GBT MEMO 203

Jerk-Minimizing Trajectory Generator in C

Donald C. Wells*

December 31, 1999

Abstract

When a mechanical system is accelerated and decelerated to perform some desired trajectory, it is likely to vibrate. The vibrations are excited by the beginning and end of acceleration/deceleration intervals, not by constant acceleration/deceleration itself. The amplitude of the vibrations can be reduced if the rate of change of acceleration (the jerk) is reduced. This memo describes the C function `jmCalcTrajectory()`, which computes multiple jerk-minimizing trajectories. This memo also documents the C function `jmPosicastTrajectory()`, which implements the "Posicast" algorithm by convolving multiple trajectories computed by `jmCalcTrajectory()` with pairs of impulses; if the time separations of the impulse pairs is half of the vibrational period(s) of the system being driven, the jerk-induced vibrations can be *cancelled* (minimized all the way to *zero*). The current version of the `jm` package is available under GNU Public License as file `ftp://fits.cv.nrao.edu/pub/gbt_dwells_jm.tar.gz` [468 kilobytes].

Contents

*`dwells@nrao.edu` `http://www.cv.nrao.edu/~dwells`
National Radio Astronomy Observatory, 520 Edgemont Road, Charlottesville, VA 22903-2475 USA

1 Vibrations, servo controllers & piecewise-parabolic trajectories

Large radio antennas are easily excited into vibration at their modal frequencies by jerky motions. Consequently many authors have discussed various methods of minimizing such jerk-induced vibration. In the early days of the GBT [Green Bank Telescope[1]] project a test problem was specified to guide this discussion: a 10×10 raster of points on the sky is to be measured; the beam is to settle within 15 arcsec of each point, observe for 10 s and then move to the next point 10 arcmin away. The assymetric, off-axis design of the GBT means that the feedarm which supports its Gregorian optics is rather flexible, so that it is peculiarly susceptible to the vibration problem, and this means that the total time to scan the raster will be substantially longer than might be expected. The contractor's design study [PCD93, Table 6-2,p.76] found that the GBT would take 1376 s (23 min) to scan the raster, compared to the theoretical minimum time 1090 s (18 min). This result was reviewed by NRAO engineers, who concluded [LW93] that

> "..moving 16 million pounds of steel with high accuracy is going to take a bit more time than people are used to."

In response to [LW93], Mellstrom [Mel93] argued for use of trajectories generated by a preprocessor which will keep servo systems operating in the linear regime with reduced vibration levels, and he mentioned the preprocessor work by Tyler which appeared in a report about a year later [Tyl94]. Simulations by Gawronski and Parvin [GP95] showed that Tyler's technique, which uses $\sin^2 t$ profiles (often called "CPP-B" [Command Pre-Processor-B] in the GBT project), will achieve substantial reductions in the vibration induced by step motions. This led to the decision to implement CPP-B in the Monitor & Control system of the GBT, and the C-code described in this report is the ultimate consequence of that decision. An early version of function `jmCalcTrajectory` (Section 2.1 [p.4]) was installed in the GBT M&C system in October 1999, and the version of `jmPosicastTrajectory` (Section 3.1 [p.28]) described in this report will soon replace it.

The philosophy of servos and trajectories which the author advocates is that *all* trajectories should be such that the servo error register *never* saturates; the error register should always contain values near zero during trajectories. The goal is to maintain linearity by generating *feasible* trajectories which will demand nearly maximum performance from the servos (i.e., no use of a "slew" mode in the servo). Servos which are always presented with *feasible* commands behave better [Mel93, Tyl94]. This strategy implies that the positions *during* trajectories are actually controlled to within the noise level of the linear servo system (assuming mechanical noise is not a problem). The GBT azimuth and elevation drive servos are specified to be able to operate in the linear regime up to their maximum velocities (40 °/min for Az, 20 °/min for El).

The interface between the GBT M&C system and the GBT servo system is implemented as a computer communication protocol which transmits new commanded positions every 100 ms (10 Hz). These transmissions specify a velocity and acceleration along with each position. The GBT servo system executes five times faster, at 50 Hz; it uses these PVA values to compute four commanded positions $p(t)$ during the intervals between 100 ms updates as $p = P + Vt + \frac{1}{2}At^2$. It follows that the code described in this report needs to be able to compute these PVA update values directly, with 100 ms spacing (see variable `dt` in Section 2.1 [p.4]), and that trajectories should be computed with durations which are an integral multiple of the update time `dt`. The latter requirement led to development of a binary search algorithm (see Step 11 [p.13] and Step 27 [p.32]) to find the optimum integer multiple total trajectory time (which is returned in variable `timeInterval` in Section 2.1 [p.4]). The use of this piecewise parabolic function approximation technique facilitated development of the Posicast algorithm described below, because the convolution over intervals which are not necessarily commensurate with `dt` requires interpolation to arbitrary time, which is easy with the PVA approach.

[1]see `http://info.gb.nrao.edu/GBT/GBT.html`

2 Tyler's "three-region" algorithm, with five profile functions

Tyler's four "three-region" algorithms [Tyl94] were motivated by the problem of acquiring and tracking satellites in low earth orbits, which are moving rapidly across the sky, and for which the angular accelerations can even be a problem. Tyler calls his concept "The Three-Region Method" [Tyl94, Sect.II, p.140].[2] He says that his concept

> "..was inspired by the following scenario. A target is far away and one wishes to move the antenna towards the target trajectory as quickly as possible. So one begins by accelerating the antenna to its maximum speed; that is region 1. Then, in region 2 one moves the antenna at maximum speed until one is near the target. Finally, in region 3, the antenna is decelerated until it matches the apparent target angular position and angular velocity.."

Tyler derived algorithms for four different acquisition situations:

Scheme 1: Match initial and final angular position and velocities. Use the maximum allowable acceleration thorughout each acceleration region.

Scheme 2: The same as scheme 1, but use a sinusoidal acceleration pattern to avoid large discontinuities in acceleration.

Scheme 3: The same as scheme 2, but match the final acceleration as well.

Scheme 4: The same as scheme 2, but match both initial and final accelerations.

The C-code described in this GBT Memo implements the first three of Tyler's schemes as profile functions `jmTylerA`, `jmTylerB` and `jmTylerC` respectively. In addition, two other profile functions suggested by von Hoerner [vH96], `jmSvH3` and `jmSvH4`, are also implemented. Other profile functions could be implemented in addition to the five which we currently have. For example, we could add a version of `jmTylerA` which supports $a_f \neq 0$, analogous to `jmTylerC`; this would enable us to generate faster Posicast trajectories for accelerating targets. In addition to the sinusoidal profiles which we currently have, two other functional forms for position trajectories have been suggested: (1) the Error Function [Woo95] (velocity trajectory is the Gaussian profile) and (2) the hyperbolic tangent (velocity trajectory is $\frac{d \tanh x}{dx} = \frac{1}{\cosh^2 x}$); the latter function has been used for waveguide transitions.[3] Both of these forms pose a problem for implementation: they extend to infinity, whereas the $\sin^n$ forms are finite. Anyway, the Posicast algorithm (jerk-cancellation) has proven to be superior to all types of jerk-minimization, and therefore effectively discourages any further experimentation with additional profile functions.

The jerk minimization [prefix "`jm`"] trajectory generator package operates as a four-phase process:

- `jmCalcTrajectory()` calculates the trajectory information and returns it in a private data structure,
- `jmHeadTrajectory()` returns key parameters from the private structure,
- `jmEvalTrajectory()` evaluates the trajectory table in the structure for specific times and
- `jmFreeTrajectory()` deletes the private data structure.

These four C functions are described in the following subsections.

[2] Johann Schraml, while he was with the GBT project in 1991 (on leave from the Max Planck Institute für Radioastronomie), gave a brief description [Sch91] of an algorithm for generating trajectories; his discussion is somewhat similar to that in [Tyl94].

[3] Michael Goldman, private communication

2.1 jmCalcTrajectory

Function jmCalcTrajectory() accepts arguments describing the trajectory to be computed and the velocity and acceleration capabilites of the system. The total duration of the trajectory timeInterval can be either an input or output argument. When trajectories are computed, a pointer to the table of PVA values returns in pTS:

```
enum jmErrorCodes jmCalcTrajectory(        /* returns enum code on error     */
                    enum   jmCalc   mode, /* jmFastest,jmSpecifyTime,..    */
                    enum   jmFunct funct, /* jmTylerA,jmTylerB,..          */
                    double            dt, /* time step, e.g. 0.1 s         */
                    int            nAxes, /* num axes for p0[],a_max[],etc */
                    double          p0[], /* initial position & velocity   */
                    double          v0[],
                    double          pf[], /* final position                */
                    double          vf[], /* final velocity                */
                    double          af[], /* final acceleration (jmTylerC) */
                    double            tf, /* time for pf+vf+af target traj */
                    double       v_max[], /* +/- limits for axes           */
                    double       a_max[],
                    double *timeInterval, /* ptr to total trajectory time  */
                    struct jmPS    **pTS) /* return ptr to private struct  */
```

If jmCalcTrajectory() is called with mode argument jmFastestTime, it will compute the fastest possible time and return the result as *timeInterval. If the mode is jmFastest, the trajectory will be computed and it will returned along with the *timeInterval. If a slower-than-optimum trajectory is desired, the mode jmSpecifyTime will cause the *timeInterval argument to be used and it will not be changed. If *TimeInterval is less than the optimum interval jmCalcTrajectory() will return an error code. In modes jmFastest and jmSpecifyTime the private data structure will be malloc()-ed and pointer **pTS will be returned; it should be passed to jmFreeTrajectory() when the trajectory information is no longer needed. Mode jmFastestTime should be used if the **pTS information is not needed. State information about the trajectory computation is kept in the private structure, which permits multiple invocations of jmCalcTrajectory() and jmEvalTrajectory() to execute simultaneously.

Argument p0[] to jmCalcTrajectory() is the initial position and v0[] the initial velocity; pf[] and vf[] are the final position and velocity. The initial accelerations are assumed to be zero, and the final accelerations are assumed to be zero also for all modes except jmTylerC, where af[] is used. The maximum velocities and accelerations are assumed to be symmetric; i.e., the maximum negative velocity for an axis has the same absolute value as the maximum positive velocity for that axis. jmCalcTrajectory() will return an error code if any of the arguments v0[] and vf[] are inconsistent with v_max[], or if af[] is inconsistent with a_max[].

If nAxes is greater than one, the *timeInterval returned by jmCalcTrajectory() in the jmFastest mode will be the time needed on the slowest of the nAxes. Function jmCalcTrajectory() will find this by calling itself recursively for the individual axes in jmFastestTime mode and getting the longest*timeInterval. It will then call itself in jmSpecifyTime mode to produce **pTS for all axes. All axes will arrive at their final (pf[]) positions simultaneously.

Trajectories will be computed by jmCalcTrajectory() with a specified time step dt; as the step dt is made smaller the *timeInterval result will approach the theoretical minimum, but the space required by the **pTS private structure will grow larger. Result *timeInterval returned by jmCalcTrajectory() will be an integral multiple of step dt.

jmCalcTrajectory() and jmEvalTrajectory() regard the time and position units of arguments dt, p0[], v_max[], etc, as dimensionless. The different axes may use different position units (e.g. the subreflector displacement interface has three translation axes [inches] and three tilt axes [degrees]).

Variable `tf` controls whether the final position, velocity and acceleration are to be regarded as values to be achieved at some arbitrary time (the `tf`=0 case), or as the trajectory of a moving target with whose trajectory our trajectory is to osculate (`tf` nonzero). If the target trajectory option is wanted, but the target PVA is specified at the starting time (zero), simply set `tf` to a small number like `1e-6`. It is generally possible to estimate the total trajectory time before calling the trajectory generator; the target PVA values will be more accurate if they are computed for such an estimated time close to the optimum total time.

Regarding the need for an accelerated `jmTylerA` *profile:* The "Posicast" algorithm (Section 3 [p.24]) eliminates the need for the `jmTylerB` (aka "CPP-B") functional form to minimize vibrations; instead, as we will see, the fastest and jerkiest functional form, `jmTylerA`, can be used instead, *if the vibrations caused by its jerkiness are being cancelled.* For the $a_f \neq 0$ case (accelerated target trajectory), this means that the `jmTylerC` function, which is an accelerated version of `jmTylerB`, is suboptimal. Obviously we need to add another functional form which will be an accelerated version of `jmTylerA`. This development effort has been deferred until operational experience has been obtained with the current code.

Regarding the use of recursion in this C function: Function `jmCalcTrajectory()` calls itself at several places in the code. This tactic is merely an elegant convenience which enables all portions of the trajectory algorithm to be incorporated into a single module of code; i.e., recursion is *not* an organic element of the algorithm implemented here. If this ANSI-C function were to be transliterated to Fortran-77, which does not support recursion, it could be broken into several (probably three) subroutines.

```
/* jmCalcTrajectory.c --- Jerk-minimizing trajectory generator algorithm
   1997-05-31: D.Wells <dwells@nrao.edu>, started, then development deferred
   1999-08-09: resumed work on this code.
   1999-08-18: code works for jmSpecifyTime & jmTylerA & nAxes=1 case
   1999-08-19: jmTylerB & jmTylerC implemented, jmFastestTime started
   1999-08-30: implementing jmSvH3 & jmSvH4 functions
   1999-10-17: changes for jmTylerC
   1999-11-11: fixed v2>v_max bug
   1999-12-07: change from numeric to enum error codes
   1999-12-08: implemented target trajectory feature
   1999-12-14: accelerated target trajectories now work.
*/
```

[GNU General Public License copyright notice omitted;see `http://www.gnu.org/copyleft/gpl.html`*]*

```
#include "jmInclude.h"

enum jmErrorCodes jmCalcTrajectory(        /* returns enum code on error     */
                      enum   jmCalc   mode, /* jmFastest,jmSpecifyTime,..     */
                      enum   jmFunct funct, /* jmTylerA,jmTylerB,..           */
                      double            dt, /* time step, e.g. 0.1 s          */
                      int            nAxes, /* num axes for p0[],a_max[],etc  */
                      double          p0[], /* initial position & velocity    */
                      double          v0[],
                      double          pf[], /* final position                 */
                      double          vf[], /* final velocity                 */
                      double          af[], /* final acceleration (jmTylerC)  */
                      double            tf, /* time for pf+vf+af target traj  */
                      double       v_max[], /* +/- limits for axes            */
                      double       a_max[],
                      double *timeInterval, /* ptr to total trajectory time   */
                      struct jmPS    **pTS) /* return ptr to private struct   */
{
  int         status, i, j, k, k_trial, k_first, k_total, i_save;
  double      T, dp, dv, am, x, y, y2, e0, ef, a1, a3, v2, t1, t3, t2,
              t, p, t_start, p_start, p_half, v, a, ww, T_save, z, Tmt, tmT,
            pfp, vfp, Ttf;
const double eps = 1e-8;
struct jmPS  *q, qTrial;

if (mode != jmTrial) {
  if ((0>=nAxes)||(nAxes>MAXNAXES)) return(jmcNAxesError);       E1
  if (dt <= 0.)                     return(jmcNegDtError);       E2
  if (v_max[0] <= 0.)               return(jmcNegVmaxError);       E3
  if (a_max[0] <= 0.)               return(jmcNegAmaxError);       E4
  if (fabs(v0[0]) > v_max[0])       return(jmcV0BigError);       E5
  if (fabs(vf[0]) > v_max[0])       return(jmcVfBigError);       E6
  if (fabs(af[0]) > a_max[0])       return(jmcAfBigError);       E7
  if ((tf == 0.) && (af[0] != 0.) && (funct != jmTylerC))
                                    return(jmcAfNonzeroError);       E8
}
```

$\boxed{\text{Step 1}}$ **One axis, or more than one?**

Tyler's algorithm is for a mechanical motion in one variable, or "axis". If the problem has more than one axis our technique will be to first determine, by calls to this function for each axis in the `jmFastestTime` mode, which axis is going to be take the longest time, and then to call this function for each axis in the `jmSpecifyTime` mode (see Step 13 [p.14]).

```
if (nAxes == 1) {
```

What is the value of argument `mode`? The actual computation of a trajectory occurs only for the `jmSpecifyTime` case. For the `jmFastest` and `jmFastestTime` modes we *search* for the fastest feasible trajectory; code for these modes appears later in this C function, at Step 11 [p.13].

```
  switch(mode) {
  case(jmSpecifyTime):
  case(jmTrial):  /* private mode used by jmFastestTime & jmFastest */
  case(jmNoAloc): /* private mode used by nAxes>1 */
```

$\boxed{\text{Step 2}}$ **Initialize private data object struct `jmPS`**

Mode `jmSpecifyTime` will check T, `malloc()` the trajectory struct, and then calculate the trajectory

```
   if ((mode != jmTrial) && (mode != jmNoAloc))
     if ((status = jmAlocTrajectory(pTS, jmAlocAndInit)) != 0) return(status);
```

$\boxed{\text{Step 3}}$ **Get total position change Δp & total velocity change Δv**

If `funct` is `jmTylerC` ($a_f \neq 0$), `dp` and `dv` are modified in order to "..switch to a frame of reference which has an acceleration equal to that of the target.." [Tyl94, p.151]. Also, if $t_f \neq 0$ the algorithm will osculate its trajectory to a parabolic target trajectory.

```
   T = *timeInterval;
   pfp = pf[0];
   vfp = vf[0];
   if (tf != 0.) { /* pf[0],vf[0],af[0] define parabolic target trajectory */
     Ttf = T - tf; /* at time tf wrt start time at zero.                  */
     vfp += (af[0]*Ttf);
     pfp += (vf[0]*Ttf + 0.5*af[0]*Ttf*Ttf);
   }
   dp = pfp - p0[0];
   dv = vfp - v0[0];
```

```
if (funct == jmTylerC) {
  dp -= 0.5 * af[0] * T*T;
  dv -= af[0] * T;
}
(*pTS)->n_T = floor((T / dt) + 1e-8);
/* T must be a positive integral multiple of dt: */
if ( (T <= 0.) ||
     ((fabs(T-(dt*((*pTS)->n_T)))/dt)>eps) ) return(jmcBadTError);    E9
```

$\boxed{\text{Step 4}}$ **Get maximum allowable acceleration a_m**

```
switch(funct) {
case(jmTylerA):
  am = a_max[0];
  break;
case(jmTylerB):
  am = a_max[0] * (1./2.);
  break;
```

See [Tyl94, p.152] for the following formula for a_m for the case of $a_f \neq 0$ (Tyler's "Third Acquisition Scheme"):

```
case(jmTylerC):
  am = a_max[0] - fabs(af[0])/2.;
  break;
```

The use of $\sin^3 t$ as an acceleration profile was advocated by von Hoerner [vH96]. In the `jmTylerB` case above, the maximum acceleration is scaled by $\frac{1}{2}$ for purposes of computing t_1 and t_3 because the mean acceleration during regions 1 and 3 is

$$\int_0^\pi \frac{1}{2}(1-\cos 2t)\,dt = \int_0^\pi \sin^2 t\,dt = \frac{1}{2} = 0.50$$

of that for `jmTylerA`. The analogous factor for the `jmSvH3` function will be

$$\int_0^\pi \sin^3 t\,dt = \frac{4}{3\pi} = 0.42 :$$

```
case(jmSvH3):
  am = a_max[0] * (4./(3.*PI));
  break;
case(jmSvH4):
  am = a_max[0] * (3./8.);
  break;
default: /* unrecognized function enum value */
  return(jmcFunctEnumError);    E10
}
```

$\boxed{\text{Step 5}}$ **Get normalized position change x & normalized velocity change y**

```
(*pTS)->x[0] = x = (dp / (am * T*T)) - (v0[0] / (am * T));
(*pTS)->y[0] = y = (dv / (am * T));
```

$\boxed{\text{Step 6}}$ **Test whether this (x, y) trajectory is feasible**

The two clauses of this test correspond to the outer boundaries of the normalized phase space illustrated in Figure 1. If the test fails (i.e., (x, y) is an impossible trajectory) we delete the private data structure and return `NULL` for its pointer and an error code for the function value.

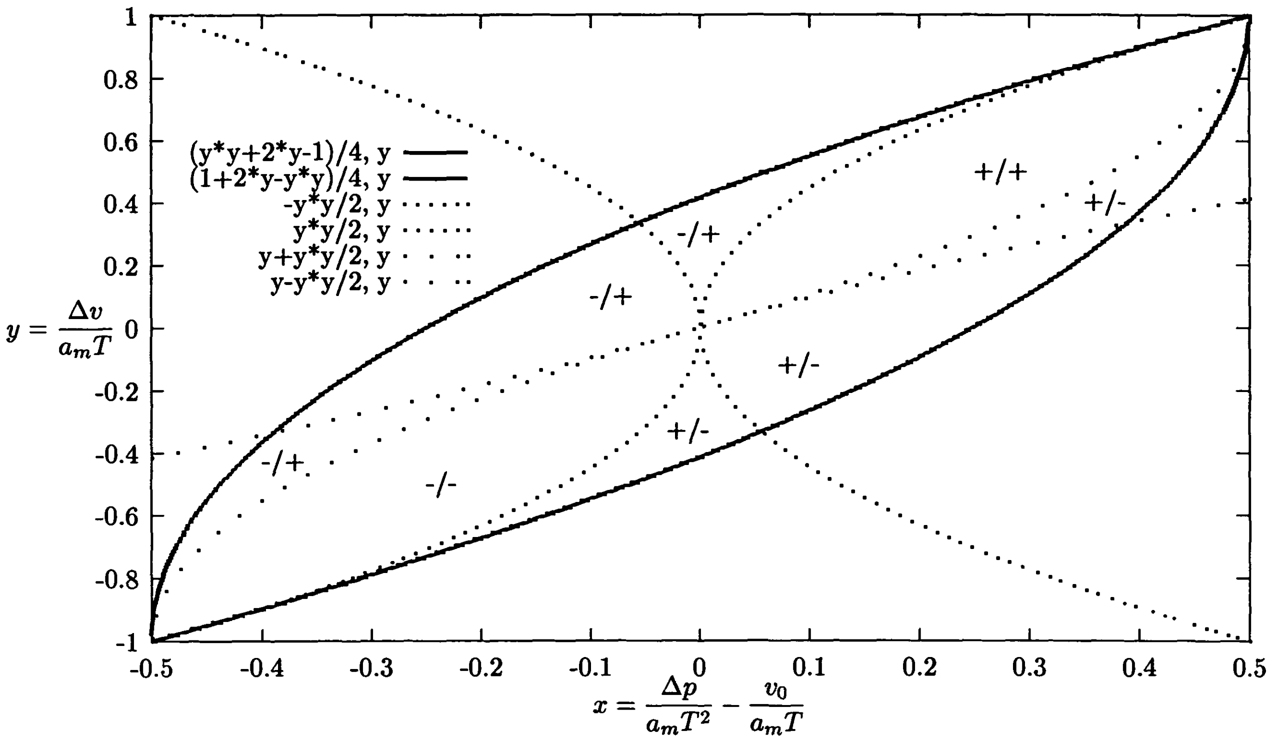

Figure 1: Areas of valid solutions for $a_0 = a_f = 0$ in normalized phase space

```
if ( (x < ((y*y + 2.*y - 1.) / 4.)) ||
     (x > ((1. + 2.*y - y*y) / 4.)) ) {
  if ((mode != jmTrial) && (mode != jmNoAloc)) jmFreeTrajectory(pTS);
  return(jmFailed);      E11
}
```

Step 7 **Determine which of four phase-space regions**

Any feasible (x, y) trajectory is in one of four categories, depending on whether the accelerations in Tyler's first and third regions are positive or negative. In Figure 1 these categories are labelled as "+/+", "-/-", "+/-" and "-/+"; for example, an area marked "-/+" means that a trajectory with (x, y) in this area starts with *negative* acceleration in region one, continues with constant velocity in region two and ends with *positive* acceleration in region three. The signs are conveyed in Tyler's algorithm by variables ε_0 and ε_f which have values ± 1. *Note:* the second test here (the one for the $\varepsilon_0 = \varepsilon_f = +1$ region) is printed incorrectly in [Tyl94]. It reads [Tyl94, Eq.(10), p.143]

$$\varepsilon_0 = \varepsilon_f = 1 \quad \text{when} \quad y < 0 \quad \text{and} \quad \frac{y^2}{2} < x \leq -y - \frac{y^2}{2} \quad (10) \quad \textit{Wrong.}$$

It should read

$$\varepsilon_0 = \varepsilon_f = 1 \quad \text{when} \quad y > 0 \quad \text{and} \quad \frac{y^2}{2} < x \leq y - \frac{y^2}{2} \quad (10) \quad \textit{Corrected.}$$

This can be seen from the pattern of the terms in the four tests, and by inspection of the topology of the regions in Figure 1 (which should resemble Tyler's Figure 5 [Tyl94, p.147]).

```
if ((y <= 0.) && ((y + y*y/2) <= x) && (x <= -y*y/2))       e0=ef=-1.;
else if ((y > 0.) && (y*y/2 <= x) && (x <= (y - y*y/2)))  e0=ef=+1.;
else if (((y >  0.) && (x > (y - y*y/2))) ||
        ((y <= 0.) && (x > -y*y/2)))            { e0=+1.; ef=-1.; }
else if (((y >  0.) && (x < y*y/2)) ||
        ((y <= 0.) && (x < (y + y*y/2))))       { e0=-1.; ef=+1.; }
else { /* it should be impossible to reach these statements: */
  jmFreeTrajectory(pTS);
  return(jmcImpossError);      E12
}
```

$\boxed{\text{Step 8}}$ **Get accelerations and times for regions 1 & 3, velocity for region 2**

The formula for t3 below corrects an omission in the list of Tyler-C changes on p.152 of [Tyl94]: equation (19) on p.144 reads

$$t_3 = \frac{v_f - v_2}{a_3} \quad (19) \quad \textit{[Misleading]} \qquad (1)$$

and should read

$$t_3 = \frac{(v_0 + \Delta v) - v_2}{a_3} \quad (19) \quad \textit{[Corrected]} \qquad (2)$$

.

```
(*pTS)->a1[0] = a1 = e0 * am;
(*pTS)->a3[0] = a3 = ef * am;
if (e0 == ef) {
  y2 = ((y*y*ef) - 2.*x) / (2.*(y*ef - 1.));
} else {
  ww = y*y*e0*ef - 2.*y*ef + 2.*x*(ef-e0) + 1.;
  y2 = (y*ef - 1. + sqrt(ww)) / (ef-e0);
}
(*pTS)->v2[0] = v2 = (am * T * y2) + v0[0];
if (mode == jmTrial) {          /* private success/failure return */
  return((fabs(v2) <= v_max[0]) ? jmNoError : jmFailed);
}
(*pTS)->t1 = t1 = (v2 - v0[0]) / a1;
(*pTS)->t3 = t3 = ((v0[0]+dv) - v2) / a3;
(*pTS)->t2 = t2 = T - t1 - t3;
if (t2 < 0.) return(jmFailed);
(*pTS)->v0[0] = v0[0]; /* p0[] not needed, first PVA specifies it */
(*pTS)->pf[0] = pfp;
(*pTS)->vf[0] = vfp;
if (funct == jmTylerC) {
  (*pTS)->af[0] = af[0];
} else {
  (*pTS)->af[0] = 0.;
}
```

$\boxed{\text{Step 9}}$ **Calculate piecewise-parabolic trajectory**

Allocate space for arrays

```
/* malloc space for pva arrays: */
if (((*pTS)->pP[0] = (double *)
     calloc((*pTS)->n_T, sizeof(double)))==NULL) return(jmcPAlocError);    E13
if (((*pTS)->pV[0] =  (double *)
     calloc((*pTS)->n_T, sizeof(double)))==NULL) return(jmcVAlocError);    E14
if (((*pTS)->pA[0] = (double *)
     calloc((*pTS)->n_T, sizeof(double)))==NULL) return(jmcAAlocError);    E15
(*pTS)->nAxes   = 1;
(*pTS)->t_first = 0.;
(*pTS)->dt      = dt;
```

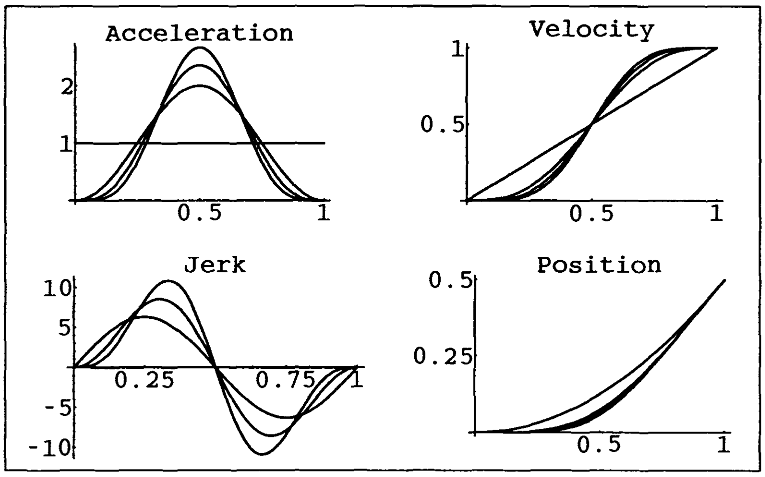

Figure 2: First-region profiles for `jmTylerA`, `jmTylerB`, `jmSvH3` & `jmSvH4`, with $a_1 = 1, t_1 = 1, v_0 = p_0 = 0$

The computation proceeds by half-steps (`dt/2`):

```
for (i = 0, j = 0; i < (2 * ((*pTS)->n_T) + 1); i++) {
  t = (*pTS)->t_first + i*(dt/2.);
```

Step 10 **Calculate $p(t)$ with specified formulæ**

Figure 2 shows the jerk, acceleration, velocity and position profiles of four of the five available functions in region-1 (region-3 is a mirror reflection). The choice of parameters normalizes the acceleration profiles to unit area.

```
    Tmt = (T-t);
    tmT = -Tmt;
    switch(funct) {
```

The `jmTylerA` formulæ (simple maximum acceleration with maximum jerk, takes shortest possible time) are from [Tyl94, pp.142-3]:

```
    case(jmTylerA):
      if (t < 0.) {                /* before  first region: */
        p = p0[0] + v0[0]*t;
      } else if (t < t1) {         /* first  region formula: */
        p = p0[0] + v0[0]*t  + (a1*t*t)/2;
      } else if (t < (t1 + t2)) {  /* second region formula: */
        p = p0[0] + v0[0]*t1 + v2*(t-t1) + (a1*t1*t1)/2;
      } else if (t < T) {          /* third  region formula: */
        p = pfp - vfp*Tmt + (a3*Tmt*Tmt)/2;
      } else {                     /* after   third region: */
    p = pfp + vfp*tmT;
  }
  break;
```

The jmTylerB formulæ, which use the raised-cosine $\frac{1}{2}(1-\cos t)=\sin^2 t$ acceleration profile for reduced jerk, but which take more time because $a_m = a_{\max}/2$, are from [Tyl94, p.148]. They have often been called "CPP-B" [Command PreProcessor B] in the GBT project.

```
case(jmTylerB):
  if (t < 0.) {                /* before   first region: */
    p = p0[0] + v0[0]*t;
  } else if (t < t1) {         /* first  region position formula: */
    p = p0[0] + v0[0]*t
      + a1*(0.5*t*t + (t1*t1/(4.*PI*PI))*(cos(2.*PI*t/t1) - 1.));
  } else if (t < (t1 + t2)) {  /* second region position formula: */
    p = p0[0] + v0[0]*t1 + v2*(t-t1) + 0.5*a1*t1*t1;
  } else if (t < T) {          /* third  region position formula: */
    p = pfp - vfp*Tmt + a3*(0.5*Tmt*Tmt +
                        (t3*t3/(4.*PI*PI))*(cos(2.*PI*Tmt/t3) - 1.));
  } else {                     /* after    third region: */
    p = pfp + vfp*tmT;
  }
  break;
```

The jmTylerC formulæ (raised-cosine acceleration for case of $a_f \neq 0$) are from [Tyl94, p.152]. These formulæ were derived by transforming the derivation of jmTylerB into an *accelerated* coordinate system.

```
case(jmTylerC):
  if (t < 0.) {                /* before   first region: */
    p = p0[0] + v0[0]*t;
  } else if (t < t1) {         /* first  region position formula: */
    p = p0[0] + v0[0]*t
      + a1*( 0.5*t*t + (t1*t1/(4.*PI*PI))*(cos(2.*PI*t/t1) - 1.))
      + 0.5*af[0]*t*t;
  } else if (t < (t1 + t2)) {  /* second region position formula: */
    p = p0[0] + v0[0]*t1 + v2*(t-t1) + 0.5*a1*t1*t1
      + 0.5*af[0]*t*t;
  } else if (t < T) {          /* third  region position formula: */
    z = (t3*t3)/(4.*PI*PI);
    p = (p0[0] + dp) - (v0[0] + dv)*Tmt
      + a3*( 0.5*Tmt*Tmt - z + z*cos((2.*PI*Tmt)/t3) )
      + 0.5*af[0]*t*t;
  } else {                     /* after    third region: */
    p = pfp + vfp*tmT
      + 0.5*af[0]*tmT*tmT;
  }
  break;
```

The jmSvH3 formulæ ($\sin^3 t$ acceleration) below were adapted from Mathematica[4] output; the formula for the first region was produced by feeding the expressions

```
d3=Integrate[Sin[t]^3, {t, 0, Pi}]/Pi
as = (1/d3) a1 Sin[ Pi t / t1]^3
vs = Simplify[v0 + Integrate[as, {t, 0, t}]]
ps = Simplify[p0 + Integrate[vs, {t, 0, t}]]
CForm[ps]
```

to Mathematica.

[4]see `http://www.wolfram.com/`

```
case(jmSvH3):
  if (t < 0.) {                /* before   first region: */
    p = p0[0] + v0[0]*t;
  } else if (t < t1) {         /* first  region position formula: */
    z = PI*t/t1;
    p = p0[0] + v0[0]*t
      + a1*(0.5*t*t1 + ((t1*t1)/(48*PI))*(sin(3.*z) - 27*sin(z)));
  } else if (t < (t1 + t2)) {  /* second region position formula: */
    p = p0[0] + v0[0]*t1 + v2*(t-t1) + 0.5*a1*t1*t1;
  } else if (t < T) {        /* third  region position formula: */
    z = PI*Tmt/t3;
    p = pfp - vfp*Tmt
      + a3*(0.5*Tmt*t3 + ((t3*t3)/(48.*PI))*(sin(3.*z) - 27*sin(z)));
  } else {                     /* after    third region: */
    p = pfp + vfp*tmT;
  }
  break;
```

The jmSvH4 formulæ ($\sin^4 t$ acceleration) below were adapted from Mathematica output; the formula for the first region was produced by feeding the expressions

```
  d4=Integrate[Sin[t]^4, {t, 0, Pi}]/Pi
  a4 = (1/d4) a1 Sin[ Pi t / t1]^4
  v4 = Simplify[v0 + Integrate[a4, {t, 0, t}]]
  p4 = Simplify[p0 + Integrate[v4, {t, 0, t}]]
  CForm[p4]
```

to Mathematica.

```
case(jmSvH4):
  if (t < 0.) {                /* before   first region: */
    p = p0[0] + v0[0]*t;
  } else if (t < t1) {         /* first  region position formula: */
    z = 2.*PI*t/t1;
    p = p0[0] + v0[0]*t
      + a1*(0.5*t*t +
            (t1*t1/(48.*PI*PI)) * (-15. +16.*cos(z) -cos(2.*z)));
  } else if (t < (t1 + t2)) {  /* second region position formula: */
    p = p0[0] + v0[0]*t1 + v2*(t-t1) + 0.5*a1*t1*t1;
  } else if (t < T) {        /* third  region position formula: */
    z = 2.*PI*Tmt/t3;
    p = pfp - vfp*Tmt +
      a3*(0.5*Tmt*Tmt +
          (t3*t3/(48.*PI*PI)) * (-15. +16.*cos(z) -cos(2.*z)));
  } else {                     /* after    third region: */
    p = pfp + vfp*tmT;
  }
  break;
default: /* unrecognized function enum value */
  return(jmcUnrecFunctError);     E16
}
```

The piecewise-parabolic approximation to the trajectory is computed by evaluating the trajectory at half-steps and setting the PVA such that in each interval dt the PVA parabola passes through the starting value, the half-step value and the final value.

```
if ((i % 2) == 0) {         /* at whole ticks: */
  if (i > 0) {
    v = (-3.*p_start +4.*p_half -1.*p) / dt;
```

```
        a = ( 4.*p_start -8.*p_half +4.*p) / (dt*dt);
        ((*pTS)->pP[0])[j] = p_start;
        ((*pTS)->pV[0])[j] = v;
        ((*pTS)->pA[0])[j] = a;
        j++;
      }
      t_start = t;
      p_start = p;
    } else {                /* at half ticks: */
      p_half = p;
    }
  }          /* end of the trajectory PVA-computation for-loop */
  return(jmNoError); /* exit from the nAxes=1/jmSpecifyTime case    */
             /* (also exit here from jmNoAloc private mode) */
```

Step 11 Find the minimum possible time for a single axis (begin jmFastestTime section)

Tyler does not give a derivation of the minimum feasible T; he suggests [Tyl94, p.158] a search procedure: "One can guess a time of 1 sec, and should that be insufficient, continue with guesses of 2, 4, 8, 16, 32, 64 and 128 sec until a solution is found. If powers of two seem inappropriate, one can try powers of 3, 1.5, 1.2, or whatever. One can be satisfied with the first acceptable solution, or one can backtrack, looking for an even better one." The implementation below is a full binary search which is designed to find the minimum possible integral multiple of dt which produces a feasible trajectory. As argument dt is made smaller, arbitrarily accurate values of *timeInterval can be calculated. Mode jmFastest returns not only the minimum time but also the actual trajectory, whereas jmFastestTime determines only the minimum feasible value for T and returns it in argument *timeInterval (with NULL returned for the trajectory pointer).

```
case(jmFastestTime):
case(jmFastest):
  q = &qTrial;
  status = jmAlocTrajectory(&q, jmInitOnly);
```

Binary search algorithm: The current implementation initializes search variable k_trial to one, and does a full binary search; typically this needs 10-20 trial calculations to converge (the limit set in the code below is 30 trials). Numerical experiments have shown that a good starting guess can often halve the number of trials. For example, one can try estimating the optimum time and then setting $k_{\text{trial}} = T_{\text{estimate}}/\Delta t$. This estimate must be less than twice as large as the true optimum, or the binary search will not converge to the true optimum time. Any k_trial less than the optimum will work because the first pass will keep doubling it until it gets within a factor of two of optimum,[5] and then the binary search will converge. k_trial need not start out as a power of two, but it must be greater than zero. The simplest, cleanest implementation is to start it as one:

```
  k_trial = 1;
  k_first = 1;
  k_total = 0;
  k = 0;
  while (k_trial > 0) { /* loop halts when division by two yields zero. */
    T = (k + k_trial) * dt;
    status = jmCalcTrajectory(jmTrial, funct, dt, nAxes,
                              p0, v0, pf, vf, af, tf, v_max, a_max,
                              &T, &q);
    if ((status != jmNoError) &&
        (status != jmFailed))
```

[5]Note that it is conceivable that the doubling strategy will fail in the $a_f \neq 0$ case because the velocity might increase fast enough at a high enough velocity that we can never "catch up", or else the accelerating target trajectory might exceed $v_{\max}$. These possibilities are unlikely in practical situations.

```
        return(status); /* Unexpected nonzero status; return it */
      k_total++;
      if (k_total > 30) return(jmcBigKtotalError);      E17
      if (status == jmFailed) {
        if (k_first) {
          k_trial *= 2; /* First pass: keep doubling until succeed */
          if (k_trial >= 1073741824) return(jmcKbigFailure); /* 2^30 */     E18
        } else {
          k = k + k_trial;
          k_trial /= 2;
        }
      } else { /* trial time succeeded: */
        T_save = T;
        if (k_first) {
          k = k + k_trial / 2; /* First pass: set up binary search */
          k_trial /= 4;
          k_first = 0;
        } else {
          k_trial /= 2;
        }
      }
    }
```

The fact that this binary search loop halts when *integer* k_trial has been reduced to zero by successive division by two implies that T_save is the smallest integral multiple of dt for which a feasible trajectory exists:

```
    *timeInterval = T_save;       /* return optimum time           */
    if (mode == jmFastestTime) {
      *pTS = NULL;
      return(jmNoError); /* exit from the nAxes=1/jmFastestTime case */
    }
```

$\boxed{\text{Step 12}}$ **Optimum T is known, compute trajectory**

```
    status = jmCalcTrajectory(jmSpecifyTime, funct, dt, nAxes,
                              p0, v0, pf, vf, af, tf, v_max, a_max,
                              timeInterval, pTS);
    return(status); /* exit from the nAxes=1/jmFastest case */

  default:     /* unrecognized jmMode value; return error code */
    return(jmcUnrecModelError);    E19
  }            /* end of nAxes=1/jmMode switch-statement */
```

$\boxed{\text{Step 13}}$ **Generate trajectories for nAxes>1 situations**

In the multiple trajectory case, if we are in jmFastest mode, we loop over the axes with jmFastestTime mode to find which axis will take the longest time, and then we loop again in jmSpecifyTime mode to calculate the actual trajectories. (If we are in jmFastestTime mode, we return the time without computing the trajectories.) If we are called with jmSpecifyTime mode we can skip the first loop.

```
} else {
  switch(mode) {
  case(jmFastest):
  case(jmFastestTime):
    T_save = 0.;
    i_save = 0;
    q = &qTrial;
    status = jmAlocTrajectory(&q, jmInitOnly);
    for (i = 0; i < nAxes; i++) {
      if ((status = jmCalcTrajectory(jmFastestTime, funct, dt, 1,
                                     &p0[i], &v0[i], &pf[i], &vf[i],
                                     &af[i], tf, &v_max[i], &a_max[i],
                                     &T, &q)) != 0) return(status);
      if (T > T_save) {
        T_save =  T;
        i_save = i;
      }
    }
    *timeInterval = T_save;
```

Mode `jmFastestTime` will return here, `jmFastest` continues as though mode were `jmSpecifyTime`:

```
    if (mode == jmFastestTime) {
      *pTS = NULL;
      return(jmNoError);
    }
```

Create instance of private struct, do `jmSpecifyTime`:

```
  case(jmSpecifyTime):
    if (mode == jmSpecifyTime) i_save = 0;
    if ((status = jmAlocTrajectory(pTS, jmAlocAndInit)) != 0) return(status);
    q = &qTrial;
    status = jmAlocTrajectory(&q, jmInitOnly);
    for (i = 0; i < nAxes; i++) {
      if ((status = jmCalcTrajectory(jmNoAloc, funct, dt, 1,
                                     &p0[i], &v0[i], &pf[i], &vf[i],
                                     &af[i], tf, &v_max[i], &a_max[i],
                                     timeInterval, &q)) != 0) return(status);
      if (i == i_save) {
        (*pTS)->slowest[i] = (mode == jmSpecifyTime) ? 0 : 1;
        (*pTS)->nAxes      = nAxes;
        (*pTS)->n_T        = q->n_T;
        (*pTS)->t_first    = q->t_first;
        (*pTS)->dt         = q->dt;
        (*pTS)->t1         = q->t1;
        (*pTS)->t2         = q->t2;
        (*pTS)->t3         = q->t3;
      } else {
        (*pTS)->slowest[i] = 0;
      }
      (*pTS)->v0[i] = q->v0[0];
      (*pTS)->vf[i] = q->vf[0];
      (*pTS)->pf[i] = q->pf[0];
      (*pTS)->af[i] = (funct == jmTylerC) ? q->af[0] : 0.;
      (*pTS)->x[i]  = q->x[0];
      (*pTS)->y[i]  = q->y[0];
      (*pTS)->a1[i]  = q->a1[0];
      (*pTS)->v2[i]  = q->v2[0];
      (*pTS)->a3[i]  = q->a3[0];
```

Move pointers to PVA vectors for axis `i` from the internal private structure `qTrial` to the `*pTS` struct:

```
      (*pTS)->pP[i] = q->pP[0];
      (*pTS)->pV[i] = q->pV[0];
      (*pTS)->pA[i] = q->pA[0];
      }
      return(jmNoError);
```

Unrecognized mode:

```
    default:
      return(jmcUnrecMode2Error);      E20
    }
    return(jmNoError);
  }
}
```

2.2 jmHeadTrajectory, jmEvalTrajectory, jmFreeTrajectory & jmPrintTrajectory

The use of the functions described below is illustrated in program jmDemoProgram (Figure 3 [p.18]) in Section 2.3 [p.18]. The reader can compare the descriptions of the calling sequences and actions of the functions given here to the code in that program.

Functions jmCalcTrajectory and jmPosicastTrajectory return their results in a private data structure. Function jmHeadTrajectory will return the key parameters from the header of that structure. Variable n_T is the number of PVA segments used in the piecewise parabolic approximation to the trajectory and dt is the time step of the segments; therefore the total duration of the trajectory is (n_T $\times$ dt). Variable t_first is the time of the first PVA segment; it is normally zero.

```
enum jmErrorCodes jmHeadTrajectory(       /* returns enum code on error     */
                  struct jmPS *pTS,       /* pointer to private struct      */
                  int      *nAxes,        /* returns nAxes   from struct    */
                  int        *n_T,        /* returns n_T     from struct    */
                  double  *t_first,       /* returns t_first from struct    */
                  double       *dt)       /* returns dt      from struct    */
```

Function jmEvalTrajectory() computes the position, velocity and acceleration of a trajectory for an arbitrary specified time t. jmEvalTrajectory() finds the trajectory table entry appropriate for t and evaluates the PVA values; if the time is not a multiple of the dt spacing of the table this amounts to interpolation. If the time is before the beginning or after the end of the interval for which jmCalcTrajectory or jmPosicastTrajectory generated the trajectory, jmEvalTrajectory() will extrapolate the trajectory.

```
enum jmErrorCodes jmEvalTrajectory(       /* returns enum code on error     */
                  struct jmPS *pTS,       /* pointer to private struct      */
                  double         t,       /* time                           */
                  double       p[],       /* returns     position[nAxes]    */
                  double       v[],       /* returns     velocity[nAxes]    */
                  double       a[])       /* returns acceleration[nAxes]    */
```

Function jmPrintTrajectory is a useful tool for debugging implementations based on this package. See Figure 4 [p.19] for an example of its output.

```
enum jmErrorCodes jmPrintTrajectory (                /* returns error code */
                                    FILE        *fp, /* output file stream */
                                    struct jmPS *pPS, /* private struct ptr */
                                    char   string[], /* up to 40 char      */
                                    int      k_rows) /* which rows to print?*/
```

Function jmFreeTrajectory deletes an instance of the private trajectory structure created by jmCalcTrajectory or jmPosicastTrajectory. See the last statements of Figure 3 [p.18] for an example of its use. Function jmAlocTrajectory is called by jmCalcTrajectory and jmPosicastTrajectory in jmAlocAndInit mode to create the structure; it is not intended for public use, and its calling sequence is illustrated here for background information.

```
enum jmErrorCodes jmFreeTrajectory(       /* returns enum code on error     */
                  struct jmPS **pTS)      /* private struct is free()-ed    */
```

```
enum jmErrorCodes jmAlocTrajectory(       /* returns enum code on error     */
     /* THIS IS A *PRIVATE* FUNCTION OF THE jm PACKAGE! NOT FOR GENERAL USE. */
                     struct jmPS **pTS, /* private struct is malloc()-ed */
                     enum jmCalc  mode) /* jmInitOnly skips malloc()     */
```

```
/* jmDemoProgram -- program to illustrate use of jm package
   1999-12-13: D.Wells, NRAO-CV */

[GNU General Public License copyright notice omitted;see http://www.gnu.org/copyleft/gpl.html]

#include "jmInclude.h"

int main()
{
  FILE   *fp;
  int    nAxes, n_rows, h_nAxes, h_n_T;
  double T, dt, tf, h_dt, h_t_first, T_e, p_e[2], v_e[2], a_e[2],
         p0[2], v0[2], pf[2], vf[2], af[2], v_max[2], a_max[2];
  char   string[JM_MAX_ERROR_MSG_LENGTH];
  enum   jmErrorCodes status;
  struct jmPS         *pPS;

  fp = fopen("jmDemoProgramOut.txt", "w");
  dt       = 0.020;         nAxes    = 2;
  /* initial conditions for trajectory: */
  p0[0]    = 104.144423;  p0[1]    = 48.766487;
  v0[0]    = -0.000404;   v0[1]    = 0.000000;
  /* target trajectory to match: */
  tf = 9.9;
  pf[0]    = 106.603651;  pf[1]    = 47.693706;
  vf[0]    = 0.003625;    vf[1]    = 0.003136;
  af[0]    = 0.;          af[1]    = 0.;
  /* capabilities of the system: */
  v_max[0] = 0.66;        v_max[1] = 0.33;
  a_max[0] = 0.2;         a_max[1] = 0.2;
  status = jmCalcTrajectory(jmFastest, jmTylerB, dt, nAxes,
                            p0, v0, pf, vf, af, tf, v_max, a_max,
                            &T, &pPS);
  jmStrError(status,string); fprintf(fp,"jmCalcTrajectory() status: %s\n",string);
  if (status!=jmNoError) exit(EXIT_FAILURE);
  status = jmHeadTrajectory(pPS, &h_nAxes, &h_n_T, &h_t_first, &h_dt);
  jmStrError(status,string); fprintf(fp,"jmHeadTrajectory() status: %s\n",string);
  if (status!=jmNoError) exit(EXIT_FAILURE);
  T_e = 9.0137;
  status = jmEvalTrajectory(pPS, T_e, p_e, v_e, a_e);
  jmStrError(status,string); fprintf(fp,"jmEvalTrajectory() status: %s\n",string);
  if (status!=jmNoError) exit(EXIT_FAILURE);
  fprintf(fp, "T=%g, nAxes=%d, n_T=%d, dt=%g, T_e=%g, pva_e[1]=%g %g %g\n",
         T, h_nAxes, h_n_T, h_dt, T_e, p_e[1], v_e[1], a_e[1]);
  n_rows = 1.0 / dt; /* specify 1-second spacing in table */
  status = jmPrintTrajectory(fp, pPS, "jmDemoProgram output", -n_rows);
  jmStrError(status,string); fprintf(fp,"jmPrintTrajectory() status: %s\n",string);
  if (status!=jmNoError) exit(EXIT_FAILURE);
  status = jmFreeTrajectory(&pPS);
  jmStrError(status,string); fprintf(fp,"jmFreeTrajectory() status: %s\n", string);
  if (status!=jmNoError) exit(EXIT_FAILURE);
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

Figure 3: Demonstration program

2.3 Demonstration program

Program jmDemoProgram (Figure 3 [p.18]) calls most of the functions described previously to compute a two-axis trajectory and print a summary of it. The intent of this program is to illustrate the appropriate syntax for utilizing the functions to accomplish a typical task. The target trajectory has constant velocity; the position for $t = 9.9$ s is specified (the rate specified is approximately the sidereal rate, so this example might be a celestial source rising in the southeastern sky, with the telescope's initial position about 2° from

```
jmCalcTrajectory() status: jmNoError -- Normal successful return
jmHeadTrajectory() status: jmNoError -- Normal successful return
jmEvalTrajectory() status: jmNoError -- Normal successful return
T=9.9, nAxes=2, n_T=495, dt=0.02, T_e=9.0137, pva_e[1]=47.7247 -0.104586 0.132149

+-----------------------------------------+
|            jmDemoProgram output         |
+-----------------------------------------+
|                      nAxes =          2 |
|                        n_T =        495 |
|                    t_first =          0 |
|                         dt =       0.02 |
+-----------------------------------------+
|                          j =          0 |
|           pPS->slowest[j] =          1 |
|                pPS->v0[j] =  -0.000404 |
|                pPS->pf[j] =   106.6037 |
|                pPS->vf[j] =   0.003625 |
|                pPS->af[j] =          0 |
|                 pPS->x[j] =  0.2513241 |
|                 pPS->y[j] = 0.004069697 |
|                pPS->a1[j] =        0.1 |
|                pPS->v2[j] =  0.4702944 |
|                pPS->a3[j] =       -0.1 |
+-----------------------------------------+
|                          j =          1 |
|           pPS->slowest[j] =          0 |
|                pPS->v0[j] =          0 |
|                pPS->pf[j] =   47.69371 |
|                pPS->vf[j] =   0.003136 |
|                pPS->af[j] =          0 |
|                 pPS->x[j] = -0.1094563 |
|                 pPS->y[j] = 0.003167677 |
|                pPS->a1[j] =       -0.1 |
|                pPS->v2[j] = -0.1243898 |
|                pPS->a3[j] =        0.1 |
+-----------------------------------------+
|                   pPS->t1 =   4.706984 |
|                   pPS->t2 =  0.5263219 |
|                   pPS->t3 =   4.666694 |
+-------------+---------------------------+---------------------------+
|   j     t   |   p[0]     v[0]     a[0]  |   p[1]     v[1]     a[1]  |
+-------------+---------------------------+---------------------------+
|  -1  -0.02  | 104.1444  -0.0004  0.0000 | 48.7665   0.0000   0.0000 |
+-------------+---------------------------+---------------------------+
|   0   0.0   | 104.1444  -0.0004  0.0000 | 48.7665   0.0000  -0.0001 |
|  50   1.0   | 104.1510   0.0268  0.0779 | 48.7191  -0.1187  -0.0621 |
| 100   2.0   | 104.2375   0.1655  0.1897 | 48.5951  -0.1244   0.0000 |
| 150   3.0   | 104.5006   0.3565  0.1640 | 48.4707  -0.1244   0.0000 |
| 200   4.0   | 104.9196   0.4603  0.0402 | 48.3463  -0.1244  -0.0000 |
| 250   5.0   | 105.3881   0.4703  0.0000 | 48.2219  -0.1244  -0.0000 |
| 300   6.0   | 105.8559   0.4574 -0.0499 | 48.0975  -0.1244  -0.0000 |
| 350   7.0   | 106.2677   0.3450 -0.1732 | 47.9731  -0.1244   0.0000 |
| 400   8.0   | 106.5175   0.1527 -0.1827 | 47.8487  -0.1244   0.0000 |
| 450   9.0   | 106.5957   0.0241 -0.0636 | 47.7261  -0.1064   0.1321 |
| 494   9.88  | 106.6036   0.0036 -0.0000 | 47.6936   0.0031   0.0001 |
+-------------+---------------------------+---------------------------+
| 495   9.9   | 106.6037   0.0036  0.0000 | 47.6937   0.0031   0.0000 |
+-------------+---------------------------+---------------------------+

jmPrintTrajectory() status: jmNoError -- Normal successful return
jmFreeTrajectory() status: jmNoError -- Normal successful return
```

Figure 4: Demonstration program output (example of `jmPrintTrajectory()` output)

the source). The `jmFastest` mode is specified in order to obtain the fastest possible trajectory which will osculate to this target trajectory. The total time returns in variable `T`; time increment `dt` is specified as 20 ms (50 Hz) (for the GBT, 100 ms would be appropriate). The `jmTylerB` profile function ($\sin^2 t$, often called "CPP-B" in the GBT project) is specified for the computation.

The program writes its output to a specified file, and this is shown in Figure 4 [p.19]. The status return lines are shown as an illustration of error-handling technique; a production program would be silent for `jmNoError`. The program calls `jmHeadTrajectory()` to obtain some key parameters, which it prints; in particular, it prints the total time for this trajectory, $T = 9.9\,\mathrm{s}$. The program calls `jmEvalTrajectory()` for an arbitrary time $t_e = 9.0137\,\mathrm{s}$, and prints the position, velocity and acceleration for the second axis at that time. The remainder of the output is produced by the utility function `jmPrintTrajectory()`, which is especially useful as a dump tool for debugging algorithms.

Variable `n_T` is the number of entries in the trajectory table of the private structure computed by `jmCalcTrajectory()`. Axis-indexed values are printed in two blocks headed by the value of the axis index `j`. `slowest[j]` is a flag which indicates the slowest axis in `jmFastest` trajectories like this one; we see that the first axis is slowest in this case. The initial and final PVA values were supplied to `jmCalcTrajectory`, while the `x`, `y`, `a1`, `v2` and `a3` values were computed by it. The x and y values can be located on Figure 1 [p.8]; the slowest axis will be closest to an outer boundary of the area of valid solutions and other axes will be closer to the origin. The acceleration used during the first Tyler "region" is a_1; a_3 is for the third region. The magnitude of these accelerations is *half* of the $a_{\max} = 0.2$ value because we are using `jmTylerB` rather than `jmTylerA` (see Step 4 [p.7]). The constant velocity in the second region is v_2, which will be closest to $v_{\max}$ for the slowest axis. The final header values shown are t_1, t_2 and t_3, the durations of Tyler's three regions; these are not multiples of `dt`, so the computed trajectory table entries do not generally match the boundaries between regions 1 and 2 and regions 2 and 3.

The trajectory table has an initial line for a time one step *before* the trajectory begins, in order to illustrate the extrapolation capability of `jmEvalTrajectory`. The trajectory table in Figure 4 [p.19] is shown for every second,[6] plus the final line of the table, which gives the PVA values for one time step before the end of the total trajectory time. Finally, a line for the end time of the trajectory is printed; if `jmEvalTrajectory` is called for later times these PVA values will be extrapolated. The reader can verify by examining the trajectory table that the second-axis PVA values for t_e printed at the top of Figure 4 [p.19] are plausible.

[6]Which lines are printed in the table is controlled by `jmPrintTrajectory()` argument `k_rows`. If `k_rows` is 0 or 1, the entire table is printed. If `k_rows` is negative, as in the demonstration program, it is the modulo factor; the formula for `n_rows` in Figure 3 [p.18] shows how to print lines for a specified time step. If `k_rows` is positive, the first `k_rows` lines of the table and the last `k_rows` lines of the table will be printed; this can be useful for debugging starting and stopping transient problems.

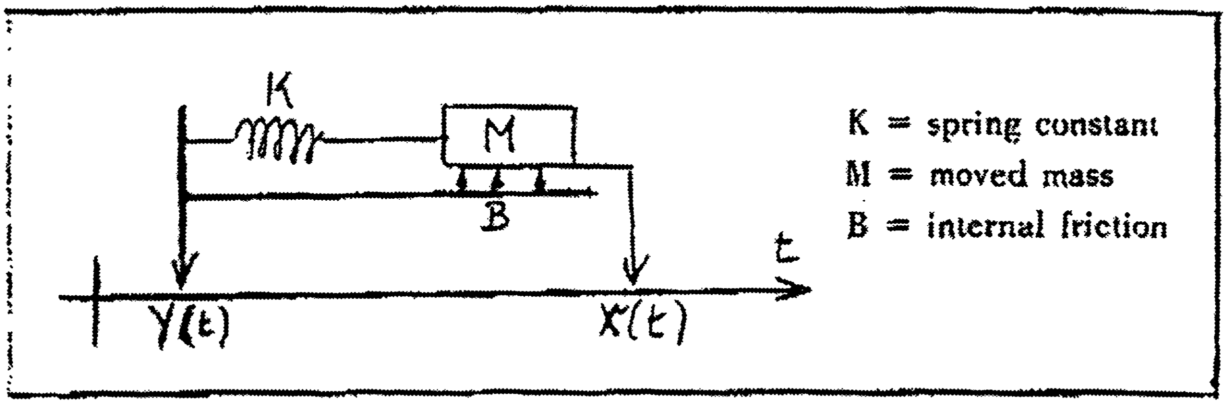
K
M
B
Y(t)
X(t)
t
K = spring constant
M = moved mass
B = internal friction
Simplified model for the telescope drive Y(t),
and its resulting movement X(t)

Figure 5: Model of damped mass driven via a spring (copy of [vH96, Fig.2])

2.4 Numerical simulations of Tyler 3-region trajectories

Figure 5 [p.21] is a simple form of the schematic model for a large radio telescope driven by a servo system, assuming that the servo has sufficient torque to maintain the driven axis nearly at the commanded position when the telescope vibrates (this will generally be true for the GBT). The spring represents the large number of steel beams of the moving structure, while M represents its mass. When the driven axis is moved by the servo, the structure will bend due to the inertia of M. If $Y(t)$ is jerky, M (i.e., X(t)) will exhibit vibration; if we are clever in constructing the form of $Y(t)$, it will have minimal vibration when the trajectory ends. In the ideal case, $(X(t) - Y(t))$ will be displaced from its equilibrium value smoothly and will return to the equilibrium value smoothly.

If we take $X(0) - Y(0) = 0$ and $v = X'(t) - Y'(t)$, the trajectory $X(t)$ and velocity $v(t)$ of M in this model can be computed by integrating the differential equations [vH96, (2)]

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\left(\frac{B}{M}\right)v - \left(\frac{K}{M}\right)(X - Y) \tag{3}$$

$$\frac{\mathrm{d}X}{\mathrm{d}t} = v \tag{4}$$

The terms on the right-hand side of (3) are the accelerations which cause changes in the velocity. The first such term is the damping, an acceleration proportional to the current velocity. The second term is the acceleration due to the spring force (Hooke's Law). The latter term assures that the mean position of the system must track the commanded trajectory, unless we have constant acceleration in which case we will have a constant offset (see Figure 14 [p.38] for an example). The frequency of the vibration will be

$$\nu = \frac{1}{2\pi}\sqrt{\frac{K}{M}} \tag{5}$$

and the logarithmic damping decrement (ratio of successive maxima) will be [vH96, (4)]

$$Q_\mathrm{d} = \frac{\pi B}{\sqrt{KM}} \tag{6}$$

```
/* jmTrajectoryDerivs.c -- Trajectory derivatives for odeint()
   1999-08-26: D.Wells <dwells@nrao.edu>, adapted from jerk_impulse.c [1995]
*/

[GNU General Public License copyright notice omitted;see http://www.gnu.org/copyleft/gpl.html]

#include "jmTrajectorySimulate.h"

void jmTrajectoryDerivs(
                        float x,       /* independent variable    */
                        float y[],     /* dependent   variables   */
                        float dydx[])  /* derivs of y[] wrt x at x */
{
  enum jmErrorCodes status;
  double t, v, p, pt, dvdt, dpdt;
  double pe[MAXNAXES], ve[MAXNAXES], ae[MAXNAXES];
  extern struct JMTS_Private JMTS;

  t = x;     /* time     */
  v = y[1]; /* velocity */
  p = y[2]; /* position */
  if ((status = jmEvalTrajectory(JMTS.pTS, t, pe, ve, ae)) != 0) {
    printf("jmTrajectoryDerivs(): jmEvalTrajectory status=%d!\n", status);
    jmPerror(status, "in jmTrajectoryDerivs()");
    exit(13);
  }
  pt = pe[0]; /* trajectory position commanded */

  dvdt = -(JMTS.B/JMTS.M)*v -(JMTS.K/JMTS.M)*(p - pt);
  dpdt = v;

  dydx[1] = dvdt;
  dydx[2] = dpdt;
}
```

Figure 6: Derivatives function supplied to Runge-Kutta integrator

The ratios in equation (3) are [vH96, (5)]

$$\frac{B}{M} = 2Q_{\rm d}\nu \quad \text{and} \quad \frac{K}{M} = (2\pi\nu)^2 \tag{7}$$

The C function `jmTrajectoryDerivs` shown in Figure 6 [p.22] implements (3) and (4) as the two statements near the end of the function. It computes $Y(t)$ (here `pt`) by calling function `jmEvalTrajectory()` with the current value of t; the previously computed trajectory data structure and the values of B, M and K are available via an externally defined `struct` called `JMTS`. This derivatives function is supplied to the Numerical Recipes function `odeint.c` [PFTV88, Section 15.2, "Adaptive Stepsize Control for Runge-Kutta"], which integrates the set of ordinary differential equations over a specified range of time.[7] A simulation driver has been built to perform this process for various combinations of parameters; it writes the results to files where they are available for plotting.

Figure 7 shows four trajectories which perform a one degree step motion, the typical demonstration case for GBT servo studies. The fastest trajectory uses the `jmTylerA` function, which applies and removes maximum acceleration abruptly; it excites vibrations in the system and the resulting position residuals are plotted with 10× enlargement. The maximum error after the trajectory completes is about ±0.01 degree (±40 arcsec), and it is slowly decreasing due to the damping $Q_{\rm d} = 0.05$ assumed in this case (damping in the actual GBT is about 10× smaller). The `jmTylerB` trajectory takes slightly longer, 6.3 versus 4.5 s, but excites a *much* smaller amount of vibration (note 100× enlargement), about one arcsecond! This is the "CPP-B" algorithm which has been much discussed, and previously simulated [GP95] for the GBT project. Two other

[7]Presumably von Hoerner [vH96] used similar technology for his trajectory calculations.

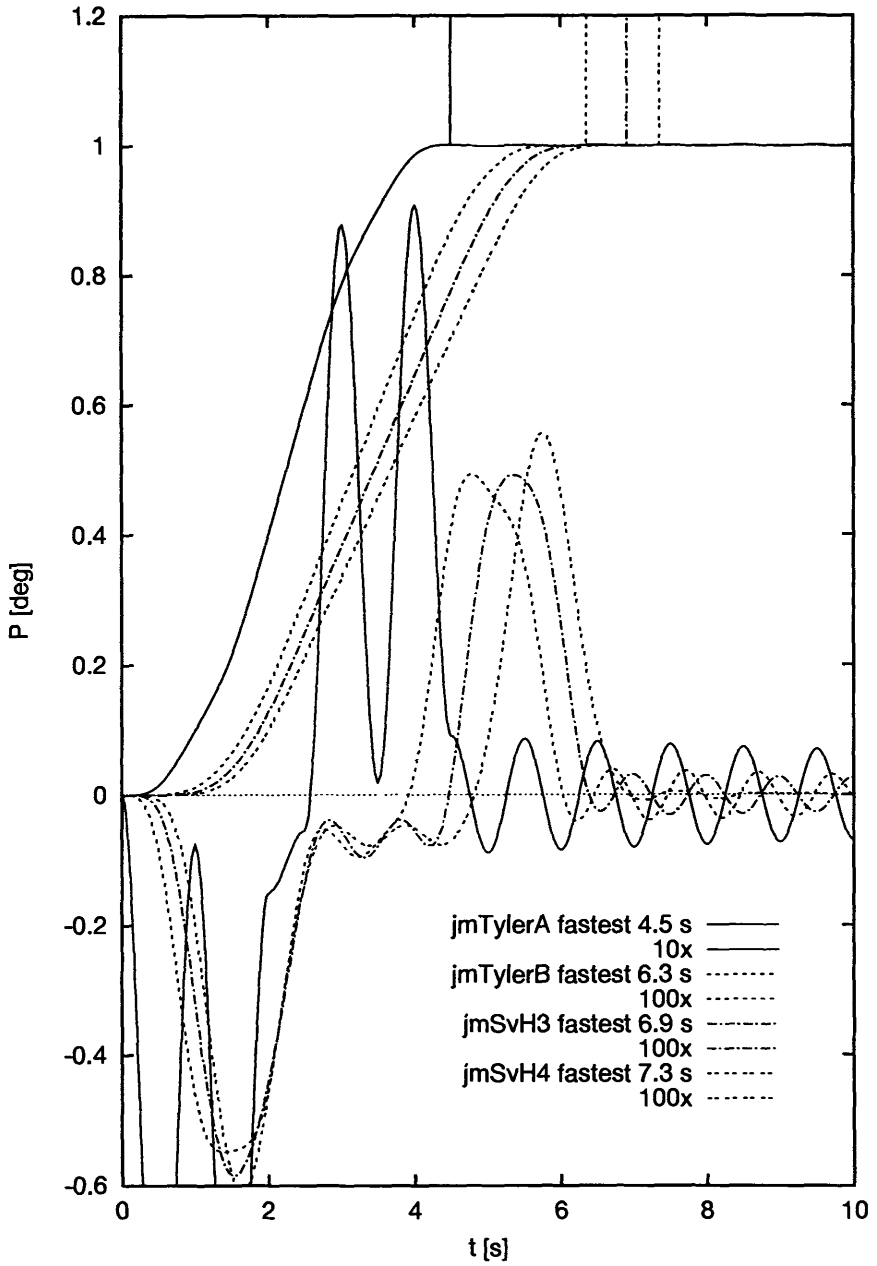

Figure 7: Four 1°-step trajectories with vibrations ($P = 1\,\mathrm{s}, Q_d = 0.05, A_{\max} = 0.2\,°/\mathrm{s}^2, V_{\max} = 0.67\,°/\mathrm{s}$)

trajectories are also plotted, using functions `jmSvH3` (recommended by von Hoerner [vH96]) and `jmSvH4`, which produce even smaller vibration levels. The values of T (total time) are plotted as vertical lines from the trajectories to the upper border of the plot so that their relative magnitudes can be estimated by eye.

Using "smoother" acceleration profiles with smaller average accelerations will generally reduce vibration levels, but experiments show that the vibration levels depend on the exact timing of the trajectories. It is easy to find combinations of parameters which will cause the `jmSvH3` profile to be inferior to the `jmTylerB` profile, a counter-intuitive result at first glance. Furthermore, as we will see, it is easy to find parameter combinations for which `jmTylerA`, the fastest and most violent trajectory function, produces *zero* net vibration!

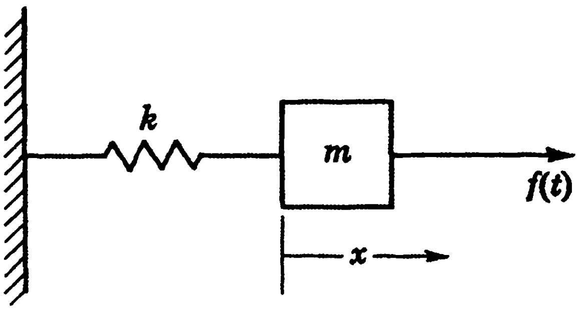

Figure 8: Mass m, spring constant k, force $f(t)$ and displacement x [Hil62, Fig.2.4, p.68]

3 Smith's "Posicast" algorithm (vibration *cancellation*)

In June 1993, while thinking about the problem of jerk-induced vibrations in the GBT, the author consulted one of his undergraduate mathematics textbooks to get the differential equation of a damped mass on a spring (i.e., equation (3) on p.21), and noticed the following discussion of the undamped case [Hil62, pp.68-71]:

> "*[p.68]* ..we take as a simple example the case of ***forced vibration*** of a mass m attached to a spring with spring constant k.. If the applied force is $f(t)$ and if *no damping is present* (Figure 8), the differential equation of motion is
>
> $$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + kx = f(t) \tag{8}$$
>
> .. *[p.70]* If a constant force $f(t) = A$ is applied when $t > 0$, there follows.. *[Laplace Transform derivation omitted]* and hence..
>
> $$x = \frac{A}{m\omega_0^2}(1 - \cos\omega_0 t). \tag{9}$$
>
> Thus, in this case, the mass oscillates with its natural frequency between the points $x = 0$ and $s = 2A/\omega_0^2 = 2A/k$, when damping is absent.
>
> If constant force is applied only over the interval $0 < t < t_0$, and no force acts when $t > t_0$, there follows.. *[another Laplace Transform derivation omitted]* [and] hence we have, *when* $0 < t < t_0$,
>
> $$x = \frac{A}{m\omega_0^2}(1 - \cos\omega_0 t); \qquad \textit{[same as (9)]} \tag{10}$$
>
> and, *when* $t > t_0$, ..
>
> $$x = \frac{2A}{k}(\sin\frac{\omega_0 t_0}{2})\sin\omega_0(t - \frac{t_0}{2}). \tag{11}$$
>
> Thus, while the force acts ($0 < t < t_0$), the mass oscillates at its natural frequency, with amplitude A/k, about the point $x = A/k$; however, after the force is removed ($t > t_0$), the mass oscillates about the point of equilibrium ($x = 0$), at the same frequency, but with an amplitude $\frac{2A}{k}\sin\frac{1}{2}\omega_0 t_0$. If $t_0 = \frac{2\pi}{\omega_0} = T$,[8] where T is the period of the natural mode of vibration, then $x = 0$ when $t \geq t_0$, so that the mass returns to its equilibrium position as the force is removed, and then remains at that position.."

The idea of that last sentence is that when the force ends at $t = t_0$, it creates a impulse which is equal and opposite to the impulse created by the initial force transient at $t = 0$, and that if the delay is exactly one vibration period the two impulses will cancel. The author noted in his logbook [1993-06-04] the "..idea of applying a precisely timed and calibrated negative impulse to stop a vibration." One month later, at the end of notes about jerk-limited trajectories, the author again noted [1993-07-08] that "..it may be possible

[8]The second term of (11) becomes $\sin\pi = 0$ in this case (actually, it will be zero for any $t_0 = nT$, with $n = 2, 3, ...$).

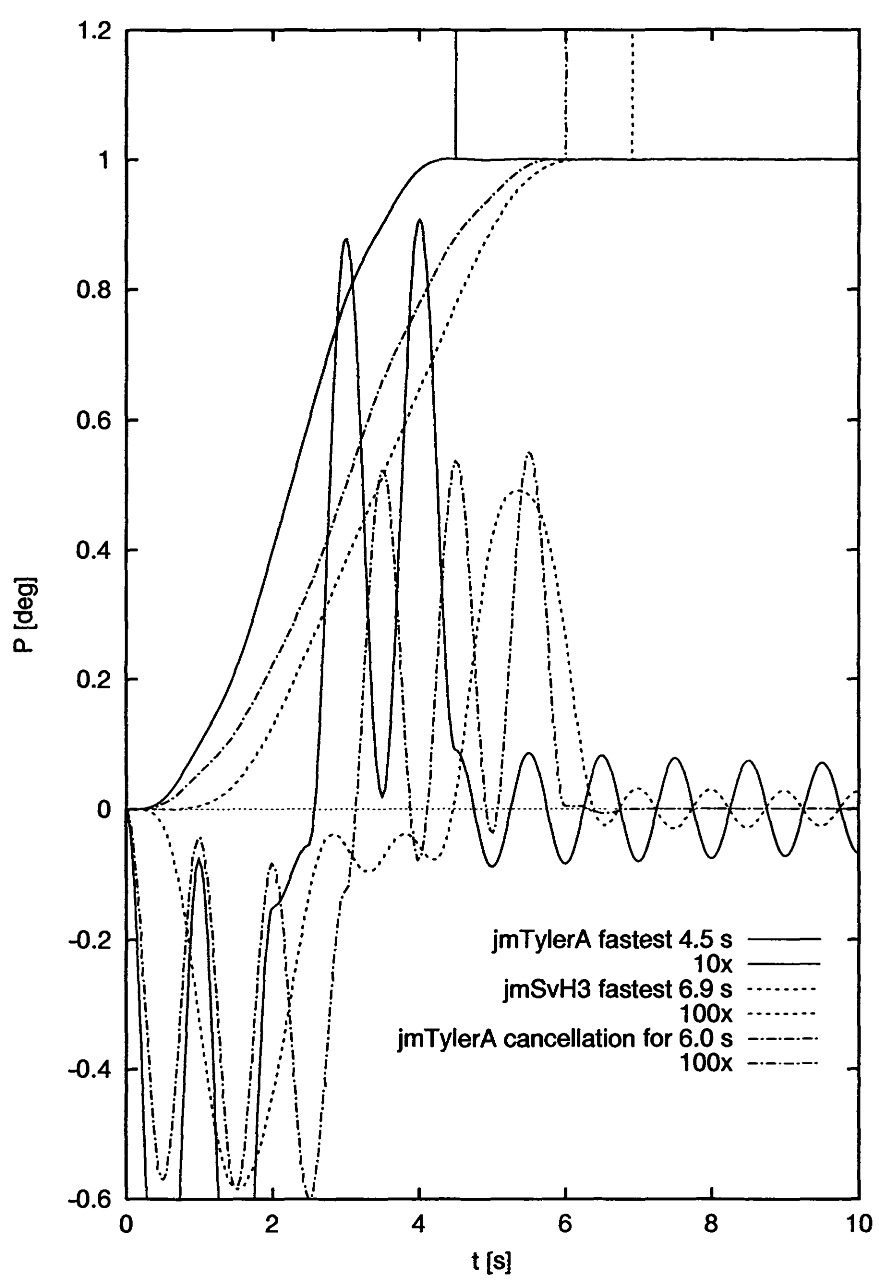

Figure 9: Jerk cancellation with precisely timed ($T = 6P$) `jmTylerA` damped trajectory

to choose a jerk value such that duration of acceleration impulse is a multiple of fundamental period.. and oscillations cancel out."

Other people associated with the GBT project were thinking the same general idea. In 1995, von Hoerner published a GBT Memo [vH95] in which he demonstrated vibration cancelling for step motions with durations of an even number of vibration cycles. Figure 9 is similar to a simulation described by von Hoerner [vH95, Fig.2] in which the duration of the trajectory is six times the vibration period (von Hoerner's case was undamped, whereas this one has some structural damping). The notable fact is that the vibration level after the precisely timed trajectory completes is much less than an arcsecond – it is effectively *zero*! Two

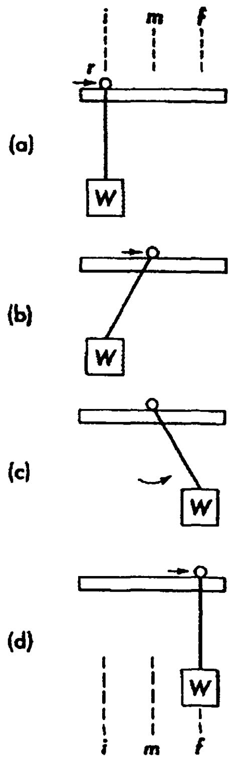

Figure 10: A travelling-crane version of Smith's 'Posicast' concept[9]

trajectories from Figure 7 [p.23] are plotted again here as a reference. The `jmTylerA` trace is the fastest possible trajectory (4.5 s) for the parameters of the problem. We see that its jerkiness excites vibrations during the motion which persist after the trajectory ends. A subsequent memo by von Hoerner [vH96] advocated use of higher-order $\sin^n t$ acceleration trajectories analogous to those advocated by Tyler to further minimize jerk. The figure shows an example of a trajectory computed with the `jmSvH3` function ($\sin^3 t$); we see that although it excites less vibration than does `jmTylerA` (note 100× scale for residuals versus 10×), it cannot compete with the exact-timing technique, which completely *cancels* the vibrations.

In September 1998 the author read Andersen's MMA Memo 231 [And98], which is concerned with the problem of minimizing vibrations in the ALMA [Atacama Large Millimeter Array] antennas when they make "fast switching" motions, 1.5° in 1.5 s. (The ALMA problem is mathematically identical to the GBT problem, of course.) Andersen showed a simulation of a step trajectory computed as a convolution with several impulses, and advocated use of this sort of technique in the ALMA project to improve response time by cancelling vibrations. Andersen cited a considerable number of papers on impulse cancellation in his memo. Although most of these papers were not readily available to the author, it was easy to do a Web search on several of the author names, which led to retrieval of copies of many of their papers (with their own citations) and thereby to discovery of the full literature and history of the subject. In particular, the author first learned of the "Posicast" algorithm from these citations.

The earliest published reference to the idea of using impulse cancellation for control of machinery was by Smith [Smi57].[10] In addition to his 1957 paper, Smith also published a book in which the concept is thoroughly discussed; he says [Smi58, p.332] that 'the [required] control motion is like casting a fly; hence the name "positive cast," or *Posicast*, control.' In Figure 10 he illustrates the concept for the case of

[9]from [Smi58, Fig.10-18, p.332]

[10]The Hildebrand book cited earlier [Hil62, pp.68-71] was a revision of an earlier Hildebrand book [Hil48]; the exact same impulse cancellation derivation appears in [Hil48], which shows that the basic concept existed before Smith's work.

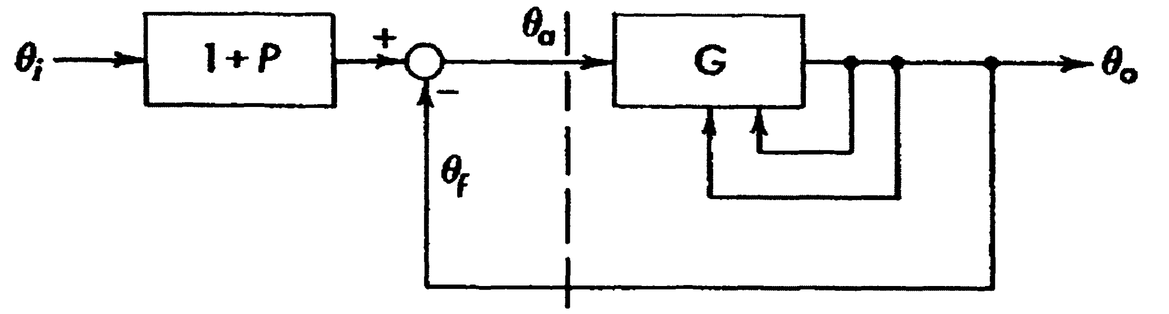

Figure 11: Smith's schematic of the "Posicast" system concept [11]

> "Posicast control of a pendulum position through force application to the suspension only. (a) Initial position. (b) Control computer takes half of a unit input step and move the support instantly to the mid-point m. Support remains fixed for one-half period. (c) Maximum swing of pendulum weight with support still fixed at mid-point m. (d) Support suddenly moved to the final position f directly above the maximum swing point." [Smi58, Fig.10-18 caption, p.332]

Smith's Posicast concept has been implemented for at least one astronomical application: the secondary mirror of an infrared telescope was being driven to "chop" the beam between two points on the sky and vibrations were being excited in the telescope structure. A Posicast controller was implemented, and it greatly reduced these vibrations [Set94].

There are two simple forms of the general impulse-cancellation idea, which Smith called "half-cycle" and "quarter-cycle" Posicast. The usual Posicast implementation is the half-cycle, in which there are two impulses separated by $P/2$ which are convolved with the input function (`jmPosicastTrajectory()` does this). Smith [Smi58, Sect.10-20,p.339,"One-quarter-cycle Response"] showed that three impulses, an initial positive plus a negative at $P/8$ plus a final positive, with a total duration of $P/4$ can also be used; this is "quarter-cycle Posicast" ([Set94] describes a quarter-cycle implementation). The author regards the even-cycle technique [vH96] as yet another variation on Posicast, and suggests "double-cycle" as a good name. The quarter-cycle technique requires impulses with amplitude greater than unity (1.707); in many real systems this option will not be available. The double-cycle technique is slow and somewhat inflexible (it implicitly assumes that the last half of the trajectory duplicates the first half). Half-cycle Posicast offers the best combination of simplicity, generality and performance, and so it was chosen for the `jmPosicastTrajectory` implementation described in the next section of this report.

The previous discussion in this section has a tutorial character, because it is intended to convey intuitive concepts to telescope control-system programmers and astronomers who are unlikely to be familiar with this surprising subject. A discussion intended for servo engineers would be expressed in the mathematical notations of their discipline, and would be more general. For readers who want such a description, the author offers the following quotation to convey the key idea:

> "The frequency response [of Posicast] can also be interpreted from the s-plane plot.. The Posicast compensating section $1+P$ [see Figure 11 [p.27]] has an infinite column of complex zeroes spaced at odd integers along the frequency scale. The lowest-frequency complex zeros are made to coincide with the resonant poles. They cancel the poles completely from a frequency-response viewpoint.." [Smi58, p.334]

Achieving low residual vibration levels with Posicast depends on accurate knowledge of period P and damping ratio ζ. In the case of the GBT, we will have a number of accelerometers and a 2-axis "quadrant detector"

[11] from [Smi58, Figure 10-24a, p.338] "Posicast control of a lightly damped feedback system.. Statement of best control as an operation on the input signal.."

which will be used to monitor the vibrational states of the important modes of the structure continuously; we will know P and ζ at all times and therefore Posicast will be the optimum algorithm for our application. One possible motivation for extensions to Smith's Posicast idea is that these two critical parameters are not known well enough (or else they vary) in many other systems. Readers interested in exploring extensions and generalizations of "command shaping" along these lines are advised to consult the following URLs:[12]

```
http://www-mit.mit.edu/shaping/www/dates.html
http://www-mit.mit.edu/shaping/www/onlinepapers.html
http://www-mit.mit.edu/shaping/www/othershaping.html
```

3.1 jmPosicastTrajectory

The argument list of this C function is identical to that of `jmCalcTrajectory` (Section 2.1 [p.4]) except for the addition of arguments `P[]` and `zeta[]`:

```
enum jmErrorCodes jmPosicastTrajectory(     /* returns enum code on error    */
                          enum   jmCalc    mode, /* jmFastest,jmSpecifyTime,..    */
                          enum   jmFunct funct, /* jmTylerA,jmTylerB,..          */
                          double             dt, /* time step, e.g. 0.1 s         */
                          int             nAxes, /* num axes for p0[],a_max[],etc */
                          double           p0[], /* initial position              */
                          double           v0[], /* initial velocity              */
                          double           pf[], /* final position                */
                          double           vf[], /* final velocity                */
                          double           af[], /* final acceleration (jmTylerC) */
                          double             tf, /* time for pf+vf+af target traj */
                          double        v_max[], /* +/- limit for velocity        */
                          double        a_max[], /* +/- limit for acceleration    */
                          double            P[], /* period       of resonant mode */
                          double         zeta[], /* damping ratio of resonant mode */
                          double *timeInterval, /* ptr to total trajectory time  */
                          struct jmPS     **pTS) /* return ptr to private struct  */
```

The `P[]` argument in the current implementation of this algorithm supports only one period per axis. The Posicast concept works for more than one period [Smi58, p.344, "Every pole of the system must have a zero superimposed"]. It should be feasible to add a `P2[]` argument to `jmPosicastTrajectory()` to specify a second period for each axis, and to implement it by a convolution of the first-period trajectory by a pair of impulses separated by $P_2/2$ (a `zeta2[]` argument would also be added, of course.) This development effort has been deferred until we have operational experience with the current single-period implementation.

```
/* jmPosicastTrajectory.c --- Command-shaping trajectory generator algorithm
   1999-08-25: D.Wells, NRAO-CV, cloned from jmCalcTrajectory.c
   1999-09-04: velocity/acceleration before/after corrections added.
   1999-12-14: Posicast for accelerated-targets works now
   1999-12-17: jmFastest and nAxes>1 work now
   */
```

[GNU General Public License copyright notice omitted;see `http://www.gnu.org/copyleft/gpl.html`*]*

```
#include "jmInclude.h"
```

[12]Otto J. M. Smith has a personal Web page, which mentions his invention of the Posicast technique, but which is more concerned with promoting his technique of operating three-phase motors on single-phase power:

`http://www.wenet.net/~ottojms/indexd.html#history`

```
enum jmErrorCodes jmPosicastTrajectory(      /* returns enum code on error      */
                      enum   jmCalc    mode, /* jmFastest,jmSpecifyTime,..      */
                      enum   jmFunct  funct, /* jmTylerA,jmTylerB,..            */
                      double             dt, /* time step, e.g. 0.1 s           */
                      int             nAxes, /* num axes for p0[],a_max[],etc   */
                      double           p0[], /* initial position                */
                      double           v0[], /* initial velocity                */
                      double           pf[], /* final position                  */
                      double           vf[], /* final velocity                  */
                      double           af[], /* final acceleration (jmTylerC)   */
                      double             tf, /* time for pf+vf+af target traj   */
                      double        v_max[], /* +/- limit for velocity          */
                      double        a_max[], /* +/- limit for acceleration      */
                      double            P[], /* period         of resonant mode */
                      double         zeta[], /* damping ratio of resonant mode  */
                      double *timeInterval, /* ptr to total trajectory time    */
                      struct jmPS     **pTS) /* return ptr to private struct    */
{
  int       status, i, j, k, k_trial, k_first, k_total, i_save;
  double    T, t, p, t_start, p_start, p_half, v, a, T_save, dtp, td,
            Tq, pe[MAXNAXES], ve[MAXNAXES], ae[MAXNAXES], t_impulse,
            t_before, t_after, delta_time[2], impulse[2],
            p0q[1], v0q[1], pfp[1], vfp[1], pfpq[1], vfpq[1];
  const double eps = 1e-8;
  struct jmPS *q, qTrial;

  if (mode != jmTrial) {
    if ((0>=nAxes)||(nAxes>MAXNAXES)) return(jmpNAxesError);     E21
    if (dt <= 0.)                     return(jmpNegDtError);      E22
    if (v_max[0] <= 0.)               return(jmpNegVmaxError);     E23
    if (a_max[0] <= 0.)               return(jmpNegAmaxError);     E24
    if (fabs(v0[0]) > v_max[0])       return(jmpV0BigError);      E25
    if (fabs(vf[0]) > v_max[0])       return(jmpVfBigError);      E26
    if (fabs(af[0]) > a_max[0])       return(jmpAfBigError);      E27
    if (P[0]    < 0.)                 return(jmpNegP0Error);      E28
    if (zeta[0] < 0.)                 return(jmpNegZetaError);     E29
  }
```

$\boxed{\text{Step 21}}$ **One axis, or more than one?**

Tyler's three-region algorithm and Smith's Posicast algorithm are for a mechanical motion in one variable, or "axis". If the problem has more than one axis our technique will be to first determine, by calls to this function for each axis in the `jmFastestTime` mode, which axis is going to be take the longest time, and then to call this function for each axis in the `jmSpecifyTime` mode (see Step 29 [p.33]).

```
  if (nAxes == 1) {
```

$\boxed{\text{Step 22}}$ **Is period argument nonzero?**

If period `P[0]` is zero, Posicast algorithm does not apply, and we can simply execute function `jmCalcTrajectory()` with `mode` and return whatever status it returns.

```
    if (P[0] == 0.) return(jmCalcTrajectory(mode, funct, dt, nAxes,
                                    p0, v0, pf, vf, af, tf, v_max, a_max,
                                    timeInterval, pTS));
```

What is the value of argument mode? The actual computation of a trajectory occurs only for the case where mode is jmSpecifyTime. For the jmFastest and jmFastestTime modes we *search* for the fastest feasible trajectory; code for these modes appears later in this C function, at Step 27 [p.32].

```
switch(mode) {
case(jmSpecifyTime):
```

Mode jmSpecifyTime will malloc() and initialize an instance of the private data object struct jmPS and return a pointer to it. It will then calculate the trajectory and return it in this struct.

```
  if ((status = jmAlocTrajectory(pTS, jmAlocAndInit))
      != jmNoError) return(status);
case(jmTrial):  /* private mode used by jmFastestTime & jmFastest */
case(jmNoAloc): /* private mode used by nAxes>1 */
  T = *timeInterval;
  (*pTS)->n_T = floor((T / dt) + eps);
  /* T must be a positive integral multiple of dt: */
  if ( (T <= 0.) ||
       ((fabs(T-(dt*((*pTS)->n_T)))/dt)>eps) ) return(jmpBadTError);     E30
```

Step 23 **Prepare timing and impulse parameters**

The Posicast method uses two impulses of nearly equal amplitude separated by half the period: the amplitude of the trajectory is divided into two parts and the second part is delayed by half a cycle of the resonance. The relative weights of the two parts depend on the damping [Smi58, p.332–3]. If $u(t)$ is the input trajectory, the output is computed as $Au(t) + Bu(t - P/2)$ with $A+B = 1$ and $\frac{B}{A} = e^{-\pi\zeta}$. This implementation uses $0 \pm P/4$ for the times of the two impulses; it does this by setting delta_time[] to the time offsets of two impulses in units of the period P[0], i.e. to (-0.25,+0.25). The weights of the two impulses (A, B) will be in impulse[]. Variables t_before and t_after are to be set to the lead and lag times of the convolution (they are multiples of dt). The total time specified for the trajectory will be adjusted by the sum of these times, and jmCalcTrajectory() will be called to produce the trajectory for forming the weighted sum.

```
  /* convention is that plus time is ahead in the trajectory: */
  delta_time[0] = +0.25;
  delta_time[1] = -0.25;
  impulse[0] = 1. / (1. + exp(-PI*zeta[0]));
  impulse[1] = 1.0 - impulse[0];
  t_before = ceil((+delta_time[0]*P[0])/dt)*dt;
  t_after  = ceil((-delta_time[1]*P[0])/dt)*dt;
```

Step 24 **Compute the basic (shorter duration) trajectory**

We will compute trajectory with dtp, which is dt divided by an integer. Tq is the duration of the trajectory we will compute; note that Tq will be multiple of dt and also multiple of dtp.

```
  dtp = dt / 4.;
  Tq = (T - t_before - t_after);
  if (jmTrial && (Tq <= 0.)) return(jmFailed);
  /* Adjust starting position for t_before: */
  pOq[0] = p0[0] + v0[0]*t_before;
  v0q[0] = v0[0];
  /* Get final position: */
  if (tf == 0.0) {
    pfp[0] = pf[0];
    vfp[0] = vf[0];
  } else { /* tf!=0 is the target-trajectory case: */
```

```
    td = (T - tf);
    pfp[0] = pf[0] + vf[0]*td + 0.5*af[0]*(td*td);
    vfp[0] = vf[0] + af[0]*td;
}
/* Adjust ending position for t_after: */
td = -t_after;
pfpq[0] = pfp[0] + vfp[0]*td + 0.5*af[0]*(td*td);
vfpq[0] = vfp[0] + af[0]*td;
```

Special adjustment: The Posicast algorithm with $a_f \neq 0$ (accelerated target, using jmTylerC function) produces a position shift in the output trajectory due to forming a weighted sum, approximately a bisector, which does not lie on the accelerated target trajectory. This can be compensated (at least for the case of damping $\zeta = 0$) by adjusting the commanded final position pfp[0] by the amount of the systematic error, using[13]

$$p_{\text{final}} \rightarrow p_{\text{final}} - \frac{a_{\text{final}} P^2}{32} :$$

```
pfpq[0] -= (af[0] * P[0]*P[0]) / 32.;
```

Compute trajectory with duration Tq and spacing dtp.

```
q = &qTrial;
status = jmAlocTrajectory(&q, jmInitOnly); /* initialize struct qTrial */
if ((status = jmCalcTrajectory(mode, funct, dtp, nAxes,
                               p0q, v0q, pfpq, vfpq, af, 0.,
                               v_max, a_max,
                               &Tq, &q)) != jmNoError) return(status);
if (mode == jmTrial) return(status); /* private success return */
q->t_first = t_before; /* starting time of trajectory qTrial is not zero */
```

$\boxed{\text{Step 25}}$ Allocate space for arrays of PVA results

```
/* malloc space for pva arrays: */
if (((*pTS)->pP[0] = (double *)
    calloc((*pTS)->n_T, sizeof(double)))==NULL) return(jmpPAlocError);    [E31]
if (((*pTS)->pV[0] =  (double *)
    calloc((*pTS)->n_T, sizeof(double)))==NULL) return(jmpVAlocError);    [E32]
if (((*pTS)->pA[0] = (double *)
    calloc((*pTS)->n_T, sizeof(double)))==NULL) return(jmpAAlocError);    [E33]
(*pTS)->nAxes   = 1;
(*pTS)->t_first = 0.;
(*pTS)->dt      = dt;
(*pTS)->v0[0]   = v0[0];
(*pTS)->pf[0]   = pfp[0];
(*pTS)->vf[0]   = vfp[0];
(*pTS)->af[0]   = af[0];
(*pTS)->x[0]    = q->x[0];
(*pTS)->y[0]    = q->y[0];
(*pTS)->t1      = q->t1;
(*pTS)->t2      = q->t2;
(*pTS)->t3      = q->t3;
(*pTS)->a1[0]   = q->a1[0];
```

[13]The following Mathematica code computed the formula above:
```
y[t_] := p + v t + (a t^2)/2
z[t_] := b + v t + (a t^2)/2
p[t_] := (z[t-P/4] + z[t+P/4])/2
TeXForm[Simplify[Solve[p[t] == y[t], b]]]
```
This derivation does not account for the differing impulse amplitudes when $\zeta \neq 0$; development of this refinement is deferred until we have operational experience with the current implementation.

```
(*pTS)->v2[0]    = q->v2[0];
(*pTS)->a3[0]    = q->a3[0];
```

Step 26 **Compute output trajectory as a convolution with the two impulses**

The output trajectory will begin with the $t = 0$ point of the trajectory computed above (which has t_first = t_before. It will have the same trajectory added to it with a half-period shift. The output trajectory will extend by $\frac{1}{4}P$ beyond the computed trajectory so that the shifted trajectory will osculate to the specified final trajectory. The computation proceeds by half-steps ($\Delta T/2$):

```
for (i = 0, j = 0; i < (2 * ((*pTS)->n_T) + 1); i++) {
  t = (*pTS)->t_first + i*(dt/2.);
  /* Form the sum of the trajectory shifted by the
     delta_time[] values and weighted by the impulse[] values: */
  for (k = 0, p = 0.; k < 2; k++) {
    t_impulse = t + delta_time[k] * P[0];
    if ((status = jmEvalTrajectory(q, t_impulse, pe, ve, ae)) != jmNoError) {
      return(status);
    }
    p += impulse[k] * pe[0];
  }
```

The piecewise-parabolic approximation to the trajectory is computed by evaluating the trajectory at half-steps and setting the PVA such that in each interval dt the PVA parabola passes through the starting value, the half-step value and the final value.

```
  if ((i % 2) == 0) {          /* at whole ticks: */
    if (i > 0) {
      v = (-3.*p_start +4.*p_half -1.*p) / dt;
      a = ( 4.*p_start -8.*p_half +4.*p) / (dt*dt);
      ((*pTS)->pP[0])[j] = p_start;
      ((*pTS)->pV[0])[j] = v;
      ((*pTS)->pA[0])[j] = a;
      j++;
    }
    t_start = t;
    p_start = p;
  } else {                     /* at half ticks: */
    p_half = p;
  }
}
return(0); /* exit from the nAxes=1/jmSpecifyTime case    */
/* (also exit here from jmNoAloc private mode) */
```

Step 27 **Find the minimum possible time for a single axis (begin jmFastestTime section)**

The *binary search algorithm* which follows should be an exact clone of the algorithm at Step 11 [p.13] of function jmCalcTrajectory(). See the comments there for the details of this algorithm. Mode jmFastest returns not only the minimum time but also the actual trajectory, whereas jmFastestTime determines only the minimum feasible value for T and returns it in argument *timeInterval (with NULL returned for the trajectory pointer).

```
case(jmFastestTime):
case(jmFastest):
  q = &qTrial;
  status = jmAlocTrajectory(&q, jmInitOnly);
```

```
    k_trial = 1;
    k_first = 1;
    k_total = 0;
    k = 0;
    while (k_trial > 0) { /* loop halts when division by two yields zero. */
      T = (k + k_trial) * dt;
      status = jmPosicastTrajectory(jmTrial, funct, dt, nAxes,
                                    p0, v0, pf, vf, af, tf, v_max, a_max,
                                    P, zeta, &T, &q);
      if ((status != jmNoError) &&
          (status != jmFailed))
        return(status); /* Unexpected nonzero status; return it */
      k_total++;
      if (k_total > 30) return(jmpBigKtotalError);        E34
      if (status == jmFailed) {
        if (k_first) {
          k_trial *= 2; /* First pass: keep doubling until succeed */
          if (k_trial >= 1073741824) return(jmpKbigFailure); /* 2^30 */     E35
        } else {
          k = k + k_trial;
          k_trial /= 2;
        }
      } else { /* trial time succeeded: */
        T_save = T;
        if (k_first) {
          k = k + k_trial / 2; /* First pass: set up binary search */
          k_trial /= 4;
          k_first = 0;
        } else {
          k_trial /= 2;
        }
      }
    }
    *timeInterval = T_save;        /* return optimum time */
    if (mode == jmFastestTime) {
      *pTS = NULL;
      return(jmNoError); /* exit from the nAxes=1/jmFastestTime case */
    }
```

$\boxed{\text{Step 28}}$ **Optimum T is known, can compute trajectory**

```
    status = jmPosicastTrajectory(jmSpecifyTime, funct, dt, nAxes,
                                  p0, v0, pf, vf, af, tf, v_max, a_max,
                                  P, zeta, timeInterval, pTS);
    return(status); /* exit from the nAxes=1/jmFastest case */

  default:      /* unrecognized jmMode value; return error code */
    return(jmpUnrecModelError);      E36
  }             /* end of nAxes=1/jmMode switch-statement */
```

$\boxed{\text{Step 29}}$ **Generate trajectories for `nAxes>1` situations**

In the multiple trajectory case, if we are in `jmFastest` mode, we loop over the axes with `jmFastestTime` mode to find which axis will take the longest time, and then we loop again in `jmSpecifyTime` mode to calculate the actual trajectories. (If we are in `jmFastestTime` mode, we return the time without computing the trajectories.) If we are called with `jmSpecifyTime` mode we can skip the first loop.

```
} else {
  switch(mode) {
  case(jmFastest):
  case(jmFastestTime):
    T_save = 0.;
    i_save = 0;
    q = &qTrial;
    for (i = 0; i < nAxes; i++) {
      if ((status = jmPosicastTrajectory(jmFastestTime, funct, dt, 1,
                                          &p0[i], &v0[i], &pf[i], &vf[i],
                                          &af[i], tf, &v_max[i], &a_max[i],
                                          &P[i], &zeta[i],
                                          &T, &q)) != jmNoError) return(status);
      if (T > T_save) {
        T_save =  T;
        i_save = i;
      }
    }
    *timeInterval = T_save;
```

Mode jmFastestTime will return here; jmFastest continues as though mode were jmSpecifyTime:

```
    if (mode == jmFastestTime) {
      *pTS = NULL;
      return(jmNoError);
    }
```

Create instance of private struct, do jmSpecifyTime:

```
  case(jmSpecifyTime):
    if (mode == jmSpecifyTime) i_save = 0;
    if ((status = jmAlocTrajectory(pTS, jmAlocAndInit))!=jmNoError) return(status);
    q = &qTrial;
    status = jmAlocTrajectory(&q, jmInitOnly);
    for (i = 0; i < nAxes; i++) {
      if ((status = jmPosicastTrajectory(jmNoAloc, funct, dt, 1,
                                          &p0[i], &v0[i], &pf[i], &vf[i],
                                          &af[i], tf, &v_max[i], &a_max[i],
                                          &P[i], &zeta[i], timeInterval,
                                          &q)) != jmNoError) return(status);
      if (i == i_save) {
        (*pTS)->slowest[i] = (mode == jmSpecifyTime) ? 0 : 1;
        (*pTS)->nAxes      = nAxes;
        (*pTS)->n_T        = q->n_T;
        (*pTS)->t_first    = q->t_first;
        (*pTS)->dt         = q->dt;
        (*pTS)->t1         = q->t1;
        (*pTS)->t2         = q->t2;
        (*pTS)->t3         = q->t3;
      } else {
        (*pTS)->slowest[i] = 0;
      }
      (*pTS)->v0[i] = q->v0[0];
      (*pTS)->pf[i] = q->pf[0];
      (*pTS)->vf[i] = q->vf[0];
      (*pTS)->af[i] = (funct == jmTylerC) ? q->af[0] : 0.;
```

Move pointers to PVA vectors for axis i from the internal private structure qTrial to the *pTS struct:

```
        (*pTS)->pP[i] = q->pP[0];
        (*pTS)->pV[i] = q->pV[0];
        (*pTS)->pA[i] = q->pA[0];
      }
      return(jmNoError);
```

Unrecognized mode:

```
    default:
      return(jmpUnrecMode2Error);     E37
    } /* end of nAxes>1/jmMode switch statement */
  }
}
```

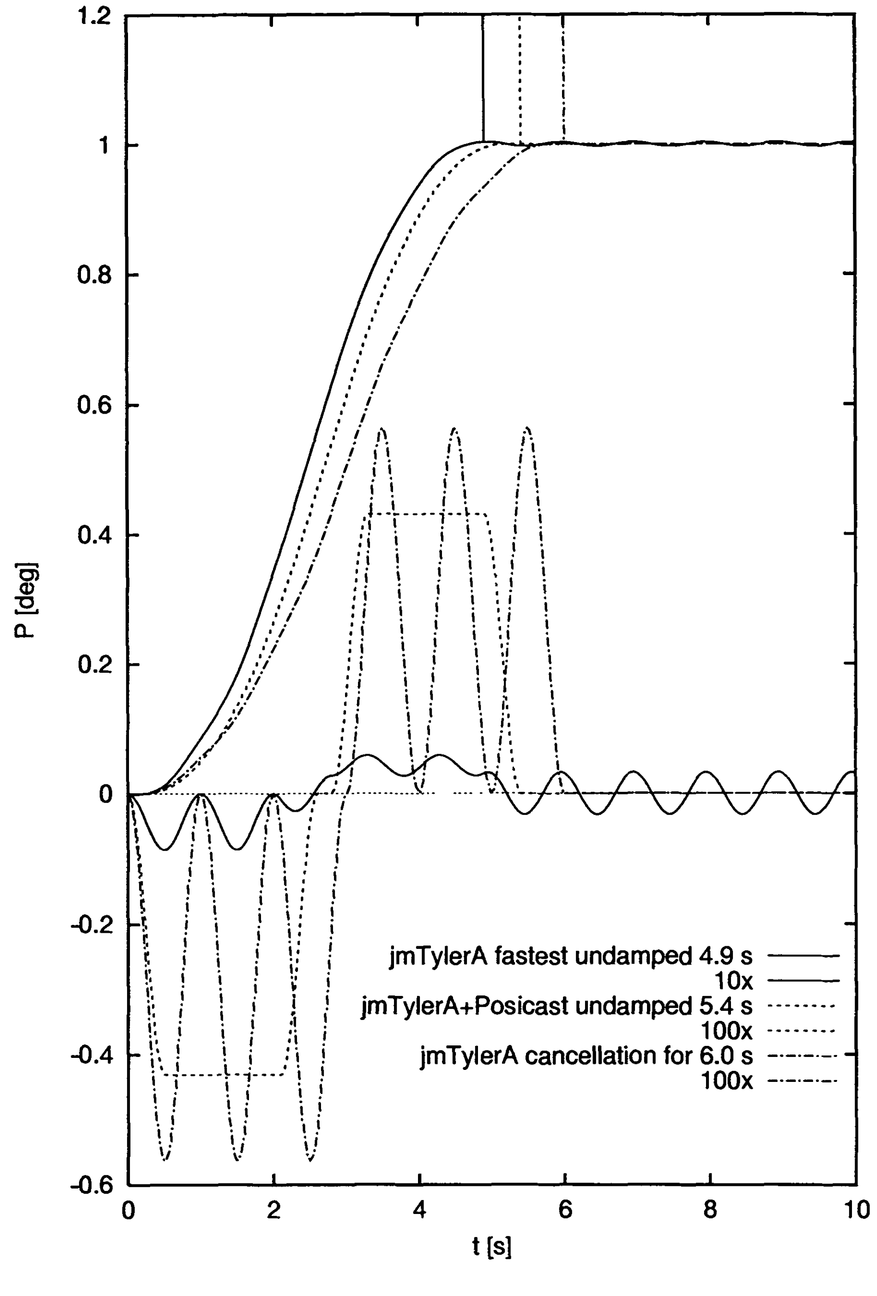

Figure 12: Jerk-cancellation with fast `jmTylerA`+Posicast undamped trajectory

3.2 Numerical simulations of Posicast jerk-cancelling trajectories

Figure 12 shows a fast Posicast trajectory compared with the fastest possible trajectory and with the six-cycle cancelling trajectory. We see that, like the six-cycle case, the final vibration level for the Posicast trajectory is effectively *zero*. We also see that the Posicast trajectory displays *no* vibration even during its acceleration and deceleration phases. The duration of the Posicast trajectory is only one-half cycle longer than the fastest possible trajectory; in this case it is a half second faster than the six-cycle cancelling trajectory. It is fascinating to note that the differences between the shape of the Posicast trajectory and the shapes of the other trajectories are nearly imperceptible, and that it has no peculiar characteristic identifiable by eye.

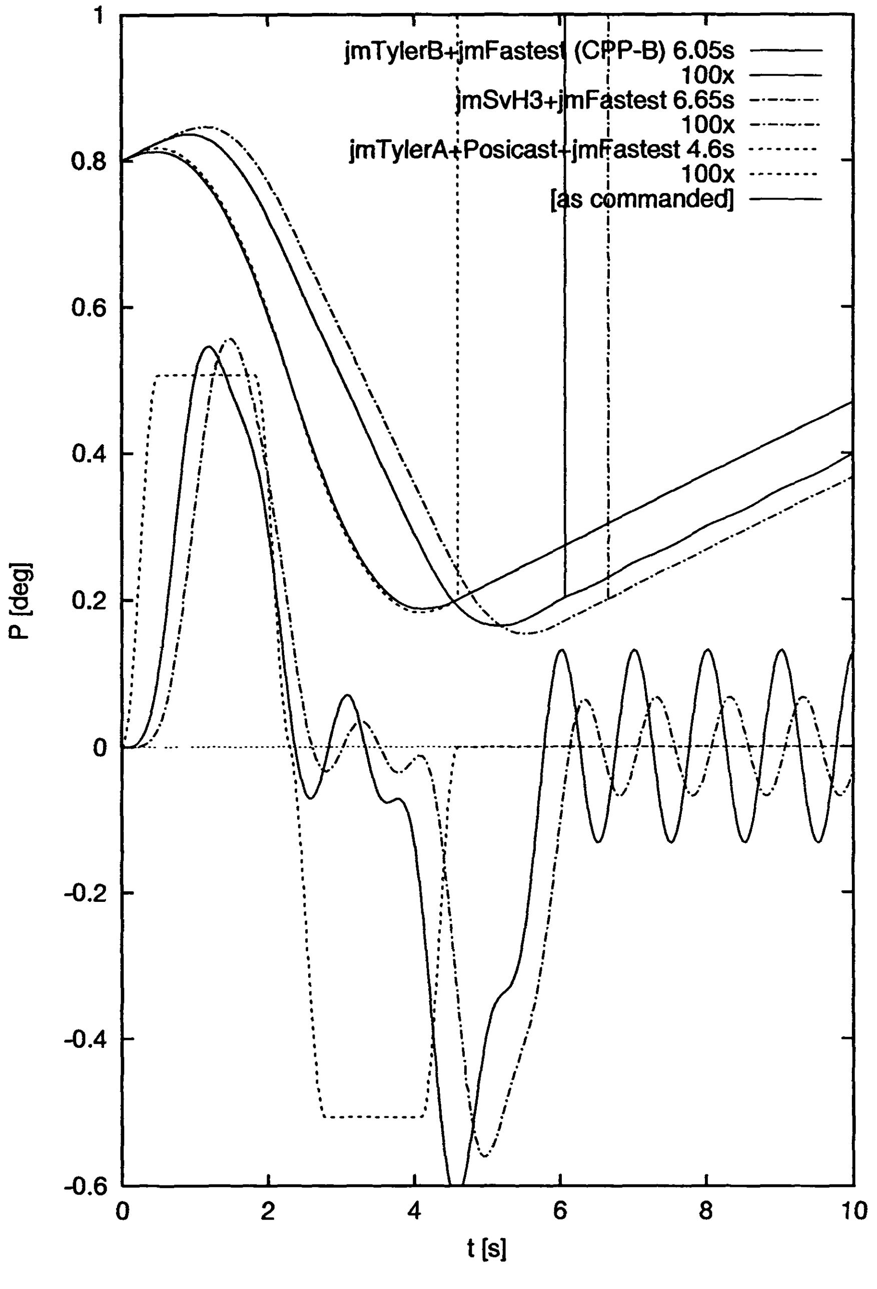

Figure 13: Three flyback-trajectories – jerk-cancellation can be faster than jerk-minimization

Figure 13 is another example of a Posicast trajectory. In this case we suppose that we are doing a raster scan with the telescope, and that we want to move from the end of one row to the beginning of the next row (like the "flyback" of a television monitor). Initially the telescope is at the end of a row and is moving with a small positive velocity (0.05°/s). It must perform a negative acceleration, reach a high negative velocity, perform a positive deceleration and arrive at the beginning of the row (0.6° length) again moving at the small positive velocity. We see that the `jmTylerB`("CPP-B") trajectory does this with a residual vibration level of about ±5 arcsec (0.13 deg × 3600 arcsec/deg × 0.01 = 4.7 arcsec). The `jmSvH3` ($\sin^3 t$) function improves on this, with only about half as much vibration, but the `jmTylerA`+Posicast trajectory is *perfect*, and it is the fastest trajectory shown! The figure shows two different Posicast trajectories: the solid curve is the

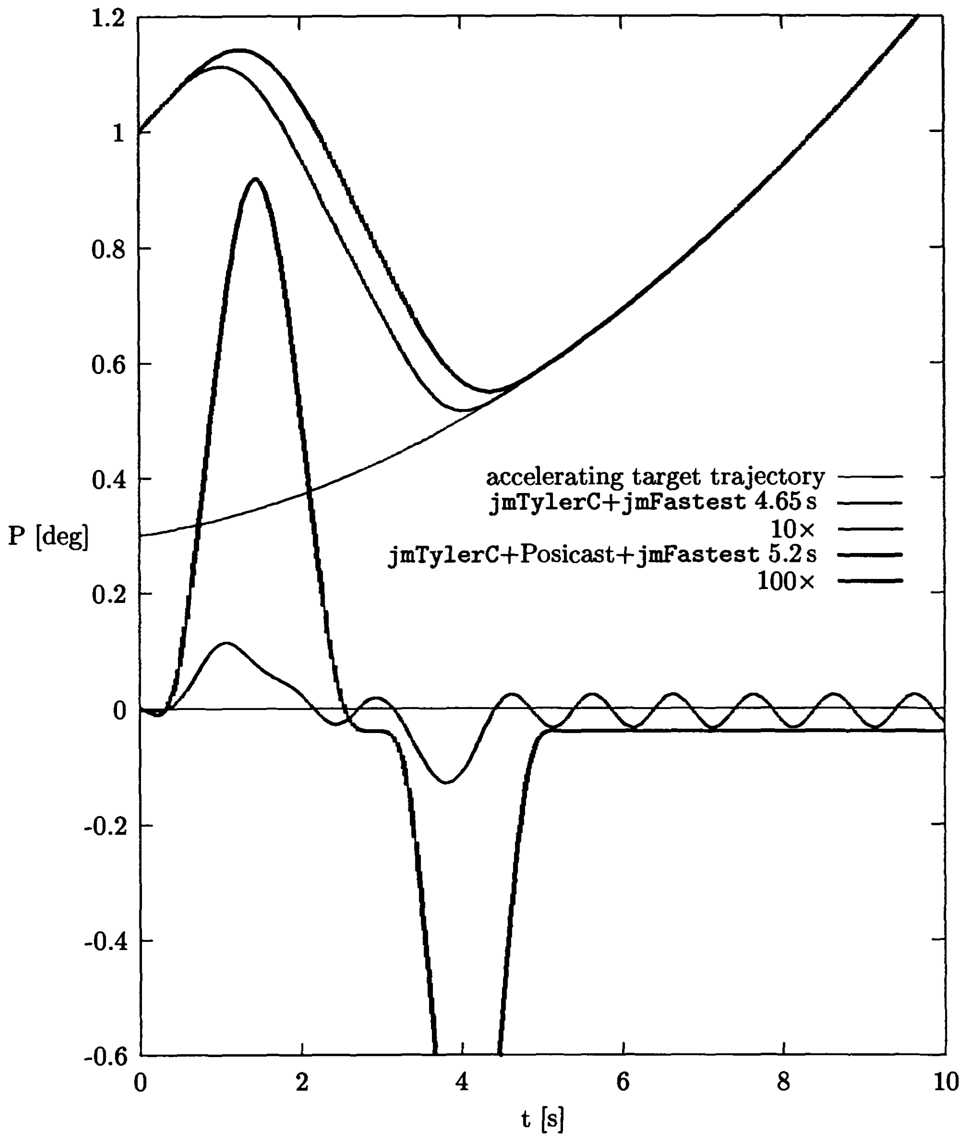

Figure 14: Jerk-cancellation while matching accelerated target trajectory (jmTylerC+Posicast)

trajectory as it is commanded, and the dashed curve is the response of the system; the difference between the two curves is plotted with 100× enlargement as a dashed-line curve below.

Figure 14 demonstrates an application of the jmTylerC function. The initial and final positions and velocities are somewhat similar to those in Figure 13, but the important difference is that we have specified a final acceleration $a_f = 0.015\,\text{deg/s}^2$ and have set argument tf $\neq 0$ to activate the target trajectory mode. The constant offset in the trajectory error is simply due to constant acceleration applying force to the spring constant (the pure jmTylerC vibrating trace with magnification 10× is offset by exactly the same amount).[14] For non-equatorial-mount radio antennas, whether acquiring fast-moving spacecraft or slewing to sidereal velocity astronomical sources, almost all targets have accelerating target trajectories, so this simulation illustrates the usual mode of operation of this jerk-minimization trajectory generator software.

[14]Note that if an accelerated version of jmTylerA were implemented in jmCalcTrajectory, the Posicast algorithm could osculate to the target trajectory even more quickly than does the jmTylerB profile used in this simulation.

A jmStrError – descriptions of jm-package error codes

```
void jmStrError(enum jmErrorCodes error, /* input error code from jm package */
                char            msgBuf[]) /* array into which msg is written  */
```

```
void jmPerror(enum jmErrorCodes code, /* input error code from jm package */
              char          userMsg[]) /* short id string supplied by user */
```

```
/* jmStrError.c --- strerror() and perror() style error decoding for
   Jerk-minimizing trajectory generator algorithm.
   1999-11-??: original by J.Brandt, NRAO-GB
   1999-12-08: adopted and extended by D.Wells, NRAO-CV
*/
```

[GNU General Public License copyright notice omitted;see `http://www.gnu.org/copyleft/gpl.html`*]*

```
#include "jmInclude.h"
#include <stdio.h>
void jmStrError(enum jmErrorCodes error, /* input error code from jm package */
                char            msgBuf[]) /* array into which msg is written  */
{
  switch(error)
    {
```

Step 41 **The normal-return status codes**

The `jmcFailed` code is returned by `jmCalcTrajectory()` in the `jmFastest` mode when the user specifies a total trajectory time `T` which is too short for the combination of position, velocity and acceleration parameters which are supplied. This code can be returned by calls to `jmPosicastTrajectory()`, because that function calls `jmCalcTrajectory()`. (Code `jmcFailed` is also heavily used internally when `jmCalcTrajectory()` is trying values of `T` in the binary search to find the fastest possible trajectory.)

```
CASF(jmNoError,        "Normal successful return");
CASE(jmFailed,         "Specified trajectory is not feasible");              see E11 on p.8
```

Step 42 **Codes for input argument errors**

Function `jmCalcTrajectory()` has many numerical input arguments which must conform to a set of obvious conditions. Function `jmPosicastTrajectory()` contains a set of input argument tests and codes which are identical to those used in `jmCalcTrajectory()`. *Note: prefixes* `jmc` *and* `jmp` *indicate codes returned by* `jmCalcTrajectory()` *and* `jmPosicastTrajectory()` *respectively.*

```
CASE(jmcNAxesError,    "Number of axes incorrect");                          see E1 on p.6
CASE(jmcNegDtError,    "dt is less than zero");                              see E2 on p.6
CASE(jmcBadTError,     "Time is not an integral multiple of dt");            see E9 on p.7
CASE(jmcNegVmaxError,  "Vmax is less than zero");                            see E3 on p.6
CASE(jmcV0BigError,    "Initial velocity is greater than Vmax");             see E5 on p.6
CASE(jmcVfBigError,    "Final velocity is greater than Vmax");               see E6 on p.6
CASE(jmcNegAmaxError,  "Amax is less than zero");                            see E4 on p.6
CASE(jmcAfBigError,    "Final acceleration is greater than Amax");           see E7 on p.6

CASE(jmpNAxesError,    "Number of axes incorrect");                          see E21 on p.29
CASE(jmpNegDtError,    "dt is less than zero");                              see E22 on p.29
CASE(jmpBadTError,     "Time is not an integral multiple of dt");            see E30 on p.30
CASE(jmpNegVmaxError,  "Vmax is less than zero");                        see E23 on p.29
CASE(jmpV0BigError,    "Initial velocity is greater than Vmax");         see E25 on p.29
CASE(jmpVfBigError,    "Final velocity is greater than Vmax");           see E26 on p.29
CASE(jmpNegAmaxError,  "Amax is less than zero");                        see E24 on p.29
CASE(jmpAfBigError,    "Final acceleration is greater than Amax");       see E27 on p.29

CASE(jmpNegP0Error,    "Period is less than zero");                      see E28 on p.29
CASE(jmpNegZetaError,  "Damping factor is less than zero");              see E29 on p.29
```

Step 43 **Malloc errors**

Errors of this type should not normally occur in the jerk-minimizing trajectory package; if they do occur it indicates something drastically wrong in the operating environment.

```
CASF(jmaPSAlocError,   "malloc error allocating A array");

CASE(jmcAAlocError,    "jmAlocTrajectory failed");                        see E15 on p.9
CASE(jmcPAlocError,    "malloc error allocating P array");                see E13 on p.9
CASE(jmcVAlocError,    "malloc error allocating V array");                see E14 on p.9

CASE(jmpAAlocError,    "jmAlocTrajectory failed");                       see E33 on p.31
CASE(jmpPAlocError,    "malloc error allocating P array");               see E31 on p.31
CASE(jmpVAlocError,    "malloc error allocating V array");               see E32 on p.31
```

Step 44 **'Impossible' errors**

The following codes should never occur, or at least that is the theory! The author should be informed at `dwells@nrao.edu` if these errors are seen.

```
CASE(jmcKbigFailure,     "k_trial reached 2**30");                       see E18 on p.14
CASE(jmcBigKtotalError,  "k_total reached 30");                          see E17 on p.14
CASE(jmcImpossError,     "Phase-space mapping error");                    see E12 on p.9
CASF(jmePNullError,      "Null trajectory");
CASF(jmeTSNullError,     "Null trajectory Object");
CASF(jmfNAxesError,      "unreasonable nAxes in private struct");
CASF(jmfTSNullError,     "private struct is NULL");
CASF(jmhTSNullError,     "private struct is NULL");
CASF(jmaUnrecModeError,  "Unrecognized mode value (e.g. not jmInitOnly)");
```

Step 45 **Enum errors**

The following codes should never occur because user code should include `jmInclude.h` which defines `enum jmErrorCodes`, and it should not be possible to compile code which calls `jmStrError` with a code which is not in the list.

```
CASE(jmcFunctEnumError, "Unrecognized function (e.g. not jmTylerB)");      see E10 on p.7
CASE(jmcUnrecFunctError,"Unrecognized function (e.g. not jmTylerB)");     see E16 on p.12
CASE(jmcUnrecMode1Error,"Unrecognized jmMode value (e.g. not jmFastest)"); see E19 on p.14
CASE(jmcUnrecMode2Error,"Unrecognized jmMode value (e.g. not jmFastest)"); see E20 on p.16
CASE(jmpUnrecMode1Error,"Unrecognized jmMode value (e.g. not jmFastest)"); see E36 on p.33
CASE(jmpUnrecMode2Error,"Unrecognized jmMode value (e.g. not jmFastest)"); see E37 on p.35

default:
  sprintf(msgBuf, "unrecognized error number %d", error);
  break;
};
```

```
}
/* A perror() style message decoding scheme.
 */
void jmPerror(enum jmErrorCodes code, /* input error code from jm package */
              char        userMsg[]) /* short id string supplied by user */
{
  char errstring[JM_MAX_ERROR_MSG_LENGTH];

  jmStrError(code, errstring);
  fprintf(stderr, "%s : %s\n", userMsg, errstring);
}
```

B The include, enum and struct sections of jmInclude.h

```
#include "math.h"
#include "stdio.h"
#include "stdlib.h"

#define PI  3.1415926535897932
#define JM_MAX_ERROR_MSG_LENGTH 80

enum jmFunct {jmTylerA, jmTylerB, jmTylerC, jmSvH3, jmSvH4};

enum jmCalc  {jmFastest, jmFastestTime, jmSpecifyTime,      /* <--public  modes */
              jmTrial, jmNoAloc, jmAlocAndInit, jmInitOnly}; /* <--private modes */

enum jmErrorCodes {jmNoError = 0, jmFailed, jmaPSAlocError, jmcAAlocError,
                   jmcAfBigError, jmcBadTError,
                   jmcFunctEnumError, jmcImpossError, jmcNAxesError,
                   jmcNegAmaxError, jmcNegDtError, jmcNegVmaxError,
                   jmcPAlocError, jmcTSAlocError, jmcUnrecFunctError,
                   jmcUnrecMode1Error, jmcUnrecMode2Error,
                   jmcV0BigError, jmcVAlocError, jmcVfBigError,
                   jmePNullError, jmeTSNullError, jmfNAxesError,
                   jmfTSNullError, jmhTSNullError, jmpAAlocError,
                   jmpAfBigError, jmpBadTError, jmpEvalError,
                   jmpNAxesError, jmpNegAmaxError, jmpNegDtError,
                   jmpNegP0Error, jmpNegVmaxError, jmpNegZetaError,
                   jmpPAlocError, jmpTSAlocError, jmpUnrecMode1Error,
                   jmpUnrecMode2Error, jmpUnrecShapeError,
                   jmpV0BigError, jmpVAlocError, jmpVfBigError,
                   jmcKbigFailure, jmcAfNonzeroError, jmcBigKtotalError,
                   jmpKbigFailure, jmpBigKtotalError, jmaUnrecModeError};

/* Private data structure for the 'jm' package: */
#define MAXNAXES 6
struct jmPS {
  int    nAxes, n_T, slowest[MAXNAXES];
  double t_first, dt, v0[MAXNAXES], pf[MAXNAXES], vf[MAXNAXES], af[MAXNAXES],
         *pP[MAXNAXES], *pV[MAXNAXES], *pA[MAXNAXES];
  double t1, t2, t3, x[MAXNAXES], y[MAXNAXES],      /* <--debug variables */
         a1[MAXNAXES], v2[MAXNAXES], a3[MAXNAXES];  /* <--debug variables */
};
```

References

[And98] Torben Andersen. A first study of MMA antenna offset performance. MMA Memo 231, NRAO, September 1998. Abstract: "..must.. offset.. MMA.. pointing by 1.5 degrees within 1.5 seconds.. To avoid structural oscillations ('ringing') after the offset, specially shaped input trajectories must be applied.." Section 6 (Suppression of ringing) gives a simulated example of telescope response when we '..[subdivide] a step into two. The first step excites the vibration and the second one cancels it..' Concludes that '..a combination of shaped inputs with feedforward techniques has a high potential and should be studied further'.

[GP95] Wodek Gawronski and Ben Parvin. Simulations of the GBT antenna with the Command Preprocesor. GBT Memo 134, National Radio Astronomy Observatory, June 1995. '..we present here the simulation results of the slewing of the GBT antenna with the new command preprocessor (CPP).. developed by S. Tyler [Tyl94] at JPL.. jerk is smooth, and vibrations are not excited.. [in version B (CPP-B)] the acceleration.. is of sinusoidal pattern.. $a = \pm a_{max}(1 - \cos 2\pi\omega)$.. to avoid.. abrupt changes in acceleration, which cause oscillations of the antenna..'.

[Hil48] F. B. Hildebrand. *Advanced Calculus for Engineers.* Prentice-Hall, New York, 1948. The concept of impulse cancellation appears on p.78; the figure, text and derivation are effectively *identical* to the version in Hildebrand's later book [Hil62]. LOC=QA303.H55.

[Hil62] Francis B. Hildebrand. *Advanced Calculus for Applications.* Prentice-Hall, New York, 1962. This was an undergraduate textbook used by D.Wells circa 1963. The concept of impulse cancellation appears on p.71, in the midst of a discussion of the application of Laplace Transforms to the solution of linear differential equations. This book is a revised version of an earlier book [Hil48].

[LW93] R. Lacasse and T. Weadon. GBT dynamics. GBT Memo 104, NRAO, April 1993. This report reviews beam-settling-time (jerk-induced-vibration) results from simulations of raster-scanning observing modes performed by the GBT contractor [PCD93].

[Mel93] Jeff Mellstrom. Comments on R. Lacasse memorandum. JPL interoffice memorandum 3324-93-040, in response to [LW93]: "..fine tuning [of a servo system] for a specific experiment is [not] worthwhile because values that work well for one experiment under a given set of operating conditions may cause problems if those conditions change, or for another experiment.. JPL has developed a new concept.. trajectory preprocessor modifies the antenna commanded position such that the servo system always operates in a linear regime. Since the system is linear, global stability is assured, and performance is easily quantifiable.. suggest.. NRAO.. use a similar algorithm", April 1993.

[PCD93] PCD. Final analysis report for the Green Bank Telescope control system. Technical Report for contract AUI-1059, Precision Controls Division of Radiation Systems, Inc. [now COMSAT], Richardson, Texas and Sterling, Virginia, February 1993. Section 6.3 [p.75] summarizes numerical simulations of the time required to scan a 10 × 10 raster. It is shown that vibrations excited by the steps can be reduced by reducing deceleration (with the amount of the reduction "tuned" for the step size), but this will increase the total time required.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988. LOC=QA76.73.C15N865(1988), `http://cfata2.harvard.edu/numerical-recipes/`.

[Sch91] Johann Schraml. Smooth tracking commands for periodic position updates. Memo [limited distribution], National Radio Astronomy Observatory, September 1991. Discusses acceleration profiles for optimum motion commands.

[Set94] Raja V. Sethupathi. Improvement of transient response of vibrating secondary mirror of infrared astronomy telescope (IRAT). In Larry M. Stepp, editor, ***Advanced Technology Optical Telescopes V (Meeting held 1994-03-13 Kailua,HI,USA)***, volume SPIE Proceedings 2199, pages 1034–1045. SPIE (The International Society for Optical Engineering), 1994. "..the problem of improving the transient response.. with square input is considered.. another scheme, called Posicast control, which involves only modification of the input signal, is designed and implemented. The transient response is found to improve drastically..". LOC=QB88.I56(1994).

[Smi57] Otto J. M. Smith. Posicast control of damped oscillatory systems. ***Proceedings of the IRE***, pages 1249–1255, September 1957. The journal is now ***Proceedings of the IEEE***, LOC=TK5700.I6.

[Smi58] Otto J. M. Smith. ***Feedback Control Systems***. McGraw-Hill Book Company, N.Y., 1958. The "Posicast" input for a step function motion is described in pp.331–346: "Posicast control of an oscillatory system divides the input into two parts.. one-half cycle.."; LOC=TJ216.S5.

[Tyl94] S. R. Tyler. A trajectory preprocessor for antenna pointing. ***[JPL] Telecommunications and Data Acquisition Progress Report 42-118***, April-June:139–159, August 1994. Abstract: "A trajectory-preprocessing algorithm has been devised which matches antenna angular position, velocity, and acceleration to those of a target. This eliminates vibrations of the antenna structure caused by discontinuities in velocity and acceleration commands, and improves antenna-pointing performance by constraining antenna motion to a linear regime..".

[vH95] Sebastian von Hoerner. Fast pointing change and small oscillation. GBT Memo 132, National Radio Astronomy Observatory, June 1995. Numerical simulations demonstrate vibration cancelling for step trajectories with durations of an even number of vibration cycles.

[vH96] Sebastian von Hoerner. Preventing oscillations of large radio telescopes after a fast stop. GBT Memo 152, National Radio Astronomy Observatory, May 1996. '..it is shown that these oscillations can be prevented if the acceleration driving the telescope has the form $A(t) = \sin^n t$, and for a duration measured in multiples of the oscillation wavelength..'.

[Woo95] David Woody. Fast position switching of resonant structures. In an Email message sent on 1995-07-07 to P. Napier, J. Cheng, J. Payne and `<lugten@toby.berkeley.edu>`, Woody `<dpw@mm.ovro.caltech.edu>` said. '..the trajectory I have come up with has a gaussian velocity profile, $v(t) = \frac{S}{t_0\sqrt{\pi}}e^{-\frac{t}{t_0}^2}$, where t_0 is the characteristic gaussian width and S is the distance from A to B. The position vs. time is the integral of the velocity.. the Error Function.. position asymtotically approaches the final position with a deviation given by $\Delta p(t) \approx 10^{-(\frac{t}{t_0})^{1.45}}$. This gives excellent settling characteristics..', July 1995.