Numerical Methods

Last Update: 8/1/18

Started By: Derek C. Richardson

Contact Info:
Department of Astronomy
University of Maryland
College Park MD 20742
Tel: 301-405-8786
E-mail: `dcr@astro.umd.edu`

Contents

1 OVERVIEW

Certain standard numerical methods are needed for various routines in `pkdgrav` and accompanying software. Rather than writing these all from scratch, the approach has been to adapt existing published methods to the code. When `pkdgrav` was first developed, several routines from *Numerical Recipes* (*NR*, various editions) were adopted, but these have a restrictive distribution license. Since the aim is to have a public release of `pkdgrav`, it is necessary to provide alternatives to the *NR* routines. For now we have chosen the *GNU Scientific Library*

(*GSL*) to fill this role. The user may choose between *NR* and *GSL* when compiling. Any public release of the code will only have *GSL* available.

This document describes the available methods, their calling structure, and how to select them during compilation. In addition, each collection of methods has a standalone test suite that can be invoked by editing the corresponding source file—instructions are in the files themselves.

2 Choosing Between *NR* and *GSL*

In `Makefile.in`, uncomment `USE_NUMREC` for *NR*, or `USE_GSL` for *GSL*, along with the corresponding `METHOD` macros. For *NR*, the `METHOD` macros are blank because the code is part of the `pkdgrav` (private) code base. For *GSL*, they provide the location of include files and libraries (and which libraries to use). The example in `Makefile.in` is for a typical *MacPorts* install on a Mac. The chosen methods will be used when the code is compiled with `make`.

3 Integration of Differential Equations (`diffeq.c`)

Synopsis

```
#include <diffeq.h>

void diffeqIntegrate(FLOAT fVars[], int nVars, FLOAT fStart, FLOAT fStop,
                     FLOAT fAccuracy, FLOAT fStepTry, FLOAT fStepMin,
                     diffeqDerivsT funcDerivs, void *pUserData);
```

`fVars[]`	input: starting values of dependent variables
`fVars[]`	output: final solution values for dependent variables
`nVars`	input: number of variables
`fStart`	input: start value of independent variable (e.g., time)
`fStop`	input: final value for independent variable
`fAccuracy`	input: accuracy parameter for the integrator
`fStepTry`	input: initial step size to try
`fStepMin`	input: smallest allowed step size (can be zero)
`funcDerivs`	input: name of function for computing derivatives
`pUserData`	input: generic pointer to optional data (can be NULL)

`diffeqIntegrate()` provides a method for integrating ordinary differential equations (ODEs) with time adapativity. The *NR* and *GSL* implementations use slightly different 5th-order Runge-Kutta integrators and adaptive step algorithms, but no noticeable difference was found in simple tests. The derivatives function has the following prototype:

```
#include "floattype.h"

typedef int (*diffeqDerivsT)(FLOAT t, const FLOAT fVars[], FLOAT fDerivs[],
                             void *pUserData);
```

`t`	input: value of independent variable (e.g., time)
`fVars`	input: values of dependent variables
`fDerivs`	output: derivative values (i.e., left-hand-sides of ODEs)
`pUserData`	input: genetic pointer to optional data (can be NULL)
	return: 0 on success

4 Matrix Inversion and Diagonalization (`matrix.c`)

Synopsis

```
#include <matrix.h>
```

```
void matrixInvert(Matrix m);
```

`m`	input: matrix to be inverted
`m`	output: inverted matrix

```
void matrixDiagonalize(const Matrix m, Vector vEvals, Matrix mEvecs, int bSort);
```

`m`	input: matrix to be diagonalized
`vEvals`	output: eigenvalues
`vEvecs`	output: eigenvectors
`bSort`	input: 0 = do not sort, otherwise sort

These functions implement matrix operations that are not just simple transformations. The vector and matrix types are defined in `matrix.h` if needed, but are simply `FLOAT[3]` and `FLOAT[3][3]` respectively, where `FLOAT` is assumed to be `double` unless overridden by `floattype.h` (`pkdgrav` only). `matrixInvert` can be used to solve systems of equations, but it is generally more efficient to use a specialized system solver for that (not needed so far). `matrixDiagonalize` requires the input matrix to be real symmetric; the resulting eigenvectors form an orthonormal set. Optionally the eigenvalues can be sorted in order of decreasing magnitude, with the eigenvectors correspondingly rearranged (not needed so far).

5 Polynomial Root Finding (`polyroots.c`)

Synopsis

```
#include <polyroots.h>
```

```
int polyQuadSolve(double a, double b, double c, double *x1, double *x2);
```

`a`, `b`, `c`	input: coefficients of $ax^2 + bx + c = 0$
`x1`, `x2`	output: pointers to roots of equation
	return: 0 if 2 real roots found, 1 otherwise

```
int polyCubicSolveLimited(double a1, double a2, double a3, double *x1,
                          double *x2, double *x3);
```

`a1`, `a2`, `a3`	input: coefficients of $x^3 + a_1x^2 + a_2x + a_3 = 0$
`x1`, `x2`, `x3`	output: pointers to roots of equation
return: 0 if 3 real roots found, 1 otherwise	

```
int polyQuarticSolveLimited(double a1, double a2, double a3, double a4,
                            double *x1, double *x2, double *x3, double *x4);
```

`a1`, `a2`, `a3`, `a4`	input: coefficients of $x^4 + a_1x^3 + a_2x^2 + a_3x + a_4 = 0$
`x1`, `x2`, `x3`, `x4`	output: pointers to roots of equation
return: 0 if 4 real roots found, 1 otherwise	

```
void polyFindRealRoots(int nDegree, double dCoefs[], double dRoots[],
                       int *nRealRoots);
```

`nDegree`	input: degree of polynomial $(n-1)$
`dCoefs`	input: coefficients of $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} = 0$
`dRoots`	output: real roots of equation
`nRealRoots`	output: pointer to number of real roots

These functions find real roots of polynomials with real coefficients. For the quadratic solver, if no real roots are found, the contents of `x1` and `x2` are not altered; otherwise, the roots are ordered such that $x_1 \leq x_2$. For the cubic solver, if only 1 real root is found, it is stored in `x1` and the contents of the other pointers are unaltered. For the quartic solver, only a few special cases will result in 4 real roots; if `a4` is 0, `x1` will contain 0, and a cubic will be solved for the remaining roots. Finally, the general root finder, `polyFindRealRoots()`, uses the macro `EPS` defined near the top of `polyroots.c` to determine whether a root is real or not, the condition being that the magnitude of the imaginary part is less than or equal to the magnitude of the real part. The standalone test queries the user for an equation to solve, calls the applicable functions, and times the general approach, which also finds complex roots. Try solving $x - 1 = 0$, $x^2 - 3x + 2 = 0$, i.e., $(x-1)(x-2) = 0$ (roots 1, 2), $x^3 - 6x^2 + 11x - 6 = 0$ (roots 1, 2, 3), $x^4 - 10x^3 + 35x^2 - 50x + 24$ (roots 1, 2, 3, 4), etc.

6 Random Number Generators (`random.c`)

Synopsis

```
#include <random.h>

Ullong randReadUrandom(void);
void randSeedGenerator(Ullong ullSeed);
double randUniform(void);
double randRayleigh(void);
double randGaussian(void);
double randPoisson(double dMean);
```

These functions provide random samples from various distribution functions. Here `Ullong` is `uint64_t` from `stdint.h`, i.e., an `unsigned long long` type (requires C99 standard or later). A brief description of each function follows.

6.1 `randReadUrandom(void)`

Returns a pseudo-random 64-bit integer by reading from the device `/dev/urandom`, which is a handy way of seeding the generators (but too slow for other purposes). Random bytes are generated in `/dev/urandom` based on the operation of the computer's devices, including keyboard and mouse use, and timings driven by disk access. These sources are said to have "high entropy" but they are still not as truly random as, say, clicks on a geigercounter.

6.2 `randSeedGenerator(Ullong ullSeed)`

Seeds the random number generator using the provided 64-bit integer. The *NR* generator is one of their own devising. The *GSL* generator is a Mersenne Twister (specifically, MT19937).

6.3 `randUniform(void)`

Returns a uniformly distributed random double-precison deviate between 0 (inclusive) and 1 (exclusive). Be sure to seed the generator first.

6.4 `randRayleigh(void)`

Returns a Rayleigh-distributed random double-precision deviate x with unit scale parameter σ, where the probability density function

$$p(x) = \frac{x}{\sigma^2} e^{-x^2/(2\sigma^2)},\ x \geq 0.$$

Just multiply the deviate by σ to scale it as desired.

6.5 `randGaussian(void)`

Returns a normal-distributed (Gaussian) random double-precision deviate x with zero mean μ and unit variance σ^2:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Multiply the deviate by σ and add μ to get the desired mean and variance.

6.6 `randPoisson(double dMean)`

Returns a Poisson-distributed random integer k (cast to `double`) with the given mean μ, where the probability of observing k events is given by:

$$p(k) = \frac{\mu^k}{k!} e^{-\mu},\ k \geq 0.$$