Call this procedure to add a point to the memory object. Any or all of the pointers may be NULL if you do not wish to copy information in to that field. This procedure will store the variables, parameters, and colors in the correct place according to the memory type. If there is only information for the header then a new point will not be added. Note that this routine assumes that all arrays have the correct dimensions. If the point is successfully stored in the memory object then it returns 0, otherwise it returns 1.

5.1.2.4 memory_add_points()

```
int
  memory_add_points(m, n, p_varb_seg, p_param_seg, p_color_seg,
       p_extra_doubles_seg, p_extra_ints_seg);
memory m;
int n, **p_color_seg, **p_extra_ints_seg;
double **p_varb_seg, **p_param_seg, **p_extra_doubles_seg
```

This procedure adds a whole array of points into the memory object Any of the pointers to segments may be NULL if these fields are not to be written into or are nonexistent. This procedure assumes that the arrays have the correct dimension. If this routine successfully stores all the requested points, then it returns 0, otherwise it returns 1.

5.1.2.5 memory_vanilla_start_new_flow()

```
int
  memory_vanilla_start_new_flow(m, n_trajs, n_dvars, n_ivars, n_points,
                              n_dparams,n_iparams)
memory m;
int n_trajs, n_dvars, n_ivars, n_points, n_dparams, n_iparams;
```

This routine is called to allocate space for a new flow inside a memory object. It will allocate space for `n_trajs` of expected length `n_points`. Each point will store n_dvars doubles and `n_ivars` integers. The header of the flow object can contain `n_dparams` doubles and n_iparams integers. If this routine fails, it returns a non-zero integer.

5.1.2.6 memory_vanilla_add_point()

```
int
  memory_vanilla_add_point(m, p_doubles, p_ints, p_dparams, p_iparams)
memory m;
double *p_doubles, *p_dparams;
int *p_ints, *p_iparams;
```

Call this procedure to add a point to the memory object. Any or all of the pointers may be NULL, in which case that data is not copied into the memory object. All arrays must be of the correct size. If successful at storing the point, the routine returns 0, otherwise it returns 1.

5.1.2.7 memory_next_traj()

```
int
  memory_next_traj(m);
memory m;
```

This procedure instructs the memory object that the next points should be added to the next trajectory. If the current trajectory is the last trajectory in the current flow, then the current trajectory is reset to be the first trajectory in the current flow.

5.1.2.8 memory_set_cur_traj()

```
int
  memory_set_traj(m,i);
memory m;
int i;
```

This procedure sets the current trajectory in the current flow. If `i` is a valid trajectory then it returns 0, otherwise it returns 1.

5.1.3 Reading Memory Objects

5.1.3.1 memory_get_first_point()

```
int
  memory_get_first_point(m, pp_varb, pp_param, pp_color, pp_extra_doubles,
                         pp_extra_ints, pp_extra_dparams, pp_extra_iparams);
memory m;
double **pp_varb, **pp_param, **pp_extra_doubles, **pp_extra_dparams;
int **pp_color, **pp_extra_ints, **pp_extra_iparams;
```

This procedure returns pointers to the elements of the first point in memory. If any of the arguments are NULL, then this info is not returned. This return returns 0 unless there are no points in the memory object, in which case, it returns 1.

5.1.3.2 memory_get_last_point()

```
int
  memory_get_last_point(m, pp_varb, pp_param, pp_color,
                        pp_extra_doubles, pp_extra_ints,
                        pp_extra_dparams, pp_extra_iparams);
memory m;
double **pp_varb, **pp_param, **pp_extra_doubles, **pp_extra_dparams;
int **pp_color, **pp_extra_ints, **pp_extra_iparams;
```

This procedure returns pointers to the elements of the last point in memory. If any of the arguments are NULL, then this info is not returned. This return returns 0 unless there are no points in the memory object, in which case, it returns 1.

5.1.3.3 memory_reset_read()

```
int
  memory_reset_read(m);
memory m;
```

This call is made to initialize the memory for being read. This routine returns 0, unless there are no flows, in which case, it returns 1.

5.1.3.4 memory_read_next_flow()

```
int
  memory_read_next_flow(m, pn_trajs, p_extra_dparams, pn_extra_dparams,
        p_extra_iparams, pn_extra_iparams)
memory m;
double **p_extra_dparams;
int *pn_extra_dparams,**p_extra_iparams, *pn_extra_iparams, *pn_trajs;
```

This procedure advances the reading of memory to the next flow in the memory object. It supplies the number of trajectories, the extra double parameters and how many, and the extra integer parameters and how many. If the pointers passed in are NULL, then this info is not returned. This routine returns 0 unless there are no more flows, in which case, it returns 1.

5.1.3.5 memory_read_next_traj()

```
int
  memory_read_next_traj(m, pn_points, pn_extra_doubles, pn_extra_ints)
memory m;
int *pn_points, *pn_extra_doubles, *pn_extra_ints;
```

This procedure advances the reading of memory to the next trajectory in the flow object being read. It supplies the number of points in the trajectory plus the dimension of the arrays of extra doubles and integers. This routine returns 0 unless there are no more trajectories in the flow being read, in which case, it returns 1.

5.1.3.6 memory_read_next_point()

```
int
  memory_read_next_point(m, pp_varb, pp_param, pp_color,
                         pp_extra_doubles, pp_extra_ints)
memory m;
double **pp_varb, **pp_param, **pp_extra_doubles;
int **pp_color, **pp_extra_ints;
```

This procedure supplies the next point in the current trajectory being read from the memory object. If any of the pointers are NULL then that information is not supplied. This routine returns 0 unless there are no more points in the trajectory being read, in which case, it returns 1.

Here is an example of how to examine all the points in a memory object:

```
if ( memory_reset_read(m) == 0 )
  while ( memory_read_next_flow(m, &n_trajs, &dparams, &n_dparams,
                                   &iparams, &n_iparams) == 0)
    while ( memory_read_next_traj(m, &n_points,
                                  &n_extra_doubles, &n_extra_ints) == 0)
    while ( memory_read_next_point(m, &varb, &param, &color,
                                   &extra_doubles, &extra_ints) == 0)
      {

        /* DO SOMETHING WITH ALL THIS POINT DATA ! */

      }
```

5.1.3.7 memory_vanilla_read_next_flow()

```
int
  memory_vanilla_read_next_flow(m, pn_trajs, p_dparams, pn_dparams,
        p_iparams, pn_iparams)
memory m;
double **p_dparams;
int *pn_dparams,**p_iparams, *pn_iparams, *pn_trajs;
```

This procedure advances the reading of memory to the next flow in the memory object. It supplies the number of trajectories, the double parameters and how many, and the integer parameters and how many. If the pointers passed in are NULL, then this info is not returned. This routine returns 0 unless there are no more flows, in which case, it returns 1.

5.1.3.8 memory_vanilla_read_next_traj()

```
int
  memory_vanilla_read_next_traj(m, pn_points, pn_doubles, pn_ints)
memory m;
int *pn_points, *pn_doubles, *pn_ints;
```

This procedure advances the reading of memory to the next trajectory in the flow object being read. It supplies the number of points in the trajectory plus the dimension of the arrays of doubles and integers. This routine returns 0 unless there are no more trajectories in the flow being read, in which case, it returns 1.

5.1.3.9 memory_vanilla_read_next_point()

```
int
  memory_vanilla_read_next_point(m, pp_doubles, pp_ints)
memory m;
double **pp_doubles;
int **pp_ints;
```

This procedure supplies the next point in the current trajectory being read from the memory object. If any of the pointers are NULL then that information is not supplied. This routine returns 0 unless there are no more points in the trajectory being read, in which case, it returns 1.

Here is an example of how to read all the points in a memory object:

```
if ( memory_reset_read(m) == 0 )
  while ( memory_vanilla_read_next_flow(m, &n_trajs, &dparams, &n_dparams,
                                        &iparams, &n_iparams) == 0)
    while ( memory_vanilla_read_next_traj(m, &n_points,
                                 &n_doubles, &n_ints) == 0)
    while ( memory_vanilla_read_next_point(m, &p_doubles, &p_ints) == 0)
      {

        /* DO SOMETHING WITH ALL THIS POINT DATA ! */

      }
```

5.1.4 Other Memory Object Procedures

5.1.4.1 memory_get_type()

```
int
  memory_get_type(m)
memory m;
```

This routine returns the type of a memory object. See the first section of this chapter for more information on memory types.

5.1.4.2 memory_exists_current_flow()

```
int
  memory_exists_current_flow(m)
memory m;
```

This routine returns 1 if there is a current flow, and 0 if there is not a current flow.

5.1.4.3 memory_closest_distance()

```
double
  memory_closest_distance(m,p,distance_f,work,manifold,flow_test)
memory m;
double *p, (*distance_f)(), *work;
Manifold *manifold;
int (*flow_test)();
```

This procedure searches through all the points in all the flows which satisfy the flow_test() function and returns the distance to the closest point as determined by the distance_f() function. The flow_test() function is passed the extra integer parameters for a flow plus the dimension of this array and should return TRUE of FALSE. If it returns TRUE, then the distance between the point `p` and each point in that flow is computed. If there are no points which pass the flow test, then a negative value is returned. The distance function is passed the work vector and manifold object. The work vector should be used so that the distance function does not need to do any vector memory allocations. The number of doubles alocated to this workspace will depend on the number of doubles needed by the distance function. The dimension of the point p should be the same as the dimension of the vector of doubles in the body of the memory object. section 5.1.4.6 for the a sample flow_test() procedure. Here is a sample distance function:

```
/*
 * return the L2 norm squared of the
 * n-vector (x1-x2)
 */
double
  dist_sqr(n, x1, x2, work, manifold);
int n;
double *x1, *x2, *work;
Manifold *manifold;
{
  double dist = 0.0;
  int i;

  for (i=0; i<n; i++)
    dist += (x1[i]-x2[i])*(x1[i]-x2[i]);
  return(dist);
}
```

5.1.4.4 memory_stored_points()

```
int
  memory_stored_points(m);
memory m;
```

This routine returns the number of points stored in the memory object

5.1.4.5 memory_clear_flows()

```
int
  memory_clear_flows(m)
memory m;
```

This routine takes a memory object and clears out all the flows. Under normal circumstances, the routine will return 0. In the case of error, a non-zero integer is returned.

5.1.4.6 memory_clear_selected_flows()

```
int
  memory_clear_selected_flows(m,flow_test);
memory m;
int (*flow_test)();
```

This routine takes a memory object and clears some of the flows. The flows cleared are the ones for which flow_test() returns TRUE. The routine flow_test() must return a decision based on the values of the extra integer parameters in the header of a flow object. For example:

```
int
  flow_test(p, n)
double *p;              /* the extra integer parameters  */
int n;                  /* the number of extra integer parameters */
{
  if ( ... )  return(TRUE);
  return(FALSE);
}
```

An example of the use of this routine is in the fixed point routines. There it is used to selectively remove either manifolds or fixed points.

5.1.4.7 memory_npoints()

```
int
  memory_npoints(m,nflow,ntraj)
  memory m;
  int nflow,ntraj;
```

This routine returns the number of points in a given trajectory of a given flow in the memory object m. Counting of flows and trajectories starts at 1.

5.1.4.8 memory_ntrajs()

```
int
  memory_ntrajs(m,nflow)
  memory m;
  int nflow;
```

This routine returns the number of trajectories in a given flow in the memory object.

5.1.4.9 memory_nflows()

```
int
  memory_nflows(m)
  memory m;
```

This routine returns the number of flows in the memory object.

5.1.4.10 memory_delete_last_flow()

```
int
  memory_delete_last_flow(m)
  memory m;
```

This procedure deletes the last flow from the memory object. If the last flow is the current flow then it will also do a memory_end_current_flow and if the last flow is the current flow for reading then memory_reset_read is called.

5.1.4.11 memory_delete_flow()

```
int
  memory_delete_flow(m, n)
  memory m;
  int n;
```

This routine deletes the n'th flow from the memory object. If the selected flow is the current flow then it will also do a memory_end_current_flow and if the selected flow is the current flow for reading then memory_reset_read is called. The range of the flow numbers is 1¡= flow_num ¡= memory_nflows().

5.1.4.12 memory_set_read()

```
int
  memory_set_read(m,nflow,ntraj,npoint,
                   p_extra_dparams, pn_extra_dparams,
    p_extra_iparams, pn_extra_iparams,
                   pn_extra_doubles, pn_extra_ints);
  memory m;
  int nflow,ntraj,npoint,*pn_extra_dparams,**p_extra_iparams,*pn_extra_iparams,
      *pn_extra_doubles,*pn_extra_ints;
  double **p_extra_dparams;
```

This call is made to set the memory so that the next point to be read is point npoint of trajectory ntraj of flow nflow in memory object m. This returns 0 is successful and 1 if there are no flows to read. If unsuccessful, then the current read pointers are left unchanged. Ranges for n* are 1 ¡= n* ¡= memory_n*s() (where * is flow,traj or point). The remaining variables in the function call are used as in the memory_read_next_flow and memory_read_next_traj routines.

5.1.4.13 memory_dump()

```
int
  memory_dump(fp,m,dump_level)
FILE *fp;
memory m;
int dump_level;
```

This routine writes out the memory structure to the file fp. This routine is used for testing purposes in order to verify the contents of the memory object. The detail of the dump (0 top level , ... , 3 every point) is specified by `dump_level`. A top level dump just summarizes the structure of the current memory object. The most detailed dump (level 3) will print out the location of every stored point and the full contents of the memory object. Dumps of levels 1 or 2 output an intermediate quantity of information.

Bibliography

[1] F. S. Acton. *Numerical Methods that Work.* Harper and Row, New York, 1970.

[2] Dan Heller. *XView Programming Manual*, volume 7 of *The X Window System Series.* O'Reilly & Associates, Inc., Sebastopol, CA, 1989.

[3] D. E. Knuth. *Seminumerical Algorithms.* Addison-Wesley Publishing Company, New York, 1981.

[4] J. M. Ortega and W. Rheinboldt. *Iterative Solutions of Nonlinear Equations in Several Variables.* Academic Press, New York, 1970.

[5] W. H. Press et al. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge UP, Cambridge, 1988.

[6] Sun Microsystems, Inc. *OpenWindows Developer's GUIDE, User's Manual*, September 1989. Beta Version.

Index