2024 European Control Conference (ECC)
June 25-28, 2024. Stockholm, Sweden

Efficient C++ implementations of generalized interpolation in Reproducing Kernel Hilbert Spaces to compute Lyapunov functions

Sigurdur Freyr Hafstein*,[1]

Abstract— **The Radial Basis Function (RBF) method to compute Lyapunov functions for nonlinear systems uses generalized interpolation in Reproducing Kernel Hilbert spaces. We present two different implementation in C++. One that is computationally efficient and one that is memory efficient. The former uses standard functions of a numerical library and the latter directly calls LAPACK routines for packed matrices to perform in-place Cholesky factorization of the interpolation matrix. The memory efficient implementation only needs one-fourth of the memory needed when using a standard numerical library and thus makes it possible to use double the amount of collocation points. Both implementations are easily adapted to different generalized interpolation problems.**

I. Motivation

As a motivation we consider a general ordinary differential equation (ODE) of the form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \text{ where } \mathbf{f}\colon \mathbb{R}^n \to \mathbb{R}^n \text{ and } \mathbf{f}(0) = 0, \tag{1}$$

and study the stability of its equilibrium at the origin.

If $\mathbf{f}$ in (1) is locally Lipschitz, the ODE has a unique solution for every initial-value $\xi \in \mathbb{R}^n$ at time $t = 0$ and we denote this solution $t \mapsto \phi(t,\xi)$. We assume that $\mathbf{f} \in C^m(\mathbb{R}^n;\mathbb{R}^n)$, $m \in \mathbb{N} := \{1,2,\dots\}$, let $p \in C^m(\mathbb{R}^n;\mathbb{R})$ be a positive definite function, i.e. $p(0) = 0$ and $p(\mathbf{x}) > 0$ if $\mathbf{x} \neq 0$, and let $q \in C^m(\mathbb{R}^n;\mathbb{R})$ be a positive function such that $\sup_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{f}(\mathbf{x})\|_2/q(\mathbf{x}) < \infty$. Then by [12, Th. 2.8] the origin is exponentially stable, i.e. there exist a neighbourhood $N \subset \mathbb{R}^n$ of the origin and constants $\alpha > 0$, $C \geq 1$, such that

$$\|\phi(t,\xi)\|_2 \leq C\|\xi\|_2 \exp(-\alpha t)$$

for all $\xi \in N$ and all $t \geq 0$, if and only if the partial differential equation (PDE)

$$\langle \nabla V(\mathbf{x}), \mathbf{f}(\mathbf{x})\rangle_2 := \nabla V(\mathbf{x}) \cdot \mathbf{f}(\mathbf{x}) = -p(\mathbf{x})q(\mathbf{x}), \tag{2}$$

has a positive definite solution $V\colon M \to \mathbb{R}$, where $M \subset \mathbb{R}^n$ is a neighbourhood of the origin. In particular, one can take M as the origins basin of attraction

$$A(0) := \{\mathbf{x} \in \mathbb{R}^n\colon \lim_{t\to\infty} \phi(t,\mathbf{x}) = 0\}.$$

The function V is a Lyapunov function for the system (1) and since

$$V'(\mathbf{x}) := \frac{d}{dt}\phi(t,\mathbf{x})\bigg|_{t=0} = \langle \nabla V(\mathbf{x}), \mathbf{f}(\mathbf{x})\rangle_2 = -p(\mathbf{x})q(\mathbf{x}) < 0$$

for all $\mathbf{x} \in M \setminus \{0\}$, the PDE forces V to be strictly decreasing along all solution trajectories of system (1) in $\mathbf{x} \in M \setminus \{0\}$.

The Lyapunov stability theory is of fundamental significance in the theory of dynamical systems and control theory and has various applications in both theory and applications, see e.g. [23], [19], [8], [20], [27], [29], [22], [4]. Since Lyapunov functions cannot be obtained analytically, except in special cases, many numerical methods for their computations have been devised; see e.g. [28], [25], [24], [7], [21] and the review [13]. In the so-called RBF method, where RBF refers to Wendland's Radial Basis Functions [30], generalized interpolation in a reproducing kernel Hilbert space (RKHS) is used so solve the PDE (2) numerically. Let us present the most relevant steps and equations here.

First one selects a Wendland function $\psi = \psi_{\ell,k}\colon [0,\infty) \to [0,\infty)$ of appropriate order $\ell, k \in \mathbb{N}$; for our problem that means $k \geq 2$ if n is odd and $k \geq 3$ if n is even and that $\ell := \lfloor \frac{n}{2} \rfloor + k + 1$. We will use the system

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} y \\ -x + \frac{1}{3}x^3 - y \end{bmatrix} =: \mathbf{f}(\mathbf{x}) \tag{3}$$

as an example for our computations so $n = 2$ and with $k = 3$ we have $\ell = 5$. We choose a constant $c > 0$ and set

$$\psi(r) = \psi_{5,3}(cr) = (1-cr)_+^8[32(cr)^3 + 25(cr)^2 + 8cr + 1],$$

where $(1-cr)_+^m := (1-cr)^m$ if $r \in [0,1/c]$ and $(1-cr)_+^m = 0$ otherwise, $m \in \mathbb{N}$. The parameter $c > 0$ fixes the support $[0,1/c]$ of the Wendland function ψ. In practice it usually works best to take the order of the Wendland function as low as possible, because then the so-called collocation matrix is better conditioned [2].

Second, one selects a set of collocation points $X := \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ to compute a numerical solution to (2). With the auxiliary functions

$$\psi_1(r) = \frac{\frac{d}{dr}\psi(r)}{r} \quad \text{and} \quad \psi_2(r) = \frac{\frac{d}{dr}\psi_1(r)}{r} \quad \text{for } r > 0, \tag{4}$$

which in our example are

$$\psi_1(r) = -22c^2(1-cr)_+^7[16(cr)^2 + 7cr + 1],$$
$$\psi_2(r) = 528c^4(1-cr)_+^6[6cr + 1],$$

and using the ansatz

$$V_R(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \langle \mathbf{x}_j - \mathbf{x}, \mathbf{f}(\mathbf{x}_j)\rangle_2 \psi_1(\|\mathbf{x} - \mathbf{x}_j\|_2), \tag{5}$$

one fixes $\alpha := (\alpha_1, \alpha_2, \dots, \alpha_N)^T$ as the solution to $A\alpha = \beta$, where $\beta := (\beta_1, \beta_2, \dots, \beta_N)^T$,

$$\beta_i = -p(\mathbf{x}_i)q(\mathbf{x}_i), \ i = 1,2,\dots,N, \tag{6}$$

*This research was partially supported by the Icelandic Research Fund in grant number 228725-051.

[1]Sigurdur Freyr Hafstein is with the Science Institute, University of Iceland, Dunhagi 5, 107 Reykjavik, Iceland shafstein@hi.is

and the so-called collocation matrix $A = (a_{ij}) \in \mathbb{R}^{N\times N}$ has the entries

$$a_{ij} = \psi_2(\|\mathbf{x}_i - \mathbf{x}_j\|_2)\langle \mathbf{x}_i - \mathbf{x}_j, \mathbf{f}(\mathbf{x}_i)\rangle_2 \langle \mathbf{x}_j - \mathbf{x}_i, \mathbf{f}(\mathbf{x}_j)\rangle_2 - \psi_1(\|\mathbf{x}_i - \mathbf{x}_j\|_2)\langle \mathbf{f}(\mathbf{x}_i), \mathbf{f}(\mathbf{x}_j)\rangle_2, \quad (7)$$

$i, j = 1, 2, \ldots, N$. Such a solution exist if X does not contain an equilibrium point for (1), i.e. $\mathbf{f}(\mathbf{x}_i) \neq 0$ for all $i = 1, 2, \ldots, N$. Indeed, then one can show that the symmetric matrix A is positive definite and hence non-singular.

Further, as shown in [12, Th. 3.9], for a given $\delta > 0$ and a compact neighbourhood $M \subset \mathbb{R}^n$ of the origin, $M \subset A(0)$, one can select the collocation points X such that

$$\max_{\mathbf{x}\in M} |V_R(\mathbf{x}) - V(\mathbf{x})| < \delta \quad \text{and}$$
$$\max_{\mathbf{x}\in M} |V_R'(\mathbf{x}) - V'(\mathbf{x})| < \delta,$$

where V_R is the numerical solution (5) and V is the true solution to (2). To achieve this one must select the collocation points such that the so-called fill distance

$$h_{X,M} := \sup_{\mathbf{x}\in M} \min_{\mathbf{x}_j \in X} \|\mathbf{x} - \mathbf{x}_j\|_2 \quad (8)$$

is small enough.

The implementations we will present in the next section can easily be adapted to other applications, e.g. generalized interpolation for the computation of Lyapunov functions for discrete-time systems, stochastic differential equations, or contraction metrics; or something completely different. We will not discuss the generalized interpolation problem in RKHS further here, but point the interested reader to [3], [31] for the general theory and [9], [18], [6], [10], [11], [14], [15] for some specific applications.

The essential property for the implementation is that one uses a symmetric and positive definite collocation matrix of dimension $N \times N$, where N is the number of the collocation points, and more collocation points lead to better approximations. Hence, one wants N to be large and thus the size of the matrix $A \in \mathbb{R}^{N\times N}$ can quickly become a limiting factor. One way to deal with large $A \in \mathbb{R}^{N\times N}$ is to use sparse matrix data structures, if possible, but this approach brings its own problems like how to organize the collocation points in order to have sparse collocation matrices, which sparse solver can be used, and what kind of preconditioning is needed. Another approach, and that is the one we will present, is to fit the matrix A into as little memory as possible without assuming sparsity and use in-place computations, i.e. numerical methods that do not need extra memory. We do this by calling LAPACK [1] functions for packed matrices directly.

II. The Implementation

The advantage of using C++ instead of other common programming languages for numerical computations, such as Matlab or Python, is that C++ code tends to be much faster and that C++, as a further development of C, gives the programmer low-level control over the computer. Apart from the standard library we use the Armadillo library for linear algebra and scientific computing [26] in our implementation, which allows Matlab like syntax for many computations. We will present a memory saving implementation using a packed collocation matrix and direct calls to LAPACK routines, but we will also show a more straight-forward implementation of the RBF method without using direct calls to LAPACK routines. The latter implementation is faster, but it needs considerably more memory (factor four). Hence, by using the former implementation one can use double the amount of collocation points used on a given computer. All the code presented here can be downloaded from `https://github.com/shafstein/RBFLya`

We did our implementation on Linux Mint 21.2 using the GNU compiler `g++`. The compiler flags used were `-O2` for speed, `-larmadillo` to use Armadillo, `-lmkl_rt` to use intel's implementation MKL of LAPACK, and `-pthread` for multithreading. On the AMD platform we set `MKL_DEBUG_CPU_TYPE=5` to avoid AVX2 being disabled in MKL, see e.g. [16].

As default integer type we use `using bint=long long` (big integer) and for thread parallelization we use the standard threads library `std::thread`; see the following code.

```
using bint = long long;
using namespace std;
void ParallelFor(const bint _end,
        function<void(bint)> parfor,
              const bint NrThread = 1000) {
  for (bint i = 0; i < _end; i += NrThread) {
    vector<thread> threads(NrThread);
    for (bint j = i; j < i + NrThread
                     && j < _end; j++) {
      threads[j%NrThread]=thread(parfor,j);
    }
    for (bint j = i; j < i + NrThread
                     && j < _end; j++) {
      threads[j%NrThread].join();
    }
  }
}

```

We implement the RBF method in `class RBFLya` defined below.

```
 using namespace arma;  // vec, mat
 class RBFLya {
   bint N;
   double c;
   function<vec(vec)> f;
   function<double(vec)> pq;
   function<double(double,double)> psi1, psi2;
   vec alpha, beta;
   vector<vec> X, fX;
   vec A;
   mat Am;
  public:
   RBFLya(function<vec(vec)> _f,
     function<double(vec)> _pq,
     function<double(double, double)> _psi1,
     function<double(double, double)> _psi2,
     double _c) :  f(_f), pq(_pq), psi1(_psi1),
                   psi2(_psi2), c(_c), N(0) {
   };
   void FixVertices(const vector<vec> &_X);
   void WriteA(void);
```

```
22     void WriteAm(void);
23     void SolveRBF(void);
24     void SolveRBFm(void);
25     double V(const vec &x) const;
26     double OrbDerV(const vec &x) const;
27   };
28
```

The data members are as follows: `N` is the number of collocation points, `c` is the support radius parameter for the Wendland function, see the definition of ψ below equation (3), `f` is the right-hand side of (1), `pq` is the function $\mathbf{x} \mapsto p(\mathbf{x})q(\mathbf{x})$ on the right-hand side of the PDE (2), `psi1` and `psi2` are the auxiliary function functions ψ_1 and ψ_2 from (4) that are used in (7) and (5), `X` contains the collocation points, and `fX` contains the values $\mathbf{f}(\mathbf{x}_j)$ for the collocation points $\mathbf{x}_j \in X$ to speed up computations. The vectors `alpha` and `beta` are the vectors $\alpha, \beta \in \mathbb{R}^N$ in $A\alpha = \beta$. The vector `A` is the packed form of the matrix A and the matrix `Am` is the unpacked form of the matrix. The constructor of the class simply initiates some of the data members. A listing of the member functions together with their implementation follows.

The member function `FixVertices` is used to read the collocation points X into `X` and compute `fX`, i.e. $\mathbf{f}(\mathbf{x}_j)$ for all $\mathbf{x}_j \in X$. For brevity, we generate a regular grid of collocation points in the example program below and read them using `FixVertices`. A better choice is to generate an optimal grid, in the sense of lowest fill distance for a given number of collocation points. An efficient implementation of such a grid is shown in Listing 1.1 in [5]. We also assert that no point in X is an equilibrium point, a common mistake that results in the collocation matrix being singular, and fix `N` as the number of collocation points.

```
1  void RBFLya::FixVertices(const vector<vec>
2                                          &_X) {
3    X = _X;
4    N = X.size();
5    fX.resize(N);
6    for (bint i = 0; i < N; i++){
7      fX[i] = f(X[i]);
8      assert(norm(fX[i]) > 1e-10);
9    }
10 }
```

The member function `WriteA` is used to write the symmetric collocation matrix A into the vector `A` in a packed form and the vector `beta`, which represent the right-hand side of the equation $A\alpha = \beta$. The code should be self-explanatory, as it is essentially the coding of formulas (6) and (7), although the numbering in `A` might be a little confusing; we are writing the elements in the upper triangle of the symmetric matrix A column-by-column consecutively into the vector `A`. To speed up the computations we use thread parallelization with `ParallelFor`.

```
1  void RBFLya::WriteA(void) {
2    A.set_size(N * (N + 1) / 2);
3    beta.set_size(N);
4    ParallelFor(N, [&](bint k) {
5      bint offset = k * (k + 1) / 2;
6      beta(k) = -pq(X[k]);
7      for (bint j = 0; j <= k; j++) {
8        vec x_j_m_x_k = X[j] - X[k];
9        double dist = norm(x_j_m_x_k, 2);
10       if (1.0 - c * dist > 0.0) {
11         A[offset + j] = -psi2(dist, c)
12                         * dot(x_j_m_x_k, fX[j])
13                         * dot(x_j_m_x_k, fX[k])
14           - psi1(dist, c) * dot(fX[j], fX[k]);
15       }
16       else {
17         A[offset + j] = 0.0;
18       }
19     }
20   });
21 }
```

The member function `WriteAm` writes the the vector `beta` and the symmetric collocation matrix A into the matrix `Am` in unpacked form.

```
1  void RBFLya::WriteAm(void) {
2    Am.set_size(N,N);
3    beta.set_size(N);
4    ParallelFor(N, [&](bint k) {
5      beta(k) = -pq(X[k]);
6      for (bint j = 0; j < N; j++) {
7        vec x_j_m_x_k = X[j] - X[k];
8        double dist = norm(x_j_m_x_k, 2);
9        if (1.0 - c * dist > 0.0) {
10         Am(j,k) = -psi2(dist, c)
11                   * dot(x_j_m_x_k, fX[j])
12                   * dot(x_j_m_x_k, fX[k])
13           - psi1(dist, c) * dot(fX[j], fX[k]);
14       }
15       else {
16         Am(j,k)=0.0;
17       }
18     }
19   });
20 }
```

We now come to the interesting part, the member function `SolveRBF`. It calls the LAPACK function `dpptrf_` to Cholesky factorize the matrix A, stored in the vector `A` in packed form, and does this without requiring any additional memory, as it does this in-place and overwrites `A` with the Cholesky factor. Note that the Cholesky factor is stored in the vector `A` just like the matrix A, i.e. the upper triangular part of the Cholesky factor is written column-by-column sequentially in `A`; the lower triangular part that is not written consists only of zeros. Now, that `A` contains the Cholesky factor of A, we can solve the equation $A\alpha = \beta$ efficiently using the LAPACK function `dpptrs_`. Note that the parameter `INFO` in the code must be of type `int` (and not `bint`). Further note that we need to declare the LAPACK functions `dpptrf_` and `dpptrs_` as `extern "C"` to get the correct binding.

```
1  extern "C" {
2    void dpptrf_(char *UPLO, bint *N,
3      double *A, int *INFO);
4    void dpptrs_(char *UPLO, bint *N,
5      bint *NRHS, double *AP, double *B,
6      bint *LDA, int *INFO);
7  }
8
```

```
void RBFLya::SolveRBF(void) {
  char UPLO = 'U';
  int INFO;
  // Cholesky factorize in-place
  dpptrf_(&UPLO, &N, A.memptr(), &INFO);
  bint NRHS = 1;
  // solve A*alpha = beta (overwrites beta)
  alpha = beta;
  dpptrs_(&UPLO, &N, &NRHS, A.memptr(),
    alpha.memptr(), &N, &INFO);
}
```

The member function `SolveRBFm` assumes that the matrix A has been written into the matrix `Am` and uses standard Armadillo functions to Cholesky factorize it and solve the equation $A\alpha = \beta$. The Armadillo functions `trimatl` and `trimatu` inform the Armadillo `solve` function that the matrix is lower triangular and upper triangular, respectively.

```
void RBFLya::SolveRBFm(void) {
  Am=chol(Am);
  alpha = solve(trimatl(Am.t()),beta);
  alpha = solve(trimatu(Am),alpha);
}

```

After the solution α to $A\alpha = \beta$ has been computed, either with `SolveRBF` or `SolveRBFm`, we can use formula (5) to compute $V_R(\mathbf{x})$ at any $\mathbf{x} \in \mathbb{R}^n$. This is done in the member function `V`, which is implemented as follows.

```
double RBFLya::V(const vec &x) const {
  double ret = 0.0;
  for (bint k = 0; k < N; k++) {
    vec x_k_m_x = X[k] - x;
    double dist = norm(x_k_m_x, 2);
    if (1.0 - c * dist > 0) {
      ret += alpha(k) * dot(x_k_m_x, fX[k])
                      * psi1(dist, c);
    }
  }
  return ret;
}
```

We also implemented the member function `OrbDerV` to compute the orbital derivative V_R' at an arbitrary $\mathbf{x} \in \mathbb{R}^n$. The formula for $V_R'(\mathbf{x})$ can be shown to be, see [9, Prop. 3.5],

$$V_R'(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \Big[\psi_2(\|\mathbf{x}-\mathbf{x}_j\|_2)\langle \mathbf{x}-\mathbf{x}_j, \mathbf{f}(\mathbf{x})\rangle_2 \langle \mathbf{x}_j-\mathbf{x}, \mathbf{f}(\mathbf{x}_j)\rangle_2 - \psi_1(\|\mathbf{x}-\mathbf{x}_j\|_2)\langle \mathbf{f}(\mathbf{x}), \mathbf{f}(\mathbf{x}_j)\rangle_2 \Big] \qquad (9)$$

and this formula is implemented in the member function `OrbDerV` as follows.

```
double RBFLya::OrbDerV(const vec &x) const {
  double ret = 0.0;
  vec fx = f(x);
  for (bint k = 0; k < N; k++) {
    vec x_k_m_x = X[k] - x;
    double dist = norm(x_k_m_x, 2);
    if (1.0 - c * dist > 0) {
      ret += -alpha(k) * (psi1(dist, c)
           * dot(fx, fX[k])
           + psi2(dist, c) * dot(x_k_m_x, fx)
           * dot(x_k_m_x, fX[k]));
    }
  }
  return ret;
}
```

Now that the class `RBFLya` is fully implemented we give a short example program of how to use it to compute a Lyapunov function using the RBF method for system (3). We write the results into files using the Armadillo command `save`, which is defined for both vectors and matrices in such a way that Matlab can easily read them for plotting.

```
#include "RBFlya.h"
const unsigned int n = 2;
int main(int argc, char **argv) {
  // use packed collocation matrix
  bool packed = true;
  function<vec(const vec &)> f =
  [](const vec &x)->vec {
    vec fx(n);
    fx(0) = x(1);
    fx(1) = -x(0) - x(1)
          + 1.0 / 3.0 * pow(x(0), 3);
    return fx;
  };
  function<double(const vec &)> pq =
  [&f](const vec &x)->double {
    return pow(norm(x, 2), 2)
         * (1 + pow(norm(f(x), 2), 2));
  };
  function<double(double,double)> psi1 =
  [](double r, double c)->double {
    return 1.0 - c * r > 0 ?
      -22 * pow(c, 2) * pow(1 - c * r, 7)
        * (1 + 7 * c * r + 16 * pow(c * r, 2))
      : 0.0;
  };
  function<double(double,double)> psi2 =
  [](double r, double c)->double {
    return 1.0 - c * r > 0 ?
      528 * pow(c, 4) * pow(1 - c * r, 6)
          * (1 + 6 * c * r)
      : 0.0;
  };
  double c = 1.5;
  RBFLya R(f,pq,psi1,psi2,c);
  // xN, yN must be even numbers to avoid (0,0)
  bint xN = 200, yN = 250;
  double xMin = -2.6, yMin = -2.6;
  double xMax = 2.6, yMax = 2.6;
  vector<vec> X(xN * yN);
  for (bint i = 0; i < xN; i++) {
    for(bint j = 0; j < yN; j++) {
      X[j + yN*i] = vec
        {xMin+i*(xMax-xMin)/(xN-1),
         yMin+j*(yMax-yMin)/(yN-1)};
    }
  }
  R.FixVertices(X);
  if (packed == true) {
    R.WriteA();
    R.SolveRBF();
  }
  else {
    R.WriteAm();
    R.SolveRBFm();
  }
  // now write results
  vec xMatlab(xN), yMatlab(yN);
  for (bint i = 0; i < xN; i++){
    xMatlab(i) = X[i*yN](0);
```

```
60    }
61    for (bint j = 0; j < yN; j++){
62      yMatlab(j) = X[j](1);
63    }
64    mat VMatlab(yN,xN), OrbDerVMatlab(yN,xN);
65    ParallelFor(yN,[&](bint j)->void {
66      for (bint i = 0; i < xN; i++) {
67            VMatlab(j,i) = R.V(X[j+yN*i]);
68            OrbDerVMatlab(j,i) =
69                    R.OrbDerV(X[j+yN*i]);
70      }
71    });
72    xMatlab.save("x.txt", raw_ascii);
73    yMatlab.save("y.txt", raw_ascii);
74    VMatlab.save("V.txt", raw_ascii);
75    OrbDerVMatlab.save("OrbDerV.txt", raw_ascii);
76  }
```

The computed Lyapunov function can now easily be plotted in Matlab using the following commands:

```
load -ascii 'x.txt'
load -ascii 'y.txt'
load -ascii 'V.txt'
[X,Y]=meshgrid(x,y);
surf(X,Y,V)
xlabel('X')
ylabel('Y')
zlabel('V(X,Y)')
```

Its orbital derivative can similarly be plotted with the commands:

```
load -ascii 'x.txt'
load -ascii 'y.txt'
load -ascii 'OrbDerV.txt'
[X,Y]=meshgrid(x,y);
surf(X,Y,OrbDerV)
xlabel('X')
ylabel('Y')
zlabel("V'(X,Y)")
```

In Figure 1 we plot the Lyapunov function V_R computed with the code and its orbital derivative V_R'.

III. Computational results

Let us discuss some computation results with the code from the last section. We use the system (3) for our computations, but note that the the time and the memory needed for the computations only depends on the number of collocation points and not on the dimension of the differential equation. The memory needed is easily computed. For example, with $N = 50,000$ collocation points the memory needed to save an $N \times N$ matrix of doubles (8 Bytes) is (recall $\mathrm{G} = 1024^3$ in computer science)

$$\frac{8N^2}{1024^3}\,\mathrm{GB} = \frac{8 \cdot 50,000^2}{1024^3}\,\mathrm{GB} = 18.63\,\mathrm{GB}$$

and as the Cholesky factorization with the Armadillo command `A=chol(A)` is not done in-place, and therefore two such matrices are needed, we need $2 \cdot 18.63\,\mathrm{GB} = 37.26\,\mathrm{GB}$ to perform the computations. In comparison the packed version of A only needs

$$\frac{8N(N+1)}{2 \cdot 1024^3}\,\mathrm{GB} = \frac{8 \cdot 50,000 \cdot 50,001}{2 \cdot 1024^3}\,\mathrm{GB} = 9.31\,\mathrm{GB}$$

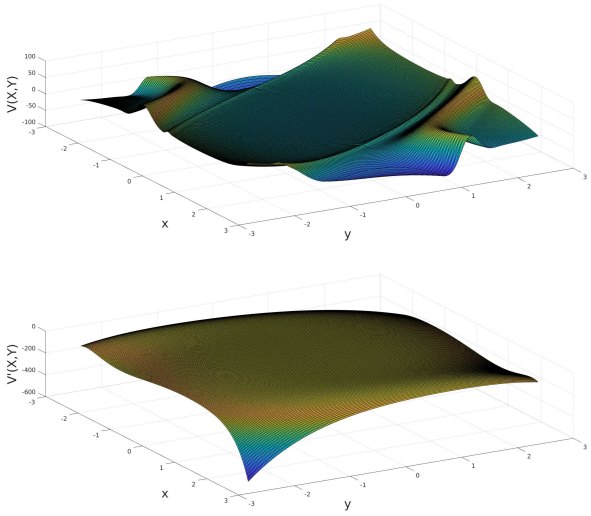

Fig. 1. The Lyapunov function (upper) and its orbital derivative (lower) computed for system (3) using the RBF method.

and as the Cholesky factorization is done in-place additional memory is not needed.

The reduction in the memory needed comes at the cost of some computational speed. In Table I we compare the speed of the implementations on an intel platform with a 13900K processor (24 cores, 128 GB) and an AMD platform with Threadripper 3990X (64 cores, 256 GB). The computational time increases by a factor of 2.4 to 4.1 when using the packed form in comparison with the unpacked form, unless memory is very tight for the computations (232.8 GB out of 256 GB available). Thus, if sufficient memory is available for the collocation points needed, it is considerably faster to use the unpacked form.

IV. Conclusions

We presented two implementations of the RBF method to compute Lyapunov functions for nonlinear systems. One implementation using straightforward calls to computational routines in the numerical library Armadillo and one using packed matrices to save memory and using direct calls to in-place Cholesky factorization in LAPACK. While the second implementation is slower it allows for the use of double the amount of collocation points. A modern personal computer can be equipped with 192 GB of RAM and on such a machine one can use ca. 210,000 collocation points when using the latter approach, instead of just ca. 105,000 when using a more straight forward implementation. It is the hope of the author that the implementations in this paper are useful to engineers and scientists, who want to use generalized approximation in Reproducing Kernel Hilbert Spaces, and enables them to solve more demanding problems faster. Although the code given is for the computation of Lyapunov functions for nonlinear systems using the RBF method, it is easily adapted to other applications in generalized approximation, e.g. the data driven approach in [17].

$N = 50{,}000$ (37.3 GB unpacked / 9.3 GB packed)

	intel 13900K - 128 GB	AMD 3990X - 256 GB
unpacked	35 seconds	37 seconds
packed	101 seconds	87 seconds

$N = 75{,}000$ (83.8 GB unpacked / 21.0 GB packed)

	intel 13900K - 128 GB	AMD 3990X - 256 GB
unpacked	79 seconds	98 seconds
packed	320 seconds	273 seconds

$N = 100{,}000$ (149 GB unpacked / 37.3 GB packed)

	intel 13900K - 128 GB	AMD 3990X - 256 GB
unpacked	out of memory	193 seconds
packed	745 seconds	605 seconds

$N = 125{,}000$ (232.8 GB unpacked / 58.2 GB packed)

	intel 13900K - 128 GB	AMD 3990X - 256 GB
unpacked	out of memory	1171 seconds
packed	1395 seconds	1138 seconds

$N = 250{,}000$ (931.3 GB unpacked / 232.8 GB packed)

	intel 13900K - 128 GB	AMD 3990X - 256 GB
unpacked	out of memory	out of memory
packed	out of memory	9797 seconds

TABLE I

COMPARISON OF THE MEMORY NEEDED AND THE COMPUTATIONAL TIMES FOR USING THE RBF METHOD TO COMPUTE A LYAPUNOV FUNCTION FOR SYSTEM (3). WE SHOW RESULTS FOR THE STRAIGHT FORWARD IMPLEMENTATION USING UNPACKED COLLOCATION MATRICES AND THE IMPLEMENTATION USING PACKED COLLOCATION MATRICES AND DIRECTLY CALLING LAPACK ROUTINES, BOTH ON AN INTEL PLATFORM AND AN AMD PLATFORM. THE REDUCTION IN MEMORY NEEDED (FACTOR FOUR) IS OFFSET BY A LONGER COMPUTATIONAL TIME (FACTOR 2.4 TO 4.1).

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 3. edition, 1999.

[2] C. Argáez, P. Giesl, and S. Hafstein. Comparison of different radial basis functions in dynamical systems. In *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 394–405, 2021.

[3] N. Aronszajn. Theory of reproducing kernels. *Trans. Am. Math. Soc.*, 68:337–404, 1950.

[4] P. Bernhard and S. Suhr. Lyapounov functions of closed cone fields: From Conley theory to time functions. *Commun. Math. Phys.*, 359:467–498, 2018.

[5] H. Bjornsson and S. Hafstein. *Informatics in Control, Automation and Robotics*, volume 793 of *Lecture Notes in Electrical Engineering*, chapter Advanced algorithm for interpolation with Wendland functions, pages 99–117. Springer, 2021.

[6] H. Bjornsson, S. Hafstein, P. Giesl, E. Scalas, and S. Gudmundsson. Computation of the stochastic basin of attraction by rigorous construction of a Lyapunov function. *Discrete Contin. Dyn. Syst Ser. B*, 24(8):4247–4269, 2019.

[7] G. Chesi. *Domain of Attraction: Analysis and Control via SOS Programming*. Lecture Notes in Control and Information Sciences, vol. 415, Springer, 2011.

[8] C. Conley. *Isolated Invariant Sets and the Morse Index*. CBMS Regional Conference Series no. 38. American Mathematical Society, 1978.

[9] P. Giesl. *Construction of Global Lyapunov Functions Using Radial Basis Functions*. Lecture Notes in Math. 1904, Springer, 2007.

[10] P. Giesl. Computation of a contraction metric for a periodic orbit using meshfree collocation. *SIAM J. Appl. Dyn. Syst.*, 18(3):1536–1564, 2019.

[11] P. Giesl, C. Argáez, S. Hafstein, and H. Wendland. Minimization with differential inequality constraints applied to complete Lyapunov functions. *Math. Comput.*, 90(331):2137–2160, 2021.

[12] P. Giesl and S. Hafstein. Computation and verification of Lyapunov functions. *SIAM J. Appl. Dyn. Syst.*, 14(4):1663–1698, 2015.

[13] P. Giesl and S. Hafstein. Review of computational methods for Lyapunov functions. *Discrete Contin. Dyn. Syst. Ser. B*, 20(8):2291–2331, 2015.

[14] P. Giesl, S. Hafstein, and I. Mehrabinezhad. Computation and verification of contraction metrics for exponentially stable equilibria. *J. Comput. Appl. Math.*, 390:Paper No. 113332, 2021.

[15] P. Giesl, S. Hafstein, and I. Mehrabinezhad. Computation and verification of contraction metrics for periodic orbits. *J. Math. Anal. Appl.*, 503(2):Paper No. 125309, 32, 2021.

[16] P. Giesl, S. Hafstein, and I. Mehrabinezhad. Computing contraction metrics: Comparison of different implementations. *IFAC PapersOnLine*, 54(9):310–316, 2021.

[17] P. Giesl, B. Hamzi, M. Rasumussen, and K. Webster. Approximation of Lyapunov functions from noisy data. *J. Comput. Dynamics*, 1(7):57–81, 2020.

[18] P. Giesl and H. Wendland. Construction of a contraction metric by meshless collocation. *Discrete Contin. Dyn. Syst. Ser. B*, 24(8):3843–3863, 2019.

[19] W. Hahn. *Stability of Motion*. Springer, Berlin, 1967.

[20] M. Hurley. Lyapunov functions and attractors in arbitrary metric spaces. *Proc. Amer. Math. Soc.*, 126:245–256, 1998.

[21] R. Kamyar and M. Peet. Polynomial optimization with applications to stability analysis and control – an alternative to sum of squares. *Discrete Contin. Dyn. Syst. Ser. B*, 20(8):2383–2417, 2015.

[22] H. Khalil. *Nonlinear Systems*. Pearson, 3. edition, 2002.

[23] A. M. Lyapunov. The general problem of the stability of motion. *Internat. J. Control*, 55(3):521–790, 1992. Translated by A. T. Fuller from Édouard Davaux's French translation (1907) of the 1892 Russian original, With an editorial (historical introduction) by Fuller, a biography of Lyapunov by V. I. Smirnov, and the bibliography of Lyapunov's works collected by J. F. Barrett, Lyapunov centenary issue.

[24] M. Peet and A. Papachristodoulou. A converse sum of squares Lyapunov result with a degree bound. *IEEE Transactions on Automatic Control*, 57(9):2281–2293, 2012.

[25] A. Polanski. On absolute stability analysis by polyhedral Lyapunov functions. *Automatica*, 36(4):573–578, 2000.

[26] C. Sanderson and R. Curtin. Armadillo: a template-based C++ library for linear algebra. *J. Open Source Softw*, 1(2):26, 2016.

[27] S. Sastry. *Nonlinear Systems: Analysis, Stability, and Control*. Springer, 1999.

[28] A. Vannelli and M. Vidyasagar. Maximal Lyapunov functions and domains of attraction for autonomous nonlinear systems. *Automatica*, 21(1):69–80, 1985.

[29] M. Vidyasagar. *Nonlinear System Analysis*. Classics in Applied Mathematics. SIAM, 2. edition, 2002.

[30] H. Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Adv. Comput. Math.*, 4(4):389–396, 1995.

[31] H. Wendland. *Scattered data approximation*, volume 17 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2005.