom track elocity and navigation fix files and calculate the x and y displacements in meters for each information source.

==========
PROGRAM: refabsbt
==========

USAGE: refabsbt <control file name>
INPUT: bottom velocity file, navigation fix file, control file
OUTPUT: For each fix interval, the output line has the time of the second fix, the x and y displacements in meters between fixes, the x and y displacements in meters based on the bottom track, and the same 3 gap info items as in the *../nav/refabs* program.

SAMPLE CONTROL FILE: *refabsbt.cnt*

```
fix_file_type:  simple
reference_file:  ah18.bt       /* output from lst_btrk */
fix_file:       ah18.edf      /* navigation fix file */
output:          ah18.rbt      /* output file */
```

year_base= 1990

ensemble_length= 300
gap_tolerance= 10

FORMAT OF OUTPUT FILE FROM *refabsbt*

time	dfix_x	dfix_y	dx	dy	gau	gav	gat

time -- time of second fix for each fix interval (decimal days)
dfix_x -- x displacement of two consecutive fixes (meters)
dfix_y -- y displacement of two consecutive fixes (meters)
dx -- x displacement based on bottom track (meters)
dy -- y displacement based on bottom track (meters)
gau -- gap accumulative velocity U (m/s)
gav -- gap accumulative velocity V (m/s)
gat -- time of gap accumulative

4. Use MATLAB with the M-files *runbtcal.m, btcal.m,* and *worst.m* to generate plots and solve for amp and alpha in the above correction equation.

a. Before starting MATLAB, edit the following lines of the M-file *runbtcal.m*:
fname = 'h18_gps1.btr'; % refabsbt output file
froot = 'h18_gps1'; % fname without the extension
label = 'HOT 18' % label to be used as title of the plot

b. Start up MATLAB.
c. Assign the value 1 to the variable "first".
d. Invoke *runbtcal.m.* The script in turn calls the function *btcal.m* to solve for amp and alpha from the least-squares fit of the bottom track and satellite fix displacements. The amp and alpha estimates are written out to a text file called *btcal.out*. The function also plots the residuals and saves the plot in a met-file.
e. Examine the plot of residuals. In general, we want them to be within +/- 50 m.
f. If the plot residuals are unsatisfactory, call *runbtcal.m* again in order to knock out the worst of any remaining outliers, and redo the regression estimates and plot the new residuals. The results are appended to the file *btcal.out*.
g. Repeat steps (e) and (f) for as many times as needed. This will yield the final estimate of the amplitude and misalignment angle for the time range under consideration.

****

Rotation des données

(d'après E. Firing et J. Ranada.)

I'll give some background and then append the more detailed documentation on the rotate program. Basically, there were two major changes to the old *rotate* program. The first was the facility to vary the rotation parameters from profile to profile (vs. from block to block). This required putting the *scale_factor* (amplitude) and *hd_misalign* (angle) values in the ANCILLARY_2 variable instead of the CONFIGURATION_1 variable--hence the need to run *convadc2* program before using the new *rotate* program. The second major change was to provide more flexibility in specifying the rotation parameters. Under the old *rotate,* the angle and amplitude were specified as two scalars. We have encountered cases, however, where the misalignment angle is correlated to the ship's heading. That is where the *angle_0, angle_sin_H* and *angle_cos_H* parameters come in. Assuming this is the first time to run the *rotate* program (so that the *hd_misalign* is set to 1.0) then the misalignment angle is more generally expressed as:

angle_0 + *angle_sin_H* * sin(ship heading) + *angle_cos_H* * cos(ship heading)

where the ship heading is taken from the *ANCILLARY_2.last_heading* variable. If the misalignment angle is simply a constant, then all you have to specify is the *angle_0* value; the *angle_sin_H* and *angle_cos_H* will then default to zero. I hope that clears things up. I am appending the documentation for the *rotate* program to provide more detailed explanations...

FILE: ROTATE.CNT
USAGE: ROTATE [control-file-name]

This program is used to correct the ADCP water and/or bottom track velocities in a CODAS database for any misalignment between the transducer and the ship's keel. The control file specifies the following:

DB_NAME:
LOG_FILE:
*_RANGE: (where * = BLOCK or TIME or DAY)
OPTION_LIST:
*_RANGE:
OPTION_LIST:
.
.

The DB_NAME specifies the name of the CODAS database (include a path if not in the current directory).

The LOG_NAME specified the name to be used for the log file.

The *_RANGE specifies which range of profiles in the database to apply the correction to as indicated in the immediately succeeding OPTION_LIST. The range may be specified by block numbers (BLOCK_RANGE), profile times (TIME_RANGE), or decimal days (DAY_RANGE). Examples:

BLOCK_RANGE: 0 to 5
TIME_RANGE: 90/01/23 12:20:00 to 90/01/31 01:25:00
DAY_RANGE: 12.5 to 20.2

If all the profiles in the database will be corrected in the same way, type any one of the following:

BLOCK_RANGE: all
TIME_RANGE: all
DAY_RANGE: all

The OPTION_LIST allows the user to specify which velocity track(s) will be corrected and how.

OPTION_LIST:

water_and_bottom_track:
water_track:
bottom_track:

If the correction to be applied is identical for both water and bottom track velocities, then select *water_and_bottom_track*. Otherwise, select *water_track* and/or *bottom_track*. The enumeration should be terminated by the keyword '*end*'.

Finally, for each _TRACK selected, one can specify one or more of the following:

---

angle_0=
angle_sin_H=
angle_cos_H=
amplitude=
store_only!
rotate_only!

The angle_0=, angle_sin_H=, angle_cos_H=, and amplitude= options allows the user to specify the correction parameters to be used in the following rotation equation:

corrected u = A * [u * cos(alpha) - v * sin(alpha)]
corrected v = A * [u * sin(alpha) + v * cos(alpha)]

where

A=amplitude*scale_factor (for that _TRACK in ANCILLARY_2 structure)
alpha= (angle + hd_misalign for that _TRACK in ANCILLARY_2 structure)
* PI / 180.0
angle=angle_0 + angle_sin_H * sin(last_heading in radians)+angle_cos_H* cos(last_heading in radians)
u = original U velocity
v = original V velocity

On an unrotated database loaded with loadping, the *scale_factor* and *hd_misalign* are initialized to 1 and 0, respectively.

Unless otherwise specified, the following default values are used :

angle_0= 0.0
angle_sin_H= 0.0
angle_cos_H= 0.0
amplitude= 1.0

In general, the correction to the database consists of rotating the indicated _TRACK velocities and storing the net values of the angle and amplitude used for rotation in the ANCILLARY_2 structure.

If, for some reason, the user wishes to just store the angle/amplitude values without rotating the velocities at the same time, then the *store_only* option should be specified. In this case, the angle and amplitude should NOT be given relative to the current values in the ANCILLARY_2 structure; rather, they should be the desired values themselves. Such a situation would arise in the instance of databases loaded with older versions of *loadping* (before 89/01/03) which did not initialize *hd_misalign* and *scale_factor* to 0 and 1, respectively.

If, for some reason, the user specifies the *rotate_only* option, this has the effect of using the angle/amplitude values already in the ANCILLARY_2 structure to perform the rotation on the velocities.

If neither *store_only* nor *rotate_only* is specified, the program defaults to the typical case of both storing and rotating.

Again, this list of options must be terminated by another '*end*'.

Before doing anything, backup your database as a precaution since all 3 following programs do their job "in place". Now first run the *convadcp* program just to see if all your structures are up-to-date (it will update them otherwise):

% convadcp <dbname>

Then you need to convert the rotate parameters to be profile-varying by typing:

% convadc2 <dbname>

Finally, create the rotate control file (*rotate.cnt*) to specify the rotate parameters, database name, etc. There is more detailed documentation in that file. When your *rotate.cnt* file is correct, you may want to make another backup of your database in case you change your mind or misspecify the rotate parameters somehow...

(I haven't had a chance to work on an "*unrotate*" option yet). Finally, run the *rotate* program:

% rotate rotate.cnt

I hope everything works out! Following is the sample rotate.cnt file.

FILE: ROTATE.CNT

```
DB_NAME:
LOG_FILE:
{choose one of the next 3}
BLOCK_RANGE:     {choose one or the other:}
TIME_RANGE:        all
DAY_RANGE:         <start> to <end>

OPTION_LIST:  {choose one or more of the ff.:}
    water_track:  {choose one or more of the ff.:}
            angle_0=
            angle_sin_H=
            angle_cos_H=
            amplitude=
            rotate_only!  {otherwise, above values will be stored}
            store_only!  {otherwise, velocities will be rotated}
    end
    bottom_track:
            angle_0=
            angle_sin_H=
            angle_cos_H=
            amplitude=
            rotate_only!  {otherwise, above values will be stored}
            store_only!  {otherwise, velocities will be rotated}
    end
    water_and_bottom_track:
            angle_0=
            angle_sin_H=
            angle_cos_H=
            amplitude=
```

```
                rotate_only! {otherwise, above values will be stored}
                store_only!  {otherwise, velocities will be rotated}
        end
    end

*/
DB_NAME: ../adcpdb/a901
LOG_FILE: a901_rot.log
TIME_RANGE: all
OPTION_LIST:
  water_and_bottom_track:
    amplitude= 1.01
    angle_0= 4.0
  end
end
```

****

Integration de la navigation

(d'après E. Firing)

This document describes a procedure for calculating absolute ship speed and position along a cruise track from position fixes and ADCP measurements of the ship's speed relative to a reference layer. The results may then be stored in the CODAS ADCP database. Of the following programs, all are found in *codas3/adcp/nav* except for adcpsect, which is in *codas3/adcp/adcpsect*. The following gives limited information about these programs as required for navigation calculations; additional information is available in the source files.

==========
1. adcpsect
==========

USAGE: adcpsect [control file name]
INPUT: CODAS database, control file
OUTPUT: 1. A file (with the extension *.nav*) containing profiles time and the velocity of the ship relative to a specified reference layer.

2. A file (with the extension *.cor*) containing profiles times; the amplitude and phase (degrees) of the complex correlation between two consecutiveprofiles, each considered as a vector of complex numbers (u+iv); and the time interval and number of good bins used in the calculation.

3. A file (with the extension *.sta*) containing some statistics of the velocity components.

4. A file (with the extension *.muv*) containing velocities (*U* and *V)* in MATLAB format.

5. A file (with the extension *.mxy*) containing other information in MATLAB format (will be described later).

adcpsect is a multifunctional program used in both the processing and the analysis of ADCP data in a CODAS database. Its functions depend on options and parameters specified in the control file. *adcpsect* always produces files 3), 4) and 5), which are of little interest for the navigation calculations considered here. The *.nav* file is a primary input for the navigation calculation. The *.cor* file can be helpful in checking for major problems in the data associated with the measurement of heading and the vector averaging of individual profiles into ensembles; it is not ordinarily required, and may be omitted.

EXAMPLE OF *adcpsect.cnt*:

```
dbname: wep3          /* database name */
output: wep3leg1       /* output file name */
step_size: 1          /* number of profiles to advance */
ndepth:   41          /* number of depth bins to read and process */
```

```
time_ranges: separate   /* combined or separate */
year_base= 1988        /* base year for calculation of decimal day form of time
*/
option list:                /* begin main option list */
pg_min= 30                 /* minimum percent good */
   navigation:              /* generate wep3leg1nav file */
     reference_bins 5 to 20 /* ref. layer bin range, inclusive; first  bin is 1 */
   end                    /* end list of suboptions  within "navigation" */
   statistics:            /* generate wep3leg1.sta */
     mean
     units= 0.01       /* cm/s instead of default m/s */
   end                     /* end list of suboptions  within "statistics" main option
*/
            /* you will get statistics by default even if you omit this option */
   flag_mask:
     ALL_BITS          /* ignore bins flagged as bad for all reasons */
     end
end                 /* terminates the main option list */

88/6/18 1:10:00 to 88/7/2 5:19:0    /* time range */
 /* year/month/day hour:min:sec to ... */
 /* It does not have to be exact, so long  as it includes the range of interest. */
```

FORMAT OF OUTPUT FILES:

1. *.nav* file:

time	*ship-ref_U*	*ship-ref_V*

time -- profile time in decimal days (days since the beginning of the year; dd 0.5 is noon on January 1, if the base year is the current year.)
ship-ref_U and *_V* -- the east and north components of the ship's velocity relative to the reference layer.

2. *.cor* file

time	*amp*	*cor_angle*	*time_interval*	*ngood*

time -- profile time in decimal days
amp -- velocity vector amplitude between two consecutive profiles
cor_angle -- correlation angle between two consecutive profiles
time_interval -- time interval between two consecutive profiles
ngood -- number of good bins used to do the calculation

3. *.sta* file | The format of these output files will be explained
4. *.muv* file | later when we need them. For detail user can refer

5. *.mxy* file | to the documentation for adcpsect.c

WHAT TO DO:

For the navigation calculation, one must have the velocity of the ship relative to the reference layer in geographical coordinates. This is the usual mode of data acquisition.

Possible exceptions are velocities recorded during the period of incorrect heading bias or any malfunction in the ADCP heading data recording. To find these kinds of velocities, the user can look up the log files (note books) where the time range of calibration run and bad heading bias setting may be recorded. Check these time ranges and compare with the *.cor* file to see the changes of correlation angle and the amplitude. If the correlation angle changes drastically but the amplitude is close to 1, then this may be the beginning of the bad heading bias setting period. If it is true, user should find the end of the bad heading bias setting that is drastic change of the correlation angle in the opposite direction and amplitude is close to 1. The *.cor* file is a crude tool which will catch only errors of about 10 degrees or more. Smaller but still serious errors are lost in the scatter due to natural variability from one profile to the next. Any questionable sections of the *.nav* file can be eliminated by deleting the lines, or by inserting the percent character (%) at the beginning of each bad line. The latter is usually recommended, since it leaves in the file a record of the editing done. Comments can also be added so long as each line is preceeded by the %.

=========
3. refabs
=========

USAGE: refabs [control file name]
INPUT: edited fix file, edited .nav output file from adcpsect
OUTPUT: A file containing navigation fix time, the raw absolute velocity of the reference layer velocity averaged between navigation fixes.

refabs calculates the raw absolute reference layer velocity averaged between navigation fixes.

ABOUT NAVIGATION FIX FILE:

The refabs program recognizes the following fix file formats:

(1) Simple format 1 (output from scanping ot ubprint)

example: (PRC5 cruise)

time	Lon	Lat	el	it	dist	f	dt
330.2838657	143.7972139	13.7135972	17	2	0.19	1	-91
330.3203356	143.5607083	13.7964222	63	2	0.62	1	-92
330.3419329	143.4221556	13.8479861	5	2	0.68	0	-91
330.3485185	143.3726750	13.8566778	64	2	0.08	1	-91
330.4156134	142.8609944	13.9945417	85	3	3.55	0	-91
330.4413889	142.7304694	14.0525000	21	2	1.17	1	-91
330.4823264	142.4513889	14.1291250	7	2	0.34	1	-91

Only the first three columns are used; *refabs* ignores the remaining columns, so they can contain anything. Above example is from *scanping* user buffer output file or *TRANSIT_summary* option with *ubprint*. For detail user can refer to documentation scanload.doc. Recall that time is in decimal days, and that the decimal day is generally a fraction of a day less than the corresponding Julian day; round the DD up to get JD.

(2) Simple format 2 (output from ubprint)

time	Lon	Lat	ns	q	hd	nd	ed	vd	al
6.3315972	-158.3274937	21.2758613	4	3	2	1	1	6	0
6.3350579	-158.3274937	21.2766981	4	3	2	1	1	5	0
6.3385417	-158.3277297	21.2774920	4	3	2	1	1	5	0
6.3420023	-158.3276868	21.2783289	4	3	2	1	1	5	0
6.3454861	-158.3276010	21.2791443	4	3	2	1	1	5	0
6.3489583	-158.3273435	21.2797022	4	3	2	2	1	5	0
6.3524306	-158.3270431	21.2803674	4	3	2	2	1	5	0

where ns = number of satellites, q = quality, hd = hdop, nd = Ndop, ed = Edop, vd = Vdop, al = altitude.

Only the first three columns are used; *refabs* ignores the remaining columns, so they can contain anything. Above example is from *GPS_summary* option with *ubprint*.

Fix files, like nav files, can be edited by preceding comment lines with a percent.

EXAMPLE OF *refabs.cnt*

```
fix_file_type:  HIG        /* or SIMPLE, the format of fix file */
reference_file:  wep3leg1.nav  /* edited output file from adcpsect */
fix_file:       wep3fix.acs   /* fix file name */
output:         wep3leg1.ref  /* output file name */
year_base=      1988          /* base year */

ensemble_length= 300  /* ensemble length in seconds */
gap_tolerance=   10   /* gap tolerances in second, usually 10 */
```

NOTE: The lower the gap_tolerance, the more gaps will be recognized, thereby generating more gap statistics.

FORMAT OF OUTPUT FILE FROM *refabs*

```
-------------------------------------------------------
   time lon  lat  abs_u  abs_v  dfix  gau  gav  gat
-------------------------------------------------------
```

time -- fix time in decimal days
lon -- longitude

lat -- latitude
abs_u -- absolute reference layer velocity U in m/s
abs_v -- absolute reference layer velocity V in m/s
dfix -- time difference between two consecutive fix in min/day
gau -- gap accumulative velocity U m/s
gav -- gap accumulative velocity V m/s
gat -- time of gap accumulative

WHAT TO DO:

At this step, user can use a MATLAB M-file to plot the absolute reference velocity vs. fix time. The purpose of doing this is to find and eliminate bad fixes from the fix file.We will discuss how to do this in section 5: callrefp.m & refplot.m

==========
4. smoothr
==========

USAGE: smoothr [control file name]
INPUT: 1) output of *refabs*: absolute ref layer velocities averaged over fix intervals, with gap information
2) output of *adcpsect*, navigation option: time, and ship relative to reference layer for each profile (the same *.nav* file that was used as input to *refabs*)

OUTPUT: 1) absolute ref layer velocity, interpolated and smoothed to the profile times.
2) absolute ship velocity and position at profile times.
3) a log file with lots of information about the fixes and the navigation calculation.

EXAMPLE OF *smoothr.cnt*

reference_file: sectw3l1.nav
refabs_output: rfabw3l1.ref
output: smthw3l1.o3
filter_hwidth= 0.125 /* half width in days; for example, 0.125 days is 3 hours half width, or 6 hours total width */
min_filter_fraction= 0.05 /* Minimum fraction of filter interpolation. 0.03 to 0.05 seems about right, but this is hard to prove */
max_gap_distance= 500 /* m; only for position integration. If you have lots of fixes you may want to pick a smaller number */
max_gap_time= 3000 /* seconds; only for position integration. As above, you may want a smaller number. */
ensemble_time= 300 /* seconds, ensemble length*/
max_speed= 5.0 /* m/s cruising speed */
min_speed= 1.0 /* m/s variability; 1 is reasonable */
iterations= 2 /* just leave it at 2 */

fix_to_dr_limit= 0.00100 /* (in degrees) roughly 100 m; make it zero if you want to know how far each fix is from the calculated cruise track */

FORMAT OF OUTPUT FILE FROM *smoothr:*

```
-------------------------------------------------------
  time abs_u  abs_v  ship_u  ship_v  lon  lat  filte_para
-------------------------------------------------------
```

WHAT TO DO:

At this step, the user can use the MATLAB M-file to plot the smoothed absolute reference velocity vs. profile time. And it also can be plotted comparing with the unsmoothed the absolute velocity. We will discuss how to do this in detail in later section.

==========
4. ref2mat & sm2mat
==========

(a) ref2mat:
USAGE: ref2mat input-file output-file start-time end-time [ncol]
input-file -- output of refabs
output-file -- an MATLAB format of the input file
start-time -- beginning time boundary
end-time -- ending time boundary
ncol -- optional, number of columns user like to retrive, default is 9. (see FORMAT OF *refabs* OUTPUT.)

(b) sm2mat:
USAGE: sm2mat input-file output-file start-time end-time [step] [ncol]
input-file -- *.bin file from output of smoothr
output-file -- a MATLAB format of the input file
start-time -- beginning time boundary
end-time -- ending time boundary
step -- optional, number of profiles to jump, defualt 1.
ncol -- optional, number of columns user like to retrive, defalut is 9. (see FORMAT OF *refabs* OUTPUT.)

Both programs convert the input-file to a Matlab *.mat* file:
ref2mat converts the output of *refabs* and *sm2mat* converts the output of *smoothr*. The user does not have to invoke the programs directly; instead the following MATLAB routines have been provided.

==========
6. callrefp.m & refplot.m
==========

Ordinarily, the user will use only *callrefp*, and need not bother running *refplot* directly. However, note that some parameters must be set first by editing *refplot.m*. It is easy to write a simple script M-file that invokes *callrefp* in a loop, incrementing the starting time, so as to plot the reference layer velocity, smoothed ref layer, and position for an entire cruise in (e.g.) 2-day increments. There is a bug in MATLAB (some versions, at least) such that memory can be progressively consumed and not released; it is sometimes necessary to run the loop for 10 iterations, exit MATLAB, reload MATLAB, and run the remaining iterations.

(a) callrefp.m
USAGE: callrefp
callrefp(<start-time>)

callrefp.m is a MATLAB function file for setting input parameters to run *ref2mat* and/or *sm2mat*. It then calls *refplot.m* with appropriate input parameters to plot absolute reference layer velocity (unsmoothed and/or smoothed). If start time is not given as an input parameter, it should be specified within the file *callrefp.m*. Other parameters that must be specified in the file are described in *callrefp.m*.

(b) refplot.m
USAGE: refplot(year,metname,ax,RefMatName,SmMatName)

refplot function call is set in *callrefp.m*. To call this function without using *callrefp.m*, set following input parameters.
year -- base year
metname -- matlab output plot file name
ax -- [t0,dt,y0,dy]
RefMatName & *SmMatName* -- names of matlab data file which contain variable 'ref' and 'sm', respectively. (set *RefMatName* = '' or *SmMatName* = '' if user does not use 'ref' or 'sm' file.)

Following variables must be initialized in the refplot.m:
mratio -- max_gap_ratio (as set in smoothr.cnt)
minvel -- min_speed (as set in smoothr.cnt)
maxvel -- max_speed (as set in smoothr.cnt)
llmarg -- degrees; plot margine for lon. and lat vs. time
pauseflag -- set to 1 if user wish to pause after each plot; set to 0 otherwise.

callrefp is invaluable in the navigation calculation. The usual sequence is something like this: run *refabs*, visually scan the output for bad fixes, edit out the offending fixes, run *refabs* again, run *smoothr*, plot everything with *callrefp*, edit the fix file again, run *refabs*, run *smoothr*, and plot with *callrefp*. If one is satisfied with the result the job is finished, and the refplot output provides a permanent record of the navigation. The *smoothr* output can then be transferred to the CODAS database using *putnav*.

=========
7. putnav
=========

USAGE: putnav [control file name]
INPUT: output file from smoothr
OUTPUT: updated ADCP database with the ship speed stored in *ACCESS_VARIABLES* structure, and position stored in the profile directory.

This is part of the ADCP navigation processing system. It takes a file with the final estimate of position and ship velocity for each profile, specified by time, and puts that information into the database. Latitude and longitude go into the directory, and ship's velocity goes into the *ACCESS_VARIABLES* structure. The data processing mask is set to indicate the navigation source.

EXAMPLE OF putnav.cnt

```
dbname:       wep3
position_file: smthw3l1.o3
year_base=    1988
tolerance=    5     /* seconds; this is just to allow for truncation error when
times are converted from YMDHMS to DD and back. */
navigation_sources:    /* list those that were used; currently recognized are:
gps transit loran omega radar  /*

     gps
end            /* end of list of navigation sources */
```

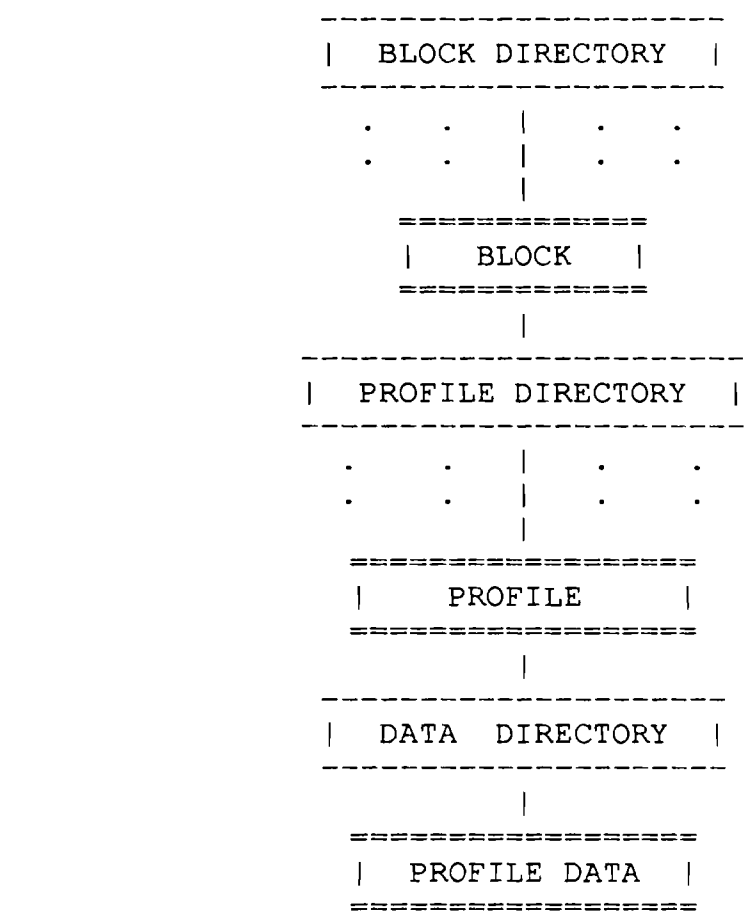
****

Variables et routines de CODAS3

(d'après E. Firing)

The working database contains data (ADCP, CTD, etc.) currently being worked on. The system is composed of a set of data blocks and a corresponding directory. Both the directory and the blocks reside on disk as stream files. The data are accessed through a set of standard access routines written in C which can be called from either C or FORTRAN programs.

Conceptual organization:



DATA TYPES IN CODAS DATABASE:

Every datum (data structure) contained in the database is identified by a numeric code. The following codes have been established:

code	data	DBADD	DBGET	DBPUT
0 - 99	USER DEFINED DATA	YES	YES	YES
100	DATASET ID	NO	YES	NO
101	PRODUCER ID	NO	YES	NO
102	PROFILE TIME	NO	YES	NO
103	PROFILE POSITION	YES	YES	YES
104	BLOCK DATA MASK	NO	YES	NO
105	BLOCK DATA PROCESSING MASK	YES	YES	YES
106	DEPTH RANGE	YES	YES	YES
200	DATABASE VERSION	NO	YES	NO
201	PRODUCER HOST	NO	YES	NO
202	DATABASE NUMBER	NO	YES	NO
203	BLOCK/PROFILE NUMBER	NO	YES	NO
300	BLOCK DIRECTORY FILE	NO	YES	NO
301	CURRENT BLOCK FILE	NO	YES	NO
302	PRODUCER DEFINITION FILE	NO	NO	NO
303	TEMPORARY DATA FILE	NO	NO	NO
304	BLOCK DIRECTORY HEADER	NO	YES	NO
305	BLOCK DIRECTORY	NO	NO	NO
306	BLOCK HEADER	NO	YES	NO
307	DATA LIST	NO	NO	NO
308	PROFILE DIRECTORY	NO	NO	NO
309	DATA DIRECTORY	NO	NO	NO
310	PROFILE DATA	NO	NO	NO
311	PRODUCER DEFINITION	NO	NO	NO
312	STRUCTURE DEFINITION	NO	YES	YES
313	STRUCTURE FORMATS	NO	NO	NO

BAD/MISSING DATA:

A bad or missing datum is represented by the maximum value of its corresponding data type:

for BYTE -- 127
UBYTE -- 255
SHORT -- 32767
USHORT -- 65535 (-1 in FORTRAN)
LONG -- 2^31 - 1
ULONG -- 2^32 - 1 (-1 in FORTRAN)
FLOAT -- roughly 10^38
DOUBLE -- roughly 10^38

FORTRAN REPRESENTATION OF DATA:

1 - TIME DATA:
Time is expressed as a 6-element array of INTEGER*2.
INTEGER*2 TIME(6)
Where: TIME(1) — year (whole year not only the last 2 digits)
TIME(2) — month
TIME(3) — day
TIME(4) — hour
TIME(5) — minute
TIME(6) — second

2 - POSITION DATA
Position is expressed as an 8-element array of INTEGER*2.
INTEGER*2 LATLON(8)
Where: LATLON(1) -- longitude degrees
LATLON(2) -- longitude minutes
LATLON(3) -- longitude seconds
LATLON(4) -- longitude hundredths of a second
LATLON(5) -- latitude degrees (if < 0 South)
LATLON(6) -- latitude minutes
LATLON(7) -- latitude seconds
LATLON(8) -- latitude hundredths of a second

3 - USER DEFINED DATA
The producer definition file contains a description of these data. Just remember the following equivalences:

BYTE	— is the same as	BYTE
CHAR	— " " " "	CHARACTER*1
SHORT	— " " " "	INTEGER*2
LONG	— " " " "	INTEGER*4
FLOAT	— " " " "	REAL*4
DOUBLE	— " " " "	REAL*8
TEXT	— " " " "	CHARACTER*1

For UBYTE, USHORT, and ULONG, it is easier and safer to use a REAL*4 and invoke the scaling functions to convert to and from the unsigned C data types.

4 - NAMES AND FILE NAMES
When calling database routines that require string arguments (e.g., DBOPEN and DBCREATE), the character array containing the string argument should be passed by reference.

5 - ALL OTHER PARAMETERS
Any other parameter to a database routine should be of type INTEGER. By default they will be INTEGER*4 in the VAX and INTEGER*2 in PC's.

ERROR CODES:

Error codes returned by database access routines are of two types: User errors and Database errors. User errors are given as a positive number. The section captioned USER ERROR gives a description of these errors. Database errors (internal problems in database) are given by a negative number containing two pieces of information:

(-IERR) / 1000 gives the database internal error code. See section DATABASE INTERNAL ERRORS.

(-IERR) mod 1000 gives the data structure being accessed when failure occurred. See the section on DATA TYPES IN ADCP DATABASE.

When Database errors occur, report the value of IERR to the database manager. Most likely you don't have privilege for accessing database files, or you gave the wrong database name, or your computer has insufficient memory.

USER ERROR:

code description

0 - OK:
Operation was completed successfully.
1 - INVALID_DATABASE_NUMBER:
A database number outside the range 1 - 5 was specified in the call to DBOPEN, DBCREATE or DBSET.
2 - DB_IS_NOT_OPEN:
Attempt to access a database that has not been opened.
3 - DB_ALREADY_OPEN
Attempt to open/create a database that is already open.
4 - NO_OPEN_FOR_WRITE
Attempt to modify (DBPUT) a database that is open for read only.
5 - NO_OPEN_FOR_CREATE
Attempt to add data to a database not opened by DBCREATE.
6 - OPEN_FOR_CREATE
Attempt to access a database opened by DBCREATE.
7 - DATABASE_IS_EMPTY
Database contains no data blocks.
8 - INCORRECT_DATABASE_VERSION
Database was created using a different (2. or older) version of CODAS.
9 - UNEXPECTED_STRUCTURE_SIZE
Differences in machine alignment make it impossible to run CODAS.
11 - NO_BLOCK_IS_OPEN
Attempt to access block data when no block has been located.
12 - NO_PROFILE_IS_OPEN
Attempt to access profile data when no profile has been located.
13 - NO_NEW_BLOCK_IS_OPEN
Attempt to add block/profile data before calling DBNEWBLK.

14 - NO_NEW_PROFILE_IS_OPEN
Attempt to add profile data before calling DBNEWPRF.
15 - NEW_BLOCK_IS_OPEN
Block must be terminated by DBENDBLK before attempted operation.
16 - NEW_PROFILE_IS_OPEN
Profile must be terminated by DBENDPRF before attempted operation.
21 - BLOCK_DIR_IS_FULL
No more blocks can be added. Database should be closed by DBCLOSE.
22 - BLOCK_IS_FULL
No more profiles can be added to current block. Block should be closed by DBENDBLK and subsequent profiles added to a new block.
23 - INSUFFICIENT_SPACE
Not enough space was reserved during database creation to DBPUT so many bytes.
31 - INVALID_TYPE_REQUEST
An invalid type was specified in call to DBSRCH, DBADD, DBGET or DBPUT.
32 - INVALID_TIME
An invalid time was used in call to DBNEWPRF or DBSRCH.
33 - INVALID_POSITION
An invalid time was used in call to DBADD, DBGET or DBPUT.
34 - INVALID_VALUE_TYPE
An invalid value type was specified in producer definition.
35 - INVALID_ACCESS_TYPE
An invalid access type was specified in producer definition.
36 - NO_SUCH_BLOCK_PROFILE
A non-existing block/profile number was specified for DBSRCH.
37 - INVALID_BUFFER_SIZE
The size of the user array to hold user-defined data retrieved from the database is less than zero.
41 - SEARCH_BEYOND_END
Attempt to search/move beyond end of database.
42 - SEARCH_BEFORE_BEGINNING
Attempt to search/move before beginning of database.
51 - ZERO_SCALE
Attempt to divide by a zero scale factor.
61 - PATHNAME_TOO_LONG
Length of database pathname exceeds 31-character maximum.
62 - DBNAME_TOO_LONG
Length of database name exceeds maximum defined by MAX_DBNAME_LENGTH in DBEXT.H (system-dependent parameter).
70 - UNDEFINED STRUCTURE
Attempt to display structure with no definition and number of bytes unknown; attempt to convert structure with no structure definition. Provide an external structure definition file containing the structure's definition.
71 - UNKNOWN_HOST_ERROR
Attempt to access database created by unrecognized machine.

72 - INCOMPATIBLE_HOST_ERROR
Attempt to access database created by different machine. Use MKBLKDIR.EXE to convert database blocks across machines.
73 - STRUCTURE_NOT_PRESENT
Attempt to DBPUT a structure definition for a structure that had not been defined at the time of database creation.
80 - FORMAT_ERROR
Structure format supplied does not match structure definition. Check element names for exact spelling, etc.

DATABASE INTERNAL ERRORS:

code description

-2 - UNABLE_TO_OPEN
A file could not be opened because it does not exist or due to insufficient privilege.
-3 - FILE_DOES_NOT_EXIST
-4 - FILE_ALREADY_EXISTS
Attempt to create a file that already exists.
-5 - FILE_PROTECTION_ERROR
Attempt to access/create a file without sufficient privilege.
-6 - READ_ERROR
Error reading file.
-7 - WRITE_ERROR
Error writing file.
-8 - SEEK_ERROR
Error positioning in file.
-9 - CLOSE_ERROR
Error closing file.
-10 - INSUFFICIENT_MEMORY
Not enough memory for attempted operation.

MISSING/BAD DATA:

Missing or bad data are stored as:

for BYTE	datum is bad if	= 127
for UBYTE	datum is bad if	= 255
for SHORT	datum is bad if	= 32767
for USHORT	datum is bad if	= 65535
for LONG	datum is bad if	= 2,147,483,647
for ULONG	datum is bad if	= 4,294,967,295
for FLOAT	datum is bad if	= 1.0e37
for DOUBLE	datum is bad if	= 1.0e37

——— DATABASE ACCESS ROUTINES ———

The following subroutines can be used from within a FORTRAN/C program for accessing the CODAS database library:

void DBOPEN(int *IDBID, char *DBNAME, int *IACCMOD, int *IMEMMOD, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBOPEN (IDBID, %REF(DBNAME), IACCMOD, IMEMMOD, IERR)

It opens an existing database for READ/MODIFY access. It loads the database directories and sets the access data structures. After the call, the database pointer is positioned at the first profile of the first block (block 0, profile 0). This subroutine must be the first one called in order to access an existing database.

Arguments:

Input:
IDBID - The database number (1 - 5).
DBNAME - Character string with the name of the database block directory file. The name can be up to 40 characters long. There must be at least 1 space following the name and no spaces before it. The string may be passed as a constant (i.e. 'WEP '), a CHARACTER variable or in a numeric array (i.e., read with format A). The compiler option %REF is needed for constant or CHARACTER strings. For systems with tree-directories like DOS or VAX/VMS you should give the path as part of the name.
IMEMMOD - If 0, the database will not load directories into memory; otherwise block and profile directories will be kept in memory. 0 should be used when memory is limited (i.e., in PCs) at the expense of a slower access time.
IACCMOD - If 0, the database will be open for READ_ONLY; otherwise it will be open for READ_WRITE.

Output:
IERR - Resulting error code.

void DBSET(int *IDBID, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBSET (IDBID, IERR)

It makes database IDBID the current database. This database must already be open.

Arguments:

Input:
IDBID - Number of new current database.

Output:
IERR - Resulting error code.

void DBCLOSE(int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBCLOSE (IERR)

It closes the database.
Arguments:

Output:
IERR - Resulting error code.

void DBSRCH(int *ITYPE, char *PARAMS, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBSRCH (ITYPE, PARAMS, IERR)

It positions the database at a specific profile (random access).
Arguments:

Input:
ITYPE - Type of search. As of now, only search by time of profile (ITYPE = 1) and search by block/profile number (ITYPE = 2) are implemented.
PARAMS - Search parameters.
For ITYPE = 1, PARAMS is a time/date array. In this case DBSRCH will position the database at the first profile whose time is >= than the time in PARAMS.
For ITYPE = 2, PARAMS is 2-element integer array with the block and profile numbers. In this case, DBSRCH will position the database at this block/profile if it exists.

Output:
IERR - Resulting error code.

Notes: If a time search goes before the first time in the database, this subroutine will leave the database pointer positioned at the first profile. If a time search goes beyond the last time in the database, the database pointer will be left at the last profile. Search for a non-existent block/profile will leave the database pointer at the profile at which it was positioned prior to the search.

void DBMOVE(int *ISTEPS, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBMOVE (ISTEPS, IERR)

It positions the database at ISTEPS profiles from the current profile.
Arguments:

Input:
ISTEPS - Number of profiles to advance.
ex. ISTEPS = 1 advances to next profile (in time)
ex. ISTEPS = 12 advances 12 profiles
ex. ISTEPS = -7 goes back 7 profiles

Output:
IERR - Resulting error code.

Notes: As with DBSRCH, moves beyond either end of the database will leave the database pointer at the first or last profile.

void DBGET(int *ITYPE, char *DATA, unsigned int *N, int *IERR)
void DBGET_F(int *ITYPE, float *DATA, unsigned int *N, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBGET (ITYPE, DATA, N, IERR)
SUBROUTINE DBGET_F (ITYPE, DATA, N, IERR)

These retrieve specific data from the database. DBGET_F also provides automatic conversion/rescaling of the data to floating point values.
Arguments:

Input:
ITYPE - Data type.
DATA - Name of array/scalar variable where requested data will be stored. You are responsible for declaring enough elements in the case of arrays.
N - Number of bytes to retrieve, in the case of DBGET
Number of floating point values to retrieve, for DBGET_F

Output:
IERR - Resulting error code.

void DBPUT(int *ITYPE, char *DATA, unsigned int *N, int *IERR)
void DBPUT_F(int *ITYPE, float *DATA, unsigned int *N, unsigned int *NBAD, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBPUT (ITYPE, DATA, N, IERR)
SUBROUTINE DBPUT_F (ITYPE, DATA, N, NBAD, IERR)

These rewrite specific data to the database. DBPUT_F also provides automatic conversion/rescaling of floating point values to the data types used in the database.
Arguments:

Input:
ITYPE - Data type.
DATA - Name of array where requested data is stored.
N - Number of bytes to rewrite, in the case of DBPUT

Number of floating point values to rewrite, for DBPUT_F

Output:
NBAD - Number of values set to BAD due to conversion/rescaling to values outside range of database data type.
IERR - Resulting error code.

Notes: You must be careful when using these subroutines. If profile data is going to be modified, you should make sure that the database is positioned at the right profile.

Ex: If you want to de-mean U profiles you would proceed as follows:
1 - Lock on a profile by calling DBSRCH and/or DBMOVE.
2 - Retrieve U by calling DBGET.
3 - De-mean U in your program.
4 - Re-store U by calling DBPUT.
If you used DBSRCH or DBMOVE in between steps 2 and 4 you should re-locate the original profile before step 4.

void DBLOADSD(char *FILENAME, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBLOADSD (%REF(FILENAME), IERR)

This loads structure definitions from an external file. See the section below on EXTERNAL STRUCTURE DEFINITION FILE.

Arguments:

Input:
FILENAME - Name of structure definition file to be loaded.

Output:
IERR - Resulting error code.

void DBPRTSTR(char *DATA, char *NAME, unsigned int *NBYTES)
for VAX/VMS FORTRAN:
SUBROUTINE DBPRTSTR(DATA, %REF(NAME), NBYTES)

This is used to print structures or arrays of structures. The <NAME> is used to locate the structure definition in order to print the structure elements in readable fashion; if the structure definition is not located, the structure is printed as an array of unsigned bytes.

Arguments:

Input:
DATA - address of the structure buffer
NAME - name of structure
NBYTES - pointer to number of bytes in data buffer

Output:

Structure is displayed on screen.

int DBERROR(int *IERR, char *MESSAGE)
for VAX/VMS FORTRAN:
FUNCTION DBERROR(IERR, %REF(MESSAGE))

This is used to check the error code <IERR> after a preceding database function call. If an error is detected, it prints a brief description of the error, and any accompanying <MESSAGE>.

Arguments:

Input:
IERR - address of the error code
MESSAGE - any message to be printed in case of error: NULL, if none desired

Output:
Returns the error code (0 if none). An error message is displayed on the screen, if the error code is nonzero; no effect otherwise.

--------- DATABASE CREATION ROUTINES ----------

void DBCREATE(int *IDBID, char *DBNAME, char *PRDFILE, int *IMEMMOD, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBCREATE (IDBID, %REF(DBNAME), %REF(PRDFILE), IMEMMOD, IERR)

It creates a new database or opens an existing one for ADD/DELETE access. It loads the database directories and set the access data structures.

Arguments:

Input:
IDBID - The database number (1 - 5).
DBNAME - Character string with the name of the database block directory file.
PRDFILE - Name of file containing the producer definition. See section below on PRODUCER DEFINITION FILE.
IMEMMOD - If 0, the database will not load directories into memory; otherwise block and profile directories will be kept in memory.

Output:
IERR - Resulting error code.

void DBNEWBLK(int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBNEWBLK (IERR)

It creates a new block in the database and leaves it ready for adding blockdata.
Arguments:

Output:
IERR - Resulting error code.

void DBENDBLK(int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBENDBLK (IERR)

It terminates a new block in the database.
Arguments:

Output:
IERR - Resulting error code.

void DBNEWPRF(char *TIME, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBNEWPRF (TIME, IERR)

It creates a new profile in the current new block and leaves it ready for adding profile data.
Arguments:

Input:
TIME - Time of profile. It is a year,month,day,hour,min,sec structure (INTEGER*2 TIME(6)).

Output:
IERR - Resulting error code.

void DBENDPRF(int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBENDPRF (IERR)

It terminates a new profile in the database.
Arguments:

Output:
IERR - Resulting error code.

void DBADD(int *ITYPE, char *DATA, unsigned int *N, int *IERR)
void DBADD_F(int *ITYPE, float *DATA, unsigned int *N, unsigned int *NBAD, int *IERR)
for VAX/VMS FORTRAN:
SUBROUTINE DBADD (ITYPE, DATA, N, IERR)
SUBROUTINE DBADD_F (ITYPE, DATA, N, NBAD, IERR)

These add specific data to the database. DBADD_F also provides automatic conversion/rescaling of floating point values to the data types used in the database.

Arguments:

Input:
ITYPE - Data type.
DATA - Name of array where requested data is stored.
N - Number of bytes to add, in the case of DBADD
Number of floating point values to add, for DBADD_F

Output:
NBAD - Number of data values set to BAD due to rescaling/conversion to values outside range of data type.
IERR - Resulting error code.

The Producer Definition File

The producer definition file is a text file that is required as input whenever a CODAS database is to be created, whether from raw Ping data files (LOADPING), ASCII text files (LOAD_ASC), or from existing CODAS block files (MKBLKDIR, DB2TODB3). It provides information pertinent to the characteristics of the database to be created: the instrument used in data gathering, dataset identification, the variables to be loaded, etc. Hence, it is also an important part of the documentation for a database, and is best kept together with the database itself. The producer definition file contains both required parameters and optional parameters. The required parameters must appear first and in a fixed order. The optional parameters follow. In either case, each parameter must be preceded by the appropriate keyword and at least one space. Comments may appear anywhere, as long as they are enclosed between a /* and */. For an example of the producer definition file, see *\CODAS3\ADCP\DEMO*.DEF*.

Required parameters.

Required parameters provide essential information for identifying and classifying a given CODAS database. The following parameters, therefore, must always be present in the producer definition file in the indicated order:

Keyword	Parameter Example
DATASET_ID	ADCP-VM
PRODUCER_ID	32R2MW0001
BLOCK_DIR_TYPE	0
PROFILE_DIR_TYPE	3

Each keyword is scanned for (case-sensitive!) in this order and the following noncomment word is interpretedaccordingly.

DATASET_ID. This is a string for identifying the type of instrument used to gather data. The string has a maximum length of 31 characters (including spaces following the first non-whitespace character), where the first 8 are the instrument name, and the remainder are currently unassigned and may be used for any other qualifiers. In the example above, the instrument name is ADCP-VM, indicating an Acoustic Doppler Current Profiler (Vessel-Mounted) was used. Following are names that can be used for the indicated instrument, based on common terminology:

Name	Instrument
ADCP-VM	vessel-mounted ADCP
ADCP-BM	bottom-mounter ADCP
ADCP-MO	moored ADCP
XBT	XBT
CTD	CTD
CTD-ROS	CTD with bottle samples
ACOUS-DS	acoustic dropsonde (Pegasus, WH)
IES	inverted echo sounder
MCM	moored current meter
SSP	sub-surface pressure gauge
TIDE	tide gauge

It should be noted that any database may contain data from more than one instrument/source, such as navigational instruments, weather observations, and so forth; the DATASET_ID identifies only the primary instrument.

PRODUCER_ID. This is a string for identifying the institution responsible for the data collected. The string has a maximum length of 31 characters (including any spaces following the first non-whitespace character). The31-character string has been subdivided into the following fields:

Character Position	Encodes
1 - 2	country
3 - 4	institution
5 - 6	platform
7 - 10	instrument code
11 - 32	unassigned

The first three fields above (country, institution, and platform) use the standard NODC codes. In the example above, 32R2MW is the NODC code for the United States (32), University of Hawaii (R2), and the R/V Moana Wave (MW). (The United States has two country codes under the NODC system: 31 and 32. Either one may be used.) The fourth field, instrument code, should be based on the institution's (or principal investigator's) own coding system, to provide unique identification for each of its own instruments. Some sequential numbering system will do. Duplication of numbers across instrument types should not pose significant problems, since the instrument name in the DATASET_ID can provide differentiation.

BLOCK_DIR_TYPE. The block directory type is intended for distinguishing different types of CODAS block directories. At present, there is only one such type, designated by the integer 0.

PROFILE_DIR_TYPE. The profile directory type is intended for distinguishing different types of CODAS profile directories. Each entry in the profile directory provides values for key variables for each profile in that file; which key variables are included in the profile directory depends on the manner in which the data have been gathered. At present, there are three different types of CODAS profile directories:

Type	Key(s)
0	time
1	time & position
2	time & depth range
3	time, depth range, & position

Hence, data gathered from a fixed installation would not require position as a key variable in the profile directory. Moving installations, on the other hand, would require a position key. A similar distinction may be drawn for the depth range key. It is best to include only those keys that would vary from profile to profile, since each additional key increases storage space requirements.

Optional Parameters.

These parameters identify and classify the variables in a CODAS database, and may also be used to define the elements for structure data types, if present, for proper interpretation during access and conversions. They may appear in any order or not at all, depending on the kind of data present. They use the following keywords:

Keyword	Parameter Example	
BLOCK_VAR	35 STRUCT CONFIGURATION_1	0 1 none
UNUSED	36 STRUCT CONFIGURATION_2	0 1 none
PROFILE_VAR	37 STRUCT ANCILLARY_1	0 1 none
DEFINE_STRUCT	CONFIGURATION_1 23	

BLOCK_VAR. This keyword is used to declare that the indicated variable is present in the CODAS database and is constant across profiles within a given block file (it varies only across blocks). Its syntax is given by:

BLOCK_VAR <id #> <value type> <variable name> <offset> <scale> <units>

<id #>: Each variable in a CODAS database must be assigned a unique identification number. In fact, we prescribe an entire numbering scheme that covers most variables that we anticipate to be included in any type of oceanographic application. This is given by the file

\CODAS3\INCLUDE\DATA_ID.H and also the example producer definition file, *\CODAS3\DOC\OCEANDAT.DEF*. The identification numbers run from 0 to a maximum of 32767. It is recommended that the user adopt a similar numbering scheme as provided in DATA_ID.H for consistency, and use the unassigned numbers for variables that may not be on that list.

<value type>: The <value type> indicates what type of values the variable may assume. It must be given by any one of the followingreserved words:

Reserved Word	C language Equivalent	FORTRAN Equivalent
CHAR	char	CHARACTER*1
BYTE	char	CHARACTER*1
UBYTE	unsigned char	CHARACTER*1
SHORT	short int	INTEGER*2
USHORT	unsigned short int	INTEGER*2
LONG	long int	INTEGER*4
ULONG	unsigned long int	INTEGER*4
FLOAT	float	REAL*4
DOUBLE	double	REAL*8
COMPLEX	struct {float real; float imaginary}	ARRAY OF REAL*4
TEXT	char[] (string)	ARRAY OF CHARACTER
STRUCT	struct	ARRAY OF CHARACTER

It should be noted that the FORTRAN equivalences given above are not quite accurate because certain C language data types are not available in FORTRAN. For example, FORTRAN does not have unsigned data types, in which case, one has the option of using the signed data type equivalent given above and remember to interpret the negative signs appropriately, or using the signed data type directly following, e.g., using a FLOAT for ULONG or a LONG for USHORT. While the latter option avoids sign confusion, it adds to either space requirements or level of complexity. Another C data type not available in FORTRAN is "struct", which can be approximated as an array of character, appropriate portions of which are equated to the appropriate element type using the EQUIVALENCE statement. For example, suppose the C structure is defined as:

```
struct
{
  SHORT first_good_bin,
      last_good_bin;
  FLOAT U_ship_absolute,
      V_ship_absolute;
  SHORT user_flag_1,
      user_flag_2,
      user_flag_3,
      user_flag_4;
} acc_var;
```

A FORTRAN program may access the same structure using the following declaration:

```
CHARACTER(20) ACCVAR
INTEGER*2   FGB, LGB
REAL*4      USA, VSA
INTEGER*2   UF1, UF2, UF3, UF4
EQUIVALENCE  (ACCVAR(1), FGB), (ACCVAR(3), LGB),
&           (ACCVAR(5), USA), (ACCVAR(9), VSA),
&           (ACCVAR(13), UF1), (ACCVAR(15), UF2),
&           (ACCVAR(17), UF3), (ACCVAR(19), UF4)
```

For each of the simple numeric value types above, the following byte sizes and hence, anticipated ranges are observed:

Value Type	Size in Bytes	Anticipated Range
BYTE	1	-128 to +127
UBYTE	1	0 to +255
SHORT	2	-32768 to +32767
USHORT	2	0 to +65535
LONG	4	-2,147,483,648 to +2,147,483,647
ULONG	4	0 to +4,294,967,295
FLOAT	4	-1.0e38 to +1.0e38
DOUBLE	8	-1.0e38 to +1.0e38

Note that the actual ranges for FLOATs and DOUBLEs are implementation-dependent and that the anticipated ranges represent an approximate minimum among various machines tested. At this point, it is not relevant to indicate the size of arrays, if any. That is done during the actual loading of the database.

<variable name>: The <variable name> is a unique string that matches a given <id #>. Each name should contain only alphanumeric characters and underscores. The maximum number of characters is 19. See the file \CODAS3\INCLUDE\DATA_ID.H for a list of recommended variable names and their identification numbers that the user can select from. One may use the unassigned numbers for new variable names, if a suitable one is not found in that file.

<offset>, <scale> and <units>: The <offset> and <scale> are given by floating point numbers separated by a space. The <units> is specified as a string (11-character maximum) with no embedded spaces. Units should be mks or one of a few special oceanographic variables. The basic unit specifications that have been defined so far are:

m	meters
s	seconds
kg	kilograms

C	degrees C
dbar	decibars
dyn_m	dynamic meters
ppt	parts per thousand
none	no standard unit, or irrelevant

Undoubtedly, more will be added. Specifications should be as simple as possible in any case. Exponents should be given by a simple positive integer following the unit. Numerator and denominator are separated by "/". If units are irrelevant for a given variable, then they should be specified as "none", NOT left blank. The offset and scale are used for automatically converting data between the form in which they are actually stored in the database and the form indicated by the units, when the user so specifies (see DBADD_F, DBGET_F, and DBPUT_F). If there is a perfect match between these two forms, then the offset and scale should be designated as 0 and 1, respectively. A major reason for rescaling data is to economize on storage. For example, U-component velocities expressed in meters/second would require FLOATs (4 bytes) in order to preserve precision. Rescaling with an offset of 0 and a scale of .001 would permit compressing storage requirements to 2-byte SHORTs.

PROFILE_VAR. This keyword is used to declare that the indicated variable is present in the CODAS database and that it varies across profiles. It should be followed by the following parameters:

PROFILE_VAR <id #> <value type> <variable name> <offset> <scale> <units>

See the section on BLOCK_VAR above for a description of each of the parameters. It should be noted that declaring a variable to be PROFILE_VAR, rather than BLOCK_VAR, causes it to be stored with each profile in a given block file. Hence, a variable whose value(s) does not vary across profiles is best declared as BLOCK_VAR in order to save space.

UNUSED. This keyword is used to declare that the indicated variable is not present in the CODAS database. Actually, one has the option of leaving out the variable from the producer definition file in order to indicate this, resulting in quicker processing of the file and very minor space saving. The preferred alternative, however, is to leave the variable in and declare it as UNUSED, in order to preserve the original file contents as much as possible.

DEFINE_STRUCT. This keyword is used to indicate that a structure-type variable is about to be defined. The variable may either be one of the BLOCK_VAR or PROFILE_VAR structure-type variables, or a structure-type element occurring in another structure definition. It does not make sense to define UNUSED structure-type variables as this would result in unnecessarily larger block files and probably slower access. A structure definition consists of a list of element specifications that comprise the structure. Its syntax is given by either of the following forms, depending on what level of structure-type variable is being defined:

DEFINE_STRUCT <variable name> <n>

DEFINE_STRUCT <element name> <n>

The <variable name> or <element name> must exactly match its initial occurrence in order to establish the association with the structure definition. The parameter <n> signifies how many elements comprise the structure, as indicated by the number of ELEM statements that must directly follow the DEFINE_STRUCT statement. The syntax of the ELEM statement is:

ELEM <count> <value type> <element name> <units>

For example:
ELEM 1 FLOAT avg_interval s
ELEM 1 SHORT num_bins none

<count>
This parameter indicates whether the element is single-valued or an array of the indicated value type.

<value type>
This parameter indicates the type of values that the element may assume. It follows the specifications described above for the value type parameter in a BLOCK_VAR statement.

<element name>
This is a string that will be used to reference the particular element. It follows the specifications described above for the variable name parameter in a BLOCK_VAR statement.

<units>
This is a string that indicates the unit of measure used for the given element value. It follows the specifications described above for the units parameter in a BLOCK_VAR statement.

The External Structure Definition File

The external structure definition file is an optional file that is useful for accessing a database that has structure-type variables. It permits:

1) printing the contents of structure-type variables in readable format, in case the structure definition was not loaded into the database at the time of creation.

2) providing a new structure definition for use during current access, overriding that stored in the database at the time of creation.

3) specifying print formats to override the default.

An external structure definition file therefore uses two major types of statements:

DEFINE_STRUCT <variable or element name> <n>
DEFINE_FORMAT <variable or element name> <n>

In both cases, the <variable or element name> refers to a structure-type variable, or a structure-type element occurring in another structure-type variable or element. (That is, the latter can be nested several levels deep.) The parameter <n> signifies the number of elements that comprise the structure, which should match the number of lower-level statements (ELEM, FORMAT) that immediately follow. See the file \CODAS3\UTIL\MORE_SD.DAT for an example of an external structure definition file.

DEFINE_STRUCT. The DEFINE_STRUCT statement has two options:

1) it can look exactly like that described under the DEFINE_STRUCT section on the producer definition file.

2) each ELEM statement in the above version can be followed by a FORMAT statement indicating the printing format to be used for that element in place of the default.

FORMAT statement in this case would look like:

FORMAT "<format string>" <pause flag> <print function name>

<format string>

The <format string> is a C language type format string that may contain plain and escape characters, and conversion specifications. The quotation marks must be provided to delimit the string contents. See any C reference manual for further details.

<pause flag>

The <pause flag> is used to control scrolling when printing is directed to the screen. The following values may be used (case is irrelevant here):

PAUSE
GO

PAUSE means that the program will stop after printing that particular element, and will resume printing the remaining elements only after the user presses the enter key. GO means that the program should continue with printing the subsequent element.

<print function name>

The print function name is used to indicate whether a special printing function should be used in place of the default generic "fprintf" function in the C language. This is useful when the element value stored must undergo some conversion prior to printing. For example, if an element contains latitude measured in hundredths of a degree, one may want to convert it to the usual degree-minutes-seconds-hundredths format and have it printed accordingly. A set of CODAS printing functions have such conversions built in and may be used for this parameter:

Print Function Name	What It Does

print_bit_pattern	it takes a ULONG value and prints it as 32 bits
prpckt	it takes a ULONG time expressed in hundredths of a second and prints it in yyyy/mm/dd hh:mm:ss.hh format
prpckp	it takes a ULONG position expressed in hundredths of a degree and prints it in ddd' mm" hh.ss format
print_ymdhms_time	it prints a YMDHMS_TIME_TYPE structure in yyyy/mm/dd hh:mm:ss.hh format
print_dmsh_position	it prints a DMSH_POSITION_TYPE structure in ddd' mm" hh.ss format
NULL	uses default printing function printf

For example:
ELEM 1 FLOAT snd_spd_used m/s
FORMAT "\nsound speed used..... %6.2e m/s" GO NULL

The element snd_spd_used is declared as a FLOAT. When printing this element, a new line is used and the indicated label is printed first, prior to printing the value in exponential notation with field width of 6 and 2 decimal places followed by the unit of measure. The GO indicates that there should be no pause in screen output between printing this element and the succeeding one. The NULL indicates that the default printing function be used. DEFINE_FORMAT. This statement is used to define a printing format for a structure whose definition has already been indicated elsewhere, either inside the block file structure definitions, or in another DEFINE_STRUCT statement in an external structure definition file. As mentioned, its syntax is given by:

DEFINE_FORMAT <variable or element name> <n>

It is immediately followed by <n> FORMAT statements, each of which provides the printing format for the corresponding element in the structure. Note that the ordering of FORMAT statements should match the arrangements of elements in the structure and that there should be a one-to-one correspondence between the FORMAT statement and the elements in the structure. The FORMAT statement has a slightly different form when used with the DEFINE_FORMAT statement:

FORMAT "<element name>" "<format string>" <pause flag> <print function name>

The additional parameter <element name> is the name of the structure element corresponding to the FORMAT statement and is used to match up the format with the element. Hence, it must exactly match the structure definition element name.

****