(J B Gallagher, London UK, 12 Mar 2012, revised 19 Mar 2012)

Everything Flows

Abstract

We suggest that 3 "spatial" degrees of freedom are dynamically emergent from unitary evolution of the state vector

Introduction

The question "why do we live in 3-dimensions of space?" has long been a puzzle, but has a remarkably simple solution if we consider a discretely defined state vector of the universe evolving under evolution by a matrix equation. The main result is Proposition 1.

Proposition 1

Let L be an n x n anti-hermitian matrix of the form:

```
L = [  0  w12 w13 w14 w15 ... w1n ]
    [ z21 0   w23 w24 w25 ... w2n ]
    [ z31 z32 0   w34 w35 ... w3n ]
    [ z41 z42 z43 0   w45 ... w4n ]
    [ z51 z52 z53 z54 0   ... w5n ]
    [ ...                 ...  .  ]
    [ ...                 ...  .  ]
    [ zn1 zn2 zn3 zn4 zn5 ...  0  ]
```

(zji = -wij* or zji = wij = 0)

Then there exists a positive real number h = h(L) such that the matrix M = (exp(hL) - I) has a (non-trivial) period-3 global attractor

ie for any non-zero U in C^n there exists period-3 fixed points R1, R2, R3 in C^n, Ri = R(U) such that M^3.Ri = Ri for i=1,2,3 and |M^(3k).U - Rj| --> 0 (some j=1,2,3) as k --> infinity

('exp' denotes exponential matrix and I is n x n identity matrix, note that exp(hL) is unitary)

Proof

A full mathematical proof is in preparation but we can check the proposition for small values of n (up to ~1000) via the computer code provided in the appendix. Simulations for matrices of size n=100, 500 and 1000 are consistent with the result of Proposition 1, the dynamics always converge to a 3-cycle when an appropriate value of h ('lambda' in the computer code) is found.

Discussion

We suggest the following discrete model of QM to apply the proposition

(i) The Universe is described by a state vector U in C^n for some finite n
(ii) (Randomness) Any element in U can randomly change its phase
(iii) (Discrete Time Evolution) When a single element in U changes its phase the universe evolves according to U(t+tdelta) = exp(hL).U(t) - U(t), for some time interval tdelta > 0.

This model naturally accounts for a cosmic "speed of light" limit and has QM superpositions without "many-worlds" splitting. Note also that only the 'change' in the universe remains after a single evolution step - the past universe is gone forever (and hence cannot be changed) - the universe exists simply as an inexorable flow of random states and nothing else (there are no "elements of reality")

The 3 dimensions of "space" are dynamically emergent from the evolution equation, they need not have existed in the early universe but are emergent as the state vector approaches the attractive period-3 fixed points.

We also suggest that the evolution equation "explains" renormalisation by subtracting the universe state vector, and make the conjecture (from rough numerical estimates) that h ~ O(1/sqrt(n)).

Appendix

```
// unitary3d.cpp - James B Gallagher 12:38 19th Mar 2012
//
// Plots a complex vector evolving under multiplication by an approx unitary matrix (minus Identity matrix)
// created by taylor expansion of exp(L) (to 13th power) where L is a randomly constructed anti-hermitian matrix.
// ie plots dynamics of (exp(L) - I)*U(t) for a complex state vector U(t), and demonstrates period-3 cycles.
//
// download link - http://jbg.f2s.com/unitary3d.cpp.final
```

```
//
// compile with: g++ -larmadillo -lX11 -O2 -o u3dtest unitary3d.cpp
// you need the armadillo matrix library and Xlib headers,
// Fedora 16 linux: 'sudo yum install armadillo armadillo-devel gcc-c++ libX11-devel'
// Ubuntu 10.04 LTS: 'sudo apt-get install libarmadillo0 libarmadillo-dev libboost-dev libx11-dev g++'
//
// usage:
//
// ./u3dtest <rnd_seed> <#points> <lambda>
//
// lambda is a scaling factor for the matrix which prevents dynamics blowing up or shrinking to a point, it is
// a finely tuned value depending on rnd_seed and #points, and can be manually found with autozoom OFF or
// automatically by pressing SPACE ( roughly, lambda ~ O(1/sqrt(NPTS)) )
//
// eg ./u3dtest 42 100 0.066681 (default parameters)
// eg ./u3dtest 42 500 0.0292285
// eg ./u3dtest 42 1000 0.02064
// eg ./u3dtest 7 1000 0.020375
//
// Controls: UP - iterate system and display a 3-cycle (display every 3rd iteration)
//           RIGHT - display every iteration (forwards in time)
//           R - restart with new random state vector (matrix remains the same)
//           U - restart with unit state vector (1,0,0,...,0)
//           S - toggle stochasticity (one state randomly changes phase per iteration, default is OFF)
//           T - restart with new randomly generated matrix
//           A - toggle autozoom (default is ON)
//           Z/X - manually zoom in/out (turns autozoom off)
//           SPACE - automatically search for stable value of lamda for the current matrix
//           any other key - randomly change phase of a randomly chosen state
//           Q - quit
//
// Hold down the UP cursor key until dynamics stabilise.
// If autozoom is off you will need to zoom in/out as required.
// if dynamics are not stable at any zoom level, adjust lambda until stability is achieved
// Press SPACE to start an automated search for a (fairly) stable lambda,
// this takes several minutes for 1000x1000 matrices on a 2GHZ Core 2 Duo with 3GB ram
// Pressing the UP cursor key displays every 3rd iteration, so that a single 3-cycle dynamics are clear,
// To view every iteration in sequence hold down the RIGHT cursor key, you'll see all the 3-cyles in the dynamics
// flashing rapidly by
//

#include <stdio.h>
#include <X11/Xlib.h>
#include <iostream>
#include <cstdlib>
#include "armadillo"

using namespace arma;
using namespace std;

Display *dsp;
Window win;
GC gc;
XImage *ximage;
typedef unsigned int PIXEL;
PIXEL *image32;
int XRES, YRES;
int height, width;
double zoom = 1.0;
int autozoom = 1;
int unitvector = 0;

int iteration = 3000000; // any large positive multiple of 3
int cycle = 0;

int rseed = 42;
int NPTS = 100;

// this scaling factor gives stable dynamics with these default parameters,
// you have to adjust it to avoid the dynamics blowing up or collapsing to zero
double lambda = 0.066681;

int stochastic = 0;

void __inline__ setPixel(int, int, PIXEL);
void draw_rec(int, int, int, int, PIXEL);
void initvec(cx_vec&);
void update_image(const cx_vec&);
void create_matrix(int rnd_seed, cx_mat&, cx_mat&);
void autolambda(cx_mat&, cx_mat&, cx_vec&);

int main(int argc, char **argv)
{
    dsp = XOpenDisplay(NULL);
```

```
    int screen = DefaultScreen(dsp);
    Visual *visual = DefaultVisual(dsp, 0);

    if (visual->c_class != TrueColor)
    {
        fprintf(stderr, "Cannot handle non true color visual ...\n");
        XCloseDisplay(dsp);
        exit(1);
    }

    XRES = DisplayWidth(dsp,screen);
    YRES = DisplayHeight(dsp,screen);

    width = XRES/2;
    height = YRES/2;

    image32 = (PIXEL *)malloc(XRES*YRES*4);
    ximage = XCreateImage(dsp, visual, 24, ZPixmap, 0, (char *)image32, XRES, YRES, 32, 0);

    win = XCreateSimpleWindow(dsp, RootWindow(dsp, 0), 0, 0, width, height, 1, 0, 0);
    gc  = XCreateGC(dsp, win, 0, NULL);

    // intercept window delete event
    Atom wmDelete=XInternAtom(dsp, "WM_DELETE_WINDOW", True);
    XSetWMProtocols(dsp, win, &wmDelete, 1);

    KeyCode keyA, keyQ, keyR, keyS, keyT, keyU, keyX, keyZ, keyUP, keyRIGHT, keySPACE;
    keyA = XKeysymToKeycode(dsp, XStringToKeysym("A"));
    keyQ = XKeysymToKeycode(dsp, XStringToKeysym("Q"));
    keyR = XKeysymToKeycode(dsp, XStringToKeysym("R"));
    keyS = XKeysymToKeycode(dsp, XStringToKeysym("S"));
    keyT = XKeysymToKeycode(dsp, XStringToKeysym("T"));
    keyU = XKeysymToKeycode(dsp, XStringToKeysym("U"));
    keyX = XKeysymToKeycode(dsp, XStringToKeysym("X"));
    keyZ = XKeysymToKeycode(dsp, XStringToKeysym("Z"));
    keyUP = XKeysymToKeycode(dsp, XStringToKeysym("Up"));
    keyRIGHT = XKeysymToKeycode(dsp, XStringToKeysym("Right"));
    keySPACE = XKeysymToKeycode(dsp, XStringToKeysym("space"));

    XSelectInput(dsp, win, KeyPressMask|ButtonPressMask|ExposureMask|StructureNotifyMask);
    XMapWindow(dsp, win);

    XEvent ev, ev1;

    if (argc > 1) rseed = atoi(argv[1]); // use 1st cmd line arg as random seed
    if (argc > 2) NPTS = atoi(argv[2]); // use 2nd cmd line arg as NPTS
    if (argc > 3) lambda = atof(argv[3]); // use 3rd cmd line arg as lambda

    if (NPTS < 1 || NPTS > 10000) { cout << "NPTS error." << endl; NPTS = 100; }
    if (lambda == 0.0) { cout << "lambda error." << endl; lambda = 0.5; }

    cout << "Random Seed, rseed =                        " << rseed << endl;
    cout << "No. of states/points, NPTS =                " << NPTS << endl;
    cout << "Scale Factor, lambda =                      " << lambda << endl;
    cout << endl;

    cx_mat M, L;
    cx_vec U; // state vector that we evolve

    cout << "Creating anti-hermitian " << NPTS << "x" << NPTS << " matrix and its exponential..." << endl;
    create_matrix(rseed, L, M);
    cout << "Done." << endl;

    initvec(U);

    int loop = 1, i, w, h;

////////////////////////////////////////////////////////
// infinite loop finds lambda for varying parameters///////
////////////////////////////////////////////////////////
    while (false) // change to false to disable
    {
      autolambda(M, L, U);

      rseed = time(0);
      cout << "rseed = " << rseed << endl;
      create_matrix(rseed, L, M);
      initvec(U);
    }
////////////////////////////////////////////////////////

    while(loop)
    {
      XNextEvent(dsp, &ev);
```

```
      switch(ev.type)
      {
      case Expose:
        if (XCheckTypedWindowEvent(dsp, win, ConfigureNotify, &ev1))
          {
            XPutBackEvent(dsp, &ev1);
            break;
          }
        if (ev.xexpose.count == 0) update_image(U);
        break;

      case ConfigureNotify:
        // update width & height in case of window resize
        w = ev.xconfigure.width;
        h = ev.xconfigure.height;
        if (w>1) width = w;
        if (h>1) height = h;
        if (!XCheckTypedWindowEvent(dsp, win, ConfigureNotify, &ev1)) update_image(U);
        break;

      case KeyPress:
        if (ev.xkey.keycode == keyQ) { loop = 0; break; }
        else if (ev.xkey.keycode == keySPACE) { autolambda(M, L, U); break; }
        else if (ev.xkey.keycode == keyA) { autozoom = 1 - autozoom; cout << "autozoom = " << autozoom << endl; }
        else if (ev.xkey.keycode == keyR) { unitvector = 0; initvec(U); break; }
        else if (ev.xkey.keycode == keyU) { unitvector = 1; initvec(U); break; }
        else if (ev.xkey.keycode == keyS) { stochastic = 1-stochastic;
                                            cout << "stochastic = " << stochastic << endl;
                                            break; }
        else if (ev.xkey.keycode == keyT) { rseed = rand();
                                            cout << "New " << NPTS << "x" << NPTS;
                                            cout << " matrix with rseed = " << rseed << endl;
                                            create_matrix(rseed, L, M);
                                            unitvector = 0;
                                            initvec(U);
                                            autozoom = 1;
                                          }
        else if (ev.xkey.keycode == keyX) { zoom = 0.5*zoom; autozoom = 0; cout << "zoom = " << zoom << endl; }
        else if (ev.xkey.keycode == keyZ) { zoom = 2.0*zoom; autozoom = 0; cout << "zoom = " << zoom << endl; }
        else {
          // if stochastic mode is set then a single state changes phase randomly each iteration
          double p = 2.0 * math::pi() * (rand()/(double)RAND_MAX);
          complex<double> eip (cos(p), sin(p));
          if (stochastic) U(rand()%NPTS) *= eip;

          if (ev.xkey.keycode == keyUP) { do { U=M*U; iteration++; } while (iteration%3 != cycle); }
          if (ev.xkey.keycode == keyRIGHT) {
            cycle = (cycle+1)%3;
            U=M*U;
            iteration++;
          }
        }
        update_image(U);
        break;

      case (ClientMessage) :
        if (ev.xclient.data.l[0] == wmDelete) loop = 0;
        break;

      default :
        break;
      }
    }

    free(image32);
    XDestroyWindow(dsp, win);
    XCloseDisplay(dsp);

    return 0;
}

void __inline__ setPixel(int x, int y, PIXEL clr)
{
  *(image32 + x + y*XRES) = clr;
}

void draw_rec(int x, int y, int side1, int side2, PIXEL clr)
{
  PIXEL *pos;
  int i,j;

  pos = image32 + x + y*XRES;

  for (j=0; j<side2; j++)
```

```
  {
    for (i=0; i<side1; i++)
    {
      *(pos++) = clr;
    }
    pos += XRES-side1;
  }
}

void initvec(cx_vec& V)
{
    int i,j;

    V = zeros<cx_vec>(NPTS);

    // When U pressed use unit vector (1,0,0,...)
    if ( unitvector ) V(0) = 1.0;
    else {
      // create random complex values so real/imag components are in (-1,1)
      complex<double> c1(0.5,0.5);
      for (i=0; i<NPTS; i++) V(i) = 2.0*(complex<double>(rand()/(double)RAND_MAX,rand()/(double)RAND_MAX) - c1);
    }

    update_image(V);
}

void update_image(const cx_vec& V)
{
  // clear window
  draw_rec(0,0,width,height,0x000000);

  // draw XY axes
  int x, y;
  for (x=0; x<=width; x++) setPixel(x, height/2, 0x888888);
  for (y=0; y<=height; y++) setPixel(width/2, y, 0x888888);

  // Plot the complex elements of vector V
  int i;

  // scale the plotting region to fit
  if (autozoom) {
    double maxmod = 0.0;
    for (i=0; i<NPTS; i++) {
      double modz = norm(V(i));
      if (modz > maxmod) maxmod = modz;
    }
    zoom = 1.0/sqrt(maxmod);
  }


  for (i=0; i<NPTS; i++) {
    double re = real(V(i));
    double im = imag(V(i));
    if (fabs(zoom*re) <= 1 && fabs(zoom*im) <=1) {
      setPixel((width+zoom*re*width)/2, (height-zoom*im*height)/2, 0xffffff);
    }
  }

  XPutImage(dsp, win, gc, ximage, 0, 0, 0, 0, width, height);
}

// creates random anti-hermitian matrix L and M = exp(lambda.L) - I
void create_matrix(int rnd_seed, cx_mat& L, cx_mat& M)
{
  srand(rnd_seed);

  L.zeros(NPTS,NPTS);

  // set all off-diagonal elements of L to random complex values,
  int i,j;
  for(i=0; i<NPTS; i++) {
    for (j=i+1; j<NPTS; j++) {
      double r1 = 1.0 - 2.0*(double)rand()/(double)RAND_MAX;
      double r2 = 1.0 - 2.0*(double)rand()/(double)RAND_MAX;
      complex<double> z1 (r1, r2);
      complex<double> z2 (-r1, r2);
      L(i,j) = z1;
      L(j,i) = z2;
    }
  }

   L = lambda*L;

   // construct an exp approx M = exp(L) - I ~= L + L^2/2 + L^3/6 + L^4/24 + L^5/120 + ... + L^13/6227020800
```

```
   cx_mat L2 = L*L;
   cx_mat L3 = L*L2;
   cx_mat L6 = L3*L3;
   M =  L + 0.5*L2 + (1.0/6.0)*L3 + ((1.0/24.0)*L2 + (1.0/120.0)*L3)*L2 + (1.0/720.0)*L6
         + ( (1.0/5040.0)*L + (1.0/40320.0)*L2 + (1.0/362880.0)*L3 + (1.0/3628800.0)*L*L3 )*L6
         + ( (1.0/39916800.0)*L2*L3 + (1.0/479001600.0)*L6 + (1.0/6227020800.0)*L*L6 )*L6;

   return;
}

void autolambda(cx_mat& M, cx_mat& L, cx_vec& U)
{
    int i;

    autozoom = 1;
    unitvector = 1;
    initvec(U);
    double delta = 0.01;
    double oldzoom;
    int ratechange = 0;
    int approx = 1;

    cout << "Searching for stable value of lambda for this matrix..." << endl;

    while (true)
    {
      cx_mat M3 = M*M*M;

      // execute a few initial cycles
      oldzoom = zoom;
      for (i=0; i<3; i++)  { U = M3*U; }
      update_image(U);

      if (zoom < 10E-6) { lambda -= delta*1.01; }
      else if (zoom > 10000) { lambda += delta*1.01; }
      else {
        for (i=0; i<200; i++) { U = M3*U;
        update_image(U); }

        oldzoom = zoom;
        for (i=0; i<3; i++) U = M3*U;
        update_image(U);

        if (fabs((zoom-oldzoom)/zoom) < 0.1 && approx == 1) {
          cout << "first approx, lambda = " << lambda << endl;
          approx=2;
        }
        if (fabs((zoom-oldzoom)/zoom) < 0.01 && approx == 2) {
          cout << "second approx, lambda = " << lambda << endl;
          approx=3;
        }
        if (fabs((zoom-oldzoom)/zoom) < 0.001) break;

        // change lambda by a constant amount until we overshoot or undershoot then halve the amount
        if (zoom-oldzoom > 0) { lambda += delta; if (ratechange == 0) delta *= 0.5; ratechange = 1; }
        else { lambda -= delta; if (ratechange == 1) delta *= 0.5; ratechange = 0; }
      }

      if (lambda <= 0.0) { lambda = 2.0*delta; delta *= 0.49; }
      if (lambda >= 1.0) { lambda = 1.0-2.0*delta; delta *= 0.51; }

      create_matrix(rseed, L, M);
      initvec(U);
    }

    cout << "found stable lambda = " << lambda << endl;

    return;
}
```