POSEMATH

Pose Mathematics

for Path Planning

POSEMATH: Pose Mathematics for Path Planning

POSEMATH is a library of software for representing and manipulating locations in three-dimensional space. POSEMATH can be used by either C or C++ programmers to define coordinate systems and the position and orientation of frames within coordinate systems, and to manipulate these frames to plan manipulator paths.

Code Organization

The POSEMATH software consists of three modules: the pose math code proper, support for printed output and diagnostics, and a sine-cosine utility which enables using the single-instruction sine and cosine on some platforms for efficiency. Each module consists of a code (`.c`) and header (`.h`) file. The code is written so that it will compile in both the C and C++ languages, using the ANSI C++ built-in compiler symbol `__cplusplus` for selection. Debug output can be enabled with the `PM_DEBUG` symbol. Defining this as a compiler switch will enable diagnostic printing messages when errors are encountered. Leaving it undefined will disable diagnostic printing messages.

The code is compiled into four separate libraries, for C or C++, and with debug output enabled or disabled. Here are the details:

`_posemath.c` C implementation of pose math functions

`posemath.cc` C++ implementation of pose math functions

`posemath.h` Declarations for pose math code, for C and C++

`_mathprnt.c` Printed output code, for C

`mathprnt.cc` Printed output code, for C++

`mathprnt.h` Declarations for printed output code, for C and C++

`sincos.c` C implementation of processor-specific single-instruction sine and cosine functions

`sincos.h` Declarations for single-instruction sine and cosine, for C and C++

For Unix and related operating systems, compiled libraries are:

`libpm.a` All code compiled for C or C++ linkage, no diagnostics

`libpmdb.a` All code compiled for C or C++ linkage, diagnostics included (`PM_PRINT_ERROR` defined)

For Microsoft Windows and related operating systems, compiled libraries are:

`libpm.a` All code compiled for C or C++ static linkage, no diagnostics

`pm.dll` All code compiled for C or C++ dynamic linkage, no diagnostics

Note that there is no diagnostics support for Microsoft Windows applications, as there is no traditional console available for these messages.

Programmers need only include `posemath.h` in their code to use the basic POSEMATH

library. If printed output is desired, programmers should include `mathprnt.h`. This file declares the C++ `iostream` operators for output, and the C counterparts to the `printf()` functions.

Naming Conventions

C data types in POSEMATH are prefixed with Pm, and follow the case-change convention where the subsequent words in the descriptive name are concatenated with leading capitals, e.g., `PmRotationVector`. C++ data types are prefixed with `PM_` , and are all capital letters with perhaps some additional underscores, e.g., `PM_ROTATION_VECTOR`. In C, they are declared as structure types with `typedef struct`. In C++, they are declared as classes. C functions begin with `pm`, and are followed by the POSEMATH C abbreviations for each type. For example,

`PmCartesian` the C Cartesian data type

`PM_CARTESIAN` the C++ Cartesian data type

`pmCartCartCross()` C function for cross product of two PmCartesian types

C++ versions of the C functions are implemented using operator overloading as much as possible. Where functions are required or more intuitive, these are implemented as overloaded functions and hence do not require a unique prefix. For example,

`norm()` C++ normalization function for all C++ types

`isNorm()` C++ normalization predicate for all C++ types

Usage

POSEMATH is based in C, with C++ added to make coding easier and more intuitive. Functions that result in a data type take pointers to the result as their last argument, and return an integer error code. For example, the function that computes the cross product of two Cartesian vectors is declared as

```
int pmCartCartCross(PmCartesian v1, PmCartesian v2, PmCartesian
* vout);
```

C programmers must supply a pointer to existing storage for a PmCartesian for the result of the cross product of `v1` and `v2`.

For some functions, such as the predicate `pmCartCartCompare`, the return value is the only result, and no pointer to results need be passed.

For C++, additional syntactic interfaces make programs easier to read. This is accomplished through overloading of functions and operators. For example, the multiplication operator * is overloaded to take a scalar and a Cartesian vector, and results in a scaled Cartesian vector. In C, this looks like:

```
PmCartesian a, b;

double s;

int error;

error = pmCartScalMult(a, s, &b);
```

The error code can be checked to see if the result is valid (although in this case scalar-vector multiplies are always valid). In C++, the code looks like:

```
PM_CARTESIAN a, b;

double s;

b = a * s;
```

which is much more intuitive. However, since the return value is the vector itself, the programmer cannot check the result for validity as before. POSEMATH provides a global integer variable, `pmErrno`, which is set to 0 when the result of a function or operation is valid, and to a non-zero error code if the function or operation is invalid. C++ programmers using the compact operator syntax should check `pmErrno` in place of the integer return code in the corresponding function.

The integer return code from the C functions and `pmErrno` are identical, and are declared in the header file posemath.h. These codes include:

`PM_ERR` unspecified error

`PM_IMPL_ERR` function not implemented

`PM_NORM_ERR` argument should have been normalized

`PM_DIV_ERR` divide by zero attempted

The use of a global variable `pmErrno` means that the C++ code is not thread-safe or reentrant. If this is a requirement, then the C functions must be used. This does not mean that the C *language* needs to be used: the C functions can be called from C++ code, but the programmer loses the syntactic compactness resulting from function- and operator overloading.

Units

Units are immaterial for translation quantities, and are assumed to be radians for angular quantities.

Translational Representations

In our three-dimensional universe, the location of a point in space is uniquely determined by three numbers. These numbers usually represent the coordinates of the point in each of three perpendicular axes, relative to some known origin. The numbers can be thought of as the distance along each axis that the point traversed (or *translated*) as it moved from the origin to its location.

Translation is often referred to as position. Common translational representations include Cartesian coordinates (X, Y, and Z) and spherical coordinates (azimuth, altitude, and range).

POSEMATH supports three translational representations: Cartesian, cylindrical, and spherical. These are defined as:

C struct	C++ class	fields	C abbreviation
PmCartesian	PM_CARTESIAN	x, y, z	Cart
PmCylindrical	PM_CYLINDRICAL	theta, r, z	Cyl
PmSpherical	PM_SPHERICAL	theta, phi, r	Sph

Fields are of type `double`. Values for the fields of each type are unconstrained. For example, the values for the angular fields is not required to lie in the range 0 to 2π, or $-\pi$ to π. However, if these types are returned or result from calculations, they may be normalized to lie within the range $[0, 2\pi)$ using the rotational equivalence that 0 equals 2π, π equals 3π, etc.

All translational representations are equivalent, in the sense that a location in space expressed in one representation can be converted to any of the other representations. The numbers will in general be different, but the location they represent is the same. POSEMATH provides functions to convert between these representations. Using the POSEMATH naming conventions, these function declarations are:

```
int pmCartCylConvert(PmCartesian, PmCylindrical *);

int pmCartSphConvert(PmCartesian, PmSpherical *);

int pmCylCartConvert(PmCylindrical, PmCartesian *);

int pmCylSphConvert(PmCylindrical, PmSpherical *);

int pmSphCartConvert(PmSpherical, PmCartesian *);

int pmSphCylConvert(PmSpherical, PmCylindrical *);
```

The first argument is the type to be converted from. The second argument is a pointer to the object to be converted to. The integer return value is 0 if successful, `PM_NORM_ERR` if unsuccessful, or `PM_IMPL_ERR` if unimplemented. For example, converting from a `PmCartesian v` to a `PmSpherical s` looks like:

```
pmCartCylConvert(v, &s);
```

For C++ programmers, converting between types can be done using the = operator, which has been overloaded to call these functions. Extending the above example, converting from a Cartesian to spherical coordinate representation in C++ looks like:

```
s = v;
```

Since the return value of the = operator is the converted type, the programmer should check the value of `pmErrno` for proper conversion if desired.

Rotational Representations

In addition to a location in space, an object has an orientation. Like location, orientation requires three numbers to be uniquely specified. The numbers can be thought of as the angles that the object is inclined to with respect to some reference planes.

Rotational representations often use more than three numbers to specify orientation. This may be to make the representation more intuitive, or for computational efficiency.

POSEMATH supports six rotational representations: rotation vectors, rotation matrices, quaternions, ZYZ Euler angles, ZYX Euler angles, and roll-pitch-yaw. These are defined as:

C struct	C++ class	fields	C abbreviation
PmRotationVector	PM_ROTATION_VECTOR	s, x, y, z	Rot
PmRotationMatrix	PM_ROTATION_MATRIX	PmCartesian x, y, z	Mat
PmQuaternion	PM_QUATERNION	s, x, y, z	Quat
PmEulerZyz	PM_EULER_ZYZ	z, y, zp	Zyz
PmEulerZyx	PM_EULER_ZYX	z, y, x	Zyx
PmRpy	PM_RPY	r, p, y	Rpy

Fields are of type `double`, except for a `PmRotationMatrix`, whose fields are of type `PmCartesian`. Values for the fields of each type may be constrained. For example, each `PmCartesian` element of a `PmRotationMatrix` must have a magnitude of 1.0 (each must be a unit vector), and they must be mutually perpendicular. Normalization functions are provided which take any of these rotational types as their first argument, a pointer to the same rotational type as their second argument, and place the normalization of the first into the second. `PM_NORM_ERR` is returned if the normalization could not take place. This will occur if the first argument is entirely 0. For example, normalizing a `PmQuaternion q` looks like:

```
PmQuaternion q;

pmQuatNorm(q, &q);
```

C++ programmers can use the overloaded function `norm()` to accomplish the same thing in a cleaner way:

```
PM_QUATERNION q;

q = norm(q);
```

Since the error code is not returned directly, C++ programmer should check `pmErrno` to make sure the

conversion was successful.

All rotational representations are equivalent, in the sense that a rotation in space expressed in one representation can be converted to any of the other representations. The numbers will in general be different, but the rotation they represent is the same. POSEMATH provides functions to convert between these representations. Using the POSEMATH naming conventions, these function declarations are:

```
int pmRotMatConvert(PmRotationVector, PmRotationMatrix *);

int pmQuatZyzConvert(PmQuaternion, PmEulerZyz *);

int pmZyxRpyConvert(PmEulerZyz, PmRpy *);
```

(Not all of the functions are listed). The first argument is the type to be converted from. The second argument is a pointer to the object to be converted to. The integer return value is 0 if successful, `PM_NORM_ERR` if unsuccessful, or `PM_IMPL_ERR` if the function is unimplemented. For example, converting from a `PmQuaternion q` to a `PmRotationMatrix m` looks like:

```
pmQuatMatConvert(q, &m);
```

For C++ programmers, converting between types can be done using the = operator, which has been overloaded to call these functions. Extending the above example, converting from a `PM_QUATERNION q` to a `PM_ROTATION_MATRIX m` in C++ looks like:

```
m = q;
```

Since the return value of the = operator is the converted type, the programmer should check the value of `pmErrno` for proper conversion if desired.

Axis-Angle Representations

Another rotational type, the axis-angle, is provided for representing rotations about one of the principal Cartesian axes by a given angle. This type, `PmAxis`, is the same for both C and C++ and may take the values `PM_X`, `PM_Y`, and `PM_Z` only, which represent rotations about the X, Y, and Z Cartesian axes, respectively. These functions are provided:

```
int pmAxisAngleQuatConvert(PmAxis, double, PmQuaternion *);

int pmQuatAxisAngleMult(PmQuaternion, PmAxis,

double, PmQuaternion *);
```

The first converts a rotation about the given axis by the given angle to a quaternion. The second computes the rotation resulting from an initial quaterion in the base frame, then a rotation by the given axis-angle in the quaternion frame.

Combined Representations

The complete representation of an objectÆs location and orientation in space requires both translational and rotational representations. Of the many combinations of translational and rotational types possible, the two most common have been defined. These are the homogeneous transform and the pose.

The homogeneous transform uses a Cartesian translation representation and a rotation matrix for rotation. The pose uses a Cartesian translation representation and a quaternion. These are defined as:

C struct	C++ class	fields	C abbreviation
PmHomogeneous	PM_HOMOGENEOUS	PmCartesian tran, PmRotationMatrix rot	Hom
PmPose	PM_POSE	PmCartesian tran,PmQuaternion rot	Pose

Since these combined representations include elements which require normalization, normalization functions are provided which take any of these combined types as their first argument, a pointer to the same combined type as their second argument, and place the normalization of the first into the second. `PM_NORM_ERR` is returned if the normalization could not take place. For example, normalizing a `PmPose p` looks like:

```
pmPoseNorm(p, &p);
```

C++ programmers can use the overloaded function `norm()` to accomplish the same thing in a cleaner way:

```
PM_POSE p;

p = norm(p);
```

Since the error code is not returned directly, C++ programmer should check `pmErrno` to make sure the conversion was successful.

All combined representations are equivalent, in the sense that a position and orientation in space expressed in one representation can be converted to any of the other representations. The numbers will in general be different, but the position and orientation they represent is the same. POSEMATH provides functions to convert between these representations. Using the POSEMATH naming conventions, these function declarations are:

```
int pmPoseHomConvert(PmPose, PmHomogeneous *);

int pmHomPoseConvert(PmHomogeneous, PmPose *);
```

The first argument is the type to be converted from. The second argument is a pointer to the object to be converted to. The integer return value is 0 if successful, `PM_NORM_ERR` if unsuccessful, or `PM_IMPL_ERR` if the function is unimplemented. For example, converting from a `PmPose p` to a `PmHomogeneous h` looks like:

```
pmPoseHomConvert(p, &h);
```

For C++ programmers, converting between types can be done using the = operator, which has been overloaded to call these functions. Extending the above example, converting from a `PM_POSE p` to a `PM_HOMOGENEOUS h` in C++ looks like:

```
h = p;
```

Since the return value of the = operator is the converted type, the programmer should check the value of `pmErrno` for proper conversion if desired.

Functions and Operators

Mathematical operations for translational and rotational types are provided to carry out calculations typical of manipulator path planning in three-space. In general, functions that work on a particular representation will also work on other equivalent representations.

Detailed descriptions of each function are found in Appendix B.

Translational Functions

The translational data representations are used to represent position vectors, and all the typical vector operations have been provided. These include comparison, the dot product, cross product, norm (unit vector), magnitude, sum, difference, magnitude of difference (displacement), scalar multiply and divide, and generalized inverse.

For C functions that take two arguments, the arguments must be the same type. For C++ overloaded functions or operators that take two arguments, they may be of different type. In this case, the constructors are called to do the conversions automatically.

Return values for all the C functions are integer error codes, with the exception of the predicate functions which return 1 if it is true or 0 if not.

Return values for all the C++ functions are the result that corresponds to the last argument of the C functions. The global variable `pmErrno` should be checked to see if the function was successful.

Following the POSEMATH naming conventions, these functions are prefixed with `pm`, the abbreviated names of their arguments, and the operation. In the table below, `X` indicates any of the translation types (`PmCartesian`, `PmCylindrical`, or `PmSpherical`), and `s` indicates a scalar `double` type.

function	C version	C++ version
equality predicate	pmXXCompare(X,X)	==, !=
type conversion	pmXXConvert(X,X)	=
dot product	pmXXDot(X,X,X*)	dot(X,X)
cross product	pmXXCross(X,X,X*)	cross(X,X)
magnitude	pmXMag(X,s*)	mag(X)
normalization (unit)	pmXNorm(X,X*)	norm(X)
norm predicate	pmXIsNorm(X)	isNorm(X)
displacement	pmXXDisp(X,X,s*)	disp(X,X)
sum	pmXXAdd(X,X,X*)	X+X
difference	pmXXSub(X,X,X*)	X-X
scalar multiply	pmXScalMult(X,s,X*)	X*s, s*X
scalar divide	pmXScalDiv(X,s,X*)	X/s
additive inverse	pmXNeg(X,X*)	-X
inverse (X dot X^{-1} = 1)	pmXInv(X,X*)	inv(X)
projection	pmXXProj(X,X,X*)	proj(X,X)

Rotational Functions

The rotational data representations are used to represent rotations in space of one reference frame relative to another. Functions that operate on these rotational types and combinations of translational and rotational types include comparison, normalization into valid range, checking for normal range, inverse, multiplication by a scalar (scaling the rotation about the constant direction), vector multiply (resulting in a rotation of the vector), and rotational multiply (concatenation of several rotations).

For C functions that take two arguments, the arguments must be the same type. For C++ overloaded functions or operators that take two arguments, they may be of different type. In this case, the constructors are called to do the conversions automatically.

Return values for all the C functions are integer error codes, with the exception of the predicate functions which return 1 if it is true or 0 if not.

Return values for all the C++ functions are the result that corresponds to the last argument of the C functions. The global variable `pmErrno` should be checked to see if the function was successful.

Following the POSEMATH naming conventions, these functions are prefixed with `pm`, the abbreviated names of their arguments, and the operation. In the table below, `R` indicates any of the rotation types (`PmRotationVector`, `PmRotationMatrix`, `PmQuaternion`, `PmEulerZyz`, `PmEulerZyx`, or `PmRpy`), `X` indicates any of the translation types (`PmCartesian`, `PmCylindrical`, or `PmSpherical`), and `s` indicates a scalar `double` type.

function	C version	C++ version
equality predicate	pmRRCompare(R,R)	==, !=
type conversion	pmRRConvert(R,R)	=
magnitude	pmRMag(R,s*)	mag(R)
normalization	pmRNorm(R,R*)	norm(R)
norm predicate	pmRIsNorm(R)	isNorm(R)
scalar multiply	pmRScalMult(R,s,R*)	R*s, s*R
scalar divide	pmRScalDiv(R,s,R*)	R/s
vector multiply	pmRXMult(R,X,X*)	R*X, X*R
rotation multiply	pmRRMult(R,R,R*)	R*R
inverse	pmRInv(R,R*)	inv(R), -R

Combined Functions

The combined representations are used to represent the position and orientation in space of one reference frame relative to another. In POSEMATH, the position and orientation of a reference frame is termed a *pose*. Although the `PmPose` type appears to have some preference as a pose representation, both `PmPose` and `PmHomogeneous` are pose types. Poses contain both translational and rotational representations. Functions which operate on poses manipulate individual poses, and convert vectors or poses in one

reference frame into another. These functions include comparison, normalization into valid range, checking for normal range, inverse, and multiplications between poses and translations, rotations, and other poses.

For C functions which take two arguments, the arguments must be the same type. For C++ overloaded functions or operators which take two arguments, they may be of different type. In this case, the constructors are called to do the conversions automatically.

Return values for all the C functions are integer error codes, with the exception of the predicate functions which return 1 if it is true or 0 if not.

Return values for all the C++ functions are the result that corresponds to the last argument of the C functions. The global variable `pmErrno` should be checked to see if the function was successful.

Following the POSEMATH naming conventions, these functions are prefixed with `pm`, the abbreviated names of their arguments, and the operation. In the table below, `R` indicates any of the translation types (`PmCartesian`, `PmCylindrical`, or `PmSpherical`), `X` indicates any of the translation types (`PmCartesian`, `PmCylindrical`, or `PmSpherical`), and `s` indicates a scalar `double` type.

function	C version	C++ version
equality predicate	pmPPCompare(P,P)	==, !=
normalization	pmPNorm(P,P*)	norm(P)
norm predicate	pmPIsNorm(R)	isNorm(P)
inverse	pmPInv(P,P*)	inv(P), -P
vector multiply	pmPVMult(R,X,X*)	R*X, X*R
rotation multiply	pmRRMult(R,R,R*)	R*R

Automatic Conversions in C++

As of this writing, not all of the C conversion functions have been implemented. C programmers should check Appendix B for functions that they intend to use. Some functions whose existence is implied may be stubbed to return `PM_IMPL_ERR`, or not exist at all. This apparent laziness is due to the combinatorics of the conversion functions, particularly among the rotational types.

For example, there is no C function `pmZyzRpyConvert` which converts a `PmEulerZyz` to a `PmRpy`. The function may not exist, in which case the compiler will flag calls to it, or it may be stubbed to return PM_IMPL_ERR, in which case a run-time error will occur. Ultimately, all the C functions will be implemented in some fashion, but currently they are not. In this case, the C programmer must convert the ZYZ Euler to a rotation matrix, and from a rotation matrix to a roll-pitch-yaw representation, as shown below for the `PmEulerZyz zyz` and `PmRpy rpy`:

```
PmRotationMatrix mat;

pmZyzMatConvert(zyz, &mat);

pmMatRpyConvert(mat, &rpy);
```

C++ programmers are luckier. Each data representation class has constructors and assignment operators which initialize or assign its representation from any other representation. This may have been

accomplished directly, if equivalent C conversion functions exist, or via conversion to some intermediate type. C++ programmers can use the = operator directly:

```
rpy = zyz;
```

In this case, the POSEMATH implementation for `PM_RPY::operator = (PM_EULER_ZYZ)` does the conversion to the intermediate rotation matrix format and out again automatically.

Appendix A: Platform Support and Compilation

The POSEMATH code has been compiled for both Unix and Microsoft platforms, in various specific versions with various compilers. These include:

Operating System Compilers

Sun Solaris Gnu C, C++; CenterLine C++

Linux Gnu C, C++

Real-time Linux Gnu C

LynxOS Gnu C, C++

Microsoft Windows 3.1 Borland C, C++

Microsoft Windows 95 Microsoft Visual C++

Microsoft Windows NT 4.0 Microsoft Visual C++

NIST-Specific Compilation

The POSEMATH libraries are available in the RCS platform-specific release directories. The libraries are in the directory

`/proj/rcslib/plat/<PLAT>/lib/`,

and the header files are located in the directory

`/proj/rcslib/plat/<PLAT>/include/`. The value for `<PLAT>` can be one of:

```
irix5

linux

rtlinux

lynxosPC

lynxosPC23

lynxosVME

sunos4
```

`sunos5`

`sunos5CC`

`vxworks5.1`

`vxworks5.2`

Appendix B: Programming Reference

C functions are intended be used by C programmers, since the arguments are of the C types. C++ programmers should use the C++ overloaded operators and functions.

C++ programmers can use the C functions by explicitly converting the C++ types to C types, and back again. `toType(src, dst)` macros have been provided to accomplish this, where `Type` is the C abbrevation (e.g., `Cart`, `Cyl`, `Pose`), `src` is the source to be converted, and `dst` is the result. These exploit the fact that the fields for both the C and C++ data types have the same names.

Unless indicated otherwise, all functions return an integer error code, which is 0 upon success or one of the error codes defined in `posemath.h`. This error code is also written to the global variable `pmErrno`.

There is a bias in the POSEMATH implementation toward using `PmCartesian` for translation types, `PmQuaternion` for rotational types, and `PmPose` for pose types. This is primarily due to the computational efficiency of the quaternion, meaning that programs that use quaternion representations will execute faster than programs using, for example, rotation matrices or Euler angles. In the descriptions of the functions that follow, any translational type can be substituted for `PmCartesian`, any rotational type can be substituted for `PmQuaternion`, and any pose type can be substituted for `PmPose`.

pmQuatCartMult(PmQuaternion q, PmCartesian v, PmCartesian * vout);

> Sets vout to the vector resulting from a rotation of v specified by q. Corresponds to C++ operator *, q * v. Operator * is not commutative.

pmQuatQuatMult(PmQuaternion q1, PmQuaternion q2, PmCartesian * qout);

> Sets qout to the quaternion representing a rotation first by q2 in the base frame, then by q1 in the q2 frame. Corresponds to C++ operator *, which is not commutative: q1*q2 ≠ q2*q1, in general.

pmPoseCartMult(PmPose p, PmCartesian v, PmCartesian vout);

> Sets vout to the vector representing a transformation by the pose p. Corresponds to C++ operator *, p * v. Operator * is not commutative (in fact, v * p yields p whose translation is added by v).

pmPosePoseMult(PmPose p1, PmPose p2, PmPose pout);

> Sets pout to the pose representating a transformation first by p1 in the base frame, then by p2 in the p1 frame. Corresponds to C++ operator *, which is not commutative: p1*p2 ≠ p2*p1, in general.